FLACCID CLI Toolkit Developer Handbook

Introduction

FLACCID is a modular command-line toolkit designed for managing FLAC audio libraries and integrating with music streaming services. The name "FLACCID" reflects its focus on FLAC audio and a CLI-first design. This toolkit enables users to download high-quality music from services like Qobuz and Tidal, tag their local FLAC files with rich metadata (including album art and lyrics), and maintain a well-organized music library database. The rebuild of FLACCID focuses on a clean architecture with clear module separation, plugin support for multiple providers, and modern Python libraries for robust functionality.

Key Features:

- **Flexible CLI:** A single `fla` command with subcommands for different tasks (downloading, tagging, library management, configuration). Built with **Typer** for intuitive CLI syntax and help.
- **Modular Plugins:** Support for multiple music services (Qobuz, Tidal, Apple Music, etc.) via a plugin system. Easy to add new providers for metadata, streaming, or lyrics.
- **Rich Metadata Tagging:** Uses **Mutagen** to write comprehensive tags to FLAC files, including album art embedding and lyrics. Metadata can be aggregated from multiple sources in a cascade.
- **Asynchronous Downloading:** Leverages **aiohttp** for efficient downloading of tracks (e.g., parallel downloads of an album) with progress display using **Rich**.
- **Robust Configuration:** Configuration managed with **Dynaconf** (TOML-based settings) and validated with **Pydantic** models. Secure credentials are handled via **Keyring** and Dynaconf's secrets support (no plain-text passwords in code or repo).
- **Library Indexing**: Maintains a local SQLite database (via **SQLAlchemy**) of your music library. Can scan directories, index metadata, and perform integrity checks (file existence, hash verification, etc.).
- **Automation**: Optionally uses **Watchdog** to monitor the filesystem for new or changed files, updating the library in real-time.
- **Cross-Platform**: Works on Windows, macOS, and Linux. Paths and storage locations are handled appropriately per OS, and keyring integration uses the OS keychain on each platform.
- **Testing & Quality**: A full test suite (using Pytest) covers modules and CLI commands, ensuring reliability. Developers can easily run tests and extend functionality with confidence.
- **Packaging & Deployment**: Distributed as a Python package with a `pyproject.toml`. Supports standard installation (`pip install .`) and editable installs for development (`pip install -e .`). A guide is provided for publishing and dependency management.

This handbook serves as a developer guide to the architecture and implementation of FLACCID. It details the directory structure, provides full source code listings of each major module, and explains how the components interact. Whether you are a developer looking to extend FLACCID or a power user wanting to understand its internals, this document offers a comprehensive walkthrough.

CLI Architecture Overview

The FLACCID toolkit is accessed via the `fla` command-line interface. The CLI is organized into subcommands that mirror the core functionalities:

- **Download Commands**: `fla get qobuz [options]` and `fla get tidal [options]` – Download albums or tracks from Qobuz or Tidal (given appropriate credentials and subscription). Options allow specifying album/track identifiers, quality levels, output paths, etc.
- **Tagging Commands**: `fla tag qobuz [options]` and `fla tag apple [options]` – Tag local FLAC files using metadata from Qobuz or Apple Music (iTunes). These commands fetch metadata (album details, track titles, artwork, lyrics) from the service and write tags to the files.
- **Library Management**: `fla lib scan [options]` and `fla lib index [options]` – Manage the local music library database. Scanning detects new or changed files in the library path and updates the database (with optional continuous watch). Indexing performs a full rebuild or creation of search indices and verifies file integrity.
- **Configuration**: `fla set auth [service]` and `fla set path [location]` – Configure credentials and paths. The `set auth` command securely stores user API credentials for a given service (Qobuz, Tidal, Apple, etc.), and `set path` defines the base path of the music library or downloads.

Using **Typer** (a library built on Click), these subcommands are implemented as grouped command families. This provides an intuitive syntax (e.g., `fla get qobuz ...`) and auto-generates help documentation. The CLI design allows future expansion; new services or features can be added as new subcommands or options.

Each top-level command group (`get`, `tag`, `lib`, `set`) is implemented as a Typer app, added to the main Typer application. This nested structure improves code organization and mirrors how users conceptualize the tasks. For example, the `get` group contains separate commands for each streaming service, and new services can be included by adding a new function/command in that group.

Example CLI Usage:

- Downloading a Qobuz album by ID: `fla get qobuz --album-id 123456 --quality FLAC --out ~/Music/Downloads`
- Downloading a specific Tidal track: `fla get tidal --track-id 7890 --quality Lossless`
- Tagging a local folder with Qobuz metadata: `fla tag qobuz --album-id 123456 "/path/to/AlbumFolder"`
- Tagging using Apple Music (iTunes) search: `fla tag apple "Artist Name - Album Title" "/path/to/AlbumFolder"`
- Scanning library for new music: `fla lib scan` (optionally `--watch` for continuous monitoring)
- Re-indexing entire library with integrity check: `fla lib index --verify`
- Setting credentials for Qobuz: `fla set auth qobuz` (prompts for username/password or token)
- Setting the music library path: `fla set path ~/Music`

Each command comes with `--help` to display usage instructions (thanks to Typer's automatic help generation). The CLI is designed to be user-friendly while providing power-user options for customization.

Project Structure and Module Layout

The FLACCID project is organized into a Python package with clearly separated modules for commands, plugins, core functionality, and configuration. Below is the directory layout of the project:

```
flaccid/
├── __init__.py
├── cli.py               # CLI entry point using Typer, defines main app and groups
├── commands/
│   ├── __init__.py
│   ├── get.py           # 'fla get' subcommands for various services (Qobuz, Tidal)
│   ├── tag.py           # 'fla tag' subcommands for metadata tagging (Qobuz, Apple)
│   ├── lib.py           # 'fla lib' subcommands for library scan and index
│   └── settings.py      # 'fla set' subcommands for auth and path configuration
├── plugins/
│   ├── __init__.py
│   ├── base.py          # Base classes/interfaces for plugins
│   ├── qobuz.py         # Qobuz plugin implementation (metadata & download)
│   ├── tidal.py         # Tidal plugin implementation (metadata & download)
│   ├── apple.py         # Apple Music plugin implementation (metadata only)
│   └── lyrics.py        # Lyrics plugin (fetch lyrics from external API, e.g., Genius)
├── core/
│   ├── config.py        # Configuration loader (Dynaconf settings and Pydantic models)
│   ├── metadata.py      # Metadata cascade and tagging logic (using Mutagen)
│   ├── downloader.py    # Download utilities (async HTTP, file I/O, Rich progress)
│   ├── library.py       # Library scanning, watching (Watchdog), and DB update logic
│   └── database.py      # Database models and setup (SQLAlchemy ORM for tracks, albums,
etc.)
├── tests/
│   ├── test_cli.py      # Tests for CLI commands and argument parsing
│   ├── test_plugins.py  # Tests for plugin outputs (using stub API responses)
│   ├── test_metadata.py # Tests for tagging logic (e.g., verify tags written correctly)
│   ├── test_library.py  # Tests for scanning and DB operations (using temp directories)
│   └── ...              # (Additional tests as needed per module)
├── pyproject.toml       # Project metadata, dependencies, build system, entry points
└── README.md            # User-facing README (installation, basic usage)
```

Module Overview:

- `flaccid/cli.py`: The main entry point that initializes the Typer app and ties together subcommands from the `commands/` package.
- `flaccid/commands/`: Package containing CLI command implementations for each top-level command group (`get`, `tag`, `lib`, `set`). Each file defines a Typer sub-app and commands corresponding to specific services or actions.
- `flaccid/plugins/`: Houses plugin modules, each implementing integration with a specific service or provider (e.g., Qobuz API, Tidal API, Apple Music, lyrics provider). `base.py` defines abstract interfaces for these plugins.
- `flaccid/core/`: Core functionality that is not tied to a specific provider or CLI command, such as configuration management, metadata tagging logic, downloading utilities, library indexing, and the database schema.

- `flaccid/tests/`: Test suite for all components. Includes unit tests for business logic and integration tests for CLI behavior and overall workflows.
- `pyproject.toml`: Defines the Python project, including its name, version, dependencies (Typer, Pydantic v2, Dynaconf, Mutagen, aiohttp, Keyring, Rich, SQLAlchemy, Watchdog, etc.), as well as console script entry point (`fla`).

This structure encourages separation of concerns: the CLI layer handles user interaction and argument parsing, plugins deal with external service APIs, core modules handle generic logic (tagging, downloading, database), and tests ensure each part works correctly. Next, we will dive into each area with code examples and explanations.

CLI Entry Point and Command Dispatching

The CLI entry point is defined in **`flaccid/cli.py`**. This module creates the main Typer application and registers subcommand groups. By using Typer, we get a clean way to define commands and parameters with Python functions and decorators, plus automatic help messages and tab-completion.

Below is the implementation of `cli.py`:

```
# flaccid/cli.py

import typer
from flaccid.commands import get, tag, lib, settings

# Create the main Typer app
app = typer.Typer(help="FLACCID CLI - A modular FLAC toolkit")

# Register sub-apps for each command group
app.add_typer(get.app, name="get", help="Download tracks or albums from streaming
services")
app.add_typer(tag.app, name="tag", help="Tag local files using online metadata")
app.add_typer(lib.app, name="lib", help="Manage local music library (scan/index)")
app.add_typer(settings.app, name="set", help="Configure credentials and paths")

if __name__ == "__main__":
    app()
```

In this code:

- We import the submodules from `flaccid.commands` (each submodule defines a Typer `app` object for its group).
- We create a top-level `app = typer.Typer(...)` and use `app.add_typer(...)` to attach each subgroup under the appropriate name. For example, the `get` module's Typer app is attached as the `get` command group[typer.tiangolo.com](https://typer.tiangolo.com/tutorial/subcommands/nested-subcommands/#:~:text=app %3D typer.Typer() app.add_typer(users.app%2C name%3D,lands).
- Each `add_typer` call also includes a help string describing that group of commands.
- Finally, if the module is executed directly, we invoke `app()`. This allows running `python -m flaccid` during development, and also the entry point for the installed CLI will call this.

By structuring the CLI with multiple Typer instances, each command category is isolated in its own module, making the code easier to maintain and extending to new subcommands straightforward[typer.tiangolo.com](https://typer.tiangolo.com/tutorial/subcommands/nested-subcommands/#:~:text=app %3D typer.Typer() app.add_typer(users.app%2C name%3D,lands). Typer automatically composes the help such that `fla --help` will list the command groups, and `fla get --help` will show the commands under `get`, and so on.

Command Group Implementations

Each command group (`get`, `tag`, `lib`, `set`) has its own module in `flaccid/commands/` containing a Typer app and the specific command functions. Let's go through each:

`fla get` – Download Commands

The `get` command group allows users to download music from supported streaming services. Currently, we implement two subcommands: `qobuz` and `tidal`. These will utilize the corresponding plugins to handle API interactions and downloads.

```
# flaccid/commands/get.py

import asyncio
from pathlib import Path
import typer
from flaccid.core import downloader, config
from flaccid.plugins import qobuz, tidal

app = typer.Typer()

@app.command("qobuz")
def get_qobuz(album_id: str = typer.Option(..., "--album-id", help="Qobuz album ID to
download", rich_help_panel="Qobuz Options"),
              track_id: str = typer.Option(None, "--track-id", help="Qobuz track ID to
download (if single track)"),
              quality: str = typer.Option("lossless", "--quality", "-q", help="Quality
format (e.g. 'lossless', 'hi-res')"),
              output: Path = typer.Option(None, "--out", "-o", help="Output directory
(defaults to library path)")):
    """
    Download an album or track from Qobuz.
    """
    # Ensure output directory is set
    dest_dir = output or config.settings.library_path
    dest_dir.mkdir(parents=True, exist_ok=True)
    typer.echo(f"Downloading from Qobuz to {dest_dir} ...")

    # Initialize Qobuz plugin (ensure credentials are loaded)
    qbz = qobuz.QobuzPlugin()
    qbz.authenticate()  # Will load stored credentials and obtain auth token

    # Fetch album or track metadata
    if album_id:
```

```
        album_meta = asyncio.run(qbz.get_album_metadata(album_id))
        tracks = album_meta.tracks
    elif track_id:
        track_meta = asyncio.run(qbz.get_track_metadata(track_id))
        tracks = [track_meta]
    else:
        typer.echo("Error: You must specify either --album-id or --track-id", err=True)
        raise typer.Exit(code=1)

    # Download all tracks (async)
    asyncio.run(qbz.download_tracks(tracks, dest_dir, quality))
    typer.secho("Qobuz download complete!", fg=typer.colors.GREEN)


@app.command("tidal")
def get_tidal(album_id: str = typer.Option(None, "--album-id", help="Tidal album ID to
download", rich_help_panel="Tidal Options"),
              track_id: str = typer.Option(None, "--track-id", help="Tidal track ID to
download"),
              quality: str = typer.Option("lossless", "--quality", "-q", help="Quality
(e.g. 'lossless', 'hi-res')"),
              output: Path = typer.Option(None, "--out", "-o", help="Output directory
(defaults to library path)")):
    """
    Download an album or track from Tidal.
    """
    dest_dir = output or config.settings.library_path
    dest_dir.mkdir(parents=True, exist_ok=True)
    typer.echo(f"Downloading from Tidal to {dest_dir} ...")

    tdl = tidal.TidalPlugin()
    tdl.authenticate()  # use stored credentials or perform login flow

    if album_id:
        album_meta = asyncio.run(tdl.get_album_metadata(album_id))
        tracks = album_meta.tracks
    elif track_id:
        track_meta = asyncio.run(tdl.get_track_metadata(track_id))
        tracks = [track_meta]
    else:
        typer.echo("Error: You must specify either --album-id or --track-id", err=True)
        raise typer.Exit(code=1)

    asyncio.run(tdl.download_tracks(tracks, dest_dir, quality))
    typer.secho("Tidal download complete!", fg=typer.colors.GREEN)
```

Key points in the `get` commands:

- Both commands accept either an album ID or a track ID (but not both) to identify what to download. They also accept a `--quality` option to request a certain quality level (the interpretation of this is handled by the plugin, e.g., mapping "lossless" to a specific stream format or bitrate).

- `--out` can specify a custom output directory; if not provided, the default library path from config is used.
- We ensure the destination directory exists (`mkdir(parents=True, exist_ok=True)`).
- We initialize the appropriate plugin (`QobuzPlugin` or `TidalPlugin`) and call an `authenticate()` method to ensure we have a valid API session (using credentials from config/Keyring).
- We fetch metadata for the album or track. The plugin provides methods like `get_album_metadata` or `get_track_metadata` which return Pydantic models containing track info (e.g., track titles, file URLs, etc.). We use `asyncio.run()` to execute these async calls synchronously within the Typer command function. Typer can also work with async functions directly, but here we manage it explicitly for clarity.
- Once we have a list of tracks to download, we call `download_tracks(tracks, dest_dir, quality)` via the plugin. This method is asynchronous (to perform concurrent downloads), hence we also wrap it in `asyncio.run()`.
- Progress feedback and logging inside `download_tracks` (using Rich) will inform the user of download progress. After completion, we print a success message in green text (`typer.secho(..., fg=typer.colors.GREEN)`).
- Error handling: if neither album_id nor track_id is given, we output an error message and exit with a non-zero code.

The separation of concerns is evident here: the CLI command mostly handles user input and output, delegates to the plugin for heavy lifting, and ensures the environment (like output directory) is prepared. By using asyncio for downloads, the user can download multiple tracks in parallel, significantly speeding up album downloads.

`fla tag` – Tagging Commands

The `tag` group provides commands to apply metadata tags to existing FLAC files from online sources. We implement `qobuz` for Qobuz metadata and `apple` for Apple Music (iTunes) metadata. These commands typically take an identifier (or search query) for the album and a target directory of files to tag.

```
# flaccid/commands/tag.py

import typer
from pathlib import Path
from flaccid.core import metadata, config
from flaccid.plugins import qobuz, apple

app = typer.Typer()

@app.command("qobuz")
def tag_qobuz(album_id: str = typer.Option(..., "--album-id", help="Qobuz album ID to
fetch metadata"),
              folder: Path = typer.Argument(..., exists=True, file_okay=False,
dir_okay=True, help="Path to the album folder to tag")):
    """
    Tag a local album's FLAC files using Qobuz metadata.
    """
    typer.echo(f"Fetching metadata from Qobuz for album {album_id} ...")
```

```
    qbz = qobuz.QobuzPlugin()
    qbz.authenticate()
    album_meta = qbz.get_album_metadata_sync(album_id)  # using a sync wrapper or
asyncio.run inside
    typer.echo(f"Applying tags to files in {folder} ...")
    metadata.apply_album_metadata(folder, album_meta)
    typer.secho("Tagging complete (Qobuz)!", fg=typer.colors.GREEN)


@app.command("apple")
def tag_apple(query: str = typer.Argument(..., help="Album search query or Apple album
ID"),
              folder: Path = typer.Argument(..., exists=True, file_okay=False,
dir_okay=True, help="Path to the album folder to tag")):
    """
    Tag a local album's FLAC files using Apple Music (iTunes) metadata.
    """
    typer.echo(f"Searching Apple Music for '{query}' ...")
    am = apple.AppleMusicPlugin()
    # The query might be an album ID or a search term
    if query.isdigit():
        album_meta = am.get_album_metadata(query)
    else:
        album_meta = am.search_album(query)
    if album_meta is None:
        typer.echo("Album not found on Apple Music.", err=True)
        raise typer.Exit(code=1)
    typer.echo(f"Found album: {album_meta.title} by {album_meta.artist}. Applying
tags...")
    metadata.apply_album_metadata(folder, album_meta)
    typer.secho("Tagging complete (Apple Music)!", fg=typer.colors.GREEN)
```

Notes on the `tag` commands:

- For `tag qobuz`, we require an `--album-id` (assuming the user knows the Qobuz album ID or has it from a link). The target folder is a positional argument (must exist) pointing to the album's files on disk. We instantiate the Qobuz plugin, authenticate, and fetch album metadata. We then call a core function `apply_album_metadata` to perform the tagging of files in that folder with the retrieved metadata.
- For `tag apple`, we allow the user to either provide an album identifier (if known) or a search query string. If the query is all digits, we treat it as an Apple album ID; otherwise, we perform a search. The AppleMusicPlugin handles the search via Apple's iTunes Search API (no credentials needed for basic search). It returns the best matching album's metadata. If nothing is found, we exit with an error. If found, we proceed to apply tags similarly with `apply_album_metadata`.
- Both commands rely on a core `metadata.apply_album_metadata` function to do the actual tagging (writing FLAC tags, cover art, etc.). This function will use Mutagen and possibly the lyrics plugin as needed (described in the Metadata section).
- The CLI provides user feedback throughout: indicating when it's fetching metadata and when it's writing tags, and a completion message. Errors (like album not found) are printed to stderr and cause a graceful exit.

By splitting the provider-specific logic (in plugins) from the tagging application logic (in `core/metadata.py`), we ensure consistency. The `apply_album_metadata` function can handle merging metadata from multiple sources (if needed) and writing to files, while the plugin simply supplies a structured `Album` data object.

`fla lib` – Library Management Commands

The `lib` command group deals with the local music library database. We have two commands: `scan` and `index`. The design is that `scan` is used to detect changes and update the library incrementally (and possibly run continuously), while `index` can build or rebuild indices and perform deeper checks.

```
# flaccid/commands/lib.py

import typer
from flaccid.core import library, config

app = typer.Typer()

@app.command("scan")
def lib_scan(watch: bool = typer.Option(False, "--watch", help="Continuously watch the
library for changes"),
             verify: bool = typer.Option(False, "--verify", help="Verify file integrity
while scanning")):
    """
    Scan the music library for new, changed, or deleted files and update the database.
    """
    lib_path = config.settings.library_path
    typer.echo(f"Scanning library at {lib_path} ...")
    library.scan_library(lib_path, verify=verify)
    typer.secho("Initial scan complete!", fg=typer.colors.GREEN)
    if watch:
        typer.echo("Watching for changes. Press Ctrl+C to stop.")
        try:
            library.watch_library(lib_path, verify=verify)
        except KeyboardInterrupt:
            typer.echo("Stopped watching the library.")


@app.command("index")
def lib_index(rebuild: bool = typer.Option(False, "--rebuild", help="Rebuild the entire
index from scratch"),
              verify: bool = typer.Option(False, "--verify", help="Perform integrity
verification on all files")):
    """
    Index the entire library and update the database. Use --rebuild to start fresh.
    """
    lib_path = config.settings.library_path
    if rebuild:
        typer.echo("Rebuilding library index from scratch...")
        library.reset_database()  # Drop and recreate tables
    else:
        typer.echo("Updating library index...")
```

```
    library.index_library(lib_path, verify=verify)
    typer.secho("Library indexing complete!", fg=typer.colors.GREEN)
```

Explanation:

- Both `lib scan` and `lib index` use the `flaccid.core.library` module, which contains the implementation of scanning and indexing logic.
- `lib_scan`: Calls `library.scan_library(path, verify=...)` for incremental scanning. The `verify` flag triggers integrity checks (if true, e.g. verifying FLAC file integrity or checksum). After initial scan, if `--watch` is provided, it then calls `library.watch_library` which uses Watchdog to monitor the directory for changes continuously. The `watch_library` function will run until interrupted (Ctrl+C), handling events like created files (adding them), modified files (re-indexing them), or deleted files (removing from DB).
- `lib_index`: If `--rebuild` is specified, it calls `library.reset_database()` to wipe and initialize the DB (so we start fresh). Then it calls `library.index_library(path, verify=...)` which likely performs a full scan of all files and ensures the database is up-to-date. Without `--rebuild`, it might do similar work but preserving existing entries (in practice, `scan_library` and `index_library` might share code; here we conceptually separate quick scan vs full index).
- Both commands use `typer.echo` to inform the user of progress and a success message at the end. The `verify` option in both triggers deeper checks (which might slow the process, so it's optional).
- We use `config.settings.library_path` to know which directory to scan/index. The user sets this via `fla set path` or it defaults to something like `~/Music` or a directory in config.

This design allows users to simply run `fla lib scan` to keep their database in sync with their filesystem, and optionally use `--watch` to keep it running as a daemon. The `index` command can be used for a thorough (or initial) indexing or when moving to a new database.

`fla set` – Configuration Commands

The `set` group provides a way for users to configure essential settings like service credentials and library path from the CLI. This writes to config or uses Keyring for secrets.

```
# flaccid/commands/settings.py

import typer
from flaccid.core import config
import keyring

app = typer.Typer()

@app.command("auth")
def set_auth(service: str = typer.Argument(..., help="Service to authenticate (e.g.
'qobuz', 'tidal', 'apple')")):
    """
    Set or update authentication credentials for a service (stored securely).
    """
    service = service.lower()
    if service not in ("qobuz", "tidal", "apple"):
```

```
        typer.echo(f"Unknown service '{service}'. Available options: qobuz, tidal, apple",
err=True)
        raise typer.Exit(code=1)
    # Prompt user for credentials or token
    if service == "apple":
        typer.echo("Apple Music typically requires a developer token and a music user
token.")
        dev_token = typer.prompt("Enter Apple Music Developer Token (JWT)",
hide_input=True)
        music_token = typer.prompt("Enter Apple Music User Token", hide_input=True)
        # Store tokens in keyring (we use service name and key names)
        keyring.set_password("flaccid_apple", "developer_token", dev_token)
        keyring.set_password("flaccid_apple", "user_token", music_token)
        typer.echo("Apple Music tokens stored securely.")
    else:
        username = typer.prompt(f"Enter {service.capitalize()} username/email")
        password = typer.prompt(f"Enter {service.capitalize()} password", hide_input=True)
        # Store credentials securely in system keyring
        keyring.set_password(f"flaccid_{service}", username, password)
        # Store username in config for reference
        config.settings[service]["username"] = username
        config.settings.save()
        typer.echo(f"{service.capitalize()} credentials stored for user {username}.")
        typer.echo("You can now use 'fla get {service} ...' or 'fla tag {service} ...'
commands.")


@app.command("path")
def set_path(location: str = typer.Argument(..., help="Path to the music library or
download folder")):
    """
    Set the base path for the music library.
    """
    import os
    # Expand user tilde if present
    normalized = os.path.expanduser(location)
    config.settings.library_path = normalized
    config.settings.save()
    typer.echo(f"Library path set to: {normalized}")
```

Details for `set` commands:

- `set auth`: Accepts a service name argument (we support 'qobuz', 'tidal', 'apple'). It then interactively prompts the user for the necessary credentials:
    - For **Apple**: Apple's Music API uses tokens. We prompt for a Developer Token (JWT issued by Apple for your app) and a User Token (authenticates the user's Apple Music account). These are then stored in the keyring under a service identifier "flaccid_apple" with distinct keys for each token. We do not store Apple password because Apple Music doesn't use a simple username/password for API access.

    - For **Qobuz/Tidal**: We prompt for username (or email) and password. We then call `keyring.set_password("flaccid_qobuz", username, password)`. This stores the password securely in the OS keychain. We also save the username in the Dynaconf config (so that the program knows which username to use by default for that service). The config is then saved to persist this change (likely writing to `settings.toml` or a user config file).
    - The user feedback confirms that credentials are stored. On subsequent runs, the plugin will fetch the password from keyring using the stored username.
- Security: By using Keyring, we avoid writing sensitive passwords to disk. Even the Dynaconf `.secrets.toml` file is not used for these (to reduce risk). If we did use Dynaconf secrets file for tokens, it would be kept out of version controldynaconf.com, but here we rely on the system key vault for maximum security.
- `set path`: Takes a directory path and updates `config.settings.library_path`. We expand `~` to the user's home directory for convenience. The new path is saved to config (persisting in the `settings.toml`). Users can thus configure where downloaded music and library files reside. On different OSes, they might set this to a platform-specific location (the tool itself might have a default like `~/Music` on Unix or `%USERPROFILE%\Music` on Windows, but this allows customization).

With these config commands, the user can prepare the environment for using the other features (download/tag). For example, they should run `fla set auth qobuz` once to store their Qobuz credentials before using `fla get qobuz`. The separation into a command means no credentials are required as plain CLI arguments, and the interactive prompt ensures sensitive input isn't shown on screen.

Configuration Design and Secure Credential Management

FLACCID uses **Dynaconf** for managing configuration, combined with **Pydantic** for validation. This provides a flexible, layered configuration system with support for multiple environments, file formats, and secure secrets management.

Dynaconf Settings

Dynaconf allows configuration via a `settings.(toml|yaml|json|py)` file, environment variables, and a special `.secrets.*` file for sensitive data. In FLACCID, we use a `settings.toml` file for general settings and optionally a `.secrets.toml` for any sensitive overrides (although most secrets are stored in keyring instead). The configuration might include:

- Default library path (if not set by user).
- Default quality setting for downloads (e.g., prefer "lossless" or specific format).
- API keys or app IDs for services (for Qobuz or Tidal if required).
- Possibly toggles for features (like enabling lyrics fetching, etc).
- Database file location or name.

