Flask + SQLite File Upload App — Explained Tutorial

This tutorial walks you step by step through building a Back-end Flask application with SQLite that lets users upload text files, store metadata, and search or view them. Each code snippet is explained with reasoning about how and why it works.

1. Project Setup

We need a project structure that separates frontend (HTML templates, CSS) from backend (Flask and database). We also have an 'uploads' folder where user files will be saved.

```
my_flask_app/
III app.py              # Main Flask backend
III people.db           # SQLite database
III uploads/            # Uploaded text files
III templates/          # HTML templates
IIII base.html
IIII index.html
IIII view_file.html
III static/
IIII style.css       # CSS styling
```

1A. Top of script

```
import sqlite3
import os
import getpass
from datetime import datetime
from flask import Flask, render_template, request, redirect, url_for, flash, abort

app = Flask(__name__)
app.secret_key = "supersecretkey"
UPLOAD_FOLDER = "uploads"
ALLOWED_EXTENSIONS = {"txt"}

os.makedirs(UPLOAD_FOLDER, exist_ok=True)
app.config["UPLOAD_FOLDER"] = UPLOAD_FOLDER
```

2. Database Initialization

We need a SQLite database to store uploaded file information. This includes the filename, the username of the uploader, the time of upload, a user message, and the genre.

```
 1: def init_db():
 2:     conn = sqlite3.connect("people.db")
 3:     c = conn.cursor()
 4:     c.execute('''CREATE TABLE IF NOT EXISTS uploads ( 5:
id INTEGER PRIMARY KEY,
 6:                     filename TEXT,
 7:                     username TEXT,
 8:                     upload_time TEXT,
 9:                     message TEXT,
 10:                     genre TEXT)''')
 11:     conn.commit()
 12:     conn.close()
```

3. Restricting Allowed File Types

We only want to allow `.txt` files for safety and simplicity.

```
 1: def allowed_file(filename):
 2:     return "." in filename and filename.rsplit(".", 1)[1].lower() in ALLOWED_EXTENSIONS
```

4. Handling File Uploads

The main route `/` handles both displaying the upload form (GET) and processing uploads (POST). When a file is uploaded: 1. We check that it exists and is `.txt`. 2. Save it into the `uploads/` folder. 3. Collect metadata: username, timestamp, message, genre. 4. Insert metadata into SQLite. 5. Flash a success message.

```
 1: @app.route("/", methods=["GET", "POST"])
 2: def index():
 3:     if request.method == "POST" and "file" in request.files: 4:
file = request.files["file"]
 5:         message = request.form.get("message", "").strip()
 6:         genre = request.form.get("genre", "Uncategorized")
 7:
 8:         if file.filename == "":
 9:             flash("No selected file")
 10:             return redirect(request.url)
 11:
 12:         if file and allowed_file(file.filename):
 13:             filepath = os.path.join(app.config["UPLOAD_FOLDER"],
file.filename)
 14:             file.save(filepath)
 15:
 16:             username = getpass.getuser()
 17:             upload_time = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
 18:
 19:             conn = sqlite3.connect("people.db")
 20:             c = conn.cursor()
 21:             c.execute("INSERT INTO uploads (filename, username,
upload_time, message, genre)VALUES (?,?,?,?,?)”,
 22:             (file.filename, username, upload_time, message, genre))
 23:             conn.commit()
 24:             conn.close()
 25:
 26:             flash(f"File '{file.filename}' uploaded by {username} at {upload_time}")
 27:         else:
 28:             flash("Invalid file type! Only .txt files are allowed.")
 29:         return redirect(url_for("index"))
```

5. Searching and Filtering Files

Users can search by filename/message or filter by genre and uploader. We dynamically build the SQL query depending on the search form.

```
search_query = request.args.get("search", "").strip()
genre_filter = request.args.get("genre_filter", "")
user_filter = request.args.get("user_filter", "")

conn = sqlite3.connect("people.db")
c = conn.cursor()
query = "SELECT filename, username, upload_time, message, genre FROM uploads WHERE 1=1"
 params = []

if search_query:
    query += " AND (filename LIKE ? OR message LIKE ?)"
    params.extend([f"%{search_query}%", f"%{search_query}%"])
if genre_filter:
    query += " AND genre=?"
    params.append(genre_filter)
if user_filter:
    query += " AND username=?"
    params.append(user_filter)

query += " ORDER BY id DESC"
c.execute(query, params)
uploads = c.fetchall()

# Distinct genres and usernames for filter dropdowns
c.execute("SELECT DISTINCT genre FROM uploads")
genres = [row[0] for row in c.fetchall() if row[0]]
c.execute("SELECT DISTINCT username FROM uploads")
users = [row[0] for row in c.fetchall() if row[0]]
conn.close()
return render_template("index.html", uploads=uploads, genres=genres, users=users)
```

6. Viewing a File

The `/view/` route loads the text content and its metadata. If the file doesn't exist, it returns a 404 error.

```
@app.route("/view/<filename>")
def view_file(filename):
    filepath = os.path.join(app.config["UPLOAD_FOLDER"], filename)
    if not os.path.isfile(filepath):
        abort(404, description="File not found")

    with open(filepath, "r", encoding="utf-8") as f:
        content = f.read()

    conn = sqlite3.connect("people.db")
    c = conn.cursor()
    c.execute("SELECT username, upload_time, message, genre FROM uploads WHERE filename=?",
(filename,))
    row = c.fetchone()
    conn.close()

    if row:
        username, upload_time, message, genre = row
    else:
        username, upload_time, message, genre = ("Unknown", "Unknown", "", "Uncategorized")

    return render_template("view_file.html", filename=filename, content=content,
                           username=username, upload_time=upload_time, message=message, genre=genre)

if __name__ == "__main__":
    init_db()
    app.run(debug=True)
```

I Summary

This project covered: - Flask for routing and web handling. - SQLite for storing metadata. - File upload handling and validation. – Now read the front end pdf to see how its handled.