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.yaml
├── 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
│   ├── spotinite.py
│   ├── uploaded.py
│   └── users.py
│
├── routes
│   ├── auth.py
│   ├── favorites.py
│   ├── __init__.py
│   ├── lyrics.py
│   ├── milvus.py
│   ├── minio.py
│   ├── monitoring.py
│   ├── music.py
│   ├── openl3.py
│   ├── spotinite.py
│   └── uploaded.py
│
├── services
│   ├── auth.py
│   ├── favorites.py
│   ├── __init__.py
│   ├── lyrics.py
│   ├── milvus.py
│   ├── minio.py
│   ├── monitoring.py
│   ├── openl3.py
│   ├── spotinite.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`

```
@router.delete("/users/{user_id}", tags=["users"], response_model=dict)
def delete_user(
    user_id: int, current_user=Depends(login_manager), db: Session =
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.
    """
    if not current_user:
        raise InvalidCredentialsException(detail="Invalid credentials")
    if current_user.id != 1:
        raise HTTPException(status_code=401, detail="Unauthorized")
    user = db.query(User).filter(User.id == user_id).first()
    if not user:
        raise HTTPException(status_code=404, detail="User not found")
    db.delete(user)
    db.commit()
    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`

```
@router.get("/gui", tags=["auth gui"], response_class=HTMLResponse)
def index():
    """
    Render a front-end GUI for testing signup/login functionality.

    - **return**: Returns an HTMLResponse containing the content of the
index.html page.
    """
    file_path = os.path.join("gui", "templates", "index.html")
    with open(file_path, "r") as f:
        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`

```
@router.get("/users", tags=["users"], response_model=list)
def list_users(user=Depends(login_manager), db: Session =
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.
    """
    users = db.query(User).all()
    users = [{"id": user.id, "email": user.email} for user in users]
    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`

```python
@router.post("/token", tags=["users"], response_model=TokenData)
def login(data: OAuth2PasswordRequestForm = 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.
    """
    email = data.username
    password = data.password
    user = get_user(email)
    if not user or not bcrypt.checkpw(
        password.encode("utf-8"), user.hashed_password.encode("utf-8")
    ):
        raise InvalidCredentialsException
    access_token_expires =
timedelta(minutes=DEFAULT_SETTINGS.access_token_expire_minutes)
    access_token = login_manager.create_access_token(
        data=dict(sub=email), expires=access_token_expires
    )
    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`

```python
@router.get("/private", tags=["users"], summary="A private route that
requires authentication.", response_model=dict)
def 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.
    """
    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`

```python
@router.get("/users/me", tags=["users"])
async def read_users_me(user: User = Depends(login_manager)):
    """
    Get the current authenticated user.

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

```python
@router.post("/register", tags=["users"], response_model=dict)
def register(user: UserCreate, db: Session = 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.
    """
    db_user = get_user(user.email)
    if db_user:
        raise HTTPException(
            status_code=400, detail="A user with this email already exists"
        )
    hashed_password = hash_password(user.password)
    db_user = User(email=user.email, hashed_password=hashed_password)
    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`

```python
@router.post("/add", tags=["favorites"])
async def add_song_to_favorites(song: SongPath, user: User =
Depends(login_manager), db: Session = Depends(get_db)):
    """
    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.
    """
    user = db.merge(user)
    db.refresh(user)

    if len(user.favorites) >= 9:
        # Remove the oldest song from the favorites
        user.favorites.pop(0)

    music_id = get_song_id_by_filepath(db, song.file_path)
    if not music_id:
        raise HTTPException(status_code=404, detail="Song not found")
    music = db.query(MusicLibrary).get(music_id)