An example `settings.toml` might look like:

```
[default]
library_path = "~/Music/FLACCID"  # default library location
default_quality = "lossless"
```

```
# Service-specific settings
[default.qobuz]
app_id = "YOUR_QOBUZ_APP_ID"
app_secret = "YOUR_QOBUZ_APP_SECRET"
username = ""  # filled after auth
# (passwords not stored here for security)

[default.tidal]
client_id = "YOUR_TIDAL_API_KEY"
client_secret = "YOUR_TIDAL_API_SECRET"
username = ""

[default.apple]
# Apple doesn't need username/password in config, uses tokens via keyring
music_user_token = ""  # (optionally store a token here, but we prefer keyring)
```

Dynaconf loads the `default` environment by default. Sensitive values (like actual passwords or tokens) can be put in a separate `.secrets.toml` which is automatically loaded by Dynaconf if presentdynaconf.com. For instance, `.secrets.toml` could contain the Qobuz app secret or an API token. The `.secrets.toml` file is kept out of source control (the project's `.gitignore` should include it by default)dynaconf.com.

Pydantic for Config Validation

We define Pydantic models to validate and access config data in a structured way. Pydantic v2 ensures that types are correct and can provide default values.

```
# flaccid/core/config.py

from pathlib import Path
from pydantic import BaseModel
from dynaconf import Dynaconf

# Initialize Dynaconf to load settings and secrets
settings = Dynaconf(
    envvar_prefix="FLA",
    settings_files=['settings.toml', '.secrets.toml'],
    environments=["default"],
)

class ServiceCredentials(BaseModel):
    username: str = ""
    # We won't store password here, it's in keyring
    # We might store a token if available (e.g., Qobuz auth token after login)
    token: str | None = None

class AppSettings(BaseModel):
    library_path: Path
    default_quality: str = "lossless"
    qobuz: ServiceCredentials = ServiceCredentials()
    tidal: ServiceCredentials = ServiceCredentials()
```

```
    apple: dict = {}  # Apple might not need credentials in the same way

# Validate and parse settings using Pydantic
app_config = AppSettings(**settings)  # This will read values from Dynaconf
```

In this snippet:

- We create a Dynaconf `settings` object to load configuration from `settings.toml` and `.secrets.toml`. We also allow environment variables prefixed with `FLA` to override settings (e.g., `FLA_LIBRARY_PATH=/mnt/music` would override the config).
- We define `ServiceCredentials` as a Pydantic model to represent credentials for a service. It holds a username and possibly a token (for example, once we authenticate to Qobuz via their API, we might get a user auth token which we can store in memory or even update the config).
- `AppSettings` defines the structure of our app's config: the library_path, a default_quality, and nested structures for each service. We use `Path` type for library_path (Pydantic will automatically expand `~` to the user home for Path).
- We then instantiate `AppSettings` with the data from `settings`. Pydantic will validate types (e.g., ensure library_path is a Path, etc.) and apply defaults for missing values. If required fields were missing, it would raise an error (though here we have defaults for all).
- We now have an `app_config` object we can use in code, or we could stick with Dynaconf's `settings`. In practice, Dynaconf's `settings` is already quite convenient (attribute access), so using Pydantic is an optional validation layer. It can catch config mistakes early and provide IDE type hints for config usage.

Secure Credentials with Keyring

As seen in the `set auth` command, we avoid storing raw passwords in config. Instead, **Keyring** is used to store and retrieve credentials from the OS-provided secure storage (like Windows Credential Vault, macOS Keychain, or Secret Service on Linux)alexwlchan.net.

Workflow for credentials:

- When the user runs `fla set auth [service]`, we prompt and then call `keyring.set_password(service_id, username, password)`. We choose a `service_id` like "flaccid_qobuz" or "flaccid_tidal" to namespace our credentials.
- We record the username in the Dynaconf config (so the program knows which account is in use).
- In the plugin's `authenticate()` method, we retrieve the password with `keyring.get_password(service_id, username)` and then use it to obtain an API token or perform login.
- **Qobuz**: Requires an app ID/secret (from config) and user credentials to get a user auth token via their API. After a successful login, we get a token which could be stored in memory for that session. (We might also store the token in keyring or config if we want to reuse it without logging in every time, as long as it's long-lived.)
- **Tidal**: Similar approach – use the user credentials (or possibly OAuth if needed) to get a session/token. (The exact implementation depends on Tidal's API – we may use an unofficial API library that handles the OAuth dance.)

- **Apple**: The `set auth apple` stores the Developer Token and User Token in keyring. These tokens are long-lived (developer token might expire after months, user token typically doesn't expire often unless user revokes access). The AppleMusicPlugin will fetch these via `keyring.get_password("flaccid_apple", "developer_token")` etc., and use them to authenticate requests.

By leveraging Keyring, credentials are never exposed in plaintext within our codebase or config files, and the user benefits from the OS-level security (they might need to unlock their keychain once, etc., depending on platform). This approach is generally more secure than environment variables or plaintext secrets filesalexwlchan.net, though we allow those in a pinch for advanced users (e.g., setting `FLA_QOBUZ_PASSWORD` env var could be another way if keyring is unavailable, but that's not the primary method).

Note: In headless or server environments without a user keyring, developers might need to configure an alternative keyring backend (e.g., using keyring's fallback to plaintext or specifying an environment keyring). By default, our design assumes a desktop environment with access to the user keyring. If keyring throws an error (no backend), we could catch it and prompt the user to either use environment variables or install a suitable backend.

Config Loading and Usage

The `flaccid/core/config.py` ensures that `Dynaconf` loads at program start (when `flaccid.cli` is imported, it will import commands which import config). This means `config.settings` is ready to use anywhere. For convenience, many modules import `config.settings` directly to get config values.

For example, the library path is accessed as `config.settings.library_path`. Typer's CLI commands can use those values at runtime. If the user changed a setting via CLI (like `set path` or `set auth`), we call `settings.save()` to write changes back to the `settings.toml` file. Dynaconf handles writing to the appropriate file (by default it might write to a user-specific config if configured, but here we load from project directory for simplicity – in a real installation, we might specify a path in the user's home directory for the config).

Environment Overrides: We allow environment vars as override for advanced usage. For example, if someone doesn't want to persist config, they could run:

```
FLA_LIBRARY_PATH="/mnt/storage/Music" fla lib scan
```

and the program would pick up that library path from the env var due to `envvar_prefix="FLA"` in our Dynaconf setup.

Credentials Flow Recap:

1. User runs `fla set auth qobuz` and enters credentials.
2. Credentials stored via Keyring; username stored in `settings.toml`.
3. User runs `fla get qobuz ...`.
4. Inside `QobuzPlugin.authenticate()`, the code looks up `username = settings.qobuz.username`, then calls `password = keyring.get_password("flaccid_qobuz", username)`. It then uses Qobuz API (with the app_id and app_secret from config) to log in and obtain a user token. The token is stored in the plugin instance for the session (and could optionally be saved to avoid login next time).

5. Subsequent API calls use that token. The token could also be cached in keyring (as part of `ServiceCredentials.token` or similar) if we want persistence without logging in each run.
6. Similarly for Tidal.

This design balances security and convenience. The user only has to enter passwords once, and they are safely stored. We leverage Dynaconf for non-sensitive settings and Keyring for secrets, following best practices of not hardcoding secrets in code or repository[dynaconf.com](dynaconf.com).

Plugin System Architecture

One of the core goals of FLACCID is modular support for multiple music providers and extensibility. The plugin system is designed to accommodate **streaming/download providers**, **metadata providers**, and other auxiliary data sources (like lyrics). By defining clear interfaces and using a plugin registry, new services can be added with minimal changes to the CLI code.

Plugin Interfaces

In `flaccid/plugins/base.py`, we define abstract base classes (ABCs) that outline the expected functionality of providers. For example:

```
# flaccid/plugins/base.py

from abc import ABC, abstractmethod
from typing import List

class TrackMetadata:
    """Simple data holder for track metadata and download info."""
    def __init__(self, id: str, title: str, artist: str, album: str,
                 track_number: int, disc_number: int = 1,
                 duration: float = 0.0,  # in seconds
                 download_url: str | None = None):
        self.id = id
        self.title = title
        self.artist = artist
        self.album = album
        self.track_number = track_number
        self.disc_number = disc_number
        self.duration = duration
        self.download_url = download_url

class AlbumMetadata:
    """Data holder for album metadata and list of tracks."""
    def __init__(self, id: str, title: str, artist: str, year: int = 0,
                 cover_url: str | None = None, tracks: List[TrackMetadata] = None):
        self.id = id
        self.title = title
        self.artist = artist
        self.year = year
        self.cover_url = cover_url
        self.tracks = tracks or []
```

```
class MusicServicePlugin(ABC):
    """Abstract base class for music service plugins (streaming & metadata)."""

    @abstractmethod
    def authenticate(self):
        """Authenticate with the service (use config credentials)."""
        raise NotImplementedError

    @abstractmethod
    async def get_album_metadata(self, album_id: str) -> AlbumMetadata:
        """Fetch album metadata (tracks, etc.) by album ID."""
        raise NotImplementedError

    @abstractmethod
    async def get_track_metadata(self, track_id: str) -> TrackMetadata:
        """Fetch track metadata by track ID."""
        raise NotImplementedError

    @abstractmethod
    async def download_tracks(self, tracks: List[TrackMetadata], dest_dir, quality: str):
        """Download the given tracks to dest_dir at the specified quality."""
        raise NotImplementedError

class MetadataProviderPlugin(ABC):
    """Base class for metadata-only providers (no direct downloads)."""
    @abstractmethod
    def search_album(self, query: str) -> AlbumMetadata | None:
        """Search for an album by name/artist and return metadata if found."""
        raise NotImplementedError

    @abstractmethod
    def get_album_metadata(self, album_id: str) -> AlbumMetadata | None:
        """Fetch album metadata by an ID specific to this provider."""
        raise NotImplementedError

class LyricsProviderPlugin(ABC):
    """Base class for lyrics providers."""
    @abstractmethod
    def get_lyrics(self, artist: str, title: str) -> str | None:
        """Fetch lyrics for a given song, or return None if not found."""
        raise NotImplementedError
```

Here:

- `TrackMetadata` and `AlbumMetadata` are simple data classes (could also be Pydantic models) to hold information about tracks and albums. This includes fields like title, artist, track number, and for tracks, possibly a direct download URL (some APIs provide a URL or we construct one after authentication).
- `MusicServicePlugin` is an abstract class defining the key operations for a service that provides both metadata and actual audio content (download). It includes methods to authenticate, retrieve album/track metadata, and download tracks.

- `MetadataProviderPlugin` is a narrower interface for services that only provide metadata lookup (no downloading). Apple Music fits here – we can search and get metadata, but we cannot download audio from Apple's API.
- `LyricsProviderPlugin` covers getting lyrics for tracks. This could be implemented by a plugin that calls an API like Genius or Musixmatch, for example.

These interfaces ensure each plugin implements a standard set of methods. The CLI and core code will call these methods without needing to know the implementation details for each service.

Plugin Implementations

Now, let's look at how specific plugins implement these interfaces.

Qobuz Plugin (`flaccid/plugins/qobuz.py`):

```
# flaccid/plugins/qobuz.py

import aiohttp
import asyncio
from flaccid.plugins.base import MusicServicePlugin, AlbumMetadata, TrackMetadata
from flaccid.core import config

class QobuzPlugin(MusicServicePlugin):
    def __init__(self):
        self.app_id = config.settings.qobuz.get("app_id", "")
        self.app_secret = config.settings.qobuz.get("app_secret", "")
        self.username = config.settings.qobuz.get("username", "")
        self._auth_token = None

    def authenticate(self):
        """Obtain Qobuz API auth token using stored credentials."""
        # Fetch password from keyring
        import keyring
        password = keyring.get_password("flaccid_qobuz", self.username)
        if not password or not self.app_id:
            raise RuntimeError("Qobuz credentials or app ID not set. Run 'fla set auth
qobuz'.")
        # Qobuz API: get user auth token
        # Example: https://www.qobuz.com/api.json/0.2/user/login?
username=...&password=...&app_id=...
        auth_url = "https://www.qobuz.com/api.json/0.2/user/login"
        params = {
            "username": self.username,
            "password": password,
            "app_id": self.app_id
        }
        response = asyncio.get_event_loop().run_until_complete(
            aiohttp.ClientSession().get(auth_url, params=params))
        data = asyncio.get_event_loop().run_until_complete(response.json())
        response.close()
        if "user_auth_token" in data:
```

```
            self._auth_token = data["user_auth_token"]
        else:
            raise RuntimeError("Failed to authenticate with Qobuz. Check credentials.")

    async def get_album_metadata(self, album_id: str) -> AlbumMetadata:
        # Qobuz album info API
        url = "https://www.qobuz.com/api.json/0.2/album/get"
        params = {"album_id": album_id, "user_auth_token": self._auth_token}
        async with aiohttp.ClientSession() as session:
            async with session.get(url, params=params) as resp:
                data = await resp.json()
        # Parse album data into AlbumMetadata
        album = AlbumMetadata(
            id=str(data["album"]["id"]),
            title=data["album"]["title"],
            artist=data["album"]["artist"]["name"],
            year=int(data["album"].get("releaseDateDigital", "0")[:4]) if
"releaseDateDigital" in data["album"] else 0,
            cover_url=data["album"].get("image", None),
        )
        tracks = []
        for track in data["tracks"]["items"]:
            t = TrackMetadata(
                id=str(track["id"]),
                title=track["title"],
                artist=track["performer"]["name"],
                album=album.title,
                track_number=track.get("trackNumber", 0),
                disc_number=track.get("mediaNumber", 1),
                duration=track.get("duration", 0.0)
            )
            tracks.append(t)
        album.tracks = tracks
        return album

    async def get_track_metadata(self, track_id: str) -> TrackMetadata:
        # Qobuz track info API
        url = "https://www.qobuz.com/api.json/0.2/track/get"
        params = {"track_id": track_id, "user_auth_token": self._auth_token}
        async with aiohttp.ClientSession() as session:
            async with session.get(url, params=params) as resp:
                data = await resp.json()
        track = data["track"]
        t = TrackMetadata(
            id=str(track["id"]),
            title=track["title"],
            artist=track["performer"]["name"],
            album=track["album"]["title"],
            track_number=track.get("trackNumber", 0),
            disc_number=track.get("mediaNumber", 1),
            duration=track.get("duration", 0.0)
        )
```

```
        return t

    async def download_tracks(self, tracks: list[TrackMetadata], dest_dir, quality: str):
        # Determine format_id based on desired quality
        format_id = 27 if quality.lower() in ("hi-res", "hires") else 6  # example: 6 =
FLAC 16-bit, 27 = FLAC 24-bit
        # Qobuz file URL API
        file_url_api = "https://www.qobuz.com/api.json/0.2/track/getFileUrl"

        import os
        from flaccid.core.downloader import download_file  # a helper for downloading with
aiohttp
        tasks = []
        async with aiohttp.ClientSession() as session:
            for track in tracks:
                # Get the direct file URL for the track
                params = {
                    "track_id": track.id,
                    "format_id": format_id,
                    "user_auth_token": self._auth_token,
                    "app_id": self.app_id
                }
                async with session.get(file_url_api, params=params) as resp:
                    file_data = await resp.json()
                if "url" in file_data:
                    track.download_url = file_data["url"]
                else:
                    typer.secho(f"Failed to get download URL for track {track.title}",
fg=typer.colors.RED)
                    continue
                # Determine file path
                filename = f"{track.track_number:02d} - {track.title}.flac"
                filepath = os.path.join(dest_dir, filename)
                # Create asyncio task for downloading this track
                tasks.append(download_file(session, track.download_url, filepath))
            # Use asyncio.gather to download all tracks concurrently
            await asyncio.gather(*tasks)
```

Important aspects of `QobuzPlugin`:

- **Authentication**: The `authenticate()` method synchronously obtains a `user_auth_token` from Qobuz by calling their login API endpoint with username, password, and app_id. It uses `aiohttp` in a somewhat synchronous way (here we directly run the event loop to perform the GET request; alternatively, we could have made `authenticate` `async` and awaited it, but for simplicity it's sync). On success, it stores the token in `self._auth_token`.
- **get_album_metadata**: Calls Qobuz's album API to retrieve album details and track list. It constructs an `AlbumMetadata` object and a list of `TrackMetadata` for each track. This is an `async` function and uses an `aiohttp.ClientSession` to perform the request and parse JSON.
- **get_track_metadata**: Similar, for an individual track.

- **download_tracks**: This handles getting actual file URLs and downloading them. Qobuz requires another API call (`track/getFileUrl`) with a `format_id` for quality. We choose a format based on the `quality` parameter (in Qobuz API, for example, 5 might be MP3, 6 is 16-bit FLAC, 27 is 24-bit FLAC; here we simplified to two options). For each track:
  - It requests the file URL and sets `track.download_url`.
  - It creates a file name (prefixed with track number for ordering).
  - We then use a helper `download_file(session, url, path)` to asynchronously fetch and save the file. We gather all tasks and await them concurrently.
- We use a single `aiohttp.ClientSession` for all downloads to reuse connections (and pass it into `download_file` tasks).
- The actual implementation of `download_file` (in `flaccid/core/downloader.py`) would handle writing the response stream to disk and possibly updating a progress bar (discussed later). For brevity, assume it's a function that does something like:

  ```
  async def download_file(session, url, path):
      async with session.get(url) as resp:
          with open(path, 'wb') as f:
              async for chunk in resp.content.iter_chunked(1024):
                  f.write(chunk)
  ```

  Possibly enhanced with Rich progress.

The Qobuz plugin thus covers both metadata and downloading, fully implementing `MusicServicePlugin`.

Tidal Plugin (`flaccid/plugins/tidal.py`):

The Tidal plugin would be similar in structure to Qobuz's. Tidal has a different API (and might involve OAuth). For brevity, here's a conceptual outline:

```
# flaccid/plugins/tidal.py

import aiohttp
import asyncio
from flaccid.plugins.base import MusicServicePlugin, AlbumMetadata, TrackMetadata
from flaccid.core import config

class TidalPlugin(MusicServicePlugin):
    def __init__(self):
        self.username = config.settings.tidal.get("username", "")
        self._session_id = None  # Tidal uses a session or access token

    def authenticate(self):
        import keyring
        password = keyring.get_password("flaccid_tidal", self.username)
        if not password:
            raise RuntimeError("Tidal credentials not set. Run 'fla set auth tidal'.")
        # Use tidalapi or direct REST calls to login.
        # For simplicity, imagine we have an unofficial API:
```

```
        # tidal_api = tidalapi.Session()
        # tidal_api.login(self.username, password)
        # self._session_id = tidal_api.session_id
        # (If successful, _session_id is set; real Tidal API may return access token)
        # In absence of actual library, this is a placeholder.
        raise NotImplementedError("Tidal authentication needs implementation")

    async def get_album_metadata(self, album_id: str) -> AlbumMetadata:
        # Call Tidal API to get album and track list (requires auth, e.g., include session
token)
        # Parse into AlbumMetadata and TrackMetadata list.
        # (Pseudo-code as actual API details are omitted)
        album = AlbumMetadata(id=album_id, title="Album Title", artist="Artist Name")
        # ... populate album.tracks ...
        return album

    async def get_track_metadata(self, track_id: str) -> TrackMetadata:
        # Call Tidal API for track info
        track = TrackMetadata(id=track_id, title="Track Title", artist="Artist",
album="Album", track_number=1)
        return track

    async def download_tracks(self, tracks: list[TrackMetadata], dest_dir, quality: str):
        # Tidal might require getting stream URLs or offline content via API.
        # This could involve requesting a stream URL for each track (possibly time-
limited).
        # Pseudo-code:
        tasks = []
        async with aiohttp.ClientSession() as session:
            for track in tracks:
                # We would use the Tidal API/SDK to get a stream URL or file:
                # e.g., url = tidal_api.get_flac_url(track.id)
                url = None  # placeholder
                if url:
                    filename = f"{track.track_number:02d} - {track.title}.flac"
                    filepath = str(dest_dir / filename)
                    tasks.append(download_file(session, url, filepath))
            await asyncio.gather(*tasks)
```

Due to the complexity of Tidal's API, the above is left as high-level pseudo-code. An actual implementation might use the unofficial `tidalapi` library, which can login and fetch stream URLs (with decryption keys if needed). The key is that the plugin would provide similar methods: authenticate (set up session), get metadata, and download using aiohttp.

Apple Music Plugin (`flaccid/plugins/apple.py`):

Apple is metadata-only for our purposes (we can't download from Apple Music). It implements `MetadataProviderPlugin` instead of `MusicServicePlugin`.

```
# flaccid/plugins/apple.py

import requests  # using requests for simplicity since Apple search is single call
```

```
from flaccid.plugins.base import MetadataProviderPlugin, AlbumMetadata, TrackMetadata
from flaccid.core import config

class AppleMusicPlugin(MetadataProviderPlugin):
    def __init__(self):
        # We might retrieve tokens if needed
        import keyring
        self.dev_token = keyring.get_password("flaccid_apple", "developer_token")
        self.music_token = keyring.get_password("flaccid_apple", "user_token")

    def search_album(self, query: str) -> AlbumMetadata | None:
        # Use Apple iTunes Search API (no auth required, limited info)
        params = {"term": query, "entity": "album", "limit": 1}
        resp = requests.get("https://itunes.apple.com/search", params=params)
        results = resp.json().get("results", [])
        if not results:
            return None
        album_info = results[0]
        # Construct AlbumMetadata (note: iTunes API uses different field names)
        album = AlbumMetadata(
            id=str(album_info.get("collectionId")),
            title=album_info.get("collectionName"),
            artist=album_info.get("artistName"),
            year=int(album_info.get("releaseDate", "0")[:4]) if
album_info.get("releaseDate") else 0,
            cover_url=album_info.get("artworkUrl100")  # 100x100 image, can modify URL for
higher res
        )
        # We need track list which the search API doesn't provide; use lookup API:
        collection_id = album.id
        return self.get_album_metadata(collection_id)

    def get_album_metadata(self, album_id: str) -> AlbumMetadata | None:
        url = f"https://itunes.apple.com/lookup?id={album_id}&entity=song"
        resp = requests.get(url)
        data = resp.json().get("results", [])
        if not data or len(data) < 2:
            return None
        # First element is album info, rest are tracks
        album_info = data[0]
        album = AlbumMetadata(
            id=str(album_info.get("collectionId")),
            title=album_info.get("collectionName"),
            artist=album_info.get("artistName"),
            year=int(album_info.get("releaseDate", "0")[:4]) if
album_info.get("releaseDate") else 0,
            cover_url=album_info.get("artworkUrl100")
        )
        tracks = []
        for item in data[1:]:
            if item.get("wrapperType") == "track":
                t = TrackMetadata(
```

```
                    id=str(item.get("trackId")),
                    title=item.get("trackName"),
                    artist=item.get("artistName"),
                    album=album.title,
                    track_number=item.get("trackNumber", 0),
                    disc_number=item.get("discNumber", 1),
                    duration=item.get("trackTimeMillis", 0) / 1000.0  # convert ms to
seconds
                )
                tracks.append(t)
        album.tracks = tracks
        return album
```

Highlights of AppleMusicPlugin:

- It uses the public iTunes Search API and Lookup API (these don't require our stored tokens, as they are public endpoints). If more detailed data from Apple Music (beyond iTunes store info) was needed, we'd use the developer token via Apple Music API. But to keep it simple, iTunes data is sufficient for tagging.
- `search_album(query)`: calls the search API with term and entity=album, retrieves the first result, then calls `get_album_metadata` on that album's collectionId.
- `get_album_metadata(album_id)`: calls the iTunes lookup API to get album and track listings. It then builds an AlbumMetadata with track list.
- The cover_url from iTunes (artworkUrl100) is a link to 100x100 image. There's a known trick: those URLs can often be modified to get a larger image (by replacing `100x100` with e.g. `600x600`). We might do that to get a high-res cover for embedding.
- We do not implement any download functionality here, as Apple tracks cannot be downloaded via this method. If a user tries `fla get apple`, we haven't provided such a command.

Lyrics Plugin (`flaccid/plugins/lyrics.py`):

For lyrics, one could use an API like Genius. This would likely require an API token (which could be stored in config or .secrets). Our lyrics plugin interface is simple: get lyrics by artist and title.

A skeletal example (assuming a hypothetical lyrics API or using a web scrape):

```
# flaccid/plugins/lyrics.py

import requests
from flaccid.plugins.base import LyricsProviderPlugin

class DummyLyricsPlugin(LyricsProviderPlugin):
    """Example lyrics plugin that returns dummy lyrics."""
    def get_lyrics(self, artist: str, title: str) -> str | None:
        # In real implementation, call an API.
        # For example, using lyrics.ovh (a free lyrics API) as a placeholder:
        try:
            resp = requests.get(f"https://api.lyrics.ovh/v1/{artist}/{title}")
            data = resp.json()
            return data.get("lyrics")
        except Exception:
```

```
            return None
```

We might integrate a real provider by using an API key (for Genius, we'd use their REST API with OAuth token). The concept remains: given an artist and title, return lyrics text or None.

Plugin Discovery and Registration

In our CLI commands, we directly imported specific plugins (qobuz, tidal, apple). This is straightforward but adding a new service would require code changes in the CLI. To make the system more extensible without modifying CLI code for each new plugin, we could implement a plugin registry or dynamic loading:

- Maintain a mapping (dictionary) of service name to plugin class or instance. E.g., `plugins = {"qobuz": QobuzPlugin, "tidal": TidalPlugin, "apple": AppleMusicPlugin}`. The `get` and `tag` commands could then look up the appropriate plugin by name. However, Typer's design (separate subcommands) makes dynamic addition slightly tricky. We can still programmatically add Typer commands by iterating over a plugin registry at startup.
- Alternatively, use Python entry points (set in pyproject) so external packages can register plugins. For example, if someone created a `flaccid-spotify` plugin package, it could declare an entry_point in group `flaccid.plugins` and we could auto-load it.
- In this design, given the known set of services, we kept things simple. But we include developer notes that to add a new provider, one should implement the appropriate class and then update CLI and possibly config accordingly.

Combining Providers (Cascade)

The plugin system also allows using multiple sources for the same data. For instance, one might combine Qobuz and Apple data: Qobuz for technical metadata (bit depth, etc.) and Apple for canonical track names or genres. Our design for cascade could be implemented in `core/metadata.py` by using more than one plugin's output:

For example, if `fla tag qobuz` is run but Apple integration is enabled for enriching metadata, the code could fetch from Qobuz (primary) and then fetch from Apple and merge fields. However, this can complicate the CLI usage (we didn't explicitly design a command to tag from multiple sources at once). A more straightforward approach: the cascade refers to adding lyrics and artwork after getting the primary metadata from one source.

Thus, the plugin system is flexible but currently one command uses one primary provider (plus possibly the lyrics provider automatically).

Downloading and Async Operations (Downloader Module)

Downloading large audio files efficiently is a crucial part of FLACCID's functionality for the `get` commands. We utilize **aiohttp** for asynchronous HTTP requests and **Rich** for a user-friendly progress display.

The `flaccid/core/downloader.py` module provides helper functions to download files and possibly to report progress:

```
# flaccid/core/downloader.py
```

```
import aiohttp
import asyncio
from rich.progress import Progress, BarColumn, DownloadColumn, TimeRemainingColumn

async def download_file(session: aiohttp.ClientSession, url: str, dest_path: str):
    """Download a file from the given URL to dest_path using the provided session."""
    async with session.get(url) as resp:
        resp.raise_for_status()
        total = int(resp.headers.get('Content-Length', 0))
        # Setup progress bar for this download
        progress = Progress(BarColumn(), "[progress.percentage]{task.percentage:>3.1f}% ",
DownloadColumn(), TimeRemainingColumn())
        task_id = progress.add_task(f"Downloading", total=total)
        # Open file for writing in binary mode
        with open(dest_path, "wb") as f:
            # Iterate over response data chunks
            async for chunk in resp.content.iter_chunked(1024 * 64):
                if chunk:
                    f.write(chunk)
                    progress.update(task_id, advance=len(chunk))
        progress.update(task_id, completed=total)
        progress.stop()
```

What this does:

- It uses an aiohttp `ClientSession` passed in (to reuse connections).
- It reads the `Content-Length` header to know total size (for progress bar).
- It sets up a **Rich Progress** object with a progress bar, percentage, download speed (via `DownloadColumn`), and estimated time remaining (`TimeRemainingColumn`).
- As it reads chunks of the response, it writes to file and updates the progress bar accordingly.
- Once done, it marks the task complete and stops the progress display.

