from fastapi import FastAPI, File, UploadFile, HTTPException, Query

from fastapi.responses import JSONResponse

import pandas as pd

import io

from typing import Dict, Any, List, Optional

import pyodbc

import os

from contextlib import contextmanager

app = FastAPI(title="CSV to JSON Converter", version="1.0.0")

*# MS SQL Server Database configuration*

DB\_NAME = "ITMS\_06\_03\_2025"

HOST\_NAME = "103.248.60.42"

DB\_USERNAME = "sa"

DB\_PASSWORD = "vtpl@123"

MSSQL\_DRIVER = "{ODBC Driver 17 for SQL Server}"

TABLE\_NAME = "MTN\_VehicleInsurance"

*# Connection string for MS SQL Server*

CONNECTION\_STRING = f"DRIVER={MSSQL\_DRIVER};SERVER={HOST\_NAME};DATABASE={DB\_NAME};UID={DB\_USERNAME};PWD={DB\_PASSWORD};TrustServerCertificate=yes;"

*# Global variable to store the uploaded CSV data (keeping for backward compatibility)*

uploaded\_data = None

@contextmanager

def get\_db\_connection():

    """Context manager for MS SQL Server database connections"""

    conn = pyodbc.connect(CONNECTION\_STRING)

    conn.autocommit = False  *# Ensure manual commit control*

    try:

        yield conn

    except Exception:

        conn.rollback()  *# Rollback on error*

        raise

    finally:

        conn.close()

def init\_database():

    """Check database connection"""

    try:

        with get\_db\_connection() as conn:

            cursor = conn.cursor()

*# Test connection*

            cursor.execute("SELECT 1")

            print("Database connection successful!")

    except Exception as e:

        print(f"Database connection failed: {e}")

def authenticate\_user(username: str, password: str) -> bool:

    """Simple authentication function"""

    return username == DB\_USERNAME and password == DB\_PASSWORD

*# Initialize database on startup*

init\_database()

@app.post("/upload-csv/")

async def upload\_csv(

    file: UploadFile = File(...),

    username: str = Query(..., description="Database username"),

    password: str = Query(..., description="Database password"),

    save\_to\_db: bool = Query(False, description="Save data to database")

):

    """

    Upload a CSV file and optionally save to database

    """

    global uploaded\_data

*# Authenticate user*

    if save\_to\_db and not authenticate\_user(username, password):

        raise HTTPException(status\_code=401, detail="Invalid username or password")

*# Check if the uploaded file is a CSV*

    if not file.filename.endswith('.csv'):

        raise HTTPException(status\_code=400, detail="Only CSV files are allowed")

    try:

*# Read the uploaded file content*

        content = await file.read()

*# Create a StringIO object to read the CSV content*

        csv\_data = io.StringIO(content.decode('utf-8'))

*# Read CSV using pandas*

        df = pd.read\_csv(csv\_data)

*# Store the data globally for filtering (backward compatibility)*

        uploaded\_data = df

        print(df)

*# Save to database if requested*

        if save\_to\_db:

            print("into save\_data\_to\_database")

            save\_data\_to\_database(df)

*# Convert DataFrame to JSON*

        json\_data = df.to\_dict(orient='records')

        response = {

            "filename": file.filename,

            "total\_records": len(json\_data),

            "columns": list(df.columns),

            "data": json\_data

        }

        if save\_to\_db:

            response["database\_status"] = "Data saved to database successfully"

        return response

    except Exception as e:

        raise HTTPException(status\_code=500, detail=f"Error processing file: {str(e)}")

def save\_data\_to\_database(df: pd.DataFrame):

    """Save DataFrame to MS SQL Server database - ALLOW DUPLICATE BusInformationId VALUES"""

    with get\_db\_connection() as conn:

        cursor = conn.cursor()

*# Get existing table schema with column details*

        cursor.execute("""

