Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Spot(Music/Podcast DB)

Administración y diseño de base de datos

José Miguel Hernández Santana
(alu0101101507@ull.edu.es)

Ibai Heras Rodrigalvarez
(alu0101767611@ull.edu.es)

Mikel Mugica Arregui
(alu0101773780@ull.edu.es)

Índice:

Introducción y descripción

Nuestro proyecto se centra en el diseño, creación e implementación de una base de datos para una aplicación dirigida a la escucha de música y podcasts, la cual llamaremos Spot a la aplicaciones y a la base de datos será Music/Podcast Database.

El objetivo propio de la aplicación Spot es la de almacenar la música y podcasts que escucharan y compartirán los usuarios, para ello hemos de empezar describiendo a los diferentes actores que entran en juego en nuestra aplicación.

Los usuarios tienen un ID de usuario, un nombre completo (que será formado por su nombre y su apellido, un nickname (nombre de usuario con el que se le conocerá en la aplicación) y un email identificativo del usuario.

Seguidamente contaremos con las Canciones que se componen de un ID de canción que lo representa, el nombre de la canción , la duración (en horas, minutos y segundos) de la canción, el número de reproducciones de la misma y el año de salida de la canción.

Después tendremos los Podcasts, formados por un ID de podcast identificativo, un nombre de podcast, la duración del programa, el número de reproducciones de la canción y la fecha de lanzamiento del episodio.

Con esto tendremos los 3 grandes atributos de nuestra aplicación , pero entre ellos tambien tienen relacion y serán las siguientes:

Los usuarios pueden crear Playlists para almacenar canciones, que también pueden compartir para que otra gente siga sus listas, así como el propio usuario puede seguir otras listas compartidas, haciendo así que más de un usuario pueda crear o seguir una playlist. Las Playlists estarán formadas por un ID playlists identificativo, un nombre de la playlist y la duración de la playlist entera.

Otra funcionalidad que tienen los usuarios es la existencia de un historial que es único para cada usuario y en el cual se almacenan las canciones escuchadas por cada usuario y este historial estará formado por ID del historial identificativo y el número de visitas totales de las canciones

Para cada canción existente viene asociado con un autor de la canción y esta entidad autor está formada por ID del autor identificativo , el nombre del propio autor y la discografía o los programas de ese autor dependiendo de qué tipo de autor sea pues estos se dividen de manera exclusiva (es decir no pueden ser de otro tipo que el que ya tienen asignado). Los autores pueden ser Grupos, formado por el Nombre del grupo, pueden ser Artistas solitarios, formados por su Nombre de artista y Hosts que están formados por su Nombre de Host y estos representan a los Hosts de los podcast.

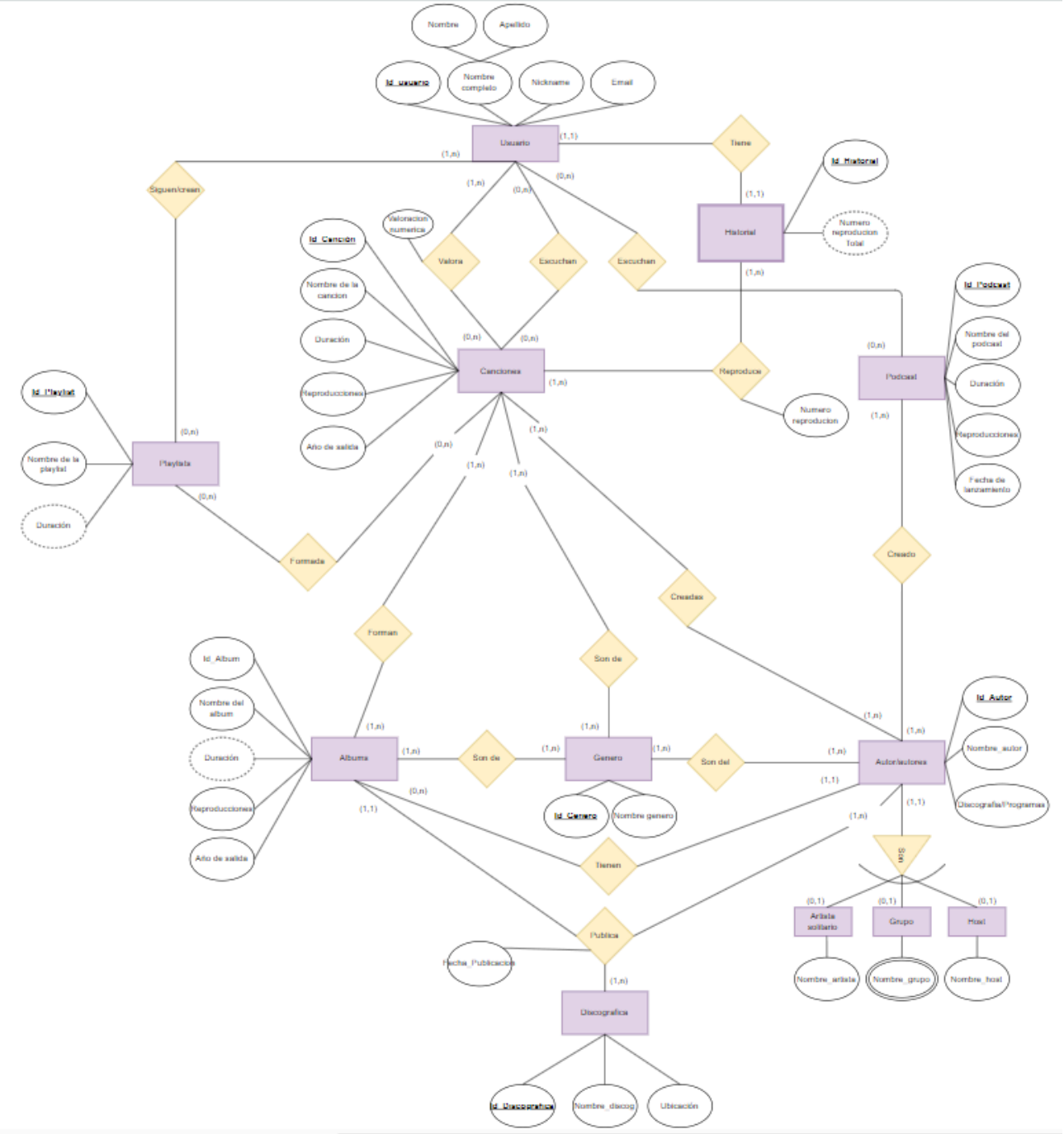
A su vez los autores tienen álbumes que están formados por canciones y estos álbumes se forman a su vez por un ID de álbum identificativo, el nombre del álbum, la duración del álbum(que se halla de la duración de las canciones que forman el álbum) , el número de reproducciones del álbum y el año de salida del propio álbum.

A su vez tanto los autores, como los álbumes y las canciones tienen que ser de uno o varios géneros y estos géneros están formados por el ID del género y el nombre del género.

