E:\expense\_tracker\backend

E:\expense\_tracker\backend\app

E:\expense\_tracker\backend\app\fonts

E:\expense\_tracker\backend\app\fonts\dejavu-fonts-ttf-2.37 (I have some files related to this)

E:\expense\_tracker\backend\app\models

E:\expense\_tracker\backend\app\models\\_\_init\_\_.py

"""

Models package initialization

Exports all database models for easy importing

"""

from app.models.user import User

from app.models.category import Category

from app.models.payment\_mode import PaymentMode

from app.models.expense import Expense

# Export all models for easy access

\_\_all\_\_ = ['User', 'Category', 'PaymentMode', 'Expense']

E:\expense\_tracker\backend\app\models\category.py

"""

Category Model - Manages expense categories

Predefined categories: Travel, Food, Payments to Friends, etc.

"""

from app.extensions import db

from datetime import datetime, timezone

class Category(db.Model):

    """

    Category model for expense classification

    Attributes:

        id: Primary key

        name: Category name (e.g., Travel, Food, Payments to Friends)

        description: Optional detailed description of category

        icon: Optional icon identifier for frontend display

        color: Optional color code for UI visualization (e.g., #FF5733)

        is\_active: Whether this category is available for selection

        created\_at: Timestamp when category was created

    Relationships:

        expenses: One-to-many relationship with Expense model

    """

    \_\_tablename\_\_ = 'categories'

    # Primary Key

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

    # Category Information

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

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

    # UI Configuration (for frontend styling)

    icon = db.Column(db.String(50), nullable=True)  # e.g., 'travel', 'food', 'payment'

    color = db.Column(db.String(7), nullable=True)  # Hex color code

    # Status

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

    # Timestamp

    created\_at = db.Column(db.DateTime, default=lambda: datetime.now(timezone.utc), nullable=False)

    # Relationships

    expenses = db.relationship(

        'Expense',

        backref='category',

        lazy='dynamic',

        cascade='all, delete-orphan'

    )

    def \_\_init\_\_(self, name, description=None, icon=None, color=None):

        """

        Initialize a new category

        Args:

            name (str): Category name

            description (str, optional): Category description

            icon (str, optional): Icon identifier

            color (str, optional): Hex color code

        """

        self.name = name

        self.description = description

        self.icon = icon

        self.color = color

    def to\_dict(self):

        """

        Convert category to dictionary for JSON serialization

        Returns:

            dict: Category data

        """

        return {

            'id': self.id,

            'name': self.name,

            'description': self.description,

            'icon': self.icon,

            'color': self.color,

            'is\_active': self.is\_active,

            'created\_at': self.created\_at.isoformat()

        }

    @staticmethod

    def get\_default\_categories():

        """

        Returns list of default categories to seed database

        Returns:

            list: Default category dictionaries

        """

        return [

            {'name': 'Travel', 'description': 'Transportation and travel expenses', 'icon': 'directions\_car', 'color': '#4CAF50'},

            {'name': 'Food', 'description': 'Food and dining expenses', 'icon': 'restaurant', 'color': '#FF9800'},

            {'name': 'Payments to Friends', 'description': 'Money sent to friends', 'icon': 'people', 'color': '#2196F3'},

            {'name': 'Self Transfer to Accounts', 'description': 'Transfers between own accounts', 'icon': 'swap\_horiz', 'color': '#9C27B0'},

            {'name': 'Wallet Recharge', 'description': 'Digital wallet top-ups', 'icon': 'account\_balance\_wallet', 'color': '#F44336'},

            {'name': 'Other', 'description': 'Miscellaneous expenses', 'icon': 'more\_horiz', 'color': '#607D8B'}

        ]

    def \_\_repr\_\_(self):

        """String representation for debugging"""

        return f'<Category {self.name}>'

E:\expense\_tracker\backend\app\models\expense.py

"""

Expense Model - Core expense tracking functionality

Stores individual expense transactions with all related data

"""

from app.extensions import db

from datetime import datetime, timezone

from sqlalchemy import func

class Expense(db.Model):

    """

    Expense model for tracking individual transactions

    Attributes:

        id: Primary key

        user\_id: Foreign key to User model

        category\_id: Foreign key to Category model

        payment\_mode\_id: Foreign key to PaymentMode model

        amount: Expense amount (decimal with 2 decimal places)

        description: Text description of the expense

        expense\_date: Date of the expense transaction

        created\_at: Timestamp when record was created

        updated\_at: Timestamp when record was last modified

    Relationships:

        user: Many-to-one relationship with User model

        category: Many-to-one relationship with Category model

        payment\_mode: Many-to-one relationship with PaymentMode model

    """

    \_\_tablename\_\_ = 'expenses'

    # Primary Key

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

    # Foreign Keys

    user\_id = db.Column(

        db.Integer,

        db.ForeignKey('users.id', ondelete='CASCADE'),

        nullable=False,

        index=True

    )

    category\_id = db.Column(

        db.Integer,

        db.ForeignKey('categories.id', ondelete='RESTRICT'),

        nullable=False,

        index=True

    )

    payment\_mode\_id = db.Column(

        db.Integer,

        db.ForeignKey('payment\_modes.id', ondelete='RESTRICT'),

        nullable=False,

        index=True

    )

    # Expense Details

    amount = db.Column(

        db.Numeric(10, 2),

        nullable=False

    )  # Up to 99,999,999.99

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

    # Date Fields

    expense\_date = db.Column(db.Date, nullable=False, index=True)

    # Timestamps

    created\_at = db.Column(

        db.DateTime,

        default=lambda: datetime.now(timezone.utc),

        nullable=False

    )

    updated\_at = db.Column(

        db.DateTime,

        default=lambda: datetime.now(timezone.utc),

        onupdate=lambda: datetime.now(timezone.utc),

        nullable=False

    )

    # Indexes for better query performance

    \_\_table\_args\_\_ = (

        db.Index('idx\_user\_date', 'user\_id', 'expense\_date'),

        db.Index('idx\_user\_category', 'user\_id', 'category\_id'),

        db.Index('idx\_user\_payment', 'user\_id', 'payment\_mode\_id'),

    )

    def \_\_init\_\_(self, user\_id, category\_id, payment\_mode\_id, amount, description, expense\_date=None):

        """

        Initialize a new expense record

        Args:

            user\_id (int): ID of the user who created the expense

            category\_id (int): ID of the expense category

            payment\_mode\_id (int): ID of the payment mode used

            amount (float): Expense amount

            description (str): Description of the expense

            expense\_date (date, optional): Date of expense (defaults to today)

        """

        self.user\_id = user\_id

        self.category\_id = category\_id

        self.payment\_mode\_id = payment\_mode\_id

        self.amount = amount

        self.description = description

        self.expense\_date = expense\_date or datetime.now(timezone.utc).date()

    def to\_dict(self, include\_relations=True):

        """

        Convert expense to dictionary for JSON serialization

        Args:

            include\_relations (bool): Whether to include related model data

        Returns:

            dict: Expense data with optional related data

        """

        data = {

            'id': self.id,

            'user\_id': self.user\_id,

            'category\_id': self.category\_id,

            'payment\_mode\_id': self.payment\_mode\_id,

            'amount': float(self.amount),

            'description': self.description,

            'expense\_date': self.expense\_date.isoformat(),

            'created\_at': self.created\_at.isoformat(),

            'updated\_at': self.updated\_at.isoformat()

        }

        # Include related model data if requested

        if include\_relations:

            data['category'] = self.category.to\_dict() if self.category else None

            data['payment\_mode'] = self.payment\_mode.to\_dict() if self.payment\_mode else None

        return data

    @staticmethod

    def get\_total\_by\_user(user\_id, start\_date=None, end\_date=None):

        """

        Calculate total expenses for a user within date range

        Args:

            user\_id (int): User ID

            start\_date (date, optional): Start date for filtering

            end\_date (date, optional): End date for filtering

        Returns:

            float: Total expense amount

        """

        query = db.session.query(func.sum(Expense.amount)).filter(Expense.user\_id == user\_id)

        if start\_date:

            query = query.filter(Expense.expense\_date >= start\_date)

        if end\_date:

            query = query.filter(Expense.expense\_date <= end\_date)

        result = query.scalar()

        return float(result) if result else 0.0

    @staticmethod

    def get\_total\_by\_category(user\_id, category\_id, start\_date=None, end\_date=None):

        """

        Calculate total expenses for a specific category

        Args:

            user\_id (int): User ID

            category\_id (int): Category ID

            start\_date (date, optional): Start date for filtering

            end\_date (date, optional): End date for filtering

        Returns:

            float: Total expense amount for category

        """

        query = db.session.query(func.sum(Expense.amount)).filter(

            Expense.user\_id == user\_id,

            Expense.category\_id == category\_id

        )

        if start\_date:

            query = query.filter(Expense.expense\_date >= start\_date)

        if end\_date:

            query = query.filter(Expense.expense\_date <= end\_date)

        result = query.scalar()

        return float(result) if result else 0.0

    @staticmethod

    def get\_total\_by\_payment\_mode(user\_id, payment\_mode\_id, start\_date=None, end\_date=None):

        """

        Calculate total expenses for a specific payment mode

        Args:

            user\_id (int): User ID

            payment\_mode\_id (int): Payment Mode ID

            start\_date (date, optional): Start date for filtering

            end\_date (date, optional): End date for filtering

        Returns:

            float: Total expense amount for payment mode

        """

        query = db.session.query(func.sum(Expense.amount)).filter(

            Expense.user\_id == user\_id,

            Expense.payment\_mode\_id == payment\_mode\_id

        )

        if start\_date:

            query = query.filter(Expense.expense\_date >= start\_date)

        if end\_date:

            query = query.filter(Expense.expense\_date <= end\_date)

        result = query.scalar()

        return float(result) if result else 0.0

    def \_\_repr\_\_(self):

        """String representation for debugging"""

        return f'<Expense {self.description[:20]} - ₹{self.amount}>'

E:\expense\_tracker\backend\app\models\payment\_mode.py

"""

PaymentMode Model - Manages payment methods

Includes GPay (with bank options), Cash, Metro Card, etc.

"""

from app.extensions import db

from datetime import datetime, timezone

class PaymentMode(db.Model):

    """

    Payment Mode model for tracking payment methods

    Attributes:

        id: Primary key

        name: Payment method name (e.g., GPay, Cash, Metro Card)

        bank\_name: Optional bank name for digital payments (e.g., SBI, HDFC, IOB)

        type: Payment type (digital, cash, card)

        is\_active: Whether this payment mode is available

        created\_at: Timestamp when payment mode was created

    Relationships:

        expenses: One-to-many relationship with Expense model

    """

    \_\_tablename\_\_ = 'payment\_modes'

    # Primary Key

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

    # Payment Mode Information

    name = db.Column(db.String(50), nullable=False, index=True)

    bank\_name = db.Column(db.String(50), nullable=True)  # For GPay - SBI, HDFC, IOB

    # Payment Type Classification

    type = db.Column(

        db.String(20),

        nullable=False,

        default='digital'

    )  # Options: 'digital', 'cash', 'card', 'other'

    # Status

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

    # Timestamp

    created\_at = db.Column(db.DateTime, default=lambda: datetime.now(timezone.utc), nullable=False)

    # Relationships

    expenses = db.relationship(

        'Expense',

        backref='payment\_mode',

        lazy='dynamic',

        cascade='all, delete-orphan'

    )

    # Unique constraint: combination of name and bank\_name must be unique

    \_\_table\_args\_\_ = (

        db.UniqueConstraint('name', 'bank\_name', name='unique\_payment\_mode'),

    )

    def \_\_init\_\_(self, name, bank\_name=None, type='digital'):

        """

        Initialize a new payment mode

        Args:

            name (str): Payment mode name

            bank\_name (str, optional): Bank name for GPay payments

            type (str): Payment type classification

        """

        self.name = name

        self.bank\_name = bank\_name

        self.type = type

    def to\_dict(self):

        """

        Convert payment mode to dictionary for JSON serialization

        Returns:

            dict: Payment mode data

        """

        return {

            'id': self.id,

            'name': self.name,

            'bank\_name': self.bank\_name,

            'type': self.type,

            'display\_name': self.get\_display\_name(),

            'is\_active': self.is\_active,

            'created\_at': self.created\_at.isoformat()

        }

    def get\_display\_name(self):

        """

        Get formatted display name for frontend

        Returns:

            str: Formatted payment mode name (e.g., "GPay - SBI" or "Cash")

        """

        if self.bank\_name:

            return f"{self.name} - {self.bank\_name}"

        return self.name

    @staticmethod

    def get\_default\_payment\_modes():

        """

        Returns list of default payment modes to seed database

        Returns:

            list: Default payment mode dictionaries

        """

        return [

            # GPay with different banks

            {'name': 'GPay', 'bank\_name': 'SBI', 'type': 'digital'},

            {'name': 'GPay', 'bank\_name': 'HDFC', 'type': 'digital'},

            {'name': 'GPay', 'bank\_name': 'IOB', 'type': 'digital'},

            # Other payment methods

            {'name': 'Cash', 'bank\_name': None, 'type': 'cash'},

            {'name': 'Metro Card', 'bank\_name': None, 'type': 'card'},

            {'name': 'Credit Card', 'bank\_name': None, 'type': 'card'},

            {'name': 'Debit Card', 'bank\_name': None, 'type': 'card'},

            {'name': 'Other', 'bank\_name': None, 'type': 'other'}

        ]

    def \_\_repr\_\_(self):

        """String representation for debugging"""

        return f'<PaymentMode {self.get\_display\_name()}>'

E:\expense\_tracker\backend\app\models\user.py

"""

User Model - Handles user authentication and account management

Stores user credentials with secure password hashing

"""

from app.extensions import db

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

from datetime import datetime, timezone

class User(db.Model):

    """

    User model for authentication and user management

    Attributes:

        id: Primary key, auto-incrementing integer

        email: Unique email address for login (max 120 chars)

        password\_hash: Bcrypt hashed password (never store plain passwords)

        full\_name: Optional user's full name

        created\_at: Timestamp when account was created

        updated\_at: Timestamp when account was last modified

        is\_active: Boolean flag for account status (soft delete)

    Relationships:

        expenses: One-to-many relationship with Expense model

    """

    \_\_tablename\_\_ = 'users'

    # Primary Key

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

    # Authentication Fields

    email = db.Column(db.String(120), unique=True, nullable=False, index=True)

    password\_hash = db.Column(db.String(255), nullable=False)

    # User Information

    full\_name = db.Column(db.String(100), nullable=True)

    # Account Status

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

    # Timestamps (UTC)

    created\_at = db.Column(db.DateTime, default=lambda: datetime.now(timezone.utc), nullable=False)

    updated\_at = db.Column(

        db.DateTime,

        default=lambda: datetime.now(timezone.utc),

        onupdate=lambda: datetime.now(timezone.utc),

        nullable=False

    )

    # Relationships

    expenses = db.relationship(

        'Expense',

        backref='user',

        lazy='dynamic',

        cascade='all, delete-orphan'

    )

    def \_\_init\_\_(self, email, password, full\_name=None):

        """

        Initialize a new user with email and password

        Args:

            email (str): User's email address

            password (str): Plain text password (will be hashed)

            full\_name (str, optional): User's full name

        """

        self.email = email

        self.set\_password(password)

        self.full\_name = full\_name

    def set\_password(self, password):

        """

        Hash and store password securely using Werkzeug's security module

        Uses PBKDF2 with SHA-256 by default

        Args:

            password (str): Plain text password to hash

        """

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

    def check\_password(self, password):

        """

        Verify password against stored hash

        Args:

            password (str): Plain text password to verify

        Returns:

            bool: True if password matches, False otherwise

        """

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

    def to\_dict(self):

        """

        Convert user object to dictionary (exclude sensitive data)

        Used for JSON serialization in API responses

        Returns:

            dict: User data without password\_hash

        """

        # ✅ ADDED: Debug logging

        print(f"🔍 to\_dict() called for user: {self.email}")

        print(f"🔍 full\_name value: {self.full\_name}")

        print(f"🔍 full\_name type: {type(self.full\_name)}")

        user\_dict = {

            'id': self.id,

            'email': self.email,

            'full\_name': self.full\_name,

            'is\_active': self.is\_active,

            'created\_at': self.created\_at.isoformat(),

            'updated\_at': self.updated\_at.isoformat()

        }

        print(f"🔍 Returning user\_dict: {user\_dict}")

        return user\_dict

    def \_\_repr\_\_(self):

        """String representation for debugging"""

        return f'<User {self.email}>'

E:\expense\_tracker\backend\app\routes

E:\expense\_tracker\backend\app\routes\\_\_init\_\_.py

"""

Routes Package Initialization

Registers all blueprints for the application

"""

from app.routes.auth import auth\_bp

from app.routes.expenses import expenses\_bp

from app.routes.analytics import analytics\_bp

from app.routes.ai import ai\_bp

# Export all blueprints

\_\_all\_\_ = ['auth\_bp', 'expenses\_bp', 'analytics\_bp', 'ai\_bp']

def register\_blueprints(app):

    """

    Register all application blueprints

    Args:

        app: Flask application instance

    Registered Blueprints:

        - auth\_bp: User authentication (login, register, token refresh)

        - expenses\_bp: Expense CRUD operations and management

        - analytics\_bp: Analytics, reports, and dashboard data

        - ai\_bp: AI-powered expense queries (OpenAI integration)

    """

    # Authentication routes (/api/auth/\*)

    app.register\_blueprint(auth\_bp, url\_prefix='/api/auth')

    # Expense management routes (/api/expenses/\*)

    app.register\_blueprint(expenses\_bp, url\_prefix='/api/expenses')

    # Analytics and dashboard routes (/api/analytics/\*)

    app.register\_blueprint(analytics\_bp, url\_prefix='/api/analytics')

    # AI query routes (/api/ai/\*) - OpenAI powered

    app.register\_blueprint(ai\_bp, url\_prefix='/api/ai')

    print("✓ All blueprints registered successfully")

E:\expense\_tracker\backend\app\routes\ai.py

"""

AI Query Routes

Handles OpenAI integration for natural language expense queries

"""

from flask import Blueprint, request, jsonify

from flask\_jwt\_extended import jwt\_required, get\_jwt\_identity

from app.extensions import db

from app.models import Expense, Category, PaymentMode

from sqlalchemy import func

from datetime import datetime, date, timedelta

from openai import OpenAI

import os

# Create Blueprint

ai\_bp = Blueprint('ai', \_\_name\_\_)

# ✅ CHANGED: Configure OpenAI instead of Gemini

OPENAI\_API\_KEY = os.environ.get('OPENAI\_API\_KEY')

MODEL\_NAME = os.environ.get('OPENAI\_MODEL', 'gpt-4o-mini')

# Available models: 'gpt-4o', 'gpt-4o-mini', 'gpt-3.5-turbo'

client = None

if OPENAI\_API\_KEY:

    try:

        client = OpenAI(api\_key=OPENAI\_API\_KEY)

        print(f"✅ OpenAI configured successfully with model: {MODEL\_NAME}")

    except Exception as e:

        print(f"❌ Failed to initialize OpenAI client: {e}")

        client = None

else:

    print("⚠️ OPENAI\_API\_KEY not found in environment variables")

    print("💡 Get your API key from: https://platform.openai.com/api-keys")

    print("💡 Add it to your .env file: OPENAI\_API\_KEY=your\_key\_here")

def get\_expense\_context(user\_id, query\_lower):

    """

    Fetch relevant expense data based on the user query

    """

    # Safe user\_id conversion

    if isinstance(user\_id, str):

        try:

            user\_id = int(user\_id)

        except ValueError:

            print(f"⚠️ Invalid user\_id: {user\_id}")

            return None

    context = {}

    today = date.today()

    # Determine time period from query

    if 'today' in query\_lower:

        start\_date = end\_date = today

        period = "today"

    elif 'yesterday' in query\_lower:

        start\_date = end\_date = today - timedelta(days=1)

        period = "yesterday"

    elif 'week' in query\_lower or 'this week' in query\_lower:

        start\_date = today - timedelta(days=7)

        end\_date = today

        period = "this week"

    elif 'month' in query\_lower or 'this month' in query\_lower:

        start\_date = today.replace(day=1)

        end\_date = today

        period = "this month"

    elif 'year' in query\_lower or 'this year' in query\_lower:

        start\_date = today.replace(month=1, day=1)

        end\_date = today

        period = "this year"

    else:

        start\_date = today - timedelta(days=30)

        end\_date = today

        period = "last 30 days"

    context['period'] = period

    context['start\_date'] = start\_date.isoformat()

    context['end\_date'] = end\_date.isoformat()

    # Get total expenses

    base\_query = Expense.query.filter(

        Expense.user\_id == user\_id,

        Expense.expense\_date >= start\_date,

        Expense.expense\_date <= end\_date

    )

    total = base\_query.with\_entities(func.sum(Expense.amount)).scalar() or 0

    context['total\_expenses'] = float(total)

    context['expense\_count'] = base\_query.count()

    # Get category breakdown if relevant to query

    category\_keywords = ['category', 'travel', 'food', 'payment', 'transfer', 'wallet', 'recharge', 'shopping', 'entertainment']

    if any(keyword in query\_lower for keyword in category\_keywords):

        category\_data = db.session.query(

            Category.name,

            func.sum(Expense.amount).label('total'),

            func.count(Expense.id).label('count')

        ).join(Expense) \

         .filter(

            Expense.user\_id == user\_id,

            Expense.expense\_date >= start\_date,

            Expense.expense\_date <= end\_date

        ).group\_by(Category.name) \

         .order\_by(func.sum(Expense.amount).desc()) \

         .all()

        context['by\_category'] = [

            {

                'name': row.name,

                'total': float(row.total),

                'count': row.count

            }

            for row in category\_data

        ]

    # Get payment mode breakdown if relevant to query

    if 'payment' in query\_lower or 'gpay' in query\_lower or 'cash' in query\_lower or 'card' in query\_lower:

        payment\_data = db.session.query(

            PaymentMode.name,

            PaymentMode.bank\_name,

            func.sum(Expense.amount).label('total'),

            func.count(Expense.id).label('count')

        ).join(Expense) \

         .filter(

            Expense.user\_id == user\_id,

            Expense.expense\_date >= start\_date,

            Expense.expense\_date <= end\_date

        ).group\_by(PaymentMode.name, PaymentMode.bank\_name) \

         .order\_by(func.sum(Expense.amount).desc()) \

         .all()

        context['by\_payment\_mode'] = [

            {

                'name': f"{row.name} - {row.bank\_name}" if row.bank\_name else row.name,

                'total': float(row.total),

                'count': row.count

            }

            for row in payment\_data

        ]

    # Get recent expenses

    recent = base\_query.order\_by(Expense.expense\_date.desc()).limit(5).all()

    context['recent\_expenses'] = [

        {

            "date": exp.expense\_date.isoformat(),

            "description": exp.description or 'No description',

            "amount": float(exp.amount),

            "category": exp.category.name if exp.category else 'Uncategorized'

        }

        for exp in recent

    ]

    return context

@ai\_bp.route('/query', methods=['POST'])

@jwt\_required()

def ai\_query():

    """Process natural language query about expenses using OpenAI"""

    try:

        # Check if OpenAI is configured

        if not client:

            return jsonify({

                "error": "OpenAI is not configured. Please add OPENAI\_API\_KEY to your environment variables.",

                "fallback": "AI features are currently unavailable. Please check your API configuration.",

                "help": "Get your API key from: https://platform.openai.com/api-keys"

            }), 503

        current\_user\_id = get\_jwt\_identity()

        if isinstance(current\_user\_id, str):

            try:

                current\_user\_id = int(current\_user\_id)

            except ValueError:

                return jsonify({"error": "Invalid user ID"}), 400

        data = request.get\_json()

        if not data or not data.get('query'):

            return jsonify({"error": "Query is required"}), 400

        user\_query = data['query'].strip()

        if not user\_query:

            return jsonify({"error": "Query cannot be empty"}), 400

        print(f"🤖 Processing AI query: '{user\_query}' for user {current\_user\_id}")

        # Get expense context

        context = get\_expense\_context(current\_user\_id, user\_query.lower())

        if not context:

            return jsonify({"error": "Failed to retrieve expense context"}), 500

        # Handle case with no expenses

        if context['expense\_count'] == 0:

            return jsonify({

                "query": user\_query,

                "answer": f"You haven't recorded any expenses for {context['period']}. Start tracking your expenses to get AI-powered insights! 📊",

                "context": {

                    "period": context['period'],

                    "total\_expenses": 0,

                    "expense\_count": 0

                }

            }), 200

        # ✅ CHANGED: Build prompt for OpenAI (same format, different API)

        prompt = f"""You are a helpful financial assistant analyzing expense data.