This function can be called for each file. In the Qobuz plugin, we created multiple tasks of `download_file` for concurrent downloads. However, we must be careful: if we start multiple progress bars concurrently, they will all print to the console. Rich's Progress can handle multiple tasks in one Progress instance, or we could create one Progress per file but need to manage how they are displayed. A simpler approach is to sequentially download tracks with one progress bar at a time, but concurrency was a goal for speed.

To handle concurrency with a combined progress display, we could do:

```
async def download_album(tracks, dest_dir):
    with Progress(BarColumn(), "[progress.percentage]{task.percentage:>3.1f}%",
DownloadColumn(), TimeRemainingColumn(), expand=True) as progress:
        tasks = []
        for track in tracks:
            # Each track gets a task and its own progress bar line
            task_id = progress.add_task(f"[white]{track.title}", total=track.size or 0)
            tasks.append(_download_track(track, dest_dir, progress, task_id))
        await asyncio.gather(*tasks)

async def _download_track(track, dest_dir, progress, task_id):
    # like download_file but updates the passed progress with given task_id
```

However, this complicates the code. For clarity in the handbook, the simpler `download_file` with its own Progress (so each download prints a progress bar sequentially) is shown. If concurrently downloading, the outputs might intermix but Rich may handle it by rendering multiple bars if the same Progress context is used.

We can note that concurrently downloading an album's tracks can drastically reduce total time (especially if network latency is a factor), and the user will see multiple progress bars at once or one after another depending on implementation details.

Rate limiting and API courtesy: If needed, one could implement delays or limits to not overwhelm the service (though for personal use, downloading an album's tracks concurrently is usually fine).

Quality selection: In Qobuz, we demonstrated choosing a format based on `quality` string. In a real scenario, we might want to map "lossless" to the best available lossless quality the user is entitled to (16-bit vs 24-bit). Qobuz's API returns available formats for a track; we could choose the highest FLAC quality automatically if `quality` is set to "lossless". The user might explicitly request hi-res with a flag. For Tidal, if the user has HiFi Plus, we might download FLAC (Tidal now uses FLAC for HiFi), etc.

Error handling: The download function should handle errors (like network issues) gracefully. In the code, `resp.raise_for_status()` will throw for HTTP errors (like 404), which we might catch and log. For brevity, not shown.

Metadata Tagging and Cascade Logic

Once music is downloaded or identified, tagging the files with correct metadata is essential. FLACCID uses **Mutagen** to write FLAC tags (Vorbis comments) and embed album art. Additionally, a **cascade**approach is used to enrich metadata from multiple sources: for example, adding lyrics via a lyrics plugin, or filling gaps in one provider's data with another's.

Metadata Cascade and Enrichment

The concept of a *metadata cascade* is to layer multiple metadata inputs in a priority order. In our context, the typical cascade when tagging might be:

1. **Primary Source**: e.g., Qobuz or Apple provides the core tags (title, artist, album, track number, etc., plus possibly album art).

2. **Secondary Source**: (Optional) Another provider could fill in missing fields or provide additional info (for instance, if Qobuz lacks genre or album artist, Apple's data might have it).
3. **Lyrics Provider**: If enabled, fetches lyrics for each track.
4. **Existing File Metadata**: If we choose not to overwrite certain fields already present (for instance, maybe a user's custom comment or a specific tag), we could choose to preserve them unless the new data has something.
5. **User Overrides**: A user might specify manual overrides (not in our CLI currently, but conceptually a config could force certain tags).

In the current implementation, we focus on using one primary source plus lyrics. But we design the `apply_album_metadata` function to be flexible:

```
# flaccid/core/metadata.py

from mutagen.flac import FLAC, Picture
from flaccid.plugins import lyrics as lyrics_plugin

def apply_album_metadata(folder_path: str, album_meta) -> None:
    """
    Apply metadata from AlbumMetadata to all FLAC files in the given folder.
    Files are matched to tracks by track number (and disc number if applicable).
    """
    # Sort files and tracks by track number for alignment
    import glob, os
    flac_files = sorted(glob.glob(os.path.join(folder_path, "*.flac")))
    tracks = sorted(album_meta.tracks, key=lambda t: (t.disc_number, t.track_number))
    if len(flac_files) != len(tracks):
        print("Warning: number of FLAC files does not match number of tracks in
metadata.")

    # Prepare cover art image if available
    cover_data = None
    if album_meta.cover_url:
        try:
            import requests
            resp = requests.get(album_meta.cover_url)
            cover_data = resp.content if resp.status_code == 200 else None
        except Exception:
            cover_data = None

    # Lyrics provider initialization
    lyr = None
    try:
        lyr = lyrics_plugin.DummyLyricsPlugin()
    except Exception:
        lyr = None

    # Iterate through files and corresponding metadata
    for i, file_path in enumerate(flac_files):
        audio = FLAC(file_path)
```

```
        if i < len(tracks):
            track = tracks[i]
        else:
            track = None
        if track:
            # Basic tags from metadata
            audio["title"] = track.title
            audio["artist"] = track.artist
            audio["album"] = track.album or album_meta.title
            audio["albumartist"] = album_meta.artist
            audio["tracknumber"] = str(track.track_number)
            audio["discnumber"] = str(track.disc_number)
            if album_meta.year:
                audio["date"] = str(album_meta.year)
            # Additional tags if available
            if hasattr(track, "genre") and track.genre:
                audio["genre"] = track.genre
            if hasattr(track, "isrc") and track.isrc:
                audio["isrc"] = track.isrc

            # Lyrics integration
            if lyr:
                lyrics_text = lyr.get_lyrics(track.artist, track.title)
                if lyrics_text:
                    audio["lyrics"] = lyrics_text

            # Embed cover art once per album (for the first track, or all tracks if
desired)
            if cover_data:
                # Remove existing pictures to avoid duplicates
                audio.clear_pictures()
                pic = Picture()
                pic.data = cover_data
                pic.type = 3  # front cover
                pic.desc = "Cover"
                pic.mime = "image/jpeg"  # assume JPEG; could detect from content
                audio.add_picture(pic)  # Embed album art:contentReference[oaicite:8]
{index=8}:contentReference[oaicite:9]{index=9}
        else:
            print(f"No metadata for file {file_path}, skipping tagging.")

        audio.save()
```

Explanation:

- We gather all `.flac` files in the folder and sort them. We sort tracks from metadata by disc and track number. We assume one disc or proper numbering; if multiple discs, the sorting should align with file naming if they're named with disc numbers in separate folders or all together.
- If the count of files and tracks differ, we warn the user. (We proceed anyway, tagging as many as possible.)

- **Album Art**: If `album_meta.cover_url` is provided, we attempt to download the image (using `requests` synchronously here for simplicity). We store the image bytes in `cover_data`. This could be None if download fails or not available.
- **Lyrics**: We attempt to instantiate a lyrics plugin. If no lyrics plugin is configured or fails, we set `lyr = None`. In practice, this could be configurable (maybe an option to enable/disable lyrics fetch).
- For each file and corresponding track metadata:
    - We load the FLAC file with Mutagen's `FLAC` class.
    - We write standard tags: title, artist, album, albumartist, tracknumber, discnumber, date (year). These fields correspond to Vorbis comments in FLAC. (Mutagen handles the mapping; e.g., `audio["date"]` will be stored as `DATE=`).
    - We check for extra fields like genre or ISRC if our metadata source provided them. (Our TrackMetadata class didn't include genre or isrc in earlier code, but Apple's API could provide genre via the album info. Qobuz might provide an ISRC per track. We illustrate that extra info could be set similarly.)
    - **Lyrics**: If a lyrics plugin is available, we call `get_lyrics(artist, title)`. If lyrics are found, we add them under the "lyrics" tag (which many players read for displaying lyrics).
    - **Album Art Embedding**: We use Mutagen's Picture class. We clear existing pictures first to avoid duplicates (Mutagen's `clear_pictures()` removes all embedded images for FLAC). Then create a Picture, set its `type` to 3 (cover front), set a description, MIME type (we assume JPEG for simplicity; ideally detect from bytes or Content-Type). We assign the raw image data to `pic.data`. Finally, `audio.add_picture(pic)` embeds the imagestackoverflow.comstackoverflow.com. In this implementation, we embed the same cover in each track's metadata. Alternatively, we could embed only in the first track to save space, but most players expect each file to have cover art, so we do all.
    - If no track metadata is available for a file (which could happen if files > metadata tracks), we skip tagging it.
    - We save the file to write changes to disk.

Mutagen takes care of writing the Vorbis comment block and picture block properly into the FLAC file. After this, the FLAC files have standardized tags.

Ensuring Correct Matching: We assumed that sorting by track number is enough to match files to metadata. This is often true if files are named in track order and no extras. However, if the files were not in order or missing numbers, a safer approach might be to match by existing tags (like matching titles or using AcoustID fingerprinting). Given a controlled use-case (just downloaded files or well-organized library), the assumption is acceptable, but developers might consider improvements.

Multiple Sources: In this function, we could incorporate a secondary metadata source by checking if certain fields are missing. For example, if `track.genre` is empty and we have another source's data, we could fill it. Implementation might involve merging two AlbumMetadata objects. As of now, we don't demonstrate that to keep it straightforward.

The cascade logic here primarily demonstrates combining the main provider's metadata with an auxiliary lyrics provider, and preserving some album-level info (like album artist, year) for all tracks.

Library Scanning and Indexing (Library Module)

Managing the local library involves scanning the filesystem for audio files, reading their metadata, and storing entries in a database. It also includes responding to changes (new files, deletions, changes) and verifying the integrity of files.

The `flaccid/core/library.py` module provides functions for scanning and indexing:

```
# flaccid/core/library.py

import os
from pathlib import Path
from mutagen.flac import FLAC
from flaccid.core import database

def scan_library(root: str, verify: bool = False):
    """
    Incrementally scan the library root for changes and update the database.
    New files are added, modified files are updated, deleted files are removed.
    """
    root_path = Path(root).expanduser()
    # Get list of all FLAC files in library
    files_on_disk = [p for p in root_path.rglob("*.flac")]
    # Query database for existing entries (paths)
    db_tracks = database.get_all_tracks()  # returns list of Track ORM objects
    db_paths = {Path(track.path) for track in db_tracks}

    files_set = set(files_on_disk)
    # New files: in files_set but not in db_paths
    new_files = files_set - db_paths
    # Deleted files: in db_paths but not in files_set
    deleted_files = db_paths - files_set
    # Possible modified files: intersection, but we'll verify by timestamp or hash
    modified_files = []
    for track in db_tracks:
        p = Path(track.path)
        if p in files_set:
            # Compare last modified time or size between DB record and actual file
            if p.stat().st_mtime > track.last_modified:
                modified_files.append(p)

    # Process deletions
    for p in deleted_files:
        database.remove_track(p)  # remove track (and possibly album if empty) from DB

    # Process new files
    for p in new_files:
        try:
            audio = FLAC(p)
        except Exception as e:
```

```
            print(f"Warning: {p} is not a valid FLAC file or could not be read.
Skipping.")
            continue
        tags = audio.tags
        track_info = {
            "title": tags.get("title", [""])[0],
            "artist": tags.get("artist", [""])[0],
            "album": tags.get("album", [""])[0],
            "albumartist": tags.get("albumartist", [""])[0] if "albumartist" in tags else
tags.get("artist", [""])[0],
            "tracknumber": int(tags.get("tracknumber", [0])[0] or 0),
            "discnumber": int(tags.get("discnumber", [1])[0] or 1),
            "year": int(tags.get("date", [0])[0][:4] or 0) if "date" in tags else 0,
            "genre": tags.get("genre", [""])[0] if "genre" in tags else "",
        }
        # Compute duration and maybe MD5 if verify
        info = audio.info
        duration = info.length
        md5 = None
        if verify and hasattr(info, "md5_signature"):
            md5 = info.md5_signature  # FLAC streaminfo MD5 of raw audio
        database.add_track(path=str(p), metadata=track_info, duration=duration, md5=md5)

    # Process modified files
    for p in modified_files:
        try:
            audio = FLAC(p)
        except Exception:
            continue
        tags = audio.tags
        track = database.get_track(p)
        if not track:
            continue
        track.title = tags.get("title", [""])[0]
        track.artist = tags.get("artist", [""])[0]
        track.album = tags.get("album", [""])[0]
        track.albumartist = tags.get("albumartist", [""])[0] if "albumartist" in tags else
track.artist
        track.tracknumber = int(tags.get("tracknumber", [0])[0] or 0)
        track.discnumber = int(tags.get("discnumber", [1])[0] or 1)
        if "date" in tags:
            track.year = int(tags.get("date", ["0"])[0][:4] or 0)
        if "genre" in tags:
            track.genre = tags.get("genre", [""])[0]
        track.duration = audio.info.length
        if verify and hasattr(audio.info, "md5_signature"):
            track.md5 = audio.info.md5_signature
        database.update_track(track)
```

And functions for full indexing and watching:

```
def index_library(root: str, verify: bool = False):
    """
    Full indexing: go through all files and ensure database matches exactly.
    Potentially slower than scan, but thorough.
    """
    # Simplest approach: reset DB and add all files (if rebuild is intended).
    scan_library(root, verify=verify)

def reset_database():
    database.reset()  # drop and recreate tables

def watch_library(root: str, verify: bool = False):
    """
    Watch the library directory for changes (using watchdog) and update DB in real-time.
    """
    from watchdog.observers import Observer
    from watchdog.events import FileSystemEventHandler

    class LibraryHandler(FileSystemEventHandler):
        def on_created(self, event):
            if not event.is_directory and event.src_path.endswith(".flac"):
                print(f"Detected new file: {event.src_path}")
                try:
                    audio = FLAC(event.src_path)
                    tags = audio.tags
                    track_info = { ... }  # similar to above extraction
                    duration = audio.info.length
                    md5 = audio.info.md5_signature if verify and hasattr(audio.info,
"md5_signature") else None
                    database.add_track(path=event.src_path, metadata=track_info,
duration=duration, md5=md5)
                    print(f"Added to library: {track_info.get('title')} -
{track_info.get('album')}")
                except Exception:
                    return

        def on_deleted(self, event):
            if not event.is_directory and event.src_path.endswith(".flac"):
                print(f"Detected deletion: {event.src_path}")
                database.remove_track(event.src_path)

        def on_modified(self, event):
            if not event.is_directory and event.src_path.endswith(".flac"):
                print(f"Detected modification: {event.src_path}")
                # We can treat modifications similar to created (re-read tags and update)
                try:
                    audio = FLAC(event.src_path)
                    tags = audio.tags
                    track = database.get_track(event.src_path)
                    if not track:
                        return
                    track.title = tags.get("title", [""])[0]
```

```
                    # ... update other fields ...
                    track.duration = audio.info.length
                    if verify and hasattr(audio.info, "md5_signature"):
                        track.md5 = audio.info.md5_signature
                    database.update_track(track)
                except Exception:
                    return

    path = str(Path(root).expanduser())
    event_handler = LibraryHandler()
    observer = Observer()
    observer.schedule(event_handler, path, recursive=True)
    observer.start()
    try:
        # Run indefinitely
        while True:
            import time
            time.sleep(1)
    finally:
        observer.stop()
        observer.join()
```

What the library module is doing:

- **scan_library**:
  - It finds all `.flac` files (using `rglob` for recursive search).
  - Compares with the database's known tracks:
    - New files: add to DB (reading tags via Mutagen FLAC).
    - Deleted files: remove from DB.
    - Modified files: we check by last modified timestamp (or could use file size or a hash). If a file's mtime is newer than what's stored in DB, we consider it modified and update the DB entry.
  - When adding, we extract basic tags from the file using `audio.tags` (Mutagen returns a dict-like object for Vorbis comments). We create a `track_info` dict and pass it to `database.add_track`, along with duration and MD5 if verify is True.
  - For updating modified files, we retrieve the track from DB (`database.get_track`) and update its fields, then call `database.update_track`.
  - The verify option: if true, we attempt to get `audio.info.md5_signature`. For FLAC, Mutagen's FLAC info typically includes the MD5 signature of the uncompressed audio stream (this is stored in the FLAC file's header when encoded). By storing this, we can later verify integrity by comparing with a newly calculated MD5 or using `flac` command-line. Here we just store it as reference. If a file's MD5 changes, it indicates the audio data changed (which is unusual unless file was transcoded or corrupted).
- **index_library**: We simplified it to just call scan_library. In a more advanced design, `index_library` might rebuild from scratch (especially if `rebuild=True` was passed, which in our CLI triggers `reset_database()` first). The distinction is minor in code; the CLI logic handled the rebuild flag. The purpose is to provide a full traversal that ensures DB is exactly in sync with disk.

- **reset_database**: Delegated to the database module to drop and create tables anew.
- **watch_library**:
  - Uses Watchdog to set up an Observer on the library path.
  - We define a nested `LibraryHandler` class extending `FileSystemEventHandler` with handlers for create, delete, modify events.
  - On created: if a new FLAC file is added, we parse it and add to DB (similar to scan's new file logic).
  - On deleted: remove from DB.
  - On modified: re-read tags and update (or potentially handle rename as a delete+create if a file is moved).
  - We print messages to console when events occur to inform the user (these could be toggled or logged instead).
  - The observer is started and runs until interrupted. The `lib scan --watch` command is intended to call this and then block. In the CLI command, we catch KeyboardInterrupt to stop the observer and exit gracefully.

Database Operations: The library module calls functions in the `database` module (like `add_track`, `remove_track`, etc.). We will define those next. The division is for clarity: `library.py` handles file system concerns, and `database.py` manages persistence.

By using Watchdog, we enable a near-real-time sync with the file system, which is very useful if the user is ripping CDs, purchasing new FLACs and dropping them in the folder, or editing tags with another tool – the database will update accordingly (depending on how modifications are detected).

Integrity Verification: When `--verify` is used, we store the FLAC audio MD5. We could also actually verify the file by decoding it partially. If deeper verification was needed, one could run `flac -t` on each file (flac test), but that's slow. Instead, storing the MD5 from the file's header is a quick consistency check: if the file is later corrupted, one could attempt to decode and compare MD5, but that's beyond scope here.

Now, let's define the database schema and operations.

Database Schema and Management (Database Module)

We use **SQLAlchemy** (the latest version, v2 style) to create a local SQLite database for the library. The database stores tracks, and implicitly albums and artists. We can either use separate tables for Albums and Artists or store everything in one tracks table and derive album/artist info via queries. A normalized schema is cleaner and allows capturing album-level data (like cover art path, year, etc.) and artist info only once. Here's a possible schema:

```
# flaccid/core/database.py

from sqlalchemy import create_engine, Column, String, Integer, Float, ForeignKey, select
from sqlalchemy.orm import declarative_base, Session, relationship

Base = declarative_base()
engine = create_engine("sqlite:///flaccid_library.db", future=True, echo=False)  # SQLite
file in current directory (or use config path)
session = Session(engine)
```

```
class Album(Base):
    __tablename__ = "albums"
    id = Column(Integer, primary_key=True, autoincrement=True)
    title = Column(String)
    artist = Column(String)
    year = Column(Integer)
    genre = Column(String)
    cover_path = Column(String, nullable=True)  # path to cover image file (if we choose
to save covers separately)

    tracks = relationship("Track", back_populates="album_obj", cascade="all, delete-
orphan")

class Track(Base):
    __tablename__ = "tracks"
    id = Column(Integer, primary_key=True, autoincrement=True)
    album_id = Column(Integer, ForeignKey("albums.id"))
    title = Column(String)
    artist = Column(String)
    albumartist = Column(String)
    tracknumber = Column(Integer)
    discnumber = Column(Integer)
    year = Column(Integer)
    genre = Column(String)
    duration = Column(Float)
    path = Column(String, unique=True)
    last_modified = Column(Float)
    md5 = Column(String, nullable=True)

    album_obj = relationship("Album", back_populates="tracks")

# Create tables if not exist
Base.metadata.create_all(engine)

def reset():
    Base.metadata.drop_all(engine)
    Base.metadata.create_all(engine)

def add_track(path: str, metadata: dict, duration: float, md5: str | None = None):
    # Check if album exists or create
    album_title = metadata.get("album", "")
    album_artist = metadata.get("albumartist", metadata.get("artist", ""))
    album = session.query(Album).filter_by(title=album_title,
artist=album_artist).one_or_none()
    if not album:
        album = Album(title=album_title, artist=album_artist, year=metadata.get("year",
0), genre=metadata.get("genre", ""))
        session.add(album)
        session.flush()  # flush to get album.id if needed
    track = Track(
        title=metadata.get("title", ""),
```

```
        artist=metadata.get("artist", ""),
        albumartist=album_artist,
        tracknumber=metadata.get("tracknumber", 0),
        discnumber=metadata.get("discnumber", 1),
        year=metadata.get("year", 0),
        genre=metadata.get("genre", ""),
        duration=duration,
        path=path,
        last_modified=os.path.getmtime(path),
        md5=md5,
        album_obj=album
    )
    session.add(track)
    session.commit()

def get_all_tracks():
    return session.query(Track).all()

def get_track(path: str):
    return session.query(Track).filter_by(path=str(path)).one_or_none()

def update_track(track: Track):
    # Assuming track is a Track object already queried/attached to session
    track.last_modified = os.path.getmtime(track.path)
    session.commit()

def remove_track(path: str):
    track = session.query(Track).filter_by(path=str(path)).one_or_none()
    if track:
        session.delete(track)
        # Optionally delete album if no more tracks
        if track.album_obj and len(track.album_obj.tracks) == 1:
            session.delete(track.album_obj)
        session.commit()
```

Schema explanation:

- **Album** table: Stores album title, artist (album artist), year, genre (we assume one genre; if multiple, could handle differently), and perhaps a cover image path. We can also store an external service album ID if needed for reference, but not included here.
- **Track** table: Stores track title, track artist, albumartist, track number, disc number, year, genre, duration (in seconds), file path (must be unique), last_modified timestamp, and MD5. It has a foreign key linking to Album. Relationship `album_obj` allows accessing the album from a track and vice versa.
- We create the tables using `Base.metadata.create_all(engine)`. The engine is a SQLite database stored in `flaccid_library.db` in the current working directory. (We might want this in a user config directory. We could adjust the path to `config.settings.library_path/flaccid.db` or similar. Keeping it simple here.)
- `reset()`: Drops all tables (erasing the library) and recreates them. Used in `lib index --rebuild`.

- `add_track()`: Adds a track to the DB. It checks if the album exists (matching by title and album artist). If not, creates a new Album entry. Then creates a Track with given metadata and links it to the album. We use `os.path.getmtime` to store the file's last modified time as `last_modified`. We commit at the end.
- `get_all_tracks()`: Returns all Track entries (used in scanning to compare).
- `get_track(path)`: Find a track by path.
- `update_track(track)`: We assume the track object's fields have been modified by the caller (as in `library.scan_library` when updating tags). We simply update the last_modified to current file time and commit. (Alternatively, we could merge a detached object, but since we hold the object from session, we can commit directly.)
- `remove_track(path)`: Deletes a track by path. Also checks if its album now has zero tracks, and deletes the album if so (prevent orphan albums). Commits the transaction.

SQLAlchemy provides a high-level ORM interface. We run operations in a session. For simplicity, we created a single global session. In a larger application, we might manage sessions differently (especially if doing multi-threading or long-running processes), but here it's fine.

Threading Note: Watchdog callbacks occur in a separate thread (the Observer thread). Our usage of a global Session from multiple threads is not thread-safe. In a production scenario, we'd need to ensure DB access is done in the main thread or use a session per thread with a thread-safe queue. For simplicity, one might avoid doing heavy DB ops directly in the Watchdog thread - perhaps just mark something and let the main thread handle it. Given the complexity, our example ignores this threading issue. Developers should be aware and possibly adjust by using SQLAlchemy's thread-local sessions or calling back to main thread for DB writes.

Cross-Platform DB Path: The engine string "sqlite:///flaccid_library.db" places the DB in current directory. We might instead want it in a standard location. For example, use `config.settings.library_path` or a subdir for database. Or use `platformdirs` to get a config or data folder:

```
from platformdirs import user_data_dir
db_path = Path(user_data_dir("FLACCID", appauthor=False)) / "library.db"
engine = create_engine(f"sqlite:///{db_path}", future=True)
```

This would store it in a proper user data directory (like `%APPDATA%\FLACCID\library.db` on Windows, `~/.local/share/FLACCID/library.db` on Linux, etc.). For clarity we didn't include platformdirs in code, but this is a recommended practice for real apps.

Now the library scanning functions use this DB layer to reflect changes. This database allows us to answer questions like "do I have this album already?", "list all tracks by X", "find tracks missing tags", etc. We have not built CLI commands for those, but it could be extended (like `fla lib list` or `fla lib find ...`). The focus here is on maintaining the index.

Testing Strategy and Architecture

A comprehensive test suite ensures that each component of FLACCID works as expected and helps future contributors make changes confidently. We use **Pytest** as the testing framework, and organize tests by module/feature.

Test Environment Setup:

- Tests should run in isolation from the user's real environment. We use temporary directories and in-memory or temporary databases for tests, so as not to modify actual music files or config.
- Use Pytest fixtures to set up and tear down contexts (like creating a temp config or sample files).

Sample Tests:

1. **CLI Command Tests:** We can use Typer's built-in test utilities or Click's CliRunner to invoke CLI commands as if a user typed them, and verify the output or resulting state.

   ```
   from typer.testing import CliRunner
   from flaccid import cli

   runner = CliRunner()

   def test_set_path(tmp_path):
       result = runner.invoke(cli.app, ["set", "path", str(tmp_path)])
       assert result.exit_code == 0
       # The settings should now have the new path
       from flaccid.core import config
       assert Path(config.settings.library_path) == tmp_path
   ```

   This test simulates `fla set path <tempdir>` and then checks that the config was updated.
2. **Plugin Tests:** We can test plugin methods by mocking external API calls. For example, for QobuzPlugin, instead of actually hitting the API (which requires credentials), we can monkeypatch `aiohttp.ClientSession.get` to return a predefined JSON response for known album IDs.
   - Alternatively, use the `responses` library or `aiohttp` test utilities to simulate HTTP responses.
   - Test that `get_album_metadata` returns an AlbumMetadata with expected fields when given a fake JSON.
   - Test that `download_tracks` correctly constructs file names and calls download (we might monkeypatch `download_file` to avoid actually downloading).
   - For AppleMusicPlugin, we can test that search is parsing correctly by injecting sample JSON from iTunes.
3. **Metadata Tagging Tests:**
   - Create a dummy FLAC file for testing (we could either include a small FLAC in test data or use mutagen to create one from scratch). Mutagen can write a FLAC file but creating a valid FLAC from nothing is non-trivial, so better to have a tiny FLAC file in `tests/data`.
   - Use the `apply_album_metadata` function on a temp directory with a copy of that FLAC file. Provide it a small AlbumMetadata with a track. Then reopen the file with Mutagen to assert that the tags were written (e.g., check audio["title"] matches what was given, and that audio.pictures is not empty if cover was provided).
   - If embedding an image, include a small image in test data to supply as cover bytes.
4. **Library Scanning Tests:**

- Create a temporary directory structure with some dummy FLAC files (could copy the same small FLAC multiple times, changing names).
- Run `library.scan_library` on it.
- Query the `database` to see if all files are added.
- Modify a file's tag (using mutagen in the test) and run scan again; check that the DB entry updated.
- Remove a file and run scan; check DB entry removed.
- These tests ensure the logic in scan correctly adds/updates/deletes entries.
- Test `watch_library` in a controlled way: we can trigger the event handlers manually by calling them (since it's hard to actually generate filesystem events in tests reliably). For example, call `LibraryHandler().on_created(FakeEvent(path))` with a dummy event object to simulate and see if database.add_track was called. Or use `tmp_path` with Pytest's capability to watch for changes (could spawn watch_library in a thread and then add a file, etc., but that's more integration testing).