            SELECT COLUMN\_NAME, DATA\_TYPE, IS\_NULLABLE, COLUMN\_DEFAULT,

                   COLUMNPROPERTY(OBJECT\_ID(TABLE\_SCHEMA+'.'+TABLE\_NAME), COLUMN\_NAME, 'IsIdentity') as IsIdentity

            FROM INFORMATION\_SCHEMA.COLUMNS

            WHERE TABLE\_NAME = ? AND TABLE\_SCHEMA = 'dbo'

            ORDER BY ORDINAL\_POSITION

        """, (TABLE\_NAME,))

        table\_schema = cursor.fetchall()

        if not table\_schema:

            raise Exception(f"Table {TABLE\_NAME} not found in database")

*# Create mapping of database columns (excluding identity and system columns)*

        db\_columns = {}

        insertable\_columns = []

        identity\_column = None

        for col\_info in table\_schema:

            col\_name = col\_info[0]

            is\_identity = col\_info[4] == 1

            if is\_identity:

                identity\_column = col\_name

                print(f"Identity column detected: {col\_name}")

            elif col\_name.lower() not in ['createdat', 'updatedat', 'createdby', 'updatedby']:

*# Skip system audit columns for CSV import*

                db\_columns[col\_name.lower()] = col\_name

                insertable\_columns.append(col\_name)

        print(f"Available columns for insert: {insertable\_columns}")

*# Map CSV columns to database columns*

        column\_mapping = {}

        unmapped\_csv\_columns = []

        for csv\_col in df.columns:

*# Try exact match first*

            if csv\_col.lower() in db\_columns:

                column\_mapping[csv\_col] = db\_columns[csv\_col.lower()]

*# Try cleaned match*

            else:

                clean\_col = csv\_col.replace(' ', '').replace('-', '').replace('\_', '').lower()

                found = False

                for db\_col\_lower, db\_col\_actual in db\_columns.items():

                    if clean\_col == db\_col\_lower.replace('\_', ''):

                        column\_mapping[csv\_col] = db\_col\_actual

                        found = True

                        break

                if not found:

                    unmapped\_csv\_columns.append(csv\_col)

        print(f"Column mapping: {column\_mapping}")

        if unmapped\_csv\_columns:

            print(f"Warning: These CSV columns could not be mapped to database columns: {unmapped\_csv\_columns}")

        if not column\_mapping:

            raise Exception("No CSV columns could be mapped to database columns")

*# Validate foreign key constraint - Check if BusInformationId values exist in MTN\_BusInformation*

        bus\_info\_column = None

        for csv\_col, db\_col in column\_mapping.items():

            if 'businformationid' in db\_col.lower():

                bus\_info\_column = csv\_col

                break

        if bus\_info\_column:

            print(f"Found BusInformationId column: {bus\_info\_column}")

*# Get unique BusInformationId values from CSV*

            unique\_bus\_ids = df[bus\_info\_column].dropna().unique()

            if len(unique\_bus\_ids) > 0:

*# Check which BusInformationId values exist in MTN\_BusInformation table*

                placeholders = ','.join(['?' for \_ in unique\_bus\_ids])

                cursor.execute(f"""

                    SELECT BusInformationId FROM MTN\_BusInformation

                    WHERE BusInformationId IN ({placeholders})

                """, tuple(unique\_bus\_ids))

                existing\_bus\_ids = set([row[0] for row in cursor.fetchall()])

                missing\_bus\_ids = set(unique\_bus\_ids) - existing\_bus\_ids

                if missing\_bus\_ids:

                    print(f"Warning: These BusInformationId values don't exist in MTN\_BusInformation: {missing\_bus\_ids}")

                    print("Records with these IDs will be skipped to maintain foreign key integrity")

                print(f"Valid BusInformationId values: {existing\_bus\_ids}")

*# Prepare insert statement*

        mapped\_db\_columns = list(column\_mapping.values())

        csv\_columns\_to\_use = [col for col in df.columns if col in column\_mapping]

        placeholders = ', '.join(['?' for \_ in mapped\_db\_columns])

        column\_names = ', '.join([f'[{col}]' for col in mapped\_db\_columns])

        insert\_sql = f"INSERT INTO [{TABLE\_NAME}] ({column\_names}) VALUES ({placeholders})"

        print(f"Insert SQL: {insert\_sql}")

