from flask import Flask, render\_template, request, redirect, url\_for, flash, send\_file, jsonify, abort

from flask\_sqlalchemy import SQLAlchemy

from flask\_migrate import Migrate

from sqlalchemy.exc import IntegrityError

from datetime import datetime

from io import BytesIO

from fpdf import FPDF

from sqlalchemy import or\_

from datetime import datetime

from apscheduler.schedulers.background import BackgroundScheduler

from sqlalchemy import func

from flask\_apscheduler import APScheduler

from flask\_mail import Message

from wtforms.validators import DataRequired, Length, Optional

import calendar

from urllib.parse import quote

from sqlalchemy import Column, Integer, String, Float, ForeignKey

from sqlalchemy.orm import relationship, backref

from sqlalchemy import event

import shutil

import mimetypes

import csv

import io

import os

from werkzeug.utils import secure\_filename

import csv

from sqlalchemy.exc import IntegrityError

from flask import send\_from\_directory

from utils.file\_upload import save\_document

from datetime import datetime, timedelta

from dateutil.relativedelta import relativedelta

from sqlalchemy.orm.session import object\_session

from sqlalchemy.ext.hybrid import hybrid\_property

from flask\_login import LoginManager, UserMixin, login\_user, logout\_user, current\_user, login\_required

from werkzeug.security import generate\_password\_hash, check\_password\_hash

from flask\_mail import Mail

from datetime import date

today = date.today()

from flask\_sqlalchemy import SQLAlchemy

from flask\_migrate import Migrate

from sqlalchemy.exc import IntegrityError

from flask import make\_response

import click

from flask import Flask

from sqlalchemy import func, extract

from sqlalchemy import Enum

import math

from typing import Dict, Union

import logging

from logging.handlers import RotatingFileHandler

import sys

from flask\_migrate import upgrade

from flask.cli import with\_appcontext

from dotenv import load\_dotenv

from sqlalchemy import Column, String, Enum, ForeignKey

from sqlalchemy.orm import validates

import enum

from concurrent\_log\_handler import ConcurrentRotatingFileHandler

from flask import Blueprint

from apscheduler.schedulers.background import BackgroundScheduler

from flask\_wtf import FlaskForm

from wtforms.validators import DataRequired, Email

from wtforms import StringField, FloatField, IntegerField, SelectField, DateField, SubmitField

import os

import re

import time

from werkzeug.utils import secure\_filename

from sqlalchemy.orm import joinedload

from flask import Flask

from flask\_mail import Mail, Message

from apscheduler.schedulers.background import BackgroundScheduler

from datetime import date, datetime, timedelta

from flask import Flask, render\_template

from flask\_mail import Mail, Message

from apscheduler.schedulers.background import BackgroundScheduler

from datetime import date, datetime, timedelta

import calendar

import os

from sqlalchemy import create\_engine, func, extract

from sqlalchemy.orm import sessionmaker

import pandas as pd

import numpy as np

import calendar

from sqlalchemy import func

# Load environment variables first

load\_dotenv()

# Create logs directory if missing

log\_dir = os.path.join(os.path.dirname(\_\_file\_\_), 'logs')

if not os.path.exists(log\_dir):

os.makedirs(log\_dir)

# Enum class definition

class SettlementTypeEnum(enum.Enum):

self\_settlement = "self"

third\_party = "third\_party"

@classmethod

def get\_display\_name(cls, value):

names = {

"self": "Self Settlement",

"third\_party": "Third Party Settlement"

}

return names.get(value.value if isinstance(value, cls) else value, value)

# Initialize Flask application

app = Flask(\_\_name\_\_)

app.secret\_key = os.getenv('SECRET\_KEY', 'supersecretkey')

# ======================

# LOGGING CONFIGURATION

# ======================

# Set base logger level

app.logger.setLevel(logging.DEBUG) # Capture all levels from DEBUG up

# Create common formatter

formatter = logging.Formatter(

'[%(asctime)s] %(levelname)s in %(module)s: %(message)s'

)

# Handler 1: Main application log (app.log)

main\_handler = ConcurrentRotatingFileHandler(

'app.log',

mode='a',

maxBytes=1024 \* 1024, # 1 MB per file

backupCount=5,

encoding='utf-8'

)

main\_handler.setLevel(logging.INFO) # Only INFO and higher

main\_handler.setFormatter(formatter)

app.logger.addHandler(main\_handler)

app.config.update({

'UPLOAD\_FOLDER': os.path.join(app.instance\_path, 'documents'),

'MAX\_CONTENT\_LENGTH': 16 \* 1024 \* 1024, # 16MB max upload

})

# Handler 2: Detailed debug log (logs/loan\_app.log)

debug\_handler = ConcurrentRotatingFileHandler(

os.path.join(log\_dir, 'loan\_app.log'),

mode='a',

maxBytes=512000, # 500 KB per file

backupCount=3,

encoding='utf-8'

)

debug\_handler.setLevel(logging.DEBUG) # All levels

debug\_handler.setFormatter(formatter)

app.logger.addHandler(debug\_handler)

# Handler 3: Console output for development

console\_handler = logging.StreamHandler()

console\_handler.setLevel(logging.DEBUG if app.debug else logging.INFO)

console\_handler.setFormatter(formatter)

app.logger.addHandler(console\_handler)

password = os.getenv('MAIL\_PASSWORD')

safe\_password = re.escape(password)

safe\_password = quote(password)

# Windows-specific UTF-8 console configuration

if os.name == 'nt':

import ctypes

try:

ctypes.windll.kernel32.SetConsoleCP(65001)

ctypes.windll.kernel32.SetConsoleOutputCP(65001)

except Exception as e:

app.logger.warning(f"Failed to set Windows console encoding: {str(e)}")

# Ensure UTF-8 encoding for stdout/stderr

if sys.stdout.encoding != 'UTF-8':

try:

sys.stdout.reconfigure(encoding='utf-8')

except AttributeError:

# Python < 3.7 compatibility

sys.stdout = open(sys.stdout.fileno(), 'w', encoding='utf-8', errors='replace')

if sys.stderr.encoding != 'UTF-8':

try:

sys.stderr.reconfigure(encoding='utf-8')

except AttributeError:

sys.stderr = open(sys.stderr.fileno(), 'w', encoding='utf-8', errors='replace')

# Initialization complete

app.logger.info("=" \* 50)

app.logger.info("Application logging initialized successfully")

app.logger.info(f"Log directory: {os.path.abspath(log\_dir)}")

app.logger.info(f"Python version: {sys.version}")

app.logger.info(f"System encoding: {sys.getdefaultencoding()}")

app.logger.info("=" \* 50)

# Add these template filters

@app.template\_filter('percent')

def percent\_filter(value):

"""Format decimal as percentage (0.035 → 3.500%)"""

try:

return f"{float(value)\*100:.3f}%"

except (ValueError, TypeError):

return "0.000%"

# Make sure this is placed BEFORE any routes that use the filter

@app.template\_filter('format\_currency')

def format\_currency\_filter(value):

"""Format number as currency (3000 → 3,000.00)"""

try:

return f"{float(value):,.2f}"

except (ValueError, TypeError):

return "0.00"

# Also add this filter as a global function for string formatting

@app.context\_processor

def utility\_processor():

def format\_currency(value):

try:

return f"{float(value):,.2f}"

except (ValueError, TypeError):

return "0.00"

return dict(format\_currency=format\_currency)

def format\_currency(value):

try:

value = float(value)

return "MWK{:,.2f}".format(value)

except (ValueError, TypeError):

return value

@app.template\_filter('money')

def money\_format(value):

try:

return f"{float(value):,.2f}"

except Exception:

return "0.00"

@app.template\_filter('time\_ago')

def time\_ago\_filter(dt):

if not isinstance(dt, datetime):

return dt

now = datetime.utcnow()

delta = relativedelta(now, dt)

if delta.years > 0:

return f"{delta.years} year(s) ago"

elif delta.months > 0:

return f"{delta.months} month(s) ago"

elif delta.days > 0:

return f"{delta.days} day(s) ago"

elif delta.hours > 0:

return f"{delta.hours} hour(s) ago"

elif delta.minutes > 0:

return f"{delta.minutes} minute(s) ago"

else:

return "just now"

# Register the filter

app.jinja\_env.filters['time\_ago'] = time\_ago\_filter

@app.template\_filter('datetimeformat')

def datetimeformat\_filter(value, format='%Y-%m-%d %H:%M'):

"""Custom datetime format filter"""

if value is None:

return ""

try:

return value.strftime(format)

except AttributeError:

return ""

app.jinja\_env.filters['datetimeformat'] = datetimeformat\_filter

# Add this to your app.py where you have other template filters

@app.template\_filter('format\_currency')

def format\_currency\_filter(value):

"""Format number as currency (3000 → 3,000.00)"""

try:

return f"{float(value):,.2f}"

except (ValueError, TypeError):

return "0.00"

# In your app.py or \_\_init\_\_.py

logging.basicConfig(

filename='app.log',

filemode='a',

encoding='utf-8', # <-- important

level=logging.INFO,

format='%(asctime)s - %(levelname)s - %(message)s'

)

# Determine the environment: "production" or "development"

db = SQLAlchemy()

migrate = Migrate()

mail = Mail()

# Create Flask app

app = Flask(\_\_name\_\_)

@app.template\_filter('unique')

def unique\_filter(items, attribute):

seen = set()

result = []

for item in items:

value = getattr(item, attribute)

if value not in seen:

seen.add(value)

result.append(item)

return result

# Use Postgres in production, SQLite locally

# Database configuration

database\_url = os.getenv("DATABASE\_URL", "")

if database\_url:

# Fix PostgreSQL URL format if needed

if database\_url.startswith("postgres://"):

database\_url = database\_url.replace("postgres://", "postgresql://", 1)

app.config['SQLALCHEMY\_DATABASE\_URI'] = database\_url

print(f"Connected to PRODUCTION DB: PostgreSQL")

else:

# Fallback to SQLite for development

app.config['SQLALCHEMY\_DATABASE\_URI'] = "sqlite:///customers.db"

print("Connected to DEVELOPMENT DB: sqlite:///customers.db")

# Email config (example: Gmail — replace with your own)

app.config.update(

MAIL\_SERVER='mail.kwachafinancialservices.com',

MAIL\_PORT=465,

MAIL\_USE\_SSL=True,

MAIL\_USERNAME=os.getenv('MAIL\_USERNAME'),

MAIL\_PASSWORD=os.getenv('MAIL\_PASSWORD'),

MAIL\_DEFAULT\_SENDER=os.getenv('MAIL\_USERNAME'),

SQLALCHEMY\_TRACK\_MODIFICATIONS=False

)

# File upload settings

app.config['UPLOAD\_FOLDER'] = 'uploads/documents'

app.config['MAX\_CONTENT\_LENGTH'] = 16 \* 1024 \* 1024 # 16MB

app.config['ALLOWED\_EXTENSIONS'] = {'png', 'jpg', 'jpeg', 'pdf', 'doc', 'docx'}

mail = Mail(app)

# Security

app.config['SECRET\_KEY'] = 'your-secret-key-123' # ✅ Change this in production!

# Initialize extensions with app

db.init\_app(app)

migrate.init\_app(app, db)

mail.init\_app(app)

login\_manager = LoginManager(app)

# Import models after initializing the db instance

from app import db

UPLOAD\_FOLDER = 'uploads/documents'

MAX\_CONTENT\_LENGTH = 16 \* 1024 \* 1024 # 16 MB max upload size

ALLOWED\_EXTENSIONS = {'png', 'jpg', 'jpeg', 'pdf', 'doc', 'docx'}

import socket

app.jinja\_env.filters['format\_currency'] = format\_currency

@app.route("/server-info")

def server\_info():

db\_uri = app.config['SQLALCHEMY\_DATABASE\_URI']

env = os.getenv("FLASK\_ENV", "development")

return {

"hostname": socket.gethostname(),

"database\_uri": db\_uri,

"environment": env

}

# app.py

from flask import send\_file, abort

from werkzeug.exceptions import NotFound

db.metadata.clear()

@app.route('/serve\_document/<int:doc\_id>')

def serve\_document(doc\_id):

doc = Document.query.get\_or\_404(doc\_id)

abs\_path = doc.get\_absolute\_path()

if not abs\_path or not os.path.exists(abs\_path):

app.logger.error(f"Document file missing: {doc.path}")

abort(404)

return send\_file(abs\_path)

# ---------------- Pricing Configuration ----------------

PRICING = {

3: {'rate': 0.035, 'origination': 0.15, 'insurance': 0.008, 'collection': 0.0025, 'crb': 3000},

6: {'rate': 0.035, 'origination': 0.15, 'insurance': 0.014, 'collection': 0.0025, 'crb': 3000},

9: {'rate': 0.035, 'origination': 0.15, 'insurance': 0.02, 'collection': 0.015, 'crb': 3000},

12: {'rate': 0.035, 'origination': 0.12, 'insurance': 0.026, 'collection': 0.01139, 'crb': 3000},

15: {'rate': 0.035, 'origination': 0.2, 'insurance': 0.0297, 'collection': 0.01493, 'crb': 3000},

18: {'rate': 0.035, 'origination': 0.2, 'insurance': 0.0358, 'collection': 0.014, 'crb': 3000},

24: {'rate': 0.035, 'origination': 0.2, 'insurance': 0.037, 'collection': 0.0125, 'crb': 3000},

36: {'rate': 0.035, 'origination': 0.3, 'insurance': 0.041, 'collection': 0.0112, 'crb': 3000},

48: {'rate': 0.035, 'origination': 0.3, 'insurance': 0.045, 'collection': 0.0095, 'crb': 3000},

}

# ---------------- Models ----------------

class PricingConfig(db.Model):

\_\_tablename\_\_ = 'pricing\_configs'

\_\_table\_args\_\_ = {'extend\_existing': True}

id = db.Column(db.Integer, primary\_key=True)

category = db.Column(db.String(50), nullable=False) # civil\_servant, private\_sector, sme

term\_months = db.Column(db.Integer, nullable=False)

interest\_rate = db.Column(db.Float, nullable=False)

origination\_fee = db.Column(db.Float, nullable=False)

insurance\_fee = db.Column(db.Float, nullable=False)

collection\_fee = db.Column(db.Float, nullable=False)

crb\_fee = db.Column(db.Float, nullable=False)

created\_at = db.Column(db.DateTime, default=datetime.utcnow)

updated\_at = db.Column(db.DateTime, default=datetime.utcnow, onupdate=datetime.utcnow)

apply\_to\_new = db.Column(db.Boolean, default=True)

apply\_to\_existing = db.Column(db.Boolean, default=False)

apply\_interest\_to\_existing = db.Column(db.Boolean, default=False)

apply\_collection\_to\_existing = db.Column(db.Boolean, default=False)

\_\_table\_args\_\_ = (

db.UniqueConstraint('category', 'term\_months', name='uq\_category\_term'),

)

from flask\_login import UserMixin

from werkzeug.security import generate\_password\_hash, check\_password\_hash

class User(db.Model, UserMixin):

\_\_tablename\_\_ = 'users'

id = db.Column(db.Integer, primary\_key=True)

username = db.Column(db.String(100), unique=True) # ✅

password\_hash = db.Column(db.String(512), nullable=False) # ✅

email = db.Column(db.String(150), nullable=False) # ✅

active = db.Column(db.Boolean, default=True, nullable=False) # ✅

role\_id = db.Column(db.Integer, db.ForeignKey('roles.id', name='fk\_users\_role\_id'))

role = db.relationship('Role', backref='users')

def set\_password(self, password):

self.password\_hash = generate\_password\_hash(password)

def check\_password(self, password):

return check\_password\_hash(self.password\_hash, password)

def \_\_repr\_\_(self):

return f'<User {self.username} - Role: {self.role.name}>' # Access role.name

\_\_table\_args\_\_ = (

db.UniqueConstraint('username', name='uq\_users\_username'),

db.UniqueConstraint('email', name='uq\_users\_email'),

)

class Role(db.Model):

\_\_tablename\_\_ = 'roles'

id = db.Column(db.Integer, primary\_key=True)

name = db.Column(db.String(50), unique=True)

permissions = db.relationship('Permission', secondary='role\_permissions')

def has\_permission(self, resource, action):

print(f"\nChecking permission for {resource}:{action}")

print(f"Role '{self.name}' has {len(self.permissions)} permissions:")

found = False

for perm in self.permissions:

print(f" - {perm.resource}:{perm.action}")

# Wildcard match

if perm.resource == '\*' and perm.action == '\*':

print(" WILDCARD PERMISSION FOUND - ACCESS GRANTED")

found = True

# Don't return yet to see all permissions

# Exact match

if perm.resource == resource and perm.action == action:

print(f" EXACT MATCH FOUND FOR {resource}:{action}")

found = True

# Also check for partial wildcards

for perm in self.permissions:

if perm.resource == '\*' and perm.action == action:

print(f" RESOURCE WILDCARD MATCH FOR \*:{action}")

found = True

if perm.resource == resource and perm.action == '\*':

print(f" ACTION WILDCARD MATCH FOR {resource}:\*")

found = True

print(f"ACCESS {'GRANTED' if found else 'DENIED'}")

return found

class Permission(db.Model):

\_\_tablename\_\_ = 'permissions'

id = db.Column(db.Integer, primary\_key=True)

resource = db.Column(db.String(50)) # e.g., 'customer', 'loan'

action = db.Column(db.String(50)) # e.g., 'create', 'approve'

role\_permissions = db.Table('role\_permissions',

db.Column('role\_id', db.Integer, db.ForeignKey('roles.id')),

db.Column('permission\_id', db.Integer, db.ForeignKey('permissions.id'))

)

from datetime import datetime

from app import db # or wherever your SQLAlchemy instance is

class Notification(db.Model):

\_\_tablename\_\_ = 'notifications'

id = db.Column(db.Integer, primary\_key=True)

# Correct FK target → users.id, matching your User model

recipient\_id = db.Column(db.Integer, db.ForeignKey('users.id'), nullable=True) # Null = global

recipient = db.relationship('User', backref='notifications') # user.notifications access

message = db.Column(db.Text, nullable=False)

type = db.Column(db.String(50), default='info') # Types: info, approval, warning, etc.

is\_read = db.Column(db.Boolean, default=False)

timestamp = db.Column(db.DateTime, default=datetime.utcnow)

def \_\_repr\_\_(self):

return f'<Notification type={self.type}, to={self.recipient\_id or "Admin"}>'

from functools import wraps

from flask import redirect, url\_for, flash

from flask\_login import current\_user

def role\_required(\*roles):

def decorator(f):

@wraps(f)

def wrapper(\*args, \*\*kwargs):

if not current\_user.is\_authenticated:

flash("Please log in to access this page.", "warning")

return redirect(url\_for("login"))

print(f"🔍 Current user: {current\_user.username}, Role: {current\_user.role.name}")

if current\_user.role.name not in roles:

flash("You do not have permission to access this page.", "danger")

return redirect(url\_for("home"))

return f(\*args, \*\*kwargs)

return wrapper

return decorator

login\_manager = LoginManager()

login\_manager.init\_app(app)

login\_manager.login\_view = 'login'

@login\_manager.user\_loader

def load\_user(user\_id):

return User.query.get(int(user\_id))

class AccountingError(Exception):

"""Custom exception for accounting discrepancies"""

def \_\_init\_\_(self, message="Accounting discrepancy detected"):

self.message = message

super().\_\_init\_\_(self.message)

class Agent(db.Model):

\_\_tablename\_\_ = 'agents'

id = db.Column(db.Integer, primary\_key=True)

name = db.Column(db.String(100), nullable=False)

contact = db.Column(db.String(20))

email = db.Column(db.String(100))

district = db.Column(db.String(100))

region = db.Column(db.String(100))

monthly\_budget = db.Column(db.Float, default=0.0)

role = db.Column(db.String(50))

active = db.Column(db.Boolean, default=True)

# Self-referencing relationship

team\_leader\_id = db.Column(db.Integer, db.ForeignKey('agents.id'), nullable=True)

team\_leader = db.relationship('Agent', remote\_side=[id], backref='team\_members')

# If you have customers

customers = db.relationship('Customer', back\_populates='agent', lazy=True)

class Customer(db.Model):

\_\_tablename\_\_ = 'customers'

\_\_table\_args\_\_ = (

db.UniqueConstraint('national\_id', name='uq\_customers\_national\_id'),

db.UniqueConstraint('file\_number', name='uq\_customers\_file\_number'),

db.UniqueConstraint('employment\_number', name='uq\_customers\_employment\_number'),

db.Index('idx\_customers\_national\_id', 'national\_id'),

db.Index('idx\_customers\_file\_number', 'file\_number'),

db.Index('idx\_customers\_employment\_number', 'employment\_number'),

db.Index('idx\_customers\_agent\_id', 'agent\_id'),

)

id = db.Column(db.Integer, primary\_key=True)

national\_id = db.Column(db.String(20), unique=True, nullable=False)

first\_name = db.Column(db.String(100), nullable=False)

last\_name = db.Column(db.String(100), nullable=False)

gender = db.Column(db.String(10))

dob = db.Column(db.String(20))

title = db.Column(db.String(20))

email = db.Column(db.String(100), nullable=False)

contact = db.Column(db.String(20))

address = db.Column(db.String(255))

next\_of\_kin\_name = db.Column(db.String(20),nullable=False)

next\_of\_kin\_relationship = db.Column(db.String(20),nullable=False)

next\_of\_kin\_contact = db.Column(db.String(20))

employer = db.Column(db.String(100), nullable=False)

job\_title = db.Column(db.String(100))

salary = db.Column(db.Float)

service\_length = db.Column(db.String(50))

bank\_name = db.Column(db.String(100))

bank\_account = db.Column(db.String(20), nullable=False)

salary\_deposited = db.Column(db.String(10))

district = db.Column(db.String(100))

region = db.Column(db.String(100))

amount\_requested = db.Column(db.Float)

status = db.Column(db.String(20), default='pending')

is\_approved\_for\_creation = db.Column(db.Boolean, default=False)

maker\_id = db.Column(db.Integer, nullable=False)

checker\_id = db.Column(db.Integer)

is\_approved\_for\_deletion = db.Column(db.Boolean, default=False)

file\_number = db.Column(db.String(20))

created\_at = db.Column(db.DateTime, default=datetime.utcnow)

date\_joined = db.Column(db.Date, nullable=True, index=True)

is\_voluntary\_retirement\_candidate = db.Column(db.Boolean, default=False)

agent\_id = db.Column(db.Integer, db.ForeignKey('agents.id'), nullable=True)

employment\_number = db.Column(db.String(20), unique=True, nullable=True)

loans = db.relationship('LoanApplication', back\_populates='customer')

agent = db.relationship('Agent', back\_populates='customers')

customer\_documents = db.relationship("Document", back\_populates="customer", lazy=True)

def \_\_repr\_\_(self):

return f'<Customer {self.first\_name} {self.last\_name}, Status: {self.status}>'

@property

def age(self):

if self.dob:

today = date.today()

return today.year - self.dob.year - ((today.month, today.day) < (self.dob.month, self.dob.day))

return None

@property

def years\_in\_service(self):

if self.date\_joined:

today = date.today()

return today.year - self.date\_joined.year - ((today.month, today.day) < (self.date\_joined.month, self.date\_joined.day))

return None

@property

def is\_voluntary\_retirement\_candidate(self):

return self.years\_in\_service is not None and self.years\_in\_service >= 20

@property

def full\_name(self):

return f"{self.first\_name} {self.last\_name}"

class CustomerQueryForm(FlaskForm):

national\_id = StringField('National ID', validators=[Optional()])

employment\_number = StringField('Employment Number', validators=[Optional()])

query\_submit = SubmitField('Search Customer')

class CutoffDateConfig(db.Model):

id = db.Column(db.Integer, primary\_key=True)

category = db.Column(db.String(50), unique=True, nullable=False) # 'civil\_servant', 'private\_sector', etc.

cutoff\_dt = db.Column(db.DateTime, nullable=False)

updated\_at = db.Column(db.DateTime, default=datetime.utcnow, onupdate=datetime.utcnow)

class LoanApplication(db.Model):

\_\_tablename\_\_ = 'loan\_applications'

\_\_table\_args\_\_ = (

db.UniqueConstraint('loan\_number', name='uq\_loan\_applications\_loan\_number'),

db.Index('idx\_loan\_applications\_loan\_number', 'loan\_number'),

db.Index('idx\_loan\_applications\_customer\_id', 'customer\_id'),

db.Index('idx\_loan\_applications\_agent\_id', 'agent\_id'),

)

id = db.Column(db.Integer, primary\_key=True)

customer\_id = db.Column(db.Integer, db.ForeignKey('customers.id'), nullable=False)

loan\_amount = db.Column(db.Float,nullable=False, default=0.0)

status = db.Column(db.String(20), default='pending')

created\_at = db.Column(db.DateTime, default=datetime.utcnow)

term\_months = db.Column(db.Integer)

monthly\_instalment = db.Column(db.Float)

total\_repayment = db.Column(db.Float)

effective\_rate = db.Column(db.Float)

category = db.Column(db.String(50))

loan\_category = db.Column(db.Integer, nullable=False)

disbursed = db.Column(db.Boolean, default=False)

disbursed\_bank = db.Column(db.String(100))

crb\_fees = db.Column(db.Float, default=3000)

origination\_fees = db.Column(db.Float)

insurance\_fees = db.Column(db.Float)

total\_fees = db.Column(db.Float)

collection\_fees = db.Column(db.Float)

schedule\_id = db.Column(

db.Integer,

db.ForeignKey('repayment\_schedules.id', use\_alter=True, name='fk\_schedule\_id'),

nullable=True

)

loan\_number = db.Column(db.String(20), nullable=True, unique=True)

file\_number = db.Column(db.String(50))

date\_created = db.Column(db.DateTime, default=datetime.utcnow)

disbursement\_date = db.Column(db.Date, nullable=True)

cash\_to\_client = db.Column(db.Float,nullable=False, default=0.0)

top\_up\_interest = db.Column(db.Float, default=0.0)

settlement\_interest = db.Column(db.Float, default=0.0)

closure\_type = db.Column(db.String(20)) # 'settlement' or 'topup'

closure\_date = db.Column(db.DateTime)

top\_up\_of = db.Column(db.Integer, db.ForeignKey('loan\_applications.id'), nullable=True)

application\_status = db.Column(db.String(20), nullable=False, default='pending')

loan\_state = db.Column(db.String(20), nullable=False, default='active')

performance\_status = db.Column(db.String(20), nullable=False, default='performing')

top\_up\_balance = db.Column(db.Float, default=0.0)

settlement\_balance = db.Column(db.Float, default=0.0)

current\_balance = db.Column(db.Float, default=0.0)

settlement\_type = Column(Enum(SettlementTypeEnum), nullable=True)

settling\_institution = Column(String(255), nullable=True) # only if third\_party

settlement\_reason = db.Column(db.String(255), nullable=True)

vote\_id = db.Column(db.Integer, db.ForeignKey('votes.id'))

outstanding\_fees = db.Column(db.Float, default=0.0)

pricing\_version = db.Column(db.Integer, default=1)

applied\_interest\_rate = db.Column(db.Float)

applied\_collection\_fee = db.Column(db.Float)

written\_off\_amount = db.Column(db.Float, default=0.0)

insurance\_settlement\_amount = db.Column(db.Float, default=0.0)

agent\_id = db.Column(db.Integer, db.ForeignKey('agents.id'), nullable=True)

parent\_loan\_id = db.Column(

db.Integer,

db.ForeignKey('loan\_applications.id'),

nullable=True

)

vote = db.relationship('Vote', backref='loan\_applications')

documents = db.relationship("Document", back\_populates="loan", lazy=True)

topups = db.relationship(

'LoanApplication',

foreign\_keys=[parent\_loan\_id], # 👈 Specify which foreign key to use

backref=db.backref('parent\_loan', remote\_side=[id]),

cascade='all, delete-orphan'

)

agent = db.relationship("Agent", backref="loan\_applications", foreign\_keys=[agent\_id])

payments = db.relationship('Payment', back\_populates='loan', cascade='all, delete-orphan')

customer = db.relationship('Customer', back\_populates='loans')

repayment\_schedules = db.relationship(

'RepaymentSchedule',

back\_populates='loan',

cascade='all, delete-orphan',

foreign\_keys='RepaymentSchedule.loan\_id'

)

@validates('settlement\_reason')

def validate\_reason(self, key, value):

allowed\_reasons = {"price", "stay debt free", "consolidation", "better terms", "other"}

if value.lower() not in allowed\_reasons:

raise ValueError(f"Invalid settlement reason: {value}")

return value.lower()

@validates("settlement\_type")

def validate\_type(self, key, value):

if value is not None and not isinstance(value, SettlementTypeEnum):

raise ValueError("Invalid settlement type")

return value

@property

def total\_arrears(self):

return sum(s.arrears\_amount for s in self.repayment\_schedules)

@property

def max\_arrears\_days(self):

max\_days = 0

for s in self.repayment\_schedules:

if s.due\_date < datetime.utcnow().date() and s.status != 'paid':

days = (datetime.utcnow().date() - s.due\_date).days

if days > max\_days:

max\_days = days

return max\_days

# In your LoanApplication model

@property

def is\_topup(self):

return self.parent\_loan\_id is not None

def \_\_repr\_\_(self):

return f'<LoanApplication for Customer ID {self.customer\_id} - Status: {self.status}>'

@property

def computed\_current\_balance(self):

from datetime import datetime

if self.loan\_state in ('settled\_client', 'write\_off', 'insurance'):

return 0.0

config = get\_pricing\_config(self.category, self.term\_months, self)

if not config:

return 0.0

def calculate\_capitalized\_amount(loan\_amount: float, config: dict) -> float:

try:

origination = loan\_amount \* config.get('origination', 0)

insurance = loan\_amount \* config.get('insurance', 0)

crb = config.get('crb', 0)

return round(loan\_amount + origination + insurance + crb, 2)

except Exception as e:

app.logger.warning(f"[TOPUP] Capitalization error: {e}")

return loan\_amount

capitalized = calculate\_capitalized\_amount(self.loan\_amount or 0, config)

monthly\_rate = config.get('rate', 0)

term = self.term\_months or 0

if monthly\_rate > 0 and term > 0:

factor = (monthly\_rate \* (1 + monthly\_rate) \*\* term) / ((1 + monthly\_rate) \*\* term - 1)

annuity = capitalized \* factor

else:

annuity = 0

payments = sorted(self.payments, key=lambda p: p.created\_at)

remaining\_balance = capitalized

payments\_made = 0

for p in payments:

if p.allocation and p.allocation.principal:

remaining\_balance -= p.allocation.principal

remaining\_balance = max(remaining\_balance, 0)

payments\_made += 1

return round(remaining\_balance, 2)

@property

def balance(self):

if self.loan\_state in ('settled\_client', 'write\_off', 'insurance'):

return 0.0

config = get\_pricing\_config(self.category, self.term\_months, self)

if not config:

return None

capitalized = (

self.loan\_amount +

(self.loan\_amount \* config.get('origination', 0)) +

(self.loan\_amount \* config.get('insurance', 0)) +

config.get('crb', 0)

)

paid\_principal = sum(

alloc.principal or 0

for p in self.payments

for alloc in p.allocations

)

return round(capitalized - paid\_principal, 2)

@property

def total\_arrears(self):

return sum(a.total\_arrears for a in self.arrears if a.status == 'unresolved')

@property

def calculated\_balance(self):

if self.loan\_state in ('settled\_client', 'write\_off', 'insurance'):

return 0.0

config = get\_pricing\_config(self.category, self.term\_months, self)

if not config:

return 0.0

capitalized = (

self.loan\_amount +

(self.loan\_amount \* config.get('origination', 0)) +

(self.loan\_amount \* config.get('insurance', 0)) +

config.get('crb', 0)

)

paid\_principal = sum(

alloc.principal or 0

for p in self.payments

for alloc in p.allocations

)

return round(capitalized - paid\_principal, 2)

def recalculate\_balance(self):

"""Production-proven balance calculation"""

# Skip if already closed

if self.loan\_state in {"settled\_client", "write\_off", "insurance"}:

self.current\_balance = 0.0

self.top\_up\_balance = 0.0

self.settlement\_balance = 0.0

return

config = get\_pricing\_config(self.category, self.term\_months, self)

if not config:

app.logger.error(f"No pricing config for {self.loan\_number}")

return

# Calculate capitalized amount

capitalized = (

self.loan\_amount +

(self.loan\_amount \* config.get('origination', 0)) +

(self.loan\_amount \* config.get('insurance', 0)) +

config.get('crb', 0)

)

# Calculate paid principal (only from successful payments)

paid\_principal = sum(

alloc.principal

for payment in self.payments

if payment.status in ['successful', 'completed']

for alloc in payment.allocations

)

# Calculate current balance

current\_balance = max(round(capitalized - paid\_principal, 2), 0.0)

# Update balances based on loan state

if self.loan\_state == 'active':

self.current\_balance = current\_balance

self.top\_up\_balance = current\_balance + (self.top\_up\_interest or 0.0)

self.settlement\_balance = current\_balance + (self.settlement\_interest or 0.0)

else:

self.current\_balance = 0.0

self.top\_up\_balance = 0.0

self.settlement\_balance = 0.0

app.logger.info(

f"[{self.loan\_number}] Recalculated: "

f"capitalized={capitalized}, paid\_principal={paid\_principal}, "

f"current\_balance={self.current\_balance}"

)

def get\_cutoff\_day\_for\_civil\_servant(self) -> int | None:

"""

Returns an integer 1-31 for the civil-servant cut-off day,

or None if the admin hasn't set one yet.