5. **Database Tests:**
   - Using an in-memory SQLite (`sqlite:///:memory:`) for speed. We can override the `engine` in tests to use in-memory by monkeypatching `database.engine` and calling `Base.metadata.create_all` on it.
   - Test that adding a track twice doesn't duplicate album.
   - Test remove_track deletes album when last track removed.
   - Essentially, ensure ORM relationships behave as expected.
6. **Config Tests:**
   - Possibly test that Dynaconf and Pydantic integration works: e.g., set environment variables and see that config picks them up.

Running tests: The project would include a `pytest.ini` or similar to configure test runs. Running `pytest` should find tests in the `tests/` folder. We may also have a coverage report generation to ensure we cover as much as possible.

Continuous Integration: It's advisable to set up CI (GitHub Actions or similar) to run tests on each commit, possibly on multiple OS (to ensure cross-platform compatibility). This way, if a contributor breaks a function on Windows that worked on Linux, tests would catch it.

Example Test Code:

```
# tests/test_metadata.py
import os
from pathlib import Path
from flaccid.core import metadata
from flaccid.plugins.base import AlbumMetadata, TrackMetadata

def create_dummy_flac(tmp_path: Path, title: str, artist: str) -> Path:
    """Helper to create a dummy FLAC file with minimal metadata for testing."""
    file_path = tmp_path / f"{title}.flac"
```

```
    # Copy a template small FLAC or generate one; here assume a template exists in
tests/data/dummy.flac
    import shutil
    shutil.copy("tests/data/dummy.flac", file_path)
    # Tag it with given title and artist
    audio = metadata.FLAC(file_path)
    audio["title"] = title
    audio["artist"] = artist
    audio.save()
    return file_path

def test_apply_album_metadata(tmp_path):
    # Create a dummy flac file
    song_file = create_dummy_flac(tmp_path, "Test Song", "Test Artist")
    # Create AlbumMetadata and TrackMetadata to apply
    track_meta = TrackMetadata(id="1", title="New Title", artist="New Artist", album="New
Album", track_number=1)
    album_meta = AlbumMetadata(id="100", title="New Album", artist="New Artist",
year=2021, cover_url=None, tracks=[track_meta])
    # Apply metadata
    metadata.apply_album_metadata(str(tmp_path), album_meta)
    # Verify the file's tags have changed
    audio = metadata.FLAC(song_file)
    assert audio["title"][0] == "New Title"
    assert audio["artist"][0] == "New Artist"
    assert audio["album"][0] == "New Album"
```

This test demonstrates how we might verify tagging functionality. In practice, we need a small FLAC file in the tests (which could be a few KB of silence encoded in FLAC).

By covering each component with tests, we ensure that the pieces (plugins, tagging, database, CLI commands) work in isolation. Integration tests (like simulating a full `fla get qobuz` flow with a fake Qobuz response, then checking the file is downloaded and tagged) could also be extremely valuable.

Packaging and Deployment Guide

FLACCID is packaged as a standard Python project with modern tooling (PEP 621 / pyproject.toml). Packaging it properly ensures that users and developers can install it easily and the CLI entry point works out of the box.

Project Metadata (pyproject.toml):

The `pyproject.toml` contains project metadata and dependencies. For example:

```
[project]
name = "flaccid"
version = "0.1.0"
description = "FLACCID: Modular FLAC CLI toolkit for music download and tagging"
authors = [
  { name="Your Name", email="you@example.com" }
]
requires-python = ">=3.10"
```

```
dependencies = [
  "typer>=0.7.0",
  "pydantic>=2.0.0",
  "dynaconf>=3.1.0",
  "mutagen>=1.45.0",
  "aiohttp>=3.8.0",
  "keyring>=23.0.0",
  "rich>=13.0.0",
  "SQLAlchemy>=2.0.0",
  "watchdog>=2.1.0",
  "requests>=2.0.0"
]

[project.urls]
Homepage = "https://github.com/yourname/flaccid"

[project.scripts]
fla = "flaccid.cli:app"
```

Notable points:

- We list all required libraries so that `pip install flaccid` will fetch them.
- The console script entry point is defined under `[project.scripts]` as `fla = "flaccid.cli:app"`. This means when installed, a command `fla` will be created, which executes `flaccid.cli:app`. Because `app` is a Typer object, it is callable and will run the CLI (Typer/Click will handle invoking the commands). This way, the user can just run `fla ...` after installation.
- Alternatively, we could have specified an explicit function (like `main`) that calls `app()`, but Typer's `app` itself is designed to be used as an entry point.
- We include `requests` in dependencies because our Apple plugin and some parts use it (though we could have used aiohttp everywhere, `requests` is fine for quick calls).

If using Setuptools instead of PEP 621, we'd have similar definitions in `setup.cfg` or setup.py using entry_points. But pyproject with the `project.scripts` field is modern and simple.

Installing the Package:

- For end users: they can install from PyPI (once published) with `pip install flaccid`. This will make `fla` available in their PATH.
- For development: developers can clone the repository and install in editable mode:

  ```
  git clone https://github.com/yourname/flaccid.git
  cd flaccid
  pip install -e .
  ```

  The `-e` (editable) means any changes to the code reflect immediately in the installed command, which is convenient for development and testing CLI changes. After this, running `fla --help` should show the CLI help.

Editable vs Standard Install: The `pip install -e .` uses the project's `pyproject.toml` and creates links in the environment to the source files. This also installs development dependencies if specified (sometimes people use `[project.optional-dependencies] dev = [...]` for test tools, etc., and then `pip install -e .[dev]` to include them). We could add:

```
[project.optional-dependencies]
dev = ["pytest", "pytest-cov", "tox", "flake8", "mypy"]
```

So that `pip install -e .[dev]` sets up a full dev environment.

Building and Publishing:

- To build distribution artifacts: use a PEP517 builder like `build`:

  ```
  pip install build
  python -m build
  ```

  This will create `dist/flaccid-0.1.0.tar.gz` and `dist/flaccid-0.1.0-py3-none-any.whl`.

- These can be uploaded to PyPI with twine:

  ```
  pip install twine
  twine upload dist/*
  ```

- Once on PyPI, users can install normally.

Cross-platform Notes for Installation:

Our dependencies are pure Python except maybe Watchdog which may have some OS-specific build steps (it uses C extensions for some platforms). `pip` will handle those (wheels for common OSes). On Windows, installing Watchdog might require Build Tools if not wheel, but likely wheels are provided.

Post-install Configuration:

After installation, when `fla` is run for the first time, it will generate a default `settings.toml` in the current directory if not found (depending on how Dynaconf is configured). We might want to have it use a user config directory instead. For example, we could set:

```
settings = Dynaconf(..., settings_files=['~/.config/flaccid/settings.toml',
'~/.config/flaccid/.secrets.toml'])
```

ensuring config is per user. But then we might want an initial config creation step. We did not implement an explicit init, but `set path` and `set auth` effectively create and modify config.

We should document in the README for end users where the config is stored and how to change it. Possibly also provide `fla set path` so they don't have to manually edit the file.

Upgrading:

If a new version is released, `pip install -U flaccid` upgrades it. The user's config and database remain (since those are in separate files).

Packaging for different Python versions:

We require Python 3.10+. Make sure to test on all supported minor versions and OS.

Now, we will provide examples of using the CLI which we partially did earlier, but let's consolidate them.

Example CLI Invocations and Usage

This section demonstrates typical uses of the FLACCID CLI, with example commands and expected outcomes. (Note: These examples assume valid credentials have been set for Qobuz/Tidal as needed.)

- **Authenticating with a service:**

```
$ fla set auth qobuz
Enter Qobuz username/email: user@example.com
Enter Qobuz password: ********
Qobuz credentials stored for user user@example.com.
You can now use 'fla get qobuz ...' or 'fla tag qobuz ...' commands.
```

(Stores Qobuz credentials securely. No output means success, aside from confirmation message.)

- **Setting library path:**

```
$ fla set path ~/Music/FLAC
Library path set to: /home/alice/Music/FLAC
```

(Now the default output directory for downloads and the directory scanned for library commands is `/home/alice/Music/FLAC`.)

- **Downloading a Qobuz album:**

```
$ fla get qobuz --album-id 123456 --quality hi-res
Downloading from Qobuz to /home/alice/Music/FLAC ...
[Downloading Cover] 100% ━━━━━━━━━━━━━━━━━━━━ 500/500 kB • 0:00 • ???  (download
cover if implemented)
Downloading track 1: Song One (FLAC 24-bit)...
Downloading track 2: Song Two (FLAC 24-bit)...
... (progress bars for each track) ...
Qobuz download complete!
```

(The command fetches album metadata, then downloads each track concurrently. After completion, the files are in `~/Music/FLAC/Artist/Album/01 - Song One.flac`, etc., properly tagged.)

- **Downloading a Tidal track:**

```
$ fla get tidal --track-id 987654 --quality lossless -o /tmp
Downloading from Tidal to /tmp ...
Downloading track: My Song (FLAC 16-bit)...
Tidal download complete!
```

(Downloads a single track to `/tmp` directory. `-o` overrides the library path.)

- **Tagging a local album with Qobuz metadata:**

```
$ fla tag qobuz --album-id 123456 "/home/alice/Music/FLAC/Artist/Album"
Fetching metadata from Qobuz for album 123456 ...
Applying tags to files in /home/alice/Music/FLAC/Artist/Album ...
Tagging complete (Qobuz)!
```

(The FLAC files in the specified folder are now updated with official metadata from Qobuz, including album art and lyrics if available.)

- **Tagging a local album with Apple Music metadata:**

```
$ fla tag apple "Artist Name - Album Title" "./Album"
Searching Apple Music for 'Artist Name - Album Title' ...
Found album: Album Title by Artist Name. Applying tags...
Tagging complete (Apple Music)!
```

(The tool searched Apple's catalog for the album and applied those tags. This is useful if, for example, Qobuz metadata was incomplete and Apple's is more detailed.)

- **Scanning the library for new music:**

```
$ fla lib scan
Scanning library at /home/alice/Music/FLAC ...
Initial scan complete!
```

(This will print warnings for any files that couldn't be read, and will update the SQLite database. On first run, it adds all tracks. Subsequent runs, it might output messages like "Added X new tracks, Updated Y tracks, Removed Z tracks" depending on implementation.)

- **Continuous monitoring of library:**

```
$ fla lib scan --watch
Scanning library at /home/alice/Music/FLAC ...
Initial scan complete!
Watching for changes. Press Ctrl+C to stop.
```

Then if the user adds a new file `new.flac` into the library folder, one might see:

```
Detected new file: /home/alice/Music/FLAC/New Artist/New Album/01 - New Song.flac
Added to library: New Song - New Album
```

This happens automatically via Watchdog. The command will continue running until interrupted.

- **Rebuilding the index:**

```
$ fla lib index --rebuild --verify
Rebuilding library index from scratch...
Library indexing complete!
```

(This drops the old database and reindexes everything, verifying each file's integrity. If a file's FLAC MD5 doesn't match or is corrupt, this could log an error or warning in a real scenario.)

- **Getting help:**

```
$ fla --help
Usage: fla [OPTIONS] COMMAND [ARGS]...

  FLACCID CLI - A modular FLAC toolkit

Options:
  --help  Show this message and exit.

Commands:
  get     Download tracks or albums from streaming services
  tag     Tag local files using online metadata
  lib     Manage local music library (scan/index)
  set     Configure credentials and paths
```

And for subcommands:

```
$ fla get --help
Usage: fla get [OPTIONS] COMMAND [ARGS]...

  Download tracks or albums from streaming services

Options:
  --help  Show this message and exit.

Commands:
  qobuz  Download an album or track from Qobuz.
  tidal  Download an album or track from Tidal.
```

These examples illustrate the typical workflow: configure once, then download or tag as needed, and keep the library updated. The output messages and progress bars (not fully shown in text) give the user feedback.

Cross-Platform Compatibility and Path Handling

FLACCID is built with cross-platform support in mind:

- **File Paths**: We use Python's `pathlib.Path` and `os.path` to handle file system paths. This ensures that paths like `~/Music` are expanded to the correct location on each OS, and that separators are handled (e.g., on Windows, `Path("C:/Music")` vs Unix `/home/user/Music`). All internal file handling functions use these abstractions, so we don't hardcode any platform-specific path strings.
- **Default Locations**: By default, on Linux/macOS we might use `~/Music/FLACCID` for library path, whereas on Windows we might use `%USERPROFILE%\Music\FLACCID`. We allow the user to override via `set path` anyway. Using `platformdirs` could further tailor default config and data paths to OS conventions (not implemented above, but recommended).

- **Keyring Backends**: The `keyring` library automatically picks the appropriate backend:
  - Windows: Credential Manager
  - macOS: Keychain
  - Linux: Secret Service (or fallback to plaintext if none, which might require the user to install something like `keyring-secretservice` or have a DE running).
    If keyring fails on a headless environment, one can configure an alternate, but for the majority of desktop users it should just work. We output a friendly message if credentials aren't found or login fails, guiding the user to re-run `set auth`.
- **Watchdog**: The Watchdog observers have different implementations per OS (inotify on Linux, FSEvents on macOS, ReadDirectoryChangesW on Windows). The `watch_library` function doesn't need to change per OS – Watchdog handles the differences. One consideration is case-insensitivity on Windows (if a file moves from A.flac to a.flac, how we handle it – likely as a modify event). But generally, it works.
- **Character Encoding**: Tags are stored in UTF-8 (Vorbis comments in FLAC are UTF-8 by standard). Mutagen handles Unicode strings well. We ensure to use Python's default UTF-8 encoding for file names and console output. Windows console historically had issues with Unicode, but modern Windows 10+ with UTF-8 mode or using Windows Terminal is fine. If a user encounters weird characters, it's more console settings than our code. We rely on Typer/Click, which handles output encoding gracefully.
- **Testing on OS**: We would test at least on one Windows and one Unix platform. Particularly, file path differences (like reserved characters in filenames) should be considered. E.g., if an album or track title has ":" or "?" which are not allowed in Windows filenames, our downloader should sanitize filenames. We didn't explicitly include a sanitization function, but that would be wise to add in `download_tracks` (e.g., replace `:` with `-`).
- **Dependencies**: All listed libraries support Windows, Linux, macOS. Rich and Typer handle ANSI colors and progress bars on each (Rich will degrade gracefully or adjust for Windows). SQLAlchemy with SQLite is fine on all OS. Watchdog requires some C extensions that pip will install (wheel availability is good on common OS). No specific platform-bound code is present.
- **Differences in behavior**: One subtle difference: file paths in the database on Windows will have drive letters and backslashes. Our code treats them as strings, which is okay. Just ensure consistency if comparing. Using Path objects for comparisons normalizes case on Windows? Actually, Path("C:\a") == Path("C:\A") is False by default because it's case-sensitive in comparison even if NTFS isn't. We might want to normalize case or use `.lower()` on paths for DB keys on Windows to avoid duplicates. This level of detail might be too granular for now, but devs should be aware if users report "duplicate track entries" because of case differences.
- **Time zones and Date**: If we use year from tags, there's no TZ issue since it's just year or date string from metadata. If we recorded the full date-time, we'd want to be careful. Not a big issue here.

In summary, FLACCID should function on all major OS with no code changes. We leveraged Python's cross-platform libraries to abstract differences. Developer testing on each platform is recommended to catch any path or encoding quirks.

Troubleshooting and Diagnostics

Even with a robust design, users may encounter issues. Here are common problems and ways to troubleshoot them:

- **Authentication Errors**: If `fla get qobuz` returns an authentication error (e.g., unauthorized or invalid token):
  - Ensure you have run `fla set auth qobuz` with correct credentials. If credentials changed (password update, etc.), run it again to update.
  - Qobuz might also require a valid app_id/secret. Check that these are set in `~/.config/flaccid/settings.toml` (if provided) or in environment. Without a valid app_id, login will fail. You may need to register an app via Qobuz or use an existing app's credentials.
  - For Tidal, ensure your subscription level supports FLAC quality. Some APIs might fail if trying to access HiFi without entitlement.
  - Check if the system keyring is accessible. On Linux, for example, if running headless, the keyring might not be initialized, causing `keyring.get_password` to return None. In such cases, you might use environment variables as a fallback or configure the keyring backend.
- **Download Failures**: If a track or album fails to download:
  - Run with more verbose logging (if we had a `--debug` flag or set an env var). Since we didn't explicitly add a debug flag in code, one could manually insert print statements or use logging. For development, you might run the `get` command in a debugger to see what URL is being fetched.
  - Check internet connectivity and that the service isn't blocking the API usage. (Qobuz might throttle or block if too many requests quickly.)
  - Check that the output path is writable and has enough space.
  - If an error like "Failed to get download URL" appears for Qobuz, it might be that the format_id requested is not available for that track (e.g., trying hi-res on a track that only has 16-bit). You can try a different quality or ensure your account has access.
- **Tagging Issues**: If `fla tag` doesn't change the files or yields warnings:
  - Ensure the folder path is correct and contains FLAC files. The command expects `.flac` files; if they are .mp3 or others, currently the code specifically looked for FLAC. We might extend to other formats but in this toolkit FLAC is focus.
  - If tags applied but not visible in your player, it could be the player caches old tags. Try reading the file tags with Mutagen or another tool to confirm.
  - If album art isn't embedding, the image URL might be broken or internet needed. Or perhaps the image was PNG and some players require JPEG. We default to JPEG mime; if the source was PNG, that could be an issue (Mutagen's Picture with correct mime should handle it though).
  - If lyrics aren't appearing, maybe the lyrics provider didn't find any. You can check manually on the provider's site or try a different provider.
- **Library Database Issues**:
  - If `fla lib scan` doesn't pick up new files, confirm `library_path` is set correctly (`fla set path`).
  - If nothing happens on `--watch`, ensure Watchdog is installed correctly. On Linux, you may need inotify watch limits increased if watching a very large library (OS config).

    - If database appears out of sync or corrupted (perhaps due to a crash), you can rebuild: run `fla lib index --rebuild`.
    - If two instances of FLACCID run concurrently (maybe one watch and one scan), the database might be locked. SQLite has locks if a transaction is open. Avoid multiple simultaneous writes. If needed, our design could be improved with a single writer or using WAL mode. For now, if you see "database is locked" errors, stop other processes and retry.
- **Performance**: Scanning a huge library (thousands of files) can be slow the first time. Using `--verify` makes it even slower. If performance is an issue:
    - Skip `--verify` unless needed.
    - The first scan/index might take time, but subsequent scans are incremental.
    - Ensure you're not scanning a network drive over slow connection; if so, performance is inherently limited by I/O.
    - Watchdog, if watching many files, also has overhead but typically fine for moderate collections.
- **Debug Mode**: We have not explicitly implemented a debug logging flag, but adding one would be useful. One approach is to use Python's logging module and set levels via an option:

    ```
    import logging
    logging.basicConfig(level=logging.DEBUG)
    logging.debug("Detailed info...")
    ```

    In development, you could sprinkle `logging.debug` in critical parts (API responses, etc.) and run with an env var or flag to enable it.
- **Crash / Traceback**: If the CLI crashes with a traceback:
    - The Typer app will normally show a clean error message if we `typer.Exit`. But if an unexpected exception occurs, Typer will show the traceback. For debugging, that's good; for user-friendliness, we might catch exceptions in plugin calls and print a friendly message instead.
    - For example, if Qobuz API is down and returns HTML, our JSON parse might throw. We could catch `aiohttp.ClientError` or JSONDecodeError and inform "API error, try again later".
    - As a developer, look at the traceback, add handling as appropriate. We aim to handle known failure modes gracefully.
- **Diagnostics**: We could add a command like `fla diag` to print out environment info (versions of libraries, OS, whether keyring backend is available, etc.). This isn't implemented but could be a future addition for helping debug user issues.

In summary, many issues can be resolved by checking configuration and using the provided commands properly. The design tries to fail safe (e.g., not overwriting files incorrectly, not leaving partial downloads without at least a message). Future improvements might include logging to a file in config directory for deeper troubleshooting (so user can send a log to developers).

Developer Onboarding and Extension Guide

Welcome to the development side of FLACCID! This section provides guidance for new contributors or developers who want to extend the toolkit's functionality.

Getting Started with the Code

1. **Setup Development Environment**:
    - Clone the repository from version control (e.g., GitHub).
    - Install the package in editable mode with dev dependencies: `pip install -e .[dev]` (this installs the package and tools like pytest, linters if listed).
    - Ensure you have the required services credentials if you plan to test integration (Qobuz, Tidal). For offline development, you can work with dummy data or mock the network interactions.
2. **Project Structure Overview**: As detailed earlier, the code is organized by feature. A new developer should familiarize themselves with:
    - `flaccid/cli.py` to see how commands are wired.
    - `flaccid/commands/` for CLI logic.
    - `flaccid/plugins/` for external service integration.
    - `flaccid/core/` for internal logic (metadata, db, etc.).
    - The `tests/` to see usage examples and to verify changes don't break existing behavior.
3. **Coding Style**:
    - Follow Python best practices (PEP8 style). The project might include a linter (like flake8) and a formatter (like black). If so, run them before committing.
    - Use type hints for new functions/classes. We've used Python 3.10+ syntax (e.g., `str | None`). Type hints improve readability and help static analysis.
    - Write docstrings or comments for any complex logic or any public-facing functions.
4. **Running Tests**:
    - Use `pytest` to run the test suite frequently. Add new tests for any new feature or bugfix.
    - Ensure tests pass on all targeted Python versions. If using tox or CI, run those to be sure.
5. **Version Control**:
    - Work on a feature branch.
    - If adding a new feature, update this developer guide section accordingly if it affects the architecture or usage.
    - Commit messages should be clear. If an issue tracker exists, reference issue IDs.

Adding New Features or Providers

New Streaming/Metadata Provider (e.g., Spotify, Deezer, etc.):

- Create a new plugin module, e.g., `flaccid/plugins/spotify.py`.
- Implement a class (e.g., `SpotifyPlugin`) either deriving from `MusicServicePlugin` if it supports downloads, or `MetadataProviderPlugin` if only metadata.
- Add any needed configuration (API keys, etc.) to `settings.toml` and `.secrets.toml` as appropriate. Document them in the README.

- If the API requires OAuth (like Spotify), you might need to implement an OAuth flow. This could be complex (like opening a browser, etc.). Possibly out-of-scope for CLI; but maybe Spotify can use a user's refresh token from somewhere. Each service will differ in complexity.
- Write tests for the plugin using dummy data or actual API if keys are available (be mindful not to hardcode secrets in tests).
- Integrate the plugin with the CLI:
  - If it's a streaming service, add a subcommand in `get.py` (and in `tag.py` if you want to allow tagging from it).
  - If we had dynamic plugin loading, you'd register it. Currently, manually add similar to how Qobuz/Tidal are done.
  - Also update `set auth` command if credentials needed (like for Spotify, maybe ask for a token or client id/secret).
- Example: adding Deezer would be similar to Qobuz (Deezer has an API that could allow downloading via their user token).
- After adding, test by actually downloading or tagging from that service if possible.

Improving Lyrics Support:

- Replace `DummyLyricsPlugin` with a real implementation. For example, integrate with Genius:
  - Add a dependency on `lyricsgenius` or implement direct API calls. If using an API token, let the user set it in `.secrets.toml` or via `set auth lyrics` possibly.
  - Ensure the lyrics plugin is used in metadata.apply_album_metadata (which we already do).
  - Might allow user to run a separate command like `fla tag lyrics [folder]` to only add lyrics to existing tagged files.
- Write tests for lyrics fetching (maybe using a known song).
- Keep in mind rate limiting for lyrics APIs.

Enhanced Tagging:

- Implement more sophisticated matching for `fla tag` when user doesn't have an ID:
  - E.g., `fla tag qobuz "Album Name by Artist" /folder` could search Qobuz by name (not currently implemented). Qobuz API might have a search endpoint. This could be added to QobuzPlugin (like a method search_album similar to Apple's).
  - Or integrate MusicBrainz as a fallback for metadata (MusicBrainz provides comprehensive metadata and identifiers).
- Implement a true cascade: E.g., a command that uses multiple sources:
  - Possibly `fla tag auto "/folder"` that tries to identify the album via AcoustID or fuzzy matching, then pulls metadata from various sources automatically.
  - This would be a bigger feature involving fingerprinting (could use `pyacoustid` and MusicBrainz).
- If adding such features, keep them modular (maybe a separate module for acoustic ID, etc.) and ensure to update tests.

Database and Library:

- If scaling up, consider using an actual database server (PostgreSQL, etc.) if user desires. We can stick to SQLite for simplicity, but maybe allow in config to specify a different DB URL.
- Add commands to query the library. For instance, `fla lib list artists` or `fla lib search "Beatles"`. This would require building query logic (which is easy with SQLAlchemy queries).
- A `fla lib stats` could show number of tracks, total playtime, etc., by querying DB.
- If implementing these, ensure to format output nicely (maybe using Rich tables).

Performance Tuning:

- If library grows huge (tens of thousands of tracks), scanning might become slow. We can optimize by using database indices, or optimizing the file scanning (maybe maintain a cache of file->last_scanned_time to skip scanning unchanged directories).
- Watchdog performance: large libraries with many events might flood. Possibly implement debouncing or batch updates (e.g., if many files added at once, processing them in one go rather than individually).
- These can be iterative improvements. Add tests for performance sensitive functions if possible (maybe time a scan on a sample structure).

Cross-Platform Testing:

- If contributing code, test on at least your OS. Ideally, run the test suite on all OS via CI.
- Watch out for platform-specific pitfalls (like path normalization as discussed).

Documentation:

- Update the README.md for any new user-facing features or options.
- If adding a plugin, mention how to configure it (e.g., where to get API keys).
- Possibly update this developer handbook if the architecture changes (since it might live in a separate doc, ensure consistency).

Collaborative Development:

- Communicate with other contributors via issues or chat if something big is being changed, to avoid duplicate work.
- Code review: if using GitHub, open a PR and request review. This helps catch issues early and disseminate knowledge of changes among maintainers.

Conclusion for Developers:
FLACCID's rebuild emphasizes modularity and clarity. The code should be approachable: CLI definitions are straightforward with Typer, business logic is isolated, and external integrations are abstracted behind plugins. The test suite should give confidence to refactor or extend parts of the system.

By following this guide, a new developer should be able to set up the environment, run the tool, and dive into improving it. Whether it's adding the latest streaming service or optimizing the tagging flow, the modular structure will support the enhancement.

Happy coding, and enjoy making FLACCID an even more powerful toolkit for music enthusiasts!

