# NEOKODI / EPIKODI

## Technical Documentation

# Table of Contents

# 1. Project Overview

NeoKodi (executable: EpiKodi) is a cross-platform desktop media center built in Rust, inspired by Kodi. It provides a unified interface for browsing, organizing, and playing local and streaming media — including music, videos, images, and IPTV streams — with support for a plugin architecture.

| Property | Value |
|---|---|
| Project Name | EpiKodi (NeoKodi) |
| Language | Rust (edition 2021) |
| UI Framework | Dioxus 0.5 (Desktop/WebView) |
| Database | SQLite via rusqlite |
| Audio Engine | rodio 0.21 |
| Version | 0.1.0 |
| Platform | Windows (primary), Linux, macOS |

**Purpose**

EpiKodi acts as a self-hosted Kodi-like media manager. It scans local folders for media files, stores metadata in a local SQLite database, and provides a reactive GUI to browse, play, and organize the content. A plugin system allows extending functionality — for example, fetching artist metadata from external APIs.

# 2. Architecture Overview

EpiKodi follows a layered architecture with clear separation between the UI, command bus, business logic, and data persistence layers.

| Layer | Technology | Responsibility |
|---|---|---|
| Presentation | Dioxus (RSX/WebView) | Renders the GUI, handles user input, dispatches commands |
| Command Bus | std::sync::mpsc channels | Decouples UI from backend; passes Commands and Events |
| Business Logic | Rust modules (library, media, scanner) | Scanning, metadata, playback control, plugin calls |
| Data Access | rusqlite + SQLite | Persists media metadata, playlists, tags, artist info |
| File Server | warp HTTP server | Serves local media files over HTTP to the WebView renderer |
| Plugin System | libloading (dynamic .dll/.so) | Loads and calls external plugin shared libraries |

## Threading Model

EpiKodi uses two primary threads:

- Main thread: Runs the Dioxus GUI event loop.
- Media thread: Spawned via launch_media_thread(). Owns the rodio audio sink, DB connection, scanner, and plugin manager. Receives Commands from the UI and sends back Events.

A warp HTTP server is also launched asynchronously to serve local files to the embedded browser. Communication between the UI and the media thread is handled exclusively through mpsc channels, ensuring thread safety.

## Command/Event Pattern

**Commands** flow from UI → Media Thread. **Events** flow from Media Thread → UI. This one-directional pattern prevents shared mutable state between threads.

# 3. Project Structure

The repository is organized as follows (simplified from the actual file tree):

```
NeoKodi/
├── EpiKodi/                    # Main Rust crate
│   ├── Cargo.toml
│   ├── config.json             # Runtime config (media root path)
│   ├── db/
│   │   ├── library.db          # SQLite database (auto-created)
│   │   └── sources.json        # Media source paths
│   ├── plugins/                # Dynamic plugin .dll/.so files
│   └── src/
│       ├── main.rs             # Entry point
│       ├── config.rs           # AppConfig loader/saver
│       ├── constants/          # App-wide constants
│       │   ├── mod.rs
│       │   └── constants.rs
│       ├── database/           # SQLite DB wrapper
│       │   ├── mod.rs
│       │   └── db.rs
│       ├── gui/                # All GUI code
│       │   ├── mod.rs
│       │   ├── init.rs         # Root App component & event loop
│       │   ├── layout.rs       # AppLayout with sidebar nav
│       │   ├── pages.rs        # All page components (Home, Music, Videos…)
│       │   ├── route.rs        # Dioxus Router routes
│       │   └── style.rs        # Global CSS string
│       ├── media/              # Media data types & playback
│       │   ├── mod.rs
│       │   ├── data.rs         # MediaInfo, MediaType structs
│       │   └── audio.rs        # rodio playback
│       ├── library/            # Scanning & source management
│       │   ├── mod.rs
│       │   ├── media_library.rs
│       │   ├── scanner.rs
│       │   └── sources.rs      # LibraryConfig (sources.json)
│       ├── threading/          # Thread launcher & command definitions
│       │   ├── mod.rs
│       │   ├── command.rs      # Command & Event enums
│       │   └── media_thread.rs # Main backend thread
│       └── plugins/            # Plugin loading & management
│           ├── mod.rs
│           ├── plugin_manager.rs
│           └── functions.rs
├── plugin_api/                 # Shared plugin trait crate
```

```
└── src/lib.rs
```

# 4. Core Modules

## 4.1 Configuration — config.rs

Manages the application's runtime configuration, stored in config.json at the working directory.

| Field | Type | Description |
|---|---|---|
| media_path | String | Root directory path used for initial media source scanning |

### Key Behaviors

- AppConfig::load() checks for config.json in the current directory. If not found, defaults to the current working directory and immediately writes a new config.json.
- AppConfig::save() serializes the struct to pretty-printed JSON and overwrites config.json.
- Config is loaded once at startup inside the App() component hook, and the media_path is exposed as a Dioxus Signal for reactive use in the UI.