*# Insert data row by row - ALLOW DUPLICATE BusInformationId VALUES*

        inserted\_count = 0

        failed\_count = 0

        skipped\_foreign\_key\_count = 0

        for index, row in df.iterrows():

            try:

*# Check foreign key constraint if BusInformationId column exists*

                if bus\_info\_column and bus\_info\_column in row:

                    bus\_id = row[bus\_info\_column]

                    if pd.notna(bus\_id) and bus\_id not in existing\_bus\_ids:

                        print(f"Skipping row {index + 1}: BusInformationId {bus\_id} doesn't exist in MTN\_BusInformation")

                        skipped\_foreign\_key\_count += 1

                        continue

*# Extract and convert values for mapped columns only*

                values = []

                for csv\_col in csv\_columns\_to\_use:

                    val = row[csv\_col]

                    db\_col = column\_mapping[csv\_col]

*# Handle None/NaN values*

                    if pd.isna(val) or val is None or val == '':

                        values.append(None)

                    else:

*# Convert based on database column type*

                        target\_col\_info = next((col for col in table\_schema if col[0] == db\_col), None)

                        if target\_col\_info:

                            data\_type = target\_col\_info[1].lower()

*# Convert based on SQL Server data types*

                            if data\_type in ['int', 'bigint', 'smallint', 'tinyint']:

                                try:

                                    values.append(int(float(val)) if val != '' else None)

                                except (ValueError, TypeError):

                                    values.append(None)

                            elif data\_type in ['decimal', 'numeric', 'float', 'real']:

                                try:

                                    values.append(float(val) if val != '' else None)

                                except (ValueError, TypeError):

                                    values.append(None)

                            elif data\_type in ['date', 'datetime', 'datetime2']:

                                try:

                                    if isinstance(val, str) and val.strip():

                                        parsed\_date = pd.to\_datetime(val, errors='coerce')

                                        values.append(parsed\_date if not pd.isna(parsed\_date) else None)

                                    else:

                                        values.append(None)

                                except:

                                    values.append(None)

                            else:

                                values.append(str(val) if val is not None else None)

                        else:

                            values.append(str(val) if val is not None else None)

*# INSERT THE RECORD - No duplicate checking for BusInformationId*

                try:

                    cursor.execute(insert\_sql, tuple(values))

                    inserted\_count += 1

*# Commit every 50 rows for better performance*

                    if inserted\_count % 50 == 0:

                        conn.commit()

                        print(f"Inserted {inserted\_count} rows so far...")

                except Exception as insert\_error:

*# Only handle actual constraint violations or database errors*

                    print(f"Error inserting row {index + 1}: {insert\_error}")

                    failed\_count += 1

*# Continue to next row*

                    continue

            except Exception as row\_error:

                print(f"Error processing row {index + 1}: {row\_error}")

                print(f"Row data: {dict(zip(csv\_columns\_to\_use, [row[col] for col in csv\_columns\_to\_use]))}")

                failed\_count += 1

*# Continue to next row*

                continue

*# Final commit*

        conn.commit()

        print(f"Data insertion completed!")

        print(f"Successfully inserted: {inserted\_count} rows")

        print(f"Skipped due to foreign key constraint: {skipped\_foreign\_key\_count} rows")

        print(f"Failed insertions: {failed\_count} rows")

*# Show some statistics about the inserted data*

        if bus\_info\_column and inserted\_count > 0:

            cursor.execute(f"""

                SELECT [{column\_mapping[bus\_info\_column]}], COUNT(\*) as count

                FROM [{TABLE\_NAME}]

                GROUP BY [{column\_mapping[bus\_info\_column]}]