FLACCID CLI Toolkit Developer Handbook

Introduction

FLACCID is a modular command-line toolkit designed for managing FLAC audio libraries and integrating with music streaming services. The name "FLACCID" reflects its focus on FLAC audio and a CLI-first design. This toolkit enables users to download high-quality music from services like Qobuz and Tidal, tag their local FLAC files with rich metadata (including album art and lyrics), and maintain a well-organized music library database. The rebuild of FLACCID focuses on a clean architecture with clear module separation, plugin support for multiple providers, and modern Python libraries for robust functionality.

Key Features:

- **Flexible CLI:** A single `fla` command with subcommands for different tasks (downloading, tagging, library management, configuration). Built with **Typer** for intuitive CLI syntax and help.
- **Modular Plugins:** Support for multiple music services (Qobuz, Tidal, Apple Music, etc.) via a plugin system. Easy to add new providers for metadata, streaming, or lyrics.
- **Rich Metadata Tagging:** Uses **Mutagen** to write comprehensive tags to FLAC files, including album art embedding and lyrics. Metadata can be aggregated from multiple sources in a cascade.
- **Asynchronous Downloading:** Leverages **aiohttp** for efficient downloading of tracks (e.g., parallel downloads of an album) with progress display using **Rich**.
- **Robust Configuration:** Configuration managed with **Dynaconf** (TOML-based settings) and validated with **Pydantic** models. Secure credentials are handled via **Keyring** and Dynaconf's secrets support (no plain-text passwords in code or repo).
- **Library Indexing**: Maintains a local SQLite database (via **SQLAlchemy**) of your music library. Can scan directories, index metadata, and perform integrity checks (file existence, hash verification, etc.).
- **Automation**: Optionally uses **Watchdog** to monitor the filesystem for new or changed files, updating the library in real-time.
- **Cross-Platform**: Works on Windows, macOS, and Linux. Paths and storage locations are handled appropriately per OS, and keyring integration uses the OS keychain on each platform.
- **Testing & Quality**: A full test suite (using Pytest) covers modules and CLI commands, ensuring reliability. Developers can easily run tests and extend functionality with confidence.
- **Packaging & Deployment**: Distributed as a Python package with a `pyproject.toml`. Supports standard installation (`pip install .`) and editable installs for development (`pip install -e .`). A guide is provided for publishing and dependency management.

This handbook serves as a developer guide to the architecture and implementation of FLACCID. It details the directory structure, provides full source code listings of each major module, and explains how the components interact. Whether you are a developer looking to extend FLACCID or a power user wanting to understand its internals, this document offers a comprehensive walkthrough.

CLI Architecture Overview

The FLACCID toolkit is accessed via the `fla` command-line interface. The CLI is organized into subcommands that mirror the core functionalities:

- **Download Commands**: `fla get qobuz [options]` and `fla get tidal [options]` – Download albums or tracks from Qobuz or Tidal (given appropriate credentials and subscription). Options allow specifying album/track identifiers, quality levels, output paths, etc.
- **Tagging Commands**: `fla tag qobuz [options]` and `fla tag apple [options]` – Tag local FLAC files using metadata from Qobuz or Apple Music (iTunes). These commands fetch metadata (album details, track titles, artwork, lyrics) from the service and write tags to the files.
- **Library Management**: `fla lib scan [options]` and `fla lib index [options]` – Manage the local music library database. Scanning detects new or changed files in the library path and updates the database (with optional continuous watch). Indexing performs a full rebuild or creation of search indices and verifies file integrity.
- **Configuration**: `fla set auth [service]` and `fla set path [location]` – Configure credentials and paths. The `set auth` command securely stores user API credentials for a given service (Qobuz, Tidal, Apple, etc.), and `set path` defines the base path of the music library or downloads.

Using **Typer** (a library built on Click), these subcommands are implemented as grouped command families. This provides an intuitive syntax (e.g., `fla get qobuz ...`) and auto-generates help documentation. The CLI design allows future expansion; new services or features can be added as new subcommands or options.

Each top-level command group (`get`, `tag`, `lib`, `set`) is implemented as a Typer app, added to the main Typer application. This nested structure improves code organization and mirrors how users conceptualize the tasks. For example, the `get` group contains separate commands for each streaming service, and new services can be included by adding a new function/command in that group.

Example CLI Usage:

- Downloading a Qobuz album by ID: `fla get qobuz --album-id 123456 --quality FLAC --out ~/Music/Downloads`
- Downloading a specific Tidal track: `fla get tidal --track-id 7890 --quality Lossless`
- Tagging a local folder with Qobuz metadata: `fla tag qobuz --album-id 123456 "/path/to/AlbumFolder"`
- Tagging using Apple Music (iTunes) search: `fla tag apple "Artist Name - Album Title" "/path/to/AlbumFolder"`
- Scanning library for new music: `fla lib scan` (optionally `--watch` for continuous monitoring)
- Re-indexing entire library with integrity check: `fla lib index --verify`
- Setting credentials for Qobuz: `fla set auth qobuz` (prompts for username/password or token)

- Setting the music library path: `fla set path ~/Music`

Each command comes with `--help` to display usage instructions (thanks to Typer's automatic help generation). The CLI is designed to be user-friendly while providing power-user options for customization.

Project Structure and Module Layout

The FLACCID project is organized into a Python package with clearly separated modules for commands, plugins, core functionality, and configuration. Below is the directory layout of the project:

```
flaccid/
├── __init__.py
├── cli.py               # CLI entry point using Typer, defines main app and groups
├── commands/
│   ├── __init__.py
│   ├── get.py           # 'fla get' subcommands for various services (Qobuz, Tidal)
│   ├── tag.py           # 'fla tag' subcommands for metadata tagging (Qobuz, Apple)
│   ├── lib.py           # 'fla lib' subcommands for library scan and index
│   └── settings.py      # 'fla set' subcommands for auth and path configuration
├── plugins/
│   ├── __init__.py
│   ├── base.py          # Base classes/interfaces for plugins
│   ├── qobuz.py         # Qobuz plugin implementation (metadata & download)
│   ├── tidal.py         # Tidal plugin implementation (metadata & download)
│   ├── apple.py         # Apple Music plugin implementation (metadata only)
│   └── lyrics.py        # Lyrics plugin (fetch lyrics from external API, e.g., Genius)
├── core/
│   ├── config.py        # Configuration loader (Dynaconf settings and Pydantic models)
│   ├── metadata.py      # Metadata cascade and tagging logic (using Mutagen)
│   ├── downloader.py    # Download utilities (async HTTP, file I/O, Rich progress)
│   ├── library.py       # Library scanning, watching (Watchdog), and DB update logic
│   └── database.py      # Database models and setup (SQLAlchemy ORM for tracks, albums,
etc.)
├── tests/
│   ├── test_cli.py      # Tests for CLI commands and argument parsing
│   ├── test_plugins.py  # Tests for plugin outputs (using stub API responses)
│   ├── test_metadata.py # Tests for tagging logic (e.g., verify tags written correctly)
│   ├── test_library.py  # Tests for scanning and DB operations (using temp directories)
│   └── ...              # (Additional tests as needed per module)
├── pyproject.toml       # Project metadata, dependencies, build system, entry points
└── README.md            # User-facing README (installation, basic usage)
```

Module Overview:

- `flaccid/cli.py`: The main entry point that initializes the Typer app and ties together subcommands from the `commands/` package.
- `flaccid/commands/`: Package containing CLI command implementations for each top-level command group (`get`, `tag`, `lib`, `set`). Each file defines a Typer sub-app and commands corresponding to specific services or actions.

- `flaccid/plugins/`: Houses plugin modules, each implementing integration with a specific service or provider (e.g., Qobuz API, Tidal API, Apple Music, lyrics provider). `base.py` defines abstract interfaces for these plugins.
- `flaccid/core/`: Core functionality that is not tied to a specific provider or CLI command, such as configuration management, metadata tagging logic, downloading utilities, library indexing, and the database schema.
- `flaccid/tests/`: Test suite for all components. Includes unit tests for business logic and integration tests for CLI behavior and overall workflows.
- `pyproject.toml`: Defines the Python project, including its name, version, dependencies (Typer, Pydantic v2, Dynaconf, Mutagen, aiohttp, Keyring, Rich, SQLAlchemy, Watchdog, etc.), as well as console script entry point (`fla`).

This structure encourages separation of concerns: the CLI layer handles user interaction and argument parsing, plugins deal with external service APIs, core modules handle generic logic (tagging, downloading, database), and tests ensure each part works correctly. Next, we will dive into each area with code examples and explanations.

CLI Entry Point and Command Dispatching

The CLI entry point is defined in **`flaccid/cli.py`**. This module creates the main Typer application and registers subcommand groups. By using Typer, we get a clean way to define commands and parameters with Python functions and decorators, plus automatic help messages and tab-completion.

Below is the implementation of `cli.py`:

```
# flaccid/cli.py

import typer
from flaccid.commands import get, tag, lib, settings

# Create the main Typer app
app = typer.Typer(help="FLACCID CLI - A modular FLAC toolkit")

# Register sub-apps for each command group
app.add_typer(get.app, name="get", help="Download tracks or albums from streaming
services")
app.add_typer(tag.app, name="tag", help="Tag local files using online metadata")
app.add_typer(lib.app, name="lib", help="Manage local music library (scan/index)")
app.add_typer(settings.app, name="set", help="Configure credentials and paths")

if __name__ == "__main__":
    app()
```

In this code:

- We import the submodules from `flaccid.commands` (each submodule defines a Typer `app` object for its group).

- We create a top-level `app = typer.Typer(...)` and use `app.add_typer(...)` to attach each subgroup under the appropriate name. For example, the `get` module's Typer app is attached as the `get` command group[typer.tiangolo.com](https://typer.tiangolo.com/tutorial/subcommands/nested-subcommands/#:~:text=app %3D typer.Typer() app.add_typer(users.app%2C name%3D,lands).
- Each `add_typer` call also includes a help string describing that group of commands.
- Finally, if the module is executed directly, we invoke `app()`. This allows running `python -m flaccid` during development, and also the entry point for the installed CLI will call this.

By structuring the CLI with multiple Typer instances, each command category is isolated in its own module, making the code easier to maintain and extending to new subcommands straightforward[typer.tiangolo.com](https://typer.tiangolo.com/tutorial/subcommands/nested-subcommands/#:~:text=app %3D typer.Typer() app.add_typer(users.app%2C name%3D,lands). Typer automatically composes the help such that `fla --help` will list the command groups, and `fla get --help` will show the commands under `get`, and so on.

Command Group Implementations

Each command group (`get`, `tag`, `lib`, `set`) has its own module in `flaccid/commands/` containing a Typer app and the specific command functions. Let's go through each:

`fla get` – Download Commands

The `get` command group allows users to download music from supported streaming services. Currently, we implement two subcommands: `qobuz` and `tidal`. These will utilize the corresponding plugins to handle API interactions and downloads.

```
# flaccid/commands/get.py

import asyncio
from pathlib import Path
import typer
from flaccid.core import downloader, config
from flaccid.plugins import qobuz, tidal

app = typer.Typer()

@app.command("qobuz")
def get_qobuz(album_id: str = typer.Option(..., "--album-id", help="Qobuz album ID to
download", rich_help_panel="Qobuz Options"),
              track_id: str = typer.Option(None, "--track-id", help="Qobuz track ID to
download (if single track)"),
              quality: str = typer.Option("lossless", "--quality", "-q", help="Quality
format (e.g. 'lossless', 'hi-res')"),
              output: Path = typer.Option(None, "--out", "-o", help="Output directory
(defaults to library path)")):
    """
    Download an album or track from Qobuz.
    """
    # Ensure output directory is set
    dest_dir = output or config.settings.library_path
```

```
    dest_dir.mkdir(parents=True, exist_ok=True)
    typer.echo(f"Downloading from Qobuz to {dest_dir} ...")

    # Initialize Qobuz plugin (ensure credentials are loaded)
    qbz = qobuz.QobuzPlugin()
    qbz.authenticate()  # Will load stored credentials and obtain auth token

    # Fetch album or track metadata
    if album_id:
        album_meta = asyncio.run(qbz.get_album_metadata(album_id))
        tracks = album_meta.tracks
    elif track_id:
        track_meta = asyncio.run(qbz.get_track_metadata(track_id))
        tracks = [track_meta]
    else:
        typer.echo("Error: You must specify either --album-id or --track-id", err=True)
        raise typer.Exit(code=1)

    # Download all tracks (async)
    asyncio.run(qbz.download_tracks(tracks, dest_dir, quality))
    typer.secho("Qobuz download complete!", fg=typer.colors.GREEN)


@app.command("tidal")
def get_tidal(album_id: str = typer.Option(None, "--album-id", help="Tidal album ID to
download", rich_help_panel="Tidal Options"),
              track_id: str = typer.Option(None, "--track-id", help="Tidal track ID to
download"),
              quality: str = typer.Option("lossless", "--quality", "-q", help="Quality
(e.g. 'lossless', 'hi-res')"),
              output: Path = typer.Option(None, "--out", "-o", help="Output directory
(defaults to library path)")):
    """
    Download an album or track from Tidal.
    """
    dest_dir = output or config.settings.library_path
    dest_dir.mkdir(parents=True, exist_ok=True)
    typer.echo(f"Downloading from Tidal to {dest_dir} ...")

    tdl = tidal.TidalPlugin()
    tdl.authenticate()  # use stored credentials or perform login flow

    if album_id:
        album_meta = asyncio.run(tdl.get_album_metadata(album_id))
        tracks = album_meta.tracks
    elif track_id:
        track_meta = asyncio.run(tdl.get_track_metadata(track_id))
        tracks = [track_meta]
    else:
        typer.echo("Error: You must specify either --album-id or --track-id", err=True)
        raise typer.Exit(code=1)
```

```
    asyncio.run(tdl.download_tracks(tracks, dest_dir, quality))
    typer.secho("Tidal download complete!", fg=typer.colors.GREEN)
```

Key points in the `get` commands:

- Both commands accept either an album ID or a track ID (but not both) to identify what to download. They also accept a `--quality` option to request a certain quality level (the interpretation of this is handled by the plugin, e.g., mapping "lossless" to a specific stream format or bitrate).
- `--out` can specify a custom output directory; if not provided, the default library path from config is used.
- We ensure the destination directory exists (`mkdir(parents=True, exist_ok=True)`).
- We initialize the appropriate plugin (`QobuzPlugin` or `TidalPlugin`) and call an `authenticate()` method to ensure we have a valid API session (using credentials from config/Keyring).
- We fetch metadata for the album or track. The plugin provides methods like `get_album_metadata` or `get_track_metadata` which return Pydantic models containing track info (e.g., track titles, file URLs, etc.). We use `asyncio.run()` to execute these async calls synchronously within the Typer command function. Typer can also work with async functions directly, but here we manage it explicitly for clarity.
- Once we have a list of tracks to download, we call `download_tracks(tracks, dest_dir, quality)` via the plugin. This method is asynchronous (to perform concurrent downloads), hence we also wrap it in `asyncio.run()`.
- Progress feedback and logging inside `download_tracks` (using Rich) will inform the user of download progress. After completion, we print a success message in green text (`typer.secho(..., fg=typer.colors.GREEN)`).
- Error handling: if neither album_id nor track_id is given, we output an error message and exit with a non-zero code.

The separation of concerns is evident here: the CLI command mostly handles user input and output, delegates to the plugin for heavy lifting, and ensures the environment (like output directory) is prepared. By using asyncio for downloads, the user can download multiple tracks in parallel, significantly speeding up album downloads.

`fla tag` – Tagging Commands

The `tag` group provides commands to apply metadata tags to existing FLAC files from online sources. We implement `qobuz` for Qobuz metadata and `apple` for Apple Music (iTunes) metadata. These commands typically take an identifier (or search query) for the album and a target directory of files to tag.

```
# flaccid/commands/tag.py

import typer
from pathlib import Path
from flaccid.core import metadata, config
from flaccid.plugins import qobuz, apple

app = typer.Typer()

@app.command("qobuz")
```

```
def tag_qobuz(album_id: str = typer.Option(..., "--album-id", help="Qobuz album ID to
fetch metadata"),
              folder: Path = typer.Argument(..., exists=True, file_okay=False,
dir_okay=True, help="Path to the album folder to tag")):
    """
    Tag a local album's FLAC files using Qobuz metadata.
    """
    typer.echo(f"Fetching metadata from Qobuz for album {album_id} ...")
    qbz = qobuz.QobuzPlugin()
    qbz.authenticate()
    album_meta = qbz.get_album_metadata_sync(album_id)  # using a sync wrapper or
asyncio.run inside
    typer.echo(f"Applying tags to files in {folder} ...")
    metadata.apply_album_metadata(folder, album_meta)
    typer.secho("Tagging complete (Qobuz)!", fg=typer.colors.GREEN)


@app.command("apple")
def tag_apple(query: str = typer.Argument(..., help="Album search query or Apple album
ID"),
              folder: Path = typer.Argument(..., exists=True, file_okay=False,
dir_okay=True, help="Path to the album folder to tag")):
    """
    Tag a local album's FLAC files using Apple Music (iTunes) metadata.
    """
    typer.echo(f"Searching Apple Music for '{query}' ...")
    am = apple.AppleMusicPlugin()
    # The query might be an album ID or a search term
    if query.isdigit():
        album_meta = am.get_album_metadata(query)
    else:
        album_meta = am.search_album(query)
    if album_meta is None:
        typer.echo("Album not found on Apple Music.", err=True)
        raise typer.Exit(code=1)
    typer.echo(f"Found album: {album_meta.title} by {album_meta.artist}. Applying
tags...")
    metadata.apply_album_metadata(folder, album_meta)
    typer.secho("Tagging complete (Apple Music)!", fg=typer.colors.GREEN)
```

Notes on the `tag` commands:

- For `tag qobuz`, we require an `--album-id` (assuming the user knows the Qobuz album ID or has it from a link). The target folder is a positional argument (must exist) pointing to the album's files on disk. We instantiate the Qobuz plugin, authenticate, and fetch album metadata. We then call a core function `apply_album_metadata` to perform the tagging of files in that folder with the retrieved metadata.
- For `tag apple`, we allow the user to either provide an album identifier (if known) or a search query string. If the query is all digits, we treat it as an Apple album ID; otherwise, we perform a search. The AppleMusicPlugin handles the search via Apple's iTunes Search API (no credentials needed for basic search). It returns the best matching album's metadata. If nothing is found, we exit with an error. If found, we proceed to apply tags similarly with `apply_album_metadata`.

- Both commands rely on a core `metadata.apply_album_metadata` function to do the actual tagging (writing FLAC tags, cover art, etc.). This function will use Mutagen and possibly the lyrics plugin as needed (described in the Metadata section).
- The CLI provides user feedback throughout: indicating when it's fetching metadata and when it's writing tags, and a completion message. Errors (like album not found) are printed to stderr and cause a graceful exit.

By splitting the provider-specific logic (in plugins) from the tagging application logic (in `core/metadata.py`), we ensure consistency. The `apply_album_metadata` function can handle merging metadata from multiple sources (if needed) and writing to files, while the plugin simply supplies a structured `Album` data object.

`fla lib` – Library Management Commands

The `lib` command group deals with the local music library database. We have two commands: `scan` and `index`. The design is that `scan` is used to detect changes and update the library incrementally (and possibly run continuously), while `index` can build or rebuild indices and perform deeper checks.

```
# flaccid/commands/lib.py

import typer
from flaccid.core import library, config

app = typer.Typer()

@app.command("scan")
def lib_scan(watch: bool = typer.Option(False, "--watch", help="Continuously watch the
library for changes"),
             verify: bool = typer.Option(False, "--verify", help="Verify file integrity
while scanning")):
    """
    Scan the music library for new, changed, or deleted files and update the database.
    """
    lib_path = config.settings.library_path
    typer.echo(f"Scanning library at {lib_path} ...")
    library.scan_library(lib_path, verify=verify)
    typer.secho("Initial scan complete!", fg=typer.colors.GREEN)
    if watch:
        typer.echo("Watching for changes. Press Ctrl+C to stop.")
        try:
            library.watch_library(lib_path, verify=verify)
        except KeyboardInterrupt:
            typer.echo("Stopped watching the library.")


@app.command("index")
def lib_index(rebuild: bool = typer.Option(False, "--rebuild", help="Rebuild the entire
index from scratch"),
              verify: bool = typer.Option(False, "--verify", help="Perform integrity
verification on all files")):
    """
    Index the entire library and update the database. Use --rebuild to start fresh.
```

```
    """
    lib_path = config.settings.library_path
    if rebuild:
        typer.echo("Rebuilding library index from scratch...")
        library.reset_database()  # Drop and recreate tables
    else:
        typer.echo("Updating library index...")
    library.index_library(lib_path, verify=verify)
    typer.secho("Library indexing complete!", fg=typer.colors.GREEN)
```

Explanation:

- Both `lib scan` and `lib index` use the `flaccid.core.library` module, which contains the implementation of scanning and indexing logic.
- `lib_scan`: Calls `library.scan_library(path, verify=...)` for incremental scanning. The `verify` flag triggers integrity checks (if true, e.g. verifying FLAC file integrity or checksum). After initial scan, if `--watch` is provided, it then calls `library.watch_library` which uses Watchdog to monitor the directory for changes continuously. The `watch_library` function will run until interrupted (Ctrl+C), handling events like created files (adding them), modified files (re-indexing them), or deleted files (removing from DB).
- `lib_index`: If `--rebuild` is specified, it calls `library.reset_database()` to wipe and initialize the DB (so we start fresh). Then it calls `library.index_library(path, verify=...)` which likely performs a full scan of all files and ensures the database is up-to-date. Without `--rebuild`, it might do similar work but preserving existing entries (in practice, `scan_library` and `index_library` might share code; here we conceptually separate quick scan vs full index).
- Both commands use `typer.echo` to inform the user of progress and a success message at the end. The `verify` option in both triggers deeper checks (which might slow the process, so it's optional).
- We use `config.settings.library_path` to know which directory to scan/index. The user sets this via `fla set path` or it defaults to something like `~/Music` or a directory in config.

This design allows users to simply run `fla lib scan` to keep their database in sync with their filesystem, and optionally use `--watch` to keep it running as a daemon. The `index` command can be used for a thorough (or initial) indexing or when moving to a new database.

`fla set` – Configuration Commands

The `set` group provides a way for users to configure essential settings like service credentials and library path from the CLI. This writes to config or uses Keyring for secrets.

```
# flaccid/commands/settings.py

import typer
from flaccid.core import config
import keyring

app = typer.Typer()

@app.command("auth")
```

```
def set_auth(service: str = typer.Argument(..., help="Service to authenticate (e.g.
'qobuz', 'tidal', 'apple')")):
    """
    Set or update authentication credentials for a service (stored securely).
    """
    service = service.lower()
    if service not in ("qobuz", "tidal", "apple"):
        typer.echo(f"Unknown service '{service}'. Available options: qobuz, tidal, apple",
err=True)
        raise typer.Exit(code=1)
    # Prompt user for credentials or token
    if service == "apple":
        typer.echo("Apple Music typically requires a developer token and a music user
token.")
        dev_token = typer.prompt("Enter Apple Music Developer Token (JWT)",
hide_input=True)
        music_token = typer.prompt("Enter Apple Music User Token", hide_input=True)
        # Store tokens in keyring (we use service name and key names)
        keyring.set_password("flaccid_apple", "developer_token", dev_token)
        keyring.set_password("flaccid_apple", "user_token", music_token)
        typer.echo("Apple Music tokens stored securely.")
    else:
        username = typer.prompt(f"Enter {service.capitalize()} username/email")
        password = typer.prompt(f"Enter {service.capitalize()} password", hide_input=True)
        # Store credentials securely in system keyring
        keyring.set_password(f"flaccid_{service}", username, password)
        # Store username in config for reference
        config.settings[service]["username"] = username
        config.settings.save()
        typer.echo(f"{service.capitalize()} credentials stored for user {username}.")
        typer.echo("You can now use 'fla get {service} ...' or 'fla tag {service} ...'
commands.")


@app.command("path")
def set_path(location: str = typer.Argument(..., help="Path to the music library or
download folder")):
    """
    Set the base path for the music library.
    """
    import os
    # Expand user tilde if present
    normalized = os.path.expanduser(location)
    config.settings.library_path = normalized
    config.settings.save()
    typer.echo(f"Library path set to: {normalized}")
```

Details for `set` commands:

- `set auth`: Accepts a service name argument (we support 'qobuz', 'tidal', 'apple'). It then interactively prompts the user for the necessary credentials:

    - For **Apple**: Apple's Music API uses tokens. We prompt for a Developer Token (JWT issued by Apple for your app) and a User Token (authenticates the user's Apple Music account). These are then stored in the keyring under a service identifier "flaccid_apple" with distinct keys for each token. We do not store Apple password because Apple Music doesn't use a simple username/password for API access.
    - For **Qobuz/Tidal**: We prompt for username (or email) and password. We then call `keyring.set_password("flaccid_qobuz", username, password)`. This stores the password securely in the OS keychain. We also save the username in the Dynaconf config (so that the program knows which username to use by default for that service). The config is then saved to persist this change (likely writing to `settings.toml` or a user config file).
    - The user feedback confirms that credentials are stored. On subsequent runs, the plugin will fetch the password from keyring using the stored username.
- Security: By using Keyring, we avoid writing sensitive passwords to disk. Even the Dynaconf `.secrets.toml` file is not used for these (to reduce risk). If we did use Dynaconf secrets file for tokens, it would be kept out of version controldynaconf.com, but here we rely on the system key vault for maximum security.
- `set path`: Takes a directory path and updates `config.settings.library_path`. We expand `~` to the user's home directory for convenience. The new path is saved to config (persisting in the `settings.toml`). Users can thus configure where downloaded music and library files reside. On different OSes, they might set this to a platform-specific location (the tool itself might have a default like `~/Music` on Unix or `%USERPROFILE%\Music` on Windows, but this allows customization).

With these config commands, the user can prepare the environment for using the other features (download/tag). For example, they should run `fla set auth qobuz` once to store their Qobuz credentials before using `fla get qobuz`. The separation into a command means no credentials are required as plain CLI arguments, and the interactive prompt ensures sensitive input isn't shown on screen.

Configuration Design and Secure Credential Management

FLACCID uses **Dynaconf** for managing configuration, combined with **Pydantic** for validation. This provides a flexible, layered configuration system with support for multiple environments, file formats, and secure secrets management.

Dynaconf Settings

Dynaconf allows configuration via a `settings.(toml|yaml|json|py)` file, environment variables, and a special `.secrets.*` file for sensitive data. In FLACCID, we use a `settings.toml` file for general settings and optionally a `.secrets.toml` for any sensitive overrides (although most secrets are stored in keyring instead). The configuration might include:

- Default library path (if not set by user).
- Default quality setting for downloads (e.g., prefer "lossless" or specific format).
- API keys or app IDs for services (for Qobuz or Tidal if required).
- Possibly toggles for features (like enabling lyrics fetching, etc).
- Database file location or name.