"""

rec = CutoffDateConfig.query.first()

if rec and rec.cutoff\_dt:

return rec.cutoff\_dt.day

return None

def get\_first\_due\_date(self, disbursement\_date=None):

"""

Determine the first instalment due date with optional custom disbursement date

"""

import calendar

from datetime import date as \_date

from dateutil.relativedelta import relativedelta

# Use custom disbursement date if provided, otherwise use loan's disbursement date

disb\_date = disbursement\_date or (self.disbursement\_date or \_date.today())

due\_day = 25

cat = (self.category or "").lower()

if cat == "private\_sector":

target\_month = disb\_date if disb\_date.day <= 15 else disb\_date + relativedelta(months=1)

elif cat == "civil\_servant":

cutoff\_day = self.get\_cutoff\_day\_for\_civil\_servant()

target\_month = disb\_date + relativedelta(months=1)

if cutoff\_day and disb\_date.day > cutoff\_day:

target\_month += relativedelta(months=1)

else:

target\_month = disb\_date + relativedelta(months=1)

last\_day = calendar.monthrange(target\_month.year, target\_month.month)[1]

return target\_month.replace(day=min(due\_day, last\_day))

def generate\_repayment\_schedule(self, disbursement\_date=None):

"""

Build/replace this loan's amortised schedule with optional backdating

"""

from dateutil.relativedelta import relativedelta

# Wipe existing rows

for sched in self.repayment\_schedules:

db.session.delete(sched)

# Get pricing config

config = get\_pricing\_config(self.category, self.term\_months, self)

if not config:

return

loan\_amt = self.loan\_amount or 0

origination = loan\_amt \* config.get('origination', 0)

insurance = loan\_amt \* config.get('insurance', 0)

crb = config.get('crb', 0)

capitalised = loan\_amt + origination + insurance + crb

monthly\_rate = config.get('rate', 0)

coll\_fee\_flat = loan\_amt \* config.get('collection', 0)

term = self.term\_months or 0

if term <= 0:

return

# Get first due date using custom disbursement date if provided

first\_due = self.get\_first\_due\_date(disbursement\_date)

# Calculate annuity payment

if monthly\_rate > 0:

fac = (monthly\_rate \* (1 + monthly\_rate) \*\* term) / ((1 + monthly\_rate) \*\* term - 1)

annuity\_princ\_int = capitalised \* fac

else:

annuity\_princ\_int = capitalised / term

remaining = capitalised

# Create new schedule

for i in range(term):

due\_date = first\_due + relativedelta(months=i)

interest = remaining \* monthly\_rate

principal = annuity\_princ\_int - interest

# Handle last instalment

if i == term - 1:

principal = remaining

interest = annuity\_princ\_int - principal

annuity\_princ\_int = principal + interest

remaining -= principal

remaining = max(0, round(remaining, 2))

schedule = RepaymentSchedule(

loan\_id=self.id,

instalment\_no=i + 1,

due\_date=due\_date,

expected\_principal=round(principal, 2),

expected\_interest=round(interest, 2),

expected\_fees=round(coll\_fee\_flat, 2),

expected\_amount=round(principal + interest + coll\_fee\_flat, 2),

remaining\_balance=remaining

)

db.session.add(schedule)

# Do NOT commit here - handled by caller

from datetime import datetime

from dateutil.relativedelta import relativedelta # already using

import calendar

# … other methods in LoanApplication …

def allocate\_payment(self, payment):

if self.loan\_state in {"settled\_client", "write\_off", "insurance"}:

app.logger.warning(f"[{self.loan\_number}] Payment not allocated. Loan state is closed: {self.loan\_state}")

return

remaining = payment.amount

method = (payment.method or "normal").lower()

if "top\_up" in method:

method = "top\_up"

elif "settlement" in method:

method = "settlement"

if method == "top\_up":

principal\_alloc = min(self.top\_up\_balance - (self.top\_up\_interest or 0.0), remaining)

self.current\_balance -= principal\_alloc

remaining -= principal\_alloc

interest\_alloc = min((self.top\_up\_interest or 0.0), remaining)

self.top\_up\_interest = max((self.top\_up\_interest or 0.0) - interest\_alloc, 0.0)

remaining -= interest\_alloc

# Save allocation

db.session.add(PaymentAllocation(

payment\_id=payment.id,

principal=principal\_alloc,

interest=0.0,

top\_up\_interest=interest\_alloc,

settlement\_interest=0.0,

fees=0.0

))

app.logger.info(f"[{self.loan\_number}] Top-up payment → principal: {principal\_alloc}, top-up interest: {interest\_alloc}")

# Commit changes to LoanApplication and allocations

db.session.commit()

# Close if fully paid

if self.current\_balance <= 0 and (self.top\_up\_interest or 0.0) <= 0:

self.status = "closed"

self.loan\_state = "settled\_client"

app.logger.info(f"[{self.loan\_number}] Loan marked as closed after top-up.")

for schedule in self.repayment\_schedules:

if schedule.status not in {"paid", "cancelled"}:

schedule.status = "cancelled"

db.session.commit()

if remaining > 0:

self.record\_loan\_credit(payment, remaining)

return # skip normal schedule allocation

if method == "settlement":

interest\_alloc = min(self.settlement\_interest or 0.0, remaining)

self.settlement\_interest = max((self.settlement\_interest or 0.0) - interest\_alloc, 0.0)

remaining -= interest\_alloc

principal\_alloc = min(self.current\_balance, remaining)

self.current\_balance -= principal\_alloc

remaining -= principal\_alloc

db.session.add(PaymentAllocation(

payment\_id=payment.id,

principal=principal\_alloc,

interest=0.0,

settlement\_interest=interest\_alloc,

fees=0.0

))

app.logger.info(f"[{self.loan\_number}] Settlement payment → settlement\_interest: {interest\_alloc}, principal: {principal\_alloc}")

# Commit changes to LoanApplication and allocations

db.session.commit()

# Close loan if fully paid

if self.current\_balance <= 0 and (self.settlement\_interest or 0.0) <= 0:

self.status = "closed"

self.loan\_state = "settled\_client"

app.logger.info(f"[{self.loan\_number}] Loan marked as closed after settlement payment.")

db.session.commit()

if remaining > 0:

self.record\_loan\_credit(payment, remaining)

return

# --- Normal payment allocation logic (fees → interest → principal) ---

schedules = sorted(self.repayment\_schedules, key=lambda s: s.due\_date)

schedule\_updated = False

for schedule in schedules:

if schedule.status in {"paid", "settled", "cancelled"}:

continue

alloc\_fees = min(schedule.fees\_due, remaining)

schedule.paid\_fees += alloc\_fees

remaining -= alloc\_fees

alloc\_interest = min(schedule.interest\_due, remaining)

schedule.paid\_interest += alloc\_interest

remaining -= alloc\_interest

alloc\_principal = min(schedule.principal\_due, remaining)

schedule.paid\_principal += alloc\_principal

remaining -= alloc\_principal

if alloc\_fees > 0 or alloc\_interest > 0 or alloc\_principal > 0:

db.session.add(PaymentAllocation(

payment\_id=payment.id,

schedule\_id=schedule.id,

principal=alloc\_principal,

interest=alloc\_interest,

fees=alloc\_fees

))

if schedule.paid\_amount >= schedule.expected\_amount:

schedule.status = "paid"

schedule\_updated = True

app.logger.info(

f"[{self.loan\_number}] Schedule {schedule.id} updated → fees={schedule.paid\_fees}, "

f"interest={schedule.paid\_interest}, principal={schedule.paid\_principal}"

)

if remaining <= 0:

break

if schedule\_updated:

self.recalculate\_balance()

db.session.commit()

if remaining > 0:

self.record\_loan\_credit(payment, remaining)

elif any(

s.status != "paid" and (s.fees\_due > 0 or s.interest\_due > 0 or s.principal\_due > 0)

for s in self.repayment\_schedules

):

self.record\_arrears(payment)

app.logger.info(f"[{self.loan\_number}] Payment of {payment.amount} allocated. Remaining: {remaining}")

def record\_loan\_credit(self, payment, amount: float):

"""

Store a loan credit when an overpayment occurs.

This is not applied or refunded yet — pending approval or further action.

"""

if amount <= 0:

raise ValueError("Credit amount must be positive")

credit = LoanCredit(

loan=self,

amount=round(amount, 2),

created\_at=datetime.utcnow()

# Optionally link to payment if your model allows

)

db.session.add(credit)

print(f"[{self.loan\_number}] Overpayment of {amount} stored as loan credit (ID will be set on flush).")

def expected\_payment\_due(self):

"""

Total due amount across all unpaid schedules.

Includes fees, interest, and principal.

"""

total\_due = 0.0

for schedule in self.repayment\_schedules:

if schedule.status != 'paid':

total\_due += (

(schedule.fees\_due - schedule.paid\_fees) +

(schedule.interest\_due - schedule.paid\_interest) +

(schedule.principal\_due - schedule.paid\_principal)

)

return round(total\_due, 2)

def record\_arrears(self, payment):

"""

Create or update arrears if payment is insufficient.

"""

for schedule in self.repayment\_schedules:

if schedule.status == 'paid' or schedule.due\_date >= date.today():

continue

existing = Arrear.query.filter\_by(schedule\_id=schedule.id).first()

if not existing:

# New arrear

arrear = Arrear(

loan\_id=self.id,

schedule\_id=schedule.id,

due\_date=schedule.due\_date,

expected\_principal=schedule.expected\_principal,

expected\_interest=schedule.expected\_interest,

expected\_fees=schedule.expected\_fees,

paid\_principal=schedule.paid\_principal,

paid\_interest=schedule.paid\_interest,

paid\_fees=schedule.paid\_fees,

status='unresolved'

)

db.session.add(arrear)

else:

# Update partial arrear

existing.paid\_principal = schedule.paid\_principal

existing.paid\_interest = schedule.paid\_interest

existing.paid\_fees = schedule.paid\_fees

existing.recorded\_at = datetime.utcnow()

if existing.total\_arrears <= 0:

existing.status = 'resolved'

def update\_arrears\_status(self):

"""Automatically update arrear status based on payments"""

for arrear in self.arrears:

if arrear.status == 'unresolved' and arrear.total\_arrears <= 0:

arrear.status = 'resolved'

arrear.resolution\_date = datetime.utcnow()

from app import db, app

from datetime import date

class Vote(db.Model):

\_\_tablename\_\_ = 'votes'

id = db.Column(db.Integer, primary\_key=True)

code = db.Column(db.String(10), unique=True, nullable=False)

description = db.Column(db.String(200), nullable=False)

is\_active = db.Column(db.Boolean, default=True)

def \_\_repr\_\_(self):

return f'<Vote {self.code}: {self.description}>'

class Disbursement(db.Model):

\_\_tablename\_\_ = 'disbursements'

id = db.Column(db.Integer, primary\_key=True)

loan\_id = db.Column(db.Integer, db.ForeignKey('loan\_applications.id'), nullable=False)

amount = db.Column(db.Float, nullable=False)

method = db.Column(db.String(20)) # e.g., 'bank', 'mpesa'

status = db.Column(db.String(20), default='pending') # 'pending', 'successful', etc.

reference = db.Column(db.String(255))

created\_at = db.Column(db.DateTime, default=datetime.utcnow)

# This sets up the relationship properly

loan = db.relationship('LoanApplication', backref='disbursements')

class RepaymentSchedule(db.Model):

\_\_tablename\_\_ = 'repayment\_schedules'

id = db.Column(db.Integer, primary\_key=True)

loan\_id = db.Column(db.Integer, db.ForeignKey('loan\_applications.id'))

instalment\_no = db.Column(db.Integer)

due\_date = db.Column(db.Date)

expected\_amount = db.Column(db.Float)

expected\_principal = db.Column(db.Float)

expected\_interest = db.Column(db.Float)

expected\_fees = db.Column(db.Float)

status = db.Column(db.String(20))

paid\_principal = db.Column(db.Float, default=0.0)

paid\_interest = db.Column(db.Float, default=0.0)

paid\_fees = db.Column(db.Float, default=0.0)

remaining\_balance = db.Column(db.Float, default=0.0)

status = db.Column(db.String(20), default='pending')

arrears\_amount = db.Column(db.Float, default=0.0)

loan = db.relationship(

"LoanApplication",

back\_populates="repayment\_schedules",

foreign\_keys=[loan\_id]

)

@property

def arrears\_days(self):

if self.due\_date < date.today() and self.status != 'paid':

return (date.today() - self.due\_date).days

return 0

@property

def paid\_amount(self):

return self.paid\_principal + self.paid\_interest + self.paid\_fees

@property

def fees\_due(self):

return self.expected\_fees - self.paid\_fees

@property

def interest\_due(self):

return self.expected\_interest - self.paid\_interest

@property

def principal\_due(self):

return self.expected\_principal - self.paid\_principal

@property

def due\_amount(self):

principal\_due = (self.expected\_principal or 0.0) - (self.paid\_principal or 0.0)

interest\_due = (self.expected\_interest or 0.0) - (self.paid\_interest or 0.0)

fees\_due = (self.expected\_fees or 0.0) - (self.paid\_fees or 0.0)

return max(0.0, principal\_due + interest\_due + fees\_due)

class LoanCredit(db.Model):

\_\_tablename\_\_ = 'loan\_credits'

id = db.Column(db.Integer, primary\_key=True)

loan\_id = db.Column(db.Integer, db.ForeignKey('loan\_applications.id'))

amount = db.Column(db.Float, default=0.0)

created\_at = db.Column(db.DateTime, default=datetime.utcnow)

applied\_at = db.Column(db.DateTime, nullable=True)

refunded\_at = db.Column(db.DateTime, nullable=True)

loan = db.relationship('LoanApplication', backref='credits')

class Payment(db.Model):

\_\_tablename\_\_ = 'payments'

id = db.Column(db.Integer, primary\_key=True)

loan\_id = db.Column(db.Integer, db.ForeignKey('loan\_applications.id'), nullable=False)

amount = db.Column(db.Float, nullable=False)

reference = db.Column(db.String(100))

method = db.Column(db.String(50))

status = db.Column(db.String(20), default='pending')

created\_at = db.Column(db.DateTime, default=datetime.utcnow, nullable=False)

settlement\_proof = db.Column(db.String(255))

is\_allocated = db.Column(db.Boolean, default=False)

# Relationships

loan = db.relationship('LoanApplication', back\_populates='payments')

allocations = db.relationship(

'PaymentAllocation',

back\_populates='payment',

cascade='all, delete-orphan'

)

from sqlalchemy.orm.session import object\_session

from sqlalchemy.orm import Session

from sqlalchemy import event

@event.listens\_for(Session, "after\_commit")

def allocate\_after\_commit(session):

try:

for obj in session.new:

if isinstance(obj, Payment):

app.logger.info(f"✅ Allocating after commit: Payment ID {obj.id}")

new\_session = None

try:

new\_session = Session(bind=session.bind)

# Eagerly load the loan relationship

payment = new\_session.query(Payment).options(

db.joinedload(Payment.loan)

).get(obj.id)

if not payment:

app.logger.warning(f"⚠️ Payment ID {obj.id} not found.")

continue

if not payment.loan:

app.logger.warning(f"⚠️ Loan not found for Payment ID {obj.id}")

continue

loan = payment.loan # ✅ Now it's safe

PaymentAllocator(payment).process()

loan.recalculate\_balance()

new\_session.commit()

except Exception as inner\_error:

app.logger.error(

f"❌ Error processing Payment ID {obj.id}: {inner\_error}",

exc\_info=True

)

if new\_session:

new\_session.rollback()

finally:

if new\_session:

new\_session.close()

except Exception as e:

app.logger.error(f"❌ Error in after\_commit outer block: {e}", exc\_info=True)

import math

def calculate\_irr\_schedule(principal: float, interest\_rate: float,

term\_months: int, start\_date: date) -> list:

"""

Generates an IRR-based amortization schedule

"""

monthly\_rate = interest\_rate

monthly\_payment = principal \* (monthly\_rate \* (1 + monthly\_rate)\*\*term\_months) / \

((1 + monthly\_rate)\*\*term\_months - 1)

balance = principal

schedule = []

for i in range(1, term\_months + 1):

interest = balance \* monthly\_rate

principal\_component = monthly\_payment - interest

balance -= principal\_component

schedule.append({

'due\_date': start\_date + relativedelta(months=i),

'total\_payment': monthly\_payment,

'principal': principal\_component,

'interest': interest,

'balance': balance

})

return schedule

# Helper methods would be implemented below...

# \_get\_overdue\_schedules, \_get\_current\_schedule, etc.

class PaymentAllocation(db.Model):

\_\_tablename\_\_ = 'payment\_allocations'

id = db.Column(db.Integer, primary\_key=True)

payment\_id = db.Column(db.Integer, db.ForeignKey('payments.id'), nullable=False)

schedule\_id = db.Column(db.Integer, db.ForeignKey('repayment\_schedules.id'), nullable=True)

principal = db.Column(db.Float, default=0.0)

interest = db.Column(db.Float, default=0.0)

fees = db.Column(db.Float, default=0.0)

# ✅ These must be defined

top\_up\_interest = db.Column(db.Float, default=0.0) # <-- THIS LINE IS REQUIRED

settlement\_interest = db.Column(db.Float, default=0.0)

created\_at = db.Column(db.DateTime, default=datetime.utcnow)

# Relationships

payment = db.relationship('Payment', back\_populates='allocations')

schedule = db.relationship(

'RepaymentSchedule',

primaryjoin='PaymentAllocation.schedule\_id == RepaymentSchedule.id',

foreign\_keys=[schedule\_id],

backref='allocations'

)

# Remove unique constraint on payment\_id to allow multiple allocations per payment

# \_\_table\_args\_\_ = (

# db.UniqueConstraint('payment\_id', name='uq\_payment\_allocation\_payment\_id'),

# )

class JournalEntry(db.Model):

\_\_tablename\_\_ = 'journal\_entries'

id = db.Column(db.Integer, primary\_key=True)

date = db.Column(db.DateTime, default=datetime.utcnow)

description = db.Column(db.String(200))

amount = db.Column(db.Float) # Positive for income, negative for assets

entry\_type = db.Column(db.String(50)) # principal\_recovery/interest\_income

gl\_account = db.Column(db.String(50)) # GL account code

user\_id = db.Column(db.Integer, db.ForeignKey('users.id'))

loan\_id = db.Column(db.Integer, db.ForeignKey('loan\_applications.id'))

user = db.relationship('User')

loan = db.relationship('LoanApplication')

class Document(db.Model):

\_\_tablename\_\_ = 'documents'

id = db.Column(db.Integer, primary\_key=True)

customer\_id = db.Column(db.Integer, db.ForeignKey('customers.id'), nullable=False)

loan\_id = db.Column(db.Integer, db.ForeignKey('loan\_applications.id')) # Optional for loan-specific docs

filename = db.Column(db.String(255), nullable=False)

filetype = db.Column(db.String(50), nullable=False) # 'id\_front', 'id\_back', 'photo', 'payslip', etc.

path = db.Column(db.String(512), nullable=False)

uploaded\_at = db.Column(db.DateTime, default=datetime.utcnow)

# Relationships

customer = db.relationship("Customer", back\_populates="customer\_documents")

loan = db.relationship("LoanApplication", back\_populates="documents")

def get\_absolute\_path(self):

"""Get absolute file path with safety checks"""

if not self.path:

return None

try:

# Handle absolute paths directly

if os.path.isabs(self.path):

return self.path

# First try: app root relative path

root\_relative = os.path.join(app.root\_path, self.path)

if os.path.exists(root\_relative):

return root\_relative

# Second try: instance documents folder

instance\_path = os.path.join(app.instance\_path, 'documents', os.path.basename(self.path))

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

return instance\_path

# Third try: original path as is

if os.path.exists(self.path):

return self.path

return None

except Exception:

# Log error in production

app.logger.error(f"Error resolving path for document {self.id}")

return None

@property

def absolute\_path(self):

return self.get\_absolute\_path()

@property

def file\_exists(self):

"""Check if file exists on filesystem"""

path = self.absolute\_path

return path and os.path.exists(path)

class PARSnapshot(db.Model):

\_\_tablename\_\_ = 'par\_snapshots'

id = db.Column(db.Integer, primary\_key=True)

snapshot\_date = db.Column(db.Date, nullable=False, unique=True)

par\_30 = db.Column(db.Float, nullable=False)

par\_60 = db.Column(db.Float, nullable=False)

par\_90 = db.Column(db.Float, nullable=False)

created\_at = db.Column(db.DateTime, server\_default=db.func.now())

class Arrear(db.Model):

\_\_tablename\_\_ = 'arrears'

id = db.Column(db.Integer, primary\_key=True)

loan\_id = db.Column(db.Integer, db.ForeignKey('loan\_applications.id'), nullable=False)

schedule\_id = db.Column(db.Integer, db.ForeignKey('repayment\_schedules.id'), nullable=True)

due\_date = db.Column(db.Date, nullable=False)

recorded\_at = db.Column(db.DateTime, default=datetime.utcnow, nullable=False)

expected\_principal = db.Column(db.Float, default=0.0)

expected\_interest = db.Column(db.Float, default=0.0)

expected\_fees = db.Column(db.Float, default=0.0)

paid\_principal = db.Column(db.Float, default=0.0)

paid\_interest = db.Column(db.Float, default=0.0)

paid\_fees = db.Column(db.Float, default=0.0)

payment\_status = db.Column(db.String(20), default='pending')

status = db.Column(db.String(20), default='unresolved') # unresolved, resolved

resolution\_date = db.Column(db.DateTime, nullable=True)

resolution\_type = db.Column(db.String(20), nullable=True) # 'payment', 'waiver', 'restructure'

resolved\_by = db.Column(db.Integer, db.ForeignKey('users.id'))

resolution\_notes = db.Column(db.Text, nullable=True)

tenure = db.Column(db.Integer)

probability\_of\_default = db.Column(db.Float, default=0.0)

loss\_given\_default = db.Column(db.Float, default=0.0)

provision\_amount = db.Column(db.Float, default=0.0)

arrear\_reason = db.Column(db.String(100), nullable=True)

action\_plan = db.Column(db.Text, nullable=True)

\_\_table\_args\_\_ = (

db.Index('ix\_loan\_schedule', 'loan\_id', 'schedule\_id'),

)

loan = db.relationship("LoanApplication", backref="arrears")

schedule = db.relationship("RepaymentSchedule", backref="arrears")

@property

def total\_due(self):

return self.expected\_principal + self.expected\_interest + self.expected\_fees

@property

def total\_paid(self):

return self.paid\_principal + self.paid\_interest + self.paid\_fees

@property

def total\_arrears(self):

return self.total\_due - self.total\_paid

@property

def is\_resolved(self):

return self.total\_arrears <= 0

@property

def customer(self):

return self.loan.customer if self.loan else None

@property

def is\_voluntary\_candidate(self):

return self.customer.is\_voluntary\_retirement\_candidate if self.customer else False

@property

def voluntary\_flag\_reason(self):

customer = self.customer

if not customer:

return None

flags = []

service\_years = customer.years\_in\_service

age = customer.age

if service\_years is not None and service\_years >= 20:

flags.append("eligible for voluntary retirement")

if age is not None and self.loan:

loan\_term\_years = self.loan.term\_months // 12

if age + loan\_term\_years >= 60:

flags.append("loan ends after retirement age")

return ", ".join(flags) if flags else None

@property

def voluntary\_flag\_reason(self):

customer = self.customer

if not customer:

return None

service\_years = calculate\_service\_years(customer.date\_joined)

age = calculate\_age(customer.dob)

retirement\_age = 60

age\_at\_loan\_end = age + (self.loan.term\_months // 12)

flags = []

if service\_years >= 20:

flags.append("eligible for voluntary retirement")

if age\_at\_loan\_end >= retirement\_age:

flags.append("loan exceeds retirement age")

return ", ".join(flags) if flags else None

@property

def days\_past\_due(self):

if self.due\_date:

delta = datetime.utcnow().date() - self.due\_date

return delta.days if delta.days > 0 else 0

return 0

@property

def aging(self):

days = self.days\_past\_due

if days <= 30:

return '1-30 days'

elif days <= 60:

return '31-60 days'

elif days <= 90:

return '61-90 days'

else:

return '90+ days'

class ProvisionSetting(db.Model):

\_\_tablename\_\_ = 'provision\_settings'

id = db.Column(db.Integer, primary\_key=True)

category = db.Column(db.String(50), nullable=False)

tenure = db.Column(db.Integer, nullable=False) # in months

probability\_of\_default = db.Column(db.Float, nullable=False)

loss\_given\_default = db.Column(db.Float, nullable=False)

created\_at = db.Column(db.DateTime, default=datetime.utcnow)

updated\_at = db.Column(db.DateTime, onupdate=datetime.utcnow)

\_\_table\_args\_\_ = (db.UniqueConstraint('category', 'tenure', name='uq\_category\_tenure'),)

from wtforms.validators import DataRequired, NumberRange

from datetime import datetime, timedelta, date

from decimal import Decimal, ROUND\_HALF\_UP

# Database Models (unchanged from your specification)

class RelationshipManager(db.Model):

id = db.Column(db.Integer, primary\_key=True)

name = db.Column(db.String(120), nullable=False)

phone = db.Column(db.String(20))

email = db.Column(db.String(120))

clients = db.relationship('Client', backref='relationship\_manager', lazy=True)

class Client(db.Model):

id = db.Column(db.Integer, primary\_key=True)

full\_name = db.Column(db.String(120), nullable=False)

national\_id = db.Column(db.String(20), unique=True, nullable=False)

phone = db.Column(db.String(20))

email = db.Column(db.String(120))

relationship\_manager\_id = db.Column(db.Integer, db.ForeignKey('relationship\_manager.id'))

placements = db.relationship('Placement', backref='client', lazy=True)

class Placement(db.Model):

id = db.Column(db.Integer, primary\_key=True)

client\_id = db.Column(db.Integer, db.ForeignKey('client.id'), nullable=False)

amount = db.Column(db.Float, nullable=False)

interest\_rate = db.Column(db.Float, nullable=False)

interest\_type = db.Column(db.String(20), default='Simple')

tenure\_months = db.Column(db.Integer, nullable=False)

start\_date = db.Column(db.Date, default=datetime.utcnow)

payment\_frequency = db.Column(db.String(20), default='Monthly')

commission\_percentage = db.Column(db.Float, default=0.0)

arrangement\_fee = db.Column(db.Float, default=0.0)

collateral = db.Column(db.String(255))

# New fields for enhanced functionality

current\_balance = db.Column(db.Float, default=0.0)

status = db.Column(db.String(20), default='Active') # Active, Partially Liquidated, Fully Liquidated

last\_interest\_calculation = db.Column(db.Date)

placement\_number = db.Column(db.String(20), unique=True, nullable=False)

interest\_payment\_frequency = db.Column(db.String(20), nullable=True)

payment\_frequency\_months = db.Column(db.Integer, nullable=True)

due\_date = db.Column(db.Date, nullable=False)

interest\_due = db.Column(db.Float, nullable=False, default=0.0)

principal\_due = db.Column(db.Float, default=0.0)

total\_due = db.Column(db.Float, nullable=False, default=0.0)

is\_paid = db.Column(db.Boolean, default=False)

schedules = db.relationship('PlacementSchedule', back\_populates='placement', cascade='all, delete-orphan')

def calculate\_daily\_interest(self, as\_of\_date=None):

if as\_of\_date is None:

as\_of\_date = date.today()

# Default to start\_date if last\_interest\_calculation is not set

from\_date = self.last\_interest\_calculation or self.start\_date

if not from\_date:

return 0.0

if self.current\_balance is None or self.interest\_rate is None:

return 0.0

days = (as\_of\_date - from\_date).days

daily\_rate = self.interest\_rate / 36500

return round(self.current\_balance \* daily\_rate \* days, 2)

def monthly\_interest(self):

if self.current\_balance is None or self.interest\_rate is None:

return 0.0

return self.current\_balance \* (self.interest\_rate / 1200)

def accrued\_interest(self):

try:

return self.calculate\_daily\_interest()

except Exception:

return 0.0

def add\_deposit(self, amount, deposit\_date=None):

"""Add additional funds to the placement"""

if not deposit\_date:

deposit\_date = datetime.utcnow().date()

# Capitalize accrued interest before deposit

self.capitalize\_interest(deposit\_date)

# Add deposit to balance

self.current\_balance += float(amount)

self.amount += float(amount) # Also update the original amount

self.last\_interest\_calculation = deposit\_date

# Record transaction

transaction = PlacementTransaction(

placement\_id=self.id,

transaction\_date=deposit\_date,

amount=amount,

transaction\_type='Deposit',

description=f"Additional deposit: ${amount:,.2f}"

)

db.session.add(transaction)

def withdraw\_funds(self, amount, withdrawal\_date=None):

"""Withdraw funds from the placement"""

if not withdrawal\_date:

withdrawal\_date = datetime.utcnow().date()

# Capitalize accrued interest before withdrawal

self.capitalize\_interest(withdrawal\_date)

# Check sufficient funds

if amount > self.current\_balance:

raise ValueError("Withdrawal amount exceeds available balance")

# Process withdrawal

self.current\_balance -= float(amount)

self.last\_interest\_calculation = withdrawal\_date

# Update status

if self.current\_balance <= 0.01: # Account for floating point precision

self.status = 'Fully Liquidated'

else:

self.status = 'Partially Liquidated'

# Record transaction

transaction = PlacementTransaction(

placement\_id=self.id,

transaction\_date=withdrawal\_date,

amount=-amount,

transaction\_type='Withdrawal',

description=f"Withdrawal: ${amount:,.2f}"

)

db.session.add(transaction)

return self.current\_balance

def change\_interest\_rate(self, new\_rate, effective\_date=None):

"""Change the interest rate for the placement"""

if not effective\_date:

effective\_date = datetime.utcnow().date()

# Capitalize accrued interest before rate change

self.capitalize\_interest(effective\_date)

# Update rate

self.interest\_rate = new\_rate

self.last\_interest\_calculation = effective\_date

# Record transaction

transaction = PlacementTransaction(

placement\_id=self.id,

transaction\_date=effective\_date,

amount=0,

transaction\_type='RateChange',

description=f"Rate changed to {new\_rate}%"

)

db.session.add(transaction)

def capitalize\_interest(self, as\_of\_date=None):

"""Capitalize accrued interest into principal"""

if not as\_of\_date:

as\_of\_date = datetime.utcnow().date()

# Calculate and capitalize interest

interest = self.calculate\_daily\_interest(as\_of\_date)

if interest > 0:

self.current\_balance += interest

self.last\_interest\_calculation = as\_of\_date

# Record transaction

transaction = PlacementTransaction(

placement\_id=self.id,

transaction\_date=as\_of\_date,

amount=interest,

transaction\_type='Interest',

description=f"Interest capitalization: ${interest:,.2f}"

)

db.session.add(transaction)

return interest

return 0.0

def liquidate(self, liquidation\_date=None):

"""Fully liquidate the placement"""

if not liquidation\_date:

liquidation\_date = datetime.utcnow().date()

# Capitalize final interest

self.capitalize\_interest(liquidation\_date)

# Withdraw remaining balance

final\_balance = self.current\_balance

self.withdraw\_funds(final\_balance, liquidation\_date)

self.status = 'Fully Liquidated'

return final\_balance

def accrued\_interest(self, as\_of\_date=None):

"""Calculate accrued interest not yet capitalized"""

if not as\_of\_date:

as\_of\_date = datetime.utcnow().date()

return self.calculate\_daily\_interest(as\_of\_date)

@property

def maturity\_date(self):

if self.start\_date and self.tenure\_months:

return self.start\_date + timedelta(days=30 \* self.tenure\_months)

return None

@property

def next\_interest\_date(self):

if self.last\_interest\_calculation:

return self.last\_interest\_calculation + timedelta(days=30)

return None

def change\_tenure(self, new\_tenure, effective\_date):

self.tenure\_months = new\_tenure

self.due\_date = self.start\_date + relativedelta(months=new\_tenure)

# Log transaction

transaction = PlacementTransaction(

placement\_id=self.id,

transaction\_type='Tenure Change',

amount=0.0,

description=f'Tenure changed to {new\_tenure} months, effective {effective\_date.strftime("%Y-%m-%d")}'

)

db.session.add(transaction)

db.session.commit()

# Optionally regenerate the schedule

new\_schedule = generate\_placement\_schedule(self)

if new\_schedule:

# Delete old unpaid schedules first

PlacementSchedule.query.filter\_by(placement\_id=self.id, is\_paid=False).delete()

db.session.bulk\_save\_objects(new\_schedule)

# New model for transaction history

class PlacementTransaction(db.Model):

id = db.Column(db.Integer, primary\_key=True)

placement\_id = db.Column(db.Integer, db.ForeignKey('placement.id'), nullable=False)

transaction\_date = db.Column(db.Date, default=datetime.utcnow)

amount = db.Column(db.Float, nullable=False)

transaction\_type = db.Column(db.String(20), nullable=False) # Deposit, Withdrawal, Interest, RateChange

description = db.Column(db.String(255))

class PlacementSchedule(db.Model):

id = db.Column(db.Integer, primary\_key=True)

placement\_id = db.Column(db.Integer, db.ForeignKey('placement.id'), nullable=True)

due\_date = db.Column(db.Date, nullable=False)

interest\_due = db.Column(db.Float, default=0.0)

principal\_due = db.Column(db.Float, default=0.0) # ← ADD THIS

total\_due = db.Column(db.Float, default=0.0)

is\_paid = db.Column(db.Boolean, default=False)

placement = db.relationship('Placement', back\_populates='schedules')

class PaymentAllocator:

NORMAL\_METHODS = {"normal", "bank\_transfer", "cash", "mobile\_money", "payroll\_deduction"}

def \_\_init\_\_(self, payment):

if not payment or not payment.loan:

raise ValueError("Payment must be associated with a loan")

# Normalize payment method

self.method = (payment.method or "normal").strip().lower().replace(" ", "\_")

if self.method in self.NORMAL\_METHODS:

self.method = "normal"

elif self.method == "internal\_topup":

self.method = "internal\_topup"

elif self.method == "settlement":

self.method = "settlement"

elif "top\_up" in self.method:

self.method = "top\_up"

else:

raise ValueError(f"Unsupported payment method: {payment.method}")

self.payment = payment

self.loan = payment.loan

self.amount = payment.amount

self.allocations = []

def process(self):

self.\_clear\_existing\_allocations()

if self.method == "normal":

self.\_allocate\_to\_schedules()

elif self.method == "internal\_topup":

self.\_apply\_direct\_principal\_only(target='top\_up')

elif self.method == "settlement":

self.\_apply\_direct\_principal\_only(target='settlement')

elif self.method == "top\_up":

self.\_allocate\_top\_up()

self.\_handle\_credit\_or\_arrears()

self.\_update\_loan\_status()

self.\_persist\_allocations()

def \_clear\_existing\_allocations(self):

PaymentAllocation.query.filter\_by(payment\_id=self.payment.id).delete()

def \_allocate\_to\_schedules(self):

schedules = sorted(

self.loan.repayment\_schedules,

key=lambda s: (s.due\_date, s.instalment\_no)

)

remaining = self.amount

for schedule in schedules:

if schedule.status == 'paid':

continue

# Get due amounts

principal\_due = schedule.principal\_due

interest\_due = schedule.interest\_due

fees\_due = schedule.fees\_due

# Allocation priority: fees → interest → principal

alloc\_fees = min(fees\_due, remaining)

schedule.paid\_fees += alloc\_fees

remaining -= alloc\_fees

alloc\_interest = min(interest\_due, remaining)

schedule.paid\_interest += alloc\_interest

remaining -= alloc\_interest

alloc\_principal = min(principal\_due, remaining)

schedule.paid\_principal += alloc\_principal

self.loan.current\_balance -= alloc\_principal

remaining -= alloc\_principal

self.allocations.append(PaymentAllocation(

payment\_id=self.payment.id,

schedule\_id=schedule.id,

principal=alloc\_principal,

interest=alloc\_interest,

fees=alloc\_fees,

created\_at=datetime.utcnow()

))

# Update schedule status

total\_paid = schedule.paid\_amount

if total\_paid >= schedule.expected\_amount:

schedule.status = 'paid'

elif total\_paid > 0:

schedule.status = 'partial'

else:

schedule.status = 'pending'

app.logger.info(

f"[{self.loan.loan\_number}] Schedule {schedule.id} → Paid: "

f"fees={alloc\_fees}, interest={alloc\_interest}, principal={alloc\_principal}"