A su vez existen discográficas que publican los albums de los diferentes autores , y estas discográficas están formadas por el ID de la discográfica identificativo , el nombre de la discografía y la ubicación de donde se encuentra físicamente la discográfica

Esto sería a grandes rasgos los campos principales que entran en juego en nuestra aplicación y los que tomaremos para el desarrollo.

Modelo Entidad/Relación



Es modelo entidad/relación de nuestra aplicación está compuesto por un conjunto de entidades fuertes y débiles , cada una con sus correspondientes atributos, y una serie de relaciones en las que se presentan diferentes cardinalidades para las relaciones entre los atributos, siendo estos 1:1, 1:N, 0:N, N:M.

Este modelo entidad/relación las entidades que se observan son:

Entidades fuertes:

- **Usuarios**: esta entidad contiene el atributo identificador id_usuario, así como el atributo compuesto Nombre completo , que está formado por los atributos descriptivos nombre y apellido, y otros atributos descriptivos como el email y el nickname.

- **Canciones**: esta entidad contiene el atributo identificador id_usuario y atributos descriptivos los cuales son el nombre de la canción, la duración de la canción ,el número de reproducciones y el año de salida.

- **Podcast**: esta entidad está formada por el atributo identificador id_podcast, y por atributos descriptivos los cuales son el nombre del podcast, la duración, el nº de reproducciones y la fecha de salida del episodio.

- **Playlists**: esta entidad está formada por el atributo identificador id_playlist, un atributo descriptivo que es el nombre de la playlist y un atributo derivado que es la duración de la playlist que se saca de la duración de las canciones de las que esté formada la playlist.

- **Albums:** esta entidad está formada por el atributo identificador id_album y por una serie de atributos descriptivos como lo son el nombre del álbum, el año de salida y el número de reproducciones así como un atributo derivado que es la duración que se estima por la duración de las canciones que lo forman.

- **Género**: esta entidad está formada por el atributo identificador id_genero y el atributo descriptivo nombre del género.

- **Autores**: esta entidad está formada por el atributo identificador id_autor y por unos atributos descriptivos los cuales son el nombre del autor y la discografía o programas que tenga dicho autor.

- **Artista solitario**: esta entidad está formada por el atributo descriptivo nombre del artista, pues heredará de la entidad autores.

- **Grupo**: esta entidad está formada por el atributo descriptivo que es el nombre del grupo y que a su vez será multivaluado porque puede tener diferentes valores y heredará también de la entidad autores.

- **Host**: esta entidad está formada por el atributo descriptivo que es el nombre del host y heredará de la entidad autores también.

- **Discográfica**: esta entidad está formada por el atributo identificador id_discográfica y los atributos descriptivos que serán el nombre de la discográfica y la ubicación de la misma.

Entidad débil:

- **Historial**: esta entidad es una entidad débil ya que depende de la existencia de la entidad usuario para poder existir y de la entidad canciones. Esta entidad está formada por el atributo identificador id_historial y por el atributo multivaluado que es el número de reproducciones totales que hay.

Relaciones entre entidades:

- **Usuarios y Canciones**: siendo esta de las más importantes tenemos que existe una relación (N:M) que implica que ninguno usuario o varios pueden escuchar o ninguna canción o varias canciones. A su vez tiene también una relación (N:M) de valoración en la que una canción es valorada por 1 o varios usuarios (1:N) y un usuario valora 0 o varias canciones (0:N) y de donde se deriva el atributo de la valoracion numerica a dar por el usuario.

- **Usuarios y Podcast**: aquí tenemos una relación (N:M) en la que implica que ninguno usuario o varios(0:N) pueden escuchar o ningún podcast o varias podcast(0:N)

- **Usuarios y Playlist**: aquí contamos con una relación (N:M) en la que ningún usuario o varios pueden seguir una playlist (o crearla si es colaborativa por ejemplo) (0:N) y también que una playlist tiene puede ser seguida/creada por uno (cuando se crea la playlist y solo tiene a su creador) o varios usuarios (1:N).

- **Playlist y Canciones**: aquí tenemos la relación (N:M) en la que una playlist puede estar formada por ninguna(cuando se crea principalmente) o varias canciones(0:N), y a su vez ninguna o varias pueden formar playlists(0:N).

- **Canciones y Álbumes:** aquí tenemos una relación (N:M) donde una o varias canciones forman un álbum (1:N) y un álbum puede estar formado por una o varias canciones (1:N).

- **Canciones y Género**: aquí tenemos una relación (N:M) ya que una o varias canciones (1:N) pueden ser de 1 o varios géneros (1:N).

- **Canciones y Autores**: aquí tenemos una relación (N:M) ya que una o varias canciones(1:N) puede ser creada por uno o varios autores (1:N).

- **Podcast y Autores**: aquí tenemos una relación (N:M) en la que uno o varios podcast(1:N) pueden ser creados por uno o varios autores (1:N).

- **Álbumes y Género**: aquí tenemos una relación (N:M) ya que uno o varios álbumes (1:N) pueden ser de uno o varios géneros (1:N).

- **Álbumes y Autores**: aquí tenemos una relación (1:N) ya que un álbum tiene como mínimo un autor y como máximo un autor (1:1) y un autor tiene uno o varios álbumes (0:n).

- **Género y Autores**: aquí tenemos una relación (N:M) ya que uno o varios géneros (1:N) pueden representar a uno o varios autores(1:N).

- **Albums, Discográfica y Autores**: en esta ocasión vemos una relación triple (1:N.N) entre álbumes, discográfica y autores donde vemos que un álbum es publicado por un autor en una discográfica, pero un autor puede publicar en una varias discográficas y una discográfica puede tener uno o varios autores, de esta relación se deriva el atributo Fecha de publicación.