User Question: "{user\_query}"

Expense Summary:

• Period: {context['period']} ({context['start\_date']} to {context['end\_date']})

• Total Spending: ₹{context['total\_expenses']:.2f}

• Number of Transactions: {context['expense\_count']}

"""

        # Add category breakdown if available

        if 'by\_category' in context and context['by\_category']:

            prompt += "\n📊 Spending by Category:\n"

            for cat in context['by\_category']:

                prompt += f"   • {cat['name']}: ₹{cat['total']:.2f} ({cat['count']} transactions)\n"

        # Add payment mode breakdown if available

        if 'by\_payment\_mode' in context and context['by\_payment\_mode']:

            prompt += "\n💳 Spending by Payment Method:\n"

            for pm in context['by\_payment\_mode']:

                prompt += f"   • {pm['name']}: ₹{pm['total']:.2f} ({pm['count']} transactions)\n"

        # Add recent expenses

        if context['recent\_expenses']:

            prompt += "\n🕐 Recent Expenses:\n"

            for exp in context['recent\_expenses'][:3]:

                prompt += f"   • {exp['date']}: {exp['description']} - ₹{exp['amount']:.2f} ({exp['category']})\n"

        prompt += """

Instructions:

• Answer the user's question directly and concisely

• Use specific numbers from the data

• Format currency as ₹X,XXX.XX

• Use emojis where appropriate (💰 📊 💳 etc.)

• Keep response under 150 words

• Use bullet points for lists

• Be friendly and conversational

"""

        print("🚀 Sending request to OpenAI...")

        try:

            # ✅ CHANGED: Use OpenAI API instead of Gemini

            response = client.chat.completions.create(

                model=MODEL\_NAME,

                messages=[

                    {

                        "role": "system",

                        "content": "You are a helpful financial assistant. Provide clear, concise answers about expense data."

                    },

                    {

                        "role": "user",

                        "content": prompt

                    }

                ],

                temperature=0.7,

                max\_tokens=500,

            )

            ai\_answer = response.choices[0].message.content

            print(f"✅ AI response received (length: {len(ai\_answer)} chars)")

        except Exception as e:

            error\_msg = str(e)

            print(f"❌ OpenAI API error: {error\_msg}")

            # Better error messages

            if 'model' in error\_msg.lower() and 'not found' in error\_msg.lower():

                return jsonify({

                    "error": f"Model '{MODEL\_NAME}' not found or not accessible.",

                    "fallback": f"Based on your data: You've spent ₹{context['total\_expenses']:.2f} over {context['expense\_count']} transactions in {context['period']}.",

                    "help": "Try using 'gpt-4o-mini' or 'gpt-3.5-turbo' in your .env file"

                }), 500

            elif 'api\_key' in error\_msg.lower() or 'authentication' in error\_msg.lower():

                return jsonify({

                    "error": "Invalid API key",

                    "fallback": "Please check your OPENAI\_API\_KEY in the .env file",

                    "help": "Get your API key from: https://platform.openai.com/api-keys"

                }), 500

            elif 'quota' in error\_msg.lower() or 'insufficient' in error\_msg.lower():

                return jsonify({

                    "error": "OpenAI API quota exceeded",

                    "fallback": "Please check your OpenAI account billing",

                    "help": "Visit: https://platform.openai.com/account/billing"

                }), 500

            else:

                return jsonify({

                    "error": f"AI query failed: {error\_msg}",

                    "fallback": "Unable to process your query at this time. Please try again or use the analytics dashboard."

                }), 500

        return jsonify({

            "query": user\_query,

            "answer": ai\_answer,

            "context": {

                "period": context['period'],

                "total\_expenses": context['total\_expenses'],

                "expense\_count": context['expense\_count']

            }

        }), 200

    except Exception as e:

        print(f"❌ AI query error: {str(e)}")

        import traceback

        traceback.print\_exc()

        return jsonify({

            "error": f"AI query failed: {str(e)}",

            "fallback": "Unable to process your query at this time. Please try again or use the analytics dashboard."

        }), 500

@ai\_bp.route('/suggestions', methods=['GET'])

@jwt\_required()

def get\_suggestions():

    """Get AI-powered spending suggestions and insights"""

    try:

        current\_user\_id = get\_jwt\_identity()

        if isinstance(current\_user\_id, str):

            try:

                current\_user\_id = int(current\_user\_id)

            except ValueError:

                return jsonify({"error": "Invalid user ID"}), 400

        # Get last 30 days data

        today = date.today()

        start\_date = today - timedelta(days=30)

        total = Expense.query.filter(

            Expense.user\_id == current\_user\_id,

            Expense.expense\_date >= start\_date

        ).with\_entities(func.sum(Expense.amount)).scalar() or 0

        # Better fallback suggestions

        fallback\_suggestions = [

            "📊 Track your daily expenses to identify spending patterns",

            "💰 Set a monthly budget for each category to stay on track",

            "🔍 Review your top spending categories regularly",

            "📈 Compare your spending month-over-month to spot trends",

            "💳 Monitor which payment methods you use most frequently",

            "🎯 Set savings goals and track your progress"

        ]

        if not client:

            return jsonify({

                "suggestions": fallback\_suggestions,

                "period": "General Tips",

                "total\_analyzed": float(total) if total else 0,

                "note": "AI suggestions unavailable - showing general tips",

                "is\_ai\_generated": False

            }), 200

        # No expenses case

        if not total or total == 0:

            return jsonify({

                "suggestions": [

                    "🎯 Start tracking your expenses to get personalized insights",

                    "📝 Add your first expense to begin building your financial profile",

                    "💡 Categorize expenses properly for better analysis",

                    "🔔 Check back after a week of tracking for AI-powered suggestions"

                ],

                "period": "Getting Started",

                "total\_analyzed": 0,

                "is\_ai\_generated": False

            }), 200

        # Get category data

        category\_data = db.session.query(

            Category.name,

            func.sum(Expense.amount).label('total'),

            func.count(Expense.id).label('count')

        ).join(Expense) \

         .filter(

            Expense.user\_id == current\_user\_id,

            Expense.expense\_date >= start\_date

        ).group\_by(Category.name) \

         .order\_by(func.sum(Expense.amount).desc()) \

         .all()

        # Build prompt

        prompt = f"""Analyze this user's spending pattern and provide 4-5 actionable insights.

Total Spending (Last 30 days): ₹{float(total):.2f}

Spending Breakdown:

"""

        for row in category\_data:

            percentage = (float(row.total) / float(total) \* 100) if total > 0 else 0

            prompt += f"• {row.name}: ₹{float(row.total):.2f} ({row.count} transactions, {percentage:.1f}%)\n"

        prompt += """

Provide insights on:

1. ✅ One positive spending habit

2. ⚠️ One area for potential savings

3. 💡 One tracking/budgeting tip

4. 📊 Overall financial health (Excellent/Good/Fair/Needs Attention)

5. 🎯 One specific action item

Format as bullet points with emojis. Keep it friendly, specific, and under 200 words.

"""

        try:

            # ✅ CHANGED: Use OpenAI API for suggestions

            response = client.chat.completions.create(

                model=MODEL\_NAME,

                messages=[

                    {

                        "role": "system",

                        "content": "You are a helpful financial advisor providing spending insights."

                    },

                    {

                        "role": "user",

                        "content": prompt

                    }

                ],

                temperature=0.8,

                max\_tokens=600,

            )

            suggestions\_text = response.choices[0].message.content

            return jsonify({

                "suggestions": suggestions\_text,

                "period": "Last 30 days",

                "total\_analyzed": float(total),

                "is\_ai\_generated": True

            }), 200

        except Exception as e:

            print(f"❌ OpenAI API error in suggestions: {str(e)}")

            return jsonify({

                "suggestions": fallback\_suggestions,

                "period": "Last 30 days",

                "total\_analyzed": float(total),

                "note": f"AI temporarily unavailable - showing general tips",

                "is\_ai\_generated": False

            }), 200

    except Exception as e:

        print(f"❌ Suggestions error: {str(e)}")

        import traceback

        traceback.print\_exc()

        return jsonify({

            "suggestions": [

                "📊 Review your spending patterns in the Analytics dashboard",

                "💰 Set monthly budgets for better financial control",

                "🔍 Identify your top spending categories",

                "📈 Track your expenses consistently for better insights"

            ],

            "period": "General Tips",

            "total\_analyzed": 0,

            "note": f"Error loading suggestions: {str(e)}",

            "is\_ai\_generated": False

        }), 200

E:\expense\_tracker\backend\app\routes\analytics.py

"""

Analytics Routes

Provides data for graphs, charts, and dashboard visualizations

"""

from flask import Blueprint, request, jsonify

from flask\_jwt\_extended import jwt\_required, get\_jwt\_identity

from app.extensions import db

from app.models import Expense, Category, PaymentMode

from sqlalchemy import func

from datetime import datetime, date, timedelta

from collections import defaultdict

# Create Blueprint

analytics\_bp = Blueprint('analytics', \_\_name\_\_)

@analytics\_bp.route('/categories-vs-expenses', methods=['GET'])

@jwt\_required()

def categories\_vs\_expenses():

    """

    Get expense totals grouped by category with counts and percentages

    Query Parameters:

        - start\_date: Filter start date (YYYY-MM-DD)

        - end\_date: Filter end date (YYYY-MM-DD)

    Returns:

        JSON response with category-wise expense breakdown

    """

    try:

        current\_user\_id = get\_jwt\_identity()

        # Build query with COUNT

        query = db.session.query(

            Category.name,

            Category.color,

            func.sum(Expense.amount).label('total'),

            func.count(Expense.id).label('count')

        ).join(Expense, Category.id == Expense.category\_id) \

         .filter(Expense.user\_id == current\_user\_id)

        # Apply date filters

        start\_date = request.args.get('start\_date')

        end\_date = request.args.get('end\_date')

        if start\_date:

            query = query.filter(Expense.expense\_date >= start\_date)

        if end\_date:

            query = query.filter(Expense.expense\_date <= end\_date)

        # Group by category and order by total

        results = query.group\_by(Category.name, Category.color) \

                      .order\_by(func.sum(Expense.amount).desc()) \

                      .all()

        # ✅ FIXED: Convert to float BEFORE summing to avoid Decimal issues

        grand\_total = float(sum(float(row.total) for row in results))

        # Format response with percentages

        data = [

            {

                "category": row.name,

                "color": row.color,

                "total": float(row.total),

                "count": row.count,

                "percentage": round((float(row.total) / grand\_total \* 100) if grand\_total > 0 else 0, 2)

            }

            for row in results

        ]

        return jsonify({

            "data": data,

            "chart\_type": "bar",

            "total": grand\_total,

            "filters": {

                "start\_date": start\_date,

                "end\_date": end\_date

            }

        }), 200

    except Exception as e:

        print(f"❌ Analytics error (categories): {str(e)}")

        import traceback

        traceback.print\_exc()

        return jsonify({"error": f"Failed to fetch analytics: {str(e)}"}), 500

@analytics\_bp.route('/payment-modes-vs-expenses', methods=['GET'])

@jwt\_required()

def payment\_modes\_vs\_expenses():

    """

    Get expense totals grouped by payment mode with counts and percentages

    Query Parameters:

        - start\_date: Filter start date (YYYY-MM-DD)

        - end\_date: Filter end date (YYYY-MM-DD)

    Returns:

        JSON response with payment mode-wise expense breakdown

    """

    try:

        current\_user\_id = get\_jwt\_identity()

        # Build query with COUNT

        query = db.session.query(

            PaymentMode.name,

            PaymentMode.bank\_name,

            func.sum(Expense.amount).label('total'),

            func.count(Expense.id).label('count')

        ).join(Expense, PaymentMode.id == Expense.payment\_mode\_id) \

         .filter(Expense.user\_id == current\_user\_id)

        # Apply date filters

        start\_date = request.args.get('start\_date')

        end\_date = request.args.get('end\_date')

        if start\_date:

            query = query.filter(Expense.expense\_date >= start\_date)

        if end\_date:

            query = query.filter(Expense.expense\_date <= end\_date)

        # Group by payment mode

        results = query.group\_by(PaymentMode.name, PaymentMode.bank\_name) \

                      .order\_by(func.sum(Expense.amount).desc()) \

                      .all()

        # ✅ FIXED: Convert to float BEFORE summing to avoid Decimal issues

        grand\_total = float(sum(float(row.total) for row in results))

        # Format response with percentages

        data = [

            {

                "paymentmode": f"{row.name} - {row.bank\_name}" if row.bank\_name else row.name,

                "payment\_mode": row.name,

                "bank\_name": row.bank\_name,

                "total": float(row.total),

                "count": row.count,

                "percentage": round((float(row.total) / grand\_total \* 100) if grand\_total > 0 else 0, 2)

            }

            for row in results

        ]

        return jsonify({

            "data": data,

            "chart\_type": "pie",

            "total": grand\_total,

            "filters": {

                "start\_date": start\_date,

                "end\_date": end\_date

            }

        }), 200

    except Exception as e:

        print(f"❌ Analytics error (payment modes): {str(e)}")

        import traceback

        traceback.print\_exc()

        return jsonify({"error": f"Failed to fetch analytics: {str(e)}"}), 500

@analytics\_bp.route('/payment-modes-vs-categories', methods=['GET'])

@jwt\_required()

def payment\_modes\_vs\_categories():

    """

    Get expense breakdown by payment mode and category (matrix view)

    Query Parameters:

        - start\_date: Filter start date (YYYY-MM-DD)

        - end\_date: Filter end date (YYYY-MM-DD)

    Returns:

        JSON response with cross-tabulation of payment modes and categories

    """

    try:

        current\_user\_id = get\_jwt\_identity()

        # Build query with COUNT

        query = db.session.query(

            Category.name.label('category'),

            PaymentMode.name.label('payment\_mode'),

            PaymentMode.bank\_name,

            func.sum(Expense.amount).label('total'),

            func.count(Expense.id).label('count')

        ).join(Expense, Category.id == Expense.category\_id) \

         .join(PaymentMode, Expense.payment\_mode\_id == PaymentMode.id) \

         .filter(Expense.user\_id == current\_user\_id)

        # Apply date filters

        start\_date = request.args.get('start\_date')

        end\_date = request.args.get('end\_date')

        if start\_date:

            query = query.filter(Expense.expense\_date >= start\_date)

        if end\_date:

            query = query.filter(Expense.expense\_date <= end\_date)

        # Group by both dimensions

        results = query.group\_by(

            Category.name,

            PaymentMode.name,

            PaymentMode.bank\_name

        ).all()

        # Format as nested structure

        data = defaultdict(dict)

        for row in results:

            payment\_display = f"{row.payment\_mode} - {row.bank\_name}" if row.bank\_name else row.payment\_mode

            data[row.category][payment\_display] = {

                "total": float(row.total),

                "count": row.count

            }

        # Convert to list format for frontend

        formatted\_data = [

            {

                "category": category,

                "payments": payments

            }

            for category, payments in data.items()

        ]

        return jsonify({

            "data": formatted\_data,

            "chart\_type": "stacked\_bar",

            "filters": {

                "start\_date": start\_date,

                "end\_date": end\_date

            }

        }), 200

    except Exception as e:

        print(f"❌ Analytics error (cross-tab): {str(e)}")

        import traceback

        traceback.print\_exc()

        return jsonify({"error": f"Failed to fetch analytics: {str(e)}"}), 500

@analytics\_bp.route('/daily-trend', methods=['GET'])

@jwt\_required()

def daily\_trend():

    """

    Get daily expense trend for dashboard graph with day names

    Query Parameters:

        - days: Number of days to look back (default: 7)

    Returns:

        JSON response with daily expense totals and day names

    """

    try:

        current\_user\_id = get\_jwt\_identity()

        # Get number of days

        days = request.args.get('days', 7, type=int)

        # Calculate date range

        end\_date = date.today()

        start\_date = end\_date - timedelta(days=days-1)

        # Query daily totals with COUNT

        results = db.session.query(

            Expense.expense\_date,

            func.sum(Expense.amount).label('total'),

            func.count(Expense.id).label('count')

        ).filter(

            Expense.user\_id == current\_user\_id,

            Expense.expense\_date >= start\_date,

            Expense.expense\_date <= end\_date

        ).group\_by(Expense.expense\_date) \

         .order\_by(Expense.expense\_date) \

         .all()

        # Create dictionary of results - ✅ FIXED: Convert Decimal to float

        expense\_dict = {

            row.expense\_date: {

                "total": float(row.total),

                "count": row.count

            }

            for row in results

        }

        # Day name mapping

        day\_names = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]

        # Fill in missing dates with 0

        data = []

        current\_date = start\_date

        while current\_date <= end\_date:

            expense\_data = expense\_dict.get(current\_date, {"total": 0.0, "count": 0})

            data.append({

                "date": current\_date.isoformat(),

                "dayname": day\_names[current\_date.weekday()],

                "total": expense\_data["total"],

                "count": expense\_data["count"],

                "average": round(expense\_data["total"] / expense\_data["count"], 2) if expense\_data["count"] > 0 else 0.0

            })

            current\_date += timedelta(days=1)

        return jsonify({

            "data": data,

            "chart\_type": "line",

            "period": f"Last {days} days"

        }), 200

    except Exception as e:

        print(f"❌ Analytics error (daily trend): {str(e)}")

        import traceback

        traceback.print\_exc()

        return jsonify({"error": f"Failed to fetch trend: {str(e)}"}), 500

@analytics\_bp.route('/monthly-summary', methods=['GET'])

@jwt\_required()

def monthly\_summary():

    """

    Get monthly expense summary for current year with counts and averages

    Returns:

        JSON response with monthly totals, counts, and averages

    """

    try:

        current\_user\_id = get\_jwt\_identity()

        # Get current year

        current\_year = date.today().year

        # Query monthly totals with COUNT

        results = db.session.query(

            func.extract('month', Expense.expense\_date).label('month'),

            func.sum(Expense.amount).label('total'),

            func.count(Expense.id).label('count'),

            func.avg(Expense.amount).label('average')

        ).filter(

            Expense.user\_id == current\_user\_id,

            func.extract('year', Expense.expense\_date) == current\_year

        ).group\_by(func.extract('month', Expense.expense\_date)) \

         .order\_by(func.extract('month', Expense.expense\_date)) \

         .all()

        # Create month name mapping

        month\_names = [

            "January", "February", "March", "April", "May", "June",

            "July", "August", "September", "October", "November", "December"

        ]

        # ✅ FIXED: Convert Decimal to float

        expense\_dict = {

            int(row.month): {

                "total": float(row.total),

                "count": row.count,

                "average": round(float(row.average), 2) if row.average else 0.0

            }

            for row in results

        }

        data = [

            {

                "month": month\_names[i],

                "month\_number": i + 1,

                "total": expense\_dict.get(i + 1, {}).get("total", 0.0),

                "count": expense\_dict.get(i + 1, {}).get("count", 0),

                "average": expense\_dict.get(i + 1, {}).get("average", 0.0)

            }

            for i in range(12)

        ]

        # Calculate year totals

        year\_total = sum(month["total"] for month in data)

        year\_count = sum(month["count"] for month in data)

        year\_average = round(year\_total / year\_count, 2) if year\_count > 0 else 0.0

        return jsonify({

            "data": data,

            "year": current\_year,

            "chart\_type": "bar",

            "summary": {

                "total": year\_total,

                "count": year\_count,

                "average": year\_average

            }

        }), 200

    except Exception as e:

        print(f"❌ Analytics error (monthly summary): {str(e)}")

        import traceback

        traceback.print\_exc()

        return jsonify({"error": f"Failed to fetch monthly summary: {str(e)}"}), 500

E:\expense\_tracker\backend\app\routes\auth.py

"""

Authentication Routes

Handles user registration, login, and logout functionality

"""

from flask import Blueprint, request, jsonify

from flask\_jwt\_extended import (

    create\_access\_token,

    jwt\_required,

    get\_jwt\_identity,

    get\_jwt

)

from app.extensions import db

from app.models import User

from datetime import timedelta

import re

# Create Blueprint

auth\_bp = Blueprint('auth', \_\_name\_\_)

# Token blacklist (in production, use Redis or database)

token\_blacklist = set()

def validate\_email(email):

    """

    Validate email format

    Args:

        email (str): Email address to validate

    Returns:

        bool: True if valid, False otherwise

    """

    pattern = r'^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

    return re.match(pattern, email) is not None

def validate\_password(password):

    """

    Validate password strength (minimum 6 characters)

    Args:

        password (str): Password to validate

    Returns:

        tuple: (bool, str) - (is\_valid, error\_message)

    """

    if len(password) < 6:

        return False, "Password must be at least 6 characters long"

    return True, ""

@auth\_bp.route('/register', methods=['POST'])

def register():

    """

    Register a new user

    Request Body:

        {

            "email": "user@example.com",

            "password": "password123",

            "full\_name": "John Doe" (optional)

        }

    Returns:

        JSON response with user data or error message

    """

    try:

        data = request.get\_json()

        print("=" \* 60)

        print("📝 REGISTRATION REQUEST")

        print("=" \* 60)

        if not data:

            print("❌ No data provided")

            return jsonify({"error": "No data provided"}), 400

        email = data.get('email', '').strip().lower()

        password = data.get('password', '')

        full\_name = data.get('full\_name', '').strip()

        print(f"📧 Email: {email}")

        print(f"👤 Full name: {full\_name}")

        if not email:

            print("❌ Email is required")

            return jsonify({"error": "Email is required"}), 400

        if not validate\_email(email):

            print("❌ Invalid email format")

            return jsonify({"error": "Invalid email format"}), 400

        if not password:

            print("❌ Password is required")

            return jsonify({"error": "Password is required"}), 400

        is\_valid, error\_msg = validate\_password(password)

        if not is\_valid:

            print(f"❌ Password validation failed: {error\_msg}")

            return jsonify({"error": error\_msg}), 400

        existing\_user = User.query.filter\_by(email=email).first()

        if existing\_user:

            print(f"❌ User already exists: {email}")

            return jsonify({"error": "User with this email already exists"}), 409

        new\_user = User(

            email=email,

            password=password,  # Will be hashed in User.\_\_init\_\_

            full\_name=full\_name if full\_name else None

        )

        db.session.add(new\_user)

        db.session.commit()

        print(f"✅ User registered successfully: ID={new\_user.id}, Email={new\_user.email}")

        print("=" \* 60)

        return jsonify({

            "message": "User registered successfully",

            "user": new\_user.to\_dict()

        }), 201

    except Exception as e:

        print(f"❌ Registration error: {str(e)}")

        print("=" \* 60)

        db.session.rollback()

        return jsonify({"error": f"Registration failed: {str(e)}"}), 500

@auth\_bp.route('/login', methods=['POST'])

def login():

    """

    Login user and return JWT access token

    Request Body:

        {

            "email": "user@example.com",

            "password": "password123"

        }

    Returns:

        JSON response with access token and user data

    """

    try:

        data = request.get\_json()

        print("=" \* 60)

        print("🔐 LOGIN REQUEST")

        print("=" \* 60)

        if not data:

            print("❌ No data provided")

            return jsonify({"error": "No data provided"}), 400

        email = data.get('email', '').strip().lower()

        password = data.get('password', '')

        print(f"📧 Login attempt for: {email}")

        print(f"🔑 Password length: {len(password)} characters")

        if not email or not password:

            print("❌ Email and password are required")

            return jsonify({"error": "Email and password are required"}), 400

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

        if not user:

            print(f"❌ User not found: {email}")

            return jsonify({"error": "Invalid email or password"}), 401

        print(f"✅ User found: ID={user.id}, Email={user.email}")

        if not user.check\_password(password):

            print("❌ Invalid password")

            return jsonify({"error": "Invalid email or password"}), 401

        print("✅ Password verified")

        if not user.is\_active:

            print("❌ Account is deactivated")

            return jsonify({"error": "Account is deactivated"}), 403

        print("✅ Account is active")

        print("🔑 Generating JWT token...")

        access\_token = create\_access\_token(

            identity=str(user.id),  # ✅ Fixed to use string identity

            expires\_delta=timedelta(hours=24),

            additional\_claims={"email": user.email}

        )

        print(f"✅ Token generated (length: {len(access\_token)} chars)")

        print(f"   Token preview: {access\_token[:30]}...")

        response\_data = {

            "message": "Login successful",

            "access\_token": access\_token,

            "token\_type": "Bearer",

            "expires\_in": 86400,  # 24 hours

            "user": user.to\_dict()

        }

        print("=" \* 60)

        print("📤 SENDING LOGIN RESPONSE:")

        for k, v in response\_data.items():

            print(f"✅ {k}: {str(v)[:50]}{'...' if isinstance(v, str) and len(str(v)) > 50 else ''}")

        print("=" \* 60)

        return jsonify(response\_data), 200

    except Exception as e:

        print(f"❌ Login error: {str(e)}")

        import traceback; traceback.print\_exc()

        print("=" \* 60)

        return jsonify({"error": f"Login failed: {str(e)}"}), 500

@auth\_bp.route('/logout', methods=['POST'])