    # Check if the song is already in the user's favorites
    if music in user.favorites:
        return {"message": "Song is already in favorites"}

    user.favorites.append(music)
    db.commit()
    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`

```python
@router.delete("/delete", tags=["favorites"])
async def delete_song_from_favorites(song: SongPath, user: User =
Depends(login_manager), db: Session = Depends(get_db)):
    """
    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.
    """
    user = db.merge(user)
    db.refresh(user)
    music_id = get_song_id_by_filepath(db, song.file_path)
    if not music_id:
        raise HTTPException(status_code=404, detail="Song not found")
    music = db.query(MusicLibrary).get(music_id)
    for favorite in user.favorites:
        if favorite.id == music.id:
            user.favorites.remove(favorite)
            db.commit()
            return {"message": "Song removed from favorites"}
    raise HTTPException(status_code=404, detail="Song not found in
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`

```python
@router.get("/", tags=["favorites"])
async def get_favorites(user=Depends(login_manager), db: Session =
Depends(get_db)):
    """
    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.
    """
    # The merge() function is used to merge a detached object back into the
session.
    # It returns a new instance that represents the existing row in the DB.
    # This is necessary because the 'user' object might have been created
in a different session and we want to associate it with the current
session.
    user = db.merge(user)

    # The refresh() function is used to update the attributes of the 'user'
instance with the current data in the DB.
    # This is necessary because the 'user' object might have stale data and
we want to ensure we're working with the most recent data.
    db.refresh(user)
    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`

```
@router.get("/random-lyrics", tags=["lyrics"])
def get_random_row(user=Depends(login_manager), db: Session =
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.
    """
    with db:
        row = db.query(MusicLibrary).order_by(func.random()).first()
        if row is None:
            raise HTTPException(status_code=404, detail="No songs found in
the library.")
        lyrics = fetch_lyrics(row.artist, row.title)
        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`

```
@router.get("/random-lyrics-metadata", tags=["lyrics"])
def get_random_row_and_lyrics_and_metadata(user=Depends(login_manager), db:
Session = 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.
    """
    with db:
        row = db.query(MusicLibrary).order_by(func.random()).first()
        if row is None:
            raise HTTPException(status_code=404, detail="No songs found in
the library.")
        lyrics = fetch_lyrics(row.artist, row.title)
        artwork = get_artwork("megasetbucket", row.filepath)
        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
        embedding = [float(x) for x in entities[0]["embedding"]]
        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`

```python
@router.post("/plot_genres", tags=["milvus"])
async def get_genres_plot(query: SongPath, user=Depends(login_manager)):
    """
    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.
    """
    collection_87 = get_milvus_87_collection()
    entity = collection_87.query(
        expr=f"path == '{query.file_path}'",
        output_fields=["predictions", "title", "artist"],
        limit=1
    )
    if not entity:
        raise HTTPException(status_code=404, detail="Entity not found")

    class_names, top_5_activations, title, artist = await
extract_plot_data(entity)
    fig = await create_plot(class_names, top_5_activations, title, artist)
    image_base64 = await convert_plot_to_base64(fig)

    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`

```python
@router.post("/similar_short_entity", tags=["milvus"],
response_model=SimilarShortEntitiesResponse)
def get_similar_9_entities_by_path(query: FilePathsQuery,
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.
    """
    collection_512 = get_milvus_512_collection()
    entities = collection_512.query(expr=f"path in {query.path}",
output_fields=["embedding"])
    if not entities:
        raise HTTPException(status_code=404, detail="Entity not found")

    embeddings = [[float(x) for x in entity["embedding"]] for entity in
entities]
    entities = collection_512.search(
        data=embeddings,
        anns_field="embedding",
        param={"nprobe": 16},
        limit=30,
        offset=1,
        output_fields=["title", "album", "artist", "path"],
    )

    sorted_entities = sort_entities(entities)
    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`

```python
@router.get("/similar/{id}", tags=["milvus"],
response_model=SimilarFullEntitiesResponse)
def get_similar_entities(id: str, 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.
    """
    collection_512 = get_milvus_512_collection()
    entities = collection_512.query(expr=f"id in [{id}]", output_fields=
["embedding"])
    if not entities:
        raise HTTPException(status_code=404, detail="Entity not found")

    embedding = [float(x) for x in entities[0]["embedding"]]
    entities = collection_512.search(
        data=[embedding],
        anns_field="embedding",
        param={"nprobe": 16},
        limit=3,
        offset=1,
        output_fields=["*"],
    )

    response_list = [full_hit_to_dict(hit) for hit in entities[0]]
    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`

