Plan de Acción para el TP de Melodia API

FASE 1: Setup inicial y estructura del proyecto

Objetivo: Tener una base sólida y funcional del proyecto.

Tareas:

- Crear estructura de carpetas (`app/`, `tests/`, etc.).
- Inicializar `main.py` con FastAPI y un endpoint de prueba (`GET /ping`).
- Crear el archivo `requirements.txt` con dependencias mínimas.
- Configurar `Dockerfile` para correr FastAPI con Uvicorn.
- Agregar `docker-compose.yml` (aunque sea mínimo).
- Agregar `.gitignore`, `README.md` y `pyproject.toml` o `setup.cfg` si es necesario.

Entregables (commits sugeridos):

- `init: create project structure and FastAPI hello world`
- `chore: add Docker support and basic config files`

FASE 2: Diseño del modelo de datos

Objetivo: Definir correctamente las entidades del sistema: playlists y canciones.

Tareas:

- Crear modelos con SQLAlchemy:
    - `Playlist` (id, name)
    - `Song` (id, title, artist)

    - Tabla intermedia `playlist_songs` (orden de canciones)

- Crear schemas Pydantic para requests/responses.
- Configurar conexión con SQLite y creación automática de tablas.

Entregables:

- `feat: define models and database setup`
- `feat: add pydantic schemas for Playlist and Song`

FASE 3: Implementación de la lógica de negocio

Objetivo: Desarrollar los endpoints RESTful con sus respectivas validaciones.

Tareas:

- Endpoints de **Playlists**:
    - Crear playlist (`POST /playlists`)
    - Ver playlist (`GET /playlists/{id}`)
    - Listar todas (`GET /playlists`)
    - Agregar canción (`POST /playlists/{id}/songs`)
    - Eliminar canción (`DELETE /playlists/{id}/songs/{song_id}`)
    - Reordenar canciones (`PUT /playlists/{id}/reorder`)
- Endpoints de **Songs** (opcional según cómo lo estructures):
    - Crear canciones o usar un pool fijo

Validaciones importantes:

- No agregar dos veces la misma canción.
- Validar IDs de playlist y canción.

- Validar el orden al reordenar.

Entregables:

- `feat: implement playlist CRUD endpoints`
- `feat: add song management and playlist-song linking`
- `refactor: add error handling and request validations`

FASE 4: Testing

Objetivo: Asegurar la robustez del sistema con cobertura razonable de pruebas.

Tareas:

- Configurar `pytest`.
- Crear `conftest.py` con una base de datos SQLite en memoria para tests.
- Tests unitarios de lógica (CRUD).
- Tests de integración (endpoints reales con `TestClient`).
- Medir cobertura (opcional con `pytest-cov`).

Entregables:

- `test: add unit tests for CRUD logic`
- `test: add integration tests for API endpoints`

FASE 5: Documentación y pulido final

Objetivo: Empaquetar todo con un enfoque profesional y autoexplicativo.

Tareas:

- Completar README:
    - Descripción del proyecto
    - Cómo correrlo (con y sin Docker)
    - Cómo ejecutar los tests
    - Posibles mejoras futuras
- Verificar que Swagger (FastAPI) muestre bien los endpoints.
- Dejar el código ordenado, con funciones comentadas si es necesario.
- Agregar etiquetas a los endpoints (`@router.get(..., tags=["Playlists"])`).

Entregables:

- `docs: complete README and Swagger documentation`
- `chore: final cleanup and code polish`