@jwt\_required()

def logout():

    """

    Logout user by blacklisting their JWT token

    Headers:

        Authorization: Bearer <access\_token>

    Returns:

        JSON response confirming logout

    """

    try:

        jti = get\_jwt()['jti']

        print("=" \* 60)

        print("🚪 LOGOUT REQUEST")

        print("=" \* 60)

        print(f"Token JTI: {jti}")

        token\_blacklist.add(jti)

        print(f"✅ Token blacklisted ({len(token\_blacklist)} total)")

        print("=" \* 60)

        return jsonify({"message": "Logout successful"}), 200

    except Exception as e:

        print(f"❌ Logout error: {str(e)}")

        print("=" \* 60)

        return jsonify({"error": f"Logout failed: {str(e)}"}), 500

@auth\_bp.route('/me', methods=['GET'])

@jwt\_required()

def get\_current\_user():

    """

    Get current authenticated user information

    Headers:

        Authorization: Bearer <access\_token>

    Returns:

        JSON response with user data

    """

    try:

        current\_user\_id = get\_jwt\_identity()

        print("=" \* 60)

        print("👤 GET CURRENT USER REQUEST")

        print("=" \* 60)

        print(f"User ID from token: {current\_user\_id}")

        user = User.query.get(current\_user\_id)

        if not user:

            print(f"❌ User not found: ID={current\_user\_id}")

            print("=" \* 60)

            return jsonify({"error": "User not found"}), 404

        print(f"✅ User found: {user.email}")

        print("=" \* 60)

        return jsonify({"user": user.to\_dict()}), 200

    except Exception as e:

        print(f"❌ Get user error: {str(e)}")

        print("=" \* 60)

        return jsonify({"error": f"Failed to fetch user: {str(e)}"}), 500

@auth\_bp.route('/refresh', methods=['POST'])

@jwt\_required()

def refresh\_token():

    """

    Refresh JWT access token

    Headers:

        Authorization: Bearer <access\_token>

    Returns:

        JSON response with new access token

    """

    try:

        current\_user\_id = get\_jwt\_identity()

        print("=" \* 60)

        print("🔄 REFRESH TOKEN REQUEST")

        print("=" \* 60)

        print(f"User ID: {current\_user\_id}")

        new\_token = create\_access\_token(

            identity=str(current\_user\_id),  # ✅ Fixed to use string identity

            expires\_delta=timedelta(hours=24)

        )

        print(f"✅ New token generated (length: {len(new\_token)})")

        print("=" \* 60)

        return jsonify({

            "access\_token": new\_token,

            "token\_type": "Bearer",

            "expires\_in": 86400

        }), 200

    except Exception as e:

        print(f"❌ Refresh error: {str(e)}")

        print("=" \* 60)

        return jsonify({"error": f"Token refresh failed: {str(e)}"}), 500

@auth\_bp.before\_app\_request

def check\_if\_token\_revoked():

    """

    Check if token is in blacklist before processing request

    This runs before every request that requires JWT

    """

    try:

        jwt\_data = get\_jwt()

        if jwt\_data and jwt\_data.get('jti') in token\_blacklist:

            print(f"⚠️ Blocked blacklisted token: {jwt\_data.get('jti')}")

            return jsonify({"error": "Token has been revoked"}), 401

    except Exception:

        pass  # No JWT present, continue normally

E:\expense\_tracker\backend\app\routes\expenses.py

"""

Expense Management Routes

Handles CRUD operations for expenses, categories, and payment modes

"""

from flask import Blueprint, request, jsonify

from flask\_jwt\_extended import jwt\_required, get\_jwt\_identity, get\_jwt

from app.extensions import db

from app.models import Expense, Category, PaymentMode

from datetime import datetime, date

from sqlalchemy import func

# Create Blueprint

expenses\_bp = Blueprint('expenses', \_\_name\_\_)

@expenses\_bp.route('/add', methods=['POST'])

@jwt\_required()

def add\_expense():

    """

    Add a new expense

    Request Body:

        {

            "category\_id": 1,

            "payment\_mode\_id": 2,

            "amount": 150.50,

            "description": "Lunch at restaurant",

            "expense\_date": "2025-10-11" (optional, defaults to today)

        }

    Returns:

        JSON response with created expense data

    """

    try:

        current\_user\_id = get\_jwt\_identity()

        print("=" \* 60)

        print("➕ ADD EXPENSE REQUEST")

        print("=" \* 60)

        print(f"👤 User ID from token: {current\_user\_id}")

        print(f"🔑 Token claims: {get\_jwt()}")

        data = request.get\_json()

        if not data:

            print("❌ No data provided")

            return jsonify({"error": "No data provided"}), 400

        category\_id = data.get('category\_id')

        payment\_mode\_id = data.get('payment\_mode\_id')

        amount = data.get('amount')

        description = data.get('description', '').strip()

        expense\_date\_str = data.get('expense\_date')

        print(f"📊 Expense data:")

        print(f"  - Category ID: {category\_id}")

        print(f"  - Payment Mode ID: {payment\_mode\_id}")

        print(f"  - Amount: {amount}")

        print(f"  - Description: {description}")

        print(f"  - Date: {expense\_date\_str or 'Today'}")

        if not category\_id:

            return jsonify({"error": "Category is required"}), 400

        if not payment\_mode\_id:

            return jsonify({"error": "Payment mode is required"}), 400

        if not amount:

            return jsonify({"error": "Amount is required"}), 400

        try:

            amount = float(amount)

            if amount <= 0:

                return jsonify({"error": "Amount must be greater than 0"}), 400

        except ValueError:

            return jsonify({"error": "Invalid amount format"}), 400

        if not description:

            return jsonify({"error": "Description is required"}), 400

        category = Category.query.get(category\_id)

        if not category or not category.is\_active:

            return jsonify({"error": "Invalid or inactive category"}), 400

        payment\_mode = PaymentMode.query.get(payment\_mode\_id)

        if not payment\_mode or not payment\_mode.is\_active:

            return jsonify({"error": "Invalid or inactive payment mode"}), 400

        if expense\_date\_str:

            try:

                expense\_date = datetime.strptime(expense\_date\_str, '%Y-%m-%d').date()

            except ValueError:

                return jsonify({"error": "Invalid date format. Use YYYY-MM-DD"}), 400

        else:

            expense\_date = date.today()

        new\_expense = Expense(

            user\_id=current\_user\_id,

            category\_id=category\_id,

            payment\_mode\_id=payment\_mode\_id,

            amount=amount,

            description=description,

            expense\_date=expense\_date

        )

        db.session.add(new\_expense)

        db.session.commit()

        print(f"✅ Expense added: ID={new\_expense.id}")

        print("=" \* 60)

        return jsonify({

            "message": "Expense added successfully",

            "expense": new\_expense.to\_dict(include\_relations=True)

        }), 201

    except Exception as e:

        print(f"❌ Add expense error: {str(e)}")

        print("=" \* 60)

        db.session.rollback()

        return jsonify({"error": f"Failed to add expense: {str(e)}"}), 500

# ✅ FIXED: Changed from '/list' to '' so GET /api/expenses works

@expenses\_bp.route('', methods=['GET'])

@jwt\_required()

def get\_expenses():

    """

    Get list of user's expenses with optional filters

    Query Parameters:

        - start\_date: Filter by start date (YYYY-MM-DD)

        - end\_date: Filter by end date (YYYY-MM-DD)

        - category\_id: Filter by category

        - payment\_mode\_id: Filter by payment mode

        - limit: Number of results (default: 50)

        - offset: Pagination offset (default: 0)

    Returns:

        JSON response with list of expenses

    """

    try:

        print("=" \* 60)

        print("📋 GET EXPENSES LIST REQUEST")

        print("=" \* 60)

        auth\_header = request.headers.get('Authorization')

        print(f"🔑 Authorization header received: {auth\_header[:50] if auth\_header else 'NONE'}...")

        current\_user\_id = get\_jwt\_identity()

        print(f"👤 User ID from token: {current\_user\_id}")

        jwt\_claims = get\_jwt()

        print(f"📧 Email from token: {jwt\_claims.get('email', 'N/A')}")

        print(f"🔐 Token JTI: {jwt\_claims.get('jti', 'N/A')}")

        query = Expense.query.filter\_by(user\_id=current\_user\_id)

        start\_date = request.args.get('start\_date')

        end\_date = request.args.get('end\_date')

        category\_id = request.args.get('category\_id')

        payment\_mode\_id = request.args.get('payment\_mode\_id')

        print(f"🔍 Filters applied:")

        print(f"  - start\_date: {start\_date or 'None'}")

        print(f"  - end\_date: {end\_date or 'None'}")

        print(f"  - category\_id: {category\_id or 'None'}")

        print(f"  - payment\_mode\_id: {payment\_mode\_id or 'None'}")

        if start\_date:

            query = query.filter(Expense.expense\_date >= start\_date)

        if end\_date:

            query = query.filter(Expense.expense\_date <= end\_date)

        if category\_id:

            query = query.filter(Expense.category\_id == category\_id)

        if payment\_mode\_id:

            query = query.filter(Expense.payment\_mode\_id == payment\_mode\_id)

        limit = request.args.get('limit', 50, type=int)

        offset = request.args.get('offset', 0, type=int)

        total\_count = query.count()

        print(f"📊 Query results:")

        print(f"  - Total count: {total\_count}")

        print(f"  - Limit: {limit}")

        print(f"  - Offset: {offset}")

        expenses = query.order\_by(Expense.expense\_date.desc(), Expense.created\_at.desc()) \

                        .limit(limit) \

                        .offset(offset) \

                        .all()

        print(f"✅ Returning {len(expenses)} expenses")

        print("=" \* 60)

        return jsonify({

            "expenses": [expense.to\_dict(include\_relations=True) for expense in expenses],

            "total\_count": total\_count,

            "limit": limit,

            "offset": offset

        }), 200

    except Exception as e:

        print(f"❌ Get expenses error: {str(e)}")

        print(f"   Error type: {type(e).\_\_name\_\_}")

        import traceback

        print(f"   Traceback: {traceback.format\_exc()}")

        print("=" \* 60)

        return jsonify({"error": f"Failed to fetch expenses: {str(e)}"}), 500

@expenses\_bp.route('/<int:expense\_id>', methods=['GET'])

@jwt\_required()

def get\_expense\_by\_id(expense\_id):

    """

    Get a single expense by ID

    """

    try:

        current\_user\_id = get\_jwt\_identity()

        print(f"🔍 GET EXPENSE BY ID: {expense\_id}")

        print(f"👤 User ID: {current\_user\_id}")

        expense = Expense.query.filter\_by(id=expense\_id, user\_id=current\_user\_id).first()

        if not expense:

            return jsonify({"error": "Expense not found"}), 404

        return jsonify({"expense": expense.to\_dict(include\_relations=True)}), 200

    except Exception as e:

        print(f"❌ Get expense error: {str(e)}")

        return jsonify({"error": f"Failed to fetch expense: {str(e)}"}), 500

@expenses\_bp.route('/<int:expense\_id>', methods=['PUT'])

@jwt\_required()

def update\_expense(expense\_id):

    """

    Update an existing expense

    """

    try:

        current\_user\_id = get\_jwt\_identity()

        print(f"✏️ UPDATE EXPENSE: {expense\_id}")

        expense = Expense.query.filter\_by(id=expense\_id, user\_id=current\_user\_id).first()

        if not expense:

            return jsonify({"error": "Expense not found"}), 404

        data = request.get\_json()

        if not data:

            return jsonify({"error": "No data provided"}), 400

        if 'category\_id' in data:

            category = Category.query.get(data['category\_id'])

            if not category:

                return jsonify({"error": "Invalid category"}), 400

            expense.category\_id = data['category\_id']

        if 'payment\_mode\_id' in data:

            payment\_mode = PaymentMode.query.get(data['payment\_mode\_id'])

            if not payment\_mode:

                return jsonify({"error": "Invalid payment mode"}), 400

            expense.payment\_mode\_id = data['payment\_mode\_id']

        if 'amount' in data:

            try:

                amount = float(data['amount'])

                if amount <= 0:

                    return jsonify({"error": "Amount must be greater than 0"}), 400

                expense.amount = amount

            except ValueError:

                return jsonify({"error": "Invalid amount format"}), 400

        if 'description' in data:

            description = data['description'].strip()

            if not description:

                return jsonify({"error": "Description cannot be empty"}), 400

            expense.description = description

        if 'expense\_date' in data:

            try:

                expense.expense\_date = datetime.strptime(data['expense\_date'], '%Y-%m-%d').date()

            except ValueError:

                return jsonify({"error": "Invalid date format. Use YYYY-MM-DD"}), 400

        db.session.commit()

        print(f"✅ Expense updated: {expense\_id}")

        return jsonify({

            "message": "Expense updated successfully",

            "expense": expense.to\_dict(include\_relations=True)

        }), 200

    except Exception as e:

        print(f"❌ Update expense error: {str(e)}")

        db.session.rollback()

        return jsonify({"error": f"Failed to update expense: {str(e)}"}), 500

@expenses\_bp.route('/<int:expense\_id>', methods=['DELETE'])

@jwt\_required()

def delete\_expense(expense\_id):

    """

    Delete an expense

    """

    try:

        current\_user\_id = get\_jwt\_identity()

        print(f"🗑️ DELETE EXPENSE: {expense\_id}")

        expense = Expense.query.filter\_by(id=expense\_id, user\_id=current\_user\_id).first()

        if not expense:

            return jsonify({"error": "Expense not found"}), 404

        db.session.delete(expense)

        db.session.commit()

        print(f"✅ Expense deleted: {expense\_id}")

        return jsonify({"message": "Expense deleted successfully"}), 200

    except Exception as e:

        print(f"❌ Delete expense error: {str(e)}")

        db.session.rollback()

        return jsonify({"error": f"Failed to delete expense: {str(e)}"}), 500

@expenses\_bp.route('/categories', methods=['GET'])

@jwt\_required()

def get\_categories():

    """

    Get all active categories

    """

    try:

        print("📁 GET CATEGORIES REQUEST")

        current\_user\_id = get\_jwt\_identity()

        print(f"👤 User ID: {current\_user\_id}")

        categories = Category.query.filter\_by(is\_active=True).all()

        print(f"✅ Found {len(categories)} categories")

        return jsonify({

            "categories": [category.to\_dict() for category in categories]

        }), 200

    except Exception as e:

        print(f"❌ Get categories error: {str(e)}")

        return jsonify({"error": f"Failed to fetch categories: {str(e)}"}), 500

@expenses\_bp.route('/payment-modes', methods=['GET'])

@jwt\_required()

def get\_payment\_modes():

    """

    Get all active payment modes

    """

    try:

        print("💳 GET PAYMENT MODES REQUEST")

        current\_user\_id = get\_jwt\_identity()

        print(f"👤 User ID: {current\_user\_id}")

        payment\_modes = PaymentMode.query.filter\_by(is\_active=True).all()

        print(f"✅ Found {len(payment\_modes)} payment modes")

        return jsonify({

            "payment\_modes": [pm.to\_dict() for pm in payment\_modes]

        }), 200

    except Exception as e:

        print(f"❌ Get payment modes error: {str(e)}")

        return jsonify({"error": f"Failed to fetch payment modes: {str(e)}"}), 500

@expenses\_bp.route('/summary', methods=['GET'])

@jwt\_required()

def get\_expense\_summary():

    """

    Get expense summary for dashboard

    """

    try:

        print("📊 GET EXPENSE SUMMARY REQUEST")

        current\_user\_id = get\_jwt\_identity()

        print(f"👤 User ID: {current\_user\_id}")

        query = Expense.query.filter\_by(user\_id=current\_user\_id)

        start\_date = request.args.get('start\_date')

        end\_date = request.args.get('end\_date')

        if start\_date:

            query = query.filter(Expense.expense\_date >= start\_date)

        if end\_date:

            query = query.filter(Expense.expense\_date <= end\_date)

        total\_amount = query.with\_entities(func.sum(Expense.amount)).scalar() or 0

        total\_expenses = query.count()

        today = date.today()

        today\_total = Expense.query.filter\_by(user\_id=current\_user\_id) \

                                    .filter(Expense.expense\_date == today) \

                                    .with\_entities(func.sum(Expense.amount)) \

                                    .scalar() or 0

        print(f"✅ Summary calculated:")

        print(f"  - Total amount: ₹{total\_amount}")

        print(f"  - Total expenses: {total\_expenses}")

        print(f"  - Today's total: ₹{today\_total}")

        return jsonify({

            "total\_amount": float(total\_amount),

            "total\_expenses": total\_expenses,

            "today\_total": float(today\_total),

            "filters\_applied": {

                "start\_date": start\_date,

                "end\_date": end\_date

            }

        }), 200

    except Exception as e:

        print(f"❌ Get summary error: {str(e)}")

        return jsonify({"error": f"Failed to fetch summary: {str(e)}"}), 500

E:\expense\_tracker\backend\app\routes\export.py

"""

Export Routes

Handles PDF and CSV export functionality with professional formatting

"""

from flask import Blueprint, request, send\_file, jsonify

from flask\_jwt\_extended import jwt\_required, get\_jwt\_identity

from app.services.export\_service import ExportService

from app.services.auth\_service import AuthService

from app.utils.helpers import parse\_date, error\_response

from datetime import datetime

# Create Blueprint

export\_bp = Blueprint('export', \_\_name\_\_)

@export\_bp.route('/csv', methods=['GET'])

@jwt\_required()

def export\_csv():

    """

    Export expenses to CSV file with proper UTF-8 encoding for ₹ symbol

    Query Parameters:

        - start\_date: Start date (YYYY-MM-DD)

        - end\_date: End date (YYYY-MM-DD)

        - category\_id: Filter by category

        - payment\_mode\_id: Filter by payment mode

        - period: Predefined period (today, week, month, year, all)

        - include\_summary: Include summary statistics (true/false)

    Returns:

        CSV file download with UTF-8 BOM encoding

    """

    try:

        # Get current user

        current\_user\_id = get\_jwt\_identity()

        user = AuthService.get\_user\_by\_id(current\_user\_id)

        if not user:

            return error\_response("User not found", 404)

        # Get filters from query parameters

        period = request.args.get('period', 'all')

        category\_id = request.args.get('category\_id', type=int)

        payment\_mode\_id = request.args.get('payment\_mode\_id', type=int)

        include\_summary = request.args.get('include\_summary', 'true').lower() == 'true'

        # Determine date range

        if period and period != 'custom' and period != 'all':

            start\_date, end\_date = ExportService.get\_period\_dates(period)

        elif period == 'all':

            start\_date, end\_date = None, None

        else:

            start\_date\_str = request.args.get('start\_date')

            end\_date\_str = request.args.get('end\_date')

            start\_date = parse\_date(start\_date\_str) if start\_date\_str else None

            end\_date = parse\_date(end\_date\_str) if end\_date\_str else None

        # Generate CSV with UTF-8 encoding

        csv\_buffer = ExportService.export\_to\_csv(

            user\_id=current\_user\_id,

            user\_name=user.full\_name or user.email,

            start\_date=start\_date,

            end\_date=end\_date,

            category\_id=category\_id,

            payment\_mode\_id=payment\_mode\_id,

            include\_summary=include\_summary

        )

        if not csv\_buffer:

            return error\_response("No data available for export", 404)

        # Generate filename with period info

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

        period\_label = period.replace('\_', '-') if period else 'custom'

        filename = f"expenses\_{period\_label}\_{timestamp}.csv"

        # Send file with UTF-8 BOM for proper Excel compatibility

        return send\_file(

            csv\_buffer,

            mimetype='text/csv; charset=utf-8',

            as\_attachment=True,

            download\_name=filename

        )

    except ValueError as ve:

        return error\_response(str(ve), 400)

    except Exception as e:

        print(f"CSV export error: {str(e)}")

        return error\_response(f"CSV export failed: {str(e)}", 500)

@export\_bp.route('/pdf', methods=['GET'])

@jwt\_required()

def export\_pdf():

    """

    Export expenses to professional PDF report with charts and summaries

    Query Parameters:

        - start\_date: Start date (YYYY-MM-DD)

        - end\_date: End date (YYYY-MM-DD)

        - category\_id: Filter by category

        - payment\_mode\_id: Filter by payment mode

        - period: Predefined period (today, week, month, year, all)

        - report\_type: 'detailed', 'summary', or 'analytics' (default: detailed)

        - include\_charts: Include visual charts (true/false, default: true)

        - group\_by: Group expenses by 'category', 'payment\_mode', 'date', or 'none'

    Returns:

        Professional PDF file download with ₹ symbol support

    """

    try:

        # Get current user

        current\_user\_id = get\_jwt\_identity()

        user = AuthService.get\_user\_by\_id(current\_user\_id)

        if not user:

            return error\_response("User not found", 404)

        # Get filters from query parameters

        period = request.args.get('period', 'month')

        category\_id = request.args.get('category\_id', type=int)

        payment\_mode\_id = request.args.get('payment\_mode\_id', type=int)

        report\_type = request.args.get('report\_type', 'detailed')

        include\_charts = request.args.get('include\_charts', 'true').lower() == 'true'

        group\_by = request.args.get('group\_by', 'none')

        # Validate parameters

        if report\_type not in ['detailed', 'summary', 'analytics']:

            report\_type = 'detailed'

        if group\_by not in ['category', 'payment\_mode', 'date', 'none']:

            group\_by = 'none'

        # Determine date range

        if period and period != 'custom' and period != 'all':

            start\_date, end\_date = ExportService.get\_period\_dates(period)

        elif period == 'all':

            start\_date, end\_date = None, None

        else:

            start\_date\_str = request.args.get('start\_date')

            end\_date\_str = request.args.get('end\_date')

            start\_date = parse\_date(start\_date\_str) if start\_date\_str else None

            end\_date = parse\_date(end\_date\_str) if end\_date\_str else None

        # Generate professional PDF

        pdf\_buffer = ExportService.export\_to\_pdf(

            user\_id=current\_user\_id,

            user\_name=user.full\_name or user.email,

            user\_email=user.email,

            start\_date=start\_date,

            end\_date=end\_date,

            category\_id=category\_id,

            payment\_mode\_id=payment\_mode\_id,

            report\_type=report\_type,

            include\_charts=include\_charts,

            group\_by=group\_by

        )

        if not pdf\_buffer:

            return error\_response("No data available for export", 404)

        # Generate descriptive filename

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

        period\_label = period.replace('\_', '-') if period else 'custom'

        filename = f"expense\_report\_{report\_type}\_{period\_label}\_{timestamp}.pdf"

        # Send file

        return send\_file(

            pdf\_buffer,

            mimetype='application/pdf',

            as\_attachment=True,

            download\_name=filename

        )

    except ValueError as ve:

        return error\_response(str(ve), 400)

    except Exception as e:

        print(f"PDF export error: {str(e)}")

        return error\_response(f"PDF export failed: {str(e)}", 500)

@export\_bp.route('/preview', methods=['GET'])

@jwt\_required()

def preview\_export():

    """

    Get preview data for export (first 10 records + summary)

    Query Parameters: Same as export endpoints

    Returns:

        JSON with preview data and statistics

    """

    try:

        current\_user\_id = get\_jwt\_identity()

        # Get filters

        period = request.args.get('period', 'month')

        category\_id = request.args.get('category\_id', type=int)

        payment\_mode\_id = request.args.get('payment\_mode\_id', type=int)

        # Determine date range

        if period and period != 'custom' and period != 'all':

            start\_date, end\_date = ExportService.get\_period\_dates(period)

        elif period == 'all':

            start\_date, end\_date = None, None

        else:

            start\_date\_str = request.args.get('start\_date')

            end\_date\_str = request.args.get('end\_date')

            start\_date = parse\_date(start\_date\_str) if start\_date\_str else None

            end\_date = parse\_date(end\_date\_str) if end\_date\_str else None

        # Get preview data

        preview\_data = ExportService.get\_export\_preview(

            user\_id=current\_user\_id,

            start\_date=start\_date,

            end\_date=end\_date,

            category\_id=category\_id,

            payment\_mode\_id=payment\_mode\_id

        )

        return jsonify(preview\_data), 200

    except Exception as e:

        return error\_response(f"Preview failed: {str(e)}", 500)

@export\_bp.route('/formats', methods=['GET'])

@jwt\_required()

def get\_export\_formats():

    """

    Get available export formats, periods, and options

    Returns:

        JSON response with comprehensive export options

    """

    return jsonify({

        "formats": [

            {"value": "csv", "label": "CSV (Excel Compatible)", "icon": "📊"},

            {"value": "pdf", "label": "PDF Report", "icon": "📄"}

        ],

        "periods": [

            {"value": "today", "label": "Today"},

            {"value": "yesterday", "label": "Yesterday"},

            {"value": "week", "label": "This Week"},

            {"value": "last\_week", "label": "Last Week"},

            {"value": "month", "label": "This Month"},

            {"value": "last\_month", "label": "Last Month"},

            {"value": "quarter", "label": "This Quarter"},

            {"value": "year", "label": "This Year"},

            {"value": "last\_year", "label": "Last Year"},

            {"value": "all", "label": "All Time"},

            {"value": "custom", "label": "Custom Date Range"}

        ],

        "report\_types": [

            {

                "value": "summary",

                "label": "Summary Report",

                "description": "Overview with totals and charts"

            },

            {

                "value": "detailed",

                "label": "Detailed Report",

                "description": "All transactions with complete information"

            },

            {

                "value": "analytics",

                "label": "Analytics Report",

                "description": "Advanced insights with trends and breakdowns"

            }

        ],

        "group\_by\_options": [

            {"value": "none", "label": "No Grouping"},

            {"value": "category", "label": "Group by Category"},

            {"value": "payment\_mode", "label": "Group by Payment Mode"},

            {"value": "date", "label": "Group by Date"}

        ],

        "currency": {

            "symbol": "₹",

            "code": "INR",

            "name": "Indian Rupee"

        }

    }), 200

@export\_bp.route('/history', methods=['GET'])

@jwt\_required()

def get\_export\_history():

    """

    Get user's export history (last 10 exports)

    Returns:

        JSON with export history

    """

    try:

        current\_user\_id = get\_jwt\_identity()

        # This would typically come from a database table tracking exports

        # For now, return a placeholder

        history = ExportService.get\_export\_history(current\_user\_id)

        return jsonify({

            "exports": history,

            "total\_count": len(history)

        }), 200

    except Exception as e:

        return error\_response(f"Failed to fetch history: {str(e)}", 500)

E:\expense\_tracker\backend\app\services

E:\expense\_tracker\backend\app\services\\_\_init\_\_.py

"""

Services Package Initialization

Exports all service classes for business logic

"""

from app.services.auth\_service import AuthService

from app.services.expense\_service import ExpenseService

from app.services.analytics\_service import AnalyticsService

from app.services.openai\_service import OpenAIService  # ✅ CHANGED

from app.services.export\_service import ExportService

# Export all services

\_\_all\_\_ = [

    'AuthService',

    'ExpenseService',

    'AnalyticsService',

    'OpenAIService',  # ✅ CHANGED

    'ExportService'

]

E:\expense\_tracker\backend\app\services\analytics\_service.py

"""

Analytics Service

Business logic for generating expense analytics and reports

"""

from app.extensions import db

from app.models import Expense, Category, PaymentMode

from sqlalchemy import func, and\_

from datetime import datetime, date, timedelta

from collections import defaultdict

class AnalyticsService:

    """

    Service class for analytics and reporting operations

    Handles data aggregation for charts and dashboards

    """

    @staticmethod

    def get\_category\_breakdown(user\_id, start\_date=None, end\_date=None):

        """

        Get expense breakdown by category

        Args:

            user\_id (int): User ID

            start\_date (date, optional): Filter start date

            end\_date (date, optional): Filter end date

        Returns:

            list: List of dicts with category data

        """

        try:

            # Build query

            query = db.session.query(

                Category.name,

                Category.color,

                Category.icon,

                func.sum(Expense.amount).label('total'),

                func.count(Expense.id).label('count')

            ).join(Expense, Category.id == Expense.category\_id) \

             .filter(Expense.user\_id == user\_id)

            # Apply date filters

            if start\_date:

                query = query.filter(Expense.expense\_date >= start\_date)

            if end\_date:

                query = query.filter(Expense.expense\_date <= end\_date)

            # Group and order

            results = query.group\_by(Category.id, Category.name, Category.color, Category.icon) \

                          .order\_by(func.sum(Expense.amount).desc()) \

                          .all()

            # Format results

            data = []

            total\_sum = sum(float(row.total) for row in results)

            for row in results:

                amount = float(row.total)

                percentage = (amount / total\_sum \* 100) if total\_sum > 0 else 0

                data.append({

                    'category': row.name,

                    'color': row.color,

                    'icon': row.icon,

                    'total': amount,

                    'count': row.count,

                    'percentage': round(percentage, 2)

                })

            return data

        except Exception as e:

            print(f"Category breakdown error: {str(e)}")

            return []

    @staticmethod

    def get\_payment\_mode\_breakdown(user\_id, start\_date=None, end\_date=None):

        """

        Get expense breakdown by payment mode

        Args:

            user\_id (int): User ID

            start\_date (date, optional): Filter start date

            end\_date (date, optional): Filter end date

        Returns:

            list: List of dicts with payment mode data

        """

        try:

            # Build query

            query = db.session.query(

                PaymentMode.name,

                PaymentMode.bank\_name,

                PaymentMode.type,

                func.sum(Expense.amount).label('total'),

                func.count(Expense.id).label('count')

            ).join(Expense, PaymentMode.id == Expense.payment\_mode\_id) \

             .filter(Expense.user\_id == user\_id)

            # Apply date filters

            if start\_date:

                query = query.filter(Expense.expense\_date >= start\_date)

            if end\_date:

                query = query.filter(Expense.expense\_date <= end\_date)

            # Group and order

            results = query.group\_by(

                PaymentMode.id,

                PaymentMode.name,

                PaymentMode.bank\_name,

                PaymentMode.type

            ).order\_by(func.sum(Expense.amount).desc()).all()

            # Format results

            data = []

            total\_sum = sum(float(row.total) for row in results)

            for row in results:

                amount = float(row.total)

                percentage = (amount / total\_sum \* 100) if total\_sum > 0 else 0

                display\_name = f"{row.name} - {row.bank\_name}" if row.bank\_name else row.name

                data.append({

                    'payment\_mode': display\_name,

                    'type': row.type,

                    'total': amount,

                    'count': row.count,

                    'percentage': round(percentage, 2)

                })

            return data

        except Exception as e:

            print(f"Payment mode breakdown error: {str(e)}")

            return []

    @staticmethod

    def get\_payment\_vs\_category\_matrix(user\_id, start\_date=None, end\_date=None):

        """

        Get cross-tabulation of payment modes vs categories

        Args:

            user\_id (int): User ID

            start\_date (date, optional): Filter start date

            end\_date (date, optional): Filter end date

        Returns:

            list: Matrix data for visualization

        """

        try:

            # Build query

            query = db.session.query(

                Category.name.label('category'),

                PaymentMode.name.label('payment\_mode'),

                PaymentMode.bank\_name,

                func.sum(Expense.amount).label('total')

            ).join(Expense, Category.id == Expense.category\_id) \

             .join(PaymentMode, Expense.payment\_mode\_id == PaymentMode.id) \

             .filter(Expense.user\_id == user\_id)

            # Apply date filters

            if start\_date:

                query = query.filter(Expense.expense\_date >= start\_date)

            if end\_date:

                query = query.filter(Expense.expense\_date <= end\_date)

            # Group by both dimensions

            results = query.group\_by(

                Category.name,

                PaymentMode.name,

                PaymentMode.bank\_name

            ).all()

            # Build matrix structure

            matrix = defaultdict(dict)

            for row in results:

                payment\_display = f"{row.payment\_mode} - {row.bank\_name}" if row.bank\_name else row.payment\_mode

                matrix[row.category][payment\_display] = float(row.total)

            # Convert to list format

            data = [

                {

                    'category': category,

                    'payments': dict(payments)

                }

                for category, payments in matrix.items()

            ]

            return data

        except Exception as e:

            print(f"Matrix calculation error: {str(e)}")

            return []

    @staticmethod

    def get\_daily\_trend(user\_id, days=7):

        """

        Get daily expense trend

        Args:

            user\_id (int): User ID

            days (int): Number of days to look back

        Returns:

            list: Daily totals

        """

        try:

            # Calculate date range

            end\_date = date.today()

            start\_date = end\_date - timedelta(days=days-1)

            # Query daily totals

            results = db.session.query(

                Expense.expense\_date,

                func.sum(Expense.amount).label('total')

            ).filter(

                and\_(

                    Expense.user\_id == user\_id,

                    Expense.expense\_date >= start\_date,

                    Expense.expense\_date <= end\_date

                )

            ).group\_by(Expense.expense\_date) \

             .order\_by(Expense.expense\_date) \

             .all()

            # Create lookup dictionary

            expense\_dict = {row.expense\_date: float(row.total) for row in results}

            # Fill in all dates (including days with no expenses)

            data = []

            current\_date = start\_date

            while current\_date <= end\_date:

                data.append({

                    'date': current\_date.isoformat(),

                    'total': expense\_dict.get(current\_date, 0),

                    'day\_name': current\_date.strftime('%A')

                })

                current\_date += timedelta(days=1)

            return data

        except Exception as e:

            print(f"Daily trend error: {str(e)}")

            return []

    @staticmethod

    def get\_monthly\_summary(user\_id, year=None):

        """

        Get monthly expense summary for a year

        Args:

            user\_id (int): User ID

            year (int, optional): Year to analyze (defaults to current year)

        Returns:

            list: Monthly totals

        """

        try:

            if year is None:

                year = date.today().year

            # Query monthly totals

            results = db.session.query(

                func.extract('month', Expense.expense\_date).label('month'),

                func.sum(Expense.amount).label('total'),

                func.count(Expense.id).label('count')

            ).filter(

                and\_(

                    Expense.user\_id == user\_id,

                    func.extract('year', Expense.expense\_date) == year

                )

            ).group\_by(func.extract('month', Expense.expense\_date)) \

             .order\_by(func.extract('month', Expense.expense\_date)) \

             .all()

            # Month names

            month\_names = [

                "January", "February", "March", "April", "May", "June",

                "July", "August", "September", "October", "November", "December"

            ]

            # Create lookup dictionary

            expense\_dict = {int(row.month): {'total': float(row.total), 'count': row.count} for row in results}

            # Fill all 12 months

            data = []

            for i in range(12):

                month\_num = i + 1

                month\_data = expense\_dict.get(month\_num, {'total': 0, 'count': 0})

                data.append({

                    'month': month\_names[i],

                    'month\_number': month\_num,

                    'total': month\_data['total'],

                    'count': month\_data['count']

                })

            return data

        except Exception as e:

            print(f"Monthly summary error: {str(e)}")

            return []

    @staticmethod

    def get\_top\_expenses(user\_id, limit=10, start\_date=None, end\_date=None):

        """

        Get top N expenses by amount

        Args:

            user\_id (int): User ID

            limit (int): Number of results

            start\_date (date, optional): Filter start date

            end\_date (date, optional): Filter end date

        Returns:

            list: Top expenses

        """

        try:

            query = Expense.query.filter\_by(user\_id=user\_id)

            if start\_date:

                query = query.filter(Expense.expense\_date >= start\_date)

            if end\_date:

                query = query.filter(Expense.expense\_date <= end\_date)

            expenses = query.order\_by(Expense.amount.desc()).limit(limit).all()

            return [expense.to\_dict(include\_relations=True) for expense in expenses]

        except Exception:

            return []

    @staticmethod

    def get\_spending\_insights(user\_id, days=30):

        """

        Generate spending insights and statistics

        Args:

            user\_id (int): User ID

            days (int): Number of days to analyze

        Returns:

            dict: Insights and statistics

        """

        try:

            end\_date = date.today()

            start\_date = end\_date - timedelta(days=days)

            # Total spending

            total = db.session.query(func.sum(Expense.amount)) \

                             .filter(Expense.user\_id == user\_id) \

                             .filter(Expense.expense\_date >= start\_date) \

                             .scalar() or 0

            # Daily average

            daily\_avg = float(total) / days if days > 0 else 0

            # Highest category

            top\_category = db.session.query(

                Category.name,

                func.sum(Expense.amount).label('total')

            ).join(Expense) \

             .filter(Expense.user\_id == user\_id) \

             .filter(Expense.expense\_date >= start\_date) \

             .group\_by(Category.name) \

             .order\_by(func.sum(Expense.amount).desc()) \

             .first()

            # Most used payment mode

            top\_payment = db.session.query(

                PaymentMode.name,

                PaymentMode.bank\_name,

                func.count(Expense.id).label('count')

            ).join(Expense) \

             .filter(Expense.user\_id == user\_id) \

             .filter(Expense.expense\_date >= start\_date) \

             .group\_by(PaymentMode.name, PaymentMode.bank\_name) \

             .order\_by(func.count(Expense.id).desc()) \

             .first()

            return {

                'period\_days': days,

                'total\_spent': float(total),

                'daily\_average': round(daily\_avg, 2),

                'top\_category': {

                    'name': top\_category.name if top\_category else None,

                    'amount': float(top\_category.total) if top\_category else 0

                },

                'most\_used\_payment': {

                    'name': f"{top\_payment.name} - {top\_payment.bank\_name}" if top\_payment and top\_payment.bank\_name else (top\_payment.name if top\_payment else None),

                    'usage\_count': top\_payment.count if top\_payment else 0

                }

            }

        except Exception as e:

            print(f"Insights error: {str(e)}")

            return {}

E:\expense\_tracker\backend\app\services\auth\_service.py

"""

Authentication Service

Business logic for user authentication, registration, and token management

"""

from app.extensions import db

from app.models import User

from flask\_jwt\_extended import create\_access\_token

from datetime import timedelta

import re

class AuthService:

    """

    Service class for authentication-related operations

    Handles user registration, login, validation, and token generation

    """

    @staticmethod

    def validate\_email(email):

        """

        Validate email format using regex

        Args:

            email (str): Email address to validate

        Returns:

            bool: True if valid email format, False otherwise

        """

        if not email:

            return False

        # RFC 5322 compliant email regex pattern

        pattern = r'^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

        return re.match(pattern, email) is not None

    @staticmethod

    def validate\_password(password):

        """

        Validate password strength requirements

        Args:

            password (str): Password to validate

        Returns:

            tuple: (is\_valid: bool, error\_message: str)

        """

        if not password:

            return False, "Password is required"

        if len(password) < 6:

            return False, "Password must be at least 6 characters long"

        if len(password) > 128:

            return False, "Password is too long (maximum 128 characters)"

        # Optional: Add more strength requirements

        # has\_uppercase = any(c.isupper() for c in password)

        # has\_lowercase = any(c.islower() for c in password)

        # has\_digit = any(c.isdigit() for c in password)

        return True, ""

    @staticmethod

    def user\_exists(email):

        """

        Check if user with given email already exists

        Args:

            email (str): Email address to check

        Returns:

            bool: True if user exists, False otherwise

        """

        email\_lower = email.strip().lower()

        existing\_user = User.query.filter\_by(email=email\_lower).first()

        return existing\_user is not None

    @staticmethod

    def register\_user(email, password, full\_name=None):

        """

        Register a new user account

        Args:

            email (str): User's email address

            password (str): User's password (will be hashed)

            full\_name (str, optional): User's full name

        Returns:

            tuple: (success: bool, user\_or\_error: User|str)

        """

        try:

            # Normalize email

            email = email.strip().lower()

            # Validate email format

            if not AuthService.validate\_email(email):

                return False, "Invalid email format"

            # Validate password

            is\_valid, error\_msg = AuthService.validate\_password(password)

            if not is\_valid:

                return False, error\_msg

            # Check if user already exists

            if AuthService.user\_exists(email):

                return False, "User with this email already exists"

            # Sanitize full\_name

            if full\_name:

                full\_name = full\_name.strip()

                if len(full\_name) > 100:

                    return False, "Full name is too long"

            # Create new user (password will be hashed in User model)

            new\_user = User(

                email=email,

                password=password,

                full\_name=full\_name if full\_name else None

            )

            # Save to database

            db.session.add(new\_user)

            db.session.commit()

            return True, new\_user

        except Exception as e:

            db.session.rollback()

            return False, f"Registration failed: {str(e)}"

    @staticmethod

    def authenticate\_user(email, password):

        """

        Authenticate user credentials

        Args:

            email (str): User's email address

            password (str): User's password

        Returns:

            tuple: (success: bool, user\_or\_error: User|str)

        """

        try:

            # Normalize email

            email = email.strip().lower()

            # Validate inputs

            if not email or not password:

                return False, "Email and password are required"

            # Find user by email

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

            # Check if user exists

            if not user:

                return False, "Invalid email or password"

            # Verify password

            if not user.check\_password(password):

                return False, "Invalid email or password"

            # Check if account is active

            if not user.is\_active:

                return False, "Account is deactivated. Please contact support."

            return True, user

        except Exception as e:

            return False, f"Authentication failed: {str(e)}"

    @staticmethod

    def generate\_access\_token(user\_id, email, expires\_hours=24):

        """

        Generate JWT access token for authenticated user

        Args:

            user\_id (int): User's ID

            email (str): User's email

            expires\_hours (int): Token expiration time in hours (default: 24)

        Returns:

            str: JWT access token

        """

        try:

            access\_token = create\_access\_token(

                identity=user\_id,

                expires\_delta=timedelta(hours=expires\_hours),

                additional\_claims={

                    "email": email,

                    "token\_type": "access"

                }

            )

            return access\_token

        except Exception as e:

            raise Exception(f"Token generation failed: {str(e)}")

    @staticmethod

    def get\_user\_by\_id(user\_id):

        """

        Retrieve user by ID

        Args:

            user\_id (int): User's ID

        Returns:

            User|None: User object if found, None otherwise

        """

        try:

            return User.query.get(user\_id)

        except Exception:

            return None

    @staticmethod

    def get\_user\_by\_email(email):

        """

        Retrieve user by email address

        Args:

            email (str): User's email address

        Returns:

            User|None: User object if found, None otherwise

        """

        try:

            email = email.strip().lower()

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

        except Exception:

            return None

    @staticmethod

    def update\_user\_profile(user\_id, full\_name=None):

        """

        Update user profile information

        Args:

            user\_id (int): User's ID

            full\_name (str, optional): New full name

        Returns:

            tuple: (success: bool, user\_or\_error: User|str)

        """

        try:

            user = User.query.get(user\_id)

            if not user:

                return False, "User not found"

            # Update full name if provided

            if full\_name is not None:

                full\_name = full\_name.strip()

                if len(full\_name) > 100:

                    return False, "Full name is too long"

                user.full\_name = full\_name if full\_name else None

            db.session.commit()

            return True, user

        except Exception as e:

            db.session.rollback()

            return False, f"Update failed: {str(e)}"

    @staticmethod

    def change\_password(user\_id, old\_password, new\_password):

        """

        Change user password

        Args:

            user\_id (int): User's ID

            old\_password (str): Current password

            new\_password (str): New password

        Returns:

            tuple: (success: bool, message: str)

        """

        try:

            user = User.query.get(user\_id)

            if not user:

                return False, "User not found"

            # Verify old password

            if not user.check\_password(old\_password):

                return False, "Current password is incorrect"

            # Validate new password

            is\_valid, error\_msg = AuthService.validate\_password(new\_password)

            if not is\_valid:

                return False, error\_msg

            # Check if new password is same as old password

            if old\_password == new\_password:

                return False, "New password must be different from current password"

            # Update password (will be hashed automatically)

            user.set\_password(new\_password)

            db.session.commit()

            return True, "Password changed successfully"

        except Exception as e:

            db.session.rollback()

            return False, f"Password change failed: {str(e)}"

    @staticmethod

    def deactivate\_account(user\_id):

        """

        Deactivate user account (soft delete)

        Args:

            user\_id (int): User's ID

        Returns:

            tuple: (success: bool, message: str)

        """

        try:

            user = User.query.get(user\_id)

            if not user:

                return False, "User not found"

            user.is\_active = False

            db.session.commit()

            return True, "Account deactivated successfully"

        except Exception as e:

            db.session.rollback()

            return False, f"Deactivation failed: {str(e)}"

E:\expense\_tracker\backend\app\services\expense\_service.py

"""

Expense Service

Business logic for expense management operations

"""

from app.extensions import db

from app.models import Expense, Category, PaymentMode

from datetime import datetime, date

from sqlalchemy import func, and\_

class ExpenseService:

    """

    Service class for expense-related operations

    Handles CRUD operations, validation, and calculations

    """

    @staticmethod

    def validate\_amount(amount):

        """

        Validate expense amount

        Args:

            amount: Amount value to validate

        Returns:

            tuple: (is\_valid: bool, validated\_amount\_or\_error: float|str)

        """

        try:

            amount = float(amount)

            if amount <= 0:

                return False, "Amount must be greater than 0"

            if amount > 99999999.99:

                return False, "Amount is too large"

            return True, round(amount, 2)

        except (ValueError, TypeError):

            return False, "Invalid amount format"

    @staticmethod

    def validate\_description(description):

        """

        Validate expense description

        Args:

            description (str): Description text

        Returns:

            tuple: (is\_valid: bool, error\_message: str)

        """

        if not description or not description.strip():

            return False, "Description is required"

        description = description.strip()

        if len(description) < 3:

            return False, "Description must be at least 3 characters long"

        if len(description) > 255:

            return False, "Description is too long (maximum 255 characters)"

        return True, ""

    @staticmethod

    def validate\_date(date\_str):

        """

        Validate and parse expense date

        Args:

            date\_str (str): Date string in YYYY-MM-DD format

        Returns:

            tuple: (is\_valid: bool, date\_or\_error: date|str)

        """

        try:

            parsed\_date = datetime.strptime(date\_str, '%Y-%m-%d').date()

            # Check if date is not in future

            if parsed\_date > date.today():

                return False, "Expense date cannot be in the future"

            # Check if date is not too old (optional: 5 years limit)

            from datetime import timedelta

            five\_years\_ago = date.today() - timedelta(days=365\*5)

            if parsed\_date < five\_years\_ago:

                return False, "Expense date is too old"

            return True, parsed\_date

        except ValueError:

            return False, "Invalid date format. Use YYYY-MM-DD"

    @staticmethod

    def create\_expense(user\_id, category\_id, payment\_mode\_id, amount, description, expense\_date=None):

        """

        Create a new expense record

        Args:

            user\_id (int): User ID

            category\_id (int): Category ID

            payment\_mode\_id (int): Payment mode ID

            amount (float): Expense amount

            description (str): Expense description

            expense\_date (date, optional): Date of expense

        Returns:

            tuple: (success: bool, expense\_or\_error: Expense|str)

        """

        try:

            # Validate amount

            is\_valid, validated\_amount = ExpenseService.validate\_amount(amount)

            if not is\_valid:

                return False, validated\_amount

            # Validate description

            is\_valid, error\_msg = ExpenseService.validate\_description(description)

            if not is\_valid:

                return False, error\_msg

            description = description.strip()

            # Verify category exists and is active

            category = Category.query.get(category\_id)

            if not category:

                return False, "Invalid category"

            if not category.is\_active:

                return False, "Selected category is inactive"

            # Verify payment mode exists and is active

            payment\_mode = PaymentMode.query.get(payment\_mode\_id)

            if not payment\_mode:

                return False, "Invalid payment mode"

            if not payment\_mode.is\_active:

                return False, "Selected payment mode is inactive"

            # Use today if date not provided

            if expense\_date is None:

                expense\_date = date.today()

            # Create new expense

            new\_expense = Expense(

                user\_id=user\_id,

                category\_id=category\_id,

                payment\_mode\_id=payment\_mode\_id,

                amount=validated\_amount,

                description=description,

                expense\_date=expense\_date

            )

            # Save to database

            db.session.add(new\_expense)

            db.session.commit()

            return True, new\_expense

        except Exception as e:

            db.session.rollback()

            return False, f"Failed to create expense: {str(e)}"

    @staticmethod

    def get\_expense\_by\_id(expense\_id, user\_id):

        """

        Get expense by ID (with user ownership check)

        Args:

            expense\_id (int): Expense ID

            user\_id (int): User ID for ownership verification

        Returns:

            Expense|None: Expense object if found and owned by user

        """

        try:

            return Expense.query.filter\_by(id=expense\_id, user\_id=user\_id).first()

        except Exception:

            return None

    @staticmethod

    def get\_user\_expenses(user\_id, start\_date=None, end\_date=None, category\_id=None,

                         payment\_mode\_id=None, limit=50, offset=0):

        """

        Get list of user expenses with filters

        Args:

            user\_id (int): User ID

            start\_date (date, optional): Filter start date

            end\_date (date, optional): Filter end date

            category\_id (int, optional): Filter by category

            payment\_mode\_id (int, optional): Filter by payment mode

            limit (int): Maximum number of results

            offset (int): Pagination offset

        Returns:

            tuple: (expenses: list, total\_count: int)

        """

        try:

            # Build query

            query = Expense.query.filter\_by(user\_id=user\_id)

