

MegaPi

Welcome to MegaPi, the dynamic backend API powering the music sim web application. Engineered to revolutionize the way users explore music, MegaPi delivers personalized song recommendations with unparalleled accuracy. At its core, MegaPi harnesses the power of the Milvus vector database and sophisticated single feature vectors, crafted using an Essentia CNN specifically trained for music genre classification. Coupled with the robust MinIO object storage system, MegaPi provides a seamless and innovative solution for navigating through an extensive music library, ensuring every user experience is both unique and engaging.

Runs in a Docker container

```
docker compose up -d
```

```
docker ps --format 'table
{{.ID}}\t{{.Image}}\t{{.Command}}\t{{.Status}}\t{{.Names}}'
```

NAMES	PORTS
megapi-megapi-1	8000:8000
megapi-minio-1	9000:9001
megapi-postgre-1	5432:5432

Project layout

```
mkdocs.yml    # The configuration file.
docs/
  index.md    # The documentation homepage.
  ...         # Other markdown pages, images and other files.
```

```
.
├── app.py
├── docker-compose.yml
├── Dockerfile
├── requirements.txt
├── mkdocs.yml
├── README.md
├──
├── core
│   ├── data
│   │   └── mtg_jamendo_genre.json
```

```

├── music.db
├── __init__.py
├── database.py
├── extract_openl3_embeddings.py
├── config.py
├── extract_openl3_embeddings.py
├── docs
│   ├── endpoints
│   ├── index.md
│   └── services
├── gui
│   └── templates
│       └── index.html
├── models
│   ├── favorites.py
│   ├── __init__.py
│   ├── milvus.py
│   ├── minio.py
│   ├── music.py
│   ├── openl3.py
│   ├── spotinote.py
│   ├── uploaded.py
│   └── users.py
├── routes
│   ├── auth.py
│   ├── favorites.py
│   ├── __init__.py
│   ├── lyrics.py
│   ├── milvus.py
│   ├── minio.py
│   ├── monitoring.py
│   ├── music.py
│   ├── openl3.py
│   ├── spotinote.py
│   └── uploaded.py
├── services
│   ├── auth.py
│   ├── favorites.py
│   ├── __init__.py
│   ├── lyrics.py
│   ├── milvus.py
│   ├── minio.py
│   ├── monitoring.py
│   ├── openl3.py
│   ├── spotinote.py
│   └── uploaded.py
├── site
│   └── ...
├── tests
│   └── __init__.py

```

```
|— test_auth.py  
|— test_files.py  
|— test_milvus.py  
|— test_minio.py
```

MkDocs Commands

- `mkdocs new [dir-name]` - Create a new project.
- `mkdocs serve` - Start the live-reloading docs server.
- `mkdocs build` - Build the documentation site.
- `mkdocs -h` - Print help message and exit.

Documentation for `routes/auth.py`

This module contains the endpoints for the authentication service. It provides routes for user registration, login, and logout and other related operations.

```
delete_user(user_id,
current_user=Depends(login_manager),
db=Depends(get_db))
```

Delete a user by their user ID.

- **user_id:** int - The ID of the user to delete.
- **current_user:** User - The current authenticated user attempting the deletion.
- **db:** Session - The database session dependency.
- **return:** Returns a dictionary with a detail message on successful deletion.

Source code in `routes/auth.py`

```

87 @router.delete("/users/{user_id}", tags=["users"], response_model=dict)
88 def delete_user(
89     user_id: int, current_user=Depends(login_manager), db: Session =
90     Depends(get_db)
91 ):
92     """
93     Delete a user by their user ID.
94
95     - **user_id**: int - The ID of the user to delete.
96     - **current_user**: User - The current authenticated user attempting
97     the deletion.
98     - **db**: Session - The database session dependency.
99     - **return**: Returns a dictionary with a detail message on successful
100     deletion.
101     """
102     if not current_user:
103         raise InvalidCredentialsException(detail="Invalid credentials")
104     if current_user.id != 1:
105         raise HTTPException(status_code=401, detail="Unauthorized")
106     user = db.query(User).filter(User.id == user_id).first()
107     if not user:
108         raise HTTPException(status_code=404, detail="User not found")
109     db.delete(user)
110     db.commit()
111     return {"detail": "User deleted"}
```

index()

Render a front-end GUI for testing signup/login functionality.

- **return:** Returns an HTMLResponse containing the content of the index.html page.

Source code in routes/auth.py

```

53 @router.get("/gui", tags=["auth gui"], response_class=HTMLResponse)
54 def index():
55     """
56     Render a front-end GUI for testing signup/login functionality.
57
58     - **return**: Returns an HTMLResponse containing the content of the
59     index.html page.
60     """
61     file_path = os.path.join("gui", "templates", "index.html")
62     with open(file_path, "r") as f:
63         return HTMLResponse(content=f.read())

```

list_users(user=Depends(login_manager), db=Depends(get_db))

List all users.

- **user:** User - The current authenticated user (unused in this function).
- **db:** Session - The database session dependency.
- **return:** Returns a list of dictionaries, each representing a user with their id and email.

Source code in routes/auth.py

```

122 @router.get("/users", tags=["users"], response_model=list)
123 def list_users(user=Depends(login_manager), db: Session =
124 Depends(get_db)):
125     """
126     List all users.
127
128     - **user**: User - The current authenticated user (unused in this
129     function).
130     - **db**: Session - The database session dependency.
131     - **return**: Returns a list of dictionaries, each representing a user
132     with their id and email.
133     """
134     users = db.query(User).all()
135     users = [{"id": user.id, "email": user.email} for user in users]
136     return users

```

login(data=Depends())

Authenticate a user and return an access token.

- **data:** OAuth2PasswordRequestForm - A form data model including username (email) and password.
- **return:** Returns a TokenData object containing the access token and token type.

Source code in routes/auth.py

```

65 @router.post("/token", tags=["users"], response_model=TokenData)
66 def login(data: OAuth2PasswordRequestForm = Depends()):
67     """
68     Authenticate a user and return an access token.
69
70     - **data**: OAuth2PasswordRequestForm - A form data model including
71     username (email) and password.
72     - **return**: Returns a TokenData object containing the access token
73     and token type.
74     """
75     email = data.username
76     password = data.password
77     user = get_user(email)
78     if not user or not bcrypt.checkpw(
79         password.encode("utf-8"), user.hashed_password.encode("utf-8")
80     ):
81         raise InvalidCredentialsException
82     access_token_expires =
83     timedelta(minutes=DEFAULT_SETTINGS.access_token_expire_minutes)
84     access_token = login_manager.create_access_token(
85         data=dict(sub=email), expires=access_token_expires
86     )
87     return {"access_token": access_token, "token_type": "bearer"}

```

private_route(user=Depends(login_manager))

A private route that requires authentication.