)

if remaining <= 0:

break

self.amount = remaining

def \_apply\_direct\_principal\_only(self, target: str):

"""Handle internal top-ups as single transactions"""

# Get target balances

if target == 'top\_up':

principal\_balance = self.loan.top\_up\_balance

interest\_balance = self.loan.top\_up\_interest or 0.0

else: # settlement

principal\_balance = self.loan.current\_balance

interest\_balance = self.loan.settlement\_interest or 0.0

# Calculate payment distribution

principal\_paid = min(principal\_balance, self.amount)

remaining = self.amount - principal\_paid

interest\_paid = min(interest\_balance, remaining)

# Create SINGLE allocation record

self.allocations = [PaymentAllocation( # Use single-element list

payment\_id=self.payment.id,

principal=principal\_paid,

interest=interest\_paid if target == 'top\_up' else 0.0,

settlement\_interest=interest\_paid if target == 'settlement' else 0.0,

fees=0.0,

created\_at=datetime.utcnow()

)]

# Update loan balances

if target == 'top\_up':

self.loan.top\_up\_balance -= principal\_paid

self.loan.top\_up\_interest = max(0, interest\_balance - interest\_paid)

else:

self.loan.current\_balance -= principal\_paid

self.loan.settlement\_interest = max(0, interest\_balance - interest\_paid)

# Close loan if fully paid

if target == 'top\_up' and self.loan.top\_up\_balance <= 0 and self.loan.top\_up\_interest <= 0:

self.loan.status = 'closed'

self.loan.loan\_state = 'settled\_client'

self.loan.closure\_type = 'top\_up'

self.loan.closure\_date = datetime.utcnow()

self.amount = 0 # Full amount applied

def \_close\_loan(self, closure\_type):

self.loan.status = "closed"

self.loan.loan\_state = "settled\_client"

self.loan.closure\_type = closure\_type

self.loan.closure\_date = datetime.utcnow()

self.loan.current\_balance = 0.0

self.loan.top\_up\_balance = 0.0

self.loan.settlement\_balance = 0.0

# Cancel all unpaid schedules

for schedule in self.loan.repayment\_schedules:

if schedule.status not in {"paid", "cancelled"}:

schedule.status = "cancelled"

def \_close\_loan(self, closure\_type):

self.loan.status = "closed"

self.loan.loan\_state = "settled\_client"

self.loan.closure\_type = closure\_type

self.loan.closure\_date = datetime.utcnow()

# Cancel all unpaid schedules

for schedule in self.loan.repayment\_schedules:

if schedule.status not in {"paid", "cancelled"}:

schedule.status = "cancelled"

app.logger.info(f"[{self.loan.loan\_number}] Loan marked as closed after {closure\_type}.")

def \_handle\_credit\_or\_arrears(self):

if self.amount > 0:

if self.method in ['internal\_topup', 'top\_up']:

self.loan.top\_up\_interest = (self.loan.top\_up\_interest or 0.0) + round(self.amount, 2)

app.logger.info(f"[{self.loan.loan\_number}] Overpayment added to top-up interest: {self.amount}")

elif self.method == 'settlement':

self.loan.settlement\_interest = (self.loan.settlement\_interest or 0.0) + round(self.amount, 2)

app.logger.info(f"[{self.loan.loan\_number}] Overpayment added to settlement interest: {self.amount}")

else:

db.session.add(LoanCredit(

loan\_id=self.loan.id,

amount=round(self.amount, 2),

created\_at=datetime.utcnow()

))

app.logger.info(f"[{self.loan.loan\_number}] Overpayment recorded as loan credit: {self.amount}")

elif any(s.status != "paid" and s.due\_amount > 0 for s in self.loan.repayment\_schedules):

self.loan.record\_arrears(self.payment)

app.logger.info(f"[{self.loan.loan\_number}] Underpayment detected; arrears recorded.")

def \_update\_loan\_status(self):

self.loan.recalculate\_balance()

# Update performance status

ARREARS\_DELINQUENT = 30

ARREARS\_DEFAULT = 60

if self.loan.total\_arrears > ARREARS\_DEFAULT:

self.loan.performance\_status = 'default'

elif self.loan.total\_arrears > ARREARS\_DELINQUENT:

self.loan.performance\_status = 'delinquent'

else:

self.loan.performance\_status = 'performing'

def \_persist\_allocations(self):

for alloc in self.allocations:

db.session.add(alloc)

from flask\_wtf import FlaskForm

from wtforms import (

StringField, SubmitField, FloatField,

IntegerField, SelectField, DateField

)

from wtforms.validators import DataRequired, Email, NumberRange, Optional

from datetime import datetime

from wtforms.validators import DataRequired, NumberRange, Optional, Email

# ==============================

# Client Form

# ==============================

class ClientForm(FlaskForm):

full\_name = StringField("Full Name", validators=[DataRequired()])

national\_id = StringField("National ID", validators=[DataRequired()])

phone = StringField("Phone", validators=[Optional()])

email = StringField("Email", validators=[Optional(), Email()])

submit = SubmitField("Save Client")

# ==============================

# Placement Form

# ==============================

from flask\_wtf import FlaskForm

from wtforms.validators import DataRequired, NumberRange, Optional, Email

from wtforms import StringField, DecimalField, IntegerField, SelectField, DateField, FloatField, IntegerField, SubmitField

class PlacementForm(FlaskForm):

client\_id = SelectField("Client", coerce=int, validators=[DataRequired()])

amount = FloatField("Amount", validators=[DataRequired(), NumberRange(min=0.01)])

interest\_rate = FloatField("Interest Rate (%)", validators=[DataRequired(), NumberRange(min=0)])

interest\_type = SelectField(

"Interest Type",

choices=[("Simple", "Simple"), ("Compound", "Compound")],

validators=[DataRequired()]

)

tenure\_months = IntegerField("Tenure (Months)", validators=[DataRequired(), NumberRange(min=1)])

start\_date = DateField("Start Date", format="%Y-%m-%d", validators=[DataRequired()])

payment\_frequency\_months = IntegerField(

"Payment Frequency (Months)",

validators=[DataRequired()],

default=1

)

commission\_percentage = FloatField("Commission to RM (%)", default=0.0, validators=[Optional()])

arrangement\_fee = FloatField("Arrangement Fee", default=0.0, validators=[Optional()])

collateral = StringField("Collateral Info", validators=[Optional()])

submit = SubmitField("Create Placement")

# ==============================

# Deposit Form

# ==============================

class DepositForm(FlaskForm):

amount = FloatField("Amount", validators=[DataRequired(), NumberRange(min=0.01)])

transaction\_date = DateField("Transaction Date", default=datetime.utcnow, validators=[DataRequired()])

submit = SubmitField("Add Deposit")

# ==============================

# Withdrawal Form

# ==============================

class WithdrawalForm(FlaskForm):

amount = FloatField("Amount", validators=[DataRequired(), NumberRange(min=0.01)])

transaction\_date = DateField("Transaction Date", default=datetime.utcnow, validators=[DataRequired()])

submit = SubmitField("Process Withdrawal")

# ==============================

# Rate Change Form

# ==============================

class RateChangeForm(FlaskForm):

new\_rate = FloatField("New Interest Rate (%)", validators=[DataRequired(), NumberRange(min=0)])

effective\_date = DateField("Effective Date", default=datetime.utcnow, validators=[DataRequired()])

submit = SubmitField("Change Rate")

# ==============================

# Capitalize Interest Form

# ==============================

class CapitalizeForm(FlaskForm):

transaction\_date = DateField("Transaction Date", default=datetime.utcnow, validators=[DataRequired()])

submit = SubmitField("Capitalize Interest")

# ==============================

# Liquidate Placement Form

# ==============================

class LiquidateForm(FlaskForm):

transaction\_date = DateField("Transaction Date", default=datetime.utcnow, validators=[DataRequired()])

submit = SubmitField("Liquidate Placement")

from flask\_wtf import FlaskForm

from wtforms import (

SelectField,

DecimalField,

IntegerField,

DateField,

StringField,

SubmitField

)

from wtforms.validators import DataRequired, Optional, NumberRange

from datetime import date

class PlacementUpdateForm(FlaskForm):

amount = DecimalField("Amount (positive = deposit, negative = withdrawal)", places=2, validators=[Optional()])

new\_interest\_rate = DecimalField("New Interest Rate (%)", places=2, validators=[Optional()])

new\_tenure\_months = IntegerField("New Tenure (Months)", validators=[Optional()])

new\_payment\_frequency = IntegerField("New Payment Frequency (Months)", validators=[Optional()])

effective\_date = DateField("Effective Date", format="%Y-%m-%d", validators=[DataRequired()])

transaction\_date = DateField("Transaction Date", default=date.today, validators=[DataRequired()])

description = StringField("Description", validators=[Optional()])

def backfill\_schedule\_ids():

allocations = PaymentAllocation.query.filter\_by(schedule\_id=None).all()

print(f"Found {len(allocations)} allocations to backfill.")

for alloc in allocations:

payment = alloc.payment

loan = payment.loan if payment else None

if not loan or not payment:

print(f"Skipping allocation {alloc.id}: missing loan or payment.")

continue

# Find first unpaid or partially paid schedule

unpaid\_schedule = next((

s for s in sorted(loan.repayment\_schedules, key=lambda r: r.due\_date)

if s.paid\_amount < s.expected\_amount

), None)

if unpaid\_schedule:

alloc.schedule\_id = unpaid\_schedule.id

else:

print(f"No unpaid schedule found for allocation {alloc.id} (Loan ID {loan.id}).")

db.session.commit()

print("✅ Schedule ID backfill complete.")

# In LoanApplication model

def create\_loan():

try:

requested\_amount = float(request.form.get('requested\_amount'))

if requested\_amount <= 0:

raise ValueError("Amount must be positive")

except (ValueError, TypeError):

flash("Invalid requested amount", "danger")

return redirect(url\_for('loan\_application\_form'))

try:

term\_months = int(request.form.get('term\_months'))

category = request.form.get('category')

if not category:

flash("Category is required", "danger")

return redirect(url\_for('loan\_application\_form'))

except (ValueError, TypeError):

flash("Invalid term or category", "danger")

return redirect(url\_for('loan\_application\_form'))

# ✅ FIX: Use a dummy loan only for config lookup

dummy\_loan = LoanApplication(category=category, term\_months=term\_months)

config = get\_pricing\_config(category, term\_months, dummy\_loan)

if not config:

flash("Pricing configuration not found", "danger")

return redirect(url\_for('loan\_application\_form'))

# Fee calculations

orig\_fee = requested\_amount \* config['origination']

ins\_fee = requested\_amount \* config['insurance']

crb\_fee = config['crb']

capitalized\_fees = orig\_fee + ins\_fee + crb\_fee

loan\_amount = requested\_amount + capitalized\_fees

# ✅ Now create the actual loan safely

loan = LoanApplication(

loan\_amount=round(loan\_amount, 2),

term\_months=term\_months,

crb\_fees=round(crb\_fee, 2),

origination\_fees=round(orig\_fee, 2),

insurance\_fees=round(ins\_fee, 2),

category=category,

cash\_to\_client=round(requested\_amount, 2),

# Add: customer\_id, file\_number, etc. if required

)

db.session.add(loan)

db.session.commit()

# Optional: create disbursement

disbursement = Disbursement(

loan\_id=loan.id,

amount=requested\_amount,

method='bank',

status='pending',

reference=f"Initial disbursement for loan {loan.id}"

)

db.session.add(disbursement)

db.session.commit()

return loan

# Add this function

def generate\_repayment\_schedule(loan):

# Clear existing

for s in loan.repayment\_schedules:

db.session.delete(s)

config = get\_pricing\_config(loan.category, loan.term\_months, loan)

if not config:

raise ValueError(f"No pricing config for {loan.term\_months} months")

# Calculate components

loan\_amount = loan.loan\_amount

monthly\_rate = config['rate']

collection\_fee = loan\_amount \* config['collection']

# Capitalized amount

capitalized = (

loan\_amount

+ (loan\_amount \* config['origination'])

+ (loan\_amount \* config['insurance'])

+ config['crb']

)

# Calculate annuity payment

if monthly\_rate > 0 and loan.term\_months > 0:

annuity\_factor = (monthly\_rate \* (1 + monthly\_rate) \*\* loan.term\_months) / \

((1 + monthly\_rate) \*\* loan.term\_months - 1)

annuity\_payment = capitalized \* annuity\_factor

else:

annuity\_payment = capitalized / loan.term\_months

# Generate schedule

start\_date = loan.disbursement\_date or datetime.utcnow().date()

balance = capitalized

for i in range(1, loan.term\_months + 1):

due\_date = start\_date + relativedelta(months=i)

# Calculate components

interest = balance \* monthly\_rate

principal = annuity\_payment - interest

# Adjust last payment

if i == loan.term\_months:

principal = balance

annuity\_payment = principal + interest

# Update balance

balance -= principal

# Create schedule

schedule = RepaymentSchedule(

loan\_id=loan.id,

due\_date=due\_date,

expected\_principal=round(principal, 2),

expected\_interest=round(interest, 2),

expected\_fees=round(collection\_fee, 2),

expected\_amount=round(annuity\_payment + collection\_fee, 2),

status='pending'

)

db.session.add(schedule)

# Set total fees

loan.outstanding\_fees = collection\_fee \* loan.term\_months

def regenerate\_schedule(self):

PlacementSchedule.query.filter\_by(placement\_id=self.id).delete()

schedule = generate\_placement\_schedule(self)

db.session.bulk\_save\_objects(schedule)

# Update disbursement route

from flask\_wtf import FlaskForm

from wtforms import StringField, TextAreaField, BooleanField, SubmitField

from wtforms.validators import DataRequired, Length, Optional

class VoteForm(FlaskForm):

code = StringField('Vote Code', validators=[

DataRequired(message='Vote code is required'),

Length(min=2, max=10, message='Code must be between 2-10 characters')

])

description = TextAreaField('Description', validators=[

DataRequired(message='Description is required'),

Length(max=200, message='Description cannot exceed 200 characters')

])

is\_active = BooleanField('Active', default=True, validators=[Optional()])

submit = SubmitField('Save')

from sqlalchemy.orm import configure\_mappers

configure\_mappers()

def create\_roles\_and\_permissions():

# Create roles

roles = ['admin', 'loan\_officer', 'customer\_support']

for role\_name in roles:

if not Role.query.filter\_by(name=role\_name).first():

role = Role(name=role\_name)

db.session.add(role)

# Create permissions (example)

permissions = [

('customer', 'create'),

('loan', 'approve'),

]

for resource, action in permissions:

if not Permission.query.filter\_by(resource=resource, action=action).first():

perm = Permission(resource=resource, action=action)

db.session.add(perm)

db.session.commit()

import os

from flask.cli import with\_appcontext

import click

from app import db, User

from dotenv import load\_dotenv

import os

import click

from flask.cli import with\_appcontext

@click.command("create-admin")

@with\_appcontext

def create\_admin():

from dotenv import load\_dotenv

load\_dotenv() # Ensure latest values

email = os.getenv("ADMIN\_EMAIL")

password = os.getenv("ADMIN\_PASSWORD")

if not email or not password:

raise click.ClickException("ADMIN\_EMAIL and ADMIN\_PASSWORD must be set")

essential\_perms = [

('\*', '\*'), # Wildcard

('loan', 'create'),

('loan', 'approve'),

('loan', 'view'),

('user', 'manage'),

# Add other essential permissions

]

# 1. Ensure essential permissions exist

for res, action in essential\_perms:

perm = Permission.query.filter\_by(resource=res, action=action).first()

if not perm:

perm = Permission(resource=res, action=action)

db.session.add(perm)

# 2. Create admin role with all permissions

admin\_role = Role.query.filter\_by(name='admin').first()

if not admin\_role:

admin\_role = Role(name='admin')

db.session.add(admin\_role)

db.session.commit() # Need ID for relationship

# Assign all permissions to admin role

all\_perms = Permission.query.all()

for perm in all\_perms:

admin\_role.permissions.append(perm)

db.session.commit()

# 3. Create or update admin user

existing = User.query.filter\_by(email=email).first()

if existing:

# Update existing user to admin

existing.username = 'admin' # <-- Ensure username is set!

existing.role\_id = admin\_role.id

existing.set\_password(password)

db.session.commit()

click.echo(f"🛡️ User promoted to admin: {email} (username: admin)")

else:

# Create new admin

admin = User(

email=email,

username='admin', # <-- Set username here!

role\_id=admin\_role.id,

)

admin.set\_password(password)

db.session.add(admin)

db.session.commit()

click.echo(f"✅ Admin created: {email} (username: admin)")

def register\_cli\_commands(app):

app.cli.add\_command(create\_admin)

# Register CLI commands

register\_cli\_commands(app)

@property

def full\_name(self):

return f"{self.first\_name} {self.last\_name}"

@property

def total\_paid(self):

return sum(payment.amount for payment in self.payments)

@property

def balance(self):

return (self.loan\_amount or 0) - self.total\_paid

def allowed\_file(filename):

return '.' in filename and \

filename.rsplit('.', 1)[1].lower() in ALLOWED\_EXTENSIONS

# Function to save documents

def save\_document(file, customer\_id, doc\_type):

"""Save uploaded document to filesystem and return (filename, filepath)"""

if not file or file.filename == '':

return None, None

try:

# Create secure filename

filename = secure\_filename(file.filename)

unique\_filename = f"{customer\_id}\_{int(time.time())}\_{filename}"

# Get absolute upload path

upload\_folder = os.path.abspath(app.config['UPLOAD\_FOLDER'])

# Ensure directory exists

os.makedirs(upload\_folder, exist\_ok=True)

# Save file

filepath = os.path.join(upload\_folder, unique\_filename)

file.save(filepath)

# Return relative path for database storage

relative\_path = os.path.relpath(filepath, start=app.root\_path)

return unique\_filename, relative\_path

except Exception as e:

app.logger.error(f"Error saving document: {str(e)}")

return None, None

def send\_notification(message, type='info', recipient\_id=None):

notification = Notification(

message=message,

type=type,

recipient\_id=recipient\_id

)

db.session.add(notification)

db.session.commit()

from datetime import date

def notify\_client(client, message):

print(f"📲 SMS to {client.phone}: {message}")

# Use Twilio here

def notify\_rm(rm, message):

print(f"📧 Email to {rm.email}: {message}")

# Use Flask-Mail or SMTP

def calculate\_age(dob):

if not dob:

return 0

today = date.today()

return today.year - dob.year - ((today.month, today.day) < (dob.month, dob.day))

def calculate\_service\_years(joined\_date):

if not joined\_date:

return 0

today = date.today()

return today.year - joined\_date.year - ((today.month, today.day) < (joined\_date.month, joined\_date.day))

def get\_voluntary\_retirement\_loan\_alerts():

today = date.today()

return Arrear.query.join(LoanApplication).join(Customer).filter(

LoanApplication.loan\_state == 'active',

Customer.is\_voluntary\_retirement\_candidate == True,

Arrear.payment\_status != 'cleared', # Optional, based on how you define "arrears"

Arrear.total\_arrears > 0

).all()

@app.route('/alerts/voluntary-retirement')

def voluntary\_retirement\_alerts():

arrears = get\_voluntary\_retirement\_loan\_alerts()

results = [{

'customer': f"{a.customer.first\_name} {a.customer.last\_name}",

'file\_number': a.loan.file\_number,

'reason': a.voluntary\_flag\_reason,

'total\_arrears': a.total\_arrears,

} for a in arrears]

return jsonify(results)

# app.py (excerpt)

from flask import request, abort, render\_template

from flask\_login import login\_required, current\_user

@app.route('/admin/dashboard')

@login\_required

def admin\_dashboard():

if current\_user.role.name != 'admin':

abort(403)

# Moved to top: Get section parameter first with default value

section = request.args.get('section', 'users') # Default to 'users'

cutoff\_configs = {

c.category: c.cutoff\_dt

for c in CutoffDateConfig.query.all()

}

if section == 'cutoff\_dates':

civil\_config = CutoffDateConfig.query.filter\_by(category='civil\_servant').first()

private\_config = CutoffDateConfig.query.filter\_by(category='private\_sector').first()

cutoff\_configs = {

'civil\_servant': civil\_config.cutoff\_dt if civil\_config else '',

'private\_sector': private\_config.cutoff\_dt if private\_config else ''

}

return render\_template(

'admin\_dashboard.html',

section=section,

cutoff\_configs=cutoff\_configs,

users=[],

roles=[],

configs\_by\_category={},

categories=[],

terms=[]

)

if section == 'pricing':

# Create initial pricing configurations if none exist

if not PricingConfig.query.first():

create\_initial\_pricing\_configs()

# Get all configurations grouped by category

configs = PricingConfig.query.order\_by(

PricingConfig.category,

PricingConfig.term\_months

).all()

configs\_by\_category = {}

for config in configs:

if config.category not in configs\_by\_category:

configs\_by\_category[config.category] = []

configs\_by\_category[config.category].append(config)

return render\_template(

'admin\_dashboard.html',

section=section,

categories=['civil\_servant', 'private\_sector', 'sme'],

terms=[3, 6, 9, 12, 18, 24, 36, 48],

configs\_by\_category=configs\_by\_category,

users=[], # Added for template consistency

roles=[] # Added for template consistency

)

# Non-pricing section (default: 'users')

users = User.query.all()

roles = Role.query.all()

return render\_template(

'admin\_dashboard.html',

users=users,

roles=roles,

section=section,

cutoff\_configs=cutoff\_configs,

configs\_by\_category={},

categories=[],

terms=[]

)

# Route to view a document

from flask\_mail import Message

# Add to app.py

@app.route('/admin/permissions')

@role\_required("admin")

def admin\_permissions():

roles = Role.query.all()

pages = ['customer', 'loan', 'disbursement', 'payment', 'loanbook', 'admin']

actions = ['create', 'approve', 'edit', 'delete', 'view']

return render\_template('admin\_dashboard.html', section='permissions',

roles=roles, pages=pages, actions=actions)

@app.route('/update\_permissions', methods=['POST'])

@role\_required("admin")

def update\_permissions():

roles = Role.query.all()

pages = ['customer', 'loan', 'disbursement', 'payment', 'loanbook', 'admin']

actions = ['create', 'approve', 'edit', 'delete', 'view']

for role in roles:

for page in pages:

for action in actions:

perm\_name = f'perm\_{role.id}\_{page}\_{action}'

permission = Permission.query.filter\_by(resource=page, action=action).first()

if not permission:

continue

if perm\_name in request.form:

if permission not in role.permissions:

role.permissions.append(permission)

else:

if permission in role.permissions:

role.permissions.remove(permission)

db.session.commit()

flash('Permissions updated successfully', 'success')

return redirect(url\_for('admin\_permissions'))

def permission\_required(resource, action):

def decorator(f):

@wraps(f)

def decorated\_function(\*args, \*\*kwargs):

if not current\_user.role.has\_permission(resource, action):

abort(403)

return f(\*args, \*\*kwargs)

return decorated\_function

return decorator

def create\_initial\_roles():

roles = [

('admin', 'Full access'),

('credit\_officer', 'Credit operations'),

('finance\_officer', 'Financial management'),

('sales\_ops', 'Sales operations'),

('chief\_operations', 'Operations management'),

('chief\_finance', 'Financial oversight'),

('chief\_executive', 'Executive oversight')

]

actions = ['create', 'view', 'edit', 'delete', 'approve']

resources = ['customer', 'loan', 'disbursement', 'payment', 'loanbook', 'admin']

# Create permissions

for resource in resources:

for action in actions:

if not Permission.query.filter\_by(resource=resource, action=action).first():

perm = Permission(resource=resource, action=action)

db.session.add(perm)

# Create roles with default permissions

for role\_name, description in roles:

role = Role.query.filter\_by(name=role\_name).first()

if not role:

role = Role(name=role\_name)

db.session.add(role)

# Assign default permissions

if role\_name == 'admin':

perms = Permission.query.all()

else:

perms = Permission.query.filter(Permission.action == 'view').all()

role.permissions.extend(perms)

db.session.commit()

# Add to manage\_users route context

# ---------- Users Tab ----------

@app.route('/admin/users')

@role\_required("admin")

def admin\_users():

users = User.query.all()

roles = Role.query.all()

pages = ['customer', 'loan', 'disbursement', 'payment', 'loanbook', 'admin']

actions = ['create', 'approve', 'edit', 'delete', 'view']

return render\_template('admin\_dashboard.html', section='users',

users=users, roles=roles, pages=pages, actions=actions)

@app.route('/create\_user', methods=['POST'])

@role\_required("admin")

def create\_user():

username = request.form.get('username')

email = request.form.get('email')

password = request.form.get('password')

role\_id = request.form.get('role\_id')

if not all([username, email, password, role\_id]):

flash('All fields are required', 'danger')

return redirect(url\_for('admin\_users'))

if User.query.filter((User.username == username) | (User.email == email)).first():

flash('Username or email already exists', 'danger')

return redirect(url\_for('admin\_users'))

user = User(username=username, email=email, role\_id=role\_id, active=True)

user.set\_password(password)

db.session.add(user)

db.session.commit()

flash('User created successfully', 'success')

return redirect(url\_for('admin\_users'))

@app.route('/update\_user\_role', methods=['POST'])

@role\_required("admin")

def update\_user\_role():

user\_id = request.form.get('user\_id')

new\_role = request.form.get('role')

user = User.query.get(user\_id)

if user:

user.role\_id = new\_role

db.session.commit()

flash(f"{user.username}'s role updated", "success")

return redirect(url\_for('admin\_users'))

@app.route('/toggle\_user\_status', methods=['POST'])

@role\_required("admin")

def toggle\_user\_status():

user\_id = request.form.get('user\_id')

user = User.query.get(user\_id)

if user:

user.active = not user.active

db.session.commit()

flash(f"{user.username} is now {'active' if user.active else 'inactive'}", "info")

return redirect(url\_for('admin\_users'))

@app.route('/delete\_user', methods=['POST'])

@role\_required("admin")

def delete\_user():

user\_id = request.form.get('user\_id')

user = User.query.get(user\_id)

if user:

db.session.delete(user)

db.session.commit()

flash(f"User {user.username} deleted", "danger")

return redirect(url\_for('admin\_users'))

# ... existing imports ...

# Add this route to your app.py

@app.route('/admin/pricing', methods=['GET', 'POST'])

@login\_required

@role\_required('admin', 'finance\_officer')

def admin\_pricing():

# Create initial pricing configurations if none exist

if not PricingConfig.query.first():

create\_initial\_pricing\_configs()

if request.method == 'POST':

category = request.form['category']

term = int(request.form['term'])

config = PricingConfig.query.filter\_by(

category=category,

term\_months=term

).first()

if not config:

config = PricingConfig(

category=category,

term\_months=term

)

config.interest\_rate = float(request.form['interest\_rate'])

config.origination\_fee = float(request.form['origination\_fee'])

config.insurance\_fee = float(request.form['insurance\_fee'])

config.collection\_fee = float(request.form['collection\_fee'])

config.crb\_fee = float(request.form['crb\_fee'])

# Application scope fields

config.apply\_to\_new = 'apply\_to\_new' in request.form

config.apply\_to\_existing = 'apply\_to\_existing' in request.form

config.apply\_interest\_to\_existing = 'apply\_interest\_to\_existing' in request.form

config.apply\_collection\_to\_existing = 'apply\_collection\_to\_existing' in request.form

db.session.add(config)

db.session.commit()

# Apply changes to existing loans if requested

if config.apply\_to\_existing:

apply\_pricing\_to\_existing\_loans(config)

flash('Pricing configuration saved', 'success')

return redirect(url\_for('admin\_pricing'))

# GET request handling

configs = PricingConfig.query.order\_by(

PricingConfig.category,

PricingConfig.term\_months

).all()

configs\_by\_category = {}

for config in configs:

if config.category not in configs\_by\_category:

configs\_by\_category[config.category] = []

configs\_by\_category[config.category].append(config)

return render\_template(

'admin\_pricing.html',

categories=['civil\_servant', 'private\_sector', 'sme'],

terms=[3, 6, 9, 12, 18, 24, 36, 48],

configs\_by\_category=configs\_by\_category

)

def create\_initial\_pricing\_configs():

"""Create default pricing configurations if none exist"""

terms = [3, 6, 9, 12, 18, 24, 36, 48]

categories = ['civil\_servant', 'private\_sector', 'sme']

# Default values based on your PRICING dictionary

for category in categories:

for term in terms:

pricing\_data = PRICING.get(term, {

'rate': 0.035,

'origination': 0.15,

'insurance': 0.026,

'collection': 0.0025,

'crb': 3000

})

config = PricingConfig(

category=category,

term\_months=term,

interest\_rate=pricing\_data['rate'],

origination\_fee=pricing\_data['origination'],

insurance\_fee=pricing\_data['insurance'],

collection\_fee=pricing\_data['collection'],

crb\_fee=pricing\_data['crb']

)

db.session.add(config)

db.session.commit()

def apply\_pricing\_to\_existing\_loans(config):

"""Apply pricing changes to existing loans based on configuration"""

from sqlalchemy import and\_

query = LoanApplication.query.filter(

and\_(

LoanApplication.category == config.category,

LoanApplication.term\_months == config.term\_months,

LoanApplication.loan\_state == 'active'

)

)

for loan in query.all():

updated = False

# Update interest rate if requested

if config.apply\_interest\_to\_existing:

loan.applied\_interest\_rate = config.interest\_rate

updated = True

# Update collection fee if requested

if config.apply\_collection\_to\_existing:

loan.applied\_collection\_fee = config.collection\_fee

updated = True

# Regenerate schedule if any rate changed

if updated:

loan.pricing\_version += 1

loan.generate\_repayment\_schedule()

db.session.commit()

# Add this helper function to your utils

def get\_pricing\_config(category, term\_months, loan=None):

# For existing loans, use the applied rates if available

if loan and loan.applied\_interest\_rate is not None:

return {

'rate': loan.applied\_interest\_rate,

'origination': loan.origination\_fees / loan.loan\_amount if loan.loan\_amount else 0.15,

'insurance': loan.insurance\_fees / loan.loan\_amount if loan.loan\_amount else 0.026,

'collection': loan.applied\_collection\_fee if loan.applied\_collection\_fee is not None else 0.0025,

'crb': loan.crb\_fees if loan.crb\_fees is not None else 3000

}

# Look for active pricing config

config = PricingConfig.query.filter\_by(

category=category,

term\_months=term\_months,

apply\_to\_new=True

).order\_by(PricingConfig.updated\_at.desc()).first()

if config:

return {

'rate': config.interest\_rate,

'origination': config.origination\_fee,

'insurance': config.insurance\_fee,

'collection': config.collection\_fee,

'crb': config.crb\_fee

}

# Fallback to default PRICING

return PRICING.get(term\_months, {

'rate': 0.035,

'origination': 0.15,

'insurance': 0.026,

'collection': 0.0025,

'crb': 3000

})

from datetime import datetime

from sqlalchemy import func, extract

from flask\_login import current\_user

from sqlalchemy import and\_

@app.route('/admin/cutoff\_dates', methods=['POST'])

@login\_required

@role\_required('admin', 'finance\_officer')

def manage\_cutoff\_dates():

for category in ['civil\_servant', 'private\_sector']:

day = request.form.get(category)

if day:

config = CutoffDateConfig.query.filter\_by(category=category).first()

if config:

config.cutoff\_dt = int(day)

else:

config = CutoffDateConfig(category=category, cutoff\_dt=int(day))

db.session.add(config)

db.session.commit()

flash('Cutoff dates updated successfully.', 'success')

return redirect(url\_for('admin\_dashboard', section='cutoff\_dates'))

@app.route('/admin/update\_cutoff', methods=['POST'])

@login\_required

@role\_required('admin', 'finance\_officer')

def update\_single\_cutoff():

category = request.form.get('category')

cutoff\_raw = request.form.get('cutoff\_dt') # e.g. '2025-07-01T15:30'

try:

cutoff\_dt = datetime.strptime(cutoff\_raw, '%Y-%m-%dT%H:%M')

except ValueError:

flash('Invalid date-time format.', 'danger')

return redirect(url\_for('admin\_dashboard', section='cutoff\_dates'))

cfg = CutoffDateConfig.query.filter\_by(category=category).first()

if not cfg:

cfg = CutoffDateConfig(category=category)

db.session.add(cfg)

cfg.cutoff\_dt = cutoff\_dt # store full datetime

db.session.commit()

flash(f'{category.replace("\_", " ").title()} cut-off updated.', 'success')

return redirect(url\_for('admin\_dashboard', section='cutoff\_dates'))

@app.route('/admin/votes')

@login\_required

@role\_required('admin')

def manage\_votes():

votes = Vote.query.order\_by(Vote.code.asc()).all()

return render\_template('admin/votes.html', votes=votes)

@app.route('/admin/dashboard/votes', methods=['GET', 'POST'])

@app.route('/admin/dashboard/votes/<int:vote\_id>', methods=['GET', 'POST'])

def admin\_votes(vote\_id=None):

form = VoteForm()

edit\_mode = vote\_id is not None

if edit\_mode:

vote = Vote.query.get\_or\_404(vote\_id)

form = VoteForm(obj=vote)

if form.validate\_on\_submit():

if edit\_mode:

# Update existing vote

vote.code = form.code.data

vote.description = form.description.data

vote.is\_active = form.is\_active.data

db.session.commit()

flash('Vote updated successfully!', 'success')

else:

# Create new vote

vote = Vote(

code=form.code.data,

description=form.description.data,

is\_active=form.is\_active.data

)

db.session.add(vote)

db.session.commit()

flash('Vote added successfully!', 'success')

return redirect(url\_for('admin\_votes'))

votes = Vote.query.order\_by(Vote.code.asc()).all()

return render\_template('admin\_dashboard.html',

section='votes',

form=form,

votes=votes,

edit\_mode=edit\_mode,

vote\_id=vote\_id)

@app.route('/admin/votes/new', methods=['GET', 'POST'])

@login\_required

@role\_required('admin')

def new\_vote():

form = VoteForm()

if form.validate\_on\_submit():

vote = Vote(

code=form.code.data,

description=form.description.data,

is\_active=form.is\_active.data

)

db.session.add(vote)

db.session.commit()

flash('Vote added successfully!', 'success')

return redirect(url\_for('manage\_votes'))

return render\_template('admin/vote\_form.html', form=form, title='Add New Vote')

@app.route('/admin/votes/edit/<int:vote\_id>', methods=['GET', 'POST'])

@login\_required

@role\_required('admin')

def edit\_vote(vote\_id):

vote = Vote.query.get\_or\_404(vote\_id)

form = VoteForm(obj=vote)

if form.validate\_on\_submit():

vote.code = form.code.data

vote.description = form.description.data

vote.is\_active = form.is\_active.data

db.session.commit()

flash('Vote updated successfully!', 'success')

return redirect(url\_for('manage\_votes'))

return render\_template('admin/vote\_form.html', form=form, title='Edit Vote')