            # Apply filters

            if start\_date:

                query = query.filter(Expense.expense\_date >= start\_date)

            if end\_date:

                query = query.filter(Expense.expense\_date <= end\_date)

            if category\_id:

                query = query.filter(Expense.category\_id == category\_id)

            if payment\_mode\_id:

                query = query.filter(Expense.payment\_mode\_id == payment\_mode\_id)

            # Get total count

            total\_count = query.count()

            # Apply pagination and ordering

            expenses = query.order\_by(

                Expense.expense\_date.desc(),

                Expense.created\_at.desc()

            ).limit(limit).offset(offset).all()

            return expenses, total\_count

        except Exception:

            return [], 0

    @staticmethod

    def update\_expense(expense\_id, user\_id, \*\*kwargs):

        """

        Update an existing expense

        Args:

            expense\_id (int): Expense ID

            user\_id (int): User ID for ownership verification

            \*\*kwargs: Fields to update (category\_id, payment\_mode\_id, amount, description, expense\_date)

        Returns:

            tuple: (success: bool, expense\_or\_error: Expense|str)

        """

        try:

            # Find expense

            expense = ExpenseService.get\_expense\_by\_id(expense\_id, user\_id)

            if not expense:

                return False, "Expense not found"

            # Update category if provided

            if 'category\_id' in kwargs:

                category = Category.query.get(kwargs['category\_id'])

                if not category:

                    return False, "Invalid category"

                expense.category\_id = kwargs['category\_id']

            # Update payment mode if provided

            if 'payment\_mode\_id' in kwargs:

                payment\_mode = PaymentMode.query.get(kwargs['payment\_mode\_id'])

                if not payment\_mode:

                    return False, "Invalid payment mode"

                expense.payment\_mode\_id = kwargs['payment\_mode\_id']

            # Update amount if provided

            if 'amount' in kwargs:

                is\_valid, validated\_amount = ExpenseService.validate\_amount(kwargs['amount'])

                if not is\_valid:

                    return False, validated\_amount

                expense.amount = validated\_amount

            # Update description if provided

            if 'description' in kwargs:

                is\_valid, error\_msg = ExpenseService.validate\_description(kwargs['description'])

                if not is\_valid:

                    return False, error\_msg

                expense.description = kwargs['description'].strip()

            # Update date if provided

            if 'expense\_date' in kwargs:

                expense.expense\_date = kwargs['expense\_date']

            # Save changes

            db.session.commit()

            return True, expense

        except Exception as e:

            db.session.rollback()

            return False, f"Failed to update expense: {str(e)}"

    @staticmethod

    def delete\_expense(expense\_id, user\_id):

        """

        Delete an expense

        Args:

            expense\_id (int): Expense ID

            user\_id (int): User ID for ownership verification

        Returns:

            tuple: (success: bool, message: str)

        """

        try:

            # Find expense

            expense = ExpenseService.get\_expense\_by\_id(expense\_id, user\_id)

            if not expense:

                return False, "Expense not found"

            # Delete expense

            db.session.delete(expense)

            db.session.commit()

            return True, "Expense deleted successfully"

        except Exception as e:

            db.session.rollback()

            return False, f"Failed to delete expense: {str(e)}"

    @staticmethod

    def get\_expense\_summary(user\_id, start\_date=None, end\_date=None):

        """

        Calculate expense summary statistics

        Args:

            user\_id (int): User ID

            start\_date (date, optional): Filter start date

            end\_date (date, optional): Filter end date

        Returns:

            dict: Summary statistics

        """

        try:

            # Build base query

            query = Expense.query.filter\_by(user\_id=user\_id)

            if start\_date:

                query = query.filter(Expense.expense\_date >= start\_date)

            if end\_date:

                query = query.filter(Expense.expense\_date <= end\_date)

            # Calculate total

            total\_amount = query.with\_entities(func.sum(Expense.amount)).scalar() or 0

            # Count expenses

            total\_expenses = query.count()

            # Calculate today's total

            today = date.today()

            today\_total = Expense.query.filter(

                and\_(

                    Expense.user\_id == user\_id,

                    Expense.expense\_date == today

                )

            ).with\_entities(func.sum(Expense.amount)).scalar() or 0

            # Calculate average

            average\_amount = float(total\_amount) / total\_expenses if total\_expenses > 0 else 0

            return {

                'total\_amount': float(total\_amount),

                'total\_expenses': total\_expenses,

                'today\_total': float(today\_total),

                'average\_amount': round(average\_amount, 2)

            }

        except Exception:

            return {

                'total\_amount': 0,

                'total\_expenses': 0,

                'today\_total': 0,

                'average\_amount': 0

            }

    @staticmethod

    def get\_all\_categories():

        """

        Get all active categories

        Returns:

            list: List of Category objects

        """

        try:

            return Category.query.filter\_by(is\_active=True).order\_by(Category.name).all()

        except Exception:

            return []

    @staticmethod

    def get\_all\_payment\_modes():

        """

        Get all active payment modes

        Returns:

            list: List of PaymentMode objects

        """

        try:

            return PaymentMode.query.filter\_by(is\_active=True).order\_by(PaymentMode.name).all()

        except Exception:

            return []

E:\expense\_tracker\backend\app\services\export\_service.py

"""

Export Service

Professional PDF and CSV generation with ₹ symbol support

"""

from io import BytesIO

import csv

from datetime import datetime, timedelta

from reportlab.lib import colors

from reportlab.lib.pagesizes import A4, letter

from reportlab.lib.styles import getSampleStyleSheet, ParagraphStyle

from reportlab.lib.units import inch

from reportlab.platypus import SimpleDocTemplate, Table, TableStyle, Paragraph, Spacer, PageBreak

from reportlab.pdfgen import canvas

from reportlab.pdfbase import pdfmetrics

from reportlab.pdfbase.ttfonts import TTFont

from reportlab.lib.enums import TA\_CENTER, TA\_RIGHT, TA\_LEFT

from app.models import Expense, Category, PaymentMode

from app.extensions import db

from sqlalchemy import func

import os

# ✅ ADDED: Register font that supports ₹ symbol

try:

    # Try to register DejaVu Sans font (supports ₹ symbol)

    font\_path = os.path.join(os.path.dirname(\_\_file\_\_), '..', 'fonts', 'DejaVuSans.ttf')

    if os.path.exists(font\_path):

        pdfmetrics.registerFont(TTFont('DejaVuSans', font\_path))

        pdfmetrics.registerFont(TTFont('DejaVuSans-Bold', font\_path.replace('.ttf', '-Bold.ttf')))

        DEFAULT\_FONT = 'DejaVuSans'

        BOLD\_FONT = 'DejaVuSans-Bold'

        print("✅ Custom fonts loaded successfully")

    else:

        # Fallback to standard fonts

        DEFAULT\_FONT = 'Helvetica'

        BOLD\_FONT = 'Helvetica-Bold'

        print("⚠️ Using default fonts (₹ symbol may not display)")

except Exception as e:

    DEFAULT\_FONT = 'Helvetica'

    BOLD\_FONT = 'Helvetica-Bold'

    print(f"⚠️ Font registration failed: {e}")

class ExportService:

    """Service for exporting expense data to various formats"""

    # ✅ CHANGED: Use "Rs." prefix instead of ₹ symbol as fallback

    CURRENCY\_SYMBOL = "₹"  # Will be used if font supports it

    CURRENCY\_PREFIX = "Rs. "  # Fallback if ₹ not supported

    USE\_RUPEE\_SYMBOL = DEFAULT\_FONT == 'DejaVuSans'  # Flag to check if we can use ₹

    @staticmethod

    def get\_period\_dates(period):

        """Get start and end dates for predefined periods"""

        today = datetime.now().date()

        period\_map = {

            'today': (today, today),

            'yesterday': (today - timedelta(days=1), today - timedelta(days=1)),

            'week': (today - timedelta(days=today.weekday()), today),

            'last\_week': (

                today - timedelta(days=today.weekday() + 7),

                today - timedelta(days=today.weekday() + 1)

            ),

            'month': (today.replace(day=1), today),

            'last\_month': (

                (today.replace(day=1) - timedelta(days=1)).replace(day=1),

                today.replace(day=1) - timedelta(days=1)

            ),

            'quarter': (

                today.replace(month=((today.month - 1) // 3) \* 3 + 1, day=1),

                today

            ),

            'year': (today.replace(month=1, day=1), today),

            'last\_year': (

                today.replace(year=today.year - 1, month=1, day=1),

                today.replace(year=today.year - 1, month=12, day=31)

            ),

        }

        return period\_map.get(period, (None, None))

    @staticmethod

    def format\_currency(amount):

        """

        Format amount with currency symbol

        Uses ₹ if font supports it, otherwise uses Rs.

        """

        if amount is None:

            amount = 0

        # Use ₹ symbol if custom font is loaded, otherwise use Rs.

        if ExportService.USE\_RUPEE\_SYMBOL:

            return f"₹{amount:,.2f}"

        else:

            return f"Rs. {amount:,.2f}"

    @staticmethod

    def get\_expenses\_query(user\_id, start\_date=None, end\_date=None, category\_id=None, payment\_mode\_id=None):

        """Build filtered expenses query"""

        query = Expense.query.filter\_by(user\_id=user\_id)

        if start\_date:

            query = query.filter(Expense.expense\_date >= start\_date)

        if end\_date:

            query = query.filter(Expense.expense\_date <= end\_date)

        if category\_id:

            query = query.filter\_by(category\_id=category\_id)

        if payment\_mode\_id:

            query = query.filter\_by(payment\_mode\_id=payment\_mode\_id)

        return query.order\_by(Expense.expense\_date.desc())

    @staticmethod

    def export\_to\_csv(user\_id, user\_name, start\_date=None, end\_date=None,

                      category\_id=None, payment\_mode\_id=None, include\_summary=True):

        """Export expenses to CSV with UTF-8 BOM encoding"""

        try:

            expenses = ExportService.get\_expenses\_query(

                user\_id, start\_date, end\_date, category\_id, payment\_mode\_id

            ).all()

            if not expenses and not include\_summary:

                return None

            output = BytesIO()

            output.write('\ufeff'.encode('utf-8'))

            writer = csv.writer(output)

            writer.writerow(['EXPENSE REPORT'])

            writer.writerow([f'Generated by: {user\_name}'])

            writer.writerow([f'Generated on: {datetime.now().strftime("%d %B %Y, %I:%M %p")}'])

            if start\_date and end\_date:

                writer.writerow([f'Period: {start\_date.strftime("%d %b %Y")} to {end\_date.strftime("%d %b %Y")}'])

            elif start\_date:

                writer.writerow([f'From: {start\_date.strftime("%d %b %Y")}'])

            elif end\_date:

                writer.writerow([f'Until: {end\_date.strftime("%d %b %Y")}'])

            else:

                writer.writerow(['Period: All Time'])

            writer.writerow([])

            writer.writerow([

                'Date',

                'Category',

                'Payment Mode',

                'Description',

                'Amount (₹)',

                'Tags'

            ])

            total\_amount = 0

            for expense in expenses:

                writer.writerow([

                    expense.expense\_date.strftime('%d-%m-%Y'),

                    expense.category.name if expense.category else 'Uncategorized',

                    expense.payment\_mode.name if expense.payment\_mode else 'N/A',

                    expense.description or '',

                    f'{expense.amount:.2f}',

                    expense.tags or ''

                ])

                total\_amount += expense.amount

            if include\_summary:

                writer.writerow([])

                writer.writerow(['SUMMARY'])

                writer.writerow(['Total Transactions:', len(expenses)])

                writer.writerow(['Total Amount:', f'₹{total\_amount:,.2f}'])

                writer.writerow(['Average Amount:', f'₹{(total\_amount / len(expenses)):,.2f}' if expenses else '₹0.00'])

                category\_totals = db.session.query(

                    Category.name,

                    func.sum(Expense.amount).label('total'),

                    func.count(Expense.id).label('count')

                ).join(Expense).filter(

                    Expense.user\_id == user\_id

                ).group\_by(Category.id).all()

                if category\_totals:

                    writer.writerow([])

                    writer.writerow(['CATEGORY BREAKDOWN'])

                    writer.writerow(['Category', 'Count', 'Total Amount (₹)', 'Percentage'])

                    for cat\_name, cat\_total, cat\_count in category\_totals:

                        percentage = (cat\_total / total\_amount \* 100) if total\_amount > 0 else 0

                        writer.writerow([

                            cat\_name,

                            cat\_count,

                            f'{cat\_total:.2f}',

                            f'{percentage:.1f}%'

                        ])

            output.seek(0)

            return output

        except Exception as e:

            print(f"CSV export error: {str(e)}")

            return None

    @staticmethod

    def export\_to\_pdf(user\_id, user\_name, user\_email, start\_date=None, end\_date=None,

                      category\_id=None, payment\_mode\_id=None, report\_type='detailed',

                      include\_charts=True, group\_by='none'):

        """Generate professional PDF report with ₹ symbol support"""

        try:

            expenses = ExportService.get\_expenses\_query(

                user\_id, start\_date, end\_date, category\_id, payment\_mode\_id

            ).all()

            if not expenses:

                return None

            buffer = BytesIO()

            doc = SimpleDocTemplate(

                buffer,

                pagesize=A4,

                rightMargin=0.5\*inch,

                leftMargin=0.5\*inch,

                topMargin=0.75\*inch,

                bottomMargin=0.75\*inch

            )

            elements = []

            styles = getSampleStyleSheet()

            # ✅ CHANGED: Use custom font that supports ₹

            title\_style = ParagraphStyle(

                'CustomTitle',

                parent=styles['Heading1'],

                fontSize=24,

                textColor=colors.HexColor('#1E40AF'),

                spaceAfter=30,

                alignment=TA\_CENTER,

                fontName=BOLD\_FONT  # Use registered font

            )

            heading\_style = ParagraphStyle(

                'CustomHeading',

                parent=styles['Heading2'],

                fontSize=14,

                textColor=colors.HexColor('#1E40AF'),

                spaceAfter=12,

                spaceBefore=12,

                fontName=BOLD\_FONT  # Use registered font

            )

            subheading\_style = ParagraphStyle(

                'CustomSubHeading',

                parent=styles['Normal'],

                fontSize=10,

                textColor=colors.HexColor('#6B7280'),

                spaceAfter=20,

                alignment=TA\_CENTER,

                fontName=DEFAULT\_FONT  # Use registered font

            )

            elements.append(Paragraph('EXPENSE REPORT', title\_style))

            period\_text = ''

            if start\_date and end\_date:

                period\_text = f'{start\_date.strftime("%d %B %Y")} to {end\_date.strftime("%d %B %Y")}'

            elif start\_date:

                period\_text = f'From {start\_date.strftime("%d %B %Y")}'

            elif end\_date:

                period\_text = f'Until {end\_date.strftime("%d %B %Y")}'

            else:

                period\_text = 'All Time'

            metadata\_text = f'''

            <b>Generated For:</b> {user\_name} ({user\_email})<br/>

            <b>Period:</b> {period\_text}<br/>

            <b>Generated On:</b> {datetime.now().strftime("%d %B %Y, %I:%M %p")}<br/>

            <b>Report Type:</b> {report\_type.title()}

            '''

            elements.append(Paragraph(metadata\_text, subheading\_style))

            elements.append(Spacer(1, 0.3\*inch))

            total\_amount = sum(e.amount for e in expenses)

            avg\_amount = total\_amount / len(expenses) if expenses else 0

            elements.append(Paragraph('SUMMARY', heading\_style))

            summary\_data = [

                ['Total Transactions', 'Total Amount', 'Average Amount', 'Highest Expense'],

                [

                    str(len(expenses)),

                    ExportService.format\_currency(total\_amount),

                    ExportService.format\_currency(avg\_amount),

                    ExportService.format\_currency(max(e.amount for e in expenses) if expenses else 0)

                ]

            ]

            summary\_table = Table(summary\_data, colWidths=[2\*inch, 2\*inch, 2\*inch, 2\*inch])

            summary\_table.setStyle(TableStyle([

                ('BACKGROUND', (0, 0), (-1, 0), colors.HexColor('#1E40AF')),

                ('TEXTCOLOR', (0, 0), (-1, 0), colors.whitesmoke),

                ('ALIGN', (0, 0), (-1, -1), 'CENTER'),

                ('FONTNAME', (0, 0), (-1, 0), BOLD\_FONT),  # ✅ Use registered font

                ('FONTSIZE', (0, 0), (-1, 0), 10),

                ('BOTTOMPADDING', (0, 0), (-1, 0), 12),

                ('BACKGROUND', (0, 1), (-1, -1), colors.beige),

                ('FONTSIZE', (0, 1), (-1, -1), 11),

                ('FONTNAME', (0, 1), (-1, -1), DEFAULT\_FONT),  # ✅ Use registered font

                ('GRID', (0, 0), (-1, -1), 1, colors.HexColor('#E5E7EB'))

            ]))

            elements.append(summary\_table)

            elements.append(Spacer(1, 0.3\*inch))

            if report\_type in ['detailed', 'analytics']:

                elements.append(Paragraph('TRANSACTION DETAILS', heading\_style))

                table\_data = [['Date', 'Category', 'Payment', 'Description', 'Amount']]

                for expense in expenses[:100]:

                    table\_data.append([

                        expense.expense\_date.strftime('%d-%m-%Y'),

                        expense.category.name[:15] if expense.category else 'N/A',

                        expense.payment\_mode.name[:10] if expense.payment\_mode else 'N/A',

                        (expense.description[:30] + '...') if expense.description and len(expense.description) > 30 else (expense.description or 'N/A'),

                        ExportService.format\_currency(expense.amount)

                    ])

                expense\_table = Table(table\_data, colWidths=[1.2\*inch, 1.5\*inch, 1.2\*inch, 2.5\*inch, 1.4\*inch])

                expense\_table.setStyle(TableStyle([

                    ('BACKGROUND', (0, 0), (-1, 0), colors.HexColor('#1E40AF')),

                    ('TEXTCOLOR', (0, 0), (-1, 0), colors.whitesmoke),

                    ('ALIGN', (0, 0), (-1, -1), 'LEFT'),

                    ('ALIGN', (4, 0), (4, -1), 'RIGHT'),

                    ('FONTNAME', (0, 0), (-1, 0), BOLD\_FONT),  # ✅ Use registered font

                    ('FONTSIZE', (0, 0), (-1, 0), 9),

                    ('BOTTOMPADDING', (0, 0), (-1, 0), 12),

                    ('BACKGROUND', (0, 1), (-1, -1), colors.white),

                    ('FONTNAME', (0, 1), (-1, -1), DEFAULT\_FONT),  # ✅ Use registered font

                    ('FONTSIZE', (0, 1), (-1, -1), 8),

                    ('ROWBACKGROUNDS', (0, 1), (-1, -1), [colors.white, colors.HexColor('#F9FAFB')]),

                    ('GRID', (0, 0), (-1, -1), 0.5, colors.HexColor('#E5E7EB'))

                ]))

                elements.append(expense\_table)

            if report\_type in ['summary', 'analytics']:

                elements.append(PageBreak())

                elements.append(Paragraph('CATEGORY BREAKDOWN', heading\_style))

                category\_stats = db.session.query(

                    Category.name,

                    func.sum(Expense.amount).label('total'),

                    func.count(Expense.id).label('count')

                ).join(Expense).filter(

                    Expense.user\_id == user\_id

                ).group\_by(Category.id).order\_by(func.sum(Expense.amount).desc()).all()

                cat\_data = [['Category', 'Transactions', 'Total Amount', 'Percentage']]

                for cat\_name, cat\_total, cat\_count in category\_stats:

                    percentage = (cat\_total / total\_amount \* 100) if total\_amount > 0 else 0

                    cat\_data.append([

                        cat\_name,

                        str(cat\_count),

                        ExportService.format\_currency(cat\_total),

                        f'{percentage:.1f}%'

                    ])

                cat\_table = Table(cat\_data, colWidths=[2.5\*inch, 1.5\*inch, 2\*inch, 1.5\*inch])

                cat\_table.setStyle(TableStyle([

                    ('BACKGROUND', (0, 0), (-1, 0), colors.HexColor('#10B981')),

                    ('TEXTCOLOR', (0, 0), (-1, 0), colors.whitesmoke),

                    ('ALIGN', (0, 0), (-1, -1), 'LEFT'),

                    ('ALIGN', (1, 0), (-1, -1), 'CENTER'),

                    ('FONTNAME', (0, 0), (-1, 0), BOLD\_FONT),  # ✅ Use registered font

                    ('FONTSIZE', (0, 0), (-1, 0), 10),

                    ('BOTTOMPADDING', (0, 0), (-1, 0), 12),

                    ('FONTNAME', (0, 1), (-1, -1), DEFAULT\_FONT),  # ✅ Use registered font

                    ('ROWBACKGROUNDS', (0, 1), (-1, -1), [colors.white, colors.HexColor('#F0FDF4')]),

                    ('GRID', (0, 0), (-1, -1), 0.5, colors.HexColor('#E5E7EB'))

                ]))

                elements.append(cat\_table)

            elements.append(Spacer(1, 0.5\*inch))

            footer\_style = ParagraphStyle(

                'Footer',

                parent=styles['Normal'],

                fontSize=8,

                textColor=colors.HexColor('#6B7280'),

                alignment=TA\_CENTER,

                fontName=DEFAULT\_FONT  # ✅ Use registered font

            )

            # ✅ Display currency info in footer

            currency\_text = "₹ (Indian Rupee)" if ExportService.USE\_RUPEE\_SYMBOL else "Rs. (Indian Rupee)"

            elements.append(Paragraph(

                f'This is a computer-generated report. Currency: {currency\_text}',

                footer\_style

            ))

            doc.build(elements)

            buffer.seek(0)

            return buffer

        except Exception as e:

            print(f"PDF export error: {str(e)}")

            import traceback

            traceback.print\_exc()

            return None

    @staticmethod

    def get\_export\_preview(user\_id, start\_date=None, end\_date=None, category\_id=None, payment\_mode\_id=None):

        """Get preview data for export"""

        expenses = ExportService.get\_expenses\_query(

            user\_id, start\_date, end\_date, category\_id, payment\_mode\_id

        ).limit(10).all()

        total\_count = ExportService.get\_expenses\_query(

            user\_id, start\_date, end\_date, category\_id, payment\_mode\_id

        ).count()

        total\_amount = db.session.query(func.sum(Expense.amount)).filter(

            Expense.user\_id == user\_id

        ).scalar() or 0

        return {

            "preview\_records": [e.to\_dict() for e in expenses],

            "total\_records": total\_count,

            "total\_amount": float(total\_amount),

            "currency": ExportService.CURRENCY\_SYMBOL if ExportService.USE\_RUPEE\_SYMBOL else "Rs."

