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 datetime import datetime

from sqlalchemy import func

import calendar

from sqlalchemy import event

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

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

# 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)

# 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)

# 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%"

@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"

@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

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\_\_)

# 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['MAIL\_SERVER'] = 'smtp.gmail.com'

app.config['MAIL\_PORT'] = 587

app.config['MAIL\_USE\_TLS'] = True

app.config['MAIL\_USERNAME'] = 'youremail@example.com' # ✅ Replace!

app.config['MAIL\_PASSWORD'] = 'your\_app\_password\_here' # ✅ Replace!

app.config['MAIL\_DEFAULT\_SENDER'] = 'youremail@example.com'

# 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'}

# 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.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

@app.route("/documents/id/<int:doc\_id>")

@login\_required

def serve\_document(doc\_id: int):

"""

Streams any uploaded document (image, PDF, DOCX …) back to the browser.

Large files are forced to download (`as\_attachment=True`) so Office apps

/ photo viewers take over.

"""

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

try:

# `download\_name` keeps the original filename for the user

return send\_file(

doc.path,

as\_attachment=doc.filetype not in ("id\_front", "id\_back", "live\_photo", "payslip", "photo"),

download\_name=doc.filename,

max\_age=0 # don’t cache sensitive docs

)

except FileNotFoundError:

raise NotFound("File is missing on the server.")

# ---------------- 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'

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 Customer(db.Model):

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

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)

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

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 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'

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)

parent\_loan\_id = db.Column(

db.Integer,

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

nullable=True

)

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

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'

)

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):

"""Recalculate loan balances after any payment"""

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

loan\_amount = self.loan\_amount or 0

# Capitalized loan amount

capitalized = (

loan\_amount

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

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

+ config.get('crb', 0)

)

# Paid principal — successful + completed only

paid\_principal = sum(

(alloc.principal or 0.0)

for p in self.payments

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

for alloc in p.allocations

)

# Paid settlement interest (if any)

paid\_settlement\_interest = sum(

(getattr(p.allocations, 'settlement\_interest', 0.0) or 0.0)

for p in self.payments

if p.allocations and p.status in ['successful', 'completed']

)

# Compute current balance (capitalized - paid principal)

computed\_balance = max(round(capitalized - paid\_principal, 2), 0)

# Business logic by loan\_state

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

if not self.settlement\_balance or self.settlement\_balance == 0.0:

self.settlement\_balance = round(paid\_principal + paid\_settlement\_interest, 2)

self.current\_balance = 0.0

self.top\_up\_balance = 0.0

elif self.loan\_state in ('write\_off', 'insurance'):

self.current\_balance = 0.0

self.top\_up\_balance = 0.0

self.settlement\_balance = 0.0

else:

self.current\_balance = computed\_balance

self.top\_up\_balance = computed\_balance + self.top\_up\_interest

if not self.settlement\_balance or self.loan\_state == 'active':

self.settlement\_balance = computed\_balance + self.settlement\_interest

# Log using app.logger instead of print

app.logger.info(

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

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

f"paid\_settlement\_interest={paid\_settlement\_interest}, "

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

f"top\_up\_balance={self.top\_up\_balance}, "

f"settlement\_balance={self.settlement\_balance}, "

f"loan\_state={self.loan\_state}" # ✅ Corrected

)

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() # add .filter\_by(vote\_id=…) if needed

if rec and rec.cutoff\_dt:

return rec.cutoff\_dt.day # e.g. 15

return None # not configured # treat 0 / NULL as “unset”

# ------------------------------------------------------------------

# 2. First-due-date calculator

# ------------------------------------------------------------------

def get\_first\_due\_date(self):

"""

Determine the first instalment due date:

• Always lands on the 25th (or the month’s last day if <25 days long).