```python
@router.post("/similar_full_entity", tags=["milvus"],
response_model=SimilarFullEntitiesResponse)
def get_similar_entities_by_path(query: FilePathsQuery,
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.
    """
    collection_512 = get_milvus_512_collection()
    entities = collection_512.query(expr=f"path in {query.path}",
output_fields=["embedding"])
    if not entities:
        raise HTTPException(status_code=404, detail="Entity not found")

    embeddings = [[float(x) for x in entity["embedding"]] for entity in
entities]
    entities = collection_512.search(
        data=embeddings,
        anns_field="embedding",
        param={"nprobe": 16},
        limit=3,
        offset=1,
        output_fields=["*"],
    )

    response_list = [short_hit_to_dict(hit) for hit in entities[0]]
    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`

```python
@router.get("/ping", tags=["milvus"])
def 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.
    """
    milvus_status = ping_milvus()
    return milvus_status
```

Documentation for `routes/minio.py`

This module contains the endpoints for the MinIO service. It provides endpoints for storing and retriveing 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`

```python
@router.post("/download-song/", tags=["MinIO"])
async def download_file(query: SongPath, user=Depends(login_manager)):
    """
    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.
    """
    try:
        data = minio_client.get_object(DEFAULT_SETTINGS.minio_bucket_name, query.file_path)
        filename = query.file_path.split('/')[-1]  # Get the filename from the file_path
        headers = {
            "Content-Disposition": f"attachment; filename={filename}",
        }
        return StreamingResponse(data.stream(32*1024), media_type="audio/mpeg", headers=headers)
    except Exception as e:
        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),
    media_type="audio/mpeg")
        except Exception as e:
            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`

```python
@router.get("/random-metadata", tags=["MinIO"])
async def get_random_song_metadata(user=Depends(login_manager), db:
Session = Depends(get_db)):
    """
    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.
    """
    try:
        count = db.query(MusicLibrary).count()
        random_id = randint(1, count)
        row = db.query(MusicLibrary).filter(MusicLibrary.id ==
random_id).first()
        metadata =
get_metadata_and_artwork(DEFAULT_SETTINGS.minio_bucket_name, row.filepath)
        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`

```python
@router.post("/metadata", tags=["MinIO"])
async def get_song_metadata(query: SongPath, user=Depends(login_manager)):
    """
    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.
    """
    try:
        metadata =
get_metadata_and_artwork(DEFAULT_SETTINGS.minio_bucket_name,
query.file_path)
        return JSONResponse(content=metadata)
    except Exception as e:
        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`

```python
@router.post("/list-objects/", response_model=List[S3Object], tags=
["MinIO"])
def list_objects_in_album_folder(query: AlbumResponse,
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.
    """
    objects = minio_client.list_objects(
        DEFAULT_SETTINGS.minio_bucket_name,
        prefix=query.album_folder,
        recursive=True)

    response = []
    for obj in objects:
        s3_object = {
            "name": obj.object_name,
            "size": obj.size,
            "etag": obj.etag,
            "last_modified": obj.last_modified.isoformat()
        }
        response.append(s3_object)

    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`

```python
@router.post("/list-uploaded-objects",
response_model=UploadMP3ResponseList, tags=["MinIO"])
def list_uploaded_objects(user=Depends(login_manager), db: Session =
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.
    """
    objects =
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`

```python
@router.post("/upload-temp", tags=["MinIO"],
response_model=UploadMP3ResponseList)
async def upload_file(file: UploadFile = File(...),
user=Depends(login_manager), db: Session = Depends(get_db)):
    """
    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.
    """
    try:   # Check content type and extension
        if file.content_type != "audio/mpeg":
            raise HTTPException(status_code=400, detail="Only MP3 files
are allowed.")
        _, file_extension = os.path.splitext(file.filename)
        if file_extension.lower() != ".mp3":
            raise HTTPException(status_code=400, detail="The uploaded file
is not an MP3 file.")