- **user:** User - The current authenticated user.
- **return:** Returns a dictionary with a welcome message for the authenticated user.

Source code in routes/auth.py

```

111 @router.get("/private", tags=["users"], summary="A private route that
112 requires authentication.", response_model=dict)
113 def private_route(user=Depends(login_manager)):
114     """
115     A private route that requires authentication.
116
117     - **user**: User - The current authenticated user.
118     - **return**: Returns a dictionary with a welcome message for the
119     authenticated user.
120     """
121     return {"detail": f"Welcome {user.email}, you are authenticated"}

```

`read_users_me(user=Depends(login_manager))` `async`

Get the current authenticated user.

- **user**: User - The current authenticated user from the session.
- **return**: Returns the user object of the currently authenticated user.

Source code in routes/auth.py

```

21 @router.get("/users/me", tags=["users"])
22 async def read_users_me(user: User = Depends(login_manager)):
23     """
24     Get the current authenticated user.
25
26     - **user**: User - The current authenticated user from the session.
27     - **return**: Returns the user object of the currently authenticated
28     user.
29     """
30     return user

```

`register(user, db=Depends(get_db))`

Register a new user with the provided email and password.

- **user**: UserCreate - A user creation object containing the email and password.
- **db**: Session - The database session dependency.
- **return**: Returns a dictionary with a detail message on successful registration.

Source code in routes/auth.py

```
32 @router.post("/register", tags=["users"], response_model=dict)
33 def register(user: UserCreate, db: Session = Depends(get_db)):
34     """
35     Register a new user with the provided email and password.
36
37     - **user**: UserCreate - A user creation object containing the email
38     and password.
39     - **db**: Session - The database session dependency.
40     - **return**: Returns a dictionary with a detail message on successful
41     registration.
42     """
43     db_user = get_user(user.email)
44     if db_user:
45         raise HTTPException(
46             status_code=400, detail="A user with this email already exists"
47         )
48     hashed_password = hash_password(user.password)
49     db_user = User(email=user.email, hashed_password=hashed_password)
50     db.add(db_user)
    db.commit()
    return {"detail": "Successfully registered"}
```


Documentation for `routes/favorites.py`

This module contains the endpoints for the favorites service. It provides routes for adding, removing, and listing favorite songs for a user.

```
add_song_to_favorites(song,  
user=Depends(login_manager), db=Depends(get_db))  
async
```

Add a song to the authenticated user's list of favorites.

- **song**: SongPath - The path of the song to be added to favorites.
- **user**: User - The authenticated user who is adding the song to favorites.
- **db**: Session - The database session for querying and updating the database.
- **return**: Returns a message indicating the song was successfully added to favorites or if it was already in favorites.

Source code in routes/favorites.py

```

35 @router.post("/add", tags=["favorites"])
36 async def add_song_to_favorites(song: SongPath, user: User =
37 Depends(login_manager), db: Session = Depends(get_db)):
38     """
39     Add a song to the authenticated user's list of favorites.
40
41     - **song**: SongPath - The path of the song to be added to favorites.
42     - **user**: User - The authenticated user who is adding the song to
43     favorites.
44     - **db**: Session - The database session for querying and updating the
45     database.
46     - **return**: Returns a message indicating the song was successfully
47     added to favorites or if it was already in favorites.
48     """
49     user = db.merge(user)
50     db.refresh(user)
51
52     if len(user.favorites) >= 9:
53         # Remove the oldest song from the favorites
54         user.favorites.pop(0)
55
56     music_id = get_song_id_by_filepath(db, song.file_path)
57     if not music_id:
58         raise HTTPException(status_code=404, detail="Song not found")
59     music = db.query(MusicLibrary).get(music_id)
60
61     # Check if the song is already in the user's favorites
62     if music in user.favorites:
63         return {"message": "Song is already in favorites"}
64
65     user.favorites.append(music)
66     db.commit()
67     return {"message": "Song added to favorites"}

```

```

delete_song_from_favorites(song,
user=Depends(login_manager), db=Depends(get_db))
async

```

Remove a song from the authenticated user's list of favorites.

- **song**: SongPath - The path of the song to be removed from favorites.
- **user**: User - The authenticated user who is removing the song from favorites.
- **db**: Session - The database session for querying and updating the database.
- **return**: Returns a message indicating the song was successfully removed from favorites or if the song was not found in favorites.

Source code in `routes/favorites.py`

```

66 @router.delete("/delete", tags=["favorites"])
67 async def delete_song_from_favorites(song: SongPath, user: User =
68 Depends(login_manager), db: Session = Depends(get_db)):
69     """
70     Remove a song from the authenticated user's list of favorites.
71
72     - **song**: SongPath - The path of the song to be removed from
73     favorites.
74     - **user**: User - The authenticated user who is removing the song from
75     favorites.
76     - **db**: Session - The database session for querying and updating the
77     database.
78     - **return**: Returns a message indicating the song was successfully
79     removed from favorites or if the song was not found in favorites.
80     """
81     user = db.merge(user)
82     db.refresh(user)
83     music_id = get_song_id_by_filepath(db, song.file_path)
84     if not music_id:
85         raise HTTPException(status_code=404, detail="Song not found")
86     music = db.query(MusicLibrary).get(music_id)
87     for favorite in user.favorites:
88         if favorite.id == music.id:
89             user.favorites.remove(favorite)
90             db.commit()
91             return {"message": "Song removed from favorites"}
92     raise HTTPException(status_code=404, detail="Song not found in
93     favorites")

```

```

get_favorites(user=Depends(login_manager),
db=Depends(get_db)) async

```

Retrieve the list of favorite songs for the authenticated user.

- **user:** User - The authenticated user whose favorites are to be retrieved.
- **db:** Session - The database session for querying the database.
- **return:** Returns a list of the user's favorite songs.

Source code in `routes/favorites.py`

```

15 @router.get("/", tags=["favorites"])
16 async def get_favorites(user=Depends(login_manager), db: Session =
17 Depends(get_db)):
18     """
19     Retrieve the list of favorite songs for the authenticated user.
20
21     - **user**: User - The authenticated user whose favorites are to be
22     retrieved.
23     - **db**: Session - The database session for querying the database.
24     - **return**: Returns a list of the user's favorite songs.
25     """
26     # The merge() function is used to merge a detached object back into the
27     session.
28     # It returns a new instance that represents the existing row in the DB.
29     # This is necessary because the 'user' object might have been created
30     in a different session and we want to associate it with the current
31     session.
32     user = db.merge(user)
33
34     # The refresh() function is used to update the attributes of the 'user'
35     instance with the current data in the DB.
36     # This is necessary because the 'user' object might have stale data and
37     we want to ensure we're working with the most recent data.
38     db.refresh(user)
39     return user.favorites

```

Documentation for `routes/lyrics.py`

This module contains the endpoints for the lyrics service. It provides routes for searching for lyrics by song title and artist, and for retrieving lyrics using <https://api.lyrics.ovh>

```
get_random_row(user=Depends(login_manager),
db=Depends(get_db))
```

Fetches a random song from the music library along with its lyrics from the lyrics.ovh API.