        }

    @staticmethod

    def get\_export\_history(user\_id):

        """Get export history (placeholder)"""

        return []

E:\expense\_tracker\backend\app\services\openai\_service.py

"""

OpenAI Service

Business logic for AI-powered expense queries and insights

"""

import os

from openai import OpenAI

from app.extensions import db

from app.models import Expense, Category, PaymentMode

from sqlalchemy import func

from datetime import date, timedelta

class OpenAIService:

    """

    Service class for OpenAI integration

    Handles natural language queries and generates insights

    """

    def \_\_init\_\_(self):

        """Initialize OpenAI with API key and model name"""

        self.api\_key = os.environ.get('OPENAI\_API\_KEY')

        self.model\_name = os.environ.get('OPENAI\_MODEL', 'gpt-4o-mini')

        # Available models: 'gpt-4o', 'gpt-4o-mini', 'gpt-3.5-turbo'

        self.client = None

        if self.api\_key:

            try:

                self.client = OpenAI(api\_key=self.api\_key)

                print(f"✅ OpenAIService: Client initialized with model: {self.model\_name}")

            except Exception as e:

                print(f"❌ OpenAIService: Client initialization error: {str(e)}")

                print(f"💡 Tried model: {self.model\_name}")

                print("💡 Available models: 'gpt-4o', 'gpt-4o-mini', 'gpt-3.5-turbo'")

                self.client = None

        else:

            print("⚠️ OPENAI\_API\_KEY not found in environment")

            print("💡 Get your API key from: https://platform.openai.com/api-keys")

            print("💡 Add it to your .env file: OPENAI\_API\_KEY=your\_key\_here")

            self.client = None

    def is\_available(self):

        """Check if OpenAI is configured and available"""

        return self.client is not None

    @staticmethod

    def parse\_query\_intent(query):

        """

        Parse query to determine date range and focus

        """

        query\_lower = query.lower()

        today = date.today()

        intent = {

            'start\_date': None,

            'end\_date': None,

            'period': None,

            'categories': [],

            'payment\_modes': []

        }

        # Determine date range

        if 'today' in query\_lower:

            intent['start\_date'] = intent['end\_date'] = today

            intent['period'] = 'today'

        elif 'yesterday' in query\_lower:

            yesterday = today - timedelta(days=1)

            intent['start\_date'] = intent['end\_date'] = yesterday

            intent['period'] = 'yesterday'

        elif 'week' in query\_lower or 'this week' in query\_lower or '7 days' in query\_lower:

            intent['start\_date'] = today - timedelta(days=7)

            intent['end\_date'] = today

            intent['period'] = 'this week'

        elif 'month' in query\_lower or 'this month' in query\_lower or '30 days' in query\_lower:

            intent['start\_date'] = today.replace(day=1)

            intent['end\_date'] = today

            intent['period'] = 'this month'

        elif 'year' in query\_lower or 'this year' in query\_lower:

            intent['start\_date'] = today.replace(month=1, day=1)

            intent['end\_date'] = today

            intent['period'] = 'this year'

        else:

            # Default to last 30 days

            intent['start\_date'] = today - timedelta(days=30)

            intent['end\_date'] = today

            intent['period'] = 'last 30 days'

        # Check for category mentions

        category\_keywords = {

            'travel': 'Travel',

            'food': 'Food',

            'payment': 'Payments to Friends',

            'transfer': 'Self Transfer to Accounts',

            'wallet': 'Wallet Recharge',

            'recharge': 'Wallet Recharge',

            'shopping': 'Shopping',

            'entertainment': 'Entertainment',

            'bills': 'Bills',

            'groceries': 'Groceries'

        }

        for keyword, category in category\_keywords.items():

            if keyword in query\_lower:

                intent['categories'].append(category)

        # Check for payment mode mentions

        payment\_keywords = {

            'gpay': 'GPay',

            'google pay': 'GPay',

            'cash': 'Cash',

            'metro': 'Metro Card',

            'card': 'Card',

            'debit': 'Debit Card',

            'credit': 'Credit Card',

            'upi': 'UPI'

        }

        for keyword, payment\_mode in payment\_keywords.items():

            if keyword in query\_lower:

                intent['payment\_modes'].append(payment\_mode)

        return intent

    @staticmethod

    def get\_expense\_context(user\_id, intent):

        """

        Fetch expense data based on parsed intent

        """

        # Safe user\_id conversion

        if isinstance(user\_id, str):

            try:

                user\_id = int(user\_id)

            except ValueError:

                print(f"⚠️ Invalid user\_id: {user\_id}")

                return None

        context = {

            'period': intent['period'],

            'start\_date': intent['start\_date'].isoformat() if intent['start\_date'] else None,

            'end\_date': intent['end\_date'].isoformat() if intent['end\_date'] else None

        }

        # Build base query

        query = Expense.query.filter(Expense.user\_id == user\_id)

        if intent['start\_date']:

            query = query.filter(Expense.expense\_date >= intent['start\_date'])

        if intent['end\_date']:

            query = query.filter(Expense.expense\_date <= intent['end\_date'])

        # Safe total calculation

        total = query.with\_entities(func.sum(Expense.amount)).scalar()

        context['total\_expenses'] = float(total) if total else 0.0

        context['expense\_count'] = query.count()

        # Category breakdown

        category\_data = db.session.query(

            Category.name,

            func.sum(Expense.amount).label('total'),

            func.count(Expense.id).label('count')

        ).join(Expense).filter(Expense.user\_id == user\_id)

        if intent['start\_date']:

            category\_data = category\_data.filter(Expense.expense\_date >= intent['start\_date'])

        if intent['end\_date']:

            category\_data = category\_data.filter(Expense.expense\_date <= intent['end\_date'])

        category\_results = category\_data.group\_by(Category.name).order\_by(func.sum(Expense.amount).desc()).all()

        # Safe number conversion

        context['by\_category'] = {

            row.name: {

                'total': float(row.total) if row.total else 0.0,

                'count': row.count

            }

            for row in category\_results if row.total

        }

        # Payment mode breakdown

        payment\_data = db.session.query(

            PaymentMode.name,

            PaymentMode.bank\_name,

            func.sum(Expense.amount).label('total'),

            func.count(Expense.id).label('count')

        ).join(Expense).filter(Expense.user\_id == user\_id)

        if intent['start\_date']:

            payment\_data = payment\_data.filter(Expense.expense\_date >= intent['start\_date'])

        if intent['end\_date']:

            payment\_data = payment\_data.filter(Expense.expense\_date <= intent['end\_date'])

        payment\_results = payment\_data.group\_by(PaymentMode.name, PaymentMode.bank\_name).order\_by(func.sum(Expense.amount).desc()).all()

        # Safe number conversion

        context['by\_payment\_mode'] = {

            f"{row.name} - {row.bank\_name}" if row.bank\_name else row.name: {

                'total': float(row.total) if row.total else 0.0,

                'count': row.count

            }

            for row in payment\_results if row.total

        }

        # Recent expenses (top 5)

        recent = query.order\_by(Expense.expense\_date.desc()).limit(5).all()

        context['recent\_expenses'] = [

            {

                'date': exp.expense\_date.isoformat(),

                'description': exp.description or 'No description',

                'amount': float(exp.amount) if exp.amount else 0.0,

                'category': exp.category.name if exp.category else 'Uncategorized'

            }

            for exp in recent

        ]

        return context

    def query\_expenses(self, user\_id, query\_text):

        """

        Process natural language query about expenses using OpenAI

        """

        try:

            if not self.is\_available():

                return False, "OpenAI is not configured. Please set OPENAI\_API\_KEY in your environment."

            # Safe user\_id conversion

            if isinstance(user\_id, str):

                try:

                    user\_id = int(user\_id)

                except ValueError:

                    return False, "Invalid user ID"

            # Parse query and get context

            intent = self.parse\_query\_intent(query\_text)

            context = self.get\_expense\_context(user\_id, intent)

            if not context:

                return False, "Failed to retrieve expense data"

            # Handle no expenses case

            if context['expense\_count'] == 0:

                return True, f"You haven't recorded any expenses for {context['period']}. Start tracking your expenses to get AI-powered insights! 📊"

            # Build prompt

            prompt = self.\_build\_query\_prompt(query\_text, context)

            # ✅ CHANGED: Use OpenAI chat completions API

            try:

                response = self.client.chat.completions.create(

                    model=self.model\_name,

                    messages=[

                        {

                            "role": "system",

                            "content": "You are a helpful financial assistant. Provide clear, concise answers about expense data."

                        },

                        {

                            "role": "user",

                            "content": prompt

                        }

                    ],

                    temperature=0.7,

                    max\_tokens=500,

                )

                answer = response.choices[0].message.content

                return True, answer

            except Exception as e:

                error\_msg = str(e)

                print(f"❌ OpenAIService: API error: {error\_msg}")

                # Better error messages

                if 'model' in error\_msg.lower() and 'not found' in error\_msg.lower():

                    return False, f"Model '{self.model\_name}' not found. Try using 'gpt-4o-mini' or 'gpt-3.5-turbo'."

                elif 'api\_key' in error\_msg.lower() or 'authentication' in error\_msg.lower():

                    return False, "Invalid API key. Please check your OPENAI\_API\_KEY."

                elif 'quota' in error\_msg.lower() or 'insufficient' in error\_msg.lower():

                    return False, "OpenAI API quota exceeded. Please check your billing."

                else:

                    return False, f"AI query failed: {error\_msg}"

        except Exception as e:

            print(f"❌ AI query error: {str(e)}")

            import traceback

            traceback.print\_exc()

            return False, f"AI query failed: {str(e)}"

    def \_build\_query\_prompt(self, query, context):

        """

        Build prompt for OpenAI API

        """

        prompt = f"""You are a helpful financial assistant analyzing expense data.

User Question: "{query}"

Expense Summary ({context['period']}):

• Period: {context['start\_date']} to {context['end\_date']}

• Total Spending: ₹{context['total\_expenses']:.2f}

• Number of Transactions: {context['expense\_count']}

"""

        # Add category breakdown if available

        if context.get('by\_category'):

            prompt += "\n📊 Spending by Category:\n"

            for category, data in sorted(context['by\_category'].items(), key=lambda x: x[1]['total'], reverse=True):

                percentage = (data['total'] / context['total\_expenses'] \* 100) if context['total\_expenses'] > 0 else 0

                prompt += f"   • {category}: ₹{data['total']:.2f} ({data['count']} transactions, {percentage:.1f}%)\n"

        # Add payment mode breakdown if available

        if context.get('by\_payment\_mode'):

            prompt += "\n💳 Payment Methods Used:\n"

            for payment, data in sorted(context['by\_payment\_mode'].items(), key=lambda x: x[1]['total'], reverse=True):

                prompt += f"   • {payment}: ₹{data['total']:.2f} ({data['count']} times)\n"

        # Add recent expenses

        if context.get('recent\_expenses'):

            prompt += "\n🕐 Recent Expenses:\n"

            for exp in context['recent\_expenses'][:3]:

                prompt += f"   • {exp['date']}: {exp['description']} - ₹{exp['amount']:.2f} ({exp['category']})\n"

        prompt += """

Instructions:

• Answer the user's question directly and clearly

• Use specific numbers from the data

• Format currency as ₹X,XXX.XX

• Use emojis where appropriate (💰 📊 💳 📈 etc.)

• Keep response under 150 words

• Use bullet points for multiple items

• Be friendly and conversational

• Provide insights when relevant

"""

        return prompt

    def generate\_suggestions(self, user\_id, days=30):

        """

        Generate AI-powered spending suggestions using OpenAI

        """

        try:

            if not self.is\_available():

                return False, "OpenAI is not configured."

            # Safe user\_id conversion

            if isinstance(user\_id, str):

                try:

                    user\_id = int(user\_id)

                except ValueError:

                    return False, "Invalid user ID"

            # Get expense context

            end\_date = date.today()

            start\_date = end\_date - timedelta(days=days)

            intent = {

                'start\_date': start\_date,

                'end\_date': end\_date,

                'period': f'last {days} days',

                'categories': [],

                'payment\_modes': []

            }

            context = self.get\_expense\_context(user\_id, intent)

            if not context:

                return False, "Failed to retrieve expense data"

            # Handle no expenses case

            if context['expense\_count'] == 0:

                return True, (

                    "🎯 Start tracking your expenses to get personalized AI insights!\n\n"

                    "📝 Add your first expense to begin building your financial profile\n\n"

                    "💡 Categorize expenses properly for detailed analysis\n\n"

                    "🔔 Check back after a week of tracking for AI-powered suggestions"

                )

            # Build prompt

            prompt = f"""Analyze this user's spending pattern and provide actionable insights.

Period: Last {days} days

Total Spent: ₹{context['total\_expenses']:.2f}

Transactions: {context['expense\_count']}

Daily Average: ₹{(context['total\_expenses'] / days):.2f}

Spending by Category:

"""

            # Add category breakdown

            for category, data in sorted(context.get('by\_category', {}).items(), key=lambda x: x[1]['total'], reverse=True):

                percentage = (data['total'] / context['total\_expenses'] \* 100) if context['total\_expenses'] > 0 else 0

                prompt += f"• {category}: ₹{data['total']:.2f} ({percentage:.1f}%, {data['count']} transactions)\n"

            prompt += """

Provide 4-5 personalized insights:

1. ✅ One positive observation about their spending habits

2. ⚠️ One area where they could potentially save money

3. 💡 One practical tip for better expense tracking

4. 📊 Overall financial health assessment (Excellent/Good/Fair/Needs Attention)

5. 🎯 One specific action item they can take this week

Format as clear bullet points with emojis. Be encouraging, specific, and actionable. Keep it under 200 words.

"""

            try:

                # ✅ CHANGED: Use OpenAI chat completions API

                response = self.client.chat.completions.create(

                    model=self.model\_name,

                    messages=[

                        {

                            "role": "system",

                            "content": "You are a helpful financial advisor providing spending insights."

                        },

                        {

                            "role": "user",

                            "content": prompt

                        }

                    ],

                    temperature=0.8,

                    max\_tokens=600,

                )

                suggestions = response.choices[0].message.content

                return True, suggestions

            except Exception as e:

                print(f"❌ OpenAIService: generate suggestions error: {str(e)}")

                return False, f"Failed to generate suggestions: {str(e)}"

        except Exception as e:

            print(f"❌ Suggestions error: {str(e)}")

            import traceback

            traceback.print\_exc()

            return False, f"Failed to generate suggestions: {str(e)}"

E:\expense\_tracker\backend\app\utils

E:\expense\_tracker\backend\app\utils\\_\_init\_\_.py

"""

Utils Package Initialization

Exports utility functions, decorators, and validators

"""

from app.utils.decorators import jwt\_required\_with\_user, admin\_required, rate\_limit

from app.utils.validators import (

    validate\_email,

    validate\_password,

    validate\_amount,

    validate\_date,

    validate\_required\_fields

)

from app.utils.helpers import (

    format\_currency,

    format\_date,

    parse\_date,

    calculate\_percentage,

    get\_date\_range,

    paginate\_query,

    success\_response,

    error\_response

)

# Export all utilities

\_\_all\_\_ = [

    # Decorators

    'jwt\_required\_with\_user',

    'admin\_required',

    'rate\_limit',

    # Validators

    'validate\_email',

    'validate\_password',

    'validate\_amount',

    'validate\_date',

    'validate\_required\_fields',

    # Helpers

    'format\_currency',

    'format\_date',

    'parse\_date',

    'calculate\_percentage',

    'get\_date\_range',

    'paginate\_query',

    'success\_response',

    'error\_response'

]

E:\expense\_tracker\backend\app\utils\decorators.py

"""

Custom Decorators

Reusable decorators for route protection, validation, and enhancement

"""

from functools import wraps

from flask import jsonify, request

from flask\_jwt\_extended import verify\_jwt\_in\_request, get\_jwt\_identity, get\_jwt

from app.models import User

from datetime import datetime, timedelta

from collections import defaultdict

import time

# Rate limiting storage (in production, use Redis)

rate\_limit\_storage = defaultdict(lambda: {'count': 0, 'reset\_time': None})

def jwt\_required\_with\_user(fn):

    """

    Custom decorator that verifies JWT and loads user object

    Usage:

        @jwt\_required\_with\_user

        def my\_route(current\_user):

            # current\_user is automatically passed as User object

            return jsonify({"user": current\_user.email})

    Args:

        fn: Function to decorate

    Returns:

        Decorated function with current\_user parameter

    """

    @wraps(fn)

    def wrapper(\*args, \*\*kwargs):

        try:

            # Verify JWT token is present and valid

            verify\_jwt\_in\_request()

            # Get current user ID from JWT

            current\_user\_id = get\_jwt\_identity()

            # Fetch user from database

            current\_user = User.query.get(current\_user\_id)

            if not current\_user:

                return jsonify({"error": "User not found"}), 404

            # Check if user account is active

            if not current\_user.is\_active:

                return jsonify({"error": "Account is deactivated"}), 403

            # Call the original function with current\_user as parameter

            return fn(current\_user=current\_user, \*args, \*\*kwargs)

        except Exception as e:

            return jsonify({"error": f"Authentication failed: {str(e)}"}), 401

    return wrapper

def admin\_required(fn):

    """

    Decorator that requires JWT authentication and admin role

    Usage:

        @admin\_required

        def admin\_only\_route():

            return jsonify({"message": "Admin access granted"})

    Note: Requires 'is\_admin' claim in JWT token

    Args:

        fn: Function to decorate

    Returns:

        Decorated function with admin verification

    """

    @wraps(fn)

    def wrapper(\*args, \*\*kwargs):

        try:

            # Verify JWT token

            verify\_jwt\_in\_request()

            # Get JWT claims

            claims = get\_jwt()

            # Check if user has admin privileges

            if not claims.get('is\_admin', False):

                return jsonify({"error": "Admin access required"}), 403

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

        except Exception as e:

            return jsonify({"error": f"Authorization failed: {str(e)}"}), 401

    return wrapper

def rate\_limit(max\_requests=10, window\_seconds=60):

    """

    Rate limiting decorator to prevent API abuse

    Usage:

        @rate\_limit(max\_requests=5, window\_seconds=60)

        def limited\_route():

            return jsonify({"message": "Success"})

    Args:

        max\_requests (int): Maximum number of requests allowed

        window\_seconds (int): Time window in seconds

    Returns:

        Decorated function with rate limiting

    """

    def decorator(fn):

        @wraps(fn)

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

            try:

                # Get client identifier (IP address or user ID)

                client\_id = request.remote\_addr

                # Try to get user ID from JWT if authenticated

                try:

                    verify\_jwt\_in\_request(optional=True)

                    user\_id = get\_jwt\_identity()

                    if user\_id:

                        client\_id = f"user\_{user\_id}"

                except:

                    pass

                # Get rate limit data for this client

                rate\_data = rate\_limit\_storage[client\_id]

                current\_time = datetime.now()

                # Reset counter if window has expired

                if rate\_data['reset\_time'] is None or current\_time > rate\_data['reset\_time']:

                    rate\_data['count'] = 0

                    rate\_data['reset\_time'] = current\_time + timedelta(seconds=window\_seconds)

                # Check if limit exceeded

                if rate\_data['count'] >= max\_requests:

                    retry\_after = int((rate\_data['reset\_time'] - current\_time).total\_seconds())

                    return jsonify({

                        "error": "Rate limit exceeded",

                        "retry\_after": retry\_after

                    }), 429

                # Increment request counter

                rate\_data['count'] += 1

                # Call the original function

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

            except Exception as e:

                return jsonify({"error": f"Rate limiting error: {str(e)}"}), 500

        return wrapper

    return decorator

def validate\_json(fn):

    """

    Decorator to ensure request contains valid JSON data

    Usage:

        @validate\_json

        def create\_item():

            data = request.get\_json()

            # data is guaranteed to be valid JSON

    Args:

        fn: Function to decorate

    Returns:

        Decorated function with JSON validation

    """

    @wraps(fn)

    def wrapper(\*args, \*\*kwargs):

        if not request.is\_json:

            return jsonify({"error": "Content-Type must be application/json"}), 400

        try:

            data = request.get\_json()

            if data is None:

                return jsonify({"error": "Invalid JSON data"}), 400

        except Exception:

            return jsonify({"error": "Malformed JSON"}), 400

        return fn(\*args, \*\*kwargs)

    return wrapper

def require\_ownership(model\_class, id\_param='id'):

    """

    Decorator to verify user owns the resource

    Usage:

        @require\_ownership(Expense, id\_param='expense\_id')

        def update\_expense(expense\_id, current\_user):

            # Automatically verifies expense belongs to current\_user

    Args:

        model\_class: SQLAlchemy model class

        id\_param (str): Name of the ID parameter in route

    Returns:

        Decorated function with ownership verification

    """

    def decorator(fn):

        @wraps(fn)

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

            try:

                # Verify JWT and get user

                verify\_jwt\_in\_request()

                current\_user\_id = get\_jwt\_identity()

                # Get resource ID from kwargs

                resource\_id = kwargs.get(id\_param)

                if not resource\_id:

                    return jsonify({"error": f"Missing {id\_param} parameter"}), 400

                # Fetch resource

                resource = model\_class.query.get(resource\_id)

                if not resource:

                    return jsonify({"error": "Resource not found"}), 404

                # Verify ownership

                if hasattr(resource, 'user\_id') and resource.user\_id != current\_user\_id:

                    return jsonify({"error": "Access denied"}), 403

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

            except Exception as e:

                return jsonify({"error": f"Ownership verification failed: {str(e)}"}), 500

        return wrapper

    return decorator

def cache\_response(timeout=300):

    """

    Simple response caching decorator (in-memory)

    Usage:

        @cache\_response(timeout=60)

        def expensive\_calculation():

            # Result will be cached for 60 seconds

    Args:

        timeout (int): Cache timeout in seconds

    Returns:

        Decorated function with caching

    """

    cache = {}

    def decorator(fn):

        @wraps(fn)

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

            # Generate cache key from function name and arguments

            cache\_key = f"{fn.\_\_name\_\_}\_{str(args)}\_{str(kwargs)}"

            # Check if result is in cache and not expired

            if cache\_key in cache:

                result, timestamp = cache[cache\_key]

                if time.time() - timestamp < timeout:

                    return result

            # Call function and cache result

            result = fn(\*args, \*\*kwargs)

            cache[cache\_key] = (result, time.time())

            return result

        return wrapper

    return decorator

def log\_request(fn):

    """

    Decorator to log incoming requests for debugging

    Usage:

        @log\_request

        def my\_route():

            return jsonify({"message": "Success"})

    Args:

        fn: Function to decorate

    Returns:

        Decorated function with logging

    """

    @wraps(fn)

    def wrapper(\*args, \*\*kwargs):

        print(f"[{datetime.now().isoformat()}] {request.method} {request.path}")

        print(f"Headers: {dict(request.headers)}")

        if request.is\_json:

            print(f"JSON Body: {request.get\_json()}")

        result = fn(\*args, \*\*kwargs)

        print(f"Response Status: {result[1] if isinstance(result, tuple) else 200}")

        return result

    return wrapper

E:\expense\_tracker\backend\app\utils\helpers.py

"""

Helper Functions

Utility functions for common operations like formatting, calculations, and responses

"""

from datetime import datetime, date, timedelta

from typing import Any, Dict, Optional, Tuple, List

from flask import jsonify

def format\_currency(amount: float, currency\_symbol: str = '₹') -> str:

    """

    Format amount as currency string

    Args:

        amount (float): Amount to format

        currency\_symbol (str): Currency symbol

    Returns:

        str: Formatted currency string (e.g., "₹1,234.56")

    """

    try:

        amount = float(amount)

        formatted = f"{currency\_symbol}{amount:,.2f}"

        return formatted

    except (ValueError, TypeError):

        return f"{currency\_symbol}0.00"

def format\_date(date\_obj: date, format\_str: str = '%Y-%m-%d') -> str:

    """

    Format date object as string

    Args:

        date\_obj (date): Date object to format

        format\_str (str): Desired date format

    Returns:

        str: Formatted date string

    """

    if not isinstance(date\_obj, (date, datetime)):

        return ""

    try:

        return date\_obj.strftime(format\_str)

    except Exception:

        return str(date\_obj)

def parse\_date(date\_str: str, format\_str: str = '%Y-%m-%d') -> Optional[date]:

    """

    Parse date string to date object

    Args:

        date\_str (str): Date string to parse

        format\_str (str): Expected date format

    Returns:

        date|None: Parsed date object or None if invalid

    """

    if not date\_str:

        return None

    try:

        return datetime.strptime(date\_str, format\_str).date()

    except ValueError:

        return None

def calculate\_percentage(part: float, total: float) -> float:

    """

    Calculate percentage with zero-division handling

    Args:

        part (float): Part value

        total (float): Total value

    Returns:

        float: Percentage (rounded to 2 decimals)

    """

    try:

        if total == 0:

            return 0.0

        percentage = (float(part) / float(total)) \* 100

        return round(percentage, 2)

    except (ValueError, TypeError, ZeroDivisionError):

        return 0.0

def get\_date\_range(period: str) -> Tuple[date, date]:

    """

    Get start and end dates for common periods

    Args:

        period (str): Period name ('today', 'yesterday', 'week', 'month', 'year')

    Returns:

        tuple: (start\_date, end\_date)

    """

    today = date.today()

    if period == 'today':

        return today, today

    elif period == 'yesterday':

        yesterday = today - timedelta(days=1)

        return yesterday, yesterday

    elif period == 'week':

        start = today - timedelta(days=7)

        return start, today

    elif period == 'month':

        start = today.replace(day=1)

        return start, today

    elif period == 'year':

        start = today.replace(month=1, day=1)

        return start, today

    elif period == 'last\_30\_days':

        start = today - timedelta(days=30)

        return start, today

    else:

        # Default: last 7 days

        start = today - timedelta(days=7)

        return start, today

def paginate\_query(query, page: int = 1, per\_page: int = 50):

    """

    Paginate SQLAlchemy query

    Args:

        query: SQLAlchemy query object

        page (int): Page number (1-indexed)

        per\_page (int): Items per page

    Returns:

        dict: Pagination data with items and metadata

    """

    try:

        page = max(1, int(page))

        per\_page = max(1, min(100, int(per\_page)))  # Cap at 100

    except (ValueError, TypeError):

        page = 1

        per\_page = 50

    # Get total count

    total\_items = query.count()

    # Calculate pagination

    total\_pages = (total\_items + per\_page - 1) // per\_page

    offset = (page - 1) \* per\_page

    # Get items for current page

    items = query.limit(per\_page).offset(offset).all()

    return {

        'items': items,

        'pagination': {

            'page': page,

            'per\_page': per\_page,

            'total\_items': total\_items,

            'total\_pages': total\_pages,

            'has\_next': page < total\_pages,

            'has\_prev': page > 1

        }

    }

def success\_response(data: Any = None, message: str = "Success", status\_code: int = 200):

    """