> **Note**
> config.json is a runtime file, not committed to source control. It must exist at the directory from which the EpiKodi binary is launched (typically the project root during development, or next to the .exe in a release build).

## 4.2 Constants — constants.rs

Centralizes all magic values and configuration constants for the application.

| Constant | Value / Type | Purpose |
|---|---|---|
| SOURCE_FILE | "db/sources.json" | Path to the media sources config file |
| MEDIA_DB_FILE | "db/library.db" | Path to the SQLite database file |
| DEBUG | bool | Enables/disables debug logging |
| LOG_FILE | "epikodi.log" | Path for the main log file |
| LOG_FILE_MEDIA_ITEMS | "media_items.log" | Path for media item log |
| LOG_IN_CONSOLE | bool | Whether to also print logs to stdout |
| AUDIO_EXTS | [mp3, wav, flac, ogg, mp4] | Extensions recognized as audio files |

| | | |
|---|---|---|
| VIDEO_EXTS | [mp4, mkv, avi, mov] | Extensions recognized as video files |
| IMAGE_EXTS | [jpg, png, bmp, gif] | Extensions recognized as image files |
| PLUGIN_DIR | "./plugins/" | Directory to scan for plugin shared libraries |
| PLUGIN_EXT | dll / dylib / so | Platform-specific plugin file extension (compile-time) |
| NOT_STARTED | 0 (i32) | Media playback status: not started |
| PLAYING | 1 (i32) | Media playback status: currently playing |
| FINISHED | 2 (i32) | Media playback status: finished |

## 4.3 Database Layer — db.rs

The DB struct wraps a rusqlite Connection and provides all data access methods for the application. The database file is auto-created at db/library.db. The db/ directory is also auto-created if missing.

### DB Struct

| Field | Type | Description |
|---|---|---|
| conn | rusqlite::Connection | The open SQLite connection |
| media_rows | Vec<MediaRow> | In-memory cache populated by get_all_media() |

### Method Groups

### Initialization

- new() — Opens or creates db/library.db, auto-creates the db/ directory.
- init_db() — Creates all tables if they do not exist (idempotent, safe to call on every startup).

### Media Table Methods

- insert_media(path, title, duration, media_type) — Inserts a new media row, ignores duplicates.
- upsert_media(ScannedMedia) — Inserts or updates a single media item by path.
- upsert_media_from_scan(Vec<ScannedMedia>) — Batch upsert inside a transaction for performance.
- get_all_media() — Loads all media rows into self.media_rows and returns a reference.

- cleanup_missing_media(Vec<ScannedMedia>) — Deletes DB entries whose paths are no longer on disk.
- update_media_status_and_time(id, status, time_stop, duration) — Persists playback progress.

### Tag Methods

- get_or_create_tag(name) → i64 — Returns existing tag ID or creates a new one.
- get_tag_id(name) → i64 — Fetches an existing tag ID (errors if not found).
- get_all_tags() → Vec<(i64, String)> — Lists all tags ordered alphabetically.
- add_tag_to_media(media_id, tag_id) — Links a tag to a media item.
- remove_tag_from_media(media_id, tag_id) — Removes a tag link.
- remove_tag(tag_id) — Deletes the tag and all its associations.
- get_media_by_tag(tag_name) → Vec<i64> — Returns media IDs with the given tag.

### Playlist Methods

- create_playlist(name) → i64 — Creates a playlist, returns its ID.
- delete_playlist(playlist_id) — Removes playlist and all its items.
- add_media_to_playlist(media_id, playlist_id) — Adds a media entry to a playlist.
- remove_media_from_playlist(media_id, playlist_id) — Removes a media entry.
- get_media_from_playlist(playlist_id) → Vec<i64> — Returns ordered media IDs.
- get_playlist_id(name) → i64 — Looks up a playlist by name.
- get_all_playlists() → Vec<(i64, String)> — Lists all playlists alphabetically.

### Artist Metadata Methods

- save_artist_metadata(name, info) — Upserts artist info fetched from plugins.
- get_all_artist_metadata() → Vec<String> — Returns all cached artist info strings.

## 4.4 GUI Layer

The GUI is built with Dioxus, a React-inspired framework for Rust that compiles to a desktop WebView (via dioxus-desktop). All GUI code lives in the gui/ module.