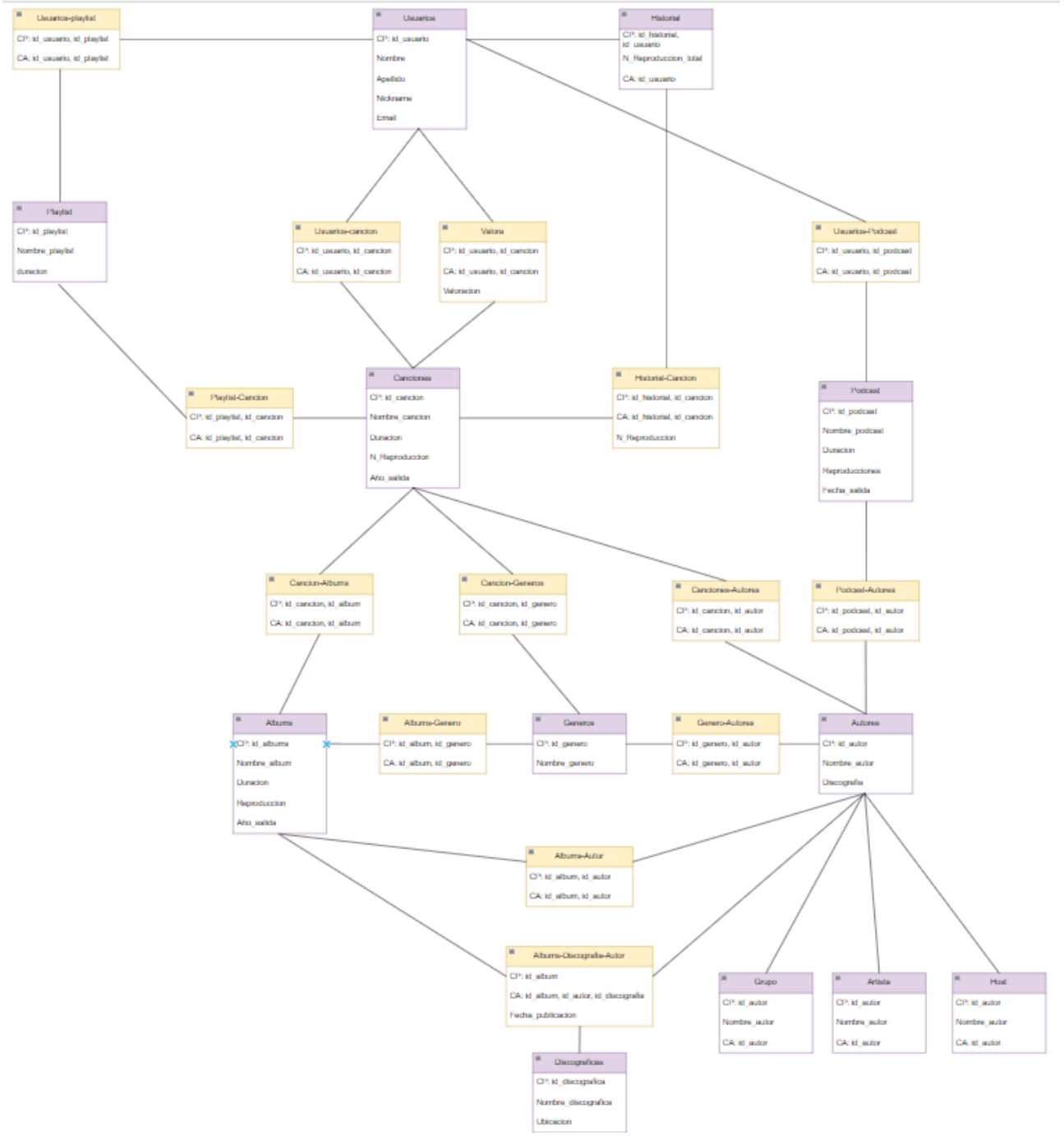
- **Historial y Usuarios**: aquí observamos que la relación que se da es (1:1) ya que un usuario tiene un solo historial y un historial pertenece a solo un usuario.

- **Historial y Canciones**: aquí observamos que la relación que se da es (N:M) ya que un historial puede almacenar la reproducción de una o varias canciones y una canción puede aparecer en uno o varios historiales de diferentes usuarios.

Herencia:

La herencia que aparece en nuestro modelo se presenta en el caso de los artistas, grupos y host pues heredan de la entidad autores, pues los autores pueden ser grupos, artistas o host pero nunca pueden ser más de uno de esos tipos, es decir un artista solitario no puede ser host a la vez por ejemplo.

Modelo relacional / Grafo Relacional



Este modelo relacional se crea a raíz de la transformación del modelo entidad relación a través de sus entidades y relaciones, de las cuales se crearán (si son necesarias por el paso de sus cardinalidades) las tablas intermedias como por ejemplo Album-Autor que nos permitirá unir la información de ambas tablas de manera que podamos hacer una consulta que nos deje ver el autor que ha creado el álbum, creando una imagen así que representa las tablas con las que contará nuestra base de datos final.

Descripción de la base de datos en SQL

La base de datos `music_db` ha sido diseñada para gestionar información sobre usuarios, canciones, álbumes, podcasts, géneros musicales, autores y discográficas, entre otros elementos relacionados con el mundo de la música. El esquema proporciona una estructura relacional que facilita la gestión de las relaciones entre usuarios, su historial de reproducción, playlists, autores y otros aspectos importantes del contenido musical.

Tablas Principales

1. Usuarios
    - Almacena la información de los usuarios.
    - Campos:
        - `id_usuario`: Identificador único del usuario (clave primaria).
        - `nombre`: Nombre del usuario.
        - `apellido`: Apellido del usuario.
        - `nickname`: Nombre de usuario único.
        - `email`: Dirección de correo electrónico del usuario.

```sql
-- Tabla: Usuarios
CREATE TABLE Usuarios (
    id_usuario SERIAL PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL,
    apellido VARCHAR(100) NOT NULL,
    nickname VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(150) UNIQUE NOT NULL,
    CHECK (CHAR_LENGTH(nickname) >= 3)
);
```

2. Playlists
    - Contiene las playlists que los usuarios pueden crear y gestionar.
    - Campos:
        - `id_playlist`: Identificador único de la playlist.
        - `nombre_playlist`: Nombre de la playlist.
        - `duracion`: Duración total de la playlist.

```sql
-- Tabla: Playlist
CREATE TABLE Playlists (
    id_playlist SERIAL PRIMARY KEY,
    nombre_playlist VARCHAR(100) NOT NULL,
    duracion TIME NOT NULL
);
```

3. Canciones
    - Almacena las canciones disponibles en la plataforma.
    - Campos:
        - `id_cancion`: Identificador único de la canción.
        - `nombre_cancion`: Nombre de la canción.
        - `duracion`: Duración de la canción.
        - `n_reproducciones`: Número de reproducciones de la canción.
        - `ano_salida`: Año en que la canción fue lanzada.

```sql
-- Tabla: Canciones
CREATE TABLE Canciones (
    id_cancion SERIAL PRIMARY KEY,
    nombre_cancion VARCHAR(150) NOT NULL,
    duracion TIME NOT NULL,
    n_reproducciones INT DEFAULT 0 CHECK (n_reproducciones >= 0),
    ano_salida INT NOT NULL
);
```

4. Historial
    - Registra el historial de reproducción de canciones por parte de los usuarios.
    - Campos:
        - `id_historial`: Identificador único del historial.
        - `id_usuario`: Referencia al usuario que realizó la reproducción.
        - `n_reproduccion_total`: Número total de reproducciones del usuario.

```sql
-- Tabla: Historial
CREATE TABLE Historial (
    id_historial SERIAL,
    id_usuario INT,
    n_reproduccion_total INT DEFAULT 0 CHECK (n_reproduccion_total >= 0),
    PRIMARY KEY (id_historial, id_usuario),
    FOREIGN KEY (id_usuario) REFERENCES Usuarios(id_usuario) ON DELETE CASCADE
);
```

5. Podcast
    - Contiene información sobre los podcasts disponibles en la plataforma.
    - Campos:
        - `id_podcast`: Identificador único del podcast.
        - `nombre_podcast`: Nombre del podcast.
        - `duracion`: Duración total del podcast.
        - `reproducciones`: Número de reproducciones del podcast.
        - `fecha_salida`: Fecha de publicación del podcast.

```sql
-- Tabla: Podcast
CREATE TABLE  Podcast (
    id_podcast SERIAL PRIMARY KEY,
    nombre_podcast VARCHAR(150) NOT NULL,
    duracion TIME NOT NULL,
    reproducciones INT DEFAULT 0 CHECK (reproducciones >= 0),
    fecha_salida DATE NOT NULL
);
```