• Logic per product:

- Private sector : 25th same month (≤15th) else next month

- Civil servant : baseline next-month 25th; if admin cut-off set

and disbursement past it → push +1 month

- SME / other : always next-month 25th

"""

import calendar

from datetime import date as \_date

from dateutil.relativedelta import relativedelta

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

due\_day = 25

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

# ── Private sector ───────────────────────────────────────────────

if cat == "private\_sector":

target\_month = disb\_date if disb\_date.day <= 15 \

else disb\_date + relativedelta(months=1)

# ── Civil servant ────────────────────────────────────────────────

elif cat == "civil\_servant":

cutoff\_day = self.get\_cutoff\_day\_for\_civil\_servant() # int | None

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

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

target\_month += relativedelta(months=1)

# ── SME & all other products ─────────────────────────────────────

else:

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

# Clamp to 25th or last calendar day

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):

"""

Build/replace this loan’s full amortised schedule.

• Uses capitalised balance (principal + orig + ins + CRB)

• Keeps collection-fee as flat monthly surcharge (loan\_amount \* collection%)

• Persists remaining\_balance after each instalment

"""

# ---- 1. Wipe existing rows ---------------------------------

for sched in self.repayment\_schedules:

db.session.delete(sched)

# ---- 2. Pricing config -------------------------------------

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

if not config:

return # nothing to calculate

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 # ← main balance

monthly\_rate = config.get('rate', 0) # decimal (e.g. 0.035)

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

term = self.term\_months or 0

if term <= 0:

return

# ---- 3. First due-date via category rules ------------------

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

# ---- 4. Constant (annuity) payment on capitalised amount ---

if monthly\_rate > 0:

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

annuity\_princ\_int = capitalised \* fac # principal+interest only

else:

annuity\_princ\_int = capitalised / term

remaining = capitalised

# ---- 5. Create rows ----------------------------------------

for i in range(term):

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

interest = remaining \* monthly\_rate

principal = annuity\_princ\_int - interest

# last instalment: sweep residual rounding

if i == term - 1:

principal = remaining

interest = annuity\_princ\_int - principal # recompute

annuity\_princ\_int= principal + interest # just to stay consistent

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)

db.session.commit()

from datetime import datetime

from dateutil.relativedelta import relativedelta # already using

import calendar

# … other methods in LoanApplication …

# ────────────────────────────────────────────────────────────────

def allocate\_payment(self, payment):

"""

Allocate a payment to the loan's repayment schedules or balances:

- Internal/top-up → prioritize top\_up\_balance, interest.

- Settlement → pay current + settlement interest.

- Write-off/insurance: do not allow further allocation.

- Normal: allocate to fees, interest, principal.

- Excess → record loan credit.

"""

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

app.logger.warning(f"[{self.loan\_number}] Cannot allocate payment to closed loan state: {self.loan\_state}")

return

remaining = payment.amount

schedule\_updated = False

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

for schedule in schedules:

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

continue

fee\_due = schedule.fees\_due

interest\_due = schedule.interest\_due

principal\_due = schedule.principal\_due

alloc\_fees = min(fee\_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

remaining -= alloc\_principal

if schedule.expected\_amount <= (

schedule.paid\_fees + schedule.paid\_interest + schedule.paid\_principal

):

schedule.status = "paid"

schedule\_updated = True

app.logger.info(

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

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

)

if remaining <= 0:

break

if schedule\_updated:

self.recalculate\_balance()

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 {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'

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

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))

# 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)

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

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

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

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

schedule\_id = db.Column(db.Integer, db.ForeignKey('repayment\_schedules.id'), nullable=True) # Keep this once only

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

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

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

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

# )

# Relationships

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

schedule = db.relationship(

'RepaymentSchedule',

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

foreign\_keys=[schedule\_id],

backref='allocations'

)

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', backref=db.backref('documents', lazy=True))

loan = db.relationship('LoanApplication', backref=db.backref('documents', lazy=True))

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

\_\_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

# Add after models

from datetime import datetime

from app import db

from app import PaymentAllocation, RepaymentSchedule, LoanCredit

class PaymentAllocator:

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

if not payment or not payment.loan:

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

self.payment = payment

self.loan = payment.loan

self.amount = payment.amount

self.allocations = []

def process(self):

self.\_clear\_existing\_allocations()

self.\_allocate\_to\_schedules()

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

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

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()

))