    Create standardized success JSON response

    Args:

        data: Response data

        message (str): Success message

        status\_code (int): HTTP status code

    Returns:

        tuple: (JSON response, status code)

    """

    response = {

        "success": True,

        "message": message

    }

    if data is not None:

        response["data"] = data

    return jsonify(response), status\_code

def error\_response(error: str, status\_code: int = 400, details: Any = None):

    """

    Create standardized error JSON response

    Args:

        error (str): Error message

        status\_code (int): HTTP status code

        details: Additional error details

    Returns:

        tuple: (JSON response, status code)

    """

    response = {

        "success": False,

        "error": error

    }

    if details is not None:

        response["details"] = details

    return jsonify(response), status\_code

def get\_current\_timestamp() -> str:

    """

    Get current timestamp in ISO format

    Returns:

        str: ISO formatted timestamp

    """

    return datetime.now().isoformat()

def truncate\_string(text: str, max\_length: int = 50, suffix: str = "...") -> str:

    """

    Truncate string to maximum length

    Args:

        text (str): Text to truncate

        max\_length (int): Maximum length

        suffix (str): Suffix to add if truncated

    Returns:

        str: Truncated string

    """

    if not text or not isinstance(text, str):

        return ""

    if len(text) <= max\_length:

        return text

    return text[:max\_length - len(suffix)] + suffix

def generate\_slug(text: str) -> str:

    """

    Generate URL-safe slug from text

    Args:

        text (str): Text to convert to slug

    Returns:

        str: URL-safe slug

    """

    import re

    if not text:

        return ""

    # Convert to lowercase

    slug = text.lower()

    # Replace spaces with hyphens

    slug = slug.replace(' ', '-')

    # Remove non-alphanumeric characters (except hyphens)

    slug = re.sub(r'[^a-z0-9-]', '', slug)

    # Remove multiple consecutive hyphens

    slug = re.sub(r'-+', '-', slug)

    # Remove leading/trailing hyphens

    slug = slug.strip('-')

    return slug

def chunk\_list(items: List, chunk\_size: int = 100) -> List[List]:

    """

    Split list into chunks of specified size

    Args:

        items (list): List to chunk

        chunk\_size (int): Size of each chunk

    Returns:

        list: List of chunks

    """

    chunks = []

    for i in range(0, len(items), chunk\_size):

        chunks.append(items[i:i + chunk\_size])

    return chunks

def safe\_divide(numerator: float, denominator: float, default: float = 0.0) -> float:

    """

    Safely divide two numbers with zero-division handling

    Args:

        numerator (float): Numerator

        denominator (float): Denominator

        default (float): Default value if division fails

    Returns:

        float: Division result or default

    """

    try:

        if denominator == 0:

            return default

        return float(numerator) / float(denominator)

    except (ValueError, TypeError, ZeroDivisionError):

        return default

def merge\_dicts(\*dicts: Dict) -> Dict:

    """

    Merge multiple dictionaries

    Args:

        \*dicts: Variable number of dictionaries to merge

    Returns:

        dict: Merged dictionary

    """

    result = {}

    for d in dicts:

        if isinstance(d, dict):

            result.update(d)

    return result

def get\_month\_name(month\_number: int) -> str:

    """

    Get month name from month number

    Args:

        month\_number (int): Month number (1-12)

    Returns:

        str: Month name

    """

    month\_names = [

        "January", "February", "March", "April", "May", "June",

        "July", "August", "September", "October", "November", "December"

    ]

    try:

        index = int(month\_number) - 1

        if 0 <= index < 12:

            return month\_names[index]

    except (ValueError, TypeError):

        pass

    return "Unknown"

def is\_valid\_json(data: Any) -> bool:

    """

    Check if data is valid JSON-serializable

    Args:

        data: Data to check

    Returns:

        bool: True if JSON-serializable

    """

    import json

    try:

        json.dumps(data)

        return True

    except (TypeError, ValueError):

        return False

E:\expense\_tracker\backend\app\utils\validators.py

"""

Input Validators

Reusable validation functions for data sanitization and verification

"""

import re

from datetime import datetime, date

from typing import Tuple, Any, Optional, List

def validate\_email(email: str) -> Tuple[bool, str]:

    """

    Validate email address format

    Args:

        email (str): Email address to validate

    Returns:

        tuple: (is\_valid: bool, error\_message: str)

    """

    if not email or not isinstance(email, str):

        return False, "Email is required"

    email = email.strip()

    if len(email) > 254:

        return False, "Email is too long"

    # RFC 5322 compliant email regex

    pattern = r'^[a-zA-Z0-9.!#$%&\'\*+/=?^\_`{|}~-]+@[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?(?:\.[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?)\*$'

    if not re.match(pattern, email):

        return False, "Invalid email format"

    return True, ""

def validate\_password(password: str, min\_length: int = 6, max\_length: int = 128) -> Tuple[bool, str]:

    """

    Validate password strength

    Args:

        password (str): Password to validate

        min\_length (int): Minimum password length

        max\_length (int): Maximum password length

    Returns:

        tuple: (is\_valid: bool, error\_message: str)

    """

    if not password or not isinstance(password, str):

        return False, "Password is required"

    if len(password) < min\_length:

        return False, f"Password must be at least {min\_length} characters long"

    if len(password) > max\_length:

        return False, f"Password is too long (maximum {max\_length} characters)"

    # Optional: Add complexity requirements

    # has\_uppercase = any(c.isupper() for c in password)

    # has\_lowercase = any(c.islower() for c in password)

    # has\_digit = any(c.isdigit() for c in password)

    # has\_special = any(c in "!@#$%^&\*()\_+-=[]{}|;:,.<>?" for c in password)

    return True, ""

def validate\_amount(amount: Any, min\_value: float = 0.01, max\_value: float = 99999999.99) -> Tuple[bool, Any]:

    """

    Validate and sanitize monetary amount

    Args:

        amount: Amount value to validate

        min\_value (float): Minimum allowed amount

        max\_value (float): Maximum allowed amount

    Returns:

        tuple: (is\_valid: bool, validated\_amount\_or\_error: float|str)

    """

    if amount is None:

        return False, "Amount is required"

    try:

        amount = float(amount)

    except (ValueError, TypeError):

        return False, "Invalid amount format. Must be a number"

    if amount < min\_value:

        return False, f"Amount must be at least {min\_value}"

    if amount > max\_value:

        return False, f"Amount is too large (maximum {max\_value})"

    # Round to 2 decimal places

    amount = round(amount, 2)

    return True, amount

def validate\_date(date\_str: str, date\_format: str = '%Y-%m-%d') -> Tuple[bool, Any]:

    """

    Validate and parse date string

    Args:

        date\_str (str): Date string to validate

        date\_format (str): Expected date format

    Returns:

        tuple: (is\_valid: bool, parsed\_date\_or\_error: date|str)

    """

    if not date\_str or not isinstance(date\_str, str):

        return False, "Date is required"

    try:

        parsed\_date = datetime.strptime(date\_str, date\_format).date()

    except ValueError:

        return False, f"Invalid date format. Expected format: {date\_format}"

    # Check if date is not in future

    if parsed\_date > date.today():

        return False, "Date cannot be in the future"

    # Optional: Check if date is not too old (e.g., 10 years)

    from datetime import timedelta

    ten\_years\_ago = date.today() - timedelta(days=365\*10)

    if parsed\_date < ten\_years\_ago:

        return False, "Date is too old (maximum 10 years ago)"

    return True, parsed\_date

def validate\_string(value: Any, field\_name: str, min\_length: int = 1, max\_length: int = 255,

                    required: bool = True) -> Tuple[bool, str]:

    """

    Validate string field with length constraints

    Args:

        value: Value to validate

        field\_name (str): Name of field for error messages

        min\_length (int): Minimum string length

        max\_length (int): Maximum string length

        required (bool): Whether field is required

    Returns:

        tuple: (is\_valid: bool, error\_message: str)

    """

    if value is None or (isinstance(value, str) and not value.strip()):

        if required:

            return False, f"{field\_name} is required"

        return True, ""

    if not isinstance(value, str):

        return False, f"{field\_name} must be a string"

    value = value.strip()

    if len(value) < min\_length:

        return False, f"{field\_name} must be at least {min\_length} characters long"

    if len(value) > max\_length:

        return False, f"{field\_name} is too long (maximum {max\_length} characters)"

    return True, ""

def validate\_integer(value: Any, field\_name: str, min\_value: Optional[int] = None,

                     max\_value: Optional[int] = None) -> Tuple[bool, Any]:

    """

    Validate integer field

    Args:

        value: Value to validate

        field\_name (str): Name of field for error messages

        min\_value (int, optional): Minimum allowed value

        max\_value (int, optional): Maximum allowed value

    Returns:

        tuple: (is\_valid: bool, validated\_value\_or\_error: int|str)

    """

    if value is None:

        return False, f"{field\_name} is required"

    try:

        value = int(value)

    except (ValueError, TypeError):

        return False, f"{field\_name} must be an integer"

    if min\_value is not None and value < min\_value:

        return False, f"{field\_name} must be at least {min\_value}"

    if max\_value is not None and value > max\_value:

        return False, f"{field\_name} must be at most {max\_value}"

    return True, value

def validate\_required\_fields(data: dict, required\_fields: List[str]) -> Tuple[bool, str]:

    """

    Check if all required fields are present in data dictionary

    Args:

        data (dict): Data dictionary to validate

        required\_fields (list): List of required field names

    Returns:

        tuple: (is\_valid: bool, error\_message: str)

    """

    if not isinstance(data, dict):

        return False, "Invalid data format"

    missing\_fields = []

    for field in required\_fields:

        if field not in data or data[field] is None or (isinstance(data[field], str) and not data[field].strip()):

            missing\_fields.append(field)

    if missing\_fields:

        return False, f"Missing required fields: {', '.join(missing\_fields)}"

    return True, ""

def validate\_choice(value: Any, field\_name: str, choices: List[Any]) -> Tuple[bool, str]:

    """

    Validate that value is one of allowed choices

    Args:

        value: Value to validate

        field\_name (str): Name of field for error messages

        choices (list): List of allowed values

    Returns:

        tuple: (is\_valid: bool, error\_message: str)

    """

    if value is None:

        return False, f"{field\_name} is required"

    if value not in choices:

        return False, f"{field\_name} must be one of: {', '.join(map(str, choices))}"

    return True, ""

def validate\_id(value: Any, field\_name: str = "ID") -> Tuple[bool, Any]:

    """

    Validate database ID (positive integer)

    Args:

        value: Value to validate

        field\_name (str): Name of field for error messages

    Returns:

        tuple: (is\_valid: bool, validated\_id\_or\_error: int|str)

    """

    if value is None:

        return False, f"{field\_name} is required"

    try:

        value = int(value)

    except (ValueError, TypeError):

        return False, f"Invalid {field\_name} format"

    if value <= 0:

        return False, f"{field\_name} must be a positive integer"

    return True, value

def sanitize\_string(value: str) -> str:

    """

    Sanitize string input by removing dangerous characters

    Args:

        value (str): String to sanitize

    Returns:

        str: Sanitized string

    """

    if not isinstance(value, str):

        return ""

    # Strip whitespace

    value = value.strip()

    # Remove null bytes

    value = value.replace('\x00', '')

    # Remove control characters except newlines and tabs

    value = ''.join(char for char in value if char.isprintable() or char in '\n\t')

    return value

def validate\_phone\_number(phone: str, country\_code: str = 'IN') -> Tuple[bool, str]:

    """

    Validate phone number format

    Args:

        phone (str): Phone number to validate

        country\_code (str): Country code for validation rules

    Returns:

        tuple: (is\_valid: bool, error\_message: str)

    """

    if not phone or not isinstance(phone, str):

        return False, "Phone number is required"

    # Remove spaces and dashes

    phone = phone.replace(' ', '').replace('-', '').replace('(', '').replace(')', '')

    if country\_code == 'IN':

        # Indian phone number: 10 digits starting with 6-9

        pattern = r'^[6-9]\d{9}$'

        if not re.match(pattern, phone):

            return False, "Invalid Indian phone number format (must be 10 digits starting with 6-9)"

    else:

        # Generic validation: 7-15 digits

        if not phone.isdigit() or len(phone) < 7 or len(phone) > 15:

            return False, "Invalid phone number format"

    return True, ""

def validate\_url(url: str) -> Tuple[bool, str]:

    """

    Validate URL format

    Args:

        url (str): URL to validate

    Returns:

        tuple: (is\_valid: bool, error\_message: str)

    """

    if not url or not isinstance(url, str):

        return False, "URL is required"

    # Basic URL regex pattern

    pattern = r'^https?:\/\/(www\.)?[-a-zA-Z0-9@:%.\_\+~#=]{1,256}\.[a-zA-Z0-9()]{1,6}\b([-a-zA-Z0-9()@:%\_\+.~#?&//=]\*)$'

    if not re.match(pattern, url):

        return False, "Invalid URL format"

    return True, ""

E:\expense\_tracker\backend\app\\_\_init\_\_.py

"""

Flask Application Factory

Creates and configures the Flask application with all extensions and blueprints

"""

from flask import Flask, jsonify

from flask\_cors import CORS

from config import Config

from app.extensions import db, migrate, jwt

from app.models import User, Category, PaymentMode, Expense

def create\_app(config\_class=Config):

    """

    Application factory function

    Creates and configures Flask application instance

    Args:

        config\_class: Configuration class (default: Config)

    Returns:

        Flask: Configured Flask application

    """

    # Create Flask app

    app = Flask(\_\_name\_\_)

    # Load configuration

    app.config.from\_object(config\_class)

    print('=' \* 60)

    print('🚀 Initializing Flask Application')

    print('=' \* 60)

    # ✅ FIXED: Initialize JWT FIRST with proper config

    jwt.init\_app(app)

    # ✅ FIXED: Configure CORS with proper OPTIONS support

    CORS(app,

         resources={

             r"/api/\*": {

                 "origins": ["http://localhost:3000", "http://127.0.0.1:3000"],

                 "methods": ["GET", "POST", "PUT", "DELETE", "OPTIONS", "PATCH"],

                 "allow\_headers": ["Content-Type", "Authorization", "X-Requested-With", "X-CSRF-TOKEN"],

                 "expose\_headers": ["Content-Type", "Authorization"],

                 "supports\_credentials": True,

                 "max\_age": 3600,

                 "send\_wildcard": False,

                 "automatic\_options": True

             }

         })

    print('✓ CORS configured with proper OPTIONS support')

    # Initialize extensions AFTER JWT

    db.init\_app(app)

    migrate.init\_app(app, db)

    # Create tables if they don't exist (development only)

    with app.app\_context():

        if app.config.get('FLASK\_ENV') == 'development':

            try:

                db.create\_all()

                print('✓ Database tables created/verified')

            except Exception as e:

                print(f'⚠️  Database warning: {str(e)}')

    # ✅ FIXED: Register JWT callbacks BEFORE blueprints

    register\_jwt\_callbacks(jwt, app)

    print('✓ JWT callbacks registered')

    # Register blueprints

    from app.routes import auth\_bp, expenses\_bp, analytics\_bp, ai\_bp

    from app.routes.export import export\_bp

    app.register\_blueprint(auth\_bp, url\_prefix='/api/auth')

    app.register\_blueprint(expenses\_bp, url\_prefix='/api/expenses')

    app.register\_blueprint(analytics\_bp, url\_prefix='/api/analytics')

    app.register\_blueprint(ai\_bp, url\_prefix='/api/ai')

    app.register\_blueprint(export\_bp, url\_prefix='/api/export')

    print(f'✓ Registered {len(app.blueprints)} blueprints')

    print('  • /api/auth - Authentication')

    print('  • /api/expenses - Expense Management')

    print('  • /api/analytics - Analytics')

    print('  • /api/ai - AI Insights')

    print('  • /api/export - Export Reports')

    # ✅ FIXED: Add global OPTIONS handler for all API routes

    @app.before\_request

    def handle\_preflight():

        from flask import request

        if request.method == "OPTIONS":

            # Allow all OPTIONS requests

            from flask import make\_response

            response = make\_response()

            response.headers.add("Access-Control-Allow-Origin", "http://localhost:3000")

            response.headers.add('Access-Control-Allow-Headers', "Content-Type,Authorization,X-Requested-With")

            response.headers.add('Access-Control-Allow-Methods', "GET,PUT,POST,DELETE,OPTIONS,PATCH")

            response.headers.add('Access-Control-Allow-Credentials', 'true')

            return response

    # Register error handlers

    register\_error\_handlers(app)

    print('✓ Error handlers registered')

    # Register CLI commands

    register\_cli\_commands(app)

    print('✓ CLI commands registered')

    # Root endpoint

    @app.route('/', methods=['GET'])

    def index():

        """Root endpoint with API information"""

        return jsonify({

            "message": "Expense Tracker API",

            "version": "1.0.0",

            "status": "running",

            "endpoints": {

                "auth": "/api/auth",

                "expenses": "/api/expenses",

                "analytics": "/api/analytics",

                "ai": "/api/ai",

                "export": "/api/export"

            },

            "health": "/health"

        }), 200

    # Health check endpoint

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

    def health():

        """Health check endpoint"""

        try:

            # Test database connection

            db.session.execute(db.text('SELECT 1'))

            db\_status = "connected"

        except Exception as e:

            db\_status = f"error: {str(e)}"

        return jsonify({

            "status": "healthy",

            "database": db\_status,

            "version": "1.0.0",

            "environment": app.config.get('FLASK\_ENV', 'unknown')

        }), 200

    print('=' \* 60)

    print('✓ Flask application initialized successfully')

    print('=' \* 60)

    return app

def register\_error\_handlers(app):

    """Register custom error handlers"""

    @app.errorhandler(400)

    def bad\_request\_error(error):

        return jsonify({

            "error": "Bad Request",

            "message": str(error.description) if hasattr(error, 'description') else "Invalid request"

        }), 400

    @app.errorhandler(401)

    def unauthorized\_error(error):

        return jsonify({

            "error": "Unauthorized",

            "message": "Authentication required"

        }), 401

    @app.errorhandler(403)

    def forbidden\_error(error):

        return jsonify({

            "error": "Forbidden",

            "message": "You don't have permission to access this resource"

        }), 403

    @app.errorhandler(404)

    def not\_found\_error(error):

        return jsonify({

            "error": "Not Found",

            "message": "The requested resource was not found"

        }), 404

    @app.errorhandler(405)

    def method\_not\_allowed\_error(error):

        return jsonify({

            "error": "Method Not Allowed",

            "message": "The HTTP method is not allowed for this endpoint",

            "allowed\_methods": error.valid\_methods if hasattr(error, 'valid\_methods') else []

        }), 405

    @app.errorhandler(500)

    def internal\_error(error):

        db.session.rollback()

        print(f'❌ Internal Server Error: {str(error)}')

        return jsonify({

            "error": "Internal Server Error",

            "message": "An internal server error occurred. Please try again later."

        }), 500

def register\_jwt\_callbacks(jwt, app):

    """Register JWT callbacks with detailed logging"""

    @jwt.expired\_token\_loader

    def expired\_token\_callback(jwt\_header, jwt\_payload):

        print('❌ JWT: Token expired')

        return jsonify({

            "error": "Token Expired",

            "message": "The access token has expired. Please login again."

        }), 401

    @jwt.invalid\_token\_loader

    def invalid\_token\_callback(error):

        print(f'❌ JWT: Invalid token - {error}')

        return jsonify({

            "error": "Invalid Token",

            "message": "Token verification failed. Please login again."

        }), 401

    @jwt.unauthorized\_loader

    def missing\_token\_callback(error):

        print(f'❌ JWT: Missing token - {error}')

        return jsonify({

            "error": "Authorization Required",

            "message": "Request does not contain a valid access token"

        }), 401

    @jwt.revoked\_token\_loader

    def revoked\_token\_callback(jwt\_header, jwt\_payload):

        print('❌ JWT: Token revoked')

        return jsonify({

            "error": "Token Revoked",

            "message": "The token has been revoked. Please login again."

        }), 401

    # ✅ FIXED: Add additional JWT callbacks for better debugging

    @jwt.additional\_claims\_loader

    def add\_claims\_to\_jwt(identity):

        user = User.query.get(identity)

        return {

            'email': user.email if user else None,

            'is\_admin': False  # Add admin check if needed

        }

    @jwt.user\_identity\_loader

    def user\_identity\_lookup(user):

        return user

    @jwt.user\_lookup\_loader

    def user\_lookup\_callback(\_jwt\_header, jwt\_data):

        identity = jwt\_data["sub"]

        user = User.query.filter\_by(id=identity).one\_or\_none()

        print(f'🔍 JWT: Looking up user with ID {identity} - Found: {user is not None}')

        return user

def register\_cli\_commands(app):

    """Register CLI commands"""

    @app.cli.command('seed-db')

    def seed\_database():

        """Seed database with default categories and payment modes"""

        print('=' \* 60)

        print('🌱 Seeding database with default data...')

        print('=' \* 60)

        try:

            # Seed categories

            categories = [

                {"name": "Food", "slug": "food", "color": "#FF6B6B"},

                {"name": "Transportation", "slug": "transportation", "color": "#4ECDC4"},

                {"name": "Shopping", "slug": "shopping", "color": "#45B7D1"},

                {"name": "Entertainment", "slug": "entertainment", "color": "#FFA07A"},

                {"name": "Bills & Utilities", "slug": "bills-utilities", "color": "#98D8C8"},

                {"name": "Healthcare", "slug": "healthcare", "color": "#FF69B4"},

                {"name": "Education", "slug": "education", "color": "#9B59B6"},

                {"name": "Travel", "slug": "travel", "color": "#3498DB"},

                {"name": "Investments", "slug": "investments", "color": "#2ECC71"},

                {"name": "Others", "slug": "others", "color": "#95A5A6"},

            ]

            for cat\_data in categories:

                existing = Category.query.filter\_by(slug=cat\_data['slug']).first()

                if not existing:

                    category = Category(\*\*cat\_data)

                    db.session.add(category)

                    print(f'✓ Added category: {cat\_data["name"]}')

                else:

                    print(f'⊙ Category already exists: {cat\_data["name"]}')

            # Seed payment modes

            payment\_modes = [

                {"name": "GPay", "bankname": "SBI", "type": "digital"},

                {"name": "GPay", "bankname": "HDFC", "type": "digital"},

                {"name": "GPay", "bankname": "IOB", "type": "digital"},

                {"name": "PhonePe", "bankname": "SBI", "type": "digital"},

                {"name": "PhonePe", "bankname": "HDFC", "type": "digital"},

                {"name": "Paytm", "type": "digital"},

                {"name": "Cash", "type": "cash"},

                {"name": "Credit Card", "type": "card"},

                {"name": "Debit Card", "type": "card"},

                {"name": "Net Banking", "type": "digital"},

                {"name": "UPI", "type": "digital"},

                {"name": "Other", "type": "other"},

            ]

            for pm\_data in payment\_modes:

                existing = PaymentMode.query.filter\_by(

                    name=pm\_data['name'],

                    bankname=pm\_data.get('bankname')

                ).first()

                if not existing:

                    payment\_mode = PaymentMode(\*\*pm\_data)

                    db.session.add(payment\_mode)

                    display = f"{pm\_data['name']}"

                    if pm\_data.get('bankname'):

                        display += f" - {pm\_data['bankname']}"

                    print(f'✓ Added payment mode: {display}')

                else:

                    display = f"{pm\_data['name']}"

                    if pm\_data.get('bankname'):

                        display += f" - {pm\_data['bankname']}"

                    print(f'⊙ Payment mode already exists: {display}')

            db.session.commit()

            print('=' \* 60)

            print('✅ Database seeded successfully!')

            print('=' \* 60)

        except Exception as e:

            db.session.rollback()

            print('=' \* 60)

            print(f'❌ Error seeding database: {str(e)}')

            print('=' \* 60)

    @app.cli.command('create-admin')

    def create\_admin():

        """Create admin user"""

        from werkzeug.security import generate\_password\_hash

        print('=' \* 60)

        print('👤 Creating admin user...')

        print('=' \* 60)

        try:

            admin\_email = 'admin@example.com'

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

            if existing:

                print(f'⊙ Admin user already exists: {admin\_email}')

            else:

                admin = User(

                    email=admin\_email,

                    full\_name='Admin User',

                    password='admin123',  # Will be hashed by User model

                    is\_active=True

                )

                db.session.add(admin)

                db.session.commit()

                print(f'✓ Admin user created: {admin\_email}')

                print('  Password: admin123')

            print('=' \* 60)

            print('✅ Admin user ready!')

            print('=' \* 60)

        except Exception as e:

            db.session.rollback()

            print('=' \* 60)

            print(f'❌ Error creating admin: {str(e)}')

            print('=' \* 60)

E:\expense\_tracker\backend\app\extensions.py"""

Flask extensions initialization

Initializes SQLAlchemy, Flask-Migrate, JWT Manager, and other extensions

"""

from flask\_sqlalchemy import SQLAlchemy

from flask\_migrate import Migrate

from flask\_jwt\_extended import JWTManager

# Initialize SQLAlchemy for database operations

db = SQLAlchemy()

# Initialize Flask-Migrate for database migrations

migrate = Migrate()

# Initialize JWT Manager for authentication

jwt = JWTManager()

def init\_extensions(app):

    """

    Alternative initialization function if needed

    Can be used instead of calling init\_app on each extension

    Args:

        app: Flask application instance

    """

    db.init\_app(app)

    migrate.init\_app(app, db)

    jwt.init\_app(app)

    print("✓ All extensions initialized")

E:\expense\_tracker\backend\migrations (related files are there inside)

E:\expense\_tracker\backend\migrations\alembic.ini# A generic, single database configuration.