- **user:** User - The authenticated user making the request.
- **db:** Session - The database session for querying the database.
- **return:** Returns a JSON object containing the song's ID, details, and lyrics.

” Source code in `routes/lyrics.py`

```
15 @router.get("/random-lyrics", tags=["lyrics"])
16 def get_random_row(user=Depends(login_manager), db: Session =
17 Depends(get_db)):
18     """
19     Fetches a random song from the music library along with its lyrics from
20     the lyrics.ovh API.
21
22     - **user**: User - The authenticated user making the request.
23     - **db**: Session - The database session for querying the database.
24     - **return**: Returns a JSON object containing the song's ID, details,
25     and lyrics.
26     """
27     with db:
28         row = db.query(MusicLibrary).order_by(func.random()).first()
29         if row is None:
30             raise HTTPException(status_code=404, detail="No songs found in
the library.")
31         lyrics = fetch_lyrics(row.artist, row.title)
32         return {"id": row.id, "row": row, "lyrics": lyrics}
```

```
get_random_row_and_lyrics_and_metadata(user=Depends(login_manager),
db=Depends(get_db))
```

Fetches a random song from the music library along with its lyrics and metadata including artwork.

- **user:** User - The authenticated user making the request.
- **db:** Session - The database session for querying the database.
- **return:** Returns a JSON object containing the song's ID, details, lyrics from the lyrics.ovh API, and artwork from the metadata.

Source code in `routes/lyrics.py`

```

32 @router.get("/random-lyrics-metadata", tags=["lyrics"])
33 def get_random_row_and_lyrics_and_metadata(user=Depends(login_manager), db:
34 Session = Depends(get_db)):
35     """
36     Fetches a random song from the music library along with its lyrics and
37     metadata including artwork.
38
39     - **user**: User - The authenticated user making the request.
40     - **db**: Session - The database session for querying the database.
41     - **return**: Returns a JSON object containing the song's ID, details,
42     lyrics from the lyrics.ovh API, and artwork from the metadata.
43     """
44     with db:
45         row = db.query(MusicLibrary).order_by(func.random()).first()
46         if row is None:
47             raise HTTPException(status_code=404, detail="No songs found in
the library.")
48         lyrics = fetch_lyrics(row.artist, row.title)
49         artwork = get_artwork("megasetbucket", row.filepath)
50         return {"id": row.id, "row": row, "lyrics": lyrics, "artwork":
artwork}

```

Documentation for `routes/milvus.py`

This module contains the endpoints for the Milvus service. It provides routes for performing queries such as similarity searches on the Milvus vector database.

`get_entity_by_id(id, user=Depends(login_manager))`

Retrieves the embedding vector of a specific entity by its ID.

- **id:** str - The unique identifier of the entity.
- **user:** User - The authenticated user making the request.
- **return:** EmbeddingResponse - The embedding vector of the entity.

Source code in `routes/milvus.py`

```

28 @router.get("/entity/{id}", response_model=EmbeddingResponse, tags=
29 ["milvus"])
30 def get_entity_by_id(id: str, user=Depends(login_manager)):
31     """
32     Retrieves the embedding vector of a specific entity by its ID.
33
34     - **id**: str - The unique identifier of the entity.
35     - **user**: User - The authenticated user making the request.
36     - **return**: EmbeddingResponse - The embedding vector of the entity.
37     """
38     collection_512 = get_milvus_512_collection()
39     entities = collection_512.query(expr=f"id in [{id}]", output_fields=
40 ["embedding"])
41     if not entities:
42         raise HTTPException(status_code=404, detail="Entity not found")
43
44     embedding = [float(x) for x in entities[0]["embedding"]]
45     return EmbeddingResponse(id=id, embedding=embedding)

```

`get_genres_plot(query, user=Depends(login_manager))`

`async`

Generates a plot of the top 5 genres for a given entity based on its file path.

- **query:** SongPath - The query containing the file path of the entity.
- **user:** User - The authenticated user making the request.

- **return:** A base64 encoded string of the plot image.

Source code in `routes/milvus.py`

```

130 @router.post("/plot_genres", tags=["milvus"])
131 async def get_genres_plot(query: SongPath, user=Depends(login_manager)):
132     """
133     Generates a plot of the top 5 genres for a given entity based on its
134     file path.
135
136     - **query**: SongPath - The query containing the file path of the
137     entity.
138     - **user**: User - The authenticated user making the request.
139     - **return**: A base64 encoded string of the plot image.
140     """
141     collection_87 = get_milvus_87_collection()
142     entity = collection_87.query(
143         expr=f"path == '{query.file_path}'",
144         output_fields=["predictions", "title", "artist"],
145         limit=1
146     )
147     if not entity:
148         raise HTTPException(status_code=404, detail="Entity not found")
149
150     class_names, top_5_activations, title, artist = await
151     extract_plot_data(entity)
152     fig = await create_plot(class_names, top_5_activations, title, artist)
153     image_base64 = await convert_plot_to_base64(fig)
154
155     return Response(content=image_base64, media_type="text/plain")

```

`get_similar_9_entities_by_path(query,`
`user=Depends(login_manager))`

Retrieves the 9 most similar entities (by title, artist, album) based on the file path of an entity.

- **query:** FilePathsQuery - The query containing the file path(s) of the entity.
- **user:** User - The authenticated user making the request.
- **return:** A list of the 9 most similar entities with short details.

Source code in routes/milvus.py

```

102 @router.post("/similar_short_entity", tags=["milvus"],
103 response_model=SimilarShortEntitiesResponse)
104 def get_similar_9_entities_by_path(query: FilePathsQuery,
105 user=Depends(login_manager)):
106     """
107     Retrieves the 9 most similar entities (by title, artist, album) based
108     on the file path of an entity.
109
110     - **query**: FilePathsQuery - The query containing the file path(s) of
111     the entity.
112     - **user**: User - The authenticated user making the request.
113     - **return**: A list of the 9 most similar entities with short
114     details.
115     """
116     collection_512 = get_milvus_512_collection()
117     entities = collection_512.query(expr=f"path in {query.path}",
118 output_fields=["embedding"])
119     if not entities:
120         raise HTTPException(status_code=404, detail="Entity not found")
121
122     embeddings = [[float(x) for x in entity["embedding"]] for entity in
123 entities]
124     entities = collection_512.search(
125         data=embeddings,
126         anns_field="embedding",
127         param={"nprobe": 16},
128         limit=30,
129         offset=1,
130         output_fields=["title", "album", "artist", "path"],
131     )
132
133     sorted_entities = sort_entities(entities)
134     return {"entities": sorted_entities}

```

`get_similar_entities(id, user=Depends(login_manager))`

Retrieves the top 3 most similar entities to a given entity ID.