An example `settings.toml` might look like:

```
[default]
library_path = "~/Music/FLACCID"  # default library location
default_quality = "lossless"

# Service-specific settings
[default.qobuz]
app_id = "YOUR_QOBUZ_APP_ID"
app_secret = "YOUR_QOBUZ_APP_SECRET"
username = ""  # filled after auth
# (passwords not stored here for security)

[default.tidal]
client_id = "YOUR_TIDAL_API_KEY"
client_secret = "YOUR_TIDAL_API_SECRET"
username = ""

[default.apple]
# Apple doesn't need username/password in config, uses tokens via keyring
music_user_token = ""  # (optionally store a token here, but we prefer keyring)
```

Dynaconf loads the `default` environment by default. Sensitive values (like actual passwords or tokens) can be put in a separate `.secrets.toml` which is automatically loaded by Dynaconf if presentdynaconf.com. For instance, `.secrets.toml` could contain the Qobuz app secret or an API token. The `.secrets.toml` file is kept out of source control (the project's `.gitignore` should include it by default)dynaconf.com.

Pydantic for Config Validation

We define Pydantic models to validate and access config data in a structured way. Pydantic v2 ensures that types are correct and can provide default values.

```
# flaccid/core/config.py

from pathlib import Path
from pydantic import BaseModel
from dynaconf import Dynaconf

# Initialize Dynaconf to load settings and secrets
settings = Dynaconf(
    envvar_prefix="FLA",
    settings_files=['settings.toml', '.secrets.toml'],
    environments=["default"],
)

class ServiceCredentials(BaseModel):
    username: str = ""
    # We won't store password here, it's in keyring
    # We might store a token if available (e.g., Qobuz auth token after login)
    token: str | None = None
```

```
class AppSettings(BaseModel):
    library_path: Path
    default_quality: str = "lossless"
    qobuz: ServiceCredentials = ServiceCredentials()
    tidal: ServiceCredentials = ServiceCredentials()
    apple: dict = {}  # Apple might not need credentials in the same way

# Validate and parse settings using Pydantic
app_config = AppSettings(**settings)  # This will read values from Dynaconf
```

In this snippet:

- We create a Dynaconf `settings` object to load configuration from `settings.toml` and `.secrets.toml`. We also allow environment variables prefixed with `FLA` to override settings (e.g., `FLA_LIBRARY_PATH=/mnt/music` would override the config).
- We define `ServiceCredentials` as a Pydantic model to represent credentials for a service. It holds a username and possibly a token (for example, once we authenticate to Qobuz via their API, we might get a user auth token which we can store in memory or even update the config).
- `AppSettings` defines the structure of our app's config: the library_path, a default_quality, and nested structures for each service. We use `Path` type for library_path (Pydantic will automatically expand `~` to the user home for Path).
- We then instantiate `AppSettings` with the data from `settings`. Pydantic will validate types (e.g., ensure library_path is a Path, etc.) and apply defaults for missing values. If required fields were missing, it would raise an error (though here we have defaults for all).
- We now have an `app_config` object we can use in code, or we could stick with Dynaconf's `settings`. In practice, Dynaconf's `settings` is already quite convenient (attribute access), so using Pydantic is an optional validation layer. It can catch config mistakes early and provide IDE type hints for config usage.

Secure Credentials with Keyring

As seen in the `set auth` command, we avoid storing raw passwords in config. Instead, **Keyring** is used to store and retrieve credentials from the OS-provided secure storage (like Windows Credential Vault, macOS Keychain, or Secret Service on Linux)alexwlchan.net.

Workflow for credentials:

- When the user runs `fla set auth [service]`, we prompt and then call `keyring.set_password(service_id, username, password)`. We choose a `service_id` like "flaccid_qobuz" or "flaccid_tidal" to namespace our credentials.
- We record the username in the Dynaconf config (so the program knows which account is in use).
- In the plugin's `authenticate()` method, we retrieve the password with `keyring.get_password(service_id, username)` and then use it to obtain an API token or perform login.

- **Qobuz**: Requires an app ID/secret (from config) and user credentials to get a user auth token via their API. After a successful login, we get a token which could be stored in memory for that session. (We might also store the token in keyring or config if we want to reuse it without logging in every time, as long as it's long-lived.)
- **Tidal**: Similar approach – use the user credentials (or possibly OAuth if needed) to get a session/token. (The exact implementation depends on Tidal's API – we may use an unofficial API library that handles the OAuth dance.)
- **Apple**: The `set auth apple` stores the Developer Token and User Token in keyring. These tokens are long-lived (developer token might expire after months, user token typically doesn't expire often unless user revokes access). The AppleMusicPlugin will fetch these via `keyring.get_password("flaccid_apple", "developer_token")` etc., and use them to authenticate requests.

By leveraging Keyring, credentials are never exposed in plaintext within our codebase or config files, and the user benefits from the OS-level security (they might need to unlock their keychain once, etc., depending on platform). This approach is generally more secure than environment variables or plaintext secrets filesalexwlchan.net, though we allow those in a pinch for advanced users (e.g., setting `FLA_QOBUZ_PASSWORD` env var could be another way if keyring is unavailable, but that's not the primary method).

Note: In headless or server environments without a user keyring, developers might need to configure an alternative keyring backend (e.g., using keyring's fallback to plaintext or specifying an environment keyring). By default, our design assumes a desktop environment with access to the user keyring. If keyring throws an error (no backend), we could catch it and prompt the user to either use environment variables or install a suitable backend.

Config Loading and Usage

The `flaccid/core/config.py` ensures that `Dynaconf` loads at program start (when `flaccid.cli` is imported, it will import commands which import config). This means `config.settings` is ready to use anywhere. For convenience, many modules import `config.settings` directly to get config values.

For example, the library path is accessed as `config.settings.library_path`. Typer's CLI commands can use those values at runtime. If the user changed a setting via CLI (like `set path` or `set auth`), we call `settings.save()` to write changes back to the `settings.toml` file. Dynaconf handles writing to the appropriate file (by default it might write to a user-specific config if configured, but here we load from project directory for simplicity – in a real installation, we might specify a path in the user's home directory for the config).

Environment Overrides: We allow environment vars as override for advanced usage. For example, if someone doesn't want to persist config, they could run:

```
FLA_LIBRARY_PATH="/mnt/storage/Music" fla lib scan
```

and the program would pick up that library path from the env var due to `envvar_prefix="FLA"` in our Dynaconf setup.

Credentials Flow Recap:

1. User runs `fla set auth qobuz` and enters credentials.
2. Credentials stored via Keyring; username stored in `settings.toml`.
3. User runs `fla get qobuz ...`.
4. Inside `QobuzPlugin.authenticate()`, the code looks up `username = settings.qobuz.username`, then calls `password = keyring.get_password("flaccid_qobuz", username)`. It then uses Qobuz API (with the app_id and app_secret from config) to log in and obtain a user token. The token is stored in the plugin instance for the session (and could optionally be saved to avoid login next time).
5. Subsequent API calls use that token. The token could also be cached in keyring (as part of `ServiceCredentials.token` or similar) if we want persistence without logging in each run.
6. Similarly for Tidal.

This design balances security and convenience. The user only has to enter passwords once, and they are safely stored. We leverage Dynaconf for non-sensitive settings and Keyring for secrets, following best practices of not hardcoding secrets in code or repositorydynaconf.com.

Plugin System Architecture

One of the core goals of FLACCID is modular support for multiple music providers and extensibility. The plugin system is designed to accommodate **streaming/download providers**, **metadata providers**, and other auxiliary data sources (like lyrics). By defining clear interfaces and using a plugin registry, new services can be added with minimal changes to the CLI code.

Plugin Interfaces

In `flaccid/plugins/base.py`, we define abstract base classes (ABCs) that outline the expected functionality of providers. For example:

```
# flaccid/plugins/base.py

from abc import ABC, abstractmethod
from typing import List

class TrackMetadata:
    """Simple data holder for track metadata and download info."""
    def __init__(self, id: str, title: str, artist: str, album: str,
                 track_number: int, disc_number: int = 1,
                 duration: float = 0.0,  # in seconds
                 download_url: str | None = None):
        self.id = id
        self.title = title
        self.artist = artist
        self.album = album
        self.track_number = track_number
        self.disc_number = disc_number
        self.duration = duration
        self.download_url = download_url

class AlbumMetadata:
```

```
    """Data holder for album metadata and list of tracks."""
    def __init__(self, id: str, title: str, artist: str, year: int = 0,
                 cover_url: str | None = None, tracks: List[TrackMetadata] = None):
        self.id = id
        self.title = title
        self.artist = artist
        self.year = year
        self.cover_url = cover_url
        self.tracks = tracks or []

class MusicServicePlugin(ABC):
    """Abstract base class for music service plugins (streaming & metadata)."""

    @abstractmethod
    def authenticate(self):
        """Authenticate with the service (use config credentials)."""
        raise NotImplementedError

    @abstractmethod
    async def get_album_metadata(self, album_id: str) -> AlbumMetadata:
        """Fetch album metadata (tracks, etc.) by album ID."""
        raise NotImplementedError

    @abstractmethod
    async def get_track_metadata(self, track_id: str) -> TrackMetadata:
        """Fetch track metadata by track ID."""
        raise NotImplementedError

    @abstractmethod
    async def download_tracks(self, tracks: List[TrackMetadata], dest_dir, quality: str):
        """Download the given tracks to dest_dir at the specified quality."""
        raise NotImplementedError

class MetadataProviderPlugin(ABC):
    """Base class for metadata-only providers (no direct downloads)."""
    @abstractmethod
    def search_album(self, query: str) -> AlbumMetadata | None:
        """Search for an album by name/artist and return metadata if found."""
        raise NotImplementedError

    @abstractmethod
    def get_album_metadata(self, album_id: str) -> AlbumMetadata | None:
        """Fetch album metadata by an ID specific to this provider."""
        raise NotImplementedError

class LyricsProviderPlugin(ABC):
    """Base class for lyrics providers."""
    @abstractmethod
    def get_lyrics(self, artist: str, title: str) -> str | None:
        """Fetch lyrics for a given song, or return None if not found."""
        raise NotImplementedError
```

Here:

- `TrackMetadata` and `AlbumMetadata` are simple data classes (could also be Pydantic models) to hold information about tracks and albums. This includes fields like title, artist, track number, and for tracks, possibly a direct download URL (some APIs provide a URL or we construct one after authentication).
- `MusicServicePlugin` is an abstract class defining the key operations for a service that provides both metadata and actual audio content (download). It includes methods to authenticate, retrieve album/track metadata, and download tracks.
- `MetadataProviderPlugin` is a narrower interface for services that only provide metadata lookup (no downloading). Apple Music fits here – we can search and get metadata, but we cannot download audio from Apple's API.
- `LyricsProviderPlugin` covers getting lyrics for tracks. This could be implemented by a plugin that calls an API like Genius or Musixmatch, for example.

These interfaces ensure each plugin implements a standard set of methods. The CLI and core code will call these methods without needing to know the implementation details for each service.

Plugin Implementations

Now, let's look at how specific plugins implement these interfaces.

Qobuz Plugin (`flaccid/plugins/qobuz.py`):

```
# flaccid/plugins/qobuz.py

import aiohttp
import asyncio
from flaccid.plugins.base import MusicServicePlugin, AlbumMetadata, TrackMetadata
from flaccid.core import config

class QobuzPlugin(MusicServicePlugin):
    def __init__(self):
        self.app_id = config.settings.qobuz.get("app_id", "")
        self.app_secret = config.settings.qobuz.get("app_secret", "")
        self.username = config.settings.qobuz.get("username", "")
        self._auth_token = None

    def authenticate(self):
        """Obtain Qobuz API auth token using stored credentials."""
        # Fetch password from keyring
        import keyring
        password = keyring.get_password("flaccid_qobuz", self.username)
        if not password or not self.app_id:
            raise RuntimeError("Qobuz credentials or app ID not set. Run 'fla set auth
qobuz'.")
        # Qobuz API: get user auth token
        # Example: https://www.qobuz.com/api.json/0.2/user/login?
username=...&password=...&app_id=...
        auth_url = "https://www.qobuz.com/api.json/0.2/user/login"
        params = {
```

```
            "username": self.username,
            "password": password,
            "app_id": self.app_id
        }
        response = asyncio.get_event_loop().run_until_complete(
            aiohttp.ClientSession().get(auth_url, params=params))
        data = asyncio.get_event_loop().run_until_complete(response.json())
        response.close()
        if "user_auth_token" in data:
            self._auth_token = data["user_auth_token"]
        else:
            raise RuntimeError("Failed to authenticate with Qobuz. Check credentials.")

    async def get_album_metadata(self, album_id: str) -> AlbumMetadata:
        # Qobuz album info API
        url = "https://www.qobuz.com/api.json/0.2/album/get"
        params = {"album_id": album_id, "user_auth_token": self._auth_token}
        async with aiohttp.ClientSession() as session:
            async with session.get(url, params=params) as resp:
                data = await resp.json()
        # Parse album data into AlbumMetadata
        album = AlbumMetadata(
            id=str(data["album"]["id"]),
            title=data["album"]["title"],
            artist=data["album"]["artist"]["name"],
            year=int(data["album"].get("releaseDateDigital", "0")[:4]) if
"releaseDateDigital" in data["album"] else 0,
            cover_url=data["album"].get("image", None),
        )
        tracks = []
        for track in data["tracks"]["items"]:
            t = TrackMetadata(
                id=str(track["id"]),
                title=track["title"],
                artist=track["performer"]["name"],
                album=album.title,
                track_number=track.get("trackNumber", 0),
                disc_number=track.get("mediaNumber", 1),
                duration=track.get("duration", 0.0)
            )
            tracks.append(t)
        album.tracks = tracks
        return album

    async def get_track_metadata(self, track_id: str) -> TrackMetadata:
        # Qobuz track info API
        url = "https://www.qobuz.com/api.json/0.2/track/get"
        params = {"track_id": track_id, "user_auth_token": self._auth_token}
        async with aiohttp.ClientSession() as session:
            async with session.get(url, params=params) as resp:
                data = await resp.json()
        track = data["track"]
```

```
        t = TrackMetadata(
            id=str(track["id"]),
            title=track["title"],
            artist=track["performer"]["name"],
            album=track["album"]["title"],
            track_number=track.get("trackNumber", 0),
            disc_number=track.get("mediaNumber", 1),
            duration=track.get("duration", 0.0)
        )
        return t

    async def download_tracks(self, tracks: list[TrackMetadata], dest_dir, quality: str):
        # Determine format_id based on desired quality
        format_id = 27 if quality.lower() in ("hi-res", "hires") else 6  # example: 6 =
FLAC 16-bit, 27 = FLAC 24-bit
        # Qobuz file URL API
        file_url_api = "https://www.qobuz.com/api.json/0.2/track/getFileUrl"

        import os
        from flaccid.core.downloader import download_file  # a helper for downloading with
aiohttp
        tasks = []
        async with aiohttp.ClientSession() as session:
            for track in tracks:
                # Get the direct file URL for the track
                params = {
                    "track_id": track.id,
                    "format_id": format_id,
                    "user_auth_token": self._auth_token,
                    "app_id": self.app_id
                }
                async with session.get(file_url_api, params=params) as resp:
                    file_data = await resp.json()
                if "url" in file_data:
                    track.download_url = file_data["url"]
                else:
                    typer.secho(f"Failed to get download URL for track {track.title}",
fg=typer.colors.RED)
                    continue
                # Determine file path
                filename = f"{track.track_number:02d} - {track.title}.flac"
                filepath = os.path.join(dest_dir, filename)
                # Create asyncio task for downloading this track
                tasks.append(download_file(session, track.download_url, filepath))
            # Use asyncio.gather to download all tracks concurrently
            await asyncio.gather(*tasks)
```

Important aspects of `QobuzPlugin`:

- **Authentication**: The `authenticate()` method synchronously obtains a `user_auth_token` from Qobuz by calling their login API endpoint with username, password, and app_id. It uses `aiohttp` in a somewhat synchronous way (here we directly run the event loop to perform the GET request; alternatively, we could have made `authenticate` `async` and awaited it, but for simplicity it's sync). On success, it stores the token in `self._auth_token`.
- **get_album_metadata**: Calls Qobuz's album API to retrieve album details and track list. It constructs an `AlbumMetadata` object and a list of `TrackMetadata` for each track. This is an `async` function and uses an `aiohttp.ClientSession` to perform the request and parse JSON.
- **get_track_metadata**: Similar, for an individual track.
- **download_tracks**: This handles getting actual file URLs and downloading them. Qobuz requires another API call (`track/getFileUrl`) with a `format_id` for quality. We choose a format based on the `quality` parameter (in Qobuz API, for example, 5 might be MP3, 6 is 16-bit FLAC, 27 is 24-bit FLAC; here we simplified to two options). For each track:
  - It requests the file URL and sets `track.download_url`.
  - It creates a file name (prefixed with track number for ordering).
  - We then use a helper `download_file(session, url, path)` to asynchronously fetch and save the file. We gather all tasks and await them concurrently.
- We use a single `aiohttp.ClientSession` for all downloads to reuse connections (and pass it into `download_file` tasks).
- The actual implementation of `download_file` (in `flaccid/core/downloader.py`) would handle writing the response stream to disk and possibly updating a progress bar (discussed later). For brevity, assume it's a function that does something like:

  ```
  async def download_file(session, url, path):
      async with session.get(url) as resp:
          with open(path, 'wb') as f:
              async for chunk in resp.content.iter_chunked(1024):
                  f.write(chunk)
  ```

  Possibly enhanced with Rich progress.

The Qobuz plugin thus covers both metadata and downloading, fully implementing `MusicServicePlugin`.

Tidal Plugin (`flaccid/plugins/tidal.py`):

The Tidal plugin would be similar in structure to Qobuz's. Tidal has a different API (and might involve OAuth). For brevity, here's a conceptual outline:

```
# flaccid/plugins/tidal.py

import aiohttp
import asyncio
from flaccid.plugins.base import MusicServicePlugin, AlbumMetadata, TrackMetadata
from flaccid.core import config

class TidalPlugin(MusicServicePlugin):
```

```
    def __init__(self):
        self.username = config.settings.tidal.get("username", "")
        self._session_id = None  # Tidal uses a session or access token

    def authenticate(self):
        import keyring
        password = keyring.get_password("flaccid_tidal", self.username)
        if not password:
            raise RuntimeError("Tidal credentials not set. Run 'fla set auth tidal'.")
        # Use tidalapi or direct REST calls to login.
        # For simplicity, imagine we have an unofficial API:
        # tidal_api = tidalapi.Session()
        # tidal_api.login(self.username, password)
        # self._session_id = tidal_api.session_id
        # (If successful, _session_id is set; real Tidal API may return access token)
        # In absence of actual library, this is a placeholder.
        raise NotImplementedError("Tidal authentication needs implementation")

    async def get_album_metadata(self, album_id: str) -> AlbumMetadata:
        # Call Tidal API to get album and track list (requires auth, e.g., include session
token)
        # Parse into AlbumMetadata and TrackMetadata list.
        # (Pseudo-code as actual API details are omitted)
        album = AlbumMetadata(id=album_id, title="Album Title", artist="Artist Name")
        # ... populate album.tracks ...
        return album

    async def get_track_metadata(self, track_id: str) -> TrackMetadata:
        # Call Tidal API for track info
        track = TrackMetadata(id=track_id, title="Track Title", artist="Artist",
album="Album", track_number=1)
        return track

    async def download_tracks(self, tracks: list[TrackMetadata], dest_dir, quality: str):
        # Tidal might require getting stream URLs or offline content via API.
        # This could involve requesting a stream URL for each track (possibly time-
limited).
        # Pseudo-code:
        tasks = []
        async with aiohttp.ClientSession() as session:
            for track in tracks:
                # We would use the Tidal API/SDK to get a stream URL or file:
                # e.g., url = tidal_api.get_flac_url(track.id)
                url = None  # placeholder
                if url:
                    filename = f"{track.track_number:02d} - {track.title}.flac"
                    filepath = str(dest_dir / filename)
                    tasks.append(download_file(session, url, filepath))
            await asyncio.gather(*tasks)
```

Due to the complexity of Tidal's API, the above is left as high-level pseudo-code. An actual implementation might use the unofficial `tidalapi` library, which can login and fetch stream URLs (with decryption keys if needed). The key is that the plugin would provide similar methods: authenticate (set up session), get metadata, and download using aiohttp.

Apple Music Plugin (`flaccid/plugins/apple.py`):

Apple is metadata-only for our purposes (we can't download from Apple Music). It implements `MetadataProviderPlugin` instead of `MusicServicePlugin`.

```
# flaccid/plugins/apple.py

import requests  # using requests for simplicity since Apple search is single call
from flaccid.plugins.base import MetadataProviderPlugin, AlbumMetadata, TrackMetadata
from flaccid.core import config

class AppleMusicPlugin(MetadataProviderPlugin):
    def __init__(self):
        # We might retrieve tokens if needed
        import keyring
        self.dev_token = keyring.get_password("flaccid_apple", "developer_token")
        self.music_token = keyring.get_password("flaccid_apple", "user_token")

    def search_album(self, query: str) -> AlbumMetadata | None:
        # Use Apple iTunes Search API (no auth required, limited info)
        params = {"term": query, "entity": "album", "limit": 1}
        resp = requests.get("https://itunes.apple.com/search", params=params)
        results = resp.json().get("results", [])
        if not results:
            return None
        album_info = results[0]
        # Construct AlbumMetadata (note: iTunes API uses different field names)
        album = AlbumMetadata(
            id=str(album_info.get("collectionId")),
            title=album_info.get("collectionName"),
            artist=album_info.get("artistName"),
            year=int(album_info.get("releaseDate", "0")[:4]) if
album_info.get("releaseDate") else 0,
            cover_url=album_info.get("artworkUrl100")  # 100x100 image, can modify URL for
higher res
        )
        # We need track list which the search API doesn't provide; use lookup API:
        collection_id = album.id
        return self.get_album_metadata(collection_id)

    def get_album_metadata(self, album_id: str) -> AlbumMetadata | None:
        url = f"https://itunes.apple.com/lookup?id={album_id}&entity=song"
        resp = requests.get(url)
        data = resp.json().get("results", [])
        if not data or len(data) < 2:
            return None
        # First element is album info, rest are tracks
```

```
        album_info = data[0]
        album = AlbumMetadata(
            id=str(album_info.get("collectionId")),
            title=album_info.get("collectionName"),
            artist=album_info.get("artistName"),
            year=int(album_info.get("releaseDate", "0")[:4]) if
album_info.get("releaseDate") else 0,
            cover_url=album_info.get("artworkUrl100")
        )
        tracks = []
        for item in data[1:]:
            if item.get("wrapperType") == "track":
                t = TrackMetadata(
                    id=str(item.get("trackId")),
                    title=item.get("trackName"),
                    artist=item.get("artistName"),
                    album=album.title,
                    track_number=item.get("trackNumber", 0),
                    disc_number=item.get("discNumber", 1),
                    duration=item.get("trackTimeMillis", 0) / 1000.0  # convert ms to
seconds
                )
                tracks.append(t)
        album.tracks = tracks
        return album
```

Highlights of AppleMusicPlugin:

- It uses the public iTunes Search API and Lookup API (these don't require our stored tokens, as they are public endpoints). If more detailed data from Apple Music (beyond iTunes store info) was needed, we'd use the developer token via Apple Music API. But to keep it simple, iTunes data is sufficient for tagging.
- `search_album(query)`: calls the search API with term and entity=album, retrieves the first result, then calls `get_album_metadata` on that album's collectionId.
- `get_album_metadata(album_id)`: calls the iTunes lookup API to get album and track listings. It then builds an AlbumMetadata with track list.
- The cover_url from iTunes (artworkUrl100) is a link to 100x100 image. There's a known trick: those URLs can often be modified to get a larger image (by replacing `100x100` with e.g. `600x600`). We might do that to get a high-res cover for embedding.
- We do not implement any download functionality here, as Apple tracks cannot be downloaded via this method. If a user tries `fla get apple`, we haven't provided such a command.

Lyrics Plugin (`flaccid/plugins/lyrics.py`):

For lyrics, one could use an API like Genius. This would likely require an API token (which could be stored in config or .secrets). Our lyrics plugin interface is simple: get lyrics by artist and title.

A skeletal example (assuming a hypothetical lyrics API or using a web scrape):

```
# flaccid/plugins/lyrics.py
```

```
import requests
from flaccid.plugins.base import LyricsProviderPlugin

class DummyLyricsPlugin(LyricsProviderPlugin):
    """Example lyrics plugin that returns dummy lyrics."""
    def get_lyrics(self, artist: str, title: str) -> str | None:
        # In real implementation, call an API.
        # For example, using lyrics.ovh (a free lyrics API) as a placeholder:
        try:
            resp = requests.get(f"https://api.lyrics.ovh/v1/{artist}/{title}")
            data = resp.json()
            return data.get("lyrics")
        except Exception:
            return None
```

We might integrate a real provider by using an API key (for Genius, we'd use their REST API with OAuth token). The concept remains: given an artist and title, return lyrics text or None.

Plugin Discovery and Registration

In our CLI commands, we directly imported specific plugins (qobuz, tidal, apple). This is straightforward but adding a new service would require code changes in the CLI. To make the system more extensible without modifying CLI code for each new plugin, we could implement a plugin registry or dynamic loading:

- Maintain a mapping (dictionary) of service name to plugin class or instance. E.g., `plugins = {"qobuz": QobuzPlugin, "tidal": TidalPlugin, "apple": AppleMusicPlugin}`. The `get` and `tag` commands could then look up the appropriate plugin by name. However, Typer's design (separate subcommands) makes dynamic addition slightly tricky. We can still programmatically add Typer commands by iterating over a plugin registry at startup.
- Alternatively, use Python entry points (set in pyproject) so external packages can register plugins. For example, if someone created a `flaccid-spotify` plugin package, it could declare an entry_point in group `flaccid.plugins` and we could auto-load it.
- In this design, given the known set of services, we kept things simple. But we include developer notes that to add a new provider, one should implement the appropriate class and then update CLI and possibly config accordingly.

Combining Providers (Cascade)

The plugin system also allows using multiple sources for the same data. For instance, one might combine Qobuz and Apple data: Qobuz for technical metadata (bit depth, etc.) and Apple for canonical track names or genres. Our design for cascade could be implemented in `core/metadata.py` by using more than one plugin's output:

For example, if `fla tag qobuz` is run but Apple integration is enabled for enriching metadata, the code could fetch from Qobuz (primary) and then fetch from Apple and merge fields. However, this can complicate the CLI usage (we didn't explicitly design a command to tag from multiple sources at once). A more straightforward approach: the cascade refers to adding lyrics and artwork after getting the primary metadata from one source.

Thus, the plugin system is flexible but currently one command uses one primary provider (plus possibly the lyrics provider automatically).

Downloading and Async Operations (Downloader Module)

Downloading large audio files efficiently is a crucial part of FLACCID's functionality for the `get` commands. We utilize **aiohttp** for asynchronous HTTP requests and **Rich** for a user-friendly progress display.

The `flaccid/core/downloader.py` module provides helper functions to download files and possibly to report progress:

```
# flaccid/core/downloader.py

import aiohttp
import asyncio
from rich.progress import Progress, BarColumn, DownloadColumn, TimeRemainingColumn

async def download_file(session: aiohttp.ClientSession, url: str, dest_path: str):
    """Download a file from the given URL to dest_path using the provided session."""
    async with session.get(url) as resp:
        resp.raise_for_status()
        total = int(resp.headers.get('Content-Length', 0))
        # Setup progress bar for this download
        progress = Progress(BarColumn(), "[progress.percentage]{task.percentage:>3.1f}% ",
DownloadColumn(), TimeRemainingColumn())
        task_id = progress.add_task(f"Downloading", total=total)
        # Open file for writing in binary mode
        with open(dest_path, "wb") as f:
            # Iterate over response data chunks
            async for chunk in resp.content.iter_chunked(1024 * 64):
                if chunk:
                    f.write(chunk)
                    progress.update(task_id, advance=len(chunk))
        progress.update(task_id, completed=total)
        progress.stop()
```

What this does:

- It uses an aiohttp `ClientSession` passed in (to reuse connections).
- It reads the `Content-Length` header to know total size (for progress bar).
- It sets up a **Rich Progress** object with a progress bar, percentage, download speed (via `DownloadColumn`), and estimated time remaining (`TimeRemainingColumn`).
- As it reads chunks of the response, it writes to file and updates the progress bar accordingly.
- Once done, it marks the task complete and stops the progress display.