# Schedule status update

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

schedule.status = 'paid'

elif schedule.paid\_amount > 0:

schedule.status = 'partial'

else:

schedule.status = 'pending'

if remaining <= 0:

break

self.amount = remaining

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

expected\_due = self.loan.expected\_payment\_due()

if self.amount > 0:

# Overpayment becomes loan credit

credit = LoanCredit(

loan\_id=self.loan.id,

amount=round(self.amount, 2),

created\_at=datetime.utcnow()

)

db.session.add(credit)

elif self.payment.amount < expected\_due:

# Underpayment triggers arrears

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

def \_update\_loan\_status(self):

self.loan.recalculate\_balance()

if self.loan.total\_arrears > 60:

self.loan.performance\_status = 'default'

elif self.loan.total\_arrears > 30:

self.loan.performance\_status = 'delinquent'

else:

self.loan.performance\_status = 'performing'

def \_persist\_allocations(self):

for alloc in self.allocations:

db.session.add(alloc)

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

# 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, document\_type):

if file and allowed\_file(file.filename):

# Create unique filename

timestamp = datetime.now().strftime("%Y%m%d\_%H%M%S")

ext = file.filename.rsplit('.', 1)[1].lower()

filename = f"{customer\_id}\_{document\_type}\_{timestamp}.{ext}"

filename = secure\_filename(filename)

# Ensure upload directory exists

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

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

# Save file

file.save(filepath)

return filename, filepath

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 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'))

@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/<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()

loans = (

LoanApplication.query

.options(joinedload(LoanApplication.payments))

.filter(LoanApplication.customer\_id == customer.id)

.all()

)

app.logger.info(f"Found {len(loans)} loans for customer {file\_number}")

app.logger.info(f"Customer ID: {customer.id}")

app.logger.info(f"Found Loans: {[(loan.loan\_number, loan.loan\_state) for loan in loans]}")

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 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)

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),

}

for loan in loans:

balances = calculate\_balances(loan)

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,

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)

)

running\_balance = capitalized\_amount

payments\_made = 0

for payment in sorted(loan.payments, key=lambda p: p.created\_at):

for allocation in payment.allocations:

if allocation.principal:

running\_balance -= allocation.principal

payments\_made += 1

current\_balance = max(round(running\_balance, 2), 0.00)

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

term = loan.term\_months

remaining\_term = term - payments\_made

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

top\_up\_balance = round(current\_balance + calculate\_projected\_interest(3), 2)

settlement\_balance = round(current\_balance + calculate\_projected\_interest(6), 2)

statement = []

running\_balance\_display = capitalized\_amount

for payment in sorted(loan.payments, key=lambda p: p.created\_at):

for allocation in payment.allocations:

running\_balance\_display -= allocation.principal or 0

allocated\_total = (allocation.principal or 0) + (allocation.interest or 0) + (allocation.fees or 0)

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': allocation.principal,

'interest': allocation.interest,

'collection\_fees': allocation.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 for p in loan.payments for a in p.allocations),

'interest': sum(a.interest for p in loan.payments for a in p.allocations),

'fees': sum(a.fees for p in loan.payments for a in p.allocations)

}

print(f"current\_balance={current\_balance}, top\_up\_balance={top\_up\_balance}, settlement\_balance={settlement\_balance}")

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>')

def view\_document(doc\_id):

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

directory = os.path.dirname(doc.path)

filename = os.path.basename(doc.path)

return send\_from\_directory(directory, filename)

@app.route('/register', methods=['GET', 'POST'])

@role\_required("sales\_ops", "admin")

def register\_customer\_debug():

if request.method == 'POST':

file = request.files.get('csv\_file')

if file and file.filename.endswith('.csv'):

# Handle CSV upload

try:

stream = io.StringIO(file.stream.read().decode("UTF8"), newline=None)

csv\_input = csv.DictReader(stream)

for row in csv\_input:

process\_customer\_registration(row)

flash("CSV upload processed successfully.", "success")

except Exception as e:

db.session.rollback()

flash(f"CSV upload failed: {str(e)}", "danger")

return redirect(url\_for('register\_customer\_debug'))

else:

# Manual entry

try:

process\_customer\_registration(request.form)

flash("Customer and loan registered successfully.", "success")

except Exception as e:

db.session.rollback()

flash(str(e), "danger")

return redirect(url\_for('register\_customer\_debug'))

return render\_template('register\_customer\_debug.html')

def process\_customer\_registration(data):

try:

loan\_amount = float(data.get('loan\_amount', 0))

if loan\_amount < 0:

raise ValueError("Loan amount cannot be negative")

except (TypeError, ValueError):

loan\_amount = 0.0

term\_months = int(data['loan\_term'])

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.")

prefix = category\_info['prefix']

category = category\_info['label']

config = get\_pricing\_config(category, 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.")

