****<<Initial M music player>>****

****As of:10/1/2025****

****Overall workflow:****

Client hits REST API → Server routes → Player orchestrates → Registry finds a decoder → Decoder produces a stream → Output plays stream (simulation).

****Functional Design of the Music Player Backend****

**This backend is designed as a modular, REST-controlled music player prototype with clear separation of concerns. Here’s how it works:**

* **Entry Point (cmd/initialm/main.go)**

1. **Starts a small HTTP REST API on localhost:8080.**
2. **Routes: /play, /pause, /resume, /next, /prev, /stop.**
3. **Each route calls into the Player methods.**
4. **Initializes the registry, output, playlist, and player.**

* **Core Orchestration (internal/core/player.go)**

1. **Player struct is the orchestrator: it glues together the registry (decoders), output (audio device), and playlist (track list).**
2. **Handles play, pause, resume, stop, next, previous.**
3. **Thread safety via sync.Mutex.**
4. **Delegates decoding to plugins (e.g., MP3), playback to the Output, and track navigation to Playlist.**

* **Interfaces (internal/core/interfaces.go)**

1. **Defines abstract interfaces for Decoder, DecodedStream, Output, Playlist, Registry.**
2. **Enables plugins and outputs to be swapped without changing player logic (Dependency Inversion Principle).**

* **Registry (internal/core/registry.go)**

1. **Manages registered decoders.**
2. **Finds the right decoder based on file extension.**

* **Output Simulation (internal/output/sim\_output.go)**

1. **Simulates audio output.**
2. **Pretends every track plays for ~5 seconds, then calls back onEnd() so Player auto-advances.**
3. **Maintains play state (Stopped, Playing, Paused).**

* **Playlist Implementation (internal/playlist/fs\_playlist.go)**

1. **File system–based playlist: scans a directory for .mp3 files.**
2. **Provides current, next, previous, jumpTo navigation.**
3. **Maintains track order with a cursor.**

* **Plugin Example (internal/plugins/mp3\_decoder.go)**

1. **MP3 decoder plugin (dummy implementation).**
2. **Recognizes .mp3 files and returns a fake DecodedStream.**
3. **Shows how new formats (FLAC, AAC, etc.) could be added.**

****Summary of SOLID in this backend:****

* **Single Responsibilities Principle:**

**Player: orchestrates playback, doesn’t decode, doesn’t scan filesystem, doesn’t handle output.**

**fsPlaylist: only responsible for listing & navigating tracks.**

**mp3Decoder: only decodes MP3 files.**

**Server: only exposes REST API, doesn’t handle music logic.**

* **Open/Closed Principle :**

**New plugins can be added without modifying existing code.**

**Be able to add new Output (later changing to real audio driver) without changing the“Player”**

**Be able to add new Playlist (later adding Spotify API, database music list)without changing the “Player”**

* **Liskov Substitution:**

**All interface implementations can substitute without breaking logic.**

* **Interface Segregation Principle:**

**As you can see interfaces are small and focused.**

* **Dependency Inversion Principle:**

**For example: “Player” depends on interfaces (out put, playlist, registry..) not direct implementations**

**simOutput’s decoder is not concrete, it depends on core.DecodedStream (it is an abstraction).**