This function can be called for each file. In the Qobuz plugin, we created multiple tasks of `download_file` for concurrent downloads. However, we must be careful: if we start multiple progress bars concurrently, they will all print to the console. Rich's Progress can handle multiple tasks in one Progress instance, or we could create one Progress per file but need to manage how they are displayed. A simpler approach is to sequentially download tracks with one progress bar at a time, but concurrency was a goal for speed.

To handle concurrency with a combined progress display, we could do:

```
async def download_album(tracks, dest_dir):
    with Progress(BarColumn(), "[progress.percentage]{task.percentage:>3.1f}%",
DownloadColumn(), TimeRemainingColumn(), expand=True) as progress:
        tasks = []
        for track in tracks:
            # Each track gets a task and its own progress bar line
            task_id = progress.add_task(f"[white]{track.title}", total=track.size or 0)
            tasks.append(_download_track(track, dest_dir, progress, task_id))
        await asyncio.gather(*tasks)

async def _download_track(track, dest_dir, progress, task_id):
    # like download_file but updates the passed progress with given task_id
```

However, this complicates the code. For clarity in the handbook, the simpler `download_file` with its own Progress (so each download prints a progress bar sequentially) is shown. If concurrently downloading, the outputs might intermix but Rich may handle it by rendering multiple bars if the same Progress context is used.

We can note that concurrently downloading an album's tracks can drastically reduce total time (especially if network latency is a factor), and the user will see multiple progress bars at once or one after another depending on implementation details.

Rate limiting and API courtesy: If needed, one could implement delays or limits to not overwhelm the service (though for personal use, downloading an album's tracks concurrently is usually fine).

Quality selection: In Qobuz, we demonstrated choosing a format based on `quality` string. In a real scenario, we might want to map "lossless" to the best available lossless quality the user is entitled to (16-bit vs 24-bit). Qobuz's API returns available formats for a track; we could choose the highest FLAC quality automatically if `quality` is set to "lossless". The user might explicitly request hi-res with a flag. For Tidal, if the user has HiFi Plus, we might download FLAC (Tidal now uses FLAC for HiFi), etc.

Error handling: The download function should handle errors (like network issues) gracefully. In the code, `resp.raise_for_status()` will throw for HTTP errors (like 404), which we might catch and log. For brevity, not shown.

Metadata Tagging and Cascade Logic

Once music is downloaded or identified, tagging the files with correct metadata is essential. FLACCID uses **Mutagen** to write FLAC tags (Vorbis comments) and embed album art. Additionally, a **cascade** approach is used to enrich metadata from multiple sources: for example, adding lyrics via a lyrics plugin, or filling gaps in one provider's data with another's.

Metadata Cascade and Enrichment

The concept of a *metadata cascade* is to layer multiple metadata inputs in a priority order. In our context, the typical cascade when tagging might be:

1. **Primary Source**: e.g., Qobuz or Apple provides the core tags (title, artist, album, track number, etc., plus possibly album art).
2. **Secondary Source**: (Optional) Another provider could fill in missing fields or provide additional info (for instance, if Qobuz lacks genre or album artist, Apple's data might have it).
3. **Lyrics Provider**: If enabled, fetches lyrics for each track.
4. **Existing File Metadata**: If we choose not to overwrite certain fields already present (for instance, maybe a user's custom comment or a specific tag), we could choose to preserve them unless the new data has something.
5. **User Overrides**: A user might specify manual overrides (not in our CLI currently, but conceptually a config could force certain tags).

In the current implementation, we focus on using one primary source plus lyrics. But we design the `apply_album_metadata` function to be flexible:

```
# flaccid/core/metadata.py

from mutagen.flac import FLAC, Picture
from flaccid.plugins import lyrics as lyrics_plugin

def apply_album_metadata(folder_path: str, album_meta) -> None:
    """
    Apply metadata from AlbumMetadata to all FLAC files in the given folder.
    Files are matched to tracks by track number (and disc number if applicable).
    """
    # Sort files and tracks by track number for alignment
    import glob, os
    flac_files = sorted(glob.glob(os.path.join(folder_path, "*.flac")))
    tracks = sorted(album_meta.tracks, key=lambda t: (t.disc_number, t.track_number))
    if len(flac_files) != len(tracks):
        print("Warning: number of FLAC files does not match number of tracks in
metadata.")

    # Prepare cover art image if available
    cover_data = None
    if album_meta.cover_url:
        try:
            import requests
            resp = requests.get(album_meta.cover_url)
            cover_data = resp.content if resp.status_code == 200 else None
        except Exception:
            cover_data = None

    # Lyrics provider initialization
    lyr = None
```

```
    try:
        lyr = lyrics_plugin.DummyLyricsPlugin()
    except Exception:
        lyr = None

    # Iterate through files and corresponding metadata
    for i, file_path in enumerate(flac_files):
        audio = FLAC(file_path)
        if i < len(tracks):
            track = tracks[i]
        else:
            track = None
        if track:
            # Basic tags from metadata
            audio["title"] = track.title
            audio["artist"] = track.artist
            audio["album"] = track.album or album_meta.title
            audio["albumartist"] = album_meta.artist
            audio["tracknumber"] = str(track.track_number)
            audio["discnumber"] = str(track.disc_number)
            if album_meta.year:
                audio["date"] = str(album_meta.year)
            # Additional tags if available
            if hasattr(track, "genre") and track.genre:
                audio["genre"] = track.genre
            if hasattr(track, "isrc") and track.isrc:
                audio["isrc"] = track.isrc

            # Lyrics integration
            if lyr:
                lyrics_text = lyr.get_lyrics(track.artist, track.title)
                if lyrics_text:
                    audio["lyrics"] = lyrics_text

            # Embed cover art once per album (for the first track, or all tracks if
desired)
            if cover_data:
                # Remove existing pictures to avoid duplicates
                audio.clear_pictures()
                pic = Picture()
                pic.data = cover_data
                pic.type = 3  # front cover
                pic.desc = "Cover"
                pic.mime = "image/jpeg"  # assume JPEG; could detect from content
                audio.add_picture(pic)  # Embed album art:contentReference[oaicite:8]
{index=8}:contentReference[oaicite:9]{index=9}
        else:
            print(f"No metadata for file {file_path}, skipping tagging.")

        audio.save()
```

Explanation:

- We gather all `.flac` files in the folder and sort them. We sort tracks from metadata by disc and track number. We assume one disc or proper numbering; if multiple discs, the sorting should align with file naming if they're named with disc numbers in separate folders or all together.
- If the count of files and tracks differ, we warn the user. (We proceed anyway, tagging as many as possible.)
- **Album Art**: If `album_meta.cover_url` is provided, we attempt to download the image (using `requests` synchronously here for simplicity). We store the image bytes in `cover_data`. This could be None if download fails or not available.
- **Lyrics**: We attempt to instantiate a lyrics plugin. If no lyrics plugin is configured or fails, we set `lyr = None`. In practice, this could be configurable (maybe an option to enable/disable lyrics fetch).
- For each file and corresponding track metadata:
    - We load the FLAC file with Mutagen's `FLAC` class.
    - We write standard tags: title, artist, album, albumartist, tracknumber, discnumber, date (year). These fields correspond to Vorbis comments in FLAC. (Mutagen handles the mapping; e.g., `audio["date"]` will be stored as `DATE=`).
    - We check for extra fields like genre or ISRC if our metadata source provided them. (Our TrackMetadata class didn't include genre or isrc in earlier code, but Apple's API could provide genre via the album info. Qobuz might provide an ISRC per track. We illustrate that extra info could be set similarly.)
    - **Lyrics**: If a lyrics plugin is available, we call `get_lyrics(artist, title)`. If lyrics are found, we add them under the "lyrics" tag (which many players read for displaying lyrics).
    - **Album Art Embedding**: We use Mutagen's Picture class. We clear existing pictures first to avoid duplicates (Mutagen's `clear_pictures()` removes all embedded images for FLAC). Then create a Picture, set its `type` to 3 (cover front), set a description, MIME type (we assume JPEG for simplicity; ideally detect from bytes or Content-Type). We assign the raw image data to `pic.data`. Finally, `audio.add_picture(pic)` embeds the imagestackoverflow.comstackoverflow.com. In this implementation, we embed the same cover in each track's metadata. Alternatively, we could embed only in the first track to save space, but most players expect each file to have cover art, so we do all.
    - If no track metadata is available for a file (which could happen if files > metadata tracks), we skip tagging it.
    - We save the file to write changes to disk.

Mutagen takes care of writing the Vorbis comment block and picture block properly into the FLAC file. After this, the FLAC files have standardized tags.

Ensuring Correct Matching: We assumed that sorting by track number is enough to match files to metadata. This is often true if files are named in track order and no extras. However, if the files were not in order or missing numbers, a safer approach might be to match by existing tags (like matching titles or using AcoustID fingerprinting). Given a controlled use-case (just downloaded files or well-organized library), the assumption is acceptable, but developers might consider improvements.

Multiple Sources: In this function, we could incorporate a secondary metadata source by checking if certain fields are missing. For example, if `track.genre` is empty and we have another source's data, we could fill it. Implementation might involve merging two AlbumMetadata objects. As of now, we don't demonstrate that to keep it straightforward.

The cascade logic here primarily demonstrates combining the main provider's metadata with an auxiliary lyrics provider, and preserving some album-level info (like album artist, year) for all tracks.

Library Scanning and Indexing (Library Module)

Managing the local library involves scanning the filesystem for audio files, reading their metadata, and storing entries in a database. It also includes responding to changes (new files, deletions, changes) and verifying the integrity of files.

The `flaccid/core/library.py` module provides functions for scanning and indexing:

```
# flaccid/core/library.py

import os
from pathlib import Path
from mutagen.flac import FLAC
from flaccid.core import database

def scan_library(root: str, verify: bool = False):
    """
    Incrementally scan the library root for changes and update the database.
    New files are added, modified files are updated, deleted files are removed.
    """
    root_path = Path(root).expanduser()
    # Get list of all FLAC files in library
    files_on_disk = [p for p in root_path.rglob("*.flac")]
    # Query database for existing entries (paths)
    db_tracks = database.get_all_tracks()  # returns list of Track ORM objects
    db_paths = {Path(track.path) for track in db_tracks}

    files_set = set(files_on_disk)
    # New files: in files_set but not in db_paths
    new_files = files_set - db_paths
    # Deleted files: in db_paths but not in files_set
    deleted_files = db_paths - files_set
    # Possible modified files: intersection, but we'll verify by timestamp or hash
    modified_files = []
    for track in db_tracks:
        p = Path(track.path)
        if p in files_set:
            # Compare last modified time or size between DB record and actual file
            if p.stat().st_mtime > track.last_modified:
                modified_files.append(p)

    # Process deletions
    for p in deleted_files:
```

```
        database.remove_track(p)  # remove track (and possibly album if empty) from DB

    # Process new files
    for p in new_files:
        try:
            audio = FLAC(p)
        except Exception as e:
            print(f"Warning: {p} is not a valid FLAC file or could not be read.
Skipping.")
            continue
        tags = audio.tags
        track_info = {
            "title": tags.get("title", [""])[0],
            "artist": tags.get("artist", [""])[0],
            "album": tags.get("album", [""])[0],
            "albumartist": tags.get("albumartist", [""])[0] if "albumartist" in tags else
tags.get("artist", [""])[0],
            "tracknumber": int(tags.get("tracknumber", [0])[0] or 0),
            "discnumber": int(tags.get("discnumber", [1])[0] or 1),
            "year": int(tags.get("date", [0])[0][:4] or 0) if "date" in tags else 0,
            "genre": tags.get("genre", [""])[0] if "genre" in tags else "",
        }
        # Compute duration and maybe MD5 if verify
        info = audio.info
        duration = info.length
        md5 = None
        if verify and hasattr(info, "md5_signature"):
            md5 = info.md5_signature  # FLAC streaminfo MD5 of raw audio
        database.add_track(path=str(p), metadata=track_info, duration=duration, md5=md5)

    # Process modified files
    for p in modified_files:
        try:
            audio = FLAC(p)
        except Exception:
            continue
        tags = audio.tags
        track = database.get_track(p)
        if not track:
            continue
        track.title = tags.get("title", [""])[0]
        track.artist = tags.get("artist", [""])[0]
        track.album = tags.get("album", [""])[0]
        track.albumartist = tags.get("albumartist", [""])[0] if "albumartist" in tags else
track.artist
        track.tracknumber = int(tags.get("tracknumber", [0])[0] or 0)
        track.discnumber = int(tags.get("discnumber", [1])[0] or 1)
        if "date" in tags:
            track.year = int(tags.get("date", ["0"])[0][:4] or 0)
        if "genre" in tags:
            track.genre = tags.get("genre", [""])[0]
        track.duration = audio.info.length
```

```
        if verify and hasattr(audio.info, "md5_signature"):
            track.md5 = audio.info.md5_signature
        database.update_track(track)
```

And functions for full indexing and watching:

```
def index_library(root: str, verify: bool = False):
    """
    Full indexing: go through all files and ensure database matches exactly.
    Potentially slower than scan, but thorough.
    """
    # Simplest approach: reset DB and add all files (if rebuild is intended).
    scan_library(root, verify=verify)

def reset_database():
    database.reset()  # drop and recreate tables

def watch_library(root: str, verify: bool = False):
    """
    Watch the library directory for changes (using watchdog) and update DB in real-time.
    """
    from watchdog.observers import Observer
    from watchdog.events import FileSystemEventHandler

    class LibraryHandler(FileSystemEventHandler):
        def on_created(self, event):
            if not event.is_directory and event.src_path.endswith(".flac"):
                print(f"Detected new file: {event.src_path}")
                try:
                    audio = FLAC(event.src_path)
                    tags = audio.tags
                    track_info = { ... }  # similar to above extraction
                    duration = audio.info.length
                    md5 = audio.info.md5_signature if verify and hasattr(audio.info,
"md5_signature") else None
                    database.add_track(path=event.src_path, metadata=track_info,
duration=duration, md5=md5)
                    print(f"Added to library: {track_info.get('title')} -
{track_info.get('album')}")
                except Exception:
                    return

        def on_deleted(self, event):
            if not event.is_directory and event.src_path.endswith(".flac"):
                print(f"Detected deletion: {event.src_path}")
                database.remove_track(event.src_path)

        def on_modified(self, event):
            if not event.is_directory and event.src_path.endswith(".flac"):
                print(f"Detected modification: {event.src_path}")
                # We can treat modifications similar to created (re-read tags and update)
```

```
                try:
                    audio = FLAC(event.src_path)
                    tags = audio.tags
                    track = database.get_track(event.src_path)
                    if not track:
                        return
                    track.title = tags.get("title", [""])[0]
                    # ... update other fields ...
                    track.duration = audio.info.length
                    if verify and hasattr(audio.info, "md5_signature"):
                        track.md5 = audio.info.md5_signature
                    database.update_track(track)
                except Exception:
                    return

    path = str(Path(root).expanduser())
    event_handler = LibraryHandler()
    observer = Observer()
    observer.schedule(event_handler, path, recursive=True)
    observer.start()
    try:
        # Run indefinitely
        while True:
            import time
            time.sleep(1)
    finally:
        observer.stop()
        observer.join()
```

What the library module is doing:

- **scan_library**:
  - It finds all `.flac` files (using `rglob` for recursive search).
  - Compares with the database's known tracks:
    - New files: add to DB (reading tags via Mutagen FLAC).
    - Deleted files: remove from DB.
    - Modified files: we check by last modified timestamp (or could use file size or a hash). If a file's mtime is newer than what's stored in DB, we consider it modified and update the DB entry.
  - When adding, we extract basic tags from the file using `audio.tags` (Mutagen returns a dict-like object for Vorbis comments). We create a `track_info` dict and pass it to `database.add_track`, along with duration and MD5 if verify is True.
  - For updating modified files, we retrieve the track from DB (`database.get_track`) and update its fields, then call `database.update_track`.
  - The verify option: if true, we attempt to get `audio.info.md5_signature`. For FLAC, Mutagen's FLAC info typically includes the MD5 signature of the uncompressed audio stream (this is stored in the FLAC file's header when encoded). By storing this, we can later verify integrity by comparing with a newly calculated MD5 or using `flac` command-line. Here we just store it as reference. If a

file's MD5 changes, it indicates the audio data changed (which is unusual unless file was transcoded or corrupted).

- **index_library**: We simplified it to just call scan_library. In a more advanced design, `index_library` might rebuild from scratch (especially if `rebuild=True` was passed, which in our CLI triggers `reset_database()` first). The distinction is minor in code; the CLI logic handled the rebuild flag. The purpose is to provide a full traversal that ensures DB is exactly in sync with disk.
- **reset_database**: Delegated to the database module to drop and create tables anew.
- **watch_library**:
  - Uses Watchdog to set up an Observer on the library path.
  - We define a nested `LibraryHandler` class extending `FileSystemEventHandler` with handlers for create, delete, modify events.
  - On created: if a new FLAC file is added, we parse it and add to DB (similar to scan's new file logic).
  - On deleted: remove from DB.
  - On modified: re-read tags and update (or potentially handle rename as a delete+create if a file is moved).
  - We print messages to console when events occur to inform the user (these could be toggled or logged instead).
  - The observer is started and runs until interrupted. The `lib scan --watch` command is intended to call this and then block. In the CLI command, we catch KeyboardInterrupt to stop the observer and exit gracefully.

Database Operations: The library module calls functions in the `database` module (like `add_track`, `remove_track`, etc.). We will define those next. The division is for clarity: `library.py` handles file system concerns, and `database.py` manages persistence.

By using Watchdog, we enable a near-real-time sync with the file system, which is very useful if the user is ripping CDs, purchasing new FLACs and dropping them in the folder, or editing tags with another tool – the database will update accordingly (depending on how modifications are detected).

Integrity Verification: When `--verify` is used, we store the FLAC audio MD5. We could also actually verify the file by decoding it partially. If deeper verification was needed, one could run `flac -t` on each file (flac test), but that's slow. Instead, storing the MD5 from the file's header is a quick consistency check: if the file is later corrupted, one could attempt to decode and compare MD5, but that's beyond scope here.

Now, let's define the database schema and operations.

Database Schema and Management (Database Module)

We use **SQLAlchemy** (the latest version, v2 style) to create a local SQLite database for the library. The database stores tracks, and implicitly albums and artists. We can either use separate tables for Albums and Artists or store everything in one tracks table and derive album/artist info via queries. A normalized schema is cleaner and allows capturing album-level data (like cover art path, year, etc.) and artist info only once. Here's a possible schema:

```
# flaccid/core/database.py
```

```
from sqlalchemy import create_engine, Column, String, Integer, Float, ForeignKey, select
from sqlalchemy.orm import declarative_base, Session, relationship

Base = declarative_base()
engine = create_engine("sqlite:///flaccid_library.db", future=True, echo=False)  # SQLite
file in current directory (or use config path)
session = Session(engine)

class Album(Base):
    __tablename__ = "albums"
    id = Column(Integer, primary_key=True, autoincrement=True)
    title = Column(String)
    artist = Column(String)
    year = Column(Integer)
    genre = Column(String)
    cover_path = Column(String, nullable=True)  # path to cover image file (if we choose
to save covers separately)

    tracks = relationship("Track", back_populates="album_obj", cascade="all, delete-
orphan")

class Track(Base):
    __tablename__ = "tracks"
    id = Column(Integer, primary_key=True, autoincrement=True)
    album_id = Column(Integer, ForeignKey("albums.id"))
    title = Column(String)
    artist = Column(String)
    albumartist = Column(String)
    tracknumber = Column(Integer)
    discnumber = Column(Integer)
    year = Column(Integer)
    genre = Column(String)
    duration = Column(Float)
    path = Column(String, unique=True)
    last_modified = Column(Float)
    md5 = Column(String, nullable=True)

    album_obj = relationship("Album", back_populates="tracks")

# Create tables if not exist
Base.metadata.create_all(engine)

def reset():
    Base.metadata.drop_all(engine)
    Base.metadata.create_all(engine)

def add_track(path: str, metadata: dict, duration: float, md5: str | None = None):
    # Check if album exists or create
    album_title = metadata.get("album", "")
    album_artist = metadata.get("albumartist", metadata.get("artist", ""))
```

```
    album = session.query(Album).filter_by(title=album_title,
artist=album_artist).one_or_none()
    if not album:
        album = Album(title=album_title, artist=album_artist, year=metadata.get("year",
0), genre=metadata.get("genre", ""))
        session.add(album)
        session.flush()  # flush to get album.id if needed
    track = Track(
        title=metadata.get("title", ""),
        artist=metadata.get("artist", ""),
        albumartist=album_artist,
        tracknumber=metadata.get("tracknumber", 0),
        discnumber=metadata.get("discnumber", 1),
        year=metadata.get("year", 0),
        genre=metadata.get("genre", ""),
        duration=duration,
        path=path,
        last_modified=os.path.getmtime(path),
        md5=md5,
        album_obj=album
    )
    session.add(track)
    session.commit()

def get_all_tracks():
    return session.query(Track).all()

def get_track(path: str):
    return session.query(Track).filter_by(path=str(path)).one_or_none()

def update_track(track: Track):
    # Assuming track is a Track object already queried/attached to session
    track.last_modified = os.path.getmtime(track.path)
    session.commit()

def remove_track(path: str):
    track = session.query(Track).filter_by(path=str(path)).one_or_none()
    if track:
        session.delete(track)
        # Optionally delete album if no more tracks
        if track.album_obj and len(track.album_obj.tracks) == 1:
            session.delete(track.album_obj)
        session.commit()
```

Schema explanation:

- **Album** table: Stores album title, artist (album artist), year, genre (we assume one genre; if multiple, could handle differently), and perhaps a cover image path. We can also store an external service album ID if needed for reference, but not included here.

- **Track** table: Stores track title, track artist, albumartist, track number, disc number, year, genre, duration (in seconds), file path (must be unique), last_modified timestamp, and MD5. It has a foreign key linking to Album. Relationship `album_obj` allows accessing the album from a track and vice versa.
- We create the tables using `Base.metadata.create_all(engine)`. The engine is a SQLite database stored in `flaccid_library.db` in the current working directory. (We might want this in a user config directory. We could adjust the path to `config.settings.library_path/flaccid.db` or similar. Keeping it simple here.)
- `reset()`: Drops all tables (erasing the library) and recreates them. Used in `lib index --rebuild`.
- `add_track()`: Adds a track to the DB. It checks if the album exists (matching by title and album artist). If not, creates a new Album entry. Then creates a Track with given metadata and links it to the album. We use `os.path.getmtime` to store the file's last modified time as `last_modified`. We commit at the end.
- `get_all_tracks()`: Returns all Track entries (used in scanning to compare).
- `get_track(path)`: Find a track by path.
- `update_track(track)`: We assume the track object's fields have been modified by the caller (as in `library.scan_library` when updating tags). We simply update the last_modified to current file time and commit. (Alternatively, we could merge a detached object, but since we hold the object from session, we can commit directly.)
- `remove_track(path)`: Deletes a track by path. Also checks if its album now has zero tracks, and deletes the album if so (prevent orphan albums). Commits the transaction.

SQLAlchemy provides a high-level ORM interface. We run operations in a session. For simplicity, we created a single global session. In a larger application, we might manage sessions differently (especially if doing multi-threading or long-running processes), but here it's fine.

Threading Note: Watchdog callbacks occur in a separate thread (the Observer thread). Our usage of a global Session from multiple threads is not thread-safe. In a production scenario, we'd need to ensure DB access is done in the main thread or use a session per thread with a thread-safe queue. For simplicity, one might avoid doing heavy DB ops directly in the Watchdog thread - perhaps just mark something and let the main thread handle it. Given the complexity, our example ignores this threading issue. Developers should be aware and possibly adjust by using SQLAlchemy's thread-local sessions or calling back to main thread for DB writes.

Cross-Platform DB Path: The engine string "sqlite:///flaccid_library.db" places the DB in current directory. We might instead want it in a standard location. For example, use `config.settings.library_path` or a subdir for database. Or use `platformdirs` to get a config or data folder:

```
from platformdirs import user_data_dir
db_path = Path(user_data_dir("FLACCID", appauthor=False)) / "library.db"
engine = create_engine(f"sqlite:///{db_path}", future=True)
```

This would store it in a proper user data directory (like `%APPDATA%\FLACCID\library.db` on Windows, `~/.local/share/FLACCID/library.db` on Linux, etc.). For clarity we didn't include platformdirs in code, but this is a recommended practice for real apps.

Now the library scanning functions use this DB layer to reflect changes. This database allows us to answer questions like "do I have this album already?", "list all tracks by X", "find tracks missing tags", etc. We have not built CLI commands for those, but it could be extended (like `fla lib list` or `fla lib find ...`). The focus here is on maintaining the index.

Testing Strategy and Architecture

A comprehensive test suite ensures that each component of FLACCID works as expected and helps future contributors make changes confidently. We use **Pytest** as the testing framework, and organize tests by module/feature.

Test Environment Setup:

- Tests should run in isolation from the user's real environment. We use temporary directories and in-memory or temporary databases for tests, so as not to modify actual music files or config.
- Use Pytest fixtures to set up and tear down contexts (like creating a temp config or sample files).

Sample Tests:

1. **CLI Command Tests:** We can use Typer's built-in test utilities or Click's CliRunner to invoke CLI commands as if a user typed them, and verify the output or resulting state.

   ```
   from typer.testing import CliRunner
   from flaccid import cli

   runner = CliRunner()

   def test_set_path(tmp_path):
       result = runner.invoke(cli.app, ["set", "path", str(tmp_path)])
       assert result.exit_code == 0
       # The settings should now have the new path
       from flaccid.core import config
       assert Path(config.settings.library_path) == tmp_path
   ```

   This test simulates `fla set path <tempdir>` and then checks that the config was updated.

2. **Plugin Tests:** We can test plugin methods by mocking external API calls. For example, for QobuzPlugin, instead of actually hitting the API (which requires credentials), we can monkeypatch `aiohttp.ClientSession.get` to return a predefined JSON response for known album IDs.
   - Alternatively, use the `responses` library or `aiohttp` test utilities to simulate HTTP responses.
   - Test that `get_album_metadata` returns an AlbumMetadata with expected fields when given a fake JSON.
   - Test that `download_tracks` correctly constructs file names and calls download (we might monkeypatch `download_file` to avoid actually downloading).
   - For AppleMusicPlugin, we can test that search is parsing correctly by injecting sample JSON from iTunes.
3. **Metadata Tagging Tests:**

- Create a dummy FLAC file for testing (we could either include a small FLAC in test data or use mutagen to create one from scratch). Mutagen can write a FLAC file but creating a valid FLAC from nothing is non-trivial, so better to have a tiny FLAC file in `tests/data`.
- Use the `apply_album_metadata` function on a temp directory with a copy of that FLAC file. Provide it a small AlbumMetadata with a track. Then reopen the file with Mutagen to assert that the tags were written (e.g., check audio["title"] matches what was given, and that audio.pictures is not empty if cover was provided).
- If embedding an image, include a small image in test data to supply as cover bytes.