                ORDER BY count DESC

            """)

            bus\_id\_counts = cursor.fetchall()

            print(f"BusInformationId distribution after insert (top 10):")

            for i, (bus\_id, count) in enumerate(bus\_id\_counts[:10]):

                print(f"  BusInformationId {bus\_id}: {count} records")

        return {

            "inserted\_count": inserted\_count,

            "skipped\_foreign\_key\_count": skipped\_foreign\_key\_count,

            "failed\_count": failed\_count,

            "total\_processed": len(df),

            "column\_mapping": column\_mapping,

            "unmapped\_columns": unmapped\_csv\_columns

        }

@app.get("/upload-csv")

async def get\_csv\_by\_company(

    company\_id: int = Query(..., description="Company ID to filter data"),

    from\_database: bool = Query(False, description="Fetch data from database"),

    username: Optional[str] = Query(None, description="Database username (required if from\_database=True)"),

    password: Optional[str] = Query(None, description="Database password (required if from\_database=True)")

):

    """

    Get CSV data filtered by company\_id from memory or database

    """

    global uploaded\_data

    if from\_database:

*# Authenticate user for database access*

        if not username or not password:

            raise HTTPException(status\_code=400, detail="Username and password required for database access")

        if not authenticate\_user(username, password):

            raise HTTPException(status\_code=401, detail="Invalid username or password")

        return get\_data\_from\_database(company\_id)

    else:

*# Use existing in-memory logic*

        if uploaded\_data is None:

            raise HTTPException(status\_code=404, detail="No CSV data found. Please upload a CSV file first using POST /upload-csv/")

        try:

*# Check if company\_id column exists (case insensitive)*

            company\_col = None

            for col in uploaded\_data.columns:

                if 'company' in col.lower() and 'id' in col.lower():

                    company\_col = col

                    break

            if company\_col is None:

                raise HTTPException(status\_code=400, detail="No company\_id column found in the uploaded data")

*# Filter data by company\_id*

            filtered\_df = uploaded\_data[uploaded\_data[company\_col] == company\_id]

            if filtered\_df.empty:

                return {

                    "company\_id": company\_id,

                    "total\_records": 0,

                    "message": f"No records found for company\_id: {company\_id}",

                    "data": []

                }

*# Convert filtered data to JSON*

            json\_data = filtered\_df.to\_dict(orient='records')

            return {

                "company\_id": company\_id,

                "total\_records": len(json\_data),

                "columns": list(filtered\_df.columns),

                "data": json\_data

            }

        except Exception as e:

            raise HTTPException(status\_code=500, detail=f"Error filtering data: {str(e)}")

def get\_data\_from\_database(company\_id: int):

    """Fetch data from MS SQL Server database filtered by company\_id"""

    try:

        with get\_db\_connection() as conn:

            cursor = conn.cursor()

*# Check if table exists*

            cursor.execute("""

                SELECT COUNT(\*) FROM INFORMATION\_SCHEMA.TABLES

                WHERE TABLE\_NAME = ?

            """, (TABLE\_NAME,))

            if cursor.fetchone()[0] == 0:

                raise HTTPException(status\_code=404, detail=f"Table '{TABLE\_NAME}' not found in database")

*# Get column names*

            cursor.execute("""

                SELECT COLUMN\_NAME FROM INFORMATION\_SCHEMA.COLUMNS

                WHERE TABLE\_NAME = ?

            """, (TABLE\_NAME,))

            columns = [row[0] for row in cursor.fetchall()]

            company\_col = None

            for col in columns:

                if 'company' in col.lower() and 'id' in col.lower():

                    company\_col = col

                    break

            if company\_col is None:

                raise HTTPException(status\_code=400, detail="No company\_id column found in the database table")

*# Fetch filtered data*

            cursor.execute(f"""

                SELECT \* FROM [{TABLE\_NAME}]

                WHERE [{company\_col}] = ?