### init.rs — Root Component

The App() function is the root Dioxus component. It:

- Sets up mpsc channels (cmd_tx / evt_rx) for UI ↔ media thread communication.
- Launches the media thread via launch_media_thread().
- Sends initial bootstrap commands: AddSource, GetAllMedia, GetAllPlaylists, GetPluginHistory.

- Provides global Signals via use_context_provider: media_list, playlists, loaded_ids, plugin_result, root_path_signal, iptv_channels, iptv_loading.
- Runs a use_coroutine event loop that polls the event channel every 50ms and dispatches events to the correct signals.

### layout.rs — App Shell

AppLayout renders the persistent two-column layout:

- Left: nav sidebar with Link components to all routes.
- Right: Outlet::<Route> that renders the active page component.

### route.rs — Routing

Defines the Dioxus Router routes. Known routes (based on layout.rs navigation):

| Route | Component | Description |
|---|---|---|
| / (Home) | Home | Landing page with category tiles |
| /videos | Videos | Video browser and player |
| /series | Series | Series/episode browser |
| /music | Music | Music player with playlists |
| /images | Images | Image gallery viewer |
| /plugins | Plugins | Add-on/plugin management |
| /settings | Settings | App configuration |
| /iptv | Iptv | IPTV / M3U stream browser |

### pages.rs — Page Components

Contains all page-level components. Notable implementation in the Music page:

**PlayMode enum**

| Variant | Icon | Behavior |
|---|---|---|
| StopAtEnd | 🛑 Stop | Stops after current track finishes |
| Sequential | ➡️ Suite | Plays next track in list order |
| Random | 🔀 Hasard | Picks a random next track |
| Loop | 🔁 Boucle | Repeats the current track |

The Music page features:

- Live search filtering by title.

- Playlist filter mode (activated by clicking a playlist, deactivated by ✖).
- Per-track context menu for adding to existing or new playlists.
- Queue management (add track to play queue, view queue popup).
- Automatic artist metadata fetching via Command::GetArtistMetadataFromPlugin on play.

### make_url() — Local File URL Builder

Converts a Windows absolute path (e.g. E:\game\...\file.mp3) to a warp server URL (http://127.0.0.1:3030/drives/e/...). Each path segment is individually URL-encoded to preserve slashes as route separators.

## 4.5 Media System

### data.rs

Defines the core data types:

- MediaType enum — Audio, Video, Image. Implements to_string() for DB storage and from_db() for deserialization.
- MediaInfo struct — Represents a media item with fields: id, path, title, artist, duration, media_type, status, last_position.

### audio.rs

Wraps rodio for audio playback. Provides functions for play, pause, stop, seek, and volume control. The rodio Sink is owned by the media thread.

## 4.6 Library System

### sources.rs — LibraryConfig

Deserializes db/sources.json into a LibraryConfig struct containing separate lists of source paths for music, video, and image media types. Source paths can be added/removed via the Settings UI and are persisted back to sources.json.

### scanner.rs

Recursively walks configured source directories using Rust's standard fs::read_dir. For each file, it checks the extension against AUDIO_EXTS, VIDEO_EXTS, and IMAGE_EXTS. Audio files are processed with lofty to extract ID3/Vorbis tags (title, artist, duration). Returns Vec<ScannedMedia>.

### media_library.rs

Coordinates the scanner with the database. Calls scanner, then calls upsert_media_from_scan and cleanup_missing_media to keep the DB in sync with the file system.

- ScannedMedia struct: path, name, duration, media_type.

## 4.7 Threading & Commands

### command.rs — Command & Event Enums

All inter-thread communication is typed through these two enums:

**Command variants (UI → Media Thread):**

- Play(i64) — Play media by DB ID.
- Pause / Stop / Seek(f64) — Playback control.
- AddSource(PathBuf, MediaType) — Register a new scan source.
- GetAllMedia() — Trigger DB load and emit MediaList event.
- GetAllPlaylists() — Emit PlaylistList event.
- AddPlaylist(String) / DeletePlaylist(i64) — Playlist CRUD.
- AddMediaToPlaylist(i64, i64) / RemoveMediaFromPlaylist(i64, i64) — Playlist membership.
- GetMediaFromPlaylist(i64) — Emit IDList with playlist contents.
- GetArtistMetadataFromPlugin(String) — Trigger plugin lookup for an artist.
- GetPluginHistory — Emit previous plugin results.
- LoadM3U(String) — Parse an M3U/M3U8 URL and emit M3UList.