[alembic]

# template used to generate migration files

# file\_template = %%(rev)s\_%%(slug)s

# set to 'true' to run the environment during

# the 'revision' command, regardless of autogenerate

# revision\_environment = false

# Logging configuration

[loggers]

keys = root,sqlalchemy,alembic,flask\_migrate

[handlers]

keys = console

[formatters]

keys = generic

[logger\_root]

level = WARN

handlers = console

qualname =

[logger\_sqlalchemy]

level = WARN

handlers =

qualname = sqlalchemy.engine

[logger\_alembic]

level = INFO

handlers =

qualname = alembic

[logger\_flask\_migrate]

level = INFO

handlers =

qualname = flask\_migrate

[handler\_console]

class = StreamHandler

args = (sys.stderr,)

level = NOTSET

formatter = generic

[formatter\_generic]

format = %(levelname)-5.5s [%(name)s] %(message)s

datefmt = %H:%M:%S

E:\expense\_tracker\backend\migrations\env.py

import logging

from logging.config import fileConfig

from flask import current\_app

from alembic import context

# this is the Alembic Config object, which provides

# access to the values within the .ini file in use.

config = context.config

# Interpret the config file for Python logging.

# This line sets up loggers basically.

fileConfig(config.config\_file\_name)

logger = logging.getLogger('alembic.env')

def get\_engine():

    try:

        # this works with Flask-SQLAlchemy<3 and Alchemical

        return current\_app.extensions['migrate'].db.get\_engine()

    except (TypeError, AttributeError):

        # this works with Flask-SQLAlchemy>=3

        return current\_app.extensions['migrate'].db.engine

def get\_engine\_url():

    try:

        return get\_engine().url.render\_as\_string(hide\_password=False).replace(

            '%', '%%')

    except AttributeError:

        return str(get\_engine().url).replace('%', '%%')

# add your model's MetaData object here

# for 'autogenerate' support

# from myapp import mymodel

# target\_metadata = mymodel.Base.metadata

config.set\_main\_option('sqlalchemy.url', get\_engine\_url())

target\_db = current\_app.extensions['migrate'].db

# other values from the config, defined by the needs of env.py,

# can be acquired:

# my\_important\_option = config.get\_main\_option("my\_important\_option")

# ... etc.

def get\_metadata():

    if hasattr(target\_db, 'metadatas'):

        return target\_db.metadatas[None]

    return target\_db.metadata

def run\_migrations\_offline():

    """Run migrations in 'offline' mode.

    This configures the context with just a URL

    and not an Engine, though an Engine is acceptable

    here as well.  By skipping the Engine creation

    we don't even need a DBAPI to be available.

    Calls to context.execute() here emit the given string to the

    script output.

    """

    url = config.get\_main\_option("sqlalchemy.url")

    context.configure(

        url=url, target\_metadata=get\_metadata(), literal\_binds=True

    )

    with context.begin\_transaction():

        context.run\_migrations()

def run\_migrations\_online():

    """Run migrations in 'online' mode.

    In this scenario we need to create an Engine

    and associate a connection with the context.

    """

    # this callback is used to prevent an auto-migration from being generated

    # when there are no changes to the schema

    # reference: http://alembic.zzzcomputing.com/en/latest/cookbook.html

    def process\_revision\_directives(context, revision, directives):

        if getattr(config.cmd\_opts, 'autogenerate', False):

            script = directives[0]

            if script.upgrade\_ops.is\_empty():

                directives[:] = []

                logger.info('No changes in schema detected.')

    conf\_args = current\_app.extensions['migrate'].configure\_args

    if conf\_args.get("process\_revision\_directives") is None:

        conf\_args["process\_revision\_directives"] = process\_revision\_directives

    connectable = get\_engine()

    with connectable.connect() as connection:

        context.configure(

            connection=connection,

            target\_metadata=get\_metadata(),

            \*\*conf\_args

        )

        with context.begin\_transaction():

            context.run\_migrations()

if context.is\_offline\_mode():

    run\_migrations\_offline()

else:

    run\_migrations\_online()

E:\expense\_tracker\backend\migrations\script.py.mako

"""${message}

Revision ID: ${up\_revision}

Revises: ${down\_revision | comma,n}

Create Date: ${create\_date}

"""

from alembic import op

import sqlalchemy as sa

${imports if imports else ""}

# revision identifiers, used by Alembic.

revision = ${repr(up\_revision)}

down\_revision = ${repr(down\_revision)}

branch\_labels = ${repr(branch\_labels)}

depends\_on = ${repr(depends\_on)}

def upgrade():

${upgrades if upgrades else "pass"}

def downgrade():

${downgrades if downgrades else "pass"}

E:\expense\_tracker\backend\venv (file related to the virtual environment

E:\expense\_tracker\backend\.env

# Database Configuration

DB\_HOST=localhost

DB\_PORT=5432

DB\_NAME=expense\_db

DB\_USER=postgres

DB\_PASSWORD=Bharadwaj2112

# Flask Configuration

SECRET\_KEY=9205e0540e4adb7d02ee88d9f4724e24

JWT\_SECRET\_KEY=9205e0540e4adb7d02ee88d9f4724e24

FLASK\_ENV=development

FLASK\_APP=run.py

# OpenAI Configuration

OPENAI\_API\_KEY=sk-proj-xzOA8GEkVwdQrKMUkPk7yDWWjiAj\_4SoVq4F7AMSolehThi3oSUg4AsDkAoalC9QNSSdgtqBXnT3BlbkFJSDN28FAAwxQ1\_OPELoncIga8jtPNT9Ol6R6DW50mi-qB4hts6arkH9ZOyJJZ5smusfKI8x89MA

OPENAI\_MODEL=gpt-4o-mini

# Optional: CORS Configuration

FRONTEND\_URL=http://localhost:3000

E:\expense\_tracker\backend\.gitignore

# Environment Variables

.env

.env.local

.env.production

# Python Cache

\_\_pycache\_\_/

\*.py[cod]

\*$py.class

\*.so

.Python

# Virtual Environment

venv/

env/

ENV/

.venv

# Flask

instance/

.webassets-cache

\*.db

\*.sqlite3

# IDE

.vscode/

.idea/

\*.swp

\*.swo

\*~

# OS

.DS\_Store

Thumbs.db

# Logs

\*.log

logs/

# Testing

.pytest\_cache/

.coverage

htmlcov/

.tox/

# Database

\*.sql

\*.dump

# Build

build/

dist/

\*.egg-info/

# Vercel

.vercel

E:\expense\_tracker\backend\config.py

"""

Configuration settings for different environments

Loads settings from .env file with security best practices

"""

import os

from dotenv import load\_dotenv

from datetime import timedelta

# Load environment variables from .env file

basedir = os.path.abspath(os.path.dirname(\_\_file\_\_))

load\_dotenv(os.path.join(basedir, '.env'))

class Config:

    """Base configuration class with common settings"""

    # Flask Configuration

    SECRET\_KEY = os.environ.get('SECRET\_KEY') or 'dev-secret-key-change-in-production'

    FLASK\_APP = os.environ.get('FLASK\_APP') or 'run.py'

    FLASK\_ENV = os.environ.get('FLASK\_ENV') or 'development'

    # Database Configuration

    DB\_HOST = os.environ.get('DB\_HOST', 'localhost')

    DB\_PORT = os.environ.get('DB\_PORT', '5432')

    DB\_NAME = os.environ.get('DB\_NAME', 'expense\_db')

    DB\_USER = os.environ.get('DB\_USER', 'postgres')

    DB\_PASSWORD = os.environ.get('DB\_PASSWORD', '')

    # Construct PostgreSQL database URI

    SQLALCHEMY\_DATABASE\_URI = os.environ.get('DATABASE\_URL') or (

        f"postgresql://{DB\_USER}:{DB\_PASSWORD}@{DB\_HOST}:{DB\_PORT}/{DB\_NAME}"

    )

    # Fix for Heroku/Vercel postgres URLs (postgres:// -> postgresql://)

    if SQLALCHEMY\_DATABASE\_URI and SQLALCHEMY\_DATABASE\_URI.startswith('postgres://'):

        SQLALCHEMY\_DATABASE\_URI = SQLALCHEMY\_DATABASE\_URI.replace('postgres://', 'postgresql://', 1)

    # SQLAlchemy Configuration

    SQLALCHEMY\_TRACK\_MODIFICATIONS = False

    SQLALCHEMY\_ECHO = False  # Set to True to see SQL queries in console

    SQLALCHEMY\_ENGINE\_OPTIONS = {

        'pool\_size': 10,

        'pool\_recycle': 3600,

        'pool\_pre\_ping': True,

        'connect\_args': {

            'connect\_timeout': 10

        }

    }

    # JWT Configuration

    JWT\_SECRET\_KEY = os.environ.get('JWT\_SECRET\_KEY') or 'jwt-secret-key-change-in-production'

    JWT\_ACCESS\_TOKEN\_EXPIRES = timedelta(hours=24)  # 24 hours

    JWT\_REFRESH\_TOKEN\_EXPIRES = timedelta(days=30)  # 30 days

    JWT\_ALGORITHM = 'HS256'

    JWT\_TOKEN\_LOCATION = ['headers']

    JWT\_HEADER\_NAME = 'Authorization'

    JWT\_HEADER\_TYPE = 'Bearer'

    # ✅ CHANGED: OpenAI Configuration (was Gemini)

    OPENAI\_API\_KEY = os.environ.get('OPENAI\_API\_KEY')

    # Valid model options: 'gpt-4o', 'gpt-4o-mini', 'gpt-3.5-turbo'

    OPENAI\_MODEL = os.environ.get('OPENAI\_MODEL', 'gpt-4o-mini')

    OPENAI\_MAX\_TOKENS = 1000

    OPENAI\_TEMPERATURE = 0.7  # Control response creativity (0.0 to 1.0)

    # CORS Configuration

    CORS\_HEADERS = 'Content-Type'

    FRONTEND\_URL = os.environ.get('FRONTEND\_URL', 'http://localhost:3000')

    # Security Configuration

    SESSION\_COOKIE\_SECURE = True  # Only send cookies over HTTPS

    SESSION\_COOKIE\_HTTPONLY = True  # Prevent JavaScript access to session cookies

    SESSION\_COOKIE\_SAMESITE = 'Lax'  # CSRF protection

    PERMANENT\_SESSION\_LIFETIME = timedelta(days=7)

    # Rate Limiting Configuration

    RATELIMIT\_ENABLED = True

    RATELIMIT\_STORAGE\_URL = os.environ.get('REDIS\_URL', 'memory://')

    # Pagination Configuration

    DEFAULT\_PAGE\_SIZE = 50

    MAX\_PAGE\_SIZE = 100

    # File Upload Configuration (if needed in future)

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

    UPLOAD\_FOLDER = os.path.join(basedir, 'uploads')

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

    # Logging Configuration

    LOG\_LEVEL = os.environ.get('LOG\_LEVEL', 'INFO')

    LOG\_FORMAT = '%(asctime)s - %(name)s - %(levelname)s - %(message)s'

    LOG\_FILE = os.path.join(basedir, 'logs', 'app.log')

    # Application Information

    APP\_NAME = 'Expense Tracker'

    APP\_VERSION = '1.0.0'

    API\_TITLE = 'Expense Tracker API'

    API\_VERSION = 'v1'

    # Timezone Configuration

    TIMEZONE = 'Asia/Kolkata'  # IST

    @staticmethod

    def init\_app(app):

        """

        Static method to perform application-specific initialization

        Can be overridden in subclasses

        """

        pass

class DevelopmentConfig(Config):

    """Development environment configuration"""

    DEBUG = True

    TESTING = False

    SQLALCHEMY\_ECHO = True  # Show SQL queries in development

    # Development-specific settings

    SESSION\_COOKIE\_SECURE = False  # Allow HTTP in development

    RATELIMIT\_ENABLED = False  # Disable rate limiting in development

    # ✅ CHANGED: More lenient OpenAI settings for development

    OPENAI\_MAX\_TOKENS = 1500  # Allow longer responses in development

    @staticmethod

    def init\_app(app):

        """Development-specific initialization"""

        print('✓ Running in DEVELOPMENT mode')

        print('⚠️  Debug mode is ON - Do not use in production!')

        # ✅ CHANGED: Check OpenAI configuration

        if app.config.get('OPENAI\_API\_KEY'):

            print(f"✓ OpenAI configured with model: {app.config.get('OPENAI\_MODEL')}")

        else:

            print('⚠️  OpenAI not configured - AI features will be disabled')

            print('💡 Get your API key from: https://platform.openai.com/api-keys')

class ProductionConfig(Config):

    """Production environment configuration"""

    DEBUG = False

    TESTING = False

    SQLALCHEMY\_ECHO = False

    # Production-specific settings

    SESSION\_COOKIE\_SECURE = True

    RATELIMIT\_ENABLED = True

    # ✅ CHANGED: Stricter OpenAI settings for production (cost control)

    OPENAI\_MAX\_TOKENS = 800  # Shorter responses in production for cost control

    OPENAI\_TEMPERATURE = 0.6  # More consistent responses in production

    # Ensure required environment variables are set

    @classmethod

    def init\_app(cls, app):

        """Production-specific initialization"""

        Config.init\_app(app)

        # Validate required environment variables

        required\_vars = [

            'SECRET\_KEY',

            'JWT\_SECRET\_KEY',

            'DATABASE\_URL',

        ]

        # ✅ CHANGED: OpenAI API key is optional, not required

        optional\_vars = ['OPENAI\_API\_KEY']

        missing\_vars = []

        for var in required\_vars:

            if not os.environ.get(var):

                missing\_vars.append(var)

        if missing\_vars:

            raise ValueError(

                f"Missing required environment variables for production: {', '.join(missing\_vars)}"

            )

        # Ensure secret keys are not default values

        if app.config['SECRET\_KEY'] == 'dev-secret-key-change-in-production':

            raise ValueError("SECRET\_KEY must be changed for production!")

        if app.config['JWT\_SECRET\_KEY'] == 'jwt-secret-key-change-in-production':

            raise ValueError("JWT\_SECRET\_KEY must be changed for production!")

        print('✓ Running in PRODUCTION mode')

        print('✓ All security checks passed')

        # Warn about optional configs

        for var in optional\_vars:

            if not os.environ.get(var):

                print(f'⚠️  Optional config {var} not set - some features may be disabled')

            else:

                print(f'✓ {var} configured')

class TestingConfig(Config):

    """Testing environment configuration"""

    TESTING = True

    DEBUG = True

    # Use in-memory SQLite database for tests

    SQLALCHEMY\_DATABASE\_URI = 'sqlite:///:memory:'

    # Disable CSRF protection in testing

    WTF\_CSRF\_ENABLED = False

    # Speed up password hashing for tests

    BCRYPT\_LOG\_ROUNDS = 4

    # Disable rate limiting in tests

    RATELIMIT\_ENABLED = False

    # Short token expiry for testing

    JWT\_ACCESS\_TOKEN\_EXPIRES = timedelta(minutes=5)

    # ✅ CHANGED: Disable OpenAI in tests

    OPENAI\_API\_KEY = None  # Don't use real API in tests

    @staticmethod

    def init\_app(app):

        """Testing-specific initialization"""

        print('✓ Running in TESTING mode')

        print('⚠️  Using in-memory database - data will not persist')

class VercelConfig(ProductionConfig):

    """Vercel-specific production configuration"""

    @staticmethod

    def init\_app(app):

        """Vercel-specific initialization"""

        ProductionConfig.init\_app(app)

        # Vercel-specific settings

        print('✓ Configured for Vercel deployment')

        # Set up logging for Vercel

        import logging

        logging.basicConfig(

            level=logging.INFO,

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

        )

# Configuration dictionary for easy access

config = {

    'development': DevelopmentConfig,

    'production': ProductionConfig,

    'testing': TestingConfig,

    'vercel': VercelConfig,

    'default': DevelopmentConfig

}

def get\_config():

    """

    Get configuration based on FLASK\_ENV environment variable

    Returns:

        Config class for the current environment

    """

    env = os.environ.get('FLASK\_ENV', 'development')

    return config.get(env, config['default'])

E:\expense\_tracker\backend\requirements.txt

Flask==3.0.3

Flask-SQLAlchemy==3.1.1

Flask-Migrate==4.0.7

Flask-JWT-Extended==4.6.0

Flask-CORS==4.0.1

psycopg2-binary==2.9.10

python-dotenv==1.0.1

werkzeug==3.0.3

google-generativeai==0.8.3

reportlab==4.2.5

pandas==2.2.3

E:\expense\_tracker\backend\run.py

"""

Application entry point for development server

Run this file to start the Flask development server

Usage:

    python run.py

Environment Variables:

    FLASK\_ENV: development, production, testing (default: development)

    PORT: Port number (default: 5000)

    HOST: Host address (default: 0.0.0.0)

"""

import os

import sys

from app import create\_app

from config import get\_config

# Create Flask application instance with environment-specific config

config\_class = get\_config()

app = create\_app(config\_class)

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

    # Get configuration from environment

    port = int(os.environ.get('PORT', 5000))

    host = os.environ.get('HOST', '0.0.0.0')

    debug = os.environ.get('FLASK\_ENV', 'development') == 'development'

    # Warning for production

    if os.environ.get('FLASK\_ENV') == 'production':

        print('\n' + '=' \* 60)

        print('⚠️  WARNING: Development Server in Production Mode!')

        print('=' \* 60)

        print('⚠️  Please use a production WSGI server instead.')

        print('⚠️  Example: gunicorn -w 4 -b 0.0.0.0:5000 wsgi:app')

        print('=' \* 60 + '\n')

        sys.exit(1)

    # Print startup information

    print('\n' + '=' \* 60)

    print(f'🚀 Starting {app.config["APP\_NAME"]} v{app.config["APP\_VERSION"]}')

    print('=' \* 60)

    print(f'📍 Environment:    {app.config["FLASK\_ENV"]}')

    print(f'🌐 Server URL:     http://localhost:{port}')

    print(f'🌐 Local Access:   http://127.0.0.1:{port}')

    print(f'🔍 Debug Mode:     {"✅ Enabled" if debug else "❌ Disabled"}')

    print(f'🔐 JWT Enabled:    ✅ Yes')

    # ✅ CHANGED: Check for OpenAI instead of Gemini

    print(f'🤖 OpenAI:         {"✅ Configured (" + app.config.get("OPENAI\_MODEL", "N/A") + ")" if app.config.get("OPENAI\_API\_KEY") else "❌ Not Configured"}')

    print(f'🗄️  Database:       ✅ PostgreSQL Connected')

    print(f'🌍 CORS:           ✅ Enabled (All origins for development)')

    print('=' \* 60)

    print('\n💡 Available API Endpoints:')

    print(f'   • http://localhost:{port}/                    - API Info')

    print(f'   • http://localhost:{port}/health              - Health Check')

    print(f'   • http://localhost:{port}/api/auth/login      - Login (POST)')

    print(f'   • http://localhost:{port}/api/auth/register   - Register (POST)')

    print(f'   • http://localhost:{port}/api/expenses        - Expense Management')

    print(f'   • http://localhost:{port}/api/analytics       - Analytics & Reports')

    print(f'   • http://localhost:{port}/api/ai              - AI Query (OpenAI)')

    print(f'   • http://localhost:{port}/api/export          - Export (CSV/PDF)')

    print('\n💡 Frontend Connection:')

    print(f'   • React App:     http://localhost:3000')

    print(f'   • API Base URL:  http://localhost:{port}/api')

    print('\n💡 CLI Commands:')

    print('   • flask seed-db        - Seed database with default data')

    print('   • flask create-admin   - Create admin user')

    print('\n💡 Test Commands:')

    print(f'   • curl http://localhost:{port}/health')

    print(f'   • curl http://localhost:{port}/api/auth/login -X POST -H "Content-Type: application/json" -d \'{{"email":"test@example.com","password":"password123"}}\'')

    print(f'   • curl http://localhost:{port}/api/ai/query -X POST -H "Content-Type: application/json" -H "Authorization: Bearer YOUR\_TOKEN" -d \'{{"query":"What did I spend this week?"}}\'')

    print('\n💡 Press CTRL+C to quit')

    print('=' \* 60 + '\n')

    # Check database connection

    try:

        with app.app\_context():

            from app.extensions import db

            db.session.execute(db.text('SELECT 1'))

            print('✅ Database connection verified')

    except Exception as e:

        print(f'❌ Database connection failed: {str(e)}')

        print('⚠️  Please check your database configuration in .env file')

        print('=' \* 60 + '\n')

        sys.exit(1)

    # ✅ ADDED: Check OpenAI configuration

    if app.config.get('OPENAI\_API\_KEY'):

        print(f'✅ OpenAI configured with model: {app.config.get("OPENAI\_MODEL", "gpt-4o-mini")}')

    else:

        print('⚠️  OpenAI not configured - AI features will be limited')

        print('💡 Add OPENAI\_API\_KEY to your .env file to enable AI features')

        print('💡 Get your API key from: https://platform.openai.com/api-keys')

    print()  # Empty line for spacing

    # Run development server

    try:

        app.run(

            host=host,

            port=port,

            debug=debug,

            use\_reloader=debug,

            threaded=True

        )

    except KeyboardInterrupt:

        print('\n\n' + '=' \* 60)

        print('✓ Server stopped by user')

        print('=' \* 60 + '\n')

    except OSError as e:

        if 'Address already in use' in str(e):

            print('\n\n' + '=' \* 60)

            print(f'❌ Port {port} is already in use!')

            print('=' \* 60)

            print(f'💡 Solutions:')

            print(f'   1. Stop the other process using port {port}')

            print(f'   2. Use a different port: PORT=5001 python run.py')

            print(f'   3. Find process: netstat -ano | findstr :{port} (Windows)')

            print('=' \* 60 + '\n')

        else:

            print('\n\n' + '=' \* 60)

            print(f'❌ Error starting server: {str(e)}')

            print('=' \* 60 + '\n')

        sys.exit(1)

    except Exception as e:

        print('\n\n' + '=' \* 60)

        print(f'❌ Unexpected error: {str(e)}')

        print('=' \* 60 + '\n')

        sys.exit(1)

For development yet didn’t do anything

E:\expense\_tracker\backend\wsgi.py

"""

WSGI Entry Point for Production Deployment

Used by production WSGI servers (Gunicorn, uWSGI, Vercel)

Usage with Gunicorn:

    gunicorn -w 4 -b 0.0.0.0:5000 wsgi:app

Usage with uWSGI:

    uwsgi --http :5000 --wsgi-file wsgi.py --callable app

Vercel:

    Automatically detects this file for serverless deployment

"""

import os

from app import create\_app

from config import config

# Determine environment

environment = os.environ.get('FLASK\_ENV', 'production')

# For Vercel, always use production config

if os.environ.get('VERCEL'):

    config\_class = config['vercel']

else:

    config\_class = config.get(environment, config['production'])

# Create application instance

app = create\_app(config\_class)

# Application initialization logging

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

    # This runs when loaded by a WSGI server

    print('=' \* 60)

    print(f'✓ {app.config["APP\_NAME"]} loaded successfully')

    print(f'✓ Environment: {environment}')

    print(f'✓ Version: {app.config["APP\_VERSION"]}')

    print('=' \* 60)

# For direct execution (not recommended for production)

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

    print('⚠️  WARNING: This file is meant for WSGI servers, not direct execution!')

    print('⚠️  For development, use: python run.py')

    print('⚠️  For production, use a WSGI server like:')

    print('     gunicorn -w 4 -b 0.0.0.0:5000 wsgi:app')

E:\expense\_tracker\backend\vercel.json

{

  "version": 2,

  "name": "expense-tracker-backend",

  "builds": [

    {

      "src": "wsgi.py",

      "use": "@vercel/python"

    }

  ],

  "routes": [

    {

      "src": "/(.\*)",

      "dest": "wsgi.py"

    }

  ],

  "env": {

    "FLASK\_ENV": "production"

  },

  "regions": ["bom1"]

}