            """, (company\_id,))

            rows = cursor.fetchall()

            if not rows:

                return {

                    "company\_id": company\_id,

                    "total\_records": 0,

                    "message": f"No records found for company\_id: {company\_id}",

                    "data": [],

                    "source": "database"

                }

*# Convert to list of dictionaries*

            data = []

            for row in rows:

                row\_dict = {}

                for i, col in enumerate(columns):

                    row\_dict[col] = row[i]

                data.append(row\_dict)

            return {

                "company\_id": company\_id,

                "total\_records": len(data),

                "columns": columns,

                "data": data,

                "source": "database"

            }

    except Exception as e:

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

@app.get("/")

async def root():

    """

    Root endpoint with API information

    """

    return {

        "message": "CSV to JSON Converter API",

        "endpoints": {

            "upload\_csv": "/upload-csv/ (POST)",

            "docs": "/docs",

            "redoc": "/redoc"

        }

    }

@app.get("/health")

async def health\_check():

    """

    Health check endpoint

    """

    return {"status": "healthy"}

@app.get("/database/all-data")

async def get\_all\_database\_data(

    username: str = Query(..., description="Database username"),

    password: str = Query(..., description="Database password"),

    table\_name: str = Query(TABLE\_NAME, description="Table name"),

    limit: int = Query(100, description="Maximum number of records to return (default: 100, max: 1000)"),

    offset: int = Query(0, description="Number of records to skip (for pagination)")

):

    """Get all data from the database table with pagination"""

    if not authenticate\_user(username, password):

        raise HTTPException(status\_code=401, detail="Invalid username or password")

*# Limit the maximum records to prevent overloading*

    if limit > 1000:

        limit = 1000

    try:

        with get\_db\_connection() as conn:

            cursor = conn.cursor()

*# Check if table exists*

            cursor.execute("""

                SELECT COUNT(\*) FROM INFORMATION\_SCHEMA.TABLES

                WHERE TABLE\_NAME = ?

            """, (table\_name,))

            table\_exists = cursor.fetchone()[0] > 0

            if not table\_exists:

                return {

                    "table\_exists": False,

                    "message": f"Table '{table\_name}' does not exist in database"

                }

*# Get total row count*

            cursor.execute(f"SELECT COUNT(\*) FROM [{table\_name}]")

            total\_records = cursor.fetchone()[0]

*# Get column names*

            cursor.execute("""

                SELECT COLUMN\_NAME FROM INFORMATION\_SCHEMA.COLUMNS

                WHERE TABLE\_NAME = ? ORDER BY ORDINAL\_POSITION

            """, (table\_name,))

            columns = [row[0] for row in cursor.fetchall()]

*# Get data with pagination*

            cursor.execute(f"""

                SELECT \* FROM [{table\_name}]

                ORDER BY (SELECT NULL)

                OFFSET ? ROWS

                FETCH NEXT ? ROWS ONLY

            """, (offset, limit))

            rows = cursor.fetchall()

*# Convert rows to dictionaries*

            data = []

            for row in rows:

                row\_dict = {}

                for i, col in enumerate(columns):

                    row\_dict[col] = row[i]

                data.append(row\_dict)

*# Get company distribution*

            company\_col = None

            for col in columns:

                if 'company' in col.lower() and 'id' in col.lower():

                    company\_col = col

                    break

            company\_stats = {}

            if company\_col:

                cursor.execute(f"""

                    SELECT [{company\_col}], COUNT(\*) as count

                    FROM [{table\_name}]

                    GROUP BY [{company\_col}]

                    ORDER BY [{company\_col}]

                """)

                company\_stats = {str(row[0]): row[1] for row in cursor.fetchall()}

            return {

                "table\_exists": True,

                "table\_name": table\_name,

                "total\_records": total\_records,

                "returned\_records": len(data),

                "offset": offset,

                "limit": limit,

                "columns": columns,

                "company\_distribution": company\_stats,

                "data": data,

                "pagination": {

                    "has\_more": (offset + limit) < total\_records,

                    "next\_offset": offset + limit if (offset + limit) < total\_records else None,

                    "total\_pages": (total\_records + limit - 1) // limit

                }

            }

    except Exception as e:

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

if \_\_name\_\_ == "\_\_main\_\_":

    import uvicorn

    uvicorn.run(app, host="0.0.0.0", port=8000)