        # Generate a secure filename
        secure_filename = sanitize_filename(file.filename)

        # Determine the size of the uploaded file by moving the cursor to
the end to get the file size
        file.file.seek(0, os.SEEK_END)
        file_size = file.file.tell()
        file.file.seek(0)

        # Stream the file directly to MinIO
        minio_client.put_object(
            bucket_name=DEFAULT_SETTINGS.minio_temp_bucket_name,
            object_name=secure_filename,
            data=file.file,
            length=file_size,
            content_type=file.content_type
        )

        # Store upload information in the database and return the updated
list of uploaded songs by the user
        # song_path_in_minio = f"
{DEFAULT_SETTINGS.minio_temp_bucket_name}/{secure_filename}"
        store_upload_info(db, user.id, secure_filename)
        uploaded_songs = get_user_uploads(db, user.id)

        return UploadMP3ResponseList(uploads=uploaded_songs)
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"An unexpected error
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`

```
@router.get("/pi", tags=["monitoring"])
async def get_all_pi(user=Depends(login_manager)):
    """
    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.
    """
    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`

```python
@router.post("/add", tags=["songs"])
def add_row(query: AddSongToMusicLibrary, user=Depends(login_manager), db:
Session = 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.
    """
    try:
        max_id = db.query(func.max(MusicLibrary.id)).scalar()  # Get the
maximum id from the music_library table
        if max_id is None: max_id = 0  # If the table is empty, set max_id
to 0

        # insert into the table
        stmt = insert(MusicLibrary).values(
            id=max_id + 1,  # Set the id to one more than the current
maximum
            filename=query.filename, filepath=query.filepath,
album_folder=query.album_folder,
            artist_folder=query.artist_folder, filesize=query.filesize,
title=query.title,
            artist=query.artist, album=query.album, year=query.year,
tracknumber=query.tracknumber,
            genre=query.genre, top_5_genres=query.top_5_genres,
        )
        db.execute(stmt)
        db.commit()
        return {"message": "Row added successfully"}
    finally:
        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`

```python
@router.get("/count", tags=["songs"])
def count_rows(db: Session = 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.
    """
    try:
        result = db.execute(text("SELECT COUNT(*) FROM music_library"))
        count = result.scalar()
        return count
    finally:
        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`

```python
@router.delete("/delete/{id}", tags=["songs"])
def delete_row(id: int, user=Depends(login_manager), db: Session =
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.
    """
    try:
        row = db.query(MusicLibrary).get(id)
        if row is None:
            raise HTTPException(status_code=404, detail="Row not found")
        db.delete(row)
        db.commit()
        return {"message": "Row deleted successfully"}
    finally:
        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`

```python
@router.post("/album_folder_by_artist_and_album", tags=["songs"])
def get_album_folder_by_artist_and_album(
    query: ArtistAlbumResponse, user=Depends(login_manager), db: Session = 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.
    """
    artist = query.artist
    album = query.album
    try:
        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`

```python
@router.get("/random", tags=["songs"])
def get_random_row(user=Depends(login_manager), db: Session =
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.
    """
    try:
        count = db.query(MusicLibrary).count()
        random_id = randint(1, count)
        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`

```python
@router.get("/song/{id}", tags=["songs"])
def get_song_by_id(id: int, user=Depends(login_manager), db: Session =
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.
    """
    try:
        row = db.query(MusicLibrary).filter(MusicLibrary.id == id).first()
        if row is None:
            raise HTTPException(status_code=404, detail="Song not found")
        return {"id": id, "row": row}
    finally:
        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`