6. Albumes
    - Almacena los álbumes de los artistas.
    - Campos:
        - `id_album`: Identificador único del álbum.
        - `nombre_album`: Nombre del álbum.
        - `duracion`: Duración total del álbum.
        - `reproduccion`: Número de reproducciones del álbum.
        - `ano_salida`: Año de lanzamiento del álbum.

```sql
-- Tabla: Albumes
CREATE TABLE  Albumes (
    id_album SERIAL PRIMARY KEY,
    nombre_album VARCHAR(150) NOT NULL,
    duracion TIME NOT NULL,
    reproduccion INT DEFAULT 0 CHECK (reproduccion >= 0),
    ano_salida INT NOT NULL
);
```

7. Generos
    - Contiene los géneros musicales disponibles en la plataforma.
    - Campos:
        - `id_genero`: Identificador único del género.
        - `nombre_genero`: Nombre del género.

```sql
-- Tabla: Generos
CREATE TABLE Generos (
    id_genero SERIAL PRIMARY KEY,
    nombre_genero VARCHAR(100) NOT NULL
);
```

8. Autores
    - Almacena información sobre los autores (artistas, grupos y hosts).
    - Campos:
        - `id_autor`: Identificador único del autor.
        - `nombre_autor`: Nombre del autor.
        - `discografia`: Discografía del autor (opcional).

```sql
-- Tabla: Autores
CREATE TABLE Autores (
    id_autor SERIAL PRIMARY KEY,
    nombre_autor VARCHAR(150) NOT NULL,
    discografia TEXT
);
```

9. Grupos, Artistas y Hosts
    - Relacionan a los autores con sus roles respectivos.
    - Los autores pueden ser grupos, artistas o hosts. Esto está controlado mediante las tablas `Grupos`, `Artista` y `Hosts`.

```sql
-- Tabla: Grupo
CREATE TABLE Grupos (
    id_autor INTEGER,
    nombre_grupo VARCHAR(150) NOT NULL,
    PRIMARY KEY (id_autor, nombre_grupo),
    FOREIGN KEY (id_autor) REFERENCES Autores(id_autor) ON DELETE CASCADE
);

-- Tabla: Artista
CREATE TABLE Artista (
    id_autor INTEGER,
    nombre_artista VARCHAR(150) NOT NULL,
    PRIMARY KEY (id_autor, nombre_artista),
    FOREIGN KEY (id_autor) REFERENCES Autores(id_autor) ON DELETE CASCADE
);

-- Tabla: Host
CREATE TABLE Hosts (
    id_autor INTEGER,
    nombre_host VARCHAR(150) NOT NULL,
    PRIMARY KEY (id_autor, nombre_host),
    FOREIGN KEY (id_autor) REFERENCES Autores(id_autor) ON DELETE CASCADE
);
```

10. Discograficas
    - Almacena las discográficas a las cuales los autores están asociados.
    - Campos:
        - `id_discografia`: Identificador único de la discográfica.
        - `nombre_discografica`: Nombre de la discográfica.
        - `ubicacion`: Ubicación de la discográfica.

```sql
-- Tabla: Discograficas
CREATE TABLE Discograficas (
    id_discografia SERIAL PRIMARY KEY,
    nombre_discografica VARCHAR(150) NOT NULL,
    ubicacion TEXT
);
```

Relaciones y Claves Foráneas

Las tablas están relacionadas mediante claves foráneas que permiten mantener la integridad referencial entre ellas. Algunas de las relaciones clave incluyen:

1. Usuarios - Playlists:
    - Los usuarios pueden tener varias playlists y cada playlist puede estar asociada a varios usuarios a través de la tabla `Usuarios_Playlist`.

```sql
-- Tabla: Usuarios-Playlist
CREATE TABLE Usuarios_Playlist (
    id_usuario INT NOT NULL,
    id_playlist INT NOT NULL,
    PRIMARY KEY (id_usuario, id_playlist),
    FOREIGN KEY (id_usuario) REFERENCES Usuarios(id_usuario),
    FOREIGN KEY (id_playlist) REFERENCES Playlists(id_playlist) ON DELETE CASCADE
);
```

2. Usuarios - Canciones:
    - Los usuarios pueden tener un historial de canciones que han escuchado, lo cual se almacena en la tabla `Historial_Cancion`.

```sql
-- Tabla: Usuarios-Canciones
CREATE TABLE Usuarios_Canciones (
    id_usuario INT NOT NULL,
    id_cancion INT NOT NULL,
    PRIMARY KEY (id_usuario, id_cancion),
    FOREIGN KEY (id_usuario) REFERENCES Usuarios(id_usuario),
    FOREIGN KEY (id_cancion) REFERENCES Canciones(id_cancion) ON DELETE CASCADE
);
```

3. Playlists - Canciones:
    - Las canciones pueden formar parte de múltiples playlists, lo cual se gestiona en la tabla `Playlist_Cancion`.

```sql
-- Tabla: Playlist-Cancion
CREATE TABLE Playlist_Cancion (
    id_playlist INT NOT NULL,
    id_cancion INT NOT NULL,
    PRIMARY KEY (id_playlist, id_cancion),
    FOREIGN KEY (id_playlist) REFERENCES Playlists(id_playlist) ON DELETE CASCADE,
    FOREIGN KEY (id_cancion) REFERENCES Canciones(id_cancion) ON DELETE CASCADE
);
```

4. Canciones - Generos:
    - Cada canción puede pertenecer a uno o más géneros, lo cual está controlado en la tabla `Cancion_Generos`.

```sql
-- Tabla: Cancion-Generos
CREATE TABLE Cancion_Generos (
    id_cancion INT NOT NULL,
    id_genero INT NOT NULL,
    PRIMARY KEY (id_cancion, id_genero),
    FOREIGN KEY (id_cancion) REFERENCES Canciones(id_cancion) ON DELETE CASCADE,
    FOREIGN KEY (id_genero) REFERENCES Generos(id_genero) ON DELETE CASCADE
);
```

5. Canciones - Autores:
    - Cada canción puede tener múltiples autores, relacionados a través de la tabla `Canciones_Autores`.

```sql
-- Tabla: Canciones-Autores
CREATE TABLE Canciones_Autores (
    id_cancion INT NOT NULL,
    id_autor INT NOT NULL,
    PRIMARY KEY (id_cancion, id_autor),
    FOREIGN KEY (id_cancion) REFERENCES Canciones(id_cancion) ON DELETE CASCADE,
    FOREIGN KEY (id_autor) REFERENCES Autores(id_autor) ON DELETE CASCADE
);
```

