

数据库 PJ 报告：音乐信息管理系统

姓名：石青芸

学号：22307140095

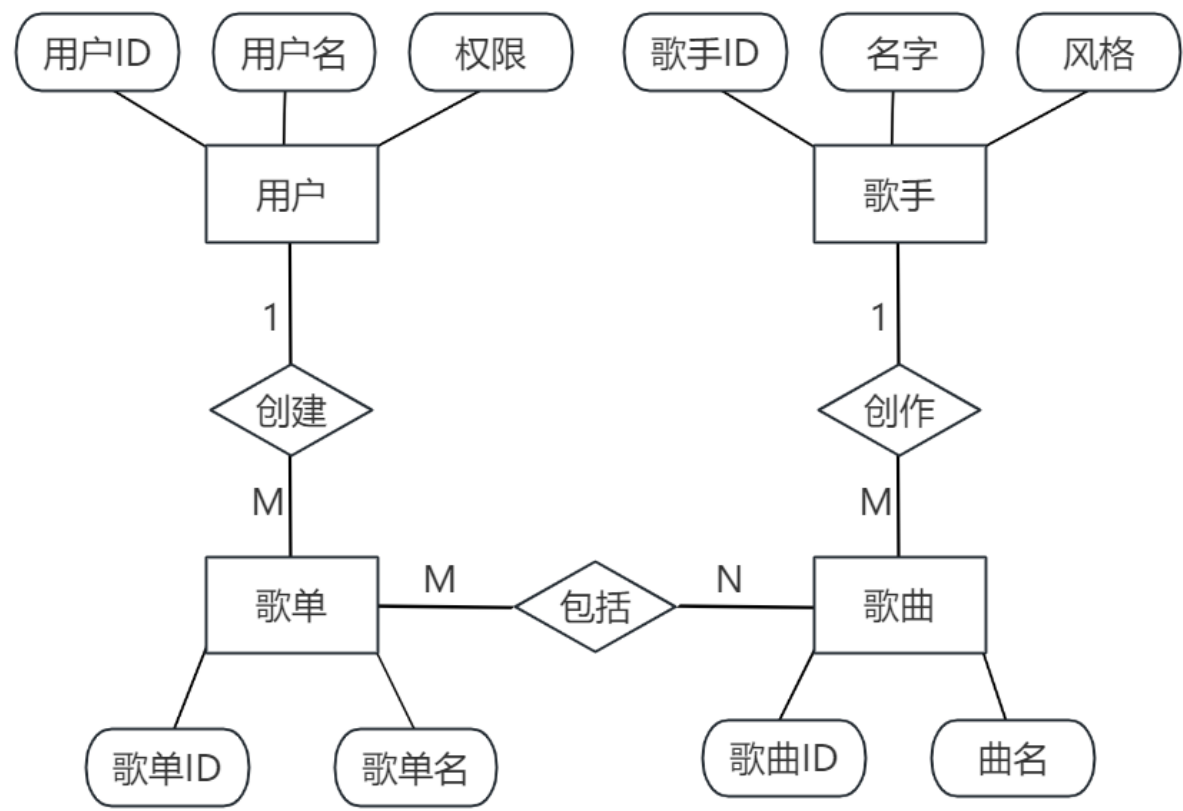
1. 需求分析

我的音乐信息数据库系统的设计目标是为用户提供一个直观、高效的音乐管理与查询平台，满足不同用户群体的需求。通过系统，用户不仅可以轻松管理自己的音乐收藏，还能够深入了解音乐资源的详细信息。管理员则可以利用系统工具对平台数据进行全面维护和优化，确保系统运行的流畅性和稳定性。

这个系统将解决以下核心需求，为音乐爱好者和平台维护者提供支持：

- 1. **用户需求**：普通用户希望能够轻松创建歌单，将自己喜爱的歌曲进行分类和保存，同时快速查找歌手或歌曲相关信息。
- 2. **平台管理需求**：管理员需要有效管理庞大的歌曲、歌手和用户数据，及时对新增或失效的音乐信息进行维护，确保系统中资源的准确性和完整性。
- 3. **高效的查询与操作**：用户需要一个友好的界面和高效的数据库查询机制，以快速获取所需信息，而无需经历繁琐的操作步骤。