- **id**: str - The unique identifier of the entity to compare.
- **user**: User - The authenticated user making the request.
- **return**: SimilarFullEntitiesResponse - A list of the most similar entities.

Source code in routes/milvus.py

```

46 @router.get("/similar/{id}", tags=["milvus"],
47 response_model=SimilarFullEntitiesResponse)
48 def get_similar_entities(id: str, user=Depends(login_manager)):
49     """
50     Retrieves the top 3 most similar entities to a given entity ID.
51
52     - **id**: str - The unique identifier of the entity to compare.
53     - **user**: User - The authenticated user making the request.
54     - **return**: SimilarFullEntitiesResponse - A list of the most similar
55     entities.
56     """
57     collection_512 = get_milvus_512_collection()
58     entities = collection_512.query(expr=f"id in [{id}]", output_fields=
59     ["embedding"])
60     if not entities:
61         raise HTTPException(status_code=404, detail="Entity not found")
62
63     embedding = [float(x) for x in entities[0]["embedding"]]
64     entities = collection_512.search(
65         data=[embedding],
66         anns_field="embedding",
67         param={"nprobe": 16},
68         limit=3,
69         offset=1,
70         output_fields=["*"],
71     )
72
73     response_list = [full_hit_to_dict(hit) for hit in entities[0]]
74     return SimilarFullEntitiesResponse(hits=response_list)

```

get_similar_entities_by_path(query,
user=Depends(login_manager))

Retrieves the top 3 most similar entities based on the file path of an entity.

- **query:** FilePathsQuery - The query containing the file path(s) of the entity.
- **user:** User - The authenticated user making the request.
- **return:** SimilarFullEntitiesResponse - A list of the most similar entities with full details.

Source code in `routes/milvus.py`

```

74 @router.post("/similar_full_entity", tags=["milvus"],
75 response_model=SimilarFullEntitiesResponse)
76 def get_similar_entities_by_path(query: FilePathsQuery,
77 user=Depends(login_manager)):
78     """
79     Retrieves the top 3 most similar entities based on the file path of an
80     entity.
81
82     - **query**: FilePathsQuery - The query containing the file path(s) of
83     the entity.
84     - **user**: User - The authenticated user making the request.
85     - **return**: SimilarFullEntitiesResponse - A list of the most similar
86     entities with full details.
87     """
88     collection_512 = get_milvus_512_collection()
89     entities = collection_512.query(expr=f"path in {query.path}",
90 output_fields=["embedding"])
91     if not entities:
92         raise HTTPException(status_code=404, detail="Entity not found")
93
94     embeddings = [[float(x) for x in entity["embedding"]] for entity in
95 entities]
96     entities = collection_512.search(
97         data=embeddings,
98         anns_field="embedding",
99         param={"nprobe": 16},
100        limit=3,
101        offset=1,
102        output_fields=["*"],
103    )
104
105     response_list = [short_hit_to_dict(hit) for hit in entities[0]]
106     return SimilarFullEntitiesResponse(hits=response_list)

```

ping_milvus_collection()

Checks the connectivity with the Milvus vector database. Mostly used to make prometheus ping milvus everyday, so milvus doesn't get idle for 7 days and shutdown.

- **return:** The status of the Milvus service.

” Source code in routes/milvus.py

```
155 @router.get("/ping", tags=["milvus"])
156 def ping_milvus_collection():
157     """
158     Checks the connectivity with the Milvus vector database. Mostly used
159     to make prometheus ping milvus everyday, so milvus doesn't get idle for 7
160     days and shutdown.
161
162     - **return**: The status of the Milvus service.
163     """
164     milvus_status = ping_milvus()
165     return milvus_status
```

Documentation for `routes/minio.py`

This module contains the endpoints for the MiniO service. It provides endpoints for storing and retrieving objects from MiniO buckets.

`download_file(query, user=Depends(login_manager))`

`async`

Downloads a song file from MinIO storage.

- **query:** SongPath - The path to the song file in MinIO storage.
- **user:** User - The authenticated user making the request.
- **return:** StreamingResponse - A streaming response for downloading the song file.

” Source code in `routes/minio.py`

```

79 @router.post("/download-song/", tags=["MinIO"])
80 async def download_file(query: SongPath, user=Depends(login_manager)):
81     """
82     Downloads a song file from MinIO storage.
83
84     - **query**: SongPath - The path to the song file in MinIO storage.
85     - **user**: User - The authenticated user making the request.
86     - **return**: StreamingResponse - A streaming response for downloading
87     the song file.
88     """
89     try:
90         data = minio_client.get_object(DEFAULT_SETTINGS.minio_bucket_name,
91         query.file_path)
92         filename = query.file_path.split('/')[-1] # Get the filename from
93         the file_path
94         headers = {
95             "Content-Disposition": f"attachment; filename={filename}",
96         }
97         return StreamingResponse(data.stream(32*1024),
98         media_type="audio/mpeg", headers=headers)
99     except Exception as e:
100         raise HTTPException(status_code=404, detail="File not found")

```

`get_file(query, user=Depends(login_manager))` `async`

Streams a song file from MinIO storage.

- **query:** SongPath - The path to the song file in MinIO storage.
- **user:** User - The authenticated user making the request.
- **return:** StreamingResponse - A streaming response of the song file.

” Source code in `routes/minio.py`

```

63 @router.post("/stream-song/", tags=["MinIO"])
64 async def get_file(query: SongPath, user=Depends(login_manager)):
65     """
66     Streams a song file from MinIO storage.
67
68     - **query**: SongPath - The path to the song file in MinIO storage.
69     - **user**: User - The authenticated user making the request.
70     - **return**: StreamingResponse - A streaming response of the song
71     file.
72     """
73     try:
74         data = minio_client.get_object(DEFAULT_SETTINGS.minio_bucket_name,
75         query.file_path)
76         return StreamingResponse(data.stream(32*1024),
77         media_type="audio/mpeg")
78     except Exception as e:
79         raise HTTPException(status_code=404, detail="File not found")

```

```

get_random_song_metadata(user=Depends(login_manager),
db=Depends(get_db)) async

```

Retrieves metadata for a random song from MinIO storage using the music-tag library.

- **user:** User - The authenticated user making the request.
- **db:** Session - Database session dependency.
- **return:** JSONResponse - The metadata of a random song.