6. Albumes - Canciones:
    - Las canciones pueden estar asociadas a uno o más álbumes, gestionado en la tabla `Cancion_Albums`.

```sql
-- Tabla: Cancion-Albums
CREATE TABLE Cancion_Albums (
    id_cancion INT NOT NULL,
    id_album INT NOT NULL,
    PRIMARY KEY (id_cancion, id_album),
    FOREIGN KEY (id_cancion) REFERENCES Canciones(id_cancion) ON DELETE CASCADE,
    FOREIGN KEY (id_album) REFERENCES Albumes(id_album) ON DELETE CASCADE
);
```

7. Autores - Podcasts:
    - Los autores pueden ser creadores de podcasts, gestionado a través de la tabla `Podcast_Autores`.

```sql
-- Tabla: Podcast-Autores
CREATE TABLE Podcast_Autores (
    id_podcast INT NOT NULL,
    id_autor INT NOT NULL,
    PRIMARY KEY (id_podcast, id_autor),
    FOREIGN KEY (id_podcast) REFERENCES Podcast(id_podcast) ON DELETE CASCADE,
    FOREIGN KEY (id_autor) REFERENCES Autores(id_autor) ON DELETE CASCADE
);
```

8. Autores - Discográficas - Álbumes
    - Los autores pueden estar asociados con una o más discográficas para producir álbumes, lo cual se gestiona a través de la tabla Albumes_Discografias_Autores.

```sql
-- Tabla: Albumes-Discografia-Autor
CREATE TABLE Albumes_Discografias_Autores (
    id_album INT NOT NULL,
    id_autor INT NOT NULL,
    id_discografia INT NOT NULL,
    fecha_publicacion DATE NOT NULL,
    PRIMARY KEY (id_album),
    FOREIGN KEY (id_album) REFERENCES Albumes(id_album) ON DELETE CASCADE,
    FOREIGN KEY (id_autor) REFERENCES Autores(id_autor) ON DELETE CASCADE,
    FOREIGN KEY (id_discografia) REFERENCES Discograficas(id_discografia) ON DELETE CASCADE
);
```

CHECKs y TRIGGERs Implementados

CHECKs

1. **Usuarios:**
    - Se verifica que el formato del email sea válido.

```
-- Tabla: Usuarios
ALTER TABLE Usuarios
ADD CONSTRAINT chk_email_format CHECK (email ~* '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$');
```

2. **Canciones:**
    - Se asegura que el año de salida esté entre 1900 y el año actual.

```
-- Tabla: Canciones
ALTER TABLE Canciones
ADD CONSTRAINT chk_ano_salida CHECK (ano_salida >= 1900 AND ano_salida <= EXTRACT(YEAR FROM CURRENT_DATE));
```

3. **Álbumes:**
    - Valida que el año de salida esté entre 1900 y el año actual.

```
-- Tabla: Albumes
ALTER TABLE Albumes
ADD CONSTRAINT chk_ano_salida_album CHECK (ano_salida >= 1900 AND ano_salida <= EXTRACT(YEAR FROM CURRENT_DATE));
```

4. **Valora:**
    - Restringe la puntuación a valores entre 1 y 5.

```
-- Tabla: Valora
ALTER TABLE Valora
ADD CONSTRAINT chk_puntuacion CHECK (puntuacion BETWEEN 1 AND 5);
```

TRIGGERs

1. **Incrementar reproducciones:**
    - Cada vez que se agrega una reproducción a `Historial_Cancion`, se incrementa el contador de reproducciones totales en `Historial`.

```sql
-- TRIGGER para incrementar reproducciones en Historial cuando se agrega una canción
CREATE OR REPLACE FUNCTION incrementar_reproducciones()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE Historial
    SET n_reproduccion_total = n_reproduccion_total + 1
    WHERE id_historial = NEW.id_historial;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_incrementar_reproducciones
AFTER INSERT ON Historial_Cancion
FOR EACH ROW
EXECUTE FUNCTION incrementar_reproducciones();
```

2. **Validar unicidad en Usuarios:**
    - Evita que dos usuarios tengan el mismo email o nickname.

```sql
-- TRIGGER para asegurar unicidad de nickname y email en Usuarios
CREATE OR REPLACE FUNCTION validar_unicidad_usuario()
RETURNS TRIGGER AS $$
BEGIN
    -- Comprobamos si hay un cambio en el email o nickname
    IF (NEW.email IS DISTINCT FROM OLD.email OR NEW.nickname IS DISTINCT FROM OLD.nickname) THEN
        IF EXISTS (
            SELECT 1 FROM Usuarios WHERE email = NEW.email OR nickname = NEW.nickname
        ) THEN
            RAISE EXCEPTION 'El email o nickname ya existe';
        END IF;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER trigger_validar_unicidad_usuario
BEFORE INSERT OR UPDATE ON Usuarios
FOR EACH ROW
EXECUTE FUNCTION validar_unicidad_usuario();
```

3. **Actualizar reproducciones de Canciones:**
    - Incrementa automáticamente el contador de reproducciones de una canción cuando se registra en `Historial_Cancion`.

```sql
-- TRIGGER para actualizar n_reproducciones en Canciones cuando se agrega una nueva reproducción
CREATE OR REPLACE FUNCTION actualizar_reproducciones_cancion()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE Canciones
    SET n_reproducciones = n_reproducciones + 1
    WHERE id_cancion = NEW.id_cancion;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_actualizar_reproducciones_cancion
AFTER INSERT ON Historial_Cancion
FOR EACH ROW
EXECUTE FUNCTION actualizar_reproducciones_cancion();
```

4. **Validar duración máxima de una Playlist:**
    - Asegura que la duración total de una playlist no supere las 5 horas.

```sql
-- TRIGGER para asegurar que una playlist no supere una duración máxima (por ejemplo, 5 horas)
CREATE OR REPLACE FUNCTION validar_duracion_playlist()
RETURNS TRIGGER AS $$
DECLARE
    duracion_total INTERVAL;
BEGIN
    SELECT SUM(duracion) INTO duracion_total
    FROM Playlist_Cancion pc
    INNER JOIN Canciones c ON pc.id_cancion = c.id_cancion
    WHERE pc.id_playlist = NEW.id_playlist;

    IF duracion_total > INTERVAL '5 hours' THEN
        RAISE EXCEPTION 'La duración total de la playlist no puede superar las 5 horas';
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_validar_duracion_playlist
AFTER INSERT OR DELETE ON Playlist_Cancion
FOR EACH ROW
EXECUTE FUNCTION validar_duracion_playlist();
```