# Validate and parse date joined

date\_joined = data.get("date\_joined")

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 is eligible for voluntary retirement (20+ years in service)", "warning")

else:

years\_in\_service = None

# Check for retirement age at loan completion

if age + (term\_months // 12) > 60:

raise Exception("Loan tenure will exceed retirement age (60 years).")

# Pricing

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

annuity\_factor = (config['rate'] \* (1 + config['rate']) \*\* term\_months) / \

((1 + config['rate']) \*\* term\_months - 1)

monthly\_instalment = (capitalized\_amount \* annuity\_factor) + 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 file number

now = datetime.utcnow()

customer\_count = db.session.query(Customer).count()

file\_sequence = str(customer\_count + 1).zfill(6)

file\_number = f"{now.year}{str(now.month).zfill(2)}{file\_sequence}"

# Create customer

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,

maker\_id=1

)

db.session.add(customer)

db.session.flush()

# Upload documents

document\_types = {

'national\_id\_front': 'id\_front',

'form': 'form',

'customer\_photo': 'photo',

'payslip': 'payslip',

'bank\_statement': 'bank\_statement',

'letter\_of\_undertaking': 'undertaking\_letter'

}

for form\_field, doc\_type in document\_types.items():

file = request.files.get(form\_field)

if file:

filename, filepath = save\_document(file, customer.id, doc\_type)

if filename:

document = Document(

customer\_id=customer.id,

filename=filename,

filetype=doc\_type,

path=filepath

)

db.session.add(document)

# Loan logic

tenure = str(term\_months).zfill(2)

loan\_count = db.session.query(LoanApplication).count()

loan\_sequence = str(loan\_count + 1).zfill(6)

loan\_number = f"{prefix}{tenure}{loan\_sequence}"

previous\_loan = LoanApplication.query.filter\_by(customer\_id=customer.id).order\_by(LoanApplication.id.desc()).first()

top\_up\_balance = 0

if previous\_loan and previous\_loan.loan\_state in ['active', 'closed']:

balances = calculate\_balances(previous\_loan)

top\_up\_balance = balances.get('top\_up\_balance', 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\_instalment, 2),

total\_repayment=round(monthly\_instalment \* term\_months, 2),

effective\_rate=calculate\_eir(loan\_amount, term\_months, config),

category=category,

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,

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()

disbursement = Disbursement(

loan\_id=loan.id,

amount=cash\_to\_client,

method='bank',

status='pending',

reference=f"Initial disbursement for {loan.loan\_number}"

)

db.session.add(disbursement)

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

if not customer.loans:

loan = LoanApplication(

customer\_id=customer.id,

loan\_amount=customer.amount\_requested or 0.0,

loan\_category="SME",

status='pending',

loan\_state='Active',

application\_status='awaiting\_approval',

vote\_id=vote.id

)

db.session.add(loan)

approved\_count += 1

else:

for loan in customer.loans:

loan.status = 'pending'

loan.vote\_id = vote.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 request handling

unapproved\_customers = Customer.query.filter\_by(is\_approved\_for\_creation=False).all()

# Get all active votes (fallback to all votes if none active)

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()

# Normalize amount\_requested

for customer in unapproved\_customers:

try:

customer.amount\_requested = float(customer.amount\_requested)

except (TypeError, ValueError):

customer.amount\_requested = 0.0

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')

def view\_loans():

loans = (

LoanApplication.query

.join(Customer)

.filter(Customer.is\_approved\_for\_creation.is\_(True))

.filter(LoanApplication.status.in\_(['pending', 'approved']))