Source code in `routes/minio.py`

```

115 @router.get("/random-metadata", tags=["MinIO"])
116 async def get_random_song_metadata(user=Depends(login_manager), db:
117     Session = Depends(get_db)):
118     """
119     Retrieves metadata for a random song from MinIO storage using the
120     music-tag library.
121
122     - **user**: User - The authenticated user making the request.
123     - **db**: Session - Database session dependency.
124     - **return**: JSONResponse - The metadata of a random song.
125     """
126     try:
127         count = db.query(MusicLibrary).count()
128         random_id = randint(1, count)
129         row = db.query(MusicLibrary).filter(MusicLibrary.id ==
130 random_id).first()
131         metadata =
132 get_metadata_and_artwork(DEFAULT_SETTINGS.minio_bucket_name, row.filepath)
133         return JSONResponse(content=metadata)
    except Exception as e:
        raise HTTPException(status_code=400, detail=str(e))
    finally:
        db.close()

```

`get_song_metadata(query, user=Depends(login_manager))`

`async`

Retrieves metadata for a specified song from MinIO storage using the music-tag library.

- **query**: SongPath - The path to the song file in MinIO storage.
- **user**: User - The authenticated user making the request.
- **return**: JSONResponse - The metadata of the specified song.

Source code in `routes/minio.py`

```

99 @router.post("/metadata", tags=["MinIO"])
100 async def get_song_metadata(query: SongPath, user=Depends(login_manager)):
101     """
102     Retrieves metadata for a specified song from MinIO storage using the
103     music-tag library.
104
105     - **query**: SongPath - The path to the song file in MinIO storage.
106     - **user**: User - The authenticated user making the request.
107     - **return**: JSONResponse - The metadata of the specified song.
108     """
109     try:
110         metadata =
111         get_metadata_and_artwork(DEFAULT_SETTINGS.minio_bucket_name,
112         query.file_path)
113         return JSONResponse(content=metadata)
114     except Exception as e:
115         raise HTTPException(status_code=400, detail=str(e))

```

```

list_objects_in_album_folder(query,
user=Depends(login_manager))

```

Retrieves a list of objects within a specified album folder in the MinIO bucket.

- **query:** AlbumResponse - The album folder to list objects from.
- **user:** User - The authenticated user making the request.
- **return:** List[S3Object] - A list of objects found in the specified album folder.

Source code in `routes/minio.py`

```

20 @router.post("/list-objects/", response_model=List[S3Object], tags=
21     ["MinIO"])
22 def list_objects_in_album_folder(query: AlbumResponse,
23     user=Depends(login_manager)):
24     """
25     Retrieves a list of objects within a specified album folder in the
26     MinIO bucket.
27
28     - **query**: AlbumResponse - The album folder to list objects from.
29     - **user**: User - The authenticated user making the request.
30     - **return**: List[S3Object] - A list of objects found in the specified
31     album folder.
32     """
33     objects = minio_client.list_objects(
34         DEFAULT_SETTINGS.minio_bucket_name,
35         prefix=query.album_folder,
36         recursive=True)
37
38     response = []
39     for obj in objects:
40         s3_object = {
41             "name": obj.object_name,
42             "size": obj.size,
43             "etag": obj.etag,
44             "last_modified": obj.last_modified.isoformat()
45         }
46         response.append(s3_object)
47
48     return response

```

`list_uploaded_objects(user=Depends(login_manager),
db=Depends(get_db))`

Lists objects uploaded by the authenticated user.

- **user**: User - The authenticated user making the request.
- **db**: Session - Database session dependency.
- **return**: UploadMP3ResponseList - A list of uploaded objects by the user.

Source code in routes/minio.py

```

47 @router.post("/list-uploaded-objects",
48 response_model=UploadMP3ResponseList, tags=["MinIO"])
49 def list_uploaded_objects(user=Depends(login_manager), db: Session =
50 Depends(get_db)):
51     """
52     Lists objects uploaded by the authenticated user.
53
54     - **user**: User - The authenticated user making the request.
55     - **db**: Session - Database session dependency.
56     - **return**: UploadMP3ResponseList - A list of uploaded objects by the
57     user.
58     """
59     objects =
60     minio_client.list_objects(DEFAULT_SETTINGS.minio_temp_bucket_name)
        # Adjusting the response to match the expected structure
        uploads = [UploadDetail(filename=obj.object_name) for obj in objects]
        response = UploadMP3ResponseList(uploads=uploads)
        return response

```

```

upload_file(file=File(...),
user=Depends(login_manager), db=Depends(get_db))
async

```

Uploads a MP3 file to MinIO storage using a temporary bucket.

- **file**: UploadFile - The MP3 file to upload.
- **user**: User - The authenticated user making the request.
- **db**: Session - Database session dependency.
- **return**: UploadMP3ResponseList - A list of uploaded MP3 files by the user.

Source code in routes/minio.py

```

136 @router.post("/upload-temp", tags=["MinIO"],
137 response_model=UploadMP3ResponseList)
138 async def upload_file(file: UploadFile = File(...),
139 user=Depends(login_manager), db: Session = Depends(get_db)):
140     """
141     Uploads a MP3 file to MinIO storage using a temporary bucket.
142
143     - **file**: UploadFile - The MP3 file to upload.
144     - **user**: User - The authenticated user making the request.
145     - **db**: Session - Database session dependency.
146     - **return**: UploadMP3ResponseList - A list of uploaded MP3 files by
147     the user.
148     """
149     try: # Check content type and extension
150         if file.content_type != "audio/mpeg":
151             raise HTTPException(status_code=400, detail="Only MP3 files
152 are allowed.")
153         _, file_extension = os.path.splitext(file.filename)
154         if file_extension.lower() != ".mp3":
155             raise HTTPException(status_code=400, detail="The uploaded file
156 is not an MP3 file.")
157
158         # Generate a secure filename
159         secure_filename = sanitize_filename(file.filename)
160
161         # Determine the size of the uploaded file by moving the cursor to
162         the end to get the file size
163         file.file.seek(0, os.SEEK_END)
164         file_size = file.file.tell()
165         file.file.seek(0)
166
167         # Stream the file directly to MinIO
168         minio_client.put_object(
169             bucket_name=DEFAULT_SETTINGS.minio_temp_bucket_name,
170             object_name=secure_filename,
171             data=file.file,
172             length=file_size,
173             content_type=file.content_type
174         )
175
176         # Store upload information in the database and return the updated
177         list of uploaded songs by the user
178         # song_path_in_minio = f"
179         {DEFAULT_SETTINGS.minio_temp_bucket_name}/{secure_filename}"
180         store_upload_info(db, user.id, secure_filename)
181         uploaded_songs = get_user_uploads(db, user.id)
182
183         return UploadMP3ResponseList(uploads=uploaded_songs)
184     except Exception as e:
185         raise HTTPException(status_code=500, detail=f"An unexpected error
186 occurred. {str(e)}")

```

Documentation for `routes/monitoring.py`

This module contains endpoint to retrives various system statistics on a Linux machine, including CPU temperature, CPU usage, memory usage, and disk usage.