5. **Verificar rol único del autor:**
    - Garantiza que un autor solo pueda ser definido como grupo, artista o host, no más de uno simultáneamente.

```sql
-- Función para verificar que un autor solo pueda ser un grupo, artista o host
CREATE OR REPLACE FUNCTION check_author_role() RETURNS TRIGGER AS $$
BEGIN
    IF (SELECT COUNT(*) FROM Grupos WHERE id_autor = NEW.id_autor) > 0 THEN
        RAISE EXCEPTION 'El autor ya es un grupo';
    ELSIF (SELECT COUNT(*) FROM Artista WHERE id_autor = NEW.id_autor) > 0 THEN
        RAISE EXCEPTION 'El autor ya es un artista';
    ELSIF (SELECT COUNT(*) FROM Hosts WHERE id_autor = NEW.id_autor) > 0 THEN
        RAISE EXCEPTION 'El autor ya es un host';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Trigger para la tabla Grupos
CREATE TRIGGER check_author_role_grupo
BEFORE INSERT ON Grupos
FOR EACH ROW
EXECUTE FUNCTION check_author_role();

-- Trigger para la tabla Artista
CREATE TRIGGER check_author_role_artista
BEFORE INSERT ON Artista
FOR EACH ROW
EXECUTE FUNCTION check_author_role();

-- Trigger para la tabla Hosts
CREATE TRIGGER check_author_role_host
BEFORE INSERT ON Hosts
FOR EACH ROW
EXECUTE FUNCTION check_author_role();
```

Testeo de consultas descriptivas

Para realizar consultas de prueba, eliminaciones y actualizaciones en las diferentes tablas creadas en Music Database.sql, y reflejar casos de disparadores, aserciones y checks, puedes seguir los siguientes ejemplos:

Consultas de Prueba

Consultar todos los usuarios

SELECT * FROM Usuarios;

```
music_db=# SELECT * FROM Usuarios;
 id_usuario | nombre | apellido  | nickname |            email
------------+--------+-----------+----------+-----------------------------
          1 | Juan   | Perez     | juanp    | juan.perez@example.com
          2 | Ana    | Gomez     | anag     | ana.gomez@example.com
          3 | Luis   | Martinez  | luism    | luis.martinez@example.com
          4 | Maria  | Lopez     | marial   | maria.lopez@example.com
          5 | Carlos | Fernandez | carlitof | carlos.fernandez@example.com
          6 | Sofia  | Hernandez | sofiaher | sofia.hernandez@example.com
          7 | Pedro  | Sanchez   | pedros   | pedro.sanchez@example.com
          8 | Laura  | Ramirez   | laurar   | laura.ramirez@example.com
          9 | Javier | Garcia    | javiers  | javier.garcia@example.com
         10 | Lucia  | Martin    | luciama  | lucia.martin@example.com
(10 filas)
```

Consultar todas las canciones

SELECT * FROM Canciones;

```
music_db=# SELECT * FROM Canciones;
 id_cancion |   nombre_cancion    | duracion | n_reproducciones | ano_salida
------------+---------------------+----------+------------------+------------
          6 | Rolling in the Deep | 00:03:48 |            13000 |       2010
          7 | Havana              | 00:03:38 |            11000 |       2017
          8 | Clocks              | 00:05:07 |             9000 |       2002
          9 | Lose Yourself       | 00:05:26 |            16000 |       2002
         10 | Uptown Funk         | 00:04:30 |            14000 |       2014
          1 | Bohemian Rhapsody   | 00:05:55 |             5001 |       1975
          2 | Shape of You        | 00:04:24 |            12001 |       2017
          3 | Imagine             | 00:03:03 |             8001 |       1971
          4 | Blinding Lights     | 00:03:20 |            15001 |       2019
          5 | Hotel California    | 00:06:30 |            10001 |       1976
(10 filas)
```

Consultar todas las playlists

SELECT * FROM Playlists;

```
music_db=# SELECT * FROM Playlists;
 id_playlist | nombre_playlist | duracion
-------------+-----------------+----------
           1 | Rock Classics   | 02:45:00
           2 | Pop Hits        | 03:30:00
           3 | Chill Vibes     | 02:15:00
           4 | Workout Beats   | 01:45:00
           5 | Indie Favorites | 02:50:00
(5 filas)
```

Inserciones de Prueba

Insertar un nuevo usuario

INSERT INTO Usuarios (nombre, apellido, nickname, email)
VALUES ('Carlos', 'Lopez', 'carlitos', 'carlos.lopez@example.com');

```
music_db=# INSERT INTO Usuarios (nombre, apellido, nickname, email)
music_db-# VALUES ('Carlos', 'Lopez', 'carlitos', 'carlos.lopez@example.com');
INSERT 0 1
```

Insertar una nueva cancion

INSERT INTO Canciones (nombre_cancion, duracion, n_reproducciones, ano_salida)
VALUES ('Nueva Cancion', '03:45', 0, 2023);

```
music_db=# INSERT INTO Canciones (nombre_cancion, duracion, n_reproducciones, ano_salida) VALUES ('Nueva Cancion', '03:45', 0, 2023);
INSERT INTO Canciones (nombre_cancion, duracion, n_reproducciones, ano_salida) VALUES ('Nueva Cancion', '03:45', 0, 2023);
INSERT 0 1
```

Insertar un nuevo autor y asociarlo a un grupo

INSERT INTO Autores (nombre_autor, discografia)
VALUES ('Nuevo Autor', 'Discografia del nuevo autor');

INSERT INTO Grupos (id_autor, nombre_grupo)
VALUES ((SELECT id_autor FROM Autores WHERE nombre_autor = 'Nuevo Autor'),
'Nuevo Grupo');

```
music_db=# INSERT INTO Autores (nombre_autor, discografia)
music_db-# VALUES ('Nuevo Autor', 'Discografia del nuevo autor');
INSERT 0 1
music_db=#
music_db=# INSERT INTO Grupos (id_autor, nombre_grupo)
music_db-# VALUES ((SELECT id_autor FROM Autores WHERE nombre_autor = 'Nuevo Autor
'), 'Nuevo Grupo');
INSERT 0 1
```

Actualizaciones de Prueba

Actualizar el nombre de un usuario

```
UPDATE Usuarios
SET nombre = 'julen'
WHERE nickname = 'juanp';
```

```
music_db=# UPDATE Usuarios
music_db-# SET nombre = 'julen'
music_db-# WHERE nickname = 'juanp';
UPDATE 1
```

Actualizar el nombre de una canción

```
UPDATE Canciones
SET nombre_cancion = 'Cancion Actualizada'
WHERE nombre_cancion = 'Nueva Cancion';
```

```
music_db=# UPDATE Canciones
music_db-# SET nombre_cancion = 'Cancion Actualizada'
music_db-# WHERE nombre_cancion = 'Nueva Cancion';
UPDATE 2
```

Eliminaciones de Prueba

Eliminar un usuario

```
DELETE FROM Usuarios
WHERE nickname = 'carlitos';
```

```
music_db=# DELETE FROM Usuarios
music_db-# WHERE nickname = 'carlitos';
DELETE 1
```

Eliminar una canción

```
DELETE FROM Canciones
WHERE nombre_cancion = 'Cancion Actualizada';
```

```
music_db=# DELETE FROM Canciones
music_db-# WHERE nombre_cancion = 'Cancion Actualizada';
DELETE 1
```

Casos de Disparadores, Aserciones y Checks

Insertar un autor que ya es un host (debería activar el disparador y lanzar una excepción)

-- Primero, insertar un autor y hacerlo host
INSERT INTO Autores (nombre_autor, discografia)
VALUES ('Autor Host', 'Discografia del autor host');