.options(joinedload(LoanApplication.documents)) # <-- eager‑load docs

.add\_entity(Customer)

.all()

)

processed\_loans = [

{

"loan": loan\_app,

"customer": customer,

"current\_balance": loan\_app.balance or 0.0,

}

for loan\_app, customer in loans

]

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()

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):

# Same calculation as in the frontend

monthly\_rate = config['rate']

balance = principal

total\_interest = 0

total\_balances = 0

for \_ in range(months):

interest = balance \* monthly\_rate

total\_interest += interest

total\_balances += balance

balance -= principal / months # Simple principal reduction

fees = (principal \* (config['origination'] + config['insurance'])) + config['crb']

average\_balance = total\_balances / months

return round(((total\_interest + fees) / average\_balance) \* (12 / months) \* 100, 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()

processed\_loans = []

for loan in all\_loans:

customer = loan.customer

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

loan\_amount = loan.loan\_amount or 0

capitalized = (

loan\_amount

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

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

+ config.get('crb', 0)

)

total\_principal\_paid = sum(

(alloc.principal or 0)

for p in loan.payments

for alloc in p.allocations

)

remaining\_balance = capitalized - (total\_principal\_paid or 0)

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\_amount,

'term': loan.term\_months,

'category': loan.category,

'monthly\_instalment': loan.monthly\_instalment,

'total\_repayment': loan.total\_repayment,

'balance': round(remaining\_balance, 2),

'disbursed': loan.disbursed,

'collection\_fee': loan\_amount \* config.get('collection', 0)

},

'fees': {

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

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

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

'total': (

config.get('crb', 0)

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

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

)

}

})

# 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 for full (non-AJAX) page

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)

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

if not category\_info or not config:

raise Exception("Invalid loan category or term.")

# --- 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, loan)

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

# Monthly repayment

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)

def projected\_interest(months):

temp\_balance = current\_balance

interest\_total = 0.0

for \_ in range(min(months, remaining\_term)):

if temp\_balance <= 0:

break

interest = temp\_balance \* rate

principal = min(annuity - interest, temp\_balance)

interest\_total += interest

temp\_balance -= principal

return round(interest\_total, 2)

top\_up\_interest = projected\_interest(3)

return round(current\_balance + top\_up\_interest, 2), top\_up\_interest

top\_up\_balance, top\_up\_interest = calculate\_topup\_balance(base\_loan)

app.logger.debug(f"[DEBUG] Base loan #{base\_loan.loan\_number} → Top-up balance: {top\_up\_balance}, Interest: {top\_up\_interest}")

# --- Validate top-up eligibility

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}"

# --- 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,

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')

]:

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:

payment = Payment(

loan\_id=base\_loan.id,

amount=top\_up\_balance,

method='internal\_topup',

status='successful',

reference=f"Top-Up from {loan\_number}",

created\_at=datetime.utcnow()

)

db.session.add(payment)

db.session.flush()

# Allocate & Recalculate

base\_loan.allocate\_payment(payment)

base\_loan.recalculate\_balance()

# Mark loan as closed

base\_loan.status = 'closed'

base\_loan.loan\_state = 'topped\_up'

base\_loan.closure\_type = 'topup'

base\_loan.closure\_date = datetime.utcnow()

return topup\_loan

@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)

def process\_additional\_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) as e:

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)

# REPLACEMENT: Use get\_pricing\_config instead of PRICING

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

if not category\_info or not config:

raise Exception("Invalid category or term.")

# --- Fees (using the new config)

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

# REPLACEMENT: Use config from get\_pricing\_config

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}"

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),

# REPLACEMENT: Use the new config

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,

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,

disbursement\_date=None,

cash\_to\_client=new\_amount,

# Store applied rates for future reference

applied\_interest\_rate=config['rate'],

applied\_collection\_fee=config['collection']

)

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')

]:

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=datetime.utcnow()

)

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)

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

from app import app, db

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()

if \_\_name\_\_ == '\_\_main\_\_':

try:

deploy()

except Exception as e:

print(f"Migration failed: {e}", file=sys.stderr)

sys.exit(1)

initialize\_roles\_permissions()

app.run(debug=True)