4. **Library Scanning Tests:**
   - Create a temporary directory structure with some dummy FLAC files (could copy the same small FLAC multiple times, changing names).
   - Run `library.scan_library` on it.
   - Query the `database` to see if all files are added.
   - Modify a file's tag (using mutagen in the test) and run scan again; check that the DB entry updated.
   - Remove a file and run scan; check DB entry removed.
   - These tests ensure the logic in scan correctly adds/updates/deletes entries.
   - Test `watch_library` in a controlled way: we can trigger the event handlers manually by calling them (since it's hard to actually generate filesystem events in tests reliably). For example, call `LibraryHandler().on_created(FakeEvent(path))` with a dummy event object to simulate and see if database.add_track was called. Or use `tmp_path` with Pytest's capability to watch for changes (could spawn watch_library in a thread and then add a file, etc., but that's more integration testing).
5. **Database Tests:**
   - Using an in-memory SQLite (`sqlite:///:memory:`) for speed. We can override the `engine` in tests to use in-memory by monkeypatching `database.engine` and calling `Base.metadata.create_all` on it.
   - Test that adding a track twice doesn't duplicate album.
   - Test remove_track deletes album when last track removed.
   - Essentially, ensure ORM relationships behave as expected.
6. **Config Tests:**
   - Possibly test that Dynaconf and Pydantic integration works: e.g., set environment variables and see that config picks them up.

Running tests: The project would include a `pytest.ini` or similar to configure test runs. Running `pytest` should find tests in the `tests/` folder. We may also have a coverage report generation to ensure we cover as much as possible.

Continuous Integration: It's advisable to set up CI (GitHub Actions or similar) to run tests on each commit, possibly on multiple OS (to ensure cross-platform compatibility). This way, if a contributor breaks a function on Windows that worked on Linux, tests would catch it.

Example Test Code:

```
# tests/test_metadata.py
import os
from pathlib import Path
from flaccid.core import metadata
from flaccid.plugins.base import AlbumMetadata, TrackMetadata

def create_dummy_flac(tmp_path: Path, title: str, artist: str) -> Path:
    """Helper to create a dummy FLAC file with minimal metadata for testing."""
    file_path = tmp_path / f"{title}.flac"
    # Copy a template small FLAC or generate one; here assume a template exists in
tests/data/dummy.flac
    import shutil
    shutil.copy("tests/data/dummy.flac", file_path)
    # Tag it with given title and artist
    audio = metadata.FLAC(file_path)
    audio["title"] = title
    audio["artist"] = artist
    audio.save()
    return file_path

def test_apply_album_metadata(tmp_path):
    # Create a dummy flac file
    song_file = create_dummy_flac(tmp_path, "Test Song", "Test Artist")
    # Create AlbumMetadata and TrackMetadata to apply
    track_meta = TrackMetadata(id="1", title="New Title", artist="New Artist", album="New
Album", track_number=1)
    album_meta = AlbumMetadata(id="100", title="New Album", artist="New Artist",
year=2021, cover_url=None, tracks=[track_meta])
    # Apply metadata
    metadata.apply_album_metadata(str(tmp_path), album_meta)
    # Verify the file's tags have changed
    audio = metadata.FLAC(song_file)
    assert audio["title"][0] == "New Title"
    assert audio["artist"][0] == "New Artist"
    assert audio["album"][0] == "New Album"
```

This test demonstrates how we might verify tagging functionality. In practice, we need a small FLAC file in the tests (which could be a few KB of silence encoded in FLAC).

By covering each component with tests, we ensure that the pieces (plugins, tagging, database, CLI commands) work in isolation. Integration tests (like simulating a full `fla get qobuz` flow with a fake Qobuz response, then checking the file is downloaded and tagged) could also be extremely valuable.

Packaging and Deployment Guide

FLACCID is packaged as a standard Python project with modern tooling (PEP 621 / pyproject.toml). Packaging it properly ensures that users and developers can install it easily and the CLI entry point works out of the box.

Project Metadata (pyproject.toml):

The `pyproject.toml` contains project metadata and dependencies. For example:

```
[project]
name = "flaccid"
version = "0.1.0"
description = "FLACCID: Modular FLAC CLI toolkit for music download and tagging"
authors = [
  { name="Your Name", email="you@example.com" }
]
requires-python = ">=3.10"

dependencies = [
  "typer>=0.7.0",
  "pydantic>=2.0.0",
  "dynaconf>=3.1.0",
  "mutagen>=1.45.0",
  "aiohttp>=3.8.0",
  "keyring>=23.0.0",
  "rich>=13.0.0",
  "SQLAlchemy>=2.0.0",
  "watchdog>=2.1.0",
  "requests>=2.0.0"
]

[project.urls]
Homepage = "https://github.com/yourname/flaccid"

[project.scripts]
fla = "flaccid.cli:app"
```

Notable points:

- We list all required libraries so that `pip install flaccid` will fetch them.
- The console script entry point is defined under `[project.scripts]` as `fla = "flaccid.cli:app"`. This means when installed, a command `fla` will be created, which executes `flaccid.cli:app`. Because `app` is a Typer object, it is callable and will run the CLI (Typer/Click will handle invoking the commands). This way, the user can just run `fla ...` after installation.
- Alternatively, we could have specified an explicit function (like `main`) that calls `app()`, but Typer's `app` itself is designed to be used as an entry point.
- We include `requests` in dependencies because our Apple plugin and some parts use it (though we could have used aiohttp everywhere, `requests` is fine for quick calls).

If using Setuptools instead of PEP 621, we'd have similar definitions in `setup.cfg` or setup.py using entry_points. But pyproject with the `project.scripts` field is modern and simple.

Installing the Package:

- For end users: they can install from PyPI (once published) with `pip install flaccid`. This will make `fla` available in their PATH.
- For development: developers can clone the repository and install in editable mode:

  ```
  git clone https://github.com/yourname/flaccid.git
  cd flaccid
  pip install -e .
  ```

  The `-e` (editable) means any changes to the code reflect immediately in the installed command, which is convenient for development and testing CLI changes. After this, running `fla --help` should show the CLI help.

Editable vs Standard Install: The `pip install -e .` uses the project's `pyproject.toml` and creates links in the environment to the source files. This also installs development dependencies if specified (sometimes people use `[project.optional-dependencies] dev = [...]` for test tools, etc., and then `pip install -e .[dev]` to include them). We could add:

```
[project.optional-dependencies]
dev = ["pytest", "pytest-cov", "tox", "flake8", "mypy"]
```

So that `pip install -e .[dev]` sets up a full dev environment.

Building and Publishing:

- To build distribution artifacts: use a PEP517 builder like `build`:

  ```
  pip install build
  python -m build
  ```

  This will create `dist/flaccid-0.1.0.tar.gz` and `dist/flaccid-0.1.0-py3-none-any.whl`.
- These can be uploaded to PyPI with twine:

  ```
  pip install twine
  twine upload dist/*
  ```

- Once on PyPI, users can install normally.

Cross-platform Notes for Installation:

Our dependencies are pure Python except maybe Watchdog which may have some OS-specific build steps (it uses C extensions for some platforms). `pip` will handle those (wheels for common OSes). On Windows, installing Watchdog might require Build Tools if not wheel, but likely wheels are provided.

Post-install Configuration:

After installation, when `fla` is run for the first time, it will generate a default `settings.toml` in the current directory if not found (depending on how Dynaconf is configured). We might want to have it use a user config directory instead. For example, we could set:

```
settings = Dynaconf(..., settings_files=['~/.config/flaccid/settings.toml',
'~/.config/flaccid/.secrets.toml'])
```

ensuring config is per user. But then we might want an initial config creation step. We did not implement an explicit init, but `set path` and `set auth` effectively create and modify config.

We should document in the README for end users where the config is stored and how to change it. Possibly also provide `fla set path` so they don't have to manually edit the file.

Upgrading:

If a new version is released, `pip install -U flaccid` upgrades it. The user's config and database remain (since those are in separate files).

Packaging for different Python versions:

We require Python 3.10+. Make sure to test on all supported minor versions and OS.

Now, we will provide examples of using the CLI which we partially did earlier, but let's consolidate them.

Example CLI Invocations and Usage

This section demonstrates typical uses of the FLACCID CLI, with example commands and expected outcomes. (Note: These examples assume valid credentials have been set for Qobuz/Tidal as needed.)

- **Authenticating with a service:**

  ```
  $ fla set auth qobuz
  Enter Qobuz username/email: user@example.com
  Enter Qobuz password: ********
  Qobuz credentials stored for user user@example.com.
  You can now use 'fla get qobuz ...' or 'fla tag qobuz ...' commands.
  ```

  (Stores Qobuz credentials securely. No output means success, aside from confirmation message.)

- **Setting library path:**

  ```
  $ fla set path ~/Music/FLAC
  Library path set to: /home/alice/Music/FLAC
  ```

  (Now the default output directory for downloads and the directory scanned for library commands is `/home/alice/Music/FLAC`.)

- **Downloading a Qobuz album:**

```
$ fla get qobuz --album-id 123456 --quality hi-res
Downloading from Qobuz to /home/alice/Music/FLAC ...
[Downloading Cover] 100% ━━━━━━━━━━━━━━━━━━━━ 500/500 kB • 0:00 • ???  (download
cover if implemented)
Downloading track 1: Song One (FLAC 24-bit)...
Downloading track 2: Song Two (FLAC 24-bit)...
... (progress bars for each track) ...
Qobuz download complete!
```

(The command fetches album metadata, then downloads each track concurrently. After completion, the files are in `~/Music/FLAC/Artist/Album/01 - Song One.flac`, etc., properly tagged.)

- **Downloading a Tidal track:**

```
$ fla get tidal --track-id 987654 --quality lossless -o /tmp
Downloading from Tidal to /tmp ...
Downloading track: My Song (FLAC 16-bit)...
Tidal download complete!
```

(Downloads a single track to `/tmp` directory. `-o` overrides the library path.)

- **Tagging a local album with Qobuz metadata:**

```
$ fla tag qobuz --album-id 123456 "/home/alice/Music/FLAC/Artist/Album"
Fetching metadata from Qobuz for album 123456 ...
Applying tags to files in /home/alice/Music/FLAC/Artist/Album ...
Tagging complete (Qobuz)!
```

(The FLAC files in the specified folder are now updated with official metadata from Qobuz, including album art and lyrics if available.)

- **Tagging a local album with Apple Music metadata:**

```
$ fla tag apple "Artist Name - Album Title" "./Album"
Searching Apple Music for 'Artist Name - Album Title' ...
Found album: Album Title by Artist Name. Applying tags...
Tagging complete (Apple Music)!
```

(The tool searched Apple's catalog for the album and applied those tags. This is useful if, for example, Qobuz metadata was incomplete and Apple's is more detailed.)

- **Scanning the library for new music:**

```
$ fla lib scan
Scanning library at /home/alice/Music/FLAC ...
Initial scan complete!
```

(This will print warnings for any files that couldn't be read, and will update the SQLite database. On first run, it adds all tracks. Subsequent runs, it might output messages like "Added X new tracks, Updated Y tracks, Removed Z tracks" depending on implementation.)

- **Continuous monitoring of library:**

```
$ fla lib scan --watch
Scanning library at /home/alice/Music/FLAC ...
Initial scan complete!
Watching for changes. Press Ctrl+C to stop.
```

  Then if the user adds a new file `new.flac` into the library folder, one might see:

```
Detected new file: /home/alice/Music/FLAC/New Artist/New Album/01 - New Song.flac
Added to library: New Song - New Album
```

  This happens automatically via Watchdog. The command will continue running until interrupted.

- **Rebuilding the index:**

```
$ fla lib index --rebuild --verify
Rebuilding library index from scratch...
Library indexing complete!
```

  (This drops the old database and reindexes everything, verifying each file's integrity. If a file's FLAC MD5 doesn't match or is corrupt, this could log an error or warning in a real scenario.)

- **Getting help:**

```
$ fla --help
Usage: fla [OPTIONS] COMMAND [ARGS]...

  FLACCID CLI - A modular FLAC toolkit

Options:
  --help  Show this message and exit.

Commands:
  get     Download tracks or albums from streaming services
  tag     Tag local files using online metadata
  lib     Manage local music library (scan/index)
  set     Configure credentials and paths
```

  And for subcommands:

```
$ fla get --help
Usage: fla get [OPTIONS] COMMAND [ARGS]...

  Download tracks or albums from streaming services

Options:
  --help  Show this message and exit.

Commands:
  qobuz  Download an album or track from Qobuz.
  tidal  Download an album or track from Tidal.
```

These examples illustrate the typical workflow: configure once, then download or tag as needed, and keep the library updated. The output messages and progress bars (not fully shown in text) give the user feedback.

Cross-Platform Compatibility and Path Handling

FLACCID is built with cross-platform support in mind:

- **File Paths**: We use Python's `pathlib.Path` and `os.path` to handle file system paths. This ensures that paths like `~/Music` are expanded to the correct location on each OS, and that separators are handled (e.g., on Windows, `Path("C:/Music")` vs Unix `/home/user/Music`). All internal file handling functions use these abstractions, so we don't hardcode any platform-specific path strings.
- **Default Locations**: By default, on Linux/macOS we might use `~/Music/FLACCID` for library path, whereas on Windows we might use `%USERPROFILE%\Music\FLACCID`. We allow the user to override via `set path` anyway. Using `platformdirs` could further tailor default config and data paths to OS conventions (not implemented above, but recommended).
- **Keyring Backends**: The `keyring` library automatically picks the appropriate backend:
  - Windows: Credential Manager
  - macOS: Keychain
  - Linux: Secret Service (or fallback to plaintext if none, which might require the user to install something like `keyring-secretservice` or have a DE running).
    If keyring fails on a headless environment, one can configure an alternate, but for the majority of desktop users it should just work. We output a friendly message if credentials aren't found or login fails, guiding the user to re-run `set auth`.
- **Watchdog**: The Watchdog observers have different implementations per OS (inotify on Linux, FSEvents on macOS, ReadDirectoryChangesW on Windows). The `watch_library` function doesn't need to change per OS – Watchdog handles the differences. One consideration is case-insensitivity on Windows (if a file moves from A.flac to a.flac, how we handle it – likely as a modify event). But generally, it works.
- **Character Encoding**: Tags are stored in UTF-8 (Vorbis comments in FLAC are UTF-8 by standard). Mutagen handles Unicode strings well. We ensure to use Python's default UTF-8 encoding for file names and console output. Windows console historically had issues with Unicode, but modern Windows 10+ with UTF-8 mode or using Windows Terminal is fine. If a user encounters weird characters, it's more console settings than our code. We rely on Typer/Click, which handles output encoding gracefully.

- **Testing on OS**: We would test at least on one Windows and one Unix platform. Particularly, file path differences (like reserved characters in filenames) should be considered. E.g., if an album or track title has ":" or "?" which are not allowed in Windows filenames, our downloader should sanitize filenames. We didn't explicitly include a sanitization function, but that would be wise to add in `download_tracks` (e.g., replace `:` with `-`).
- **Dependencies**: All listed libraries support Windows, Linux, macOS. Rich and Typer handle ANSI colors and progress bars on each (Rich will degrade gracefully or adjust for Windows). SQLAlchemy with SQLite is fine on all OS. Watchdog requires some C extensions that pip will install (wheel availability is good on common OS). No specific platform-bound code is present.
- **Differences in behavior**: One subtle difference: file paths in the database on Windows will have drive letters and backslashes. Our code treats them as strings, which is okay. Just ensure consistency if comparing. Using Path objects for comparisons normalizes case on Windows? Actually, Path("C:\a") == Path("C:\A") is False by default because it's case-sensitive in comparison even if NTFS isn't. We might want to normalize case or use `.lower()` on paths for DB keys on Windows to avoid duplicates. This level of detail might be too granular for now, but devs should be aware if users report "duplicate track entries" because of case differences.
- **Time zones and Date**: If we use year from tags, there's no TZ issue since it's just year or date string from metadata. If we recorded the full date-time, we'd want to be careful. Not a big issue here.

In summary, FLACCID should function on all major OS with no code changes. We leveraged Python's cross-platform libraries to abstract differences. Developer testing on each platform is recommended to catch any path or encoding quirks.

Troubleshooting and Diagnostics

Even with a robust design, users may encounter issues. Here are common problems and ways to troubleshoot them:

- **Authentication Errors**: If `fla get qobuz` returns an authentication error (e.g., unauthorized or invalid token):
  - Ensure you have run `fla set auth qobuz` with correct credentials. If credentials changed (password update, etc.), run it again to update.
  - Qobuz might also require a valid app_id/secret. Check that these are set in `~/.config/flaccid/settings.toml` (if provided) or in environment. Without a valid app_id, login will fail. You may need to register an app via Qobuz or use an existing app's credentials.
  - For Tidal, ensure your subscription level supports FLAC quality. Some APIs might fail if trying to access HiFi without entitlement.
  - Check if the system keyring is accessible. On Linux, for example, if running headless, the keyring might not be initialized, causing `keyring.get_password` to return None. In such cases, you might use environment variables as a fallback or configure the keyring backend.
- **Download Failures**: If a track or album fails to download:
  - Run with more verbose logging (if we had a `--debug` flag or set an env var). Since we didn't explicitly add a debug flag in code, one could manually insert print statements or use logging. For development, you might run the `get` command in a debugger to see what URL is being fetched.

    - Check internet connectivity and that the service isn't blocking the API usage. (Qobuz might throttle or block if too many requests quickly.)
    - Check that the output path is writable and has enough space.
    - If an error like "Failed to get download URL" appears for Qobuz, it might be that the format_id requested is not available for that track (e.g., trying hi-res on a track that only has 16-bit). You can try a different quality or ensure your account has access.
- **Tagging Issues**: If `fla tag` doesn't change the files or yields warnings:
    - Ensure the folder path is correct and contains FLAC files. The command expects `.flac` files; if they are .mp3 or others, currently the code specifically looked for FLAC. We might extend to other formats but in this toolkit FLAC is focus.
    - If tags applied but not visible in your player, it could be the player caches old tags. Try reading the file tags with Mutagen or another tool to confirm.
    - If album art isn't embedding, the image URL might be broken or internet needed. Or perhaps the image was PNG and some players require JPEG. We default to JPEG mime; if the source was PNG, that could be an issue (Mutagen's Picture with correct mime should handle it though).
    - If lyrics aren't appearing, maybe the lyrics provider didn't find any. You can check manually on the provider's site or try a different provider.
- **Library Database Issues**:
    - If `fla lib scan` doesn't pick up new files, confirm `library_path` is set correctly (`fla set path`).
    - If nothing happens on `--watch`, ensure Watchdog is installed correctly. On Linux, you may need inotify watch limits increased if watching a very large library (OS config).
    - If database appears out of sync or corrupted (perhaps due to a crash), you can rebuild: run `fla lib index --rebuild`.
    - If two instances of FLACCID run concurrently (maybe one watch and one scan), the database might be locked. SQLite has locks if a transaction is open. Avoid multiple simultaneous writes. If needed, our design could be improved with a single writer or using WAL mode. For now, if you see "database is locked" errors, stop other processes and retry.
- **Performance**: Scanning a huge library (thousands of files) can be slow the first time. Using `--verify` makes it even slower. If performance is an issue:
    - Skip `--verify` unless needed.
    - The first scan/index might take time, but subsequent scans are incremental.
    - Ensure you're not scanning a network drive over slow connection; if so, performance is inherently limited by I/O.
    - Watchdog, if watching many files, also has overhead but typically fine for moderate collections.
- **Debug Mode**: We have not explicitly implemented a debug logging flag, but adding one would be useful. One approach is to use Python's logging module and set levels via an option:

```
import logging
logging.basicConfig(level=logging.DEBUG)
logging.debug("Detailed info...")
```

In development, you could sprinkle `logging.debug` in critical parts (API responses, etc.) and run with an env var or flag to enable it.

- **Crash / Traceback**: If the CLI crashes with a traceback:
  - The Typer app will normally show a clean error message if we `typer.Exit`. But if an unexpected exception occurs, Typer will show the traceback. For debugging, that's good; for user-friendliness, we might catch exceptions in plugin calls and print a friendly message instead.
  - For example, if Qobuz API is down and returns HTML, our JSON parse might throw. We could catch `aiohttp.ClientError` or JSONDecodeError and inform "API error, try again later".
  - As a developer, look at the traceback, add handling as appropriate. We aim to handle known failure modes gracefully.
- **Diagnostics**: We could add a command like `fla diag` to print out environment info (versions of libraries, OS, whether keyring backend is available, etc.). This isn't implemented but could be a future addition for helping debug user issues.

In summary, many issues can be resolved by checking configuration and using the provided commands properly. The design tries to fail safe (e.g., not overwriting files incorrectly, not leaving partial downloads without at least a message). Future improvements might include logging to a file in config directory for deeper troubleshooting (so user can send a log to developers).

Developer Onboarding and Extension Guide

Welcome to the development side of FLACCID! This section provides guidance for new contributors or developers who want to extend the toolkit's functionality.

Getting Started with the Code

1. **Setup Development Environment**:
   - Clone the repository from version control (e.g., GitHub).
   - Install the package in editable mode with dev dependencies: `pip install -e .[dev]` (this installs the package and tools like pytest, linters if listed).
   - Ensure you have the required services credentials if you plan to test integration (Qobuz, Tidal). For offline development, you can work with dummy data or mock the network interactions.
2. **Project Structure Overview**: As detailed earlier, the code is organized by feature. A new developer should familiarize themselves with:
   - `flaccid/cli.py` to see how commands are wired.
   - `flaccid/commands/` for CLI logic.
   - `flaccid/plugins/` for external service integration.
   - `flaccid/core/` for internal logic (metadata, db, etc.).
   - The `tests/` to see usage examples and to verify changes don't break existing behavior.

3. **Coding Style**:
    - Follow Python best practices (PEP8 style). The project might include a linter (like flake8) and a formatter (like black). If so, run them before committing.
    - Use type hints for new functions/classes. We've used Python 3.10+ syntax (e.g., `str | None`). Type hints improve readability and help static analysis.
    - Write docstrings or comments for any complex logic or any public-facing functions.
4. **Running Tests**:
    - Use `pytest` to run the test suite frequently. Add new tests for any new feature or bugfix.
    - Ensure tests pass on all targeted Python versions. If using tox or CI, run those to be sure.
5. **Version Control**:
    - Work on a feature branch.
    - If adding a new feature, update this developer guide section accordingly if it affects the architecture or usage.
    - Commit messages should be clear. If an issue tracker exists, reference issue IDs.

Adding New Features or Providers

New Streaming/Metadata Provider (e.g., Spotify, Deezer, etc.):

- Create a new plugin module, e.g., `flaccid/plugins/spotify.py`.
- Implement a class (e.g., `SpotifyPlugin`) either deriving from `MusicServicePlugin` if it supports downloads, or `MetadataProviderPlugin` if only metadata.
- Add any needed configuration (API keys, etc.) to `settings.toml` and `.secrets.toml` as appropriate. Document them in the README.
- If the API requires OAuth (like Spotify), you might need to implement an OAuth flow. This could be complex (like opening a browser, etc.). Possibly out-of-scope for CLI; but maybe Spotify can use a user's refresh token from somewhere. Each service will differ in complexity.
- Write tests for the plugin using dummy data or actual API if keys are available (be mindful not to hardcode secrets in tests).
- Integrate the plugin with the CLI:
    - If it's a streaming service, add a subcommand in `get.py` (and in `tag.py` if you want to allow tagging from it).
    - If we had dynamic plugin loading, you'd register it. Currently, manually add similar to how Qobuz/Tidal are done.
    - Also update `set auth` command if credentials needed (like for Spotify, maybe ask for a token or client id/secret).
- Example: adding Deezer would be similar to Qobuz (Deezer has an API that could allow downloading via their user token).
- After adding, test by actually downloading or tagging from that service if possible.

Improving Lyrics Support:

- Replace `DummyLyricsPlugin` with a real implementation. For example, integrate with Genius:
  - Add a dependency on `lyricsgenius` or implement direct API calls. If using an API token, let the user set it in `.secrets.toml` or via `set auth lyrics` possibly.
  - Ensure the lyrics plugin is used in metadata.apply_album_metadata (which we already do).
  - Might allow user to run a separate command like `fla tag lyrics [folder]` to only add lyrics to existing tagged files.
- Write tests for lyrics fetching (maybe using a known song).
- Keep in mind rate limiting for lyrics APIs.

Enhanced Tagging:

- Implement more sophisticated matching for `fla tag` when user doesn't have an ID:
  - E.g., `fla tag qobuz "Album Name by Artist" /folder` could search Qobuz by name (not currently implemented). Qobuz API might have a search endpoint. This could be added to QobuzPlugin (like a method search_album similar to Apple's).
  - Or integrate MusicBrainz as a fallback for metadata (MusicBrainz provides comprehensive metadata and identifiers).
- Implement a true cascade: E.g., a command that uses multiple sources:
  - Possibly `fla tag auto "/folder"` that tries to identify the album via AcoustID or fuzzy matching, then pulls metadata from various sources automatically.
  - This would be a bigger feature involving fingerprinting (could use `pyacoustid` and MusicBrainz).
- If adding such features, keep them modular (maybe a separate module for acoustic ID, etc.) and ensure to update tests.

Database and Library:

- If scaling up, consider using an actual database server (PostgreSQL, etc.) if user desires. We can stick to SQLite for simplicity, but maybe allow in config to specify a different DB URL.
- Add commands to query the library. For instance, `fla lib list artists` or `fla lib search "Beatles"`. This would require building query logic (which is easy with SQLAlchemy queries).
- A `fla lib stats` could show number of tracks, total playtime, etc., by querying DB.
- If implementing these, ensure to format output nicely (maybe using Rich tables).

Performance Tuning:

- If library grows huge (tens of thousands of tracks), scanning might become slow. We can optimize by using database indices, or optimizing the file scanning (maybe maintain a cache of file->last_scanned_time to skip scanning unchanged directories).
- Watchdog performance: large libraries with many events might flood. Possibly implement debouncing or batch updates (e.g., if many files added at once, processing them in one go rather than individually).
- These can be iterative improvements. Add tests for performance sensitive functions if possible (maybe time a scan on a sample structure).

Cross-Platform Testing:

- If contributing code, test on at least your OS. Ideally, run the test suite on all OS via CI.
- Watch out for platform-specific pitfalls (like path normalization as discussed).

Documentation:

- Update the README.md for any new user-facing features or options.
- If adding a plugin, mention how to configure it (e.g., where to get API keys).
- Possibly update this developer handbook if the architecture changes (since it might live in a separate doc, ensure consistency).

Collaborative Development:

- Communicate with other contributors via issues or chat if something big is being changed, to avoid duplicate work.
- Code review: if using GitHub, open a PR and request review. This helps catch issues early and disseminate knowledge of changes among maintainers.

Conclusion for Developers:
FLACCID's rebuild emphasizes modularity and clarity. The code should be approachable: CLI definitions are straightforward with Typer, business logic is isolated, and external integrations are abstracted behind plugins. The test suite should give confidence to refactor or extend parts of the system.

By following this guide, a new developer should be able to set up the environment, run the tool, and dive into improving it. Whether it's adding the latest streaming service or optimizing the tagging flow, the modular structure will support the enhancement.

Happy coding, and enjoy making FLACCID an even more powerful toolkit for music enthusiasts!