**Event variants (Media Thread → UI):**

- MediaList(Vec<MediaInfo>) — Full media library snapshot.
- PlaylistList(Vec<(i64, String)>) — All playlists.
- IDList(Vec<i64>) — Media IDs for a specific playlist.
- NowPlaying(i64) — Currently playing media ID.
- Info(MediaInfo) — Metadata for now-playing track.
- PluginDataReceived(String) — Artist info string from plugin.
- M3UList(Vec<TVChannel>) — Parsed IPTV channel list.

### media_thread.rs

The launch_media_thread function spawns a dedicated OS thread that:

- Initializes the DB, loads sources, and performs the initial scan.
- Owns the rodio audio output stream and Sink.
- Runs a blocking loop processing Command messages from the mpsc receiver.
- Uses a local Tokio runtime (tokio::runtime::Builder) for async operations like HTTP requests in plugins.

## 4.8 Plugin System

### plugin_api crate

A separate Rust crate (plugin_api/) defines the shared plugin interface. Any plugin must implement the Plugin trait defined here. This crate is a dependency of both EpiKodi (consumer) and any plugin (implementor).

### plugin_manager.rs

Uses libloading to dynamically load .dll/.so/.dylib files from the ./plugins/ directory. For each file matching PLUGIN_EXT, it attempts to load a symbol (typically create_plugin or similar) and cast it to the Plugin trait object.

### functions.rs

Contains higher-level plugin invocation helpers, such as calling a plugin's artist lookup function and formatting the result into a string suitable for the PluginDataReceived event.

> **Plugin Development**
> To create a plugin: (1) Add plugin_api as a dependency. (2) Implement the Plugin trait. (3) Export a C-compatible constructor function. (4) Compile as a cdylib. (5) Place the resulting .dll/.so in the plugins/ directory.

## 4.9 Scanner

The scanner walks directories breadth-first using std::fs::read_dir. It filters files by extension (case-insensitive) and uses lofty to read audio tags from supported formats. Scan results (Vec<ScannedMedia>) are passed to the library layer for DB upsert.

# 5. Data Flow

Below is the typical flow for a user clicking a track to play it in the Music page:

| Step | Actor | Action |
|---|---|---|
| 1 | User | Clicks a track row in the Music page |
| 2 | Music component (GUI) | Dispatches Command::Play(media_id) via cmd_tx.send() |
| 3 | Media thread | Receives Command::Play, looks up the media path in DB |
| 4 | Media thread | Opens file via make_url() HTTP path, feeds to rodio Sink |
| 5 | Media thread | Sends Event::NowPlaying(id) and Event::Info(media_info) back |
| 6 | App event loop (init.rs) | Receives events via evt_rx, updates current_audio signal |
| 7 | Music component (GUI) | Reactively re-renders: highlights active track, shows artist info |
| 8 | Media thread (async) | Dispatches Command::GetArtistMetadataFromPlugin(artist) |
| 9 | Plugin manager | Calls plugin, gets info string, sends Event::PluginDataReceived |
| 10 | GUI | Inserts plugin result into plugin_result signal, UI updates |

# 6. Database Schema

The SQLite database at db/library.db contains the following tables:

### media

| Column | Type | Constraints | Description |
|---|---|---|---|
| id | INTEGER | PRIMARY KEY | Auto-increment row ID |
| path | TEXT | UNIQUE NOT NULL | Absolute file path |
| title | TEXT | | Track/file title (from tags or filename) |
| duration | REAL | | Duration in seconds |
| media_type | TEXT | | "audio", "video", or "image" |
| status | INTEGER | DEFAULT 0 | 0=not started, 1=playing, 2=finished |
| time_stop | FLOAT | DEFAULT 0.0 | Last playback position in seconds |

### tags

| Column | Type | Constraints | Description |
|---|---|---|---|
| id | INTEGER | PRIMARY KEY | Auto-increment |
| name | TEXT | UNIQUE NOT NULL | Tag label (e.g. "favorites") |

### media_tags

| Column | Type | Constraints | Description |
|---|---|---|---|
| media_id | INTEGER | FK → media.id | References the media item |
| tag_id | INTEGER | FK → tags.id | References the tag |
| (composite) | | PRIMARY KEY | Prevents duplicate tag assignments |

### playlists

| Column | Type | Constraints | Description |
|---|---|---|---|
| id | INTEGER | PRIMARY KEY | Auto-increment |
| name | TEXT | UNIQUE COLLATE NOCASE | Playlist name (case-insensitive unique) |