`get_all_pi(user=Depends(login_manager))` `async`

Retrieves comprehensive monitoring statistics for a Linux host machine.

These statistics include CPU usage, memory usage, disk space, temperature readings.

- **user:** User - The authenticated user making the request, verified through the `login_manager`.
- **return:** A JSON response containing the linux host's monitoring statistics.

” Source code in `routes/monitoring.py`

```
9 @router.get("/pi", tags=["monitoring"])
10 async def get_all_pi(user=Depends(login_manager)):
11     """
12     Retrieves comprehensive monitoring statistics for a Linux host machine.
13
14     These statistics include CPU usage, memory usage, disk space,
15     temperature readings.
16
17     - **user**: User - The authenticated user making the request, verified
18     through the `login_manager`.
19     - **return**: A JSON response containing the linux host's monitoring
    statistics.
    """
    return get_all_pi_stats()
```


Documentation for `routes/music.py`

This module contains the endpoints for operations on the music database. It provides routes for searching for music by song title and artist.

```
add_row(query, user=Depends(login_manager),  
db=Depends(get_db))
```

Adds a new song to the `music_library` table.

- **Parameters:**
 - **query:** AddSongToMusicLibrary object containing the song details to be added.
 - **user:** User object, automatically provided by the `login_manager` dependency.
- **Returns:** A message indicating successful addition of the song.

Source code in `routes/music.py`

```

69 @router.post("/add", tags=["songs"])
70 def add_row(query: AddSongToMusicLibrary, user=Depends(login_manager), db:
71     Session = Depends(get_db)):
72     """
73     Adds a new song to the music_library table.
74
75     - **Parameters**:
76       - **query**: AddSongToMusicLibrary object containing the song
77       details to be added.
78       - **user**: User object, automatically provided by the
79       login_manager dependency.
80       - **Returns**: A message indicating successful addition of the song.
81     """
82     try:
83         max_id = db.query(func.max(MusicLibrary.id)).scalar() # Get the
84         maximum id from the music_library table
85         if max_id is None: max_id = 0 # If the table is empty, set max_id
86         to 0
87
88         # insert into the table
89         stmt = insert(MusicLibrary).values(
90             id=max_id + 1, # Set the id to one more than the current
91             maximum
92             filename=query.filename, filepath=query.filepath,
93             album_folder=query.album_folder,
94             artist_folder=query.artist_folder, filesize=query.filesize,
95             title=query.title,
96             artist=query.artist, album=query.album, year=query.year,
97             tracknumber=query.tracknumber,
98             genre=query.genre, top_5_genres=query.top_5_genres,
99         )
100         db.execute(stmt)
101         db.commit()
102         return {"message": "Row added successfully"}
103     finally:
104         db.close()

```

`count_rows(db=Depends(get_db))`

Returns the total number of rows in the music_library table.

- **Parameters:** None
- **Returns:** An integer representing the total number of rows in the music_library table.

Source code in `routes/music.py`

```

16 @router.get("/count", tags=["songs"])
17 def count_rows(db: Session = Depends(get_db)):
18     """
19     Returns the total number of rows in the music_library table.
20
21     - **Parameters**: None
22     - **Returns**: An integer representing the total number of rows in the
23     music_library table.
24     """
25     try:
26         result = db.execute(text("SELECT COUNT(*) FROM music_library"))
27         count = result.scalar()
28         return count
29     finally:
30         db.close()

```

```

delete_row(id, user=Depends(login_manager),
db=Depends(get_db))

```

Deletes a specific song from the music_library table by its ID.

- **Parameters:**
 - **id:** Integer, the ID of the song to delete.
 - **user:** User object, automatically provided by the login_manager dependency.
- **Returns:** A message indicating successful deletion of the song. Raises a 404 HTTPException if the song is not found.

Source code in routes/music.py

```

98 @router.delete("/delete/{id}", tags=["songs"])
99 def delete_row(id: int, user=Depends(login_manager), db: Session =
100 Depends(get_db)):
101     """
102     Deletes a specific song from the music_library table by its ID.
103
104     - **Parameters**:
105         - **id**: Integer, the ID of the song to delete.
106         - **user**: User object, automatically provided by the
107         login_manager dependency.
108     - **Returns**: A message indicating successful deletion of the song.
109     Raises a 404 HTTPException if the song is not found.
110     """
111     try:
112         row = db.query(MusicLibrary).get(id)
113         if row is None:
114             raise HTTPException(status_code=404, detail="Row not found")
115         db.delete(row)
116         db.commit()
117         return {"message": "Row deleted successfully"}
118     finally:
119         db.close()

```

```

get_album_folder_by_artist_and_album(query,
user=Depends(login_manager), db=Depends(get_db))

```

Retrieves the album folder for a specific artist and album combination in the music_library table.

- **Parameters:**
 - **query:** ArtistAlbumResponse object containing the artist's name and album title.
 - **user:** User object, automatically provided by the login_manager dependency.
- **Returns:** A dictionary containing the album folder name. Raises a 404 HTTPException if the album is not found.

Source code in `routes/music.py`

```

222 @router.post("/album_folder_by_artist_and_album", tags=["songs"])
223 def get_album_folder_by_artist_and_album(
224     query: ArtistAlbumResponse, user=Depends(login_manager), db: Session =
225     Depends(get_db)
226 ):
227     """
228     Retrieves the album folder for a specific artist and album combination
229     in the music_library table.
230
231     - **Parameters**:
232       - **query**: ArtistAlbumResponse object containing the artist's
233       name and album title.
234       - **user**: User object, automatically provided by the
235       login_manager dependency.
236       - **Returns**: A dictionary containing the album folder name. Raises a
237       404 HTTPException if the album is not found.
238     """
239     artist = query.artist
240     album = query.album
241     try:
242         row =
db.query(MusicLibrary.album_folder).filter(MusicLibrary.artist == artist,
MusicLibrary.album == album).first()
        if row is None:
            raise HTTPException(status_code=404, detail="Album not found")
        return {"album_folder": row.album_folder}
    finally:
        db.close()

```

```

get_random_row(user=Depends(login_manager),
db=Depends(get_db))

```

Retrieves a random song from the music_library table.

- **Parameters:**
 - **user:** User object, automatically provided by the login_manager dependency.
- **Returns:** A dictionary containing the ID of the randomly selected song and the song's row data.

Source code in `routes/music.py`

```

32 @router.get("/random", tags=["songs"])
33 def get_random_row(user=Depends(login_manager), db: Session =
34 Depends(get_db)):
35     """
36     Retrieves a random song from the music_library table.
37
38     - **Parameters**:
39       - **user**: User object, automatically provided by the
40       login_manager dependency.
41       - **Returns**: A dictionary containing the ID of the randomly selected
42       song and the song's row data.
43     """
44     try:
45         count = db.query(MusicLibrary).count()
46         random_id = randint(1, count)
47         row = db.query(MusicLibrary).filter(MusicLibrary.id ==
         random_id).first()
         return {"id": random_id, "row": row}
     finally:
         db.close()

```

`get_song_by_id(id, user=Depends(login_manager), db=Depends(get_db))`

Fetches a specific song from the music_library table by its ID.