@app.route('/delete\_vote/<int:vote\_id>', methods=['POST'])

@login\_required

@role\_required('admin')

def delete\_vote(vote\_id):

vote = Vote.query.get\_or\_404(vote\_id)

# Check if vote is used in any loan applications

if LoanApplication.query.filter\_by(vote\_id=vote\_id).count() > 0:

flash('Cannot delete vote because it is used in loan applications!', 'danger')

return redirect(url\_for('manage\_votes'))

db.session.delete(vote)

db.session.commit()

flash('Vote deleted successfully!', 'success')

return redirect(url\_for('manage\_votes'))

@app.route('/admin/votes/merge', methods=['GET', 'POST'])

@login\_required

@role\_required('admin')

def merge\_votes():

if request.method == 'POST':

source\_id = request.form.get('source\_vote')

target\_id = request.form.get('target\_vote')

if not source\_id or not target\_id or source\_id == target\_id:

flash('Please select two different votes to merge', 'danger')

return redirect(url\_for('merge\_votes'))

source\_vote = Vote.query.get(source\_id)

target\_vote = Vote.query.get(target\_id)

if not source\_vote or not target\_vote:

flash('Invalid vote selection', 'danger')

return redirect(url\_for('merge\_votes'))

# Update all loan applications using the source vote

LoanApplication.query.filter\_by(vote\_id=source\_id).update({LoanApplication.vote\_id: target\_id})

# Delete the source vote

db.session.delete(source\_vote)

db.session.commit()

flash(f'Successfully merged "{source\_vote.code}" into "{target\_vote.code}"', 'success')

return redirect(url\_for('manage\_votes'))

votes = Vote.query.order\_by(Vote.code.asc()).all()

return render\_template('admin/merge\_votes.html', votes=votes)

@app.route('/admin/notifications')

@login\_required

@role\_required('admin')

def admin\_notifications():

# Create disbursement-related notifications

duplicate\_disbursements = (

db.session.query(

Customer.national\_id,

Customer.file\_number,

func.count(LoanApplication.id).label('disbursed\_count')

)

.join(LoanApplication, LoanApplication.customer\_id == Customer.id)

.filter(LoanApplication.disbursement\_date != None)

.group\_by(Customer.national\_id, Customer.file\_number)

.having(func.count(LoanApplication.id) > 1)

.all()

)

for nd in duplicate\_disbursements:

msg = f"Disbursement made more than once for National ID: {nd.national\_id} / File Number: {nd.file\_number} ({nd.disbursed\_count} times)"

exists = Notification.query.filter\_by(message=msg).first()

if not exists:

db.session.add(Notification(message=msg, recipient\_id=None, type='warning')) # Global to all admins

# Create duplicate payment notifications

duplicate\_payments = (

db.session.query(

LoanApplication.loan\_number,

extract('year', Payment.created\_at).label('year'),

extract('month', Payment.created\_at).label('month'),

func.count(Payment.id).label('payment\_count')

)

.join(Payment, Payment.loan\_id == LoanApplication.id)

.group\_by(LoanApplication.loan\_number, 'year', 'month')

.having(func.count(Payment.id) > 1)

.all()

)

for np in duplicate\_payments:

msg = f"Payment processed more than once for Loan Number: {np.loan\_number} in {int(np.month)}/{int(np.year)} ({np.payment\_count} times)"

exists = Notification.query.filter\_by(message=msg).first()

if not exists:

db.session.add(Notification(message=msg, recipient\_id=None, type='warning'))

db.session.commit()

# Show global (recipient\_id is NULL) + admin's own notifications (if needed)

notifications = Notification.query.filter(

Notification.recipient\_id == None # Only global

).order\_by(Notification.timestamp.desc()).all()

notifications = Notification.query.filter\_by(recipient\_id=None).order\_by(Notification.timestamp.desc()).all()

unread\_count = Notification.query.filter\_by(recipient\_id=None, is\_read=False).count()

return render\_template(

'admin\_dashboard.html',

section='notifications',

notifications=notifications,

unread\_count=unread\_count

)

@app.route('/admin/notifications/mark\_read/<int:notification\_id>')

@login\_required

@role\_required('admin')

def mark\_notification\_read(notification\_id):

notification = Notification.query.get\_or\_404(notification\_id)

if notification.recipient\_id is not None:

# Optional: ensure the right user is reading it

abort(403)

if not notification.is\_read:

notification.is\_read = True

db.session.commit()

return redirect(url\_for('admin\_notifications'))

def generate\_placement\_schedule(placement):

rate = placement.interest\_rate / 100

principal = placement.amount

frequency = placement.repayment\_frequency\_days

total\_days = placement.tenure\_months \* 30 # approx.

num\_payments = total\_days // frequency

for i in range(1, num\_payments + 1):

due\_date = placement.start\_date + timedelta(days=i \* frequency)

if placement.interest\_type == 'Simple':

interest = (principal \* rate \* frequency) / 365

else: # Compound

interest = principal \* ((1 + rate / 365) \*\* frequency - 1)

schedule = PlacementSchedule(

placement\_id=placement.id,

due\_date=due\_date,

interest\_due=round(interest, 2)

)

db.session.add(schedule)

db.session.commit()

@app.route('/admin/reports')

@role\_required("admin")

def admin\_reports():

return render\_template('admin\_dashboard.html', section='reports')

@app.route('/login', methods=['GET', 'POST'])

def login():

if current\_user.is\_authenticated:

return redirect(url\_for('home'))

if request.method == 'POST':

username = request.form.get('username')

password = request.form.get('password')

user = User.query.filter\_by(username=username).first()

if user and user.check\_password(password):

if not user.active:

flash('This account is deactivated', 'danger')

return redirect(url\_for('login'))

login\_user(user)

return redirect(url\_for('home'))

flash('Invalid username or password', 'danger')

return render\_template('login.html')

def notify\_ceo\_loan\_approved(customer\_name, loan\_amount):

ceo = User.query.filter\_by(role='ceo').first()

if ceo:

msg = Message("Loan Ready for CEO Approval", recipients=[ceo.email])

msg.body = f"The loan for customer {customer\_name} (amount: {loan\_amount}) has been approved by the CFO and awaits your review."

mail.send(msg)

@app.cli.command("init-rbac")

def init\_rbac():

"""Initialize roles and permissions"""

roles = [

('admin', 'Full access'),

('credit\_officer', 'Credit operations'),

('finance\_officer', 'Financial management'),

('sales\_ops', 'Sales operations'),

('chief\_operations', 'Operations management'),

('chief\_finance', 'Financial oversight'),

('chief\_executive', 'Executive oversight')

]

# Create roles

for name, description in roles:

if not Role.query.filter\_by(name=name).first():

role = Role(name=name)

db.session.add(role)

# Create permissions

resources = ['customer', 'loan', 'disbursement', 'payment', 'loanbook', 'admin']

actions = ['create', 'view', 'edit', 'delete', 'approve']

for resource in resources:

for action in actions:

if not Permission.query.filter\_by(resource=resource, action=action).first():

perm = Permission(resource=resource, action=action)

db.session.add(perm)

# Assign all permissions to admin

admin\_role = Role.query.filter\_by(name='admin').first()

admin\_role.permissions = Permission.query.all()

db.session.commit()

print("RBAC system initialized")

from sqlalchemy.orm import joinedload

from sqlalchemy.orm import joinedload

def calculate\_annuity\_payment(capitalized\_amount: float, term: int, rate: float) -> float:

"""Calculate fixed monthly annuity payment with safety checks."""

try:

if term <= 0 or rate <= 0 or capitalized\_amount <= 0:

return 0.0

# Handle potential overflow in exponential calculations

factor\_numerator = rate \* (1 + rate) \*\* term

factor\_denominator = (1 + rate) \*\* term - 1

# Prevent division by zero

if factor\_denominator < 1e-9: # Near-zero check

return capitalized\_amount / term # Fallback to simple division

return (capitalized\_amount \* factor\_numerator) / factor\_denominator

except (TypeError, ValueError, OverflowError):

return 0.0

def calculate\_capitalized\_amount(loan\_amount: float, config: dict) -> float:

"""Calculate capitalized amount from fees + CRB"""

try:

origination\_fee = loan\_amount \* config.get('origination', 0)

insurance\_fee = loan\_amount \* config.get('insurance', 0)

crb\_fee = config.get('crb', 0)

return round(loan\_amount + origination\_fee + insurance\_fee + crb\_fee, 2)

except Exception as e:

print(f"Capitalized amount error: {e}")

return loan\_amount

from datetime import datetime

def calculate\_balances(loan):

config = get\_pricing\_config(loan.category, loan.term\_months, loan)

if not config:

return {}

def calculate\_capitalized\_amount(loan\_amount, config):

try:

origination = loan\_amount \* config.get('origination', 0)

insurance = loan\_amount \* config.get('insurance', 0)

crb = config.get('crb', 0)

return round(loan\_amount + origination + insurance + crb, 2)

except Exception as e:

app.logger.warning(f"Capitalization error: {e}")

return loan\_amount

capitalized = calculate\_capitalized\_amount(loan.loan\_amount or 0, config)

monthly\_rate = config.get('rate', 0)

term = loan.term\_months or 0

if monthly\_rate > 0 and term > 0:

factor = (monthly\_rate \* (1 + monthly\_rate) \*\* term) / ((1 + monthly\_rate) \*\* term - 1)

annuity = capitalized \* factor

else:

annuity = 0

payments = sorted(loan.payments, key=lambda p: p.created\_at)

remaining\_balance = capitalized

payments\_made = 0

# Define valid statuses:

VALID\_PAYMENT\_STATUSES = ('successful', 'completed')

INVALID\_PAYMENT\_STATUSES = ('failed', 'reversed', 'cancelled')

for p in payments:

payment\_status = getattr(p, 'status', '').lower()

payment\_method = getattr(p, 'method', '').lower()

# Skip invalid payments

if payment\_status in INVALID\_PAYMENT\_STATUSES:

continue

# For computing remaining\_balance — count all valid principal allocations:

if p.allocation and p.allocation.principal:

remaining\_balance -= p.allocation.principal

remaining\_balance = max(remaining\_balance, 0)

payments\_made += 1

current\_balance = round(remaining\_balance, 2)

remaining\_term = max(term - payments\_made, 0)

# Projected interest

def projected\_interest(months\_ahead):

temp\_balance = current\_balance

total\_interest = 0.0

for \_ in range(min(months\_ahead, remaining\_term)):

if temp\_balance <= 0:

break

interest = temp\_balance \* monthly\_rate

principal = annuity - interest

principal = min(principal, temp\_balance)

total\_interest += interest

temp\_balance -= principal

return round(total\_interest, 2)

top\_up\_interest = projected\_interest(3)

settlement\_interest = projected\_interest(6)

return {

'capitalized\_amount': capitalized,

'current\_balance': current\_balance,

'top\_up\_balance': round(current\_balance + top\_up\_interest, 2),

'settlement\_balance': round(current\_balance + settlement\_interest, 2),

'top\_up\_interest': top\_up\_interest,

'settlement\_interest': settlement\_interest,

}

from flask import render\_template, request, redirect, url\_for, flash

from flask\_login import login\_required

from sqlalchemy.orm import joinedload

from datetime import datetime

@app.route('/customer/enquiry', methods=['POST'])

@login\_required

@role\_required('admin')

def customer\_enquiry():

query\_form = CustomerQueryForm()

if query\_form.validate\_on\_submit():

national\_id = query\_form.national\_id.data

employment\_number = query\_form.employment\_number.data

section = request.form.get('section', 'topup')

if not national\_id and not employment\_number:

flash('Please provide National ID or Employment Number', 'danger')

return redirect(url\_for('register\_customer'))

query = Customer.query

if national\_id:

query = query.filter(Customer.national\_id == national\_id)

if employment\_number:

query = query.filter(Customer.employment\_number == employment\_number)

customer = query.first()

if not customer:

flash('Customer not found', 'danger')

return redirect(url\_for('register\_customer'))

flash(f'Customer found: {customer.first\_name} {customer.last\_name}', 'success')

return redirect(url\_for('customer\_account', employment\_number=customer.employment\_number or 'PLACEHOLDER', section=section))

flash('Invalid input', 'danger')

return redirect(url\_for('register\_customer'))

@app.route('/customer/<file\_number>/account')

@login\_required

@role\_required('admin')

def customer\_account(file\_number: str):

try:

customer = Customer.query.filter\_by(file\_number=file\_number).first\_or\_404()

# Use Agent model instead of User model

agents = Agent.query.filter\_by(active=True).order\_by(Agent.name).all()

team\_leaders = Agent.query.filter\_by(role="Team Leader").all()

app.logger.info(f"Found {len(agents)} agents and {len(team\_leaders)} team leaders")

loans = (

LoanApplication.query

.options(joinedload(LoanApplication.payments).joinedload(Payment.allocations))

.filter(LoanApplication.customer\_id == customer.id)

.all()

)

app.logger.info(f"Found {len(loans)} loans for customer {file\_number}")

def calculate\_capitalized\_amount(loan\_amount: float, config: dict) -> float:

try:

origination = loan\_amount \* config.get('origination', 0)

insurance = loan\_amount \* config.get('insurance', 0)

crb = config.get('crb', 0)

return round(loan\_amount + origination + insurance + crb, 2)

except Exception as e:

app.logger.warning(f"Capitalization error: {e}")

return loan\_amount

def calculate\_balances(loan):

config = get\_pricing\_config(loan.category, loan.term\_months, loan)

if not config:

return {}

capitalized = calculate\_capitalized\_amount(loan.loan\_amount or 0, config)

monthly\_rate = config.get('rate', 0)

term = loan.term\_months or 0

if monthly\_rate > 0 and term > 0:

factor = (monthly\_rate \* (1 + monthly\_rate) \*\* term) / ((1 + monthly\_rate) \*\* term - 1)

annuity = capitalized \* factor

else:

annuity = 0

payments = sorted(loan.payments, key=lambda p: p.created\_at)

remaining\_balance = capitalized

payments\_made = 0

for p in payments:

for a in p.allocations:

if a.principal:

remaining\_balance -= a.principal

remaining\_balance = max(remaining\_balance, 0)

payments\_made += 1

current\_balance = round(remaining\_balance, 2)

remaining\_term = max(term - payments\_made, 0)

def projected\_interest(months\_ahead):

temp\_balance = current\_balance

total\_interest = 0.0

for \_ in range(min(months\_ahead, remaining\_term)):

if temp\_balance <= 0:

break

interest = temp\_balance \* monthly\_rate

principal = annuity - interest

principal = min(principal, temp\_balance)

total\_interest += interest

temp\_balance -= principal

return round(total\_interest, 2)

# For settled loans, balances = 0

if loan.status == 'closed' or loan.loan\_state == 'settled\_client':

return {

'capitalized\_amount': capitalized,

'current\_balance': 0.0,

'top\_up\_balance': 0.0,

'settlement\_balance': 0.0,

'top\_up\_interest': 0.0,

'settlement\_interest': 0.0,

}

return {

'capitalized\_amount': capitalized,

'current\_balance': current\_balance,

'top\_up\_balance': round(current\_balance + projected\_interest(3), 2),

'settlement\_balance': round(current\_balance + projected\_interest(6), 2),

'top\_up\_interest': projected\_interest(3),

'settlement\_interest': projected\_interest(6),

}

statement = []

for loan in loans:

balances = calculate\_balances(loan)

capitalized\_amount = balances.get('capitalized\_amount', 0.0)

running\_balance\_display = capitalized\_amount

# Build payment statement lines

payments\_sorted = sorted(loan.payments, key=lambda p: p.created\_at)

for payment in payments\_sorted:

for allocation in payment.allocations:

principal = allocation.principal or 0

interest = allocation.interest or 0

fees = allocation.fees or 0

# Apply principal reduction to running balance

running\_balance\_display -= principal

running\_balance\_display = max(running\_balance\_display, 0)

allocated\_total = principal + interest + fees

valid\_allocation = abs(allocated\_total - payment.amount) < 0.01

statement.append({

'id': payment.id,

'date': payment.created\_at.strftime('%Y-%m-%d'),

'total': payment.amount,

'principal': principal,

'interest': interest,

'collection\_fees': fees,

'remaining\_balance': round(running\_balance\_display, 2),

'method': payment.method,

'reference': payment.reference,

'valid\_allocation': valid\_allocation

})

# Attach balances to each loan for template display

loan.capitalized\_amount = balances.get('capitalized\_amount', 0.0)

loan.current\_balance = balances.get('current\_balance', 0.0)

loan.top\_up\_balance = balances.get('top\_up\_balance', 0.0)

loan.settlement\_balance = balances.get('settlement\_balance', 0.0)

loan.top\_up\_interest = balances.get('top\_up\_interest', 0.0)

loan.settlement\_interest = balances.get('settlement\_interest', 0.0)

if loan.loan\_state == 'active':

loan.cash\_to\_client = round(loan.loan\_amount - loan.top\_up\_balance, 2)

else:

loan.cash\_to\_client = loan.loan\_amount

return render\_template(

'customer\_account.html',

customer=customer,

loans=loans,

agents=agents, # Pass agents to template

team\_leaders=team\_leaders,

statement=statement,

section=request.args.get('section', 'statement')

)

except Exception as e:

app.logger.error(f"Account view error: {str(e)}")

flash("Error loading account details", "danger")

return redirect(url\_for('home'))

from datetime import date

@app.route('/loan/<loan\_number>/statement')

def loan\_statement(loan\_number):

try:

loan = (

LoanApplication.query

.options(

db.joinedload(LoanApplication.customer),

db.joinedload(LoanApplication.payments).joinedload(Payment.allocations)

)

.filter\_by(loan\_number=loan\_number)

.first\_or\_404()

)

config = get\_pricing\_config(loan.category, loan.term\_months, loan)

loan\_amount = loan.loan\_amount or 0

capitalized\_amount = (

loan\_amount +

(loan\_amount \* config.get('origination', 0)) +

(loan\_amount \* config.get('insurance', 0)) +

config.get('crb', 0)

)

monthly\_rate = config.get('rate', 0)

term = loan.term\_months or 0

# Compute current balance

if loan.status == 'closed' or loan.loan\_state == 'settled\_client':

current\_balance = 0.0

payments\_made = 0

else:

running\_balance = capitalized\_amount

payments\_made = 0

for payment in sorted(loan.payments, key=lambda p: p.created\_at):

for allocation in payment.allocations:

running\_balance -= allocation.principal or 0

running\_balance = max(running\_balance, 0)

payments\_made += 1

current\_balance = round(running\_balance, 2)

remaining\_term = max(term - payments\_made, 0)

# Calculate annuity

if monthly\_rate > 0 and term > 0:

annuity\_factor = (

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

) / ((1 + monthly\_rate) \*\* term - 1)

annuity\_payment = capitalized\_amount \* annuity\_factor

else:

annuity\_payment = 0

def calculate\_projected\_interest(months\_ahead):

temp\_balance = current\_balance

total\_interest = 0.0

for \_ in range(min(months\_ahead, remaining\_term)):

if temp\_balance <= 0:

break

interest = temp\_balance \* monthly\_rate

principal = annuity\_payment - interest

principal = min(principal, temp\_balance)

total\_interest += interest

temp\_balance -= principal

return total\_interest

if current\_balance > 0:

top\_up\_balance = round(current\_balance + calculate\_projected\_interest(3), 2)

settlement\_balance = round(current\_balance + calculate\_projected\_interest(6), 2)

else:

top\_up\_balance = settlement\_balance = 0.0

# Build statement

statement = []

running\_balance\_display = capitalized\_amount

for payment in sorted(loan.payments, key=lambda p: p.created\_at):

is\_top\_up = payment.method.lower() == 'top\_up'

for allocation in payment.allocations:

principal = allocation.principal or 0

interest = allocation.interest or 0

fees = allocation.fees or 0

# If this is a top-up payment, it clears the balance

if is\_top\_up:

running\_balance\_display = 0.0

else:

running\_balance\_display -= principal

running\_balance\_display = max(running\_balance\_display, 0)

allocated\_total = principal + interest + fees

valid\_allocation = abs(allocated\_total - payment.amount) < 0.01

statement.append({

'id': payment.id,

'date': payment.created\_at.strftime('%Y-%m-%d'),

'total': payment.amount,

'principal': principal,

'interest': interest,

'collection\_fees': fees,

'remaining\_balance': round(running\_balance\_display, 2),

'method': payment.method,

'reference': payment.reference,

'valid\_allocation': valid\_allocation

})

totals = {

'paid': sum(p.amount for p in loan.payments),

'principal': sum(a.principal or 0 for p in loan.payments for a in p.allocations),

'interest': sum(a.interest or 0 for p in loan.payments for a in p.allocations),

'fees': sum(a.fees or 0 for p in loan.payments for a in p.allocations)

}

return render\_template(

'loan\_statement.html',

loan=loan,

loan\_state=loan.loan\_state,

statement=statement,

capitalized\_amount=round(capitalized\_amount, 2),

current\_balance=current\_balance,

top\_up\_balance=top\_up\_balance,

settlement\_balance=settlement\_balance,

date=date,

totals=totals

)

except Exception as e:

flash(f"Error generating statement: {str(e)}", "danger")

return redirect(url\_for('loanbook'))

@app.route("/customer/debug-loans/<file\_number>")

def customer\_debug\_loans(file\_number):

customer = Customer.query.filter\_by(file\_number=file\_number).first\_or\_404()

loans = LoanApplication.query.filter\_by(customer\_id=customer.id).all()

return {

"customer": f"{customer.first\_name} {customer.last\_name}",

"loan\_count": len(loans),

"loans": [{ "id": l.id, "number": l.loan\_number, "amount": l.loan\_amount } for l in loans]

}

@app.route('/create-admin')

def create\_admin():

if not User.query.filter\_by(username='admin').first():

# 🔍 Fetch the actual Role object

admin\_role = Role.query.filter\_by(name='admin').first()

if not admin\_role:

flash("Admin role doesn't exist. Run `flask init-rbac` first.", "danger")

return redirect(url\_for('home'))

admin = User(

username='admin',

email='admin@example.com',

role=admin\_role, # ✅ Assign the Role object

active=True

)

admin.set\_password('admin123') # ✅ hash password

db.session.add(admin)

db.session.commit()

flash('Admin user created', 'success')

else:

flash('Admin user already exists', 'warning')

return redirect(url\_for('home'))

# Add these routes to your app.py

@app.cli.command("init-pricing")

def init\_pricing():

"""Create initial pricing configurations"""

terms = [3, 6, 9, 12, 18, 24, 36, 48]

categories = ['civil\_servant', 'private\_sector', 'sme']

for category in categories:

for term in terms:

pricing\_data = PRICING.get(term, {

'rate': 0.035,

'origination': 0.15,

'insurance': 0.026,

'collection': 0.0025,

'crb': 3000

})

if not PricingConfig.query.filter\_by(category=category, term\_months=term).first():

config = PricingConfig(

category=category,

term\_months=term,

interest\_rate=pricing\_data['rate'],

origination\_fee=pricing\_data['origination'],

insurance\_fee=pricing\_data['insurance'],

collection\_fee=pricing\_data['collection'],

crb\_fee=pricing\_data['crb']

)

db.session.add(config)

db.session.commit()

print("Initial pricing configurations created")

@app.route('/logout')

def logout():

logout\_user()

return redirect(url\_for('login'))

@app.route('/')

@login\_required

def home():

return render\_template('home.html')

@app.route('/document/<int:doc\_id>')

@login\_required

def view\_document(doc\_id):

"""Serve a document file with proper security checks"""

try:

# Retrieve document with existence check

doc = Document.query.get(doc\_id)

if not doc:

app.logger.warning(f"Document not found: {doc\_id}")

abort(404, description="Document not found")

# Get absolute path with fallbacks

abs\_path = doc.absolute\_path

if not abs\_path:

app.logger.error(f"Document path missing: {doc\_id}")

abort(404, description="Document path not configured")

# Verify file exists

if not os.path.exists(abs\_path):

app.logger.error(f"Document file missing: {abs\_path}")

abort(404, description="Document file not found")

# Determine MIME type

mime\_type, \_ = mimetypes.guess\_type(doc.filename)

if not mime\_type:

# Fallback for common types

if doc.filename.lower().endswith('.pdf'):

mime\_type = 'application/pdf'

elif doc.filename.lower().endswith(('.png', '.jpg', '.jpeg')):

mime\_type = 'image/' + doc.filename.split('.')[-1].lower()

else:

mime\_type = 'application/octet-stream'

# Send file with security headers

response = send\_file(

abs\_path,

mimetype=mime\_type,

as\_attachment=False,

conditional=True

)

# Security headers

response.headers['X-Content-Type-Options'] = 'nosniff'

response.headers['Content-Disposition'] = f'inline; filename="{doc.filename}"'

return response

except Exception as e:

app.logger.exception(f"Error serving document {doc\_id}: {str(e)}")

abort(500, description="Internal server error")

def convert\_legacy\_paths():

"""Convert any existing paths to the new relative format"""

docs = Document.query.all()

updated = 0

for doc in docs:

if doc.path and os.path.isabs(doc.path):

try:

doc.path = os.path.relpath(doc.path, start=app.root\_path)

updated += 1

except ValueError:

continue

if updated:

db.session.commit()

return updated

@app.route('/register', methods=['GET', 'POST'])

@role\_required("sales\_ops", "admin")

def register\_customer\_debug():

agents = Agent.query.filter\_by(active=True).order\_by(Agent.name).all()

team\_leaders = Agent.query.filter\_by(role="Team Leader").all()

if request.method == 'POST':

file = request.files.get('csv\_file')

try:

if file and file.filename.endswith('.csv'):

stream = io.StringIO(file.stream.read().decode("UTF8"), newline=None)

for row in csv.DictReader(stream):

process\_customer\_registration(row)

flash("✅ CSV upload processed successfully.", "success")

else:

process\_customer\_registration(request.form, files=request.files)

flash("✅ Customer and loan registered successfully.", "success")

except Exception as e:

db.session.rollback()

flash(f"❌ Error: {str(e)}", "danger")

return redirect(url\_for('register\_customer\_debug'))

return render\_template(

'register\_customer\_debug.html',

agents=agents,

team\_leaders=team\_leaders

)

def process\_customer\_registration(data, files=None):

try:

loan\_amount = max(float(data.get('loan\_amount', 0)), 0.0)

except (TypeError, ValueError):

raise Exception("Invalid loan amount.")

# Loan category setup

category\_code = int(data.get('loan\_category'))

CATEGORY\_MAP = {

1: {'prefix': '1', 'label': 'civil\_servant'},

2: {'prefix': '2', 'label': 'private\_sector'},

3: {'prefix': '3', 'label': 'sme'}

}

category\_info = CATEGORY\_MAP.get(category\_code)

if not category\_info:

raise Exception("Invalid loan category selected.")

term\_months = int(data.get('loan\_term', 0))

config = get\_pricing\_config(category\_info['label'], term\_months)

if not config:

raise Exception("Invalid loan term selected.")

# Validate date of birth

dob = datetime.strptime(data['dob'], "%Y-%m-%d").date()

today = date.today()

age = today.year - dob.year - ((today.month, today.day) < (dob.month, dob.day))

if age < 16:

raise Exception("Customer must be at least 16 years old.")

if age + (term\_months // 12) > 60:

raise Exception("Loan tenure will exceed retirement age (60 years).")

# Validate employment start date

date\_joined = data.get("date\_joined")

years\_in\_service = None

if date\_joined:

date\_joined = datetime.strptime(date\_joined, "%Y-%m-%d").date()

years\_in\_service = today.year - date\_joined.year - ((today.month, today.day) < (date\_joined.month, date\_joined.day))

if years\_in\_service >= 20:

flash("⚠️ Customer may be eligible for voluntary retirement.", "warning")

# Pricing fees

crb\_fees = 3000

origination\_fees = loan\_amount \* config['origination']

insurance\_fees = loan\_amount \* config['insurance']

collection\_fees = loan\_amount \* config['collection']

capitalized\_amount = loan\_amount + origination\_fees + insurance\_fees + crb\_fees

# Calculate monthly repayment

r = config['rate']

annuity = (r \* (1 + r) \*\* term\_months) / ((1 + r) \*\* term\_months - 1)

monthly\_payment = (capitalized\_amount \* annuity) + collection\_fees

# Check for duplicates

if Customer.query.filter\_by(email=data['email']).first():

raise Exception("Email already exists.")

if Customer.query.filter\_by(national\_id=data['national\_id']).first():

raise Exception("National ID already exists.")

# Generate customer file number

now = datetime.utcnow()

file\_number = f"{now.year}{now.month:02d}{db.session.query(Customer).count() + 1:06d}"

agent\_id = int(data['agent\_id']) if data.get('agent\_id') else None

# Create customer record

customer = Customer(

national\_id=data['national\_id'],

first\_name=data['first\_name'],

last\_name=data['last\_name'],

gender=data.get('gender'),

dob=dob,

date\_joined=date\_joined,

title=data.get('title'),

email=data['email'],

contact=data.get('contact'),

address=data.get('address'),

employer=data['employer'],

job\_title=data.get('job\_title'),

salary=float(data.get('salary') or 0),

bank\_name=data.get('bank\_name'),

bank\_account=data['bank\_account'],

salary\_deposited=data.get('salary\_deposited'),

district=data.get('district'),

region=data.get('region'),

amount\_requested=loan\_amount,

next\_of\_kin\_relationship=data.get("next\_of\_kin\_relationship"),

next\_of\_kin\_contact=data.get("next\_of\_kin\_contact"),

next\_of\_kin\_name=data.get('next\_of\_kin\_name'),

file\_number=file\_number,

status=data.get('status', 'pending'),

is\_approved\_for\_creation=False,

agent\_id=agent\_id,

maker\_id=current\_user.id

)

db.session.add(customer)

db.session.flush()

# Attach documents

document\_fields = {

'national\_id\_front': 'id\_front',

'form': 'form',

'customer\_photo': 'photo',

'payslip': 'payslip',

'bank\_statement': 'bank\_statement',

'letter\_of\_undertaking': 'undertaking\_letter'

}

for field, dtype in document\_fields.items():

if files and (file := files.get(field)) and file.filename:

filename, path = save\_document(file, customer.id, dtype)

db.session.add(Document(

customer\_id=customer.id,

filename=filename,

filetype=dtype,

path=path

))

# Loan number generation

loan\_number = f"{category\_info['prefix']}{str(term\_months).zfill(2)}{db.session.query(LoanApplication).count() + 1:06d}"

# Top-up balance if applicable

previous = LoanApplication.query.filter\_by(customer\_id=customer.id).order\_by(LoanApplication.id.desc()).first()

top\_up\_balance = calculate\_balances(previous).get('top\_up\_balance', 0) if previous else 0

cash\_to\_client = max(loan\_amount - top\_up\_balance, 0)

loan = LoanApplication(

customer\_id=customer.id,

loan\_amount=loan\_amount,

term\_months=term\_months,

monthly\_instalment=round(monthly\_payment, 2),

total\_repayment=round(monthly\_payment \* term\_months, 2),

effective\_rate=calculate\_eir(loan\_amount, term\_months, config),

category=category\_info['label'],

loan\_category=category\_code,

loan\_number=loan\_number,

file\_number=file\_number,

application\_status='pending',

loan\_state='application',

performance\_status='pending',

crb\_fees=crb\_fees,

agent\_id=current\_user.id,

origination\_fees=round(origination\_fees, 2),

insurance\_fees=round(insurance\_fees, 2),

collection\_fees=round(collection\_fees, 2),

cash\_to\_client=round(cash\_to\_client, 2),

applied\_interest\_rate=config['rate'],

applied\_collection\_fee=config['collection']

)

db.session.add(loan)

db.session.flush()

loan.generate\_repayment\_schedule()

db.session.add(Disbursement(

loan\_id=loan.id,

amount=cash\_to\_client,

method='bank',

status='pending',

reference=f"Initial disbursement for {loan.loan\_number}"

))

db.session.commit()

@app.route('/customers')

def customers():

approved\_customers = Customer.query.filter\_by(is\_approved\_for\_creation=True).all()

return render\_template('customers\_list.html', customers=approved\_customers)

@app.route('/approve\_customers', methods=['GET', 'POST'])

@login\_required

def approve\_customers():

if request.method == 'POST':

selected\_ids = request.form.getlist('customer\_ids')

if selected\_ids:

customers = Customer.query.filter(Customer.id.in\_(selected\_ids)).all()

approved\_count = 0

for customer in customers:

vote\_id = request.form.get(f'vote\_{customer.id}')

if not vote\_id:

flash(f"No vote selected for {customer.first\_name} {customer.last\_name}", "warning")

continue

vote = Vote.query.get(vote\_id)

if not vote:

flash(f"Invalid vote selected for {customer.first\_name} {customer.last\_name}", "danger")

continue

customer.is\_approved\_for\_creation = True

customer.checker\_id = current\_user.id

# Only create a loan if customer doesn't already have one

if not customer.loans:

loan = LoanApplication(

customer\_id=customer.id,

loan\_amount=customer.amount\_requested or 0.0,

loan\_category="SME", # You can make dynamic

status='pending',

loan\_state='Active',

application\_status='awaiting\_approval',

vote\_id=vote.id

)

db.session.add(loan)

db.session.flush()

# ✅ Link customer documents to this new loan

for doc in customer.customer\_documents:

doc.loan\_id = loan.id

approved\_count += 1

else:

for loan in customer.loans:

loan.status = 'pending'

loan.vote\_id = vote.id

for doc in customer.customer\_documents:

if not doc.loan\_id:

doc.loan\_id = loan.id

approved\_count += 1

db.session.commit()

flash(f"{approved\_count} customer(s) approved with vote assignments!", "success")

else:

flash("No customers selected.", "warning")

return redirect(url\_for('approve\_customers'))

# GET

unapproved\_customers = Customer.query \

.options(joinedload(Customer.customer\_documents)) \

.filter\_by(is\_approved\_for\_creation=False) \

.all()

active\_votes = Vote.query.filter\_by(is\_active=True).order\_by(Vote.code.asc()).all()

if not active\_votes:

active\_votes = Vote.query.order\_by(Vote.code.asc()).all()

return render\_template('approve\_customers.html',

customers=unapproved\_customers,

votes=active\_votes)

@app.route('/customer/<int:customer\_id>')

def view\_customer(customer\_id):

customer = Customer.query.get\_or\_404(customer\_id)

return render\_template('view\_customer.html', customer=customer)

@app.route('/customer/<int:customer\_id>/edit', methods=['GET', 'POST'])

@role\_required("admin")

def edit\_customer(customer\_id):

customer = Customer.query.get\_or\_404(customer\_id)

if request.method == 'POST':

customer.first\_name = request.form['first\_name']

customer.last\_name = request.form['last\_name']

customer.email = request.form['email']

customer.contact = request.form['contact']

customer.address = request.form['address']

customer.employer = request.form['employer']

customer.bank\_account = request.form['bank\_account']

customer.gender = request.form['gender']

customer.district = request.form['district']

customer.region = request.form['region']

db.session.commit()

flash('Customer details updated successfully!', 'success')

return redirect(url\_for('view\_customer', customer\_id=customer.id))

return render\_template('edit\_customer.html', customer=customer)