系统的ER图如下：



2. 概要设计

为了满足需求，系统设计了多项核心功能模块，涵盖普通用户的操作需求和管理员的管理需求。以下是系统的主要功能模块及其说明：

2.1 用户模块

普通用户功能：

- 用户可以创建属于自己的歌单，例如“运动歌单”或“睡前歌单”，将不同风格的歌曲分类整理。
- 当某些由用户自己创建歌单不再需要时，用户可以删除这些歌单，从而保持界面的整洁。
- 系统支持用户通过歌手名、歌曲名、歌单名进行快速搜索，对公开的歌手、歌曲、歌单信息进行访问，帮助用户发现或定位所需的音乐资源。

管理员功能：

- 管理员可以维护系统中歌曲和歌手的信息，例如新增一首最新流行的歌曲，或者更新某位歌手的个人信息。
- 当发现用户提交的违规内容时，管理员可以对用户账户进行处理，或者调整其权限。

2.2 歌曲模块

- 系统为每一首歌曲提供了完整的属性记录，例如编号、名称、风格和作者（歌手）。
- 管理员可以对这些数据进行增、删、改操作，例如为某首歌更新新的风格标签、增加新的歌曲或下架不合规的歌曲相关信息等。

2.3 歌手模块

- 每位歌手都在系统中有其专属的档案，记录其创作的歌曲及其他相关信息。
- 管理员可以为新出道的歌手创建档案，也可以更新已有歌手的资料。

2.4 歌单模块

- 系统允许普通用户创建个性化的歌单，例如“旅行必听歌单”或“经典怀旧金曲”。
- 歌单中可以包含任意数量的歌曲，用户可以灵活添加或删除歌曲，打造专属的音乐收藏。

2.5 搜索功能

系统支持模糊查询和关键字搜索。例如，输入“周杰伦”，用户可以快速找到他演唱的所有歌曲，或者通过歌曲名称搜索指定曲目。

2.6 权限管理

系统根据用户角色的不同，分为普通用户和管理员两种权限模式。普通用户的操作范围仅限于个人内容管理，而管理员则可以执行系统级别的操作，例如数据维护和权限管理。

3. 详细设计

这个部分将会介绍系统的核心模块、功能的设计，主要分成用户（权限）管理、歌曲管理、歌手管理、歌单管理四个板块进行介绍。

3.1 用户（权限）管理

用户管理模块的主要职责是通过数据库模型存储用户信息，并实现用户的权限控制和身份验证功能。普通用户和管理员权限的差异通过字段 `is_admin` 控制，鉴权功能通过 `Depends(get_current_user)` 方法实现，确保操作安全性。用户登陆和验证的流程如下：

- 1. 用户通过登录接口提交用户名和密码。
- 2. 系统校验用户信息，生成 JWT Token 并返回给客户端。
- 3. 客户端在后续请求中附带 Token，服务器端通过 `get_current_user` 解码 Token 验证身份。
- 4. 根据用户权限返回相应的资源或执行操作。

3.1.1 数据库表设计

用户表 (`users`)

用户表的定义如下：

```
class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True, index=True) # 用户唯一标识
    username = Column(String, unique=True, nullable=False) # 用户名，必须唯一
    hashed_password = Column(String, nullable=False) # 哈希后的密码
    is_admin = Column(Boolean, default=False) # 是否为管理员，默认为普通用户

    playlists = relationship("Playlist", back_populates="owner") # 与歌单的外键关系
```

字段说明：

- `id`：主键，用于唯一标识用户。
- `username`：存储用户的登录名，设为唯一以防止重复注册。
- `hashed_password`：保存用户密码的哈希值，确保密码安全性。
- `is_admin`：标识用户权限等级（`True` 表示管理员，`False` 表示普通用户）。
- `playlists`：建立与歌单表的关系，用于查询用户的歌单信息。

3.1.2 鉴权功能实现

用户权限的鉴权通过 `Depends(get_current_user)` 方法实现，该方法依赖 JWT（JSON Web Token）进行用户身份验证。具体流程如下：