INSERT INTO Hosts (id_autor, nombre_host)
VALUES ((SELECT id_autor FROM Autores WHERE nombre_autor = 'Autor Host'), 'Host Autor');

```
music_db=# INSERT INTO Autores (nombre_autor, discografia)
music_db-# VALUES ('Autor Host', 'Discografia del autor host');
INSERT 0 1
music_db=#
music_db=# INSERT INTO Hosts (id_autor, nombre_host)
music_db-# VALUES ((SELECT id_autor FROM Autores WHERE nombre_autor = 'Autor Host'), 'Host Autor');
INSERT 0 1
```

-- Intentar insertar el mismo autor en la tabla Grupos (debería fallar)
INSERT INTO Grupos (id_autor, nombre_grupo)
VALUES ((SELECT id_autor FROM Autores WHERE nombre_autor = 'Autor Host'), 'Grupo del Autor Host');

```
music_db=# INSERT INTO Grupos (id_autor, nombre_grupo)
music_db-# VALUES ((SELECT id_autor FROM Autores WHERE nombre_autor = 'Autor Host'), 'Grupo del Autor Host');
ERROR:  El autor ya es un host
CONTEXTO:  función PL/pgSQL check_author_role() en la línea 8 en RAISE
```

Insertar una canción con duración negativa (debería fallar por el check)

INSERT INTO Canciones (nombre_cancion, duracion, n_reproducciones, ano_salida)
VALUES ('Cancion Invalida', '-03:45', 0, 2023);

```
music_db=# INSERT INTO Canciones (nombre_cancion, duracion, n_reproducciones, ano_salida)
music_db-# VALUES ('Cancion Invalida', '-03:45', 0, 2023);
ERROR:  la sintaxis de entrada no es válida para tipo time: «-03:45»
LÍNEA 2: VALUES ('Cancion Invalida', '-03:45', 0, 2023);
```

Actualizar el número de reproducciones de una canción a un valor negativo (debería fallar por el check)

UPDATE Canciones
SET n_reproducciones = -10
WHERE nombre_cancion = 'Nueva Cancion';

```
music_db=# UPDATE Canciones
music_db-# SET n_reproducciones = -10
music_db-# WHERE nombre_cancion = 'Nueva Cancion';
ERROR:  el nuevo registro para la relación «canciones» viola la restricción «check» «canciones_n_reproducciones_check»
DETALLE:  La fila que falla contiene (12, Nueva Cancion, 03:45:00, -10, 2023).
```

Estos ejemplos cubren inserciones, actualizaciones y eliminaciones en las tablas, así como casos que deberían activar disparadores, aserciones y checks definidos en tu base de datos.

Implementación API REST con Flask

¿Qué es una API REST?

Una API REST (Representational State Transfer) es un conjunto de reglas que permiten la comunicación entre sistemas utilizando el protocolo HTTP. En una API REST, los recursos (datos o funcionalidades) se representan como URLs y se manipulan mediante métodos HTTP estándar como GET, POST, PUT y DELETE.

Las APIs REST son ampliamente utilizadas debido a su simplicidad, escalabilidad y compatibilidad con múltiples lenguajes de programación y plataformas.

¿Qué es Flask?

Flask es un microframework para Python que facilita la creación de aplicaciones web. A pesar de ser ligero, Flask es muy flexible y permite construir desde aplicaciones simples hasta sistemas complejos. Incluye herramientas para gestionar rutas, manejar solicitudes HTTP y trabajar con plantillas, entre otras funcionalidades.

Ventajas de Flask:

- **Ligereza**: No incluye herramientas innecesarias, pero permite integrarlas si es necesario.
- **Flexibilidad**: Puedes estructurar tu proyecto según tus necesidades.
- **Amplia Comunidad**: Cuenta con una gran comunidad y extensiones que facilitan el desarrollo.

Implementación en el Proyecto

Configuración del Proyecto

El proyecto se organiza de la siguiente manera:

- `app.py`: Archivo principal donde se definen las rutas de la API.
- **Estructura de carpetas**:
  - `/templates`: Carpeta que contiene los archivos HTML renderizados por Flask para cada entidad del modelo.¡
  - `/initdb`: Archivos SQL para inicializar la base de datos.
  - `/static`: Archivo estático de estilos para html.

Uso de Psycopg2

El proyecto utiliza `psycopg2` como controlador para interactuar con la base de datos PostgreSQL. Aunque SQLAlchemy es la herramienta principal para ORM (Mapeo Objeto Relacional), `psycopg2` permite ejecutar consultas SQL personalizadas cuando se requiere mayor control.

```python
import psycopg2

conn = psycopg2.connect(
    dbname="music_db",
    user="postgres",
    password="admin",
    host="db",
    port="5432"
)
```

Implementación de la Gestión de Podcasts en la Aplicación

Introducción

La funcionalidad de gestión de podcasts permite al usuario realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre los datos de los podcasts almacenados en la base de datos. Este módulo utiliza Flask como framework para gestionar las rutas y renderizar las plantillas HTML correspondientes, mientras que psycopg2 se emplea para interactuar directamente con la base de datos PostgreSQL.

Rutas de Flask para la Gestión de Podcasts

El archivo `app.py` contiene las siguientes rutas relacionadas con los podcasts:

Visualización de Podcasts:

```python
@app.route('/podcast', methods=['GET'])
def podcasts_page():
    cursor.execute("SELECT * FROM Podcast")
    podcasts = cursor.fetchall()

    return render_template('podcast.html', podcasts=podcasts)
```

1. Esta ruta permite obtener todos los podcasts de la base de datos y renderizar la plantilla `podcast.html`, que muestra los datos en una tabla.

Creación de Podcasts:

```python
@app.route('/podcast/create', methods=['POST'])
def create_podcast():
    nombre_podcast = request.form['nombre_podcast']
    duracion = request.form['duracion']
    fecha_salida = request.form['fecha_salida']
    reproducciones = request.form['reproducciones']

    cursor.execute("""
        INSERT INTO Podcast (nombre_podcast, duracion, fecha_salida, reproducciones)
        VALUES (%s, %s, %s, %s)
    """, (nombre_podcast, duracion, fecha_salida, reproducciones))
    conn.commit()

    return redirect(url_for('podcasts_page'))
```

2. Permite agregar un nuevo podcast a la base de datos mediante un formulario HTML.

Eliminación de Podcasts:

```python
@app.route('/podcast/delete', methods=['POST'])
def delete_podcast():
    id_podcast = request.form['id_podcast']

    cursor.execute("DELETE FROM Podcast WHERE id_podcast = %s", (id_podcast,))
    conn.commit()

    return redirect(url_for('podcasts_page'))
```