@app.route('/loans')

@login\_required

def view\_loans():

# MODIFIED: Eager load all necessary data

loans = db.session.query(LoanApplication, Customer) \

.join(Customer, LoanApplication.customer\_id == Customer.id) \

.options(

joinedload(LoanApplication.customer).load\_only(Customer.first\_name, Customer.last\_name),

joinedload(LoanApplication.documents),

joinedload(LoanApplication.customer).joinedload(Customer.customer\_documents)

) \

.filter(Customer.is\_approved\_for\_creation == True) \

.filter(LoanApplication.application\_status.in\_(['pending', 'approved'])) \

.all()

# Combine documents and deduplicate

processed\_loans = []

for loan, customer in loans:

# Combine all documents

all\_docs = list(loan.documents) + list(customer.customer\_documents)

# Deduplicate by filename

seen\_filenames = set()

unique\_docs = []

for doc in all\_docs:

if doc.filename not in seen\_filenames:

unique\_docs.append(doc)

seen\_filenames.add(doc.filename)

processed\_loans.append({

'loan': loan,

'customer': customer,

'unique\_docs': unique\_docs

})

return render\_template('view\_loans.html', loans=processed\_loans)

@app.route('/process\_loan/<int:loan\_id>/<action>', methods=['POST'])

def process\_loan(loan\_id, action):

loan = LoanApplication.query.get\_or\_404(loan\_id)

if action == 'approve':

loan.application\_status = 'approved'

loan.loan\_state = 'active'

flash('Loan approved successfully.', 'info')

elif action == 'reject':

loan.application\_status = 'rejected'

loan.loan\_state = None # or leave as-is

flash('Loan rejected successfully.', 'info')

db.session.commit()

return redirect(url\_for('view\_loans'))

@app.route('/loan-form/<int:customer\_id>', methods=['GET', 'POST'])

def loan\_form(customer\_id):

customer = Customer.query.get\_or\_404(customer\_id)

if not customer.is\_approved\_for\_creation:

flash("Customer not approved yet.", "danger")

return redirect(url\_for('customers'))

loan = LoanApplication.query.filter\_by(customer\_id=customer.id).first()

if not loan:

flash("Loan application not found.", "warning")

return redirect(url\_for('customers'))

if request.method == 'POST':

loan.amount = float(request.form.get('loan\_amount') or loan.amount)

loan.status = 'approved'

db.session.commit()

flash("Loan updated successfully.", "success")

return render\_template('loan\_form.html', customer=customer, loan=loan)

@app.route('/approve\_loans', methods=['GET', 'POST'])

def approve\_loans():

if request.method == 'POST':

loan\_ids = request.form.getlist('loan\_ids')

if loan\_ids:

loans = LoanApplication.query.filter(LoanApplication.id.in\_(loan\_ids)).all()

for loan in loans:

loan.application\_status = 'approved' # Correct field

db.session.commit()

flash(f'{len(loans)} loan(s) approved.', 'success')

else:

flash("No loans selected.", 'warning')

return redirect(url\_for('approve\_loans'))

loans = LoanApplication.query\

.join(Customer)\

.filter(Customer.is\_approved\_for\_creation == True)\

.filter(LoanApplication.status == 'pending')\

.all()

return render\_template('approve\_loans.html', loans=loans)

def generate\_loan\_and\_file\_number(category\_prefix: str, term\_months: int, customer\_count: int, loan\_count: int):

file\_sequence = str(customer\_count + 1).zfill(6)

loan\_sequence = str(loan\_count + 1).zfill(6)

now = datetime.utcnow()

file\_number = f"{now.year}{str(now.month).zfill(2)}{file\_sequence}"

loan\_number = f"{category\_prefix}{str(term\_months).zfill(2)}{loan\_sequence}"

return loan\_number, file\_number

@app.route('/customer/check/<int:customer\_id>', methods=['GET', 'POST'])

def check\_customer(customer\_id):

customer = Customer.query.get\_or\_404(customer\_id)

if request.method == 'POST':

customer.checker\_id = 2

customer.is\_approved\_for\_creation = True

if customer.loan:

customer.loan.status = 'approved'

db.session.commit()

flash("Customer and loan approved.", "success")

return redirect(url\_for('customers'))

return render\_template('check\_customer.html', customer=customer)

@app.route('/create\_existing\_loan', methods=['GET', 'POST'])

def create\_existing\_loan():

if request.method == 'POST':

try:

data = request.form

customer\_id = int(data['customer\_id'])

term\_months = int(data['loan\_term'])

loan\_amount = float(data['loan\_amount'])

category = data.get('category')

# Find the existing customer

customer = Customer.query.get(customer\_id)

if not customer:

flash("Customer not found.", "danger")

return redirect(url\_for('create\_existing\_loan'))

# ✅ Use a dummy loan for config lookup — DO NOT use 'loan' yet

dummy\_loan = LoanApplication(category=category, term\_months=term\_months)

config = get\_pricing\_config(category, term\_months, dummy\_loan)

if not config:

flash("Invalid loan term selected", "danger")

return redirect(url\_for('create\_existing\_loan'))

# Fee calculations

crb\_fees = 3000 # Fixed CRB fee

origination\_fees = loan\_amount \* config['origination']

insurance\_fees = loan\_amount \* config['insurance']

collection\_fees = loan\_amount \* config['collection']

capitalized\_amount = loan\_amount + origination\_fees + insurance\_fees + crb\_fees

annuity\_factor = (config['rate'] \* (1 + config['rate']) \*\* term\_months) / \

((1 + config['rate']) \*\* term\_months - 1)

monthly\_instalment = (capitalized\_amount \* annuity\_factor) + collection\_fees

# Generate loan + file number

loan\_number, file\_number = generate\_loan\_and\_file\_number(category, term\_months, db.session)

# ✅ Now define the actual loan

loan = LoanApplication(

customer\_id=customer.id,

loan\_amount=loan\_amount,

term\_months=term\_months,

monthly\_instalment=round(monthly\_instalment, 2),

total\_repayment=round(monthly\_instalment \* term\_months, 2),

effective\_rate=calculate\_eir(loan\_amount, term\_months, config),

category=category,

loan\_category=1, # Adjust if needed

status='pending',

crb\_fees=crb\_fees,

origination\_fees=round(origination\_fees, 2),

insurance\_fees=round(insurance\_fees, 2),

collection\_fees=round(collection\_fees, 2),

loan\_number=loan\_number,

file\_number=file\_number

)

db.session.add(loan)

db.session.commit()

flash("Loan created successfully for existing customer. Awaiting approval.", "success")

return redirect(url\_for('loanbook'))

except (KeyError, ValueError) as e:

db.session.rollback()

flash(f"Input error: {str(e)}", "danger")

except IntegrityError:

db.session.rollback()

flash("Duplicate entry detected. Check loan details.", "danger")

except Exception as e:

db.session.rollback()

flash(f"An unexpected error occurred: {e}", "danger")

return redirect(url\_for('create\_existing\_loan'))

return render\_template('create\_existing\_loan.html')

import logging

# -------- Disbursement Routes --------

@app.route('/disbursements', methods=['GET', 'POST'])

@login\_required

@role\_required("admin", "finance\_officer")

def disbursements():

from sqlalchemy.orm import joinedload

from datetime import datetime

if request.method == 'POST':

selected\_ids = request.form.getlist('loan\_ids[]')

selected\_bank = request.form.get('bank')

if not selected\_bank:

flash("Please select a bank.", "warning")

return redirect(url\_for('disbursements'))

if not selected\_ids:

flash("No loans selected for disbursement.", "warning")

return redirect(url\_for('disbursements'))

loans\_to\_process = LoanApplication.query \

.options(joinedload(LoanApplication.payments).joinedload(Payment.allocations)) \

.join(Customer) \

.filter(LoanApplication.id.in\_(selected\_ids)) \

.filter(LoanApplication.application\_status == 'approved') \

.filter(LoanApplication.disbursed == False) \

.filter(Customer.is\_approved\_for\_creation == True) \

.all()

for loan in loans\_to\_process:

try:

# Use precomputed value or fallback to loan\_amount

cash = loan.cash\_to\_client if loan.cash\_to\_client is not None else loan.loan\_amount

loan.cash\_to\_client = round(float(cash), 2)

loan.disbursed = True

loan.disbursed\_bank = selected\_bank

loan.disbursement\_date = datetime.utcnow().date()

if not loan.repayment\_schedule or len(loan.repayment\_schedule) == 0:

loan.generate\_repayment\_schedule()

db.session.add(loan)

disbursement = Disbursement(

loan\_id=loan.id,

amount=loan.cash\_to\_client,

method=selected\_bank.lower(),

status='successful',

reference=f"{loan.loan\_number}-{datetime.utcnow().strftime('%Y%m%d%H%M%S')}"

)

db.session.add(disbursement)

except Exception as e:

db.session.rollback()

app.logger.error(f"[DISBURSE] Error for loan {loan.loan\_number}: {e}")

flash(f"Error processing loan #{loan.loan\_number}", "danger")

db.session.commit()

flash("Disbursement complete.", "success")

try:

generate\_disbursement\_letter(loans\_to\_process, selected\_bank)

except Exception as e:

app.logger.error(f"[DISBURSEMENT LETTER] Failed to generate letter: {e}")

return redirect(url\_for('disbursements'))

# GET request

loans = db.session.query(LoanApplication) \

.join(Customer, LoanApplication.customer\_id == Customer.id) \

.options(

joinedload(LoanApplication.customer),

joinedload(LoanApplication.parent\_loan)

.joinedload(LoanApplication.payments)

.joinedload(Payment.allocations)

) \

.filter(LoanApplication.application\_status == 'approved') \

.filter(LoanApplication.disbursed == False) \

.filter(Customer.is\_approved\_for\_creation == True) \

.all()

# Use stored `cash\_to\_client` or fallback to loan\_amount

for loan in loans:

if loan.cash\_to\_client is None:

loan.cash\_to\_client = float(loan.loan\_amount)

app.logger.info(f"[DISBURSEMENT] Loan #{loan.loan\_number} | Top-up of: {loan.top\_up\_of} | Cash to Client: {loan.cash\_to\_client}")

return render\_template('disbursements.html', loans=loans, selected\_bank=request.form.get('bank'))

def calculate\_eir(principal, months, config, fees=None):

"""

Calculate Effective Interest Rate (EIR) with flexible input

Supports both regular loan creation (with config) and batch import (with fees)

"""

# Extract rate from config

rate = config.get('rate', 0)

# Calculate total fees based on input method

if fees is not None:

# Use directly provided fees (batch import)

total\_fees = fees

else:

# Calculate fees from config (regular loan creation)

origination\_rate = config.get('origination', 0)

insurance\_rate = config.get('insurance', 0)

crb\_fee = config.get('crb', 0)

origination\_fees = principal \* origination\_rate

insurance\_fees = principal \* insurance\_rate

total\_fees = origination\_fees + insurance\_fees + crb\_fee

# Calculate total interest

balance = principal

total\_interest = 0

total\_balances = 0

for \_ in range(months):

interest = balance \* rate

total\_interest += interest

total\_balances += balance

balance -= principal / months # Simple principal reduction

# Calculate EIR

average\_balance = total\_balances / months

eir = ((total\_interest + total\_fees) / average\_balance) \* (12 / months) \* 100

return round(eir, 2)

def generate\_disbursement\_letter(loans, bank\_name):

bank\_headers = {

"NBS Bank": {

"address": "P.O. Box 30322, Blantyre 3, Malawi",

"phone": "+265 1 822 488",

"email": "nbs@nbs.mw",

"attention": "The Branch Manager\nNBS Bank"

},

"National Bank": {

"address": "P.O. Box 945, Blantyre, Malawi",

"phone": "+265 1 820 622",

"email": "info@natbankmw.com",

"attention": "The Branch Manager\nNational Bank"

},

"Standard Bank": {

"address": "P.O. Box 30380, Blantyre 3, Malawi",

"phone": "+265 1 820 600",

"email": "info@standardbank.co.mw",

"attention": "The Branch Manager\nStandard Bank"

},

"First Capital Bank": {

"address": "P.O. Box 1111, Blantyre, Malawi",

"phone": "+265 1 822 123",

"email": "info@firstcapitalbank.co.mw",

"attention": "The Branch Manager\nFirst Capital Bank"

}

}

bank\_info = bank\_headers.get(bank\_name, {

"address": "Unknown Address",

"phone": "Unknown",

"email": "Unknown",

"attention": f"The Branch Manager\n{bank\_name}"

})

pdf = FPDF()

pdf.set\_auto\_page\_break(auto=True, margin=15)

pdf.add\_page()

pdf.set\_font("Arial", size=12)

# Add company logo with proper error handling

try:

image\_url = "https://i.ibb.co/fdhSBv37/Kwacha-Access-Header-Better.png"

pdf.image(image\_url, x=20, y=10, w=170) # Centered image with proper dimensions

pdf.ln(40) # Add space after the image

except Exception as e:

app.logger.error(f"Error inserting image: {str(e)}")

pdf.ln(20) # Add default space if image fails

# Bank details

pdf.set\_font("Arial", '', 12)

pdf.multi\_cell(0, 8,

f"""{bank\_info['attention']}

{bank\_info['address']}

Phone: {bank\_info['phone']}

Email: {bank\_info['email']}

Dear Sir/Madam,

SUBJECT: FUNDS TRANSFER INSTRUCTION ACCOUNT NUMBER: 24975600

Please find below the list of beneficiaries for funds transfer from our account:

""")

pdf.ln(8)

# Table with new columns

col\_widths = {

'name': 50, # 50mm

'bank': 30, # 30mm

'account': 40, # 40mm

'amount': 30 # 30mm (total 140mm will auto-wrap)

}

pdf.set\_font("Arial", 'B', 11)

pdf.cell(col\_widths['name'], 8, "Customer Name", border=1)

pdf.cell(col\_widths['bank'], 8, "Bank Name", border=1)

pdf.cell(col\_widths['account'], 8, "Account Number", border=1)

pdf.cell(col\_widths['amount'], 8, "Amount (MWK)", border=1, ln=True)

pdf.set\_font("Arial", size=11)

for loan in loans:

customer = loan.customer

pdf.cell(col\_widths['name'], 8, f"{customer.first\_name} {customer.last\_name}", border=1)

pdf.cell(col\_widths['bank'], 8, customer.bank\_name, border=1)

pdf.cell(col\_widths['account'], 8, customer.bank\_account, border=1)

pdf.cell(col\_widths['amount'], 8, f"{loan.cash\_to\_client:,.2f}", border=1, ln=True)

pdf.ln(10)

pdf.multi\_cell(0, 8, "Thank you for your continued support.\n\nYours faithfully,\n\nSlyvester Malumba\nCHIEF EXECUTIVE OFFICER")

pdf.set\_y(-30)

if pdf.page\_no() == 1:

pdf.set\_font("Arial", 'I', 8)

pdf.cell(0, 10, "Directors: Joe Kamalizeni, Margaret Munthali, Hariet Marian, Naomi Nyirenda, Grace Chipofya, Dr Damiano Kaufa, Dyson Mwadzera", 0, 0, 'C')

# Return PDF as download

pdf\_output = BytesIO()

pdf\_bytes = pdf.output(dest='S') # Keep as bytes without additional encoding

pdf\_output.write(pdf\_bytes)

pdf\_output.seek(0)

return send\_file(

pdf\_output,

as\_attachment=True,

download\_name='funds\_transfer\_instruction.pdf',

mimetype='application/pdf'

)

@app.route('/payments', methods=['GET', 'POST'], endpoint='payments')

@role\_required("finance\_officer", "admin")

def handle\_payments():

loan = None

if request.method == 'POST':

is\_batch = 'file' in request.files

try:

if is\_batch:

file = request.files['file']

if not file.filename.endswith('.csv'):

flash('Only CSV files are allowed', 'danger')

return redirect(url\_for('payments'))

stream = io.TextIOWrapper(file.stream, encoding='utf8')

csv\_reader = csv.DictReader(stream)

success = 0

errors = []

for row in csv\_reader:

try:

normalized\_row = {k.strip().lower(): v.strip() for k, v in row.items()}

loan\_number = normalized\_row.get('loan\_number', '').strip()

if not loan\_number:

errors.append(f"Missing loan\_number in row: {row}")

continue

amount\_str = normalized\_row.get('amount', '').replace(',', '')

try:

amount = float(amount\_str)

except ValueError:

errors.append(f"Invalid amount '{amount\_str}' for loan {loan\_number}")

continue

loan = LoanApplication.query.filter\_by(loan\_number=loan\_number).first()

if not loan:

errors.append(f"Loan {loan\_number} not found")

continue

payment = Payment(

loan\_id=loan.id,

amount=amount,

method=normalized\_row.get('method', 'Batch Upload'),

reference=normalized\_row.get('reference', '')

)

db.session.add(payment)

db.session.flush() # Ensure payment.id is available

PaymentAllocator(payment).process()

loan.recalculate\_balance()

success += 1

except Exception as e:

errors.append(f"Error processing row {row}: {str(e)}")

continue

db.session.commit()

flash(f"Processed {success} payments, {len(errors)} errors", 'info')

if errors:

flash('First 5 errors: ' + ' | '.join(errors[:5]), 'warning')

else:

# Single payment

loan\_number = request.form.get('loan\_number', '').strip().upper()

amount = float(request.form.get('amount'))

loan = LoanApplication.query.filter\_by(loan\_number=loan\_number).first()

if not loan:

flash(f'Loan {loan\_number} not found', 'danger')

return redirect(url\_for('payments'))

payment = Payment(

loan\_id=loan.id,

amount=amount,

method=request.form.get('method'),

reference=request.form.get('reference')

)

db.session.add(payment)

db.session.flush()

PaymentAllocator(payment).process()

loan.recalculate\_balance()

db.session.commit()

flash('Payment recorded successfully', 'success')

except Exception as e:

db.session.rollback()

app.logger.error(f"Payment error: {str(e)}", exc\_info=True)

flash(f'Payment failed: {str(e)}', 'danger')

return redirect(url\_for('payments'))

# GET method

loan\_number = request.args.get('loan\_number')

if loan\_number:

loan = LoanApplication.query.filter\_by(loan\_number=loan\_number).first()

return render\_template('payments.html', loan=loan)

@app.route('/payment/<int:payment\_id>/edit', methods=['GET', 'POST'])

@login\_required

@role\_required('finance\_officer', 'admin')

def edit\_payment(payment\_id):

payment = Payment.query.get\_or\_404(payment\_id)

loan = payment.loan

if request.method == 'POST':

try:

new\_amount = float(request.form['amount'])

if new\_amount <= 0:

flash("Amount must be positive", "danger")

return redirect(url\_for('edit\_payment', payment\_id=payment.id))

# Update amount

payment.amount = new\_amount

# Re-allocate using PaymentAllocator

db.session.flush()

PaymentAllocator(payment).process()

loan.recalculate\_balance()

db.session.commit()

flash("Payment updated successfully", "success")

return redirect(url\_for('loan\_statement', loan\_number=loan.loan\_number))

except ValueError:

flash("Invalid amount format", "danger")

except Exception as e:

db.session.rollback()

flash(f"Error updating payment: {str(e)}", "danger")

return render\_template('edit\_payment.html', payment=payment, loan=loan)

@app.route('/api/verify\_loan/<loan\_number>')

def verify\_loan(loan\_number):

loan = LoanApplication.query.filter\_by(loan\_number=loan\_number).first()

if not loan:

return jsonify({"error": "Loan not found"}), 404

return jsonify({

"loan\_number": loan.loan\_number,

"customer": f"{loan.customer.first\_name} {loan.customer.last\_name}",

"balance": loan.balance,

"monthly\_instalment": loan.monthly\_instalment

})

@app.route('/admin/delete\_test\_payments')

@role\_required("finance\_officer", "admin")

def delete\_test\_payments():

test\_payments = Payment.query.filter(Payment.reference.like("TEST%")).all()

count = len(test\_payments)

for p in test\_payments:

db.session.delete(p)

db.session.commit()

flash(f"Deleted {count} test payments", "success")

return redirect(url\_for("admin.dashboard"))

def update\_schedule\_status(loan):

schedules = sorted(loan.repayment\_schedule, key=lambda r: r.due\_date)

payments = sorted(loan.payments, key=lambda p: p.created\_at)

# Total allocations (from your allocation model)

payment\_map = [

{

'date': p.created\_at.date(),

'principal': p.allocation.principal if p.allocation else 0,

'interest': p.allocation.interest if p.allocation else 0,

'fees': p.allocation.fees if p.allocation else 0

}

for p in payments if p.allocation

]

for sched in schedules:

if sched.status == 'paid':

continue

# Expected total for this schedule

expected\_total = (sched.expected\_principal or 0) + (sched.expected\_interest or 0) + (sched.expected\_fees or 0)

# Try to find a payment to cover this

total\_paid = 0

for p in payment\_map:

p\_total = p['principal'] + p['interest'] + p['fees']

if p\_total <= 0:

continue

# Use it

total\_paid += p\_total

# Consume the allocation

p['principal'] = 0

p['interest'] = 0

p['fees'] = 0

if total\_paid >= expected\_total:

break

# Update schedule status

if total\_paid >= expected\_total:

sched.status = 'paid'

elif total\_paid > 0:

sched.status = 'partial'

else:

sched.status = 'due'

db.session.commit()

from flask import request, jsonify, render\_template\_string

from flask import request, jsonify, render\_template, flash, redirect, url\_for

from sqlalchemy.orm import joinedload

@app.route('/loanbook')

def loanbook():

try:

page = int(request.args.get("page", 1))

per\_page = 1000

ajax = request.args.get("ajax") == "true"

loans\_query = LoanApplication.query \

.join(Customer, LoanApplication.customer\_id == Customer.id) \

.options(joinedload(LoanApplication.payments).joinedload(Payment.allocations))

all\_loans = loans\_query.all()

def calculate\_capitalized\_amount(loan\_amount, config):

try:

origination = loan\_amount \* config.get('origination', 0)

insurance = loan\_amount \* config.get('insurance', 0)

crb = config.get('crb', 0)

return round(loan\_amount + origination + insurance + crb, 2)

except Exception:

return loan\_amount

def calculate\_balances(loan):

config = get\_pricing\_config(loan.category, loan.term\_months, loan)

if not config:

return {}

capitalized = calculate\_capitalized\_amount(loan.loan\_amount or 0, config)

monthly\_rate = config.get('rate', 0)

term = loan.term\_months or 0

if monthly\_rate > 0 and term > 0:

factor = (monthly\_rate \* (1 + monthly\_rate) \*\* term) / ((1 + monthly\_rate) \*\* term - 1)

annuity = capitalized \* factor

else:

annuity = 0

payments = sorted(loan.payments, key=lambda p: p.created\_at)

remaining\_balance = capitalized

payments\_made = 0

for p in payments:

for allocation in p.allocations:

if allocation.principal:

remaining\_balance -= allocation.principal

remaining\_balance = max(remaining\_balance, 0)

payments\_made += 1

current\_balance = round(remaining\_balance, 2)

remaining\_term = max(term - payments\_made, 0)

def projected\_interest(months\_ahead):

temp\_balance = current\_balance

total\_interest = 0.0

for \_ in range(min(months\_ahead, remaining\_term)):

if temp\_balance <= 0:

break

interest = temp\_balance \* monthly\_rate

principal = annuity - interest

principal = min(principal, temp\_balance)

total\_interest += interest

temp\_balance -= principal

return round(total\_interest, 2)

# For closed & settled loans, balances are zero

if loan.status == 'closed' and loan.loan\_state == 'settled\_client':

return {

'capitalized\_amount': capitalized,

'current\_balance': 0.0,

'top\_up\_balance': 0.0,

'settlement\_balance': 0.0,

'top\_up\_interest': 0.0,

'settlement\_interest': 0.0,

}

return {

'capitalized\_amount': capitalized,

'current\_balance': current\_balance,

'top\_up\_balance': round(current\_balance + projected\_interest(3), 2),

'settlement\_balance': round(current\_balance + projected\_interest(6), 2),

'top\_up\_interest': projected\_interest(3),

'settlement\_interest': projected\_interest(6),

}

processed\_loans = []

for loan in all\_loans:

customer = loan.customer

balances = calculate\_balances(loan)

processed\_loans.append({

'customer': {

'first\_name': customer.first\_name,

'last\_name': customer.last\_name,

'file\_number': customer.file\_number

},

'loan': {

'loan\_number': loan.loan\_number,

'amount': loan.loan\_amount or 0,

'term': loan.term\_months,

'category': loan.category,

'monthly\_instalment': loan.monthly\_instalment,

'total\_repayment': loan.total\_repayment,

'balance': balances.get('current\_balance', 0.0),

'disbursed': loan.disbursed,

'collection\_fee': (loan.loan\_amount or 0) \* get\_pricing\_config(loan.category, loan.term\_months, loan).get('collection', 0)

},

'fees': {

'crb': get\_pricing\_config(loan.category, loan.term\_months, loan).get('crb', 0),

'origination': (loan.loan\_amount or 0) \* get\_pricing\_config(loan.category, loan.term\_months, loan).get('origination', 0),

'insurance': (loan.loan\_amount or 0) \* get\_pricing\_config(loan.category, loan.term\_months, loan).get('insurance', 0),

'total': (

get\_pricing\_config(loan.category, loan.term\_months, loan).get('crb', 0)

+ (loan.loan\_amount or 0) \* get\_pricing\_config(loan.category, loan.term\_months, loan).get('origination', 0)

+ (loan.loan\_amount or 0) \* get\_pricing\_config(loan.category, loan.term\_months, loan).get('insurance', 0)

)

},

'balances': balances

})

# Pagination logic

total\_loans = len(processed\_loans)

start = (page - 1) \* per\_page

end = start + per\_page

paginated\_loans = processed\_loans[start:end]

has\_next = end < total\_loans

if ajax:

html = render\_template("partials/\_loan\_rows.html", loans=paginated\_loans)

ajax\_totals = {

"loan\_amount": sum(loan["loan"]["amount"] or 0 for loan in paginated\_loans),

"crb\_fees": sum(loan["fees"]["crb"] or 0 for loan in paginated\_loans),

"origination\_fees": sum(loan["fees"]["origination"] or 0 for loan in paginated\_loans),

"insurance\_fees": sum(loan["fees"]["insurance"] or 0 for loan in paginated\_loans),

"total\_fees": sum(loan["fees"]["total"] or 0 for loan in paginated\_loans),

"collection\_fees": sum(

loan["loan"]["collection\_fee"] \* (loan["loan"]["term"] or 0)

for loan in paginated\_loans

),

"total\_balance": sum(loan["loan"]["balance"] or 0 for loan in paginated\_loans),

"monthly\_instalment": sum(loan["loan"]["monthly\_instalment"] or 0 for loan in paginated\_loans),

"total\_repayment": sum(loan["loan"]["total\_repayment"] or 0 for loan in paginated\_loans)

}

return jsonify({

"html": html,

"has\_next": has\_next,

"totals": ajax\_totals

})

totals = {

"loan\_amount": sum(loan["loan"]["amount"] or 0 for loan in processed\_loans),

"crb\_fees": sum(loan["fees"]["crb"] or 0 for loan in processed\_loans),

"origination\_fees": sum(loan["fees"]["origination"] or 0 for loan in processed\_loans),

"insurance\_fees": sum(loan["fees"]["insurance"] or 0 for loan in processed\_loans),

"total\_fees": sum(loan["fees"]["total"] or 0 for loan in processed\_loans),

"collection\_fees": sum(

loan["loan"]["collection\_fee"] \* (loan["loan"]["term"] or 0)

for loan in processed\_loans

),

"total\_balance": sum(loan["loan"]["balance"] or 0 for loan in processed\_loans),

"monthly\_instalment": sum(loan["loan"]["monthly\_instalment"] or 0 for loan in processed\_loans),

"total\_repayment": sum(loan["loan"]["total\_repayment"] or 0 for loan in processed\_loans)

}

return render\_template(

'loanbook.html',

loans=paginated\_loans,

page=page,

has\_next=has\_next,

loan\_categories={loan['loan']['category'] for loan in processed\_loans if loan['loan']['category']},

loan\_tenures=sorted({loan['loan']['term'] for loan in processed\_loans if loan['loan']['term'] is not None}),

totals=totals

)

except Exception as e:

flash(f"Error loading loan book: {str(e)}", "danger")

return redirect(url\_for('home'))

def save\_file(file\_obj, subfolder=''):

if file\_obj:

filename = secure\_filename(file\_obj.filename)

upload\_dir = os.path.join(app.root\_path, 'uploads', subfolder)

os.makedirs(upload\_dir, exist\_ok=True)

file\_path = os.path.join(upload\_dir, filename)

file\_obj.save(file\_path)

return filename, file\_path

return None, None

# Updated process\_topup\_registration with automatic payment allocation to original loan

def process\_topup\_registration(data, base\_loan, loan\_form=None, bank\_payslip=None, live\_photo=None):

try:

new\_amount = float(data['amount\_requested'])

term\_months = int(data['tenure'])

except (ValueError, KeyError):

raise Exception("Invalid input for loan amount or tenure.")

category\_code = base\_loan.loan\_category

CATEGORY\_MAP = {

1: {'prefix': '1', 'label': 'civil\_servant'},

2: {'prefix': '2', 'label': 'private\_sector'},

3: {'prefix': '3', 'label': 'sme'}

}

category\_info = CATEGORY\_MAP.get(category\_code)

if not category\_info:

raise Exception("Unknown loan category.")

# Use base\_loan.category (label string) for pricing config lookup

config = get\_pricing\_config(base\_loan.category, term\_months)

if not config:

raise Exception("Pricing configuration unavailable.")

# --- Fees

crb\_fees = config.get('crb', 3000)

origination\_fees = new\_amount \* config.get('origination', 0)

insurance\_fees = new\_amount \* config.get('insurance', 0)

collection\_fees = new\_amount \* config.get('collection', 0)

# --- Top-up Balance Calculation

def calculate\_topup\_balance(loan):

loan\_config = get\_pricing\_config(loan.category, loan.term\_months)

def capitalized\_amount(amt, cfg):

return round(

amt +

(amt \* cfg.get('origination', 0)) +

(amt \* cfg.get('insurance', 0)) +

cfg.get('crb', 0),

2

)

capitalized = capitalized\_amount(loan.loan\_amount or 0, loan\_config)

rate = loan\_config.get('rate', 0)

term = loan.term\_months or 0

annuity = (capitalized \* rate \* (1 + rate) \*\* term) / ((1 + rate) \*\* term - 1) if rate > 0 and term > 0 else 0

payments = sorted(loan.payments, key=lambda p: p.created\_at)

paid\_principal = sum(

alloc.principal for p in payments for alloc in p.allocations if alloc.principal

)

current\_balance = round(capitalized - paid\_principal, 2)

payments\_made = sum(

1 for p in payments for alloc in p.allocations if alloc.principal

)

remaining\_term = max(term - payments\_made, 0)

interest\_total = 0.0

temp\_balance = current\_balance

for \_ in range(min(3, remaining\_term)):

if temp\_balance <= 0:

break

interest = temp\_balance \* rate

principal = min(annuity - interest, temp\_balance)

interest\_total += interest

temp\_balance -= principal

top\_up\_interest = round(interest\_total, 2)

return round(current\_balance + top\_up\_interest, 2), top\_up\_interest

top\_up\_balance, top\_up\_interest = calculate\_topup\_balance(base\_loan)

cash\_to\_client = new\_amount - top\_up\_balance

if cash\_to\_client <= 0:

raise Exception("Requested amount too low to cover top-up of previous balance.")

# --- Prepare New Loan

capitalized\_amount = new\_amount + origination\_fees + insurance\_fees + crb\_fees

annuity\_factor = (config['rate'] \* (1 + config['rate']) \*\* term\_months) / \

((1 + config['rate']) \*\* term\_months - 1)

monthly\_instalment = (capitalized\_amount \* annuity\_factor) + collection\_fees

now = datetime.utcnow()

loan\_sequence = str(db.session.query(LoanApplication).count() + 1).zfill(6)

loan\_number = f"{category\_info['prefix']}{str(term\_months).zfill(2)}{loan\_sequence}"

agent\_id = int(data['agent\_id']) if data.get('agent\_id') else None

# --- Create New Top-up Loan

topup\_loan = LoanApplication(

customer\_id=base\_loan.customer\_id,

loan\_amount=new\_amount,

term\_months=term\_months,

monthly\_instalment=round(monthly\_instalment, 2),

total\_repayment=round(monthly\_instalment \* term\_months, 2),

effective\_rate=calculate\_eir(new\_amount, term\_months, config),

category=category\_info['label'],

loan\_category=category\_code,

loan\_number=loan\_number,

file\_number=base\_loan.file\_number,

agent\_id=agent\_id,

application\_status='pending',

loan\_state='application',

performance\_status='pending',

crb\_fees=crb\_fees,

origination\_fees=round(origination\_fees, 2),

insurance\_fees=round(insurance\_fees, 2),

collection\_fees=round(collection\_fees, 2),

top\_up\_of=base\_loan.id,

top\_up\_balance=top\_up\_balance,

top\_up\_interest=top\_up\_interest,

cash\_to\_client=round(cash\_to\_client, 2),

applied\_interest\_rate=config['rate'],

applied\_collection\_fee=config['collection'],

date\_created=now

)

db.session.add(topup\_loan)

db.session.flush()

# --- Disbursement Record

disbursement = Disbursement(

loan\_id=topup\_loan.id,

amount=cash\_to\_client,

method='bank',

status='pending',

reference=f"Top-up disbursement for {loan\_number}"

)

db.session.add(disbursement)

# --- Document Uploads

for file\_obj, filetype in [

(loan\_form, 'loan\_form'),

(bank\_payslip, 'bank\_payslip'),

(live\_photo, 'live\_photo')

]:

if file\_obj:

filename, filepath = save\_file(file\_obj, subfolder=f"topup\_loan\_{topup\_loan.id}")

if filename:

db.session.add(Document(

customer\_id=base\_loan.customer\_id,

loan\_id=topup\_loan.id,

filename=filename,

filetype=filetype,

path=filepath,

uploaded\_at=datetime.utcnow()

))

# --- Generate Repayment Schedule

topup\_loan.generate\_repayment\_schedule()

# --- Pay Off Base Loan

if top\_up\_balance > 0:

# --- Create payment for top-up balance ---

payment = Payment(

loan\_id=base\_loan.id,

amount=top\_up\_balance,

method='top\_up',

status='successful',

reference=f"Top-Up from {loan\_number}",

created\_at=datetime.utcnow()

)

db.session.add(payment)

db.session.flush()

# Allocate payment while loan is still open

base\_loan.allocate\_payment(payment)

# Commit so allocations are persisted

db.session.commit()

# Now close the base loan ONLY if fully paid

if base\_loan.top\_up\_balance <= 0 and (base\_loan.top\_up\_interest or 0) <= 0:

base\_loan.status = 'closed'

base\_loan.loan\_state = 'settled\_client'

base\_loan.closure\_type = 'topup'

base\_loan.closure\_date = datetime.utcnow()

# Cancel all remaining repayment schedules

for schedule in base\_loan.repayment\_schedules:

if schedule.status not in {"paid", "cancelled"}:

schedule.status = "cancelled"

db.session.commit()

@app.route('/loans', methods=['POST'])

def create\_loan():

data = request.get\_json()

try:

new\_loan = LoanApplication(

customer\_id=data.get('customer\_id'),

loan\_amount=data.get('amount'),

parent\_loan\_id=data.get('parent\_loan\_id') # optional

)

db.session.add(new\_loan)

db.session.commit()

return jsonify({'id': new\_loan.id, 'message': 'Loan created'}), 201

except Exception as e:

db.session.rollback()

return jsonify({'error': str(e)}), 500

def allowed\_file(filename: str) -> bool:

return '.' in filename and \

filename.rsplit('.', 1)[1].lower() in {'pdf', 'png', 'jpg', 'jpeg'}

@app.route('/topup/<int:loan\_id>', methods=['POST'])

@role\_required('sales\_ops', 'admin')

def submit\_topup(loan\_id):

app.logger.info(f"[TOPUP] Submit route hit for loan {loan\_id}")

base\_loan = LoanApplication.query.get\_or\_404(loan\_id)

data = request.form

files = request.files

try:

new\_loan = process\_topup\_registration(

data=data,

base\_loan=base\_loan,

loan\_form=files.get('loan\_form'),

bank\_payslip=files.get('bank\_payslip'),

live\_photo=files.get('live\_photo')

)

# cash\_to\_client = loan\_amount - old loan balance (calculated inside process\_topup\_registration)

db.session.commit()

flash("Top-up loan submitted successfully.", "success")

return redirect(url\_for('customer\_account', file\_number=base\_loan.file\_number, section='statement'))

except Exception as e:

db.session.rollback()

flash(f"Failed to submit top-up: {str(e)}", "danger")

return redirect(request.referrer)

from datetime import datetime, timezone

def process\_additional\_registration(data, base\_loan, new\_category=None,

loan\_form=None, bank\_payslip=None, live\_photo=None):

try:

new\_amount = float(data['amount\_requested'])

term\_months = int(data['tenure'])

except (ValueError, KeyError):

raise ValueError("Invalid loan amount or tenure input.")