核心代码： `get_current_user` 函数

```
async def get_current_user(
    token: str = Depends(oauth2_scheme),
    session: AsyncSession = Depends(get_session),
):
    try:
        # 从客户端获取 JWT Token
        print(f"Received token: {token}")
```

```
# 解码 JWT Token, 获取负载信息
payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
print(f"Decoded payload: {payload}")

# 从负载中提取用户 ID
user_id = payload.get("sub")
if user_id is None:
    raise HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED, detail="Invalid token1"
    )

# 查询数据库, 获取用户信息
result = await session.execute(select(User).where(User.id ==
int(user_id)))
user = result.scalars().first()
if not user:
    raise HTTPException(
        status_code=status.HTTP_404_NOT_FOUND, detail="User not found"
    )

# 输出用户名 (调试用)
print(user.username)
return user

# 处理 Token 过期异常
except jwt.ExpiredSignatureError:
    raise HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED, detail="Token expired"
    )

# 处理 Token 无效异常
except jwt.InvalidTokenError as e:
    print(f"Token error: {e}")
    raise HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED, detail="Invalid token2"
    )
```

实现流程:

1. 获取 Token

- 从 `Depends(oauth2_scheme)` 中获取客户端传递的 JWT Token。
- 打印 Token 内容 (调试阶段使用)。

2. 解码 Token

- 使用预设的密钥和算法解码 Token, 提取负载 (payload) 信息。
- 从负载中获取 `sub` (用户 ID), 如果未找到, 则返回“无效 Token”错误。

3. 查询数据库

- 使用 SQLAlchemy 异步查询用户表, 通过 `user_id` 查找对应用户。
- 如果未找到用户, 则返回“用户未找到”错误。

4. 返回用户信息

- 成功找到用户后，返回用户对象，用于后续操作（如判断权限）。

5. 异常处理

- 捕获 Token 过期错误，返回相应的“Token 过期”提示。
- 捕获 Token 无效错误，返回相应的“Token 无效”提示。

3.1.3 权限控制

系统通过用户对象中的 `is_admin` 字段实现权限控制。例如：

- **普通用户**
普通用户只能访问与其歌单、个人信息相关的操作，权限范围较小。
- **管理员**
管理员的操作范围更广，可新增、修改和删除系统资源（如歌曲、歌手信息等）。在接口中可通过以下逻辑判断权限：

```
# 以添加歌曲的端点为例
@app.post("/admin/songs/")
async def add_song(song: SongCreate, current_user: User = Depends(get_current_user)):
    # 验证是否为管理员
    if not current_user.is_admin:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN, detail="您没有权限访问此资源"
        )
    ...
```

3.2 歌曲信息管理

歌曲信息管理模块负责对歌曲数据的全生命周期管理，包括创建、读取、更新和删除操作。普通用户可通过关键词搜索歌曲，管理员可管理歌曲信息（新增、修改、删除）。系统中，歌曲与歌手建立了多对一的关联关系，并通过中间表实现与歌单的多对多关系。以下从数据库设计、前端服务、后端接口三个方面详细描述其实现。

3.2.1 数据库设计

歌曲表 (`songs`)

```
class Song(Base):
    __tablename__ = "songs"

    id = Column(Integer, primary_key=True, index=True) # 歌曲唯一标识
    title = Column(String, nullable=False) # 歌曲名称
    genre = Column(String) # 歌曲风格
    artist_id = Column(Integer, ForeignKey("artists.id")) # 关联的歌手 ID
```

```
artist = relationship("Artist", back_populates="songs") # 与歌手表的关系
playlists = relationship("Playlist", secondary="playlist_songs",
back_populates="songs") # 歌曲所属歌单关系
```

字段说明:

- **id**: 歌曲的唯一标识, 用于数据库主键。
- **title**: 存储歌曲名称, 不允许为空。
- **genre**: 歌曲风格, 如流行、摇滚等, 可为空。
- **artist_id**: 外键关联歌手表 (**artists**), 表示歌曲的创作者。
- **artist**: 定义与歌手的多对一关系。
- **playlists**: 定义与歌单的多对多关系, 通过中间表 **playlist_songs** 实现。