- **Parameters:**
 - **id:** Integer, the ID of the song to retrieve.
 - **user:** User object, automatically provided by the login_manager dependency.
- **Returns:** A dictionary containing the ID of the song and the song's row data. Raises a 404 HTTPException if the song is not found.

Source code in routes/music.py

```

50 @router.get("/song/{id}", tags=["songs"])
51 def get_song_by_id(id: int, user=Depends(login_manager), db: Session =
52     Depends(get_db)):
53     """
54     Fetches a specific song from the music_library table by its ID.
55
56     - **Parameters**:
57         - **id**: Integer, the ID of the song to retrieve.
58         - **user**: User object, automatically provided by the
59         login_manager dependency.
60     - **Returns**: A dictionary containing the ID of the song and the
61     song's row data. Raises a 404 HTTPException if the song is not found.
62     """
63     try:
64         row = db.query(MusicLibrary).filter(MusicLibrary.id == id).first()
65         if row is None:
66             raise HTTPException(status_code=404, detail="Song not found")
67         return {"id": id, "row": row}
68     finally:
69         db.close()

```

```

list_all_albums(user=Depends(login_manager),
db=Depends(get_db))

```

Lists all albums in the music_library table, ordered by release date.

- **Parameters:** None
- **Returns:** A list of dictionaries, each containing the album name, album folder, and release year, ordered by release year.

Source code in `routes/music.py`

```

203 @router.get("/albums", tags=["songs"])
204 def list_all_albums(user=Depends(login_manager), db: Session =
205 Depends(get_db)):
206     """
207     Lists all albums in the music_library table, ordered by release date.
208
209     - **Parameters**: None
210     - **Returns**: A list of dictionaries, each containing the album name,
211     album folder, and release year, ordered by release year.
212     """
213     try:
214         query = (
215             db.query(MusicLibrary.album, MusicLibrary.album_folder,
216 MusicLibrary.year)
217             .distinct()
218             .order_by(MusicLibrary.year.asc())
219         )
220         return [{"album": row.album, "album_folder": row.album_folder} for
221 row in query.all()]
222     finally:
223         db.close()

```

```

list_all_albums_from_artist(artist_folder,
user=Depends(login_manager), db=Depends(get_db))

```

Lists all albums by a specific artist in the music_library table, ordered by release date.

- **Parameters:**
 - **artist_folder:** ArtistFolderResponse object containing the artist's folder name.
 - **user:** User object, automatically provided by the login_manager dependency.
- **Returns:** A list of album names for the given artist, ordered by release date.

Source code in `routes/music.py`

```

134 @router.post("/albums", tags=["songs"])
135 def list_all_albums_from_artist(artist_folder: ArtistFolderResponse,
136 user=Depends(login_manager), db: Session = Depends(get_db)):
137     """
138     Lists all albums by a specific artist in the music_library table,
139     ordered by release date.
140
141     - **Parameters**:
142       - **artist_folder**: ArtistFolderResponse object containing the
143       artist's folder name.
144       - **user**: User object, automatically provided by the
145       login_manager dependency.
146       - **Returns**: A list of album names for the given artist, ordered by
147       release date.
148     """
149     if artist_folder is None or artist_folder.artist_folder is None:
150         raise HTTPException(status_code=400, detail="Missing artist_folder
151 parameter")
152     try:
153         query = (
154             db.query(MusicLibrary.album)
155             .filter(MusicLibrary.artist_folder ==
156                   artist_folder.artist_folder)
157             .distinct()
158         )
159         return [row.album for row in query.all()]
160     finally:
161         db.close()

```

```
list_all_artists(user=Depends(login_manager),
db=Depends(get_db))
```

Lists all artists in the music_library table in alphabetical order.

- **Parameters:** None
- **Returns:** A list of artist names in alphabetical order.

Source code in `routes/music.py`

```

119 @router.get("/artists", tags=["songs"])
120 def list_all_artists(user=Depends(login_manager), db: Session =
121     Depends(get_db)):
122     """
123     Lists all artists in the music_library table in alphabetical order.
124
125     - **Parameters**: None
126     - **Returns**: A list of artist names in alphabetical order.
127     """
128     try:
129         query =
130     (db.query(MusicLibrary.artist_folder).distinct().order_by(MusicLibrary.artis
131         return [row.artist_folder for row in query.all()]
132     finally:
133         db.close()

```

```

list_all_songs_from_album(album_folder=None,
user=Depends(login_manager), db=Depends(get_db))

```

Lists all songs from a specific album in the music_library table.

- **Parameters:**
 - **album_folder:** AlbumResponse object containing the album's folder name.
 - **user:** User object, automatically provided by the login_manager dependency.
- **Returns:** A list of dictionaries, each containing the track number and title of a song from the specified album.

Source code in `routes/music.py`

```

157 @router.post("/songs", tags=["songs"])
158 def list_all_songs_from_album(album_folder: AlbumResponse = None,
159 user=Depends(login_manager), db: Session = Depends(get_db)):
160     """
161     Lists all songs from a specific album in the music_library table.
162
163     - **Parameters**:
164       - **album_folder**: AlbumResponse object containing the album's
165       folder name.
166       - **user**: User object, automatically provided by the
167       login_manager dependency.
168       - **Returns**: A list of dictionaries, each containing the track
169       number and title of a song from the specified album.
170     """
171     if album_folder is None or album_folder.album_folder is None:
172         raise HTTPException(status_code=400, detail="Missing album_folder
173 parameter")
174     try:
175         query = db.query(MusicLibrary).filter(MusicLibrary.album_folder ==
176 album_folder.album_folder)
177         return [
178             {"tracknumber": row.tracknumber, "title": row.title}
179             for row in
180 query.order_by(MusicLibrary.tracknumber.asc()).all()
181         ]
182     finally:
183         db.close()

```

```
list_all_songs_from_artist_and_album(query,
user=Depends(login_manager), db=Depends(get_db))
```

Lists all songs by a specific artist and from a specific album in the music_library table.

- **Parameters:**
 - **query:** ArtistAlbumResponse object containing the artist's name and album title.
 - **user:** User object, automatically provided by the login_manager dependency.
- **Returns:** A list of dictionaries, each containing the track number, file path, and title of a song from the specified artist and album.