CATEGORY\_MAP = {

1: ('1', 'civil\_servant'),

2: ('2', 'private\_sector'),

3: ('3', 'sme')

}

# Use the new\_category if provided; else fallback to base\_loan.loan\_category

if new\_category:

# Find prefix and category\_label from CATEGORY\_MAP values by matching new\_category

# CATEGORY\_MAP values are tuples like ('1', 'civil\_servant'), so find matching label

category\_info = next(((prefix, label) for prefix, label in CATEGORY\_MAP.values() if label == new\_category), None)

if not category\_info:

raise ValueError("Unknown new loan category.")

else:

category\_info = CATEGORY\_MAP.get(base\_loan.loan\_category)

if not category\_info:

raise ValueError("Unknown base loan category.")

prefix, category\_label = category\_info

config = get\_pricing\_config(category\_label, term\_months)

if not config:

raise ValueError("Pricing configuration unavailable.")

crb\_fees = config.get('crb', 3000)

origination\_fees = new\_amount \* config.get('origination', 0)

insurance\_fees = new\_amount \* config.get('insurance', 0)

collection\_fees = new\_amount \* config.get('collection', 0)

capitalized\_amount = new\_amount + origination\_fees + insurance\_fees + crb\_fees

rate = config['rate']

annuity\_factor = (rate \* (1 + rate) \*\* term\_months) / ((1 + rate) \*\* term\_months - 1)

monthly\_instalment = capitalized\_amount \* annuity\_factor + collection\_fees

now\_utc = datetime.now(timezone.utc)

loan\_sequence = str(db.session.query(LoanApplication).count() + 1).zfill(6)

loan\_number = f"{prefix}{str(term\_months).zfill(2)}{loan\_sequence}"

agent\_id = int(data['agent\_id']) if data.get('agent\_id') else None

additional\_loan = LoanApplication(

customer\_id=base\_loan.customer\_id,

loan\_amount=new\_amount,

term\_months=term\_months,

monthly\_instalment=round(monthly\_instalment, 2),

total\_repayment=round(monthly\_instalment \* term\_months, 2),

effective\_rate=calculate\_eir(new\_amount, term\_months, config),

category=category\_label,

loan\_category=int(prefix), # keep the numeric loan\_category consistent

loan\_number=loan\_number,

file\_number=base\_loan.file\_number,

agent\_id=agent\_id,

application\_status='pending',

loan\_state='active',

crb\_fees=crb\_fees,

origination\_fees=round(origination\_fees, 2),

insurance\_fees=round(insurance\_fees, 2),

collection\_fees=round(collection\_fees, 2),

parent\_loan\_id=None,

date\_created=now\_utc,

disbursement\_date=None,

cash\_to\_client=new\_amount,

applied\_interest\_rate=rate,

applied\_collection\_fee=config.get('collection', 0),

)

db.session.add(additional\_loan)

db.session.flush()

for file\_obj, filetype in [

(loan\_form, 'loan\_form'),

(bank\_payslip, 'bank\_payslip'),

(live\_photo, 'live\_photo')

]:

if file\_obj:

filename, filepath = save\_file(file\_obj, subfolder=f"additional\_loan\_{additional\_loan.id}")

if filename:

doc = Document(

customer\_id=base\_loan.customer\_id,

loan\_id=additional\_loan.id,

filename=filename,

filetype=filetype,

path=filepath,

uploaded\_at=now\_utc

)

db.session.add(doc)

additional\_loan.generate\_repayment\_schedule()

return additional\_loan

@app.route('/apply\_additional/<int:loan\_id>', methods=['POST'])

@role\_required('sales\_ops', 'admin')

def apply\_additional\_loan(loan\_id):

base\_loan = LoanApplication.query.get\_or\_404(loan\_id)

data = request.form

files = request.files

try:

# Similar logic to process\_topup\_registration, or you can reuse that function with a different flag

new\_loan = process\_additional\_registration(

data=data,

base\_loan=base\_loan,

loan\_form=files.get('loan\_form'),

bank\_payslip=files.get('bank\_payslip'),

live\_photo=files.get('live\_photo')

)

db.session.commit()

flash("Additional loan submitted successfully.", "success")

return redirect(url\_for('customer\_account', file\_number=base\_loan.file\_number, section='statement'))

except Exception as e:

db.session.rollback()

flash(f"Failed to submit additional loan: {str(e)}", "danger")

return redirect(request.referrer)

@app.route('/debug\_loans')

def debug\_loans():

loans = LoanApplication.query.all()

output = []

for loan in loans:

output.append({

'Loan ID': loan.id,

'Loan Status': loan.status,

'Customer ID': loan.customer\_id,

'Customer Approved': loan.customer.is\_approved\_for\_creation,

'Customer Name': f"{loan.customer.first\_name} {loan.customer.last\_name}"

})

return {'loans': output}

@app.route("/settle\_loan/<int:loan\_id>", methods=["POST"])

@login\_required

@role\_required("finance\_officer", "admin")

def settle\_loan(loan\_id):

loan = LoanApplication.query.get\_or\_404(loan\_id)

customer = loan.customer

try:

closure\_type = request.form.get("closure\_type", "settlement").strip()

settlement\_str = request.form.get("settlement\_type", "").strip()

institution = request.form.get("settling\_institution", "").strip() or None

reason = request.form.get("settlement\_reason", "").strip() or None

VALID\_CLOSURES = {"settlement", "insurance", "write\_off"}

if closure\_type not in VALID\_CLOSURES:

flash("Invalid closure type selected.", "danger")

return redirect\_to\_referrer(customer.file\_number)

if closure\_type in {"write\_off", "settlement"} and not reason:

flash("Settlement reason is required.", "danger")

return redirect\_to\_referrer(customer.file\_number)

if closure\_type == "settlement":

if settlement\_str not in {"self", "third\_party"}:

flash("Invalid settlement sub-type selected.", "danger")

return redirect\_to\_referrer(customer.file\_number)

loan.settlement\_type = (

SettlementTypeEnum.self\_settlement if settlement\_str == "self"

else SettlementTypeEnum.third\_party

)

else:

loan.settlement\_type = None

loan.settling\_institution = institution

loan.settlement\_reason = reason

if loan.loan\_state != "active":

flash("Cannot settle a non-active loan.", "danger")

return redirect\_to\_referrer(customer.file\_number)

file = request.files.get("settle\_file")

if not file or file.filename == "":

flash("Settlement proof document is required.", "danger")

return redirect\_to\_referrer(customer.file\_number)

filename = secure\_filename(

f"settlement\_{loan.loan\_number}\_{datetime.utcnow():%Y%m%d%H%M%S}{os.path.splitext(file.filename)[1]}"

)

folder = os.path.join(app.config["UPLOAD\_FOLDER"], "settlements")

os.makedirs(folder, exist\_ok=True)

file.save(os.path.join(folder, filename))

loan.recalculate\_balance()

db.session.flush()

bal = calculate\_balances(loan)

if closure\_type == "settlement":

principal = bal["current\_balance"]

interest = bal["settlement\_interest"]

amount\_to\_pay = bal["settlement\_balance"]

else:

principal = bal["current\_balance"]

interest = 0.0

amount\_to\_pay = principal

if closure\_type == "write\_off":

loan.written\_off\_amount = round(amount\_to\_pay, 2)

elif closure\_type == "insurance":

loan.insurance\_settlement\_amount = round(amount\_to\_pay, 2)

if amount\_to\_pay <= 0:

flash("Settlement balance must be greater than zero.", "danger")

return redirect\_to\_referrer(customer.file\_number)

payment = Payment(

loan\_id=loan.id,

amount=amount\_to\_pay,

method="settlement",

status="completed",

reference=f"{closure\_type.replace('\_',' ').title()} for {loan.loan\_number}",

settlement\_proof=filename,

)

db.session.add(payment)

db.session.flush()

alloc = PaymentAllocation(

payment\_id=payment.id,

principal=principal,

interest=0.0,

fees=0.0,

settlement\_interest=interest,

)

db.session.add(alloc)

loan.loan\_state = "settled\_client" if closure\_type == "settlement" else closure\_type

loan.closure\_type = closure\_type

loan.closure\_date = datetime.utcnow()

loan.settlement\_balance = round(principal + interest, 2)

loan.current\_balance = 0.0

loan.top\_up\_balance = 0.0

for sched in loan.repayment\_schedules:

if sched.status != "paid":

sched.status = "settled"

db.session.commit()

flash(f"Loan {loan.loan\_number} settled successfully.", "success")

except Exception as e:

db.session.rollback()

app.logger.error(f"Settlement error for loan {loan\_id}: {str(e)}", exc\_info=True)

flash(f"Error settling loan: {str(e)}", "danger")

return redirect\_to\_referrer(customer.file\_number)

# Helper function for consistent redirects

def redirect\_to\_referrer(file\_number):

return redirect(request.referrer or url\_for("customer\_account",

file\_number=file\_number,

section="settlement"))

@app.route('/settlement\_report')

@login\_required

@role\_required('finance\_officer', 'admin')

def settlement\_report():

settled\_loans = (

db.session.query(

LoanApplication.loan\_number,

LoanApplication.loan\_state,

LoanApplication.settlement\_balance,

Payment.amount.label('paid\_amount'),

PaymentAllocation.principal.label('paid\_principal'),

PaymentAllocation.settlement\_interest.label('paid\_settlement\_interest'),

Payment.created\_at.label('payment\_date')

)

.join(Payment, Payment.loan\_id == LoanApplication.id)

.join(PaymentAllocation, PaymentAllocation.payment\_id == Payment.id)

.filter(

LoanApplication.loan\_state == 'settled\_client',

Payment.method == 'settlement',

Payment.status == 'completed'

)

.order\_by(Payment.created\_at.desc())

.all()

)

print(f"[DEBUG] Settled loans count: {len(settled\_loans)}") # helps you debug

return render\_template(

'settlement\_report.html',

settled\_loans=settled\_loans

)

import csv

from io import StringIO

from flask import Response

@app.route('/export\_settlement\_report\_csv')

@login\_required

@role\_required('finance\_officer', 'admin')

def export\_settlement\_report\_csv():

settled\_loans = (

db.session.query(

LoanApplication.loan\_number,

LoanApplication.loan\_state,

LoanApplication.settlement\_balance,

Payment.amount.label('paid\_amount'),

PaymentAllocation.principal.label('paid\_principal'),

PaymentAllocation.settlement\_interest.label('paid\_settlement\_interest'),

Payment.created\_at.label('payment\_date')

)

.join(Payment, Payment.loan\_id == LoanApplication.id)

.join(PaymentAllocation, PaymentAllocation.payment\_id == Payment.id)

.filter(

LoanApplication.loan\_state == 'settled\_client',

Payment.method == 'settlement',

Payment.status == 'completed'

)

.order\_by(Payment.created\_at.desc())

.all()

)

si = StringIO()

cw = csv.writer(si)

cw.writerow(['Loan Number', 'Loan State', 'Settlement Balance', 'Amount Paid', 'Principal Paid', 'Settlement Interest Paid', 'Payment Date'])

for loan in settled\_loans:

cw.writerow([

loan.loan\_number,

loan.loan\_state,

loan.settlement\_balance or 0.0,

loan.paid\_amount or 0.0,

loan.paid\_principal or 0.0,

loan.paid\_settlement\_interest or 0.0,

loan.payment\_date.strftime('%Y-%m-%d %H:%M') if loan.payment\_date else ''

])

output = Response(si.getvalue(), mimetype='text/csv')

output.headers["Content-Disposition"] = "attachment; filename=settlement\_report.csv"

return output

@app.route('/batch\_write\_off', methods=['POST'])

@login\_required

@role\_required('finance\_officer', 'admin')

def batch\_write\_off():

if 'csv\_file' not in request.files:

flash('No file part', 'danger')

return redirect(request.referrer)

file = request.files['csv\_file']

if file.filename == '':

flash('No selected file', 'danger')

return redirect(request.referrer)

if file and allowed\_file(file.filename):

try:

stream = io.StringIO(file.stream.read().decode("UTF8"), newline=None)

csv\_reader = csv.DictReader(stream)

processed = 0

for row in csv\_reader:

loan\_number = row.get('loan\_number')

reason = row.get('reason', 'No reason provided')

loan = LoanApplication.query.filter\_by(loan\_number=loan\_number).first()

if loan:

loan.loan\_state = 'written\_off'

loan.closure\_type = 'write\_off'

loan.closure\_date = datetime.utcnow()

db.session.add(loan)

processed += 1

db.session.commit()

flash(f'Successfully processed {processed} loans', 'success')

except Exception as e:

db.session.rollback()

flash(f'Error processing CSV: {str(e)}', 'danger')

else:

flash('Invalid file type', 'danger')

return redirect(request.referrer)

from datetime import datetime, timedelta, date

import calendar

import csv

from io import StringIO

from flask import render\_template, request, Response

from sqlalchemy import func

def \_parse\_date(s, default):

try:

return datetime.strptime(s, '%Y-%m-%d').date()

except (TypeError, ValueError):

return default

@app.route('/income\_report', methods=['GET', 'POST'])

@login\_required

@role\_required('finance\_officer', 'admin')

def income\_report():

today = date.today()

default\_start = date(today.year, today.month, 1)

default\_end = date(today.year, today.month, calendar.monthrange(today.year, today.month)[1])

if request.method == 'POST':

start\_date = \_parse\_date(request.form.get('start\_date'), default\_start)

end\_date = \_parse\_date(request.form.get('end\_date'), default\_end)

else:

start\_date, end\_date = default\_start, default\_end

one\_time, scheduled, event\_based = \_compute\_income\_db(start\_date, end\_date)

categories = ['civil\_servant', 'private\_sector', 'sme']

report = {cat: dict(origination=0, crb=0, insurance=0, collection=0, interest=0) for cat in categories}

for row in one\_time:

if row.category in report:

report[row.category]['origination'] += row.origination

report[row.category]['crb'] += row.crb

report[row.category]['insurance'] += row.insurance

for row in scheduled:

if row.category in report:

report[row.category]['interest'] += row.interest

report[row.category]['collection'] += row.collection

for row in event\_based:

if row.category in report:

total\_event\_interest = row.settlement\_interest + row.top\_up\_interest

report[row.category]['interest'] += total\_event\_interest

totals = {k: sum(report[cat][k] for cat in categories) for k in report[categories[0]]}

grand\_total = sum(totals.values())

return render\_template('income\_report.html',

report=report,

totals=totals,

grand\_total=grand\_total,

start\_date=start\_date.isoformat(),

end\_date=end\_date.isoformat(),

categories=categories)

# === Income computation helper ===

def \_compute\_income\_db(start\_date, end\_date):

start\_dt = datetime.combine(start\_date, datetime.min.time())

end\_dt = datetime.combine(end\_date + timedelta(days=1), datetime.min.time())

# One-time income: fees at loan creation

one\_time = (

db.session.query(

LoanApplication.category,

func.coalesce(func.sum(LoanApplication.origination\_fees), 0).label('origination'),

func.coalesce(func.sum(LoanApplication.crb\_fees), 0).label('crb'),

func.coalesce(func.sum(LoanApplication.insurance\_fees), 0).label('insurance')

)

.filter(

LoanApplication.created\_at >= start\_dt,

LoanApplication.created\_at < end\_dt,

LoanApplication.application\_status == 'approved'

)

.group\_by(LoanApplication.category)

.all()

)

# Scheduled income: interest and collection fees

scheduled\_income = (

db.session.query(

LoanApplication.category,

func.coalesce(func.sum(RepaymentSchedule.expected\_interest), 0).label('interest'),

func.coalesce(func.sum(RepaymentSchedule.expected\_fees), 0).label('collection')

)

.join(LoanApplication, LoanApplication.id == RepaymentSchedule.loan\_id)

.filter(

RepaymentSchedule.due\_date >= start\_date,

RepaymentSchedule.due\_date <= end\_date,

LoanApplication.loan\_state.in\_(['active', 'topped\_up'])

)

.group\_by(LoanApplication.category)

.all()

)

# Settlement and top-up interest (recognized at event month)

event\_interest = (

db.session.query(

LoanApplication.category,

func.coalesce(func.sum(LoanApplication.settlement\_interest), 0).label('settlement\_interest'),

func.coalesce(func.sum(LoanApplication.top\_up\_interest), 0).label('top\_up\_interest')

)

.filter(

LoanApplication.closure\_date != None,

LoanApplication.closure\_date >= start\_dt,

LoanApplication.closure\_date < end\_dt

)

.group\_by(LoanApplication.category)

.all()

)

return one\_time, scheduled\_income, event\_interest

from flask import render\_template, request

from datetime import datetime, timedelta, date

from sqlalchemy import func

@app.route('/detailed\_income\_breakdown', methods=['GET', 'POST'])

@login\_required

@role\_required('finance\_officer', 'admin')

def detailed\_income\_breakdown():

today = date.today()

default\_start = date(today.year, today.month, 1)

default\_end = date(today.year, today.month, calendar.monthrange(today.year, today.month)[1])

if request.method == 'POST':

start\_date = \_parse\_date(request.form.get('start\_date'), default\_start)

end\_date = \_parse\_date(request.form.get('end\_date'), default\_end)

else:

start\_date, end\_date = default\_start, default\_end

# Get loans with scheduled income (interest & collection fees)

results = (

db.session.query(

LoanApplication.id.label('loan\_id'),

LoanApplication.loan\_number,

LoanApplication.category,

LoanApplication.loan\_amount,

func.sum(RepaymentSchedule.expected\_interest).label('expected\_interest'),

func.sum(RepaymentSchedule.expected\_fees).label('expected\_fees')

)

.join(RepaymentSchedule, RepaymentSchedule.loan\_id == LoanApplication.id)

.filter(

RepaymentSchedule.due\_date >= start\_date,

RepaymentSchedule.due\_date <= end\_date,

LoanApplication.loan\_state.in\_(['active', 'topped\_up'])

)

.group\_by(

LoanApplication.id,

LoanApplication.loan\_number,

LoanApplication.category,

LoanApplication.loan\_amount

)

.having(

or\_(

func.sum(RepaymentSchedule.expected\_interest) > 0,

func.sum(RepaymentSchedule.expected\_fees) > 0

)

))

return render\_template(

'detailed\_income\_breakdown.html',

results=results,

start\_date=start\_date.isoformat(),

end\_date=end\_date.isoformat()

)

@app.route('/export\_income\_report\_csv')

@login\_required

@role\_required('finance\_officer', 'admin')

def export\_income\_report\_csv():

start\_date = \_parse\_date(request.args.get('start\_date'), date.today().replace(day=1))

end\_date = \_parse\_date(request.args.get('end\_date'), date.today())

one\_time, accrued, recurring = \_compute\_income\_db(start\_date, end\_date)

categories = ['civil\_servant', 'private\_sector', 'sme']

report = {cat: dict(origination=0, crb=0, insurance=0, collection=0, interest=0) for cat in categories}

for row in one\_time:

if row.category in report:

report[row.category].update(origination=row.origination, crb=row.crb, insurance=row.insurance)

for row in accrued:

if row.category in report:

report[row.category]['interest'] = row.interest

report[row.category]['collection'] = row.collection

totals = {k: sum(report[cat][k] for cat in categories) for k in report[categories[0]]}

grand\_total = sum(totals.values())

si = StringIO()

cw = csv.writer(si)

cw.writerow(['Category', 'Origination Fees', 'CRB Fees', 'Insurance Fees',

'Collection Fees', 'Interest Income', 'Total Income'])

for cat in categories:

vals = report[cat]

cat\_total = sum(vals.values())

cw.writerow([cat.replace('\_', ' ').title(),

vals['origination'], vals['crb'], vals['insurance'],

vals['collection'], vals['interest'], cat\_total])

cw.writerow(['TOTAL',

totals['origination'], totals['crb'], totals['insurance'],

totals['collection'], totals['interest'], grand\_total])

output = Response(si.getvalue(), mimetype='text/csv')

output.headers["Content-Disposition"] = (

f"attachment; filename=income\_report\_{start\_date}\_{end\_date}.csv"

)

return output

from flask import request, flash, redirect, url\_for, render\_template, Response, current\_app

from flask\_login import login\_required, current\_user

import csv

import io

import os

import traceback

import re

from datetime import datetime

from werkzeug.utils import secure\_filename

from sqlalchemy.exc import IntegrityError

# Configuration

MAX\_FILE\_SIZE = 5 \* 1024 \* 1024 # 5MB

MAX\_ROWS = 1000

ALLOWED\_EXTENSIONS = {'csv'}

def allowed\_file(filename):

return '.' in filename and \

filename.rsplit('.', 1)[1].lower() in ALLOWED\_EXTENSIONS

def safe\_int(value, field\_name, default=None):

"""Safely convert to integer with error handling"""

if not value or value.strip() == '':

if default is not None:

return default

return None

try:

return int(value)

except ValueError:

raise ValueError(f"Invalid integer value for {field\_name}: '{value}'")

def safe\_float(value, field\_name, default=None):

"""Safely convert to float with error handling"""

if not value or value.strip() == '':

if default is not None:

return default

return None

# Clean currency values

clean\_value = re.sub(r'[^\d.]', '', value)

try:

return float(clean\_value)

except ValueError:

raise ValueError(f"Invalid float value for {field\_name}: '{value}'")

def safe\_date(value, field\_name):

"""Safely convert to date with error handling"""

if not value or value.strip() == '':

raise ValueError(f"{field\_name} is required and cannot be empty")

try:

return datetime.strptime(value.strip(), '%Y-%m-%d')

except ValueError:

# Try alternative formats

try:

return datetime.strptime(value.strip(), '%m/%d/%Y')

except ValueError:

try:

return datetime.strptime(value.strip(), '%d-%m-%Y')

except ValueError:

raise ValueError(f"Invalid date format for {field\_name}. Use YYYY-MM-DD")

def get\_field(row, field\_name, aliases=None, default=''):

"""

Robust field retrieval with:

- Case insensitivity

- Alias support

- Whitespace trimming

- Default values

"""

if aliases is None:

aliases = []

# Normalize all keys to lowercase

normalized\_row = {k.strip().lower(): v for k, v in row.items()}

search\_terms = [field\_name.lower()] + [alias.lower() for alias in aliases]

for term in search\_terms:

if term in normalized\_row:

value = normalized\_row[term]

return value.strip() if isinstance(value, str) else str(value)

return default

@app.route('/admin/batch\_import\_loans', methods=['GET', 'POST'])

@login\_required

@role\_required('admin')

def batch\_import\_loans():

if request.method == 'POST':

# Check if file was uploaded

if 'file' not in request.files:

flash('No file part', 'danger')

return redirect(request.url)

file = request.files['file']

# Check if file is selected

if file.filename == '':

flash('No selected file', 'danger')

return redirect(request.url)

# Check file size

if request.content\_length > MAX\_FILE\_SIZE:

flash(f'File size exceeds {MAX\_FILE\_SIZE//(1024\*1024)}MB limit', 'danger')

return redirect(request.url)

# Check file extension

if not allowed\_file(file.filename):

flash('Invalid file type. Only CSV files are allowed', 'danger')

return redirect(request.url)

# Secure filename and read stream

filename = secure\_filename(file.filename)

try:

stream = io.StringIO(file.stream.read().decode('UTF-8', errors='replace'), newline=None)

except UnicodeDecodeError:

flash('Invalid file encoding. Please use UTF-8 encoded CSV files.', 'danger')

return redirect(request.url)

try:

# Parse CSV

reader = csv.DictReader(stream)

if not reader.fieldnames:

flash('CSV file is empty or missing headers', 'danger')

return redirect(request.url)

# Log headers for debugging

current\_app.logger.info(f"CSV headers: {reader.fieldnames}")

# Validate required columns

required\_columns = ['loan\_number']

missing\_columns = [col for col in required\_columns if col not in reader.fieldnames]

if missing\_columns:

flash(f'Missing required columns: {", ".join(missing\_columns)}', 'danger')

return redirect(request.url)

updated\_count = 0

zeroized\_count = 0

row\_errors = []

empty\_row\_count = 0

# Define all editable fields and their handlers

editable\_fields = {

# Financial fields (trigger schedule regeneration)

'loan\_amount': {

'handler': lambda v: safe\_float(v, 'loan\_amount'),

'field': 'loan\_amount',

'trigger\_regeneration': True

},

'term\_months': {

'handler': lambda v: safe\_int(v, 'term\_months'),

'field': 'term\_months',

'trigger\_regeneration': True

},

'interest\_rate': {

'handler': lambda v: safe\_float(v, 'interest\_rate'),

'field': 'interest\_rate',

'trigger\_regeneration': True

},

'disbursement\_date': {

'handler': lambda v: safe\_date(v, 'disbursement\_date'),

'field': 'disbursement\_date',

'trigger\_regeneration': True

},

# Status fields

'application\_status': {

'handler': str,

'field': 'application\_status',

'trigger\_regeneration': False

},

'loan\_state': {

'handler': str,

'field': 'loan\_state',

'trigger\_regeneration': False

},

'performance\_status': {

'handler': str,

'field': 'performance\_status',

'trigger\_regeneration': False

},

# Client information fields

'client\_name': {

'handler': str,

'field': 'client\_name',

'trigger\_regeneration': False

},

'national\_id': {

'handler': str,

'field': 'national\_id',

'trigger\_regeneration': False

},

'phone\_number': {

'handler': str,

'field': 'phone\_number',

'trigger\_regeneration': False

},

'email': {

'handler': str,

'field': 'email',

'trigger\_regeneration': False

},

'business\_name': {

'handler': str,

'field': 'business\_name',

'trigger\_regeneration': False

},

'business\_sector': {

'handler': str,

'field': 'business\_sector',

'trigger\_regeneration': False

},

'business\_address': {

'handler': str,

'field': 'business\_address',

'trigger\_regeneration': False

},

# Special backdating field

'created\_at': {

'handler': lambda v: safe\_date(v, 'created\_at'),

'field': None, # Handled separately

'trigger\_regeneration': True

}

}

# Process rows

for idx, row in enumerate(reader, start=1):

if idx > MAX\_ROWS:

flash(f'Stopped processing at row {MAX\_ROWS} (max limit reached)', 'warning')

break

# Skip completely empty rows

if all(v.strip() == '' for v in row.values() if isinstance(v, str)):

empty\_row\_count += 1

continue

try:

# Get loan number (required)

loan\_number = get\_field(row, 'loan\_number', aliases=['loan\_id', 'id'])

if not loan\_number:

raise ValueError("loan\_number is required")

# Find existing loan

loan = LoanApplication.query.filter\_by(loan\_number=loan\_number).first()

if not loan:

raise ValueError(f"Loan with number {loan\_number} not found")

# Track if we need to regenerate schedule

regenerate\_schedule = False

set\_loan\_amount\_to\_zero = False

# Process all editable fields

for field\_name, config in editable\_fields.items():

value\_str = get\_field(row, field\_name)

if value\_str:

try:

# Special handling for created\_at (backdating)

if field\_name == 'created\_at':

created\_at = config['handler'](value\_str)

loan.created\_at = created\_at

loan.disbursement\_date = created\_at

loan.date\_created = created\_at

regenerate\_schedule = True

continue

# Process other fields

value = config['handler'](value\_str)

setattr(loan, config['field'], value)

# Track if loan amount is being set to zero

if field\_name == 'loan\_amount' and value == 0:

set\_loan\_amount\_to\_zero = True

# Mark for regeneration if needed

if config['trigger\_regeneration']:

regenerate\_schedule = True

except Exception as e:

raise ValueError(f"Error processing '{field\_name}': {str(e)}")

# Handle zeroization if loan amount is set to 0

if set\_loan\_amount\_to\_zero:

zeroized\_count += 1

loan.monthly\_instalment = 0

loan.total\_repayment = 0

loan.cash\_to\_client = 0

loan.crb\_fees = 0

loan.performance\_status = 'zeroized'

# Regenerate repayment schedule if needed

if regenerate\_schedule:

RepaymentSchedule.query.filter\_by(loan\_id=loan.id).delete()

if loan.loan\_amount > 0:

loan.generate\_repayment\_schedule(disbursement\_date=loan.disbursement\_date)

# Only regenerate if not zeroized

if loan.loan\_amount > 0:

# Use the current disbursement date for schedule generation

start\_date = loan.disbursement\_date

loan.generate\_repayment\_schedule(disbursement\_date=start\_date)

# Update loan in database

db.session.add(loan)

updated\_count += 1

except Exception as e:

# Log detailed error for debugging

current\_app.logger.error(f"Error in row {idx}: {str(e)}")

current\_app.logger.error(f"Row data: {row}")

current\_app.logger.error(traceback.format\_exc())

row\_errors.append(f"Row {idx}: {str(e)}")

# Commit all changes

db.session.commit()

# Show results

if updated\_count:

flash(f'Successfully updated {updated\_count} loans', 'success')

if zeroized\_count:

flash(f'Zeroized {zeroized\_count} loans (set amount to 0)', 'info')

if empty\_row\_count:

flash(f'Skipped {empty\_row\_count} empty rows', 'info')

if row\_errors:

flash\_errors = "\n".join(row\_errors[:10]) # Show first 10 errors

if len(row\_errors) > 10:

flash\_errors += f"\n...and {len(row\_errors)-10} more errors"

flash(f'Errors encountered:\n{flash\_errors}', 'warning')

return redirect(url\_for('batch\_import\_loans'))

except Exception as e:

db.session.rollback()

current\_app.logger.error(f"Batch update failed: {str(e)}")

current\_app.logger.error(traceback.format\_exc())

flash(f'Failed to process file: {str(e)}', 'danger')

return redirect(request.url)

# For GET requests, show the import form

return render\_template('admin\_dashboard.html', section='batch\_import')

# Sample CSV Download Endpoint

@app.route('/admin/download\_sample\_csv')

@login\_required

@role\_required('admin')

def download\_sample\_csv():

sample\_data = """loan\_number,created\_at,loan\_amount,term\_months,interest\_rate,application\_status,loan\_state,performance\_status,client\_name,national\_id,phone\_number,email,business\_name,business\_sector,business\_address

LOAN-2023-001,2023-05-15,0,12,15,Approved,Closed,Zeroized,John Doe,12345678,0712345678,john@example.com,John Enterprises,Retail,Nairobi

LOAN-2023-002,2023-06-01,500000.00,24,12,Approved,Active,Performing,Jane Smith,87654321,0798765432,jane@example.com,Jane Ltd,Manufacturing,Mombasa

LOAN-2023-003,2023-07-10,,,,Approved,Active,Performing,Robert Brown,,,,,,"""

return Response(

sample\_data,

mimetype="text/csv",

headers={"Content-disposition": "attachment; filename=loan\_updates\_sample.csv"}

)

# utils/arrears.p

def refresh\_all\_arrears():

print("Refreshing arrears...")

db.session.query(Arrear).delete() # Clear all previous arrears

loans = LoanApplication.query.all()

for loan in loans:

overdue\_schedules = [s for s in loan.repayment\_schedule if s.due\_date < date.today() and not s.fully\_paid]

for schedule in overdue\_schedules:

arrear = Arrear(

loan\_id=loan.id,

schedule\_id=schedule.id,

due\_date=schedule.due\_date,

recorded\_at=date.today(),

expected\_principal=schedule.principal\_due or 0,

expected\_interest=schedule.interest\_due or 0,

expected\_fees=schedule.fees\_due or 0,

paid\_principal=schedule.paid\_principal or 0,

paid\_interest=schedule.paid\_interest or 0,

paid\_fees=schedule.paid\_fees or 0,

payment\_status='unpaid',

status='unresolved',

category=loan.category,

tenure=loan.tenure

)

# Calculate aging

age = (date.today() - arrear.due\_date).days

arrear.aging = age # store on model

if age <= 30:

bracket = "1-30"

elif age <= 60:

bracket = "31-60"

elif age <= 90:

bracket = "61-90"

elif age <= 180:

bracket = "91-180"

else:

bracket = "180+"

# Lookup provision rule

rule = ProvisionSetting.query.filter\_by(category=loan.category, tenure=loan.tenure).first()

if rule:

total\_due = arrear.expected\_principal + arrear.expected\_interest + arrear.expected\_fees

provision = total\_due \* rule.probability\_of\_default \* rule.loss\_given\_default

arrear.probability\_of\_default = rule.probability\_of\_default

arrear.loss\_given\_default = rule.loss\_given\_default

arrear.provision\_amount = round(provision, 2)

db.session.add(arrear)

db.session.commit()

print("Arrears refreshed.")