3.2.2 功能实现流程

1. 获取歌曲列表

- 用户可以按关键字搜索歌曲, 或获取全部歌曲列表 (默认返回前十条数据)。
- 实现逻辑:
 - 前端 **fetchSongs(search)** 函数发送请求, 携带可选的搜索关键词和分页参数。
 - 后端根据关键词在数据库中筛选标题包含指定内容的记录, 并返回结果。
 - 返回的每条记录中附带关联歌手的名称信息。

2. 添加新歌曲

- 管理员可以添加歌曲, 并选择关联的歌手。
- 实现逻辑:
 - 前端 **createSong(songData)** 提交歌曲信息, 包括标题、风格、和歌手 ID, 需附带 Token 进行权限验证。
 - 后端验证当前用户是否为管理员。
 - 检查提交的歌手 ID 是否存在, 若无关联歌手, 可设为空。
 - 插入新歌曲数据到数据库并返回结果。创建成功后会再次调用 **fetchSongs(search)** 刷新界面上的歌曲列表。

3. 更新歌曲信息

- 管理员可以更新已有歌曲的标题、风格或关联歌手。
- 实现逻辑:
 - 前端 **updateSong(songData)** 提交修改后的歌曲数据, 包括歌曲 ID 和更新内容。需附带 Token。
 - 后端验证当前用户是否为管理员, 并检查对应的歌曲是否存在。
 - 根据提供的数据更新字段值并保存至数据库。修改成功后会再次调用 **fetchSongs(search)** 刷新界面上的歌曲列表。

4. 删除歌曲

- 管理员可以删除指定的歌曲。
- 实现逻辑:

- 前端请求删除某歌曲，提交对应的歌曲 ID。需附带 Token。
- 后端验证用户身份后，检查歌曲是否存在。
- 删除数据库中对应的歌曲记录，并维护相关联表数据的完整性（如解除歌单与歌曲的关联）。删除成功后会再次调用`fetchSongs(search)`刷新界面上的歌曲列表。

3.3 歌手信息管理

歌手信息管理模块主要用于对歌手数据的维护，包括创建、修改、和删除歌手信息。管理员拥有管理权限，可以通过该模块对歌手进行完整的生命周期管理（增删改），并确保与歌曲的关联关系保持一致。

数据库中的 `artists` 表存储歌手的核心信息：

- `id`：歌手的唯一标识，用作主键。
- `name`：歌手姓名，需唯一且不能为空，确保每个歌手信息唯一。
- `genre`：歌手的主要风格（如流行、摇滚、古典等），为可选字段。
- 关系定义：与 `songs` 表建立了一对多的关联关系，便于通过歌手查询其创作的所有歌曲。

和歌曲信息的管理同理，歌手信息模块也实现了类似的权限管理、查增删改功能。此处从略。

3.4 歌单信息管理

歌单管理模块的主要功能是：用户能够创建、删除歌单并向自己创建的歌单添加、删除歌曲。管理员能够管理所有用户的歌单信息以及歌单中的歌曲。歌单通过与用户和歌曲的关联关系，构成系统内的重要模块，支持用户自定义内容组织与管理。

3.4.1 数据库设计

1. `playlists` 表（歌单表）

- 主要字段：
 - `id`：歌单的唯一标识，主键。
 - `name`：歌单名称，不能为空。
 - `owner_id`：歌单所有者，外键指向 `users` 表的 `id` 字段，表示歌单的归属用户。
- 设计目的：

用于存储歌单的基本信息，并通过 `owner_id` 字段与用户建立一对多关系，支持多用户歌单管理。

2. `playlist_songs` 表（歌单-歌曲多对多关系表）

- 主要字段：
 - `playlist_id`：外键，指向 `playlists` 表的 `id` 字段，表示关联的歌单。
 - `song_id`：外键，指向 `songs` 表的 `id` 字段，表示关联的歌曲。
- 设计目的：

建立歌单与歌曲的多对多关系，支持歌单包含多首歌曲以及单首歌曲被多个歌单引用的需求。

3. `songs` 表（歌曲表）

- 主要字段：
 - `id`：歌曲的唯一标识，主键。
 - `title`：歌曲名称。
 - `genre`：歌曲风格（可选）。

- `artist_id`: 外键, 指向 `artists` 表的 `id` 字段, 表示歌曲的所属歌手。
 - 设计目的:
为歌单提供可选的音乐内容, 并支持通过 `artist_id` 字段查询歌曲的创作者信息。
-