3. Elimina un podcast especificado por su ID a través de un formulario.

Actualización de Podcasts:

```python
@app.route('/podcast/update', methods=['POST'])
def update_podcast():
    id_podcast = request.form['id_podcast']
    nombre_podcast = request.form['nombre_podcast']
    duracion = request.form['duracion']
    fecha_salida = request.form['fecha_salida']
    reproducciones = request.form['reproducciones']

    # Recuperar el valor actual de 'reproducciones' para no modificarlo si no se proporciona un nuevo valor
    cursor.execute("SELECT reproducciones FROM Podcast WHERE id_podcast = %s", (id_podcast,))
    current_reproducciones = cursor.fetchone()[0]

    # Si 'reproducciones' no se ha modificado, mantiene el valor actual
    if reproducciones == "":
        reproducciones = current_reproducciones  # Mantener el valor actual de reproducciones

    cursor.execute("""
        UPDATE Podcast
        SET nombre_podcast = %s, duracion = %s, fecha_salida = %s, reproducciones = %s
        WHERE id_podcast = %s
    """, (nombre_podcast, duracion, fecha_salida, reproducciones, id_podcast))
    conn.commit()

    return redirect(url_for('podcasts_page'))
```

4. Permite modificar la información de un podcast existente.

Plantilla HTML: `podcast.html`

La plantilla HTML `podcast.html` muestra la información de los podcasts y proporciona formularios para realizar operaciones CRUD.

```html
<div class="content">
    <h1>Podcasts</h1>
    <table>
        <thead>
            <tr>
                <th>ID</th>
                <th>Nombre</th>
                <th>Duracion</th>
                <th>Reproducciones</th>
                <th>Fecha de Salida</th>
            </tr>
        </thead>
        <tbody>
            {% for podcast in podcasts %}
            <tr>
                <td>{{ podcast[0] }}</td> <!-- ID del podcast -->
                <td>{{ podcast[1] }}</td> <!-- Nombre del podcast -->
                <td>{{ podcast[2] }}</td> <!-- Duración -->
                <td>{{ podcast[3] }}</td> <!-- Reproducciones -->
                <td>{{ podcast[4] }}</td> <!-- Fecha de salida -->
            </tr>
            {% endfor %}
        </tbody>
    </table>
```

Esta parte del codigo refleja la visualización de los registros de la tabla podcast, ya que como se puede ver en la imagen, recoge todos los registros que antes se han enviado desde el endpoint. En la pagina web se vería así:

Podcasts

ID	NOMBRE	DURACION	REPRODUCCIONES	FECHA DE SALIDA
1	True Crime Stories	00:45:00	5000	2022-01-15
2	Tech Talks	00:50:00	7000	2021-08-22
3	The Daily News	00:30:00	8000	2023-03-10
4	Sports Central	00:40:00	3000	2021-11-05
5	History Hour	00:55:00	9000	2022-07-12

Por otro lado, como se ha comentado antes, están las funcionalidades de añadir, eliminar o actualizar los registros de esta tabla. El código html de esta parte del código se vería así

```html
<h2 class="section-title">Añadir Podcast</h2>
<form method="POST" action="/podcast/create">
    <label for="nombre_podcast">Nombre del Podcast:</label>
    <input type="text" id="nombre_podcast" name="nombre_podcast" required>

    <label for="duracion">Duracion (HH:MM:SS):</label>
    <input type="text" id="duracion" name="duracion" required placeholder="00:00:00">

    <label for="fecha_salida">Fecha de Salida:</label>
    <input type="date" id="fecha_salida" name="fecha_salida" required>

    <label for="reproducciones">Reproducciones:</label>
    <input type="number" id="reproducciones" name="reproducciones" min="0" value="0">

    <button type="submit">Añadir Podcast</button>
</form>

<h2 class="section-title">Eliminar Podcast</h2>
<form method="POST" action="/podcast/delete">
    <label for="id_podcast">ID del Podcast:</label>
    <input type="number" id="id_podcast" name="id_podcast" required>

    <button type="submit">Eliminar Podcast</button>
</form>
```

```html
<h2 class="section-title">Actualizar Podcast</h2>
<form method="POST" action="/podcast/update">
    <label for="id_podcast">Podcast ID:</label>
    <input type="number" id="id_podcast" name="id_podcast" required>

    <label for="nombre_podcast">Nombre del Podcast:</label>
    <input type="text" id="nombre_podcast" name="nombre_podcast" required>

    <label for="duracion">Duracion (HH:MM:SS):</label>
    <input type="text" id="duracion" name="duracion" required placeholder="00:00:00">

    <label for="fecha_salida">Fecha de Salida:</label>
    <input type="date" id="fecha_salida" name="fecha_salida" required>

    <label for="reproducciones">Reproducciones:</label>
    <input type="number" id="reproducciones" name="reproducciones" min="0">

    <button type="submit">Actualizar Podcast</button>
</form>
```

Como se puede ver, se envian los datos de los formularios mediante el método POST para que se gestione en la parte del endpoint. La visualización de la página web sería la siguiente:

Cada una de las modificaciones en los registros de la tabla podcast como cualquier otro se verá reflejada inmediatamente en el visualizador de registros. Para estos métodos, los pasos son los siguientes:

- **Interacción del Usuario**:
  El usuario llena un formulario en la página web, proporcionando la información necesaria para realizar una operación CRUD (Crear, Leer, Actualizar o Eliminar). Por ejemplo, al añadir un podcast, el usuario completa campos como nombre, duración, fecha de salida y reproducciones, y presiona el botón "Añadir".
- **Envío de la Solicitud**:
  Al enviar el formulario, el navegador realiza una solicitud HTTP al servidor Flask. Dependiendo de la acción, esta solicitud puede ser un método `POST` (para crear, actualizar o eliminar) o un método `GET` (para leer o visualizar datos).
- **Procesamiento por Flask**:
  Flask recibe la solicitud en la ruta correspondiente definida en el archivo `app.py`. Por ejemplo, si se envía el formulario de creación de un podcast, la ruta `/podcast/create` procesa la solicitud.
- **Recuperación de Datos del Formulario**:
  Flask utiliza `request.form` para extraer los datos enviados por el formulario. Estos datos se asignan a variables en el código.
- **Consulta a la Base de Datos**:
  Flask, con la ayuda de psycopg2, construye y ejecuta una consulta SQL basada en los datos proporcionados por el usuario. Por ejemplo, para crear un podcast, se ejecuta una consulta `INSERT INTO`.
- **Confirmación de Cambios**:
  Después de ejecutar la consulta, se llama al método `commit()` de la conexión a la base de datos para confirmar y aplicar los cambios realizados.
- **Redirección o Respuesta**:
  Una vez completada la operación, el servidor redirige al usuario a una página relevante (por ejemplo, la lista actualizada de podcasts) utilizando `redirect()` y `url_for()`.
- **Actualización de la Página Web**:
  El navegador del usuario recibe la nueva página desde el servidor Flask, que incluye los cambios realizados en la base de datos. Esto se logra mediante una nueva consulta a la base de datos para obtener datos actualizados y renderizarlos en una plantilla HTML

Conclusión

Esta implementación combina Flask para gestionar las rutas y renderizar plantillas HTML, y psycopg2 para realizar consultas SQL directas. La estructura del sistema permite a los usuarios interactuar de manera intuitiva con los datos de los podcasts, promoviendo una experiencia de usuario eficiente y fácil de usar.