@click.command("refresh-arrears")

@with\_appcontext

def refresh\_arrears():

"""Recalculate and update all arrears from repayment schedules."""

refresh\_all\_arrears()

app.cli.add\_command(refresh\_arrears)

@app.route('/arrears')

@login\_required

@role\_required('admin', 'finance\_officer')

def view\_arrears():

status = request.args.get('status', 'unresolved')

aging = request.args.get('aging')

tenure = request.args.get('tenure')

category = request.args.get('category')

query = Arrear.query.join(LoanApplication).join(Customer)

if status:

query = query.filter(Arrear.status == status)

if aging:

try:

days = int(aging.split('-')[0])

cutoff\_date = datetime.utcnow().date() - timedelta(days=days)

query = query.filter(Arrear.due\_date <= cutoff\_date)

except:

flash("Invalid aging filter", "danger")

if tenure:

query = query.filter(Arrear.tenure == int(tenure))

if category:

query = query.filter(LoanApplication.category == category)

arrears = query.order\_by(Arrear.due\_date.asc()).all()

# Distinct filters

tenures = sorted(set([a.tenure for a in Arrear.query.distinct(Arrear.tenure).all() if a.tenure]))

categories = sorted(set([la.category for la in LoanApplication.query.distinct(LoanApplication.category).all() if la.category]))

# Tab 1: Aging Schedule

aging\_brackets = ['1-30', '31-60', '61-90', '91-180', '180+']

aging\_schedule = []

for bracket in aging\_brackets:

loans = [a for a in arrears if get\_bracket(a.days\_past\_due) == bracket]

total\_value = sum((a.expected\_principal + a.expected\_interest + a.expected\_fees) for a in loans)

total\_prov = sum(a.provision\_amount or 0 for a in loans)

aging\_schedule.append({

'bracket': bracket.replace("-", " - "),

'num\_loans': len(loans),

'value': total\_value,

'provision': total\_prov

})

# Tab 2: Aging Summary

total\_arrears\_value = sum((a.expected\_principal + a.expected\_interest + a.expected\_fees) for a in arrears)

aging\_summary = []

for row in aging\_schedule:

percentage = (row['value'] / total\_arrears\_value) \* 100 if total\_arrears\_value else 0

aging\_summary.append({

'bracket': row['bracket'],

'amount': row['value'],

'percentage': percentage

})

return render\_template(

'portfolio/arrears\_report.html',

arrears=arrears,

status=status,

aging=aging,

tenure=tenure,

category=category,

tenures=tenures,

categories=categories,

aging\_schedule=aging\_schedule,

aging\_summary=aging\_summary

)

def get\_bracket(age):

if age <= 30:

return "1-30"

elif age <= 60:

return "31-60"

elif age <= 90:

return "61-90"

elif age <= 180:

return "91-180"

return "180+"

@app.route('/resolve\_arrear/<int:arrear\_id>', methods=['POST'])

@login\_required

@role\_required('admin', 'finance\_officer')

def resolve\_arrear(arrear\_id):

arrear = Arrear.query.get\_or\_404(arrear\_id)

resolution\_type = request.form.get('resolution\_type')

notes = request.form.get('notes', '')

if resolution\_type == 'waiver':

journal = JournalEntry(

description=f"Arrear waiver for loan {arrear.loan.loan\_number}",

amount=arrear.total\_arrears,

entry\_type='waiver',

gl\_account='income\_waiver',

user\_id=current\_user.id,

loan\_id=arrear.loan\_id

)

db.session.add(journal)

arrear.status = 'resolved'

arrear.resolution\_type = resolution\_type

arrear.resolution\_notes = notes

arrear.resolved\_by = current\_user.id

arrear.resolution\_date = datetime.utcnow()

db.session.commit()

flash('Arrear resolved successfully', 'success')

return redirect(url\_for('arrears\_waterfall'))

def create\_notification(message, recipient\_id=None, type='info'):

note = Notification(

message=message,

recipient\_id=recipient\_id,

type=type,

is\_read=False

)

db.session.add(note)

db.session.commit()

def alert\_on\_new\_arrear(arrear):

# Fetch users to notify — e.g., credit officers role users

credit\_officers = User.query.filter(User.role == 'credit\_officer').all()

for officer in credit\_officers:

create\_notification(

message=f"New arrear recorded on Loan #{arrear.loan\_id} with due date {arrear.due\_date}.",

recipient\_id=officer.id,

type='warning'

)

@app.context\_processor

def inject\_notifications():

if current\_user.is\_authenticated:

unread = Notification.query.filter\_by(recipient\_id=current\_user.id, is\_read=False).order\_by(Notification.timestamp.desc()).all()

return {'unread\_notifications': unread}

return {}

@app.route('/dashboard/credit\_officer')

@login\_required

@role\_required('credit\_officer', 'admin')

def credit\_officer\_dashboard():

# Get unread notifications for current user

unread\_notifications = Notification.query.filter\_by(

recipient\_id=current\_user.id,

is\_read=False

).order\_by(Notification.timestamp.desc()).all()

# Define tasks with their display names and URLs (adjust URLs as needed)

tasks = [

{'name': 'Approve Loans', 'url': url\_for('approve\_loans')},

{'name': 'View Arrears', 'url': url\_for('view\_arrears')},

# add other tasks here...

]

return render\_template('dashboard/credit\_officer.html',

unread\_notifications=unread\_notifications,

tasks=tasks)

@app.route('/provision-rule', methods=['POST'])

@login\_required

def save\_provision\_rule():

category = request.form['category']

tenure\_group = request.form['tenure\_group']

if '+' in tenure\_group:

tenure = int(tenure\_group.replace('+', ''))

else:

tenure = int(tenure\_group.split('-')[0])

# or parse tenure\_group appropriately

pd = float(request.form['pd'])

lgd = float(request.form['lgd'])

rule = ProvisionSetting.query.filter\_by(category=category, tenure=tenure).first()

if rule:

rule.probability\_of\_default = pd

rule.loss\_given\_default = lgd

else:

rule = ProvisionSetting(category=category, tenure=tenure, probability\_of\_default=pd, loss\_given\_default=lgd)

db.session.add(rule)

db.session.commit()

flash(f'Provision rule saved for {category} [{tenure}]', 'success')

return redirect(url\_for('view\_arrears'))

def calculate\_par(loans, threshold):

total = sum(loan.balance for loan in loans)

overdue = sum(loan.balance for loan in loans if loan.days\_past\_due >= threshold)

return round((overdue / total) \* 100, 2) if total > 0 else 0.0

def snapshot\_par():

with app.app\_context():

loans = LoanApplication.query.all()

snapshot\_date = date.today()

if PARSnapshot.query.filter\_by(snapshot\_date=snapshot\_date).first():

return

par\_30 = calculate\_par(loans, 30)

par\_60 = calculate\_par(loans, 60)

par\_90 = calculate\_par(loans, 90)

snapshot = PARSnapshot(snapshot\_date=snapshot\_date, par\_30=par\_30, par\_60=par\_60, par\_90=par\_90)

db.session.add(snapshot)

db.session.commit()

scheduler = BackgroundScheduler()

scheduler.add\_job(snapshot\_par, 'cron', day\_of\_week='sun', hour=0, minute=0)

scheduler.start()

from flask import request, render\_template, make\_response

from datetime import datetime, timedelta

import csv, io

def get\_par\_data\_for\_period(period\_key, category=None, tenure=None):

today = datetime.today()

if period\_key == 'this\_quarter':

start = today - timedelta(weeks=12)

factor = 1.0

elif period\_key == 'last\_quarter':

start = today - timedelta(weeks=24)

factor = 0.8

elif period\_key == 'same\_quarter\_last\_year':

start = today - timedelta(weeks=56)

factor = 1.3

elif period\_key == '12w':

start = today - timedelta(weeks=12)

factor = 0.9

else:

start = today - timedelta(weeks=12)

factor = 1.0

weeks = list(range(12))

dates = [(start + timedelta(weeks=i)).strftime('%Y-%m-%d') for i in weeks]

# Simulated PAR % values

par30\_percent = [round(factor \* (5 + i \* 0.4), 2) for i in weeks]

par60\_percent = [round(factor \* (3 + i \* 0.25), 2) for i in weeks]

par90\_percent = [round(factor \* (1 + i \* 0.15), 2) for i in weeks]

# Compute MWK amounts from % of total portfolio

base\_portfolio\_amount = 100000 # In MWK

par30\_amount = [round(base\_portfolio\_amount \* p / 100) for p in par30\_percent]

par60\_amount = [round(base\_portfolio\_amount \* p / 100) for p in par60\_percent]

par90\_amount = [round(base\_portfolio\_amount \* p / 100) for p in par90\_percent]

return dates, weeks, {

'par30': {'percent': par30\_percent, 'amount': par30\_amount},

'par60': {'percent': par60\_percent, 'amount': par60\_amount},

'par90': {'percent': par90\_percent, 'amount': par90\_amount},

}

@app.route('/par-trend', methods=['GET'])

def par\_trend():

# Get query parameters

period1 = request.args.get('period1', '12w')

period2 = request.args.get('period2', 'last\_quarter')

category = request.args.get('category')

tenure = request.args.get('tenure')

par\_filter = int(request.args.get('par', 30)) # 30, 60, or 90

# Get structured data for both periods

dates1, weeks1, data1 = get\_par\_data\_for\_period(period1, category, tenure)

dates2, weeks2, data2 = get\_par\_data\_for\_period(period2, category, tenure)

# Extract relevant PAR level data

key = f'par{par\_filter}'

par\_percent1 = data1[key]['percent']

par\_amount1 = data1[key]['amount']

par\_percent2 = data2[key]['percent']

par\_amount2 = data2[key]['amount']

# Combine data for display or chart

zipped1 = list(zip(dates1, par\_amount1, par\_percent1))

zipped2 = list(zip(dates2, par\_amount2, par\_percent2))

return render\_template(

'par\_trend.html',

period1=period1,

period2=period2,

category=category,

tenure=tenure,

par=par\_filter,

week\_labels=dates1,

par\_amount1=par\_amount1,

par\_percent1=par\_percent1,

par\_amount2=par\_amount2,

par\_percent2=par\_percent2,

zipped1=zipped1,

zipped2=zipped2

)

@app.route('/par-trend/download', methods=['GET'])

def download\_par\_csv():

period1 = request.args.get('period1', 'this\_quarter')

period2 = request.args.get('period2', 'last\_quarter')

category = request.args.get('category')

tenure = request.args.get('tenure')

par\_bucket = request.args.get('par\_bucket', 'all')

dates1, weeks1, data1 = get\_par\_data\_for\_period(period1, category, tenure)

dates2, weeks2, data2 = get\_par\_data\_for\_period(period2, category, tenure)

if par\_bucket in ['par30', 'par60', 'par90']:

parPercent\_1 = data1[par\_bucket]['percent']

parPercent\_2 = data2[par\_bucket]['percent']

parAmount\_1 = data1[par\_bucket]['amount']

parAmount\_2 = data2[par\_bucket]['amount']

else:

parPercent\_1 = [round((data1['par30']['percent'][i] + data1['par60']['percent'][i] + data1['par90']['percent'][i]) / 3, 2) for i in range(12)]

parPercent\_2 = [round((data2['par30']['percent'][i] + data2['par60']['percent'][i] + data2['par90']['percent'][i]) / 3, 2) for i in range(12)]

parAmount\_1 = [data1['par30']['amount'][i] + data1['par60']['amount'][i] + data1['par90']['amount'][i] for i in range(12)]

parAmount\_2 = [data2['par30']['amount'][i] + data2['par60']['amount'][i] + data2['par90']['amount'][i] for i in range(12)]

output = io.StringIO()

writer = csv.writer(output)

writer.writerow([

'Date (Period 1)', 'PAR % (P1)', 'PAR Amount (P1)',

'Date (Period 2)', 'PAR % (P2)', 'PAR Amount (P2)'

])

max\_len = max(len(dates1), len(dates2))

for i in range(max\_len):

row = []

row.extend([dates1[i], parPercent\_1[i], parAmount\_1[i]] if i < len(dates1) else ['', '', ''])

row.extend([dates2[i], parPercent\_2[i], parAmount\_2[i]] if i < len(dates2) else ['', '', ''])

writer.writerow(row)

output.seek(0)

response = make\_response(output.getvalue())

response.headers["Content-Disposition"] = f"attachment; filename=par\_trend\_{period1}\_vs\_{period2}.csv"

response.headers["Content-type"] = "text/csv"

return response

def get\_arrears\_data(period\_key, category=None, tenure=None):

today = datetime.today()

if period\_key == 'this\_quarter':

start = today - timedelta(weeks=12)

offset = 0

elif period\_key == 'last\_quarter':

start = today - timedelta(weeks=24)

offset = 20

elif period\_key == 'same\_quarter\_last\_year':

start = today - timedelta(weeks=56)

offset = -15

else:

start = today - timedelta(weeks=12)

offset = 10

weeks = [f'WK{i+1}' for i in range(12)] # Create week labels

dates = [(start + timedelta(weeks=i)).strftime('%Y-%m-%d') for i in range(12)]

count = [int(40 + offset + i \* 1.5 + (10 if category else 0)) for i in range(12)]

amount = [round(8000 + offset \* 50 + i \* 600 + (100 \* int(tenure) if tenure else 0), 2) for i in range(12)]

return dates, weeks, count, amount

@app.route('/arrears-trend', methods=['GET'])

def arrears\_trend():

p1 = request.args.get('period1', '12w')

p2 = request.args.get('period2', 'last\_quarter')

cat = request.args.get('category')

ten = request.args.get('tenure')

dates1, weeks1, count1, amount1 = get\_arrears\_data(p1, cat, ten)

dates2, weeks2, count2, amount2 = get\_arrears\_data(p2, cat, ten)

return render\_template(

'portfolio/arrears\_trend.html',

period1=p1,

period2=p2,

category=cat,

tenure=ten,

week\_labels=weeks1,

count1=count1,

amount1=amount1,

count2=count2,

amount2=amount2,

zipped1=zip(dates1, count1, amount1),

zipped2=zip(dates2, count2, amount2)

)

@app.route('/arrears-trend/download')

def download\_arrears\_csv():

period1 = request.args.get('period1', 'this\_quarter')

period2 = request.args.get('period2', 'last\_quarter')

category = request.args.get('category')

tenure = request.args.get('tenure')

dates1, count1, amount1 = get\_arrears\_data(period1, category, tenure)

dates2, count2, amount2 = get\_arrears\_data(period2, category, tenure)

output = io.StringIO()

writer = csv.writer(output)

writer.writerow([

'Date (Period 1)', 'Count (P1)', 'Amount (P1)',

'Date (Period 2)', 'Count (P2)', 'Amount (P2)'

])

max\_len = max(len(dates1), len(dates2))

for i in range(max\_len):

row = [

dates1[i] if i < len(dates1) else '',

count1[i] if i < len(count1) else '',

amount1[i] if i < len(amount1) else '',

dates2[i] if i < len(dates2) else '',

count2[i] if i < len(count2) else '',

amount2[i] if i < len(amount2) else ''

]

writer.writerow(row)

output.seek(0)

response = make\_response(output.getvalue())

response.headers["Content-Disposition"] = f"attachment; filename=arrears\_trend\_{period1}\_vs\_{period2}.csv"

response.headers["Content-type"] = "text/csv"

return response

from collections import defaultdict

from datetime import datetime, timedelta

from flask import request, render\_template, redirect, url\_for, flash

from flask\_login import current\_user, login\_required

CATEGORY\_MAP = {

1: 'civil\_servant',

2: 'private\_sector',

3: 'sme'

}

@app.route('/arrears-waterfall', methods=['GET', 'POST'])

@role\_required('credit\_officer', 'admin')

def arrears\_waterfall():

status = request.args.get('status', 'unresolved')

tenure = request.args.get('tenure')

category = request.args.get('category')

# POST: update reason/action for arrears

if request.method == 'POST':

for key, value in request.form.items():

if key.startswith('reason\_') or key.startswith('action\_'):

field, arrear\_id = key.split('\_')

arrear = Arrear.query.get(int(arrear\_id))

if arrear:

if field == 'reason':

arrear.arrear\_reason = value.strip()

elif field == 'action':

arrear.action\_plan = value.strip()

db.session.commit()

flash("Arrears updated.", "success")

return redirect(url\_for('arrears\_waterfall', \*\*request.args))

arrears\_query = Arrear.query.join(LoanApplication)

if status == 'resolved':

arrears\_query = arrears\_query.filter(Arrear.status == 'resolved')

else:

arrears\_query = arrears\_query.filter(Arrear.status != 'resolved')

# Filter by tenure

if tenure:

arrears\_query = arrears\_query.filter(Arrear.tenure == int(tenure))

# Filter by category

if category:

arrears\_query = arrears\_query.filter(LoanApplication.category == category)

arrears = arrears\_query.order\_by(Arrear.due\_date.desc()).all()

# Prepare summary data for waterfall chart by arrear\_reason

summary = defaultdict(lambda: {'count': 0, 'amount': 0.0})

for a in arrears:

reason = a.arrear\_reason or 'Unspecified'

summary[reason]['count'] += 1

summary[reason]['amount'] += a.total\_arrears or 0.0

# Sort by count descending

sorted\_summary = sorted(summary.items(), key=lambda x: x[1]['count'], reverse=True)

labels = [item[0] for item in sorted\_summary]

counts = [item[1]['count'] for item in sorted\_summary]

amounts = [round(item[1]['amount'], 2) for item in sorted\_summary]

summary\_data = {

'labels': labels,

'counts': counts,

'amounts': amounts

}

# Get distinct values for filters

tenures = sorted({a.tenure for a in Arrear.query.with\_entities(Arrear.tenure).distinct() if a.tenure})

categories = list({app.category for app in LoanApplication.query.with\_entities(LoanApplication.category).distinct()})

return render\_template('portfolio/arrears\_waterfall.html',

arrears=arrears,

status=status,

tenure=tenure,

category=category,

tenures=tenures,

categories=categories,

summary\_data=summary\_data)

def validate\_journal\_entries(loan: LoanApplication):

try:

principal\_entries = sum(e.amount for e in loan.journal\_entries

if e.entry\_type == 'principal\_recovery')

interest\_entries = sum(e.amount for e in loan.journal\_entries

if e.entry\_type == 'interest\_income')

expected\_principal = loan.calculated.current\_balance

expected\_interest = getattr(loan, f"{loan.closure\_type}\_interest", 0)

if not math.isclose(abs(principal\_entries), expected\_principal, abs\_tol=0.01):

raise AccountingError(f"Principal mismatch: Expected {expected\_principal}, Found {abs(principal\_entries)}")

if not math.isclose(interest\_entries, expected\_interest, abs\_tol=0.01):

raise AccountingError(f"Interest mismatch: Expected {expected\_interest}, Found {interest\_entries}")

except AccountingError as ae:

app.logger.error(f"Accounting validation failed: {str(ae)}")

raise

except Exception as e:

app.logger.error(f"Validation error: {str(e)}")

raise AccountingError("General accounting validation failure") from e

def generate\_placement\_schedule(placement):

from datetime import datetime

from dateutil.relativedelta import relativedelta

start\_date = placement.start\_date

tenure = placement.tenure\_months or 0

frequency = placement.payment\_frequency\_months or 1 # fallback default

rate\_annual = (placement.interest\_rate or 0) / 100

principal = placement.amount or 0.0

# Validate inputs

if frequency <= 0 or tenure <= 0 or principal <= 0:

return []

num\_periods = tenure // frequency

if num\_periods <= 0:

return []

periodic\_rate = rate\_annual \* (frequency / 12)

schedule = []

# Compound Interest (Amortized)

if placement.interest\_type == 'Compound':

if periodic\_rate > 0:

emi = principal \* (periodic\_rate \* (1 + periodic\_rate) \*\* num\_periods) / ((1 + periodic\_rate) \*\* num\_periods - 1)

else:

emi = principal / num\_periods

emi = round(emi, 2)

balance = principal

for i in range(num\_periods):

due\_date = start\_date + relativedelta(months=frequency \* (i + 1))

interest\_due = round(balance \* periodic\_rate, 2)

principal\_due = round(emi - interest\_due, 2)

balance = round(balance - principal\_due, 2)

if i == num\_periods - 1:

principal\_due += balance

interest\_due = round(emi - principal\_due, 2)

balance = 0.0

schedule.append(PlacementSchedule(

placement\_id=placement.id,

due\_date=due\_date,

interest\_due=interest\_due,

principal\_due=principal\_due,

total\_due=round(interest\_due + principal\_due, 2),

is\_paid=False

))

# Simple Interest

elif placement.interest\_type == 'Simple':

total\_interest = principal \* rate\_annual \* (tenure / 12)

interest\_per\_period = round(total\_interest / num\_periods, 2)

for i in range(num\_periods):

due\_date = start\_date + relativedelta(months=frequency \* (i + 1))

schedule.append(PlacementSchedule(

placement\_id=placement.id,

due\_date=due\_date,

interest\_due=interest\_per\_period,

principal\_due=0.0,

total\_due=interest\_per\_period,

is\_paid=False

))

return schedule

@app.route('/placement/new', methods=['GET', 'POST'])

def new\_placement():

form = PlacementForm()

form.client\_id.choices = [(c.id, c.full\_name) for c in Client.query.all()]

if form.validate\_on\_submit():

placement\_number = generate\_placement\_number(form.tenure\_months.data)

# Set a safe default for payment frequency

payment\_freq = form.payment\_frequency\_months.data or 1

placement = Placement(

client\_id=form.client\_id.data,

amount=form.amount.data,

interest\_rate=form.interest\_rate.data,

interest\_type=form.interest\_type.data,

tenure\_months=form.tenure\_months.data,

start\_date=form.start\_date.data,

payment\_frequency\_months=payment\_freq,

commission\_percentage=form.commission\_percentage.data,

arrangement\_fee=form.arrangement\_fee.data,

collateral=form.collateral.data,

due\_date=form.start\_date.data + relativedelta(months=form.tenure\_months.data),

current\_balance=form.amount.data,

placement\_number=placement\_number,

interest\_due=0.0,

principal\_due=0.0,

total\_due=0.0,

is\_paid=False,

status="Active",

last\_interest\_calculation=form.start\_date.data

)

db.session.add(placement)

db.session.commit()

# Generate repayment schedule

schedule = generate\_placement\_schedule(placement)

if schedule:

db.session.bulk\_save\_objects(schedule)

# Create initial deposit transaction

transaction = PlacementTransaction(

placement\_id=placement.id,

transaction\_type='Deposit',

amount=form.amount.data,

description='Initial deposit'

)

db.session.add(transaction)

db.session.commit()

flash("Placement created successfully", "success")

return redirect(url\_for('placement\_details', placement\_id=placement.id))

return render\_template('placement/new.html', form=form)

@app.route('/placement/<int:placement\_id>')

def placement\_details(placement\_id):

placement = Placement.query.get\_or\_404(placement\_id)

transactions = PlacementTransaction.query.filter\_by(placement\_id=placement\_id)\

.order\_by(PlacementTransaction.transaction\_date.desc()).all()

accrued\_interest = placement.accrued\_interest()

return render\_template('placement/details.html',

placement=placement,

transactions=transactions,

accrued\_interest=accrued\_interest)

@app.route('/placement/<int:placement\_id>/deposit', methods=['GET', 'POST'])

def add\_deposit(placement\_id):

placement = Placement.query.get\_or\_404(placement\_id)

form = DepositForm()

if form.validate\_on\_submit():

try:

placement.add\_deposit(form.amount.data, form.transaction\_date.data)

db.session.commit()

flash("Deposit added successfully", "success")

return redirect(url\_for('placement\_details', placement\_id=placement.id))

except Exception as e:

flash(str(e), "danger")

return render\_template('placement/deposit.html', form=form, placement=placement)

@app.route('/placement/<int:placement\_id>/withdraw', methods=['GET', 'POST'])

def withdraw\_funds(placement\_id):

placement = Placement.query.get\_or\_404(placement\_id)

form = WithdrawalForm()

if form.validate\_on\_submit():

try:

placement.withdraw\_funds(form.amount.data, form.transaction\_date.data)

db.session.commit()

flash("Withdrawal processed successfully", "success")

return redirect(url\_for('placement\_details', placement\_id=placement.id))

except Exception as e:

flash(str(e), "danger")

return render\_template('placement/withdraw.html', form=form, placement=placement)

@app.route('/placement/<int:placement\_id>/change\_rate', methods=['GET', 'POST'])

def change\_interest\_rate(placement\_id):

placement = Placement.query.get\_or\_404(placement\_id)

form = RateChangeForm()

if form.validate\_on\_submit():

placement.change\_interest\_rate(form.new\_rate.data, form.effective\_date.data)

db.session.commit()

flash("Interest rate changed successfully", "success")

return redirect(url\_for('placement\_details', placement\_id=placement\_id))

return render\_template('placement/change\_rate.html', form=form, placement=placement)

@app.route('/placement/<int:placement\_id>/capitalize', methods=['GET', 'POST'])

def capitalize\_interest(placement\_id):

placement = Placement.query.get\_or\_404(placement\_id)

form = CapitalizeForm()

if form.validate\_on\_submit():

interest = placement.capitalize\_interest(form.transaction\_date.data)

if interest > 0:

db.session.commit()

flash(f"${interest:,.2f} interest capitalized", "success")

else:

flash("No interest to capitalize", "info")

return redirect(url\_for('placement\_details', placement\_id=placement\_id))

return render\_template('placement/capitalize.html', form=form, placement=placement)

@app.route('/placement/<int:placement\_id>/liquidate', methods=['GET', 'POST'])

def liquidate\_placement(placement\_id):

placement = Placement.query.get\_or\_404(placement\_id)

form = LiquidateForm()

if form.validate\_on\_submit():

final\_balance = placement.liquidate(form.transaction\_date.data)

db.session.commit()

flash(f"Placement liquidated. Final balance: ${final\_balance:,.2f}", "success")

return redirect(url\_for('placement\_details', placement\_id=placement\_id))

return render\_template('placement/liquidate.html', form=form, placement=placement)

@app.route('/placements')

def view\_placements():

from datetime import date

today = date.today()

month\_start = today.replace(day=1)

if today.month == 12:

month\_end = today.replace(year=today.year + 1, month=1, day=1)

else:

month\_end = today.replace(month=today.month + 1, day=1)

placements\_query = Placement.query

interest\_start = request.args.get('interest\_start')

interest\_end = request.args.get('interest\_end')

maturity\_start = request.args.get('maturity\_start')

maturity\_end = request.args.get('maturity\_end')

if interest\_start:

placements\_query = placements\_query.filter(Placement.next\_interest\_date >= interest\_start)

if interest\_end:

placements\_query = placements\_query.filter(Placement.next\_interest\_date <= interest\_end)

if maturity\_start:

placements\_query = placements\_query.filter(Placement.start\_date >= maturity\_start)

if maturity\_end:

placements\_query = placements\_query.filter(Placement.maturity\_date <= maturity\_end)

placements = placements\_query.all()

# Get scheduled interest per placement this month

scheduled\_map = {

row.placement\_id: row.interest\_due

for row in db.session.query(

PlacementSchedule.placement\_id,

db.func.sum(PlacementSchedule.interest\_due).label('interest\_due')

).filter(

PlacementSchedule.due\_date >= month\_start,

PlacementSchedule.due\_date < month\_end

).group\_by(PlacementSchedule.placement\_id).all()

}

total\_amount = 0

weighted\_cost\_sum = 0

total\_interest = 0

total\_scheduled\_interest = 0

total\_balance = 0

for p in placements:

p.scheduled\_interest = scheduled\_map.get(p.id, 0.0)

p.total\_cost = (p.interest\_rate or 0) + (p.commission\_percentage or 0) + (p.arrangement\_fee or 0)

total\_interest += p.accrued\_interest() or 0

total\_scheduled\_interest += p.scheduled\_interest

total\_balance += p.current\_balance or 0

total\_amount += p.amount or 0

weighted\_cost\_sum += (p.amount or 0) \* p.total\_cost

weighted\_cost = (weighted\_cost\_sum / total\_amount) if total\_amount else 0.0

return render\_template(

'placement/list.html',

placements=placements,

total\_interest=total\_interest,

total\_scheduled\_interest=total\_scheduled\_interest,

total\_balance=total\_balance,

weighted\_cost=weighted\_cost

)

@app.route('/placements/export')

def export\_placements\_excel():

import pandas as pd

from io import BytesIO

from flask import send\_file

placements = Placement.query.all()

data = [{

'Client': p.client.full\_name if p.client else 'N/A',

'Start Date': p.start\_date,

'Maturity Date': p.maturity\_date,

'Interest Rate': p.interest\_rate,

'Commission %': p.commission\_percentage,

'Arrangement Fee %': p.arrangement\_fee,

'Total Cost %': (p.interest\_rate or 0) + (p.commission\_percentage or 0) + (p.arrangement\_fee or 0),

'Balance': p.current\_balance,

'Status': p.status,

} for p in placements]

df = pd.DataFrame(data)

output = BytesIO()

df.to\_excel(output, index=False)

output.seek(0)

return send\_file(output, mimetype='application/vnd.openxmlformats-officedocument.spreadsheetml.sheet',

download\_name='placements.xlsx', as\_attachment=True)

from datetime import date

from collections import defaultdict

from datetime import date

def get\_scheduled\_interest\_by\_placement():

today = date.today()

month\_start = today.replace(day=1)

if today.month == 12:

month\_end = today.replace(year=today.year + 1, month=1, day=1)

else:

month\_end = today.replace(month=today.month + 1, day=1)

results = db.session.query(

PlacementSchedule.placement\_id,

db.func.sum(PlacementSchedule.interest\_due)

).filter(

PlacementSchedule.due\_date >= month\_start,

PlacementSchedule.due\_date < month\_end

).group\_by(PlacementSchedule.placement\_id).all()

return {pid: total for pid, total in results}

def get\_monthly\_scheduled\_interest():

today = date.today()

month\_start = today.replace(day=1)

if today.month == 12:

month\_end = today.replace(year=today.year + 1, month=1, day=1)

else:

month\_end = today.replace(month=today.month + 1, day=1)

return db.session.query(

db.func.sum(PlacementSchedule.interest\_due)

).filter(

PlacementSchedule.due\_date >= month\_start,

PlacementSchedule.due\_date < month\_end

).scalar() or 0.0

@app.route('/placements-interest')

def placements\_interest():

total\_interest = get\_monthly\_scheduled\_interest()

placements = Placement.query.all()

# Optionally, attach scheduled interest per placement if needed:

# This requires a query per placement or a batch query to be efficient,

# but if you want just the total scheduled interest overall, this is enough.

return render\_template('placement/list.html', placements=placements, total\_interest=total\_interest)

# Add Jinja currency filter

@app.template\_filter('format\_currency')

def format\_currency\_filter(value):

try:

return "${:,.2f}".format(value or 0)

except Exception:

return "$0.00"

from sqlalchemy import func

def generate\_placement\_number(tenure\_months):

now = datetime.utcnow()

month = now.strftime('%m')

year = now.strftime('%y')

# Get the last sequence number

last\_placement = Placement.query.order\_by(Placement.id.desc()).first()

last\_seq = 0

if last\_placement and last\_placement.placement\_number:

try:

last\_seq = int(last\_placement.placement\_number[-4:])

except:

pass

next\_seq = last\_seq + 1

return f"{int(tenure\_months):02d}{month}{year}{next\_seq:04d}"

@app.route('/placement/<int:placement\_id>/statement')

def placement\_statement(placement\_id):

placement = Placement.query.get\_or\_404(placement\_id)

transactions = PlacementTransaction.query.filter\_by(placement\_id=placement\_id)\

.order\_by(PlacementTransaction.transaction\_date.asc()).all()

return render\_template('placement/statement.html',

placement=placement,

transactions=transactions)

from dateutil.relativedelta import relativedelta

def calculate\_interest\_due(placement, period\_index, total\_periods):

"""You can customize interest calculation per frequency."""

