Backend Development for Spotify Clone

Here’s a complete list of **backend files and folders** you should create for your **Spotify clone backend in Python (FastAPI)** — structured for scalability, clean code, and future extensions like recommendation systems.

✅ High-Level Folder Structure (Your Backend Project Tree)

bash

```bash
spotify_clone_backend/
├── app/
│   ├── main.py            # Entry point
│   ├── config.py          # Load environment variables & settings
│   ├── database.py          # DB connection setup
│   ├── models/            # SQLAlchemy ORM models (tables)
│   │   ├── user.py
│   │   ├── song.py
│   │   ├── artist.py
│   │   ├── album.py
│   │   ├── playlist.py
│   │   ├── like.py
│   │   ├── recent.py
│   ├── schemas/            # Pydantic models (request/response validation)
│   │   ├── user.py
│   │   ├── song.py
│   │   ├── artist.py
│   │   ├── album.py
│   │   ├── playlist.py
│   │   ├── auth.py
│   ├── routes/            # API routers
│   │   ├── auth.py
│   │   ├── user.py
│   │   ├── song.py
│   │   ├── artist.py
│   │   ├── album.py
│   │   ├── playlist.py
│   │   ├── recommendation.py
│   ├── controllers/        # Logic handling
│   │   ├── auth_controller.py
│   │   ├── song_controller.py
│   │   ├── playlist_controller.py
│   ├── services/           # ML / recommendation engine
│   │   ├── content_based.py
│   │   ├── collaborative_filtering.py
│   ├── utils/              # Helper functions
│   │   ├── hashing.py
│   │   ├── jwt_handler.py
│   │   ├── streaming.py
│   ├── middlewares/        # JWT auth middleware, CORS, etc.
│   │   ├── auth.py
│
├── media/                  # Uploaded audio files (or S3 if cloud)
├── .env                    # Secrets, DB URL, JWT secret
├── requirements.txt
├── README.md
```

📂 Folder-wise Breakdown

🔸 `main.py`

- Runs the FastAPI app.
- Includes CORS setup, route registration, DB startup, etc.

🔸 `config.py`

- Loads settings from `.env` (e.g., database URL, secret key)
- Uses `pydantic.BaseSettings`

🔸 `database.py`

- Creates and manages SQLAlchemy engine + session
- Can include Alembic config for migrations (optional)

📦 `models/` (Database Tables)

- `user.py`: id, name, email, hashed_password, subscription
- `song.py`: title, artist_id, album_id, genre, duration, file_path
- `artist.py`: name, bio, image
- `album.py`: title, artist_id, cover_image
- `playlist.py`: name, user_id, list of songs (many-to-many)
- `like.py`: user_id, song_id
- `recent.py`: user_id, song_id, timestamp

📘 `schemas/` (Pydantic for validation)

- `auth.py`: `LoginRequest`, `TokenData`
- `user.py`: `UserCreate`, `UserResponse`
- `song.py`: `SongUpload`, `SongOut`
- `playlist.py`: `PlaylistCreate`, `PlaylistDetail`, etc.

🚏 `routes/` (APIs)

Each file defines a group of related endpoints.

Examples:

- `auth.py` → `/login`, `/register`
- `song.py` → `/songs`, `/songs/upload`, `/songs/{id}/stream`
- `playlist.py` → `/playlists`, `/playlists/{id}/add`
- `recommendation.py` → `/recommend/songs`, etc.

🔧 `controllers/` (Business Logic)

- Connects route requests with DB and logic
- Avoids bloating route files

🤖 `services/` (Recommendation System)

You'll use:

- `content_based.py`: TF-IDF + cosine similarity
- `collaborative_filtering.py`: SVD or implicit matrix factorization

This is where ML lives.

🔐 `utils/` and `middlewares/`

- `hashing.py`: Bcrypt password hash & verify
- `jwt_handler.py`: Encode/decode JWTs
- `streaming.py`: Byte-range logic for audio streaming
- `auth.py`: Dependency to protect private routes

🧪 (Optional) `tests/`

- Unit & integration tests for your APIs and logic using `pytest`.

🌐 `.env` (Environment Variables)

Example:

```ini
DATABASE_URL=postgresql://user:pass@localhost/spotify_clone
JWT_SECRET=your_super_secret_key
```

```
AWS_ACCESS_KEY_ID=xxx
AWS_SECRET_ACCESS_KEY=xxx
```

Would you like me to:

- Generate **starter templates** for these files?
- Provide **audio streaming code in FastAPI**?
- Share **PostgreSQL schema** for models?

Let me know what part to build next.

Here's a clear breakdown of **what parts (features) you need to build in the backend** of your **Spotify clone using Python**, and **how to build each part step-by-step** — including tools, file locations, and core logic.