3.4.2 核心功能实现

1. 创建歌单

- 用户登录后可通过提交歌单名称创建歌单。
- 验证用户身份, 确保创建的歌单归属于当前用户。
- 在 `playlists` 表中插入新记录并返回歌单的详细信息。
- 约束: 歌单名称不可为空。

2. 查看歌单

- 对普通用户, 支持按关键字搜索歌单名称或获取所有用户的歌单。
- 返回歌单的详细信息, 包括名称、拥有者和包含的歌曲。
- 此外, 用户自己的歌单还会单独列出。通过token解码得到的 `owner_id` 查询特定用户的所有歌单并返回歌单的详细信息 (包括歌曲列表), 在页面上进行展示。

3. 添加歌曲到歌单

- 用户选择歌单后, 通过歌曲名称和歌手名称指定待添加的歌曲。
- 验证当前用户是否有权限操作 (歌单创建者或管理员)。
- 检索 `songs` 表和 `artists` 表, 验证待添加的歌曲及其歌手是否存在。
- 将歌单与歌曲的关联关系插入 `playlist_songs` 表, 完成关联。
- 约束: 歌单中不能重复添加相同歌曲; 且普通用户无权限向他人歌单添加歌曲。

4. 从歌单中移除歌曲

- 用户指定歌单和待移除的歌曲。
- 验证当前用户是否有权限操作 (歌单创建者或管理员)。
- 检索 `playlist_songs` 表, 移除对应的歌单-歌曲关联关系。
- 约束: 待移除的歌曲必须存在于指定歌单中, 否则返回错误。

5. 删除歌单

- 用户指定待删除的歌单。
- 验证歌单归属用户是否有权限操作 (歌单创建者或管理员)。
- 删除 `playlists` 表中的指定歌单记录, 同时删除 `playlist_songs` 表中与该歌单相关的所有关联记录。
- 歌单只能由其拥有者或管理员删除。

3.4.3 权限控制与数据完整性

用户仅能管理自己创建的歌单; 管理员可管理所有用户的歌单。添加、移除歌曲等操作需验证歌单归属权限。删除歌单时, 自动清理 `playlist_songs` 表中的冗余关联记录。若歌单中的歌曲被删除, 需自动解除其与歌单的关联关系。

上述设计的歌单管理模块实现了用户对歌单的灵活管理以及与歌曲的高效关联，同时保证了多用户系统中数据的一致性和权限安全。

4. 部署方法

这个信息系统使用PostgreSQL提供数据库服务，采用FastApi后端框架和Vue前端框架。用户使用前应先下载好PostgreSQL，创建好用户和密码、音乐系统数据库（schema）。同时为了支持前端npm包管理还应安装Nodejs，此处略。

1. 连接数据库

系统代码中，在`database.py`中将数据库url修改为用户自己的url，格式为：`DATABASE_URL = "postgresql+asyncpg://{用户名}:{用户密码}@localhost:5432/{schema名称}"`。以下是一个示例：

```
DATABASE_URL = "postgresql+asyncpg://postgres:123456@localhost:5432/music"
```

完成后初始化数据库并创建相关的表，在项目根目录下，命令行输入：

```
python init_db.py
```

2. 运行后端服务器

安装FastApi环境，新建虚拟环境后在项目根目录下命令行输入：

```
pip install -r requirements.txt
```

安装完成后命令行输入启动服务器：

```
uvicorn main:app
```

3. 运行前端页面

根据启动后端服务器时的输出，替换前端服务接口发送的目标url，例如：

输出为：

```
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

则将`./music_pj/src/services/`文件夹下的`user_service.js`、`artist_service.js`、`song_service.js`和`playlist_service`中`API_BASE_URL`替换为对应的后端API地址，这里是`'http://localhost:8000'`。

然后在项目根目录下进入前端项目工作目录并启动前端页面即可：

```
cd music_pj  
npm run dev
```