# libmunin Documentation

***Release 0.0.1***

**Christopher Pahl**

October 20, 2013

CONTENTS

CHAPTER

ONE

# DESIGN

## 1.1 Overview

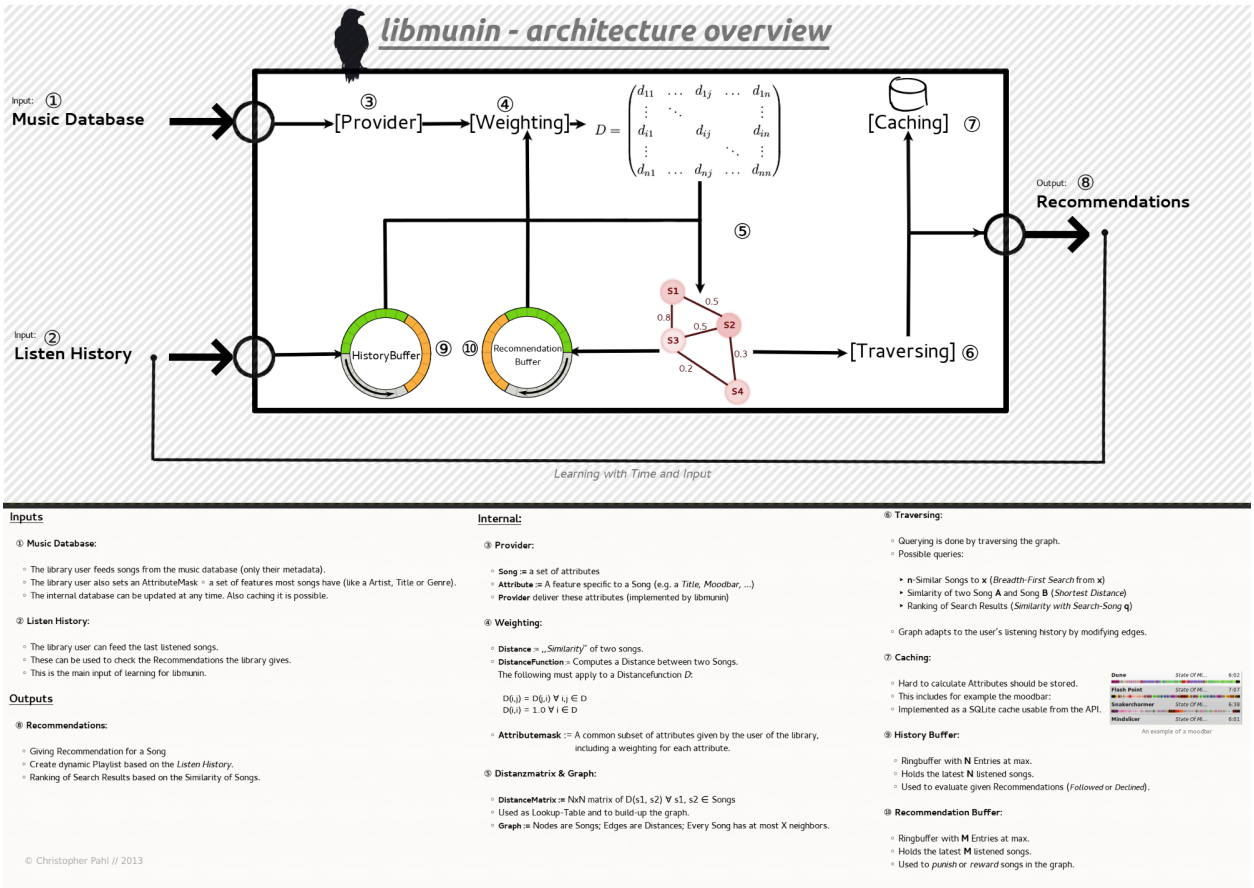


Figure 1.1: Complete Overview of one libmunin *Context*.
Click the Image to enlarge.

### 1.1.1 Inputs

1. **Music Database**

The Music Database is the set of songs you want to generate recomnendations from. Initally you define how a *Song* looks like, i.e. you define it has an `artist`, `album` and `title` field for example.

A single *Song* can actually have more attributes than these, but only those three would be used by the computation.

2. **Listen History**

   The second input is the **Listen History**. In contrast to the **Music Database** it is optional (*).

   (*) Although you effectively disable libmunins main feature this way.

### 1.1.2 Internal

**Todo**

Actually write this.

### 1.1.3 Outputs

**Todo**

Actually write this.