### playlist_items

| Column | Type | Constraints | Description |
|---|---|---|---|
| playlist_id | INTEGER | FK → playlists.id | Owning playlist |
| media_id | INTEGER | FK → media.id | Media item |
| position | INTEGER | DEFAULT 0 | Order within playlist |
| (composite) | | PRIMARY KEY | No duplicate entries per playlist |

### artist_metadata

| Column | Type | Constraints | Description |
|---|---|---|---|
| id | INTEGER | PRIMARY KEY | Auto-increment |
| artist_name | TEXT | UNIQUE NOT NULL | Artist name (cache key) |
| info | TEXT | | Serialized artist info from plugin |
| last_updated | DATETIME | DEFAULT CURRENT_TIMESTAMP | Cache timestamp |

# 7. Dependencies

| Crate | Version | Purpose |
|---|---|---|
| dioxus | 0.5 | Reactive desktop GUI framework (desktop + router + hooks features) |
| dioxus-desktop | 0.5 | Desktop WebView renderer for Dioxus |
| rodio | 0.21.1 | Audio playback (mp3, wav, flac, ogg) |
| rusqlite | 0.31 (bundled) | SQLite wrapper with bundled libsqlite3 |
| lofty | 0.22.4 | Audio tag reader (ID3, Vorbis, etc.) |
| tokio | 1 (full) | Async runtime used in media thread for HTTP |
| warp | 0.3 | Async HTTP server for serving local media files |
| libloading | 0.8 | Dynamic library loading for plugins |
| plugin-api | local path | Shared plugin trait (../plugin_api) |
| serde | 1.0 (derive) | Serialization/deserialization framework |
| serde_json | 1.0 | JSON parsing for config.json, sources.json |
| base64 | 0.21 | Base64 encoding (likely for embedded images) |
| rfd | 0.14 | Native file/folder picker dialogs |
| directories | 5.0 | Platform-specific user directory resolution |
| dirs | 5.0 | Simpler directory access (home, data, etc.) |
| once_cell | 1.18 | Lazily initialized statics (RELOAD_SIGNAL) |
| reqwest | 0.11 (blocking) | HTTP client for plugin metadata fetches |
| chrono | 0.4.43 | Date/time utilities |
| urlencoding | 2.1 | URL percent-encoding for file paths |
| lazy_static | 1.4.0 | Macro for lazy static initialization |
| rand | 0.8.0 | Random number generation (shuffle/random play mode) |

# 8. Build & Run

### Prerequisites

- Rust stable toolchain (rustup recommended).
- Windows: MSVC build tools or MinGW (for rusqlite bundled feature).
- Linux/macOS: gcc, libssl-dev, pkg-config.

### Development Build

```
cd EpiKodi
cargo run
```

### Release Build

```
cd EpiKodi
cargo build --release
# Binary: EpiKodi/target/release/EpiKodi.exe (Windows)
```

### Running

The binary must be run from the EpiKodi/ directory (or wherever config.json and db/ live), because all paths are relative to the working directory.

```
# From EpiKodi/ directory:
.\target\release\EpiKodi.exe   # Windows
./target/release/EpiKodi         # Linux/macOS
```

### First Run

- config.json is auto-created with media_path set to the current directory.
- db/ directory and library.db are auto-created.
- Go to Settings to configure your actual media source paths.

# 9. Configuration Files

### config.json

```
{
  "media_path": "E:\\game\\Projet Pro\\NeoKodi"
}
```

Sets the root media path used for initial source bootstrapping. Editable via Settings UI.

### db/sources.json

```
{
  "sources": [],
  "music_sources": [{ "path": "C:\\Users\\..." }],
  "video_sources": [{ "path": "C:\\Users\\..." }],
  "image_sources": [{ "path": "C:\\Users\\..." }]
}
```

Stores the list of scan source directories per media type. Multiple paths can be added for each type. Modified at runtime when user adds/removes sources in Settings.

# 10. Logging

EpiKodi uses a file-based logging system controlled by the constants in constants.rs.

| Constant | Default | Effect |
|---|---|---|
| DEBUG | true | Enables log output |
| LOG_FILE | "epikodi.log" | Main application log file |
| LOG_FILE_MEDIA_ITEMS | "media_items.log" | Detailed log for media items |
| LOG_IN_CONSOLE | false | If true, also prints to stdout |

Set LOG_IN_CONSOLE = true during development for easier debugging. The log files are written relative to the working directory.

# 11. Testing

Unit tests are located in db.rs using Rust's built-in #[cfg(test)] module. Tests use in-memory SQLite databases (Connection::open_in_memory()) to avoid touching the filesystem.

### Running Tests

```
cd EpiKodi
cargo test
```