🎯 COMPLETE BACKEND FEATURE CHECKLIST (Spotify Clone)

Part #	Feature	Description	How to Build (Key Steps)
1️⃣	User Authentication (JWT)	Signup, login, token auth	Use FastAPI + `bcrypt` + `python-jose` JWT
2️⃣	User Profile	View/update profile	Protected routes using `Depends(get_current_user)`
3️⃣	Song Upload	Upload songs (MP3) with metadata	File upload API + store in `/media` folder or AWS S3
4️⃣	Audio Streaming	Stream songs with seek support	Implement byte-range support with `StreamingResponse`
5️⃣	Song Search	Search songs by name/genre/artist	Use SQL `LIKE` queries or full-text search

Part #	Feature	Description	How to Build (Key Steps)
6️⃣	Playlist Management	Create, update, delete playlists	CRUD API + song-playlist many-to-many relation
7️⃣	Recently Played	Store user's latest plays	Track play event with timestamps
8️⃣	Like/Unlike Songs	Like songs and fetch liked list	Store `user_id` + `song_id` mapping
9️⃣	Artist & Album Management	Browse artists and albums	Fetch & filter data via artist/album tables
🔟	Recommendation System	Suggest songs based on behavior/content	ML logic: TF-IDF or Collaborative Filtering
🔁	Audio + Cover Upload Handling	Upload audio files + cover images	Use `UploadFile`, validate type, store locally or S3
⚙️	Admin Panel (optional)	Admin can upload songs, artists, albums	Use `is_admin` flag in user model
🚀	Deployment	Host API & media	Use Render/Railway + PostgreSQL or SQLite for local

---

✅ STEP-BY-STEP — HOW TO BUILD EACH PART

---

1. 🧑‍💻 User Authentication

Files:

- `routes/auth.py`, `controllers/auth_controller.py`
- `schemas/auth.py`, `models/user.py`, `utils/jwt_handler.py`

Steps:

- `POST /register` → Create user (hash password)

- `POST /login` → Return JWT token
- Protect routes using:

  ```python
  Depends(get_current_user)
  ```

---

2. 🙋 User Profile

Files: `routes/user.py`, `schemas/user.py`, `models/user.py`

Steps:

- `GET /me` → get current user info
- `PUT /me` → update profile

---

3. 🎵 Song Upload

Files: `routes/song.py`, `controllers/song_controller.py`, `models/song.py`

Steps:

- `POST /songs/upload`:
  - Accept file with metadata
  - Store file in `/media/` or AWS S3
  - Save metadata in DB

---

4. 📡 Audio Streaming

Files: `utils/streaming.py`, `routes/song.py`

Steps:

- `GET /songs/{id}/stream`
- Read file with `open()` and return using `StreamingResponse`

- Implement byte-range support for seek/scrub

---

5. 🔍 Song Search

Files: `routes/song.py`, `controllers/song_controller.py`

Steps:

- `GET /songs/search?q=`
- Use `ilike()` (PostgreSQL) or regex filter
- Return matched songs

---

6. 📂 Playlist Management

Files: `routes/playlist.py`, `models/playlist.py`, `schemas/playlist.py`

Steps:

- `POST /playlists` – create
- `PUT /playlists/{id}/add` – add song
- `DELETE /playlists/{id}/remove` – remove song
- `GET /playlists` – fetch user's playlists

---

7. ⏱️ Recently Played

Files: `models/recent.py`, `routes/song.py`

Steps:

- On song stream:
  - Add entry to recent table: `user_id`, `song_id`, `timestamp`
- Fetch latest `N` played songs

---

8. ❤️ Like/Unlike Songs

Files: `routes/like.py`, `models/like.py`

Steps:

- `POST /songs/{id}/like` – add to like
- `DELETE /songs/{id}/unlike`
- `GET /me/likes`

---

9. 🎤 Artist & Album Management

Files: `routes/artist.py`, `routes/album.py`, `models/artist.py`, `album.py`

Steps:

- `GET /artists`, `GET /albums`
- Create/update as admin
- Join with song table to fetch related data

🔟 🤖 Recommendation Engine

Files: `services/content_based.py`, `services/collaborative_filtering.py`

Steps (Basic TF-IDF):

- Combine genre, artist name, lyrics
- Use:

```python
from sklearn.feature_extraction.text import TfidfVectorizer
cosine_similarity(...)
```

- Recommend songs with highest similarity

🖼️ Audio/Cover Upload Handling

Files: `utils/upload.py`

Steps:

- Use FastAPI's `UploadFile`
- Validate content type: only MP3, JPG, PNG
- Store with unique filename

👑 Admin Panel (Optional)

Steps:

- Add `is_admin: bool` to User model
- Create routes that require admin check:

```python
if not current_user.is_admin:
    raise HTTPException(...)
```

🚀 Deployment

Tools: Render / Railway / Fly.io

Steps:

- Use Uvicorn for ASGI
- `.env` for secrets
- PostgreSQL or SQLite
- Frontend will consume JSON from your backend

✅ Summary of Main Files You Must Write

File	Purpose
`main.py`	App entry
`models/*.py`	DB schema
`schemas/*.py`	Request/response validation
`routes/*.py`	All APIs (auth, song, playlist, etc.)
`controllers/*.py`	Core logic of each route
`services/*.py`	ML/Recommendation logic
`utils/*.py`	Helpers (JWT, hash, file handling)
`media/`	Local file store

Would you like me to:

- ✅ Generate base code for one full module (e.g., `songs/`)?
- ✅ Provide working `StreamingResponse` code?
- ✅ Build a basic recommender code block?

Just tell me what part you want to implement first.