"""Connect model with mcp tools in Python

Run this python script

> pip install mcp azure-ai-inference

> python <this-script-path>.py

"""

Import Section

"""

Required libraries for API interactions, data handling, and authentication

"""

```
import os

import logging

import asyncio

import pandas as pd

from typing import Dict, List, Optional

from ytmusicapi import YTMusic

import unicodedata

from dotenv import load_dotenv

from tenacity import retry, stop_after_attempt, wait_exponential

from aiohttp import ClientError

from youtube_auth import setup_oauth_with_account_selection, setup_headers_auth,
setup_cookie_auth

from datetime import datetime

import csv

import shutil


# Load environment variables
```

```
load_dotenv()

# Enhanced logging configuration
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(name)s - %(message)s',
    handlers=[
        logging.FileHandler('playlist_transfer.log'),
        logging.StreamHandler()
    ]
)
logger = logging.getLogger(__name__)

class RateLimiter:
    def __init__(self, calls_per_second: int = 2):
        self.calls_per_second = calls_per_second
        self.minimum_interval = 1.0 / calls_per_second
        self.last_call_time = 0

    async def wait(self):
        current_time = asyncio.get_event_loop().time()
        time_since_last_call = current_time - self.last_call_time
        if time_since_last_call < self.minimum_interval:
            await asyncio.sleep(self.minimum_interval - time_since_last_call)
        self.last_call_time = current_time
```

```
class YouTubeMusicHandler:
    def __init__(self):
        """Initialize without auth file - will be set during setup"""
        self.ytmusic = None
        self.rate_limiter = RateLimiter(
            int(os.getenv('RATE_LIMIT_CALLS_PER_SECOND', 2))
        )

    async def setup_auth(self):
        """Setup authentication using the enhanced auth system"""
        print("\n🎵 Choose YouTube Music Authentication Method:")
        print("1. OAuth (Recommended)")
        print("2. Browser Headers")
        print("3. Cookie-based")

        choice = input("\nEnter your choice (1-3): ")

        try:
            if choice == "1":
                self.ytmusic = setup_oauth_with_account_selection()
            elif choice == "2":
                self.ytmusic = setup_headers_auth()
            elif choice == "3":
                self.ytmusic = setup_cookie_auth()
            else:
                raise ValueError("Invalid authentication choice")
```

```
        if not self.ytmusic:
            raise ValueError("Authentication failed")

        logger.info("Authentication successful!")
        return True

    except Exception as e:
        logger.error(f"Authentication failed: {e}")
        return False

def _sanitize_text(self, text: str) -> str:
    try:
        if not text:
            return ""
        # Handle various encodings
        if isinstance(text, bytes):
            text = text.decode('utf-8', errors='ignore')

        # Normalize unicode characters
        normalized = unicodedata.normalize('NFKC', str(text))

        # Remove control characters but keep spaces and special characters
        sanitized = ''.join(char for char in normalized
                            if char.isprintable() or char.isspace())
```

```
        # Handle common special characters in music titles
        special_chars = {'–': '-', '—': '-', ''': "'", ''': "'", '"': '"', '"': '"'}
        for old, new in special_chars.items():
            sanitized = sanitized.replace(old, new)

        return sanitized.strip()
    except Exception as e:
        logger.error(f"Text sanitization failed: {e}")
        return str(text)

@retry(
    stop=stop_after_attempt(3),
    wait=wait_exponential(multiplier=1, min=4, max=10)
)
async def create_playlist(self, name: str, description: str = "") -> str:
    await self.rate_limiter.wait()
    try:
        sanitized_name = self._sanitize_text(name)
        sanitized_desc = self._sanitize_text(description)
        playlist_id = self.ytmusic.create_playlist(sanitized_name, sanitized_desc)
        logger.info(f"Created playlist: {sanitized_name}")
        return playlist_id
    except Exception as e:
        logger.error(f"Playlist creation failed: {e}")
        raise
```

```
@retry(
    stop=stop_after_attempt(3),
    wait=wait_exponential(multiplier=1, min=4, max=10)
)
async def search_song(self, query: str) -> Optional[List[Dict]]:
    await self.rate_limiter.wait()
    try:
        sanitized_query = self._sanitize_text(query)
        results = self.ytmusic.search(sanitized_query, filter="songs")
        return results[:3] if results else None
    except ClientError as e:
        logger.error(f"Network error during search: {e}")
        raise
    except Exception as e:
        logger.error(f"Search failed for '{query}': {e}")
        return None


@retry(
    stop=stop_after_attempt(3),
    wait=wait_exponential(multiplier=1, min=4, max=10)
)
async def add_to_playlist(self, playlist_id: str, video_id: str) -> bool:
    await self.rate_limiter.wait()
    try:
        self.ytmusic.add_playlist_items(playlist_id, [video_id])
        return True
```

```
        except Exception as e:
            logger.error(f"Failed to add song {video_id}: {e}")
            raise

class PlaylistTransfer:
    def __init__(self):
        self.yt_handler = YouTubeMusicHandler()

    async def setup(self):
        """Setup authentication through YouTube Music Handler"""
        return await self.yt_handler.setup_auth()

    async def verify_csv_structure(self, csv_path: str) -> bool:
        """Verify CSV has required column"""
        try:
            df = pd.read_csv(csv_path)
            required_column = 'Song'

            if required_column not in df.columns:
                logger.error(f"Missing required column: {required_column}")
                return False

            if df.empty:
                logger.error("CSV file is empty")
                return False
```

```
            return True