```
@router.get("/albums", tags=["songs"])
def list_all_albums(user=Depends(login_manager), db: Session =
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.
    """
    try:
        query = (
            db.query(MusicLibrary.album, MusicLibrary.album_folder,
MusicLibrary.year)
            .distinct()
            .order_by(MusicLibrary.year.asc())
        )
        return [{"album": row.album, "album_folder": row.album_folder} for
row in query.all()]
    finally:
        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`

```python
@router.post("/albums", tags=["songs"])
def list_all_albums_from_artist(artist_folder: ArtistFolderResponse,
user=Depends(login_manager), db: Session = 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.
    """
    if artist_folder is None or artist_folder.artist_folder is None:
        raise HTTPException(status_code=400, detail="Missing artist_folder
parameter")
    try:
        query = (
            db.query(MusicLibrary.album)
            .filter(MusicLibrary.artist_folder ==
artist_folder.artist_folder)
            .distinct()
        )
        return [row.album for row in query.all()]
    finally:
        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`

```
@router.get("/artists", tags=["songs"])
def list_all_artists(user=Depends(login_manager), db: Session =
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.
    """
    try:
        query =
(db.query(MusicLibrary.artist_folder).distinct().order_by(MusicLibrary.artis
        return [row.artist_folder for row in query.all()]
    finally:
        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`

```
@router.post("/songs", tags=["songs"])
def list_all_songs_from_album(album_folder: AlbumResponse = None,
user=Depends(login_manager), db: Session = 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.
    """
    if album_folder is None or album_folder.album_folder is None:
        raise HTTPException(status_code=400, detail="Missing album_folder
parameter")
    try:
        query = db.query(MusicLibrary).filter(MusicLibrary.album_folder ==
album_folder.album_folder)
        return [
            {"tracknumber": row.tracknumber, "title": row.title}
            for row in
query.order_by(MusicLibrary.tracknumber.asc()).all()
        ]
    finally:
        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`

```python
@router.post("/songs/by_artist_and_album", tags=["songs"])
def list_all_songs_from_artist_and_album(
    query: ArtistAlbumResponse, user=Depends(login_manager), db: Session = 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.
    """
    artist = query.artist
    album = query.album
    try:
        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`

```python
@router.post("/embeddings/", response_model=EmbeddingResponse, tags=
["OpenL3"])
def get_embeddings(file_path: str, user=Depends(login_manager), db: Session
= 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.

    """
    print(f"Starting to get embeddings for file: {file_path}")
    try:
        embedding_512_model = load_model_from_minio()
        temp_file_path = get_temp_file_from_minio(file_path)

        # Compute embeddings using the temporary file path
        vector = embedding_512_model.compute(temp_file_path)
        embedding = vector.mean(axis=0)

        # Clean up the temporary file
        os.unlink(temp_file_path)

        print(f"Successfully processed embeddings for file: {file_path}")
        return EmbeddingResponse(file_name=file_path,
embedding=embedding.tolist())
    except Exception as e:
        print(f"Failed to get embeddings for file: {file_path}. Error:
{e}")
        raise HTTPException(status_code=500, detail=f"Failed to process the
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`

```python
@router.post("/similar_tracks", response_model=List[SpotiniteResponse],
tags=["spotinite"])
async def similar_tracks(query: SpotiniteQuery,
user=Depends(login_manager), db: Session = Depends(get_db)):
    """
    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.

    """
    try:
        spotify_id = get_track_id(query.title, query.artist)
        similar_track_ids = fetch_similar_tracks(spotify_id)
    except Exception as e:
        raise HTTPException(status_code=400, detail=str(e))

    # Fetch 15 similar tracks and return the first 3 that are not by the
same artist if possible
    similar_tracks = []
    added_artists = set()
    backup_tracks = []
    for track_id in similar_track_ids:
        track_info = get_track_info(track_id)
        artist_lower = track_info['Artist'].lower()
        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.