#!/usr/bin/env bash

set -euo pipefail

export FLASK\_ENV=production

echo "🚀  Starting deployment script…"

###############################################################################

# 0. Sanity‑check required secrets

###############################################################################

if [[ -z "${ADMIN\_EMAIL:-}" || -z "${ADMIN\_PASSWORD:-}" ]]; then

  echo "❌  ADMIN\_EMAIL and ADMIN\_PASSWORD must be set" >&2

  exit 1

fi

###############################################################################

# 1. Make sure the critical columns exist (works even on Render free tier)

###############################################################################

echo "🆘  Ensuring critical columns exist in loan\_applications…"

python - <<'PY'

from app import app, db

from sqlalchemy import text

CRITICAL\_COLUMNS = {

    'current\_balance':      'NUMERIC(12, 2) DEFAULT 0.0',

    'top\_up\_balance':       'NUMERIC(12, 2)',

    'settlement\_balance':   'NUMERIC(12, 2)',

    'settlement\_type':      'VARCHAR(50)',

    'settling\_institution': 'VARCHAR(255)',

    'settlement\_reason':    'TEXT',

    'parent\_loan\_id':       'INTEGER',

}

with app.app\_context():

    conn = db.engine.connect()

    for col, dtype in CRITICAL\_COLUMNS.items():

        print(f"🔧  ALTER TABLE loan\_applications ADD COLUMN IF NOT EXISTS {col} {dtype}")

        try:

            conn.execute(text(f"""

                ALTER TABLE loan\_applications

                ADD COLUMN IF NOT EXISTS {col} {dtype}

            """))

            print(f"✅  Column {col} ready")

        except Exception as e:

            # Never fail hard – just warn and continue

            print(f"⚠️  Could not add {col}: {e}")

PY

echo "✅  Column check complete"

###############################################################################

# 2. Robust Alembic migration handling (recovers from lost revision)

###############################################################################

echo "🗄️  Applying database migrations…"

if flask db upgrade; then

  echo "✅  Alembic upgraded cleanly"

else

  echo "⚠️  Alembic upgrade failed – starting recovery"

  # Get the actual head revision from the repo

  HEAD\_REV="$(alembic heads | awk 'NR==1{print $1}')"

  echo "🔎  Repo head revision is ${HEAD\_REV}"

  # Force‑stamp DB with that revision

  python - <<PY

import os, sys

from sqlalchemy import create\_engine, text

db\_url = os.environ["DATABASE\_URL"].replace("postgres://", "postgresql://", 1)

engine  = create\_engine(db\_url, isolation\_level="AUTOCOMMIT")

with engine.connect() as conn:

    conn.execute(text("""

        CREATE TABLE IF NOT EXISTS alembic\_version (

            version\_num VARCHAR(32) NOT NULL

        )

    """))

    conn.execute(text("DELETE FROM alembic\_version"))

    conn.execute(text("INSERT INTO alembic\_version (version\_num) VALUES (:rev)"),

                 {"rev": "${HEAD\_REV}"})

    print(f"✅  Force‑stamped alembic\_version with ${HEAD\_REV}")

PY

  # Retry the upgrade one more time; if it still fails we abort

  if flask db upgrade; then

    echo "✅  Upgrade succeeded after recovery"

  else

    echo "❌  Final Alembic upgrade failed – aborting deploy" >&2

    exit 1

  fi

fi

###############################################################################

# 3. Seed / update RBAC data

###############################################################################

echo "👥  Seeding roles / permissions…"

python - <<'PY'

from app import app, initialize\_roles\_permissions

with app.app\_context():

    initialize\_roles\_permissions()

    print("✅  RBAC initialised")