Source code in routes/music.py

```

179 @router.post("/songs/by_artist_and_album", tags=["songs"])
180 def list_all_songs_from_artist_and_album(
181     query: ArtistAlbumResponse, user=Depends(login_manager), db: Session =
182     Depends(get_db)
183 ):
184     """
185     Lists all songs by a specific artist and from a specific album in the
186     music_library table.
187
188     - **Parameters**:
189       - **query**: ArtistAlbumResponse object containing the artist's
190       name and album title.
191       - **user**: User object, automatically provided by the
192       login_manager dependency.
193       - **Returns**: A list of dictionaries, each containing the track
194       number, file path, and title of a song from the specified artist and
195       album.
196     """
197     artist = query.artist
198     album = query.album
199     try:
200         query = db.query(MusicLibrary).filter(MusicLibrary.artist ==
artist, MusicLibrary.album == album)
        return [
            {"tracknumber": row.tracknumber, "path": row.filepath,
"title": row.title}
            for row in
query.order_by(MusicLibrary.tracknumber.asc()).all()
        ]
    finally:
        db.close()

```

Documentation for `routes/openl3.py`

This module contains the endpoints for the OpenL3 service. It provides routes for extracting audio embeddings using the OpenL3 model. The embeddings can then be used to perform similarity searches on the embeddings using the Milvus service.

```
get_embeddings(file_path,  
user=Depends(login_manager), db=Depends(get_db))
```

Retrieves the embeddings for a specified audio file.

This function loads a model from MinIO, retrieves the specified audio file as a temporary file, computes the embeddings using the loaded model, and then cleans up the temporary file. If successful, it returns an `EmbeddingResponse` object containing the file name and its embeddings. If the process fails, it raises an `HTTPException` with status code 500.

Parameters: - `file_path` (str): The path to the audio file for which embeddings are to be computed. - `user`: The current user object, automatically provided by the `login_manager` dependency. - `db`: The database session, automatically provided by the `get_db` dependency.

Returns: - `EmbeddingResponse`: An object containing the file name and its computed embeddings.

Source code in routes/openl3.py

```

15 @router.post("/embeddings/", response_model=EmbeddingResponse, tags=
16 ["OpenL3"])
17 def get_embeddings(file_path: str, user=Depends(login_manager), db: Session
18 = Depends(get_db)):
19     """
20     Retrieves the embeddings for a specified audio file.
21
22     This function loads a model from MinIO, retrieves the specified audio
23     file as a temporary file,
24     computes the embeddings using the loaded model, and then cleans up the
25     temporary file. If successful,
26     it returns an EmbeddingResponse object containing the file name and its
27     embeddings. If the process fails,
28     it raises an HTTPException with status code 500.
29
30     Parameters:
31     - file_path (str): The path to the audio file for which embeddings are
32     to be computed.
33     - user: The current user object, automatically provided by the
34     login_manager dependency.
35     - db: The database session, automatically provided by the get_db
36     dependency.
37
38     Returns:
39     - EmbeddingResponse: An object containing the file name and its
40     computed embeddings.
41
42     """
43     print(f"Starting to get embeddings for file: {file_path}")
44     try:
45         embedding_512_model = load_model_from_minio()
46         temp_file_path = get_temp_file_from_minio(file_path)
47
48         # Compute embeddings using the temporary file path
49         vector = embedding_512_model.compute(temp_file_path)
50         embedding = vector.mean(axis=0)
51
52         # Clean up the temporary file
53         os.unlink(temp_file_path)
54
55         print(f"Successfully processed embeddings for file: {file_path}")
56         return EmbeddingResponse(file_name=file_path,
57                                 embedding=embedding.tolist())
58     except Exception as e:
59         print(f"Failed to get embeddings for file: {file_path}. Error:
60 {e}")
61         raise HTTPException(status_code=500, detail=f"Failed to process the
62 request: {e}")

```

Documentation for `routes/spotinite.py`

This module contains the endpoints for what we call, the 'Spotinite service'. It provides routes for using the python spotipy library to interact with the Cyanite API. The Cyanite API is a music recommendation service. It uses the Spotify's ID to identify songs and can recommend songs based on a given artist or band name and a track title.

```
similar_tracks(query, user=Depends(login_manager),
               db=Depends(get_db)) async
```

Fetches and returns a list of tracks similar to the specified song and artist.

This endpoint takes a song title and artist as input, retrieves a Spotify ID for the song, and then fetches a list of similar tracks based on that ID. It aims to return 3 similar tracks that are not by the same artist as the input song, if possible. If not enough non-artist matches are found, it will include tracks by the same artist in the response.

Parameters: - query (SpotiniteQuery): The query object containing the title and artist of the song. - user: The current user object, automatically provided by the login_manager dependency. - db: The database session, automatically provided by the get_db dependency.

Returns: - List[SpotiniteResponse]: A list of similar tracks, each represented by a SpotiniteResponse object.

Source code in routes/spotinite.py

```

14 @router.post("/similar_tracks", response_model=List[SpotiniteResponse],
15 tags=["spotinite"])
16 async def similar_tracks(query: SpotiniteQuery,
17 user=Depends(login_manager), db: Session = Depends(get_db)):
18     """
19     Fetches and returns a list of tracks similar to the specified song and
20     artist.
21
22     This endpoint takes a song title and artist as input, retrieves a
23     Spotify ID for the song,
24     and then fetches a list of similar tracks based on that ID. It aims to
25     return 3 similar tracks
26     that are not by the same artist as the input song, if possible. If not
27     enough non-artist matches
28     are found, it will include tracks by the same artist in the response.
29
30     Parameters:
31     - query (SpotiniteQuery): The query object containing the title and
32     artist of the song.
33     - user: The current user object, automatically provided by the
34     login_manager dependency.
35     - db: The database session, automatically provided by the get_db
36     dependency.
37
38     Returns:
39     - List[SpotiniteResponse]: A list of similar tracks, each represented
40     by a SpotiniteResponse object.
41
42     """
43     try:
44         spotify_id = get_track_id(query.title, query.artist)
45         similar_track_ids = fetch_similar_tracks(spotify_id)
46     except Exception as e:
47         raise HTTPException(status_code=400, detail=str(e))
48
49     # Fetch 15 similar tracks and return the first 3 that are not by the
50     same artist if possible
51     similar_tracks = []
52     added_artists = set()
53     backup_tracks = []
54     for track_id in similar_track_ids:
55         track_info = get_track_info(track_id)
56         artist_lower = track_info['Artist'].lower()
57         if artist_lower != query.artist.lower() and artist_lower not in
added_artists:
            similar_tracks.append(track_info)
            added_artists.add(artist_lower)
        else:
            backup_tracks.append(track_info)
    if len(similar_tracks) == 3:
        break
    if len(similar_tracks) < 3:
        similar_tracks.extend(backup_tracks[:3-len(similar_tracks)])
    return similar_tracks

```

Documentation for `routes/uploaded.py`

This module provides endpoints to interact with a MiniO bucket for storing and retrieving user uploaded songs.