# Simple example: evenly divide total interest

total\_interest = placement.amount \* (placement.interest\_rate / 100) \* (placement.tenure\_months / 12)

return round(total\_interest / total\_periods, 2)

@app.route('/placement/<int:placement\_id>/schedule')

def view\_placement\_schedule(placement\_id):

placement = Placement.query.get\_or\_404(placement\_id)

schedule = PlacementSchedule.query.filter\_by(placement\_id=placement\_id).order\_by(PlacementSchedule.due\_date).all()

return render\_template('placement/schedule.html', placement=placement, schedule=schedule)

@app.route('/add-client', methods=['GET', 'POST'])

def add\_client():

form = ClientForm()

if form.validate\_on\_submit():

existing = Client.query.filter\_by(national\_id=form.national\_id.data).first()

if existing:

flash('Client already exists', 'warning')

else:

new\_client = Client(

full\_name=form.full\_name.data,

national\_id=form.national\_id.data,

phone=form.phone.data,

email=form.email.data

)

db.session.add(new\_client)

db.session.commit()

flash('Client added successfully', 'success')

return redirect(url\_for('add\_client'))

return render\_template('add\_client.html', form=form)

def change\_tenure(self, new\_tenure, effective\_date):

from dateutil.relativedelta import relativedelta

# Capitalize interest up to effective\_date

self.capitalize\_interest(effective\_date)

# Update tenure and due\_date

self.tenure\_months = new\_tenure

self.due\_date = self.start\_date + relativedelta(months=new\_tenure)

self.last\_interest\_calculation = effective\_date

# Delete old schedules

PlacementSchedule.query.filter\_by(placement\_id=self.id).delete()

# Generate new schedule

new\_schedule = generate\_placement\_schedule(self)

if new\_schedule:

db.session.bulk\_save\_objects(new\_schedule)

db.session.commit()

def update\_frequency(self, new\_frequency, effective\_date):

from dateutil.relativedelta import relativedelta

# Capitalize interest up to effective\_date

self.capitalize\_interest(effective\_date)

# Update frequency

freq\_map = {

'monthly': 1,

'quarterly': 3,

'annually': 12

}

self.payment\_frequency\_months = freq\_map.get(new\_frequency, 1)

self.payment\_frequency = new\_frequency.capitalize()

self.last\_interest\_calculation = effective\_date

# Delete old schedules

PlacementSchedule.query.filter\_by(placement\_id=self.id).delete()

# Generate new schedule

new\_schedule = generate\_placement\_schedule(self)

if new\_schedule:

db.session.bulk\_save\_objects(new\_schedule)

db.session.commit()

@app.route("/placement/<int:placement\_id>/update", methods=["GET", "POST"])

def update\_placement(placement\_id):

placement = Placement.query.get\_or\_404(placement\_id)

form = PlacementUpdateForm()

if form.validate\_on\_submit():

changes\_made = False

effective\_date = form.effective\_date.data

description = form.description.data or "Placement update"

# Handle deposit or withdrawal

if form.amount.data:

amount = float(form.amount.data)

if amount > 0:

placement.amount += amount

placement.current\_balance += amount

tx = PlacementTransaction(

placement\_id=placement.id,

transaction\_type='Deposit',

amount=amount,

description=description,

transaction\_date=effective\_date

)

db.session.add(tx)

elif amount < 0:

withdrawal\_amount = abs(amount)

if withdrawal\_amount > placement.current\_balance:

flash("Insufficient balance for withdrawal", "danger")

return redirect(request.url)

placement.amount -= withdrawal\_amount

placement.current\_balance -= withdrawal\_amount

tx = PlacementTransaction(

placement\_id=placement.id,

transaction\_type='Withdrawal',

amount=withdrawal\_amount,

description=description,

transaction\_date=effective\_date

)

db.session.add(tx)

changes\_made = True

# Handle interest rate change

if form.new\_interest\_rate.data is not None:

placement.interest\_rate = float(form.new\_interest\_rate.data)

tx = PlacementTransaction(

placement\_id=placement.id,

transaction\_type='Interest Rate Change',

amount=0.0,

description=f"Rate changed to {form.new\_interest\_rate.data}%",

transaction\_date=effective\_date

)

db.session.add(tx)

changes\_made = True

# Handle tenure change

if form.new\_tenure\_months.data is not None:

placement.tenure\_months = form.new\_tenure\_months.data

tx = PlacementTransaction(

placement\_id=placement.id,

transaction\_type='Tenure Change',

amount=0.0,

description=f"Tenure changed to {form.new\_tenure\_months.data} months",

transaction\_date=effective\_date

)

db.session.add(tx)

changes\_made = True

# Handle frequency change

if form.new\_payment\_frequency.data is not None:

placement.payment\_frequency\_months = form.new\_payment\_frequency.data

tx = PlacementTransaction(

placement\_id=placement.id,

transaction\_type='Frequency Change',

amount=0.0,

description=f"Frequency changed to {form.new\_payment\_frequency.data} months",

transaction\_date=effective\_date

)

db.session.add(tx)

changes\_made = True

if changes\_made:

# Recalculate due date

placement.due\_date = placement.start\_date + relativedelta(months=placement.tenure\_months)

# Delete old schedule and regenerate

PlacementSchedule.query.filter\_by(placement\_id=placement.id).delete()

new\_schedule = generate\_placement\_schedule(placement)

db.session.bulk\_save\_objects(new\_schedule)

db.session.commit()

flash("Placement updated and schedule regenerated.", "success")

return redirect(url\_for('placement\_details', placement\_id=placement.id))

else:

flash("No changes submitted.", "warning")

# Pre-fill form with current values

if request.method == 'GET':

form.new\_interest\_rate.data = placement.interest\_rate

form.new\_tenure\_months.data = placement.tenure\_months

form.new\_payment\_frequency.data = placement.payment\_frequency\_months

form.effective\_date.data = date.today()

return render\_template("placement/update.html", form=form, placement=placement)

from flask import render\_template, request, redirect, url\_for, flash, jsonify

from datetime import datetime, timedelta, date

from sqlalchemy import func, distinct, cast, Date

from app import app, db, mail

from flask\_mail import Message

from apscheduler.schedulers.background import BackgroundScheduler

import calendar

@app.route('/dashboard/sales')

def sales\_dashboard():

region = request.args.get('region')

district = request.args.get('district')

team\_leader = request.args.get('team\_leader')

query = Agent.query.filter(Agent.active.is\_(True), Agent.role != 'Team Leader')

filters = {'region': region, 'district': district, 'team\_leader': team\_leader}

for field, value in filters.items():

if value:

query = query.filter(getattr(Agent, field) == value)

agents = query.all()

regions = [r[0] for r in db.session.query(Agent.region).distinct()]

districts = [d[0] for d in db.session.query(Agent.district).distinct()]

team\_leads = [t[0] for t in db.session.query(Agent.team\_leader).distinct() if t[0]]

stats = []

for agent in agents:

applications = LoanApplication.query.filter(

LoanApplication.agent\_id == agent.id

).count()

loan\_amounts = db.session.query(func.sum(LoanApplication.loan\_amount)) \

.filter(LoanApplication.agent\_id == agent.id).scalar() or 0

active\_days = db.session.query(

func.count(distinct(cast(LoanApplication.created\_at, Date)))

).filter(LoanApplication.agent\_id == agent.id).scalar()

applications\_per\_day = round(applications / active\_days, 2) if active\_days else 0

stats.append({

'name': agent.name,

'region': agent.region,

'district': agent.district,

'applications': applications,

'total\_loan\_amount': loan\_amounts,

'applications\_per\_day': applications\_per\_day,

'active\_days': active\_days,

'budget': agent.monthly\_budget,

'budget\_achievement': round((loan\_amounts / agent.monthly\_budget) \* 100, 1) if agent.monthly\_budget else None

})

notif = get\_sales\_notification()

return render\_template(

"dashboard/sales\_dashboard.html",

stats=stats,

regions=regions,

districts=districts,

team\_leads=team\_leads,

active\_tab="agent",

notif=notif

)

@app.route('/notifications/sales-summary')

def sales\_summary\_notification():

today = datetime.today().date()

start\_of\_month = today.replace(day=1)

def working\_days(start, end):

return sum(1 for d in (start + timedelta(n) for n in range((end - start).days + 1)) if d.weekday() < 5)

working\_days\_passed = working\_days(start\_of\_month, today)

total\_working\_days = working\_days(start\_of\_month, today.replace(day=calendar.monthrange(today.year, today.month)[1]))

todays\_sales = db.session.query(func.sum(LoanApplication.loan\_amount)) \

.filter(func.date(LoanApplication.created\_at) == today).scalar() or 0

mtd\_sales = db.session.query(func.sum(LoanApplication.loan\_amount)) \

.filter(LoanApplication.created\_at >= start\_of\_month).scalar() or 0

total\_budget = db.session.query(func.sum(Agent.monthly\_budget)).scalar() or 0

budget\_achievement = (mtd\_sales / total\_budget \* 100) if total\_budget else 0

working\_days\_ratio = (working\_days\_passed / total\_working\_days \* 100) if total\_working\_days else 0

return jsonify({

"today\_sales": round(todays\_sales, 2),

"mtd\_sales": round(mtd\_sales, 2),

"total\_budget": round(total\_budget, 2),

"achievement\_percent": round(budget\_achievement, 2),

"working\_days\_percent": round(working\_days\_ratio, 2),

"on\_track": budget\_achievement >= working\_days\_ratio

})

@app.route('/dashboard/team-leader')

def team\_leader\_dashboard():

from sqlalchemy.orm import aliased

TeamLeader = aliased(Agent)

stats = (

db.session.query(

TeamLeader,

func.count(LoanApplication.id).label('applications'),

func.sum(LoanApplication.loan\_amount).label('total\_loan\_amount'),

func.sum(Agent.monthly\_budget).label('team\_budget')

)

.join(Agent, LoanApplication.agent\_id == Agent.id)

.join(TeamLeader, Agent.team\_leader\_id == TeamLeader.id)

.group\_by(TeamLeader)

.all()

)

# Build list for template

team\_stats = []

for row in stats:

team\_leader = row[0] # TeamLeader is the first element of the row tuple

achievement = round((row.total\_loan\_amount / row.team\_budget) \* 100, 2) if row.team\_budget else 0

team\_stats.append({

"team\_leader": team\_leader, # Full object, access .name, .email, etc.

"applications": row.applications,

"total\_loan\_amount": row.total\_loan\_amount,

"team\_budget": row.team\_budget,

"achievement": achievement

})

return render\_template("dashboard/team\_leader\_dashboard.html", stats=team\_stats, active\_tab="team")

@app.route('/agents/team/add', methods=['GET', 'POST'])

def add\_team\_with\_agents():

if request.method == 'POST':

try:

# Leader: existing or new

existing\_leader\_id = request.form.get('existing\_leader\_id')

if existing\_leader\_id:

leader = Agent.query.get(int(existing\_leader\_id))

print(f"Using existing team leader: {leader.name} (ID: {leader.id})")

else:

leader = Agent(

name=request.form['leader\_name'],

contact=request.form.get('leader\_phone'),

email=request.form.get('leader\_email'),

role='Team Leader'

)

db.session.add(leader)

db.session.flush() # assigns leader.id

print(f"Created new team leader: {leader.name} (ID: {leader.id})")

# Fetch submitted agent data

names = request.form.getlist('agent\_name[]')

phones = request.form.getlist('agent\_phone[]')

emails = request.form.getlist('agent\_email[]')

districts = request.form.getlist('agent\_district[]')

regions = request.form.getlist('agent\_region[]')

budgets = request.form.getlist('agent\_budget[]')

print("Agents submitted:", names)

added\_agents = 0

for name, phone, email, district, region, budget\_str in zip(

names, phones, emails, districts, regions, budgets

):

if not name.strip():

print("Skipped agent with empty name.")

continue

try:

budget = float(budget\_str) if budget\_str else 0.0

except ValueError:

budget = 0.0

agent = Agent(

name=name.strip(),

contact=phone.strip(),

email=email.strip(),

district=district.strip(),

region=region.strip(),

monthly\_budget=budget,

role='Agent',

team\_leader\_id=leader.id

)

db.session.add(agent)

added\_agents += 1

print(f"Added agent: {agent.name}, team leader: {leader.name}")

if added\_agents == 0:

raise Exception("No valid agents submitted.")

db.session.commit()

flash(f"Team created with {added\_agents} agent(s)!", "success")

return redirect(url\_for('view\_agents'))

except Exception as e:

db.session.rollback()

flash(f"Error creating team: {str(e)}", "danger")

print("Exception:", e)

# GET

team\_leaders = Agent.query.filter\_by(role='Team Leader').order\_by(Agent.name).all()

return render\_template('agents/add\_team\_with\_agents.html', team\_leaders=team\_leaders)

def get\_sales\_notification():

today = date.today()

start\_of\_month = today.replace(day=1)

total\_days = calendar.monthrange(today.year, today.month)[1]

working\_days = [

start\_of\_month + timedelta(days=i)

for i in range(total\_days)

if (start\_of\_month + timedelta(days=i)).weekday() < 5

]

days\_passed = [d for d in working\_days if d <= today]

today\_sales = db.session.query(func.sum(LoanApplication.loan\_amount)) \

.filter(func.date(LoanApplication.created\_at) == today).scalar() or 0

mtd\_sales = db.session.query(func.sum(LoanApplication.loan\_amount)) \

.filter(LoanApplication.created\_at >= start\_of\_month).scalar() or 0

total\_budget = db.session.query(func.sum(Agent.monthly\_budget)).scalar() or 0

sales\_pct = (mtd\_sales / total\_budget \* 100) if total\_budget else 0

time\_pct = (len(days\_passed) / len(working\_days) \* 100) if working\_days else 0

return {

"today\_sales": round(today\_sales, 2),

"mtd\_sales": round(mtd\_sales, 2),

"total\_budget": round(total\_budget, 2),

"sales\_pct": round(sales\_pct, 1),

"time\_pct": round(time\_pct, 1),

"working\_days": len(working\_days),

"days\_passed": len(days\_passed),

"report\_date": today.strftime("%B %d, %Y")

}

def get\_historical\_comparisons(today):

"""Calculate comparisons with last month and last year"""

# Current period

start\_of\_month = today.replace(day=1)

# Last month period

if today.month == 1:

last\_month = today.replace(year=today.year-1, month=12)

else:

last\_month = today.replace(month=today.month-1)

start\_of\_last\_month = last\_month.replace(day=1)

# Last year period

last\_year = today.replace(year=today.year-1)

start\_of\_last\_year = last\_year.replace(day=1)

# Get day ranges

current\_days = min(today.day, calendar.monthrange(today.year, today.month)[1])

last\_month\_days = min(today.day, calendar.monthrange(last\_month.year, last\_month.month)[1])

last\_year\_days = min(today.day, calendar.monthrange(last\_year.year, last\_year.month)[1])

return {

'start\_of\_month': start\_of\_month,

'start\_of\_last\_month': start\_of\_last\_month,

'start\_of\_last\_year': start\_of\_last\_year,

'current\_days': current\_days,

'last\_month\_days': last\_month\_days,

'last\_year\_days': last\_year\_days

}

def get\_sales\_data():

today = date.today()

comparisons = get\_historical\_comparisons(today)

# Calculate working days

total\_days = calendar.monthrange(today.year, today.month)[1]

working\_days = [

comparisons['start\_of\_month'] + timedelta(days=i)

for i in range(total\_days)

if (comparisons['start\_of\_month'] + timedelta(days=i)).weekday() < 5

]

days\_passed = [d for d in working\_days if d <= today]

db\_session = Session()

try:

# TODAY'S SALES

today\_sales = db\_session.query(func.sum(LoanApplication.loan\_amount)) \

.filter(func.date(LoanApplication.created\_at) == today).scalar() or 0

# MTD SALES

mtd\_sales = db\_session.query(func.sum(LoanApplication.loan\_amount)) \

.filter(LoanApplication.created\_at >= comparisons['start\_of\_month']).scalar() or 0

# TOTAL BUDGET

total\_budget = db\_session.query(func.sum(Agent.monthly\_budget)).scalar() or 0

# HISTORICAL COMPARISONS

# Last month same day

last\_month\_same\_day = db\_session.query(func.sum(LoanApplication.loan\_amount)) \

.filter(extract('day', LoanApplication.created\_at) == today.day,

extract('month', LoanApplication.created\_at) == comparisons['start\_of\_last\_month'].month,

extract('year', LoanApplication.created\_at) == comparisons['start\_of\_last\_month'].year) \

.scalar() or 0

# Last month MTD

last\_month\_mtd = db\_session.query(func.sum(LoanApplication.loan\_amount)) \

.filter(LoanApplication.created\_at >= comparisons['start\_of\_last\_month'],

LoanApplication.created\_at <= comparisons['start\_of\_last\_month'] + timedelta(days=comparisons['last\_month\_days'] - 1)) \

.scalar() or 0

# Last year MTD

last\_year\_mtd = db\_session.query(func.sum(LoanApplication.loan\_amount)) \

.filter(LoanApplication.created\_at >= comparisons['start\_of\_last\_year'],

LoanApplication.created\_at <= comparisons['start\_of\_last\_year'] + timedelta(days=comparisons['last\_year\_days'] - 1)) \

.scalar() or 0

# REGIONAL PERFORMANCE

regional\_data = db\_session.query(

Agent.region,

func.sum(LoanApplication.loan\_amount).label('total\_sales'),

func.count(LoanApplication.id).label('loan\_count')

) \

.join(LoanApplication) \

.filter(LoanApplication.created\_at >= comparisons['start\_of\_month']) \

.group\_by(Agent.region).all()

# Convert to dictionary

regional\_performance = [{

'region': r[0],

'total\_sales': r[1] or 0,

'loan\_count': r[2],

'budget\_percent': (r[1] / total\_budget \* 100) if total\_budget else 0

} for r in regional\_data]

# LOAN CATEGORY ANALYSIS

category\_data = db\_session.query(

LoanApplication.category,

func.sum(LoanApplication.loan\_amount).label('total\_sales'),

func.count(LoanApplication.id).label('loan\_count')

) \

.filter(LoanApplication.created\_at >= comparisons['start\_of\_month']) \

.group\_by(LoanApplication.category).all()

# Convert to dictionary

category\_performance = [{

'category': c[0],

'total\_sales': c[1] or 0,

'loan\_count': c[2],

'percent\_of\_total': (c[1] / mtd\_sales \* 100) if mtd\_sales else 0

} for c in category\_data]

# Calculate percentages

sales\_pct = (mtd\_sales / total\_budget \* 100) if total\_budget else 0

time\_pct = (len(days\_passed) / len(working\_days) \* 100) if working\_days else 0

# Calculate growth rates

today\_growth = ((today\_sales - last\_month\_same\_day) / last\_month\_same\_day \* 100) if last\_month\_same\_day else 0

mtd\_growth\_month = ((mtd\_sales - last\_month\_mtd) / last\_month\_mtd \* 100) if last\_month\_mtd else 0

mtd\_growth\_year = ((mtd\_sales - last\_year\_mtd) / last\_year\_mtd \* 100) if last\_year\_mtd else 0

return {

"today\_sales": round(today\_sales, 2),

"mtd\_sales": round(mtd\_sales, 2),

"total\_budget": round(total\_budget, 2),

"sales\_pct": round(sales\_pct, 1),

"time\_pct": round(time\_pct, 1),

"working\_days": len(working\_days),

"days\_passed": len(days\_passed),

"report\_date": today.strftime("%B %d, %Y"),

"regional\_performance": regional\_performance,

"category\_performance": category\_performance,

"comparisons": {

"last\_month\_same\_day": round(last\_month\_same\_day, 2),

"last\_month\_mtd": round(last\_month\_mtd, 2),

"last\_year\_mtd": round(last\_year\_mtd, 2),

"today\_growth": round(today\_growth, 1),

"mtd\_growth\_month": round(mtd\_growth\_month, 1),

"mtd\_growth\_year": round(mtd\_growth\_year, 1)

}

}

finally:

db\_session.close()

def generate\_performance\_table(data, title, columns):

"""Generate HTML table for performance data"""

rows = ''.join(

f"<tr><td>{item[columns[0]]}</td>"

f"<td>{item[columns[1]]:,.2f}</td>"

f"<td>{item[columns[2]]}</td>"

f"<td>{item[columns[3]]:,.1f}%</td></tr>"

for item in data

)

return f"""

<h3>{title}</h3>

<table class="performance-table">

<tr>

<th>{columns[0].title()}</th>

<th>{columns[1].replace('\_', ' ').title()}</th>

<th>{columns[2].replace('\_', ' ').title()}</th>

<th>{columns[3].replace('\_', ' ').title()}</th>

</tr>

{rows}

</table>

"""

def send\_sales\_notification\_email():

print("📨 Running scheduled email job at", datetime.now())

try:

# Get sales data

notif = get\_sales\_data()

# Generate performance tables

regional\_table = generate\_performance\_table(

notif['regional\_performance'],

"Regional Performance (MTD)",

['region', 'total\_sales', 'loan\_count', 'budget\_percent']

)

category\_table = generate\_performance\_table(

notif['category\_performance'],

"Loan Category Analysis (MTD)",

['category', 'total\_sales', 'loan\_count', 'percent\_of\_total']

)

# Create HTML email content

html\_body = f"""

<!DOCTYPE html>

<html>

<head>

<style>

body {{ font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif; line-height: 1.6; }}

.report-card {{

max-width: 1000px;

margin: 20px auto;

border: 1px solid #e1e4e8;

border-radius: 10px;

overflow: hidden;

}}

.header {{

background: linear-gradient(135deg, #2c3e50, #1a2a3a);

color: white;

padding: 25px;

text-align: center;

}}

.metrics-container {{

display: grid;

grid-template-columns: repeat(auto-fit, minmax(250px, 1fr));

gap: 20px;

padding: 25px;

background-color: #f8f9fa;

}}

.metric-card {{

background: white;

border-radius: 8px;

padding: 20px;

box-shadow: 0 2px 10px rgba(0,0,0,0.05);

text-align: center;

}}

.metric-value {{

font-size: 28px;

font-weight: 700;

margin: 10px 0;

color: #2c3e50;

}}

.metric-label {{

color: #6c757d;

font-size: 14px;

}}

.growth-positive {{ color: #28a745; font-weight: bold; }}

.growth-negative {{ color: #dc3545; font-weight: bold; }}

.comparison {{

font-size: 13px;

margin-top: 5px;

}}

.performance-section {{

padding: 25px;

background: white;

}}

.performance-table {{

width: 100%;

border-collapse: collapse;

margin: 20px 0;

}}

.performance-table th {{

background-color: #f1f5f9;

padding: 12px 15px;

text-align: left;

font-weight: 600;

color: #2c3e50;

border-bottom: 2px solid #e1e4e8;

}}

.performance-table td {{

padding: 12px 15px;

border-bottom: 1px solid #e1e4e8;

}}

.performance-table tr:hover {{ background-color: #f8fafc; }}

.section-title {{

border-left: 4px solid #3498db;

padding-left: 15px;

margin-top: 30px;

color: #2c3e50;

}}

.footer {{

text-align: center;

padding: 20px;

background-color: #f1f5f9;

color: #6c757d;

font-size: 13px;

}}

.highlight {{

background-color: #e3f2fd;

padding: 3px 6px;

border-radius: 4px;

font-weight: 600;

}}

</style>

</head>

<body>

<div class="report-card">

<div class="header">

<h1>📊 Comprehensive Sales Report</h1>

<p>{notif['report\_date']} | Kwacha Financial Services</p>

</div>

<div class="metrics-container">

<div class="metric-card">

<div class="metric-label">Today's Sales</div>

<div class="metric-value">KES {notif['today\_sales']:,.2f}</div>

<div class="comparison">

{get\_growth\_indicator(notif['comparisons']['today\_growth'])}

vs {notif['comparisons']['last\_month\_same\_day']:,.2f} last month

</div>

</div>

<div class="metric-card">

<div class="metric-label">MTD Sales</div>

<div class="metric-value">KES {notif['mtd\_sales']:,.2f}</div>

<div class="comparison">

{get\_growth\_indicator(notif['comparisons']['mtd\_growth\_month'])}

vs last month<br>

{get\_growth\_indicator(notif['comparisons']['mtd\_growth\_year'])}

vs last year

</div>

</div>

<div class="metric-card">

<div class="metric-label">Budget Achievement</div>

<div class="metric-value">{notif['sales\_pct']:,.1f}%</div>

<div class="comparison">

Target: KES {notif['total\_budget']:,.2f}

</div>

</div>

<div class="metric-card">

<div class="metric-label">Time Progress</div>

<div class="metric-value">{notif['time\_pct']:,.1f}%</div>

<div class="comparison">

{notif['days\_passed']} of {notif['working\_days']} days

</div>

</div>

</div>

<div class="performance-section">

<h2 class="section-title">Performance Analysis</h2>

{regional\_table}

{category\_table}

<div class="historical-comparison">

<h3>Historical Comparison</h3>

<p>

<span class="highlight">Today vs Last Month:</span>

KES {notif['today\_sales']:,.2f} vs

KES {notif['comparisons']['last\_month\_same\_day']:,.2f} •

<span class="{get\_growth\_class(notif['comparisons']['today\_growth'])}">

{notif['comparisons']['today\_growth']:,.1f}%

</span>

</p>

<p>

<span class="highlight">MTD vs Last Month:</span>

KES {notif['mtd\_sales']:,.2f} vs

KES {notif['comparisons']['last\_month\_mtd']:,.2f} •

<span class="{get\_growth\_class(notif['comparisons']['mtd\_growth\_month'])}">

{notif['comparisons']['mtd\_growth\_month']:,.1f}%

</span>

</p>

<p>

<span class="highlight">MTD vs Last Year:</span>

KES {notif['mtd\_sales']:,.2f} vs

KES {notif['comparisons']['last\_year\_mtd']:,.2f} •

<span class="{get\_growth\_class(notif['comparisons']['mtd\_growth\_year'])}">

{notif['comparisons']['mtd\_growth\_year']:,.1f}%

</span>

</p>

</div>

</div>

<div class="footer">

<p>This is an automated report generated daily at 5:00 PM</p>

<p>Kwacha Financial Services • {datetime.now().year}</p>

</div>

</div>

</body>

</html>

"""

# Create and send message

msg = Message(

subject=f"📈 Advanced Sales Report: {notif['report\_date']}",

recipients=["sales-team@kwachafinancialservices.com", "management@kwachafinancialservices.com"],

html=html\_body

)

mail.send(msg)

print(f"✅ Comprehensive sales report sent at {datetime.now()}")

return True

except Exception as e:

print(f"❌ Failed to send sales report: {str(e)}")

return False

def get\_growth\_indicator(growth):

"""Return growth indicator with appropriate icon"""

if growth > 0:

return f"<span style='color:#28a745'>▲ {growth:,.1f}%</span>"

elif growth < 0:

return f"<span style='color:#dc3545'>▼ {growth:,.1f}%</span>"

return f"<span>{growth:,.1f}%</span>"

def get\_growth\_class(growth):

"""Return CSS class for growth value"""

if growth > 0:

return "growth-positive"

elif growth < 0:

return "growth-negative"

return ""

# Initialize scheduler

# Test route for manual triggering

@app.route('/test-report')

def test\_report():

result = send\_sales\_notification\_email()

return "Test report sent!" if result else "Failed to send test report"

@app.route('/dashboard/analytics')

def sales\_charts():

return render\_template("dashboard/sales\_charts.html", active\_tab="analytics")

from flask import jsonify

@app.route('/api/loans-by-region')

def api\_loans\_by\_region():

region = request.args.get('region')

district = request.args.get('district')

team\_leader = request.args.get('team\_lead')

query = db.session.query(

Agent.region,

func.sum(LoanApplication.loan\_amount).label('total\_loan')

).join(LoanApplication, LoanApplication.agent\_id == Agent.id)

if region:

query = query.filter(Agent.region == region)

if district:

query = query.filter(Agent.district == district)

if team\_leader:

query = query.filter(Agent.team\_leader == team\_leader)

query = query.group\_by(Agent.region)

data = query.all()

return jsonify({

"labels": [r.region for r in data],

"values": [float(r.total\_loan or 0) for r in data]

})

@app.route('/api/loans-by-district')

def api\_loans\_by\_district():

from sqlalchemy import func

data = (

db.session.query(

Agent.district,

func.count(LoanApplication.id).label('applications')

)

.join(LoanApplication, LoanApplication.agent\_id == Agent.id)

.group\_by(Agent.district)

.all()

)

labels = [row.district for row in data]

values = [row.applications for row in data]

return jsonify({"labels": labels, "values": values})

@app.route('/api/average-ticket-size')

def api\_average\_ticket\_size():

from sqlalchemy import func

data = (

db.session.query(

Agent.region,

func.avg(LoanApplication.loan\_amount).label('avg\_ticket')

)

.join(LoanApplication, LoanApplication.agent\_id == Agent.id)

.group\_by(Agent.region)

.all()

)

labels = [row.region for row in data]

values = [round(float(row.avg\_ticket or 0), 2) for row in data]

return jsonify({"labels": labels, "values": values})

@app.route('/debug/storage')

def debug\_storage():

return {

'UPLOAD\_FOLDER': app.config.get('UPLOAD\_FOLDER', ''),

'CWD': os.getcwd(),

'INSTANCE\_PATH': app.instance\_path,

'ABSOLUTE\_UPLOAD\_PATH': os.path.abspath(app.config.get('UPLOAD\_FOLDER', ''))

}

@app.route('/repair/documents', methods=['POST'])

@role\_required('admin')

def repair\_documents():

"""Fix document paths in database and move files to correct location"""

try:

updated\_count = 0

errors = []

base\_path = os.path.join(app.instance\_path, 'documents')

# Ensure the new location exists

os.makedirs(base\_path, exist\_ok=True)

# Get all documents

documents = Document.query.all()

for doc in documents:

try:

# Skip if path is already correct

if doc.path and 'instance/documents' in doc.path.replace('\\', '/'):

continue

# Handle missing paths

if not doc.path:

# Try to find by filename in the new location

possible\_path = os.path.join(base\_path, doc.filename)

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

doc.path = os.path.relpath(possible\_path, start=app.root\_path)

updated\_count += 1

continue

# Get current absolute path

if os.path.isabs(doc.path):

current\_path = doc.path

else:

current\_path = os.path.join(app.root\_path, doc.path)

# Skip if file doesn't exist

if not os.path.exists(current\_path):

errors.append(f"Document {doc.id}: File not found at {current\_path}")

continue

# New path in instance folder

new\_filename = f"{doc.customer\_id}\_{doc.id}\_{doc.filename}"

new\_path = os.path.join(base\_path, new\_filename)

# Move file

shutil.move(current\_path, new\_path)

# Update database

doc.path = os.path.relpath(new\_path, start=app.root\_path)

doc.filename = new\_filename # Update filename to include IDs

updated\_count += 1

except Exception as e:

errors.append(f"Document {doc.id}: {str(e)}")

db.session.commit()

return jsonify({

'status': 'success',

'updated\_count': updated\_count,

'total\_documents': len(documents),

'base\_path': base\_path,

'new\_location': os.path.relpath(base\_path, start=app.root\_path),

'errors': errors

})

except Exception as e:

return jsonify({

'status': 'error',

'message': str(e)

}), 500

@app.route('/repair/documents/page')

@role\_required('admin')

def repair\_documents\_page():

return render\_template('repair\_documents.html',

instance\_path=app.instance\_path,

root\_path=app.root\_path,

upload\_folder=app.config.get('UPLOAD\_FOLDER', ''),

absolute\_path=os.path.join(app.instance\_path, 'documents')

)

@app.route('/repair/paths')

@role\_required('admin')

def repair\_paths():

updated = convert\_legacy\_paths()

return jsonify({

'status': 'success',

'updated': updated,

'upload\_folder': app.config['UPLOAD\_FOLDER'],

'instance\_path': app.instance\_path

})

@app.route('/debug/documents/page')

@role\_required('admin')

def debug\_documents\_page():

"""Advanced document debugger page"""

return render\_template('debug\_documents.html')

@app.route('/verify/documents')

@role\_required('admin')

def verify\_documents():

"""Verify all document paths"""

docs = Document.query.all()

results = []

for doc in docs:

# Get absolute path

if doc.path:

if os.path.isabs(doc.path):

abs\_path = doc.path

else:

abs\_path = os.path.join(app.root\_path, doc.path)

exists = os.path.exists(abs\_path)

else:

abs\_path = None

exists = False

results.append({

'id': doc.id,

'filename': doc.filename,

'stored\_path': doc.path,

'absolute\_path': abs\_path,

'exists': exists,

'customer\_id': doc.customer\_id,

'loan\_id': doc.loan\_id

})

return jsonify({

'instance\_path': app.instance\_path,

'root\_path': app.root\_path,

'upload\_folder': app.config.get('UPLOAD\_FOLDER', ''),

'documents': results

})

@app.route('/debug/documents')

@role\_required('admin')

def debug\_documents\_api():

"""API endpoint for document debugging"""

customer\_id = request.args.get('customer\_id')

loan\_id = request.args.get('loan\_id')

results = {

'customer': None,

'loan': None,

'documents': [],

'instance\_path': app.instance\_path,

'root\_path': app.root\_path,

'upload\_folder': app.config.get('UPLOAD\_FOLDER', '')

}

# Get customer documents

if customer\_id:

try:

customer = Customer.query.get(customer\_id)

if customer:

results['customer'] = {

'id': customer.id,

'name': f"{customer.first\_name} {customer.last\_name}",

'file\_number': customer.file\_number

}

# Get all documents for customer

for doc in customer.customer\_documents:

abs\_path = doc.absolute\_path

results['documents'].append({

'id': doc.id,

'filename': doc.filename,

'filetype': doc.filetype,

'stored\_path': doc.path,

'absolute\_path': abs\_path,

'exists': os.path.exists(abs\_path) if abs\_path else False

})

except Exception as e:

results['error'] = str(e)

# Get loan documents

if loan\_id:

try:

loan = LoanApplication.query.get(loan\_id)

if loan:

results['loan'] = {

'id': loan.id,

'loan\_number': loan.loan\_number,

'status': loan.application\_status,

'customer\_id': loan.customer\_id

}

# Get all documents for loan

for doc in loan.documents:

abs\_path = doc.absolute\_path

results['documents'].append({

'id': doc.id,

'filename': doc.filename,

'filetype': doc.filetype,

'stored\_path': doc.path,

'absolute\_path': abs\_path,

'exists': os.path.exists(abs\_path) if abs\_path else False

})

except Exception as e:

results['error'] = str(e)

return jsonify(results)

@app.cli.command('repair-docs')

def repair\_docs\_command():

"""Command line document repair tool"""

from app import repair\_documents

with app.test\_request\_context():

response = repair\_documents()

data = response.get\_json()

print(f"Repair Results:")

print(f"Updated: {data.get('updated\_count', 0)} documents")

print(f"Total: {data.get('total\_documents', 0)} documents")

print(f"Base Path: {data.get('base\_path', '')}")

if errors := data.get('errors'):

print("\nErrors:")

for error in errors:

print(f" - {error}")

@app.cli.command('migrate-docs')

def migrate\_documents():

"""Migrate documents to proper location"""

import shutil

docs = Document.query.all()

new\_base = os.path.join(app.instance\_path, 'documents')

os.makedirs(new\_base, exist\_ok=True)

migrated = 0

for doc in docs:

if not doc.path:

continue

try:

# Get current absolute path

if os.path.isabs(doc.path):

current\_path = doc.path

else:

current\_path = os.path.join(app.root\_path, doc.path)

if not os.path.exists(current\_path):

continue

# New path in instance folder

new\_path = os.path.join(new\_base, os.path.basename(doc.path))

# Skip if already in correct location

if os.path.normpath(current\_path) == os.path.normpath(new\_path):

continue

# Move file

shutil.move(current\_path, new\_path)

# Update database with relative path

doc.path = os.path.relpath(new\_path, start=app.root\_path)

migrated += 1

except Exception as e:

app.logger.error(f"Error migrating document {doc.id}: {str(e)}")

db.session.commit()

print(f"Migrated {migrated} documents to {new\_base}")

@app.route('/debug/documents')

@role\_required('admin')

def debug\_documents():

"""Advanced document debugging page"""

return render\_template('debug\_documents.html')

from app import app, db

import smtplib

def test\_smtp\_connection():

try:

with smtplib.SMTP\_SSL('mail.kwachafinancialservices.com', 465) as server:

server.login(

'alfred@kwachafinancialservices.com',

'~hHjb;m{urRh'

)

print("✅ SMTP login successful!")

return True

except Exception as e:

print(f"❌ SMTP connection failed: {str(e)}")

return False

def initialize\_roles\_permissions():

with app.app\_context():

create\_roles\_and\_permissions()

@app.route('/db-check')

def db\_check():

try:

db.engine.connect()

return "Database connection successful!", 200

except Exception as e:

return f"Database connection failed: {str(e)}", 500

def deploy():

with app.app\_context():

# Apply any pending migrations

upgrade()

@app.route('/<path:path>')

def catch\_all(path):

return f"404: The URL /{path} was not found.", 404

if \_\_name\_\_ == '\_\_main\_\_':

with app.app\_context():

print("\n=== Registered Routes ===")

for rule in app.url\_map.iter\_rules():

methods = ','.join(rule.methods - {'HEAD', 'OPTIONS'})

print(f"{rule.endpoint:30} | {methods:10} | {rule.rule}")

print("=========================\n")

try:

deploy()

except Exception as e:

print(f"Migration failed: {e}", file=sys.stderr)

sys.exit(1)

initialize\_roles\_permissions()

if os.environ.get("WERKZEUG\_RUN\_MAIN") == "true":

from apscheduler.schedulers.background import BackgroundScheduler

scheduler = BackgroundScheduler(daemon=True)

scheduler.add\_job(

func=send\_sales\_notification\_email,

trigger='interval',

minutes=1

)

scheduler.start()

print("⏰ Scheduler started")

import atexit

atexit.register(lambda: scheduler.shutdown())

app.run(debug=True, port=5000)