### 1.1.4 Attributes

**Todo**

Actually write this.

## 1.2 Glossary

**Song** In libmunin's Context a Song is a set of attributes that have a name and a value. For example a Song might have an `artist` attribute with the value **Amon Amarth**.

Apart from the Attributes, every Song has a unique ID.

**Distance** A distance is the similarity of two songs **a** and **b** expressed in a number between 0.0 and 1.0. The Distance is calculated by the *DistanceFunction* and is cached in the *DistanceMatrix*.

**DistanceFunction** A **DF** is a function that takes two songs and calculates the *Distance* between them.

More specifically, the **DF** looks at all Common Attributes of two songs **a** and **b** and calls a special **DF** attribute-wise. These results are weighted, so that e.g. `genre` gets a higher precedence, and summed up to one number.

**DistanceMatrix** A **DM** caches all calculated Distances. The size of the matrix `D` is the `NxN` if `N` is the number of songs loaded in a *Context*.

You can assume:

$$D(i,j) = D(j,i) \forall i,j \in D$$

$$D(i,i) = 1.0 \forall i \in D$$

**Context** A Context is one handle of libmunin. One Context has one Music Database and one Listen History as Input and outputs Recomnendations based on that.

You can have more than one Context, and therefore more than one Stream of Recomnendations.

CHAPTER

TWO

# DEVELOPER SECTION

**Public API:**

## 2.1 context.h

### 2.1.1 Description

A *Context* is a Handle to libmunin. You can generate recomnendations by feeding a *Context* with a set of songs and, optionally, with the listening history. The structure on C-side is called `MuninCtx`.

You can create a `MuninCtx` with `munin_ctx_create()`. When done you should pass it to `munin_ctx_destroy()`

The main purpose of a *Context* is holding the set of songs you want to generate recomnendations from. In order to add *Songs* to the *Context* you can use `munin_ctx_feed()`, but it is very advisable to call `munin_ctx_begin()`/`munin_ctx_commit()` before/after if you add many songs. You should be aware that adding a song means calculating quite some stuff. Packing it in a Transaction reduces this overhead significantly.

---

**Todo**

Tell reader about AttributeMask.

---

### 2.1.2 Usage Example

```
#include <stdlib.h>
#include <munin/context.h>

int main(void)
{
    /* Create a new Context */
    MuninCtx *ctx = munin_ctx_create();

    /* Begin a new Transaction */
    munin_ctx_begin(ctx);

    for(int i = 0; i < 100; ++i) {
        long song_id = munin_song_new();
        muning_song_set(song_id, "artist", "Amon Amarth");
        munin_ctx_feed(ctx, song_id);
    }
```

```
    /* Commit all feeded songs to the db */
    munin_ctx_commit(ctx);

    /* Kill all associated ressources */
    munin_ctx_destroy(ctx);
    return EXIT_SUCCESS;
}
```

### 2.1.3 Reference

**Types:**

**MuninCtx**
Member of this structure should not be accessed directly.

---

**Functions:**

MuninCtx * **munin_ctx_create** (void)
Allocates a new *Context*.

**Returns** A MuninCtx, pass it to `munin_ctx_destroy()` when done

void **munin_ctx_destroy** (MuninCtx * *ctx*)
Destroys a *Context* and all associated memory.

**Ctx** On what context to operate.

void **munin_ctx_begin** (MuninCtx * *ctx*)
Before adding songs to the database a transaction has to be opened. This speeds up adding many songs (like the initial import) quite a bit since adding a song involves calculating a *Distance* to every other *Song*.

You can call `munin_ctx_feed()` in a begin/commit block.

**Ctx** On what context to operate.

void **munin_ctx_commit** (MuninCtx * *ctx*)
Add all feeded songs to the database at once.

Calling this without `munin_ctx_begin()` before is an error.

**Ctx** On what context to operate.

void **munin_ctx_feed** (MuninCtx * *ctx*, long *song_id*)
Feed a Song to the Context. Future Recomnendations might contain this song now.

**Ctx** On what context to operate.

**Song_id** The Song to add, it is referenced by an ID.

void **munin_ctx_remove** (MuninCtx **ctx*, long *song*)
Removes a song from the Context.

**Ctx** The context to operate on.

**Song** a SongID

## 2.2 song.h

### 2.2.1 Description

A Song is the elementar node in *libmunin*.