        except Exception as e:
            logger.error(f"Error verifying CSV structure: {e}")
            return False

    async def process_playlist(self, csv_path: str, playlist_name: str) -> Dict:
        """Process playlist and update CSV with transfer status"""
        results = {
            'total_songs': 0,
            'matched_songs': 0,
            'unmatched_songs': [],
            'errors': []
        }

        try:
            if not await self.verify_csv_structure(csv_path):
                raise ValueError("Invalid CSV structure")

            # Create backup of original CSV
            backup_path = f"{csv_path}.backup_{datetime.now().strftime('%Y%m%d_%H%M%S')}"
            shutil.copy2(csv_path, backup_path)
            logger.info(f"Created backup at: {backup_path}")

            try:
                df = pd.read_csv(csv_path, encoding='utf-8')
```

```
except UnicodeDecodeError:
    df = pd.read_csv(csv_path, encoding='latin-1')

# Add status columns if they don't exist
if 'Transfer_Status' not in df.columns:
    df['Transfer_Status'] = ''
if 'Transfer_Date' not in df.columns:
    df['Transfer_Date'] = ''
if 'YTMusic_URL' not in df.columns:
    df['YTMusic_URL'] = ''
if 'Error_Details' not in df.columns:
    df['Error_Details'] = ''

playlist_id = await self.yt_handler.create_playlist(
    playlist_name,
    "Transferred from Spotify"
)

total_songs = len(df)
logger.info(f"Processing {total_songs} songs...")

for index, row in df.iterrows():
    results['total_songs'] += 1
    progress = (index + 1) / total_songs * 100

    try:
```

```
query = self.yt_handler._sanitize_text(row['Song'])
logger.info(f"Processing ({progress:.1f}%): {query}")

matches = await self.yt_handler.search_song(query)

if matches:
    video_id = matches[0]['videoId']
    await self.yt_handler.add_to_playlist(playlist_id, video_id)

    # Update success status
    df.at[index, 'Transfer_Status'] = 'Success'
    df.at[index, 'Transfer_Date'] = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    df.at[index, 'YTMusic_URL'] = f"https://music.youtube.com/watch?v={video_id}"
    df.at[index, 'Error_Details'] = ''

    results['matched_songs'] += 1
else:
    # Update failure status
    df.at[index, 'Transfer_Status'] = 'Not Found'
    df.at[index, 'Transfer_Date'] = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    df.at[index, 'Error_Details'] = 'No matches found on YouTube Music'

    results['unmatched_songs'].append({
        'song': row['Song']
    })
```

```
            except Exception as e:
                # Update error status
                df.at[index, 'Transfer_Status'] = 'Error'
                df.at[index, 'Transfer_Date'] = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
                df.at[index, 'Error_Details'] = str(e)

                results['errors'].append({
                    'song': row['Song'],
                    'error': str(e)
                })
                logger.error(f"Error processing song: {e}")
                continue

            # Save progress after each song
            try:
                df.to_csv(csv_path, index=False, encoding='utf-8')
            except Exception as e:
                logger.error(f"Error saving progress to CSV: {e}")

        return results

    except Exception as e:
        logger.error(f"Transfer failed: {e}")
        raise

async def generate_summary_report(self, csv_path: str, results: Dict):
```

```
"""Generate a detailed HTML report of the transfer"""
try:
    report_path = f"transfer_report_{datetime.now().strftime('%Y%m%d_%H%M%S')}.html"
    df = pd.read_csv(csv_path)

    html_content = f"""
    <html>
      <head>
        <title>Playlist Transfer Report</title>
        <style>
          body {{ font-family: Arial, sans-serif; margin: 20px; }}
          .success {{ color: green; }}
          .error {{ color: red; }}
          .not-found {{ color: orange; }}
        </style>
      </head>
      <body>
        <h1>Playlist Transfer Report</h1>
        <p>Total songs: {results['total_songs']}</p>
        <p>Successfully transferred: {results['matched_songs']}</p>
        <p>Failed: {len(results['unmatched_songs']) + len(results['errors'])}</p>
        {df.to_html(classes='table', escape=False)}
      </body>
    </html>
    """
```

```
            with open(report_path, 'w', encoding='utf-8') as f:
                f.write(html_content)

            logger.info(f"Generated report at: {report_path}")

        except Exception as e:
            logger.error(f"Error generating report: {e}")

# Modify the main function to include report generation
async def main():
    try:
        # Initialize transfer
        transfer = PlaylistTransfer()

        # Setup authentication
        if not await transfer.setup():
            logger.error("Authentication failed. Exiting.")
            return

        # Get CSV path and playlist name
        csv_path = input("\nEnter path to Spotify playlist CSV file: ").strip()
        playlist_name = input("Enter name for the new YouTube Music playlist: ").strip()

        # Process playlist
        results = await transfer.process_playlist(csv_path, playlist_name)
```

```
        # Generate detailed report
        await transfer.generate_summary_report(csv_path, results)

        # Print detailed results
        print("\n✨ Transfer Complete! ✨")
        print(f"Total songs processed: {results['total_songs']}")
        print(f"Successfully matched: {results['matched_songs']}")
        print(f"Unmatched songs: {len(results['unmatched_songs'])}")

        if results['unmatched_songs']:
            print("\n📝 Unmatched songs for manual review:")
            for song in results['unmatched_songs']:
                print(f"• {song['song']}")

        if results['errors']:
            print("\n⚠️ Errors occurred:")
            for error in results['errors']:
                print(f"• {error['song']}: {error['error']}")

    except Exception as e:
        logger.error(f"Error: {e}")

if __name__ == "__main__":
    asyncio.run(main())
```