Troi

Release 0.1.0

MetaBrainz Foundation

Apr 26, 2024

TROI DOCUMENTATION INDEX

The Troi Playlisting Engine combines all of ListenBrainz' playlist efforts:

1. Playlist generation: Music recommendations and algorithmic playlist generation using a pipeline architecture that allows easy construction of custom pipelines that output playlists. You can see this part in action on ListenBrainz's Created for You pages, where we show of Weekly jams and Weekly Discovery playlists. The playlist generation tools use an API-first approach were users don't need to download massive amounts of data, but instead fetch the data via APIs as needed.

2. Local content database: Using these tools a user can scan their music collection on disk or via a Subsonic API (e.g. Navidrome, Funkwhale, Gonic), download metadata for it and then resolve global playlists (playlist with only MBIDs) to files available in a local collection. We also have support for duplicate file detection, top tags in your collection and other insights.

3. Playlist exchange: We're in the process of building this toolkit out to support saving/loading playlists in a number of format to hopefully break playlists free from the music silos (Spotify, Apple, etc)

CHAPTER

ONE

USER GUIDE

For end-user guide on how to run Troi and what the various command line arguments are, please see our *User Guide*.

CHAPTER

TWO

METABRAINZ APIS FOR PLAYLISTING AND RECOMMENDATION

To accomplish the