In order to be fully threadsafe there is no structure named `MuninCtx`, since it may be freed behind your back when you look. If you'd continue to use it, BadThings™ would happen. Instead a Song is identified by a unique Integer-ID.

The main purpose of a Song is to set attributes to it. You can set all attributes you previously set in the *Context*'s AttributeMask.

The ususal attributes are:

- `artist`
- `album`
- `title`
- `releaseartist`
- `duration`
- `genre`
- `mood`
- `track`
- `rating`
- `date`

It is recomned to use these as a convention. You can of course define tags as you wish to. Here's a list of attributes you can get inspiration from:

http://wiki.musicbrainz.org/MusicBrainz_Picard/Tags/Mapping

**Warning: Memory Management:**
*libmunin* will **NOT** copy the attributes you set. BadThings™ will happen if you free the data you set. This decision was made in order to be able to handle very large sets of songs without memory penality. If you wish to copy the attribute use `strdup()` and register a free function when creating the AttributeMask.

### 2.2.2 Usage Example

```
long song = munin_song_create(ctx);

munin_song_begin(ctx, song);
munin_song_set(ctx, song, "artist", "Debauchery");
munin_song_set(ctx, song, "artist", "Death Metal Warmachine");
munin_song_commit(ctx, song);

printf("%s\n", munin_song_get(ctx, "artist"));

/* Add it to the set */
munin_ctx_feed(ctx, song);

/* Oh, crap didn't want to feed it actually */
munin_ctx_remove(ctx, song);
```

### 2.2.3 Reference

**Functions:**

long **munin_song_create** (MuninCtx *ctx)
Create a new song.

**Ctx** The context to operate on.

**Returns** An ID that references a Song.

void **munin_song_begin** (MuninCtx *ctx, long song)
Begin editing a song.

**Ctx** The context to operate on.

**Song** a SongID

void **munin_song_commit** (MuninCtx *ctx, long song)
Commit edits to a song. Causes every *Distance* to be rebuild for this song.

**Ctx** The context to operate on.

**Song** a SongID

const char * **mn_song_get** (MuninCtx *ctx, long song, const char *key)
Get an Attribute from a song.

**Ctx** The context to operate on.

**Song** a SongID

**Key** The attribute name

void **munin_song_set** (MuninCtx *ctx, long song, const char *key, const char *value)
Set an attribute of the song.

**Ctx** The context to operate on.

**Song** a SongID

**Key** The attribute name

**Value** The value to set

bool **munin_song_is_valid** (MuninCtx *ctx, long song)
Check if the ID passed as **song** is actually valid, i.e. if the ID exists and the song was not removed.

**Ctx** The context to operate on.

**Song** a SongID

**Returns** True if the Song is valid

---

**Todo**

Define API for MuninAttrIter

---

## 2.3 iterator.rst

### 2.3.1 Description

### 2.3.2 Usage Example

```
//
```

### 2.3.3 Reference

**Functions:**

## 2.4 history.rst

### 2.4.1 Description

The Listen History is the second Input to to *libmunin*. You can feed listened songs as a hint to *libmunin* which will base the next recomnendations based on them. These are the different scenarios:

| Playcount | Recent Recommendation? | Effect |
|---|---|---|
| n | False | |
| n | True | |

### 2.4.2 Usage Example

```
//
```

### 2.4.3 Reference

**Functions:**

bool **munin_history_feed**(MuninCtx * *ctx*, long *song*)

Add a song the the History Buffer. *libmunin* will automatically check if the song was recomned lately and use this for further recomendations.

**Ctx** The Context to operate on.

**Song** A song to feed.

**Returns** True if the song was in the recomendations recently given.

**Private API:**

## 2.5 persistence.rst

### 2.5.1 Description

### 2.5.2 Usage Example

```
//
```

### 2.5.3 Reference

**Functions:**

CHAPTER

THREE

# USER SECTION

## 3.1 Command Line Utility

This document is about the command utility of *libmunin* called *naglfar*.

### 3.1.1 Options

```
Synopsis:
    naglfar [genopts] [command] [options]

Usage:
    naglfar -h | --help
    naglfar -v | --version
    naglfar database [-f|--file <FILE>] [--subsitute|-s]
    naglfar history  [-f|--file <FILE>]

database:
```

**-f** | -file <FILE>
: Read Database from <FILE>. If no file is specified, read from stdin.

  Format:

  **Todo**

  Specify format

**-s** | -subsitute
: Subistute current database contents with this

CHAPTER

FOUR

# INDICES AND TABLES

- *genindex*
- *modindex*
- *search*