PY

###############################################################################

# 4. Ensure admin user exists

###############################################################################

echo "👑  Ensuring admin account…"

python - <<'PY'

from app import app, db, User

from werkzeug.security import generate\_password\_hash

import os, time

email    = os.environ["ADMIN\_EMAIL"]

password = os.environ["ADMIN\_PASSWORD"]

with app.app\_context():

    admin = User.query.filter\_by(email=email).first()

    if admin:

        print("✅  Admin already present")

    else:

        username = "admin"

        if User.query.filter\_by(username=username).first():

            username = f"admin\_{int(time.time())}"

        admin = User(username=username,

                     email=email,

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

        db.session.add(admin)

        db.session.commit()

        print(f"✅  Created admin user {email} ({username})")

PY

###############################################################################

# 5. Launch Gunicorn

###############################################################################

echo "🚀  Launching Gunicorn…"

exec gunicorn --workers 4 --bind 0.0.0.0:${PORT:-8000} app:app

import os

from flask\_migrate import Migrate

from app import app, db  # ⬅️ Import the existing app and db directly

# Initialize migrate with the already created app

migrate = Migrate(app, db)

def reset\_migrations():

    # Backup existing migrations

    if os.path.exists('migrations'):

        os.system('rm -rf migrations\_backup')

        os.system('mkdir migrations\_backup')

        os.system('cp -r migrations/versions migrations\_backup/')

    # Reset migrations

    os.system('rm -rf migrations')

    os.system('flask db init')

    os.system('flask db migrate -m "Reset migration history"')

    os.system('flask db stamp head')

    # Create marker file

    with open('.migration\_reset\_complete', 'w') as f:

        f.write('Migration reset completed')

    print("✅ Migration history reset complete")

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

    with app.app\_context():

        reset\_migrations()

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

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

class SettlementTypeEnum(enum.Enum):

self\_settlement = "self"

third\_party = "third\_party"

load\_dotenv()

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

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

os.makedirs(log\_dir)

# Initialize the Flask application

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

app.secret\_key = 'supersecretkey'

app.logger.setLevel(logging.INFO)

handler = RotatingFileHandler(

'app.log',

maxBytes=1024 \* 1024,

backupCount=5,

encoding='utf-8'

)

handler.setFormatter(logging.Formatter(

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

))

app.logger.addHandler(handler)

log\_handler = RotatingFileHandler('logs/loan\_app.log', maxBytes=512000, backupCount=3)

log\_handler.setLevel(logging.DEBUG) # Capture all logs

formatter = logging.Formatter(

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

)

log\_handler.setFormatter(formatter)

app.logger.addHandler(log\_handler)

app.logger.setLevel(logging.DEBUG)

if os.name == 'nt':

import ctypes

ctypes.windll.kernel32.SetConsoleCP(65001)

ctypes.windll.kernel32.SetConsoleOutputCP(65001)

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

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

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

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

console\_handler = logging.StreamHandler()

console\_handler.setLevel(logging.DEBUG)

console\_handler.setFormatter(formatter)

app.logger.addHandler(console\_handler)

@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

import logging

from logging.handlers import RotatingFileHandler

import os

from flask import Flask

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

# Configure logging

def setup\_logging():

# Create logger

logger = logging.getLogger('LoanAppLogger')

logger.setLevel(logging.INFO)

# Create formatter

formatter = logging.Formatter(

'%(asctime)s - %(name)s - %(levelname)s - %(message)s [%(pathname)s:%(lineno)d]'

)

# Rotating file handler

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

file\_handler = RotatingFileHandler(

filename=log\_file,

maxBytes=512000,

backupCount=3,

encoding='utf-8'

)

file\_handler.setFormatter(formatter)

logger.addHandler(file\_handler)

# Add console handler for development

if app.debug:

console\_handler = logging.StreamHandler()

console\_handler.setFormatter(formatter)

logger.addHandler(console\_handler)

# Initialize logging

setup\_logging()

@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 ""

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

}

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 ----------------

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)

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

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

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

@property

def full\_name(self):

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

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)

parent\_loan\_id = db.Column(

db.Integer,

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

nullable=True

)

topups = db.relationship(

'LoanApplication',

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

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

cascade='all, delete-orphan'

)

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 not isinstance(value, SettlementTypeEnum):

raise ValueError("Invalid settlement type")

return value

# 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}>'

def generate\_repayment\_schedule(self):

config = PRICING.get(self.term\_months or 0, {})

if not config:

raise ValueError("No pricing config for loan term.")

loan\_amount = self.loan\_amount or 0

term = self.term\_months or 0

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

# Fees (capitalized into loan)

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

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

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

capitalized = loan\_amount + origination + insurance + crb

# Monthly collection fee

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

# Annuity (fixed monthly P+I)

annuity\_factor = (rate \* (1 + rate) \*\* term) / ((1 + rate) \*\* term - 1)

monthly\_p\_and\_i = capitalized \* annuity\_factor

# Start repayment from disbursement date

start\_date = self.disbursement\_date or datetime.utcnow().date()

remaining\_principal = capitalized

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

due\_date = start\_date + relativedelta(months=i)

interest = remaining\_principal \* rate

principal = monthly\_p\_and\_i - interest

if i == term:

principal = remaining\_principal

monthly\_p\_and\_i = principal + interest

remaining\_principal -= principal

schedule = RepaymentSchedule(

loan\_id=self.id,

due\_date=due\_date,

expected\_amount=round(monthly\_p\_and\_i + collection\_fee, 2),

expected\_principal=round(principal, 2),

expected\_interest=round(interest, 2),

expected\_fees=round(collection\_fee, 2),

status='due'

)

@property

def computed\_current\_balance(self):

from datetime import datetime

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

return 0.0

config = PRICING.get(self.term\_months, {})

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 = PRICING.get(self.term\_months)

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 for p in self.payments if p.allocation for alloc in [p.allocation]

)

return round(capitalized - paid\_principal, 2)

@property

def calculated\_balance(self):

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

return 0.0

config = PRICING.get(self.term\_months)

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 for p in self.payments if p.allocation for alloc in [p.allocation]

)

return round(capitalized - paid\_principal, 2)

def recalculate\_balance(self):

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

config = PRICING.get(self.term\_months, {})

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(

(p.allocation.principal or 0.0) for p in self.payments

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

)

# Paid settlement interest (if any)

paid\_settlement\_interest = sum(

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

for p in self.payments

if p.allocation 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':

# For settled\_client:

# → Keep existing settlement\_balance (do not recompute from capitalized!)

# → If settlement\_balance was 0 (fallback), compute based on what was paid

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

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

# Always clear current + top\_up balances for settled\_client

self.current\_balance = 0.0

self.top\_up\_balance = 0.0

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

# For hard closures — everything zeroed

self.current\_balance = 0.0

self.top\_up\_balance = 0.0

self.settlement\_balance = 0.0

else:

# For active and other loans — recompute fully

self.current\_balance = computed\_balance

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

# Only update settlement\_balance if not already locked

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

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

# Optional debug log (keep this for now to verify correct flow!)

print(

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}"

)

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'))

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))

loan = db.relationship(

"LoanApplication",

back\_populates="repayment\_schedules",

foreign\_keys=[loan\_id]

)

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')

allocation = db.relationship('PaymentAllocation',

back\_populates='payment',

uselist=False,

cascade='all, delete-orphan')

from sqlalchemy.orm.session import object\_session

# or from wherever you define it

from sqlalchemy.orm.session import object\_session

from sqlalchemy import event

@event.listens\_for(Payment, 'after\_insert')

def allocate\_payment\_listener(mapper, connection, target):

session = object\_session(target)

if target.method in ('settlement', 'internal\_topup', 'internal\_settlement'):

app.logger.info(f"Skipping allocation for {target.method} payment")

return

if not session:

raise RuntimeError("No active session found for payment allocation.")

# Skip auto allocation for internal operations

if target.method in ('internal\_topup', 'internal\_settlement'):

app.logger.info(f"Skipping allocation for {target.method} payment ID {target.id}")

return

# Avoid duplicate allocations

existing = session.query(PaymentAllocation).filter\_by(payment\_id=target.id).first()

if existing:

return

# Fetch associated loan

loan = session.query(LoanApplication).get(target.loan\_id)

if not loan:

raise ValueError(f"Loan not found for payment ID {target.id} with loan\_id {target.loan\_id}")

# Allocate payment as normal

allocate\_payment(target, loan)

def allocate\_payment(payment: Payment, loan: LoanApplication):

session = object\_session(payment)

if payment.method == 'settlement':

return

if not session:

raise RuntimeError("No active session for allocation")

# Skip if already allocated

if session.query(PaymentAllocation).filter\_by(payment\_id=payment.id).first():

return

config = PRICING.get(loan.term\_months)

if not config:

raise ValueError(f"No pricing config for {loan.term\_months} months")

loan\_amount = loan.loan\_amount or 0

capitalized = loan\_amount + (loan\_amount \* config['origination']) + (loan\_amount \* config['insurance']) + config['crb']

paid\_principal = sum(p.allocation.principal for p in loan.payments if p.allocation and p.id != payment.id)

remaining\_principal = max(capitalized - paid\_principal, 0)

rate = config.get('rate', 0.0)

interest = round(remaining\_principal \* rate, 2)

method = (payment.method or '').lower()

allocation = PaymentAllocation(payment\_id=payment.id, principal=0.0, interest=0.0, fees=0.0)

if method in ('internal\_topup', 'internal\_settlement'):

# Internal settlement: no fees, allocate to principal + capture interest to special fields

interest = min(interest, payment.amount)

principal = round(payment.amount - interest, 2)

# Cap principal

principal = min(principal, remaining\_principal)

allocation.principal = principal

allocation.interest = 0.0

allocation.fees = 0.0

if loan.closure\_type == 'topup':

loan.top\_up\_interest = interest

elif loan.closure\_type == 'settlement':

loan.settlement\_interest = interest

else:

# Normal repayment

collection\_fee = round(loan\_amount \* config['collection'], 2)

fees = min(collection\_fee, payment.amount)

remaining = payment.amount - fees

interest = min(interest, remaining)

principal = max(remaining - interest, 0)

principal = min(principal, remaining\_principal)

allocation.principal = round(principal, 2)

allocation.interest = round(interest, 2)

allocation.fees = round(fees, 2)

session.add(allocation)

session.add(loan)

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)

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

# Enforce one allocation per payment

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

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

)

# Relationship

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

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))

# In LoanApplication model

def create\_loan(requested\_amount, term\_months):

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'))

# Get pricing config

term\_months = int(request.form.get('term\_months'))

config = PRICING.get(term\_months)

if not config:

flash("Invalid loan term", "danger")

return redirect(url\_for('loan\_application\_form'))

# Calculate fees

orig\_fee = requested\_amount \* config['origination']

ins\_fee = requested\_amount \* config['insurance']

crb\_fee = config['crb']

# Calculate total loan amount (capitalized)

capitalized\_fees = orig\_fee + ins\_fee + crb\_fee

loan\_amount = requested\_amount + capitalized\_fees

# Create loan with validated values

loan = LoanApplication(

requested\_amount=round(requested\_amount, 2),

loan\_amount=round(loan\_amount, 2),

term\_months=term\_months,

outstanding\_balance=round(loan\_amount, 2), # Initial balance = full loan amount

crb\_fees=round(crb\_fee, 2),

origination\_fees=round(orig\_fee, 2),

insurance\_fees=round(ins\_fee, 2),

# ... other required fields ...

)

db.session.add(loan)

db.session.commit()

disbursement = Disbursement(

loan\_id=loan.id,

amount=requested\_amount,

method='bank',

status='pending',

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

)

db.session.add(disbursement)

db.session.commit()

return loan

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()

@app.route('/admin/dashboard')

@login\_required

def admin\_dashboard():

if current\_user.role.name != 'admin':

abort(403)

users = User.query.all()

roles = Role.query.all()

section = request.args.get('section', 'users') # Default to 'users' tab

return render\_template('admin\_dashboard.html', users=users, roles=roles, section=section)

# 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'))

from datetime import datetime

from sqlalchemy import func, extract

from flask\_login import current\_user

from sqlalchemy import and\_

@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 = PRICING.get(loan.term\_months, {})

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 = PRICING.get(loan.term\_months, {})

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:

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)

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'))

@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.allocation)

)

.filter\_by(loan\_number=loan\_number)

.first\_or\_404())

config = PRICING.get(loan.term\_months or 0, {})

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):

if payment.allocation and payment.allocation.principal:

running\_balance -= payment.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):

if not payment.allocation:

continue

allocation = payment.allocation

running\_balance\_display -= allocation.principal

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({

'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(p.allocation.principal for p in loan.payments if p.allocation),

'interest': sum(p.allocation.interest for p in loan.payments if p.allocation),

'fees': sum(p.allocation.fees for p in loan.payments if p.allocation)

}

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,

totals=totals

)

except Exception as e:

flash(f"Error generating statement: {str(e)}", "danger")

return redirect(url\_for('loanbook'))

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.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'])

loan\_amount = float(data['loan\_amount'])

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 = PRICING.get(term\_months)

if not config:

raise Exception("Invalid loan term selected.")

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

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.")

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}"

customer = Customer(

national\_id=data['national\_id'],

first\_name=data['first\_name'],

last\_name=data['last\_name'],

gender=data.get('gender'),

dob=data.get('dob'),

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),

service\_length=data.get('service\_length'),

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()

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)

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}"

# 🧠 UNIVERSAL cash to client logic

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

top\_up\_balance = 0

if previous\_loan:

# Example of a valid top-up loan: if it's active or closed recently

if 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)

)

db.session.add(loan)

db.session.flush()

loan.generate\_repayment\_schedule()

# --- Insert disbursement record

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()

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'])

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()

for customer in customers:

customer.is\_approved\_for\_creation = True

customer.checker\_id = 2

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'

)

db.session.add(loan)

else:

for loan in customer.loans:

loan.status = 'pending'

db.session.commit()

flash(f"{len(customers)} customer(s) approved!", "success")

else:

flash("No customers selected.", "warning")

return redirect(url\_for('approve\_customers'))

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

# Normalize amount\_requested safely here before passing to template

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)

@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 == True) \

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

.add\_entity(Customer) \

.all()

processed\_loans = []

for loan\_app, customer in loans:

processed\_loans.append({

'loan': loan\_app,

'customer': customer,

'current\_balance': loan\_app.balance or 0.0

})

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']) # Assuming the customer ID is passed in the form

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

loan\_amount = float(data['loan\_amount'])

category = data.get('category')

# Find the existing customer by ID

customer = Customer.query.get(customer\_id)

if not customer:

flash("Customer not found.", "danger")

return redirect(url\_for('create\_existing\_loan'))

# Check if loan details are valid

config = PRICING.get(term\_months)

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

# Monthly repayment calculation

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 Number and File Number ---

loan\_number, file\_number = generate\_loan\_and\_file\_number(category, term\_months, db.session)

# --- Create Loan Application ---

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 necessary

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, # Assign the loan number

file\_number=file\_number # Assign the file number

)

db.session.add(loan)

db.session.commit()

flash("Loan created successfully for existing customer. Awaiting approval.", "success")

return redirect(url\_for('loanbook')) # Adjust to where you want to redirect

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.allocation)) \

.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.allocation)

) \

.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:

# Normalize keys to lowercase and strip whitespace

normalized\_row = {k.strip().lower(): v.strip() for k, v in row.items()}

# Get required fields with case-insensitive keys

loan\_number = normalized\_row.get('loan\_number', '').strip()

if not loan\_number:

errors.append(f"Missing loan\_number in row: {row}")

continue

# Handle amount formatting (e.g., "107,624.61" -> 107624.61)

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

# Find loan (case-sensitive match)

loan = LoanApplication.query.filter\_by(loan\_number=loan\_number).first()

if not loan:

errors.append(f"Loan {loan\_number} not found")

continue

# Create payment with optional fields

payment = Payment(

loan\_id=loan.id,

amount=amount,

method=normalized\_row.get('method', 'Batch Upload'), # Use CSV method if available

reference=normalized\_row.get('reference', '')

)

db.session.add(payment)

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 processing

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.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 request handling

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('/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 = 25

ajax = request.args.get("ajax") == "true"

loans\_query = LoanApplication.query \

.join(Customer, LoanApplication.customer\_id == Customer.id) \

.options(joinedload(LoanApplication.payments).joinedload(Payment.allocation))

all\_loans = loans\_query.all()

processed\_loans = []

for loan in all\_loans:

customer = loan.customer

config = PRICING.get(loan.term\_months or 0, {})

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(

(p.allocation.principal or 0) for p in loan.payments if p.allocation

)

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) 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)

config = PRICING.get(term\_months)

if not category\_info or not config:

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

# --- Fees

crb\_fees = 3000

origination\_fees = new\_amount \* config['origination']

insurance\_fees = new\_amount \* config['insurance']

collection\_fees = new\_amount \* config['collection']

def calculate\_topup\_balance(loan):

from datetime import datetime

config = PRICING.get(loan.term\_months, {})

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(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:

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)

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\_balance = round(current\_balance + projected\_interest(3), 2)

return top\_up\_balance

# --- Get amount to settle the old loan

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

app.logger.debug(f"[DEBUG] Balance calc for base loan #{base\_loan.loan\_number}: {top\_up\_balance}")

cash\_to\_client = new\_amount - top\_up\_balance

if cash\_to\_client <= 0:

raise Exception("Requested amount is too low to cover top up.")

# --- Capitalized amount includes fees (but full new loan is used for annuity math)

capitalized\_amount = new\_amount + origination\_fees + insurance\_fees + crb\_fees

# --- Monthly instalment (add collection fee after annuity)

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

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

monthly\_instalment = (capitalized\_amount \* annuity\_factor) + collection\_fees

# --- Generate new loan number

now = datetime.utcnow()

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

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

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

prefix = category\_info['prefix']

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

# --- Create 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, # ✅ stored here

cash\_to\_client=round(cash\_to\_client, 2), # ✅ stored here

date\_created=now,

disbursement\_date=None

)

db.session.add(topup\_loan)

db.session.flush() # Ensure we get topup\_loan.id

disbursement = Disbursement(

loan\_id=topup\_loan.id,

amount=cash\_to\_client,

method='bank', # or determine from UI

status='pending', # or 'success' if auto-approved

reference=f"Top-up disbursement for {topup\_loan.loan\_number}"

)

db.session.add(disbursement)

# --- Attach documents

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:

doc = Document(

customer\_id=base\_loan.customer\_id,

loan\_id=topup\_loan.id,

filename=filename,

filetype=filetype,

path=filepath,

uploaded\_at=datetime.utcnow()

)

db.session.add(doc)

# --- Repayment Schedule

topup\_loan.generate\_repayment\_schedule()

# --- Pay off old 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\_payment(payment, base\_loan)

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):

new\_amount = float(data['amount\_requested'])

term\_months = int(data['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 = PRICING.get(term\_months)

if not category\_info or not config:

raise Exception("Invalid category or term.")

crb\_fees = 3000

origination\_fees = new\_amount \* config['origination']

insurance\_fees = new\_amount \* config['insurance']

collection\_fees = new\_amount \* config['collection']

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}"

additional\_loan = LoanApplication(

customer\_id=base\_loan.customer\_id,

loan\_amount=new\_amount,

term\_months=term\_months,

monthly\_instalment=round(monthly\_instalment, 2),

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

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

category=category\_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 # 💡 assign full amount as cash to client

)

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)

closure\_type = request.form.get('closure\_type', 'settlement')

if not loan or loan.loan\_state != 'active':

flash('Loan is not active', 'danger')

return redirect(request.referrer)

if 'settle\_file' not in request.files:

flash('No settlement proof provided', 'danger')

return redirect(request.referrer)

file = request.files['settle\_file']

if file.filename == '':

flash('No selected file', 'danger')

return redirect(request.referrer)

try:

# Save settlement file

filename = secure\_filename(

f"settlement\_{loan.loan\_number}\_{datetime.utcnow().strftime('%Y%m%d%H%M%S')}{os.path.splitext(file.filename)[1]}"

)

filepath = os.path.join(app.config['UPLOAD\_FOLDER'], 'settlements', filename)

os.makedirs(os.path.dirname(filepath), exist\_ok=True)

file.save(filepath)

# Force recalculate BEFORE starting

loan.recalculate\_balance()

db.session.flush()

# Compute balances first

balances = calculate\_balances(loan)

# Determine amount to pay

if closure\_type in ('insurance', 'write\_off'):

principal = balances['current\_balance']

settlement\_interest = 0.0

amount\_to\_pay = principal

elif closure\_type == 'settlement':

principal = balances['current\_balance']

settlement\_interest = balances['settlement\_interest']

amount\_to\_pay = balances['settlement\_balance']

if amount\_to\_pay is None or amount\_to\_pay <= 0:

flash('Settlement balance is not available or zero', 'danger')

return redirect(request.referrer)

else:

flash('Invalid closure type provided', 'danger')

return redirect(request.referrer)

loan.settlement\_type = request.form.get('settlement\_type')

loan.settling\_institution = request.form.get('settling\_institution')

loan.settlement\_reason = request.form.get('settlement\_reason')

# 💡 Lock in settlement balance to store later in Loan:

settlement\_balance\_paid = round(principal + settlement\_interest, 2)

# 1️⃣ Create and commit Payment FIRST

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.commit() # important: commit first!

# 2️⃣ Reload loan fresh so balances are correct

loan = LoanApplication.query.get(loan.id)

loan.recalculate\_balance()

# 3️⃣ Determine allocation amounts AFTER payment included

# Note: loan.current\_balance should now be 0, but we take principal from balances

allocation = PaymentAllocation(

payment\_id=payment.id,

principal=principal,

interest=0.0,

fees=0.0,

settlement\_interest=settlement\_interest

)

db.session.add(allocation)

db.session.commit()

# 4️⃣ Reload loan again and finalize closure state

loan = LoanApplication.query.get(loan.id)

loan.loan\_state = 'settled\_client' if closure\_type == 'settlement' else closure\_type

loan.closure\_type = closure\_type

loan.closure\_date = datetime.utcnow()

# 🔥 Lock in settlement\_balance correctly (critical step)

loan.settlement\_balance = settlement\_balance\_paid

loan.current\_balance = 0.0

loan.top\_up\_balance = 0.0

# Recalculate balances one last time (will now RESPECT locked settlement\_balance)

loan.recalculate\_balance()

# 5️⃣ Update repayment schedules

for schedule in loan.repayment\_schedules:

if schedule.status != 'paid':

schedule.status = 'settled'

db.session.commit()

flash(f'Loan {loan.loan\_number} settled successfully. Balance cleared.', 'success')

except Exception as e:

db.session.rollback()

flash(f'Error settling loan: {str(e)}', 'danger')

app.logger.error(f"Settlement error: {str(e)}", exc\_info=True)

return redirect(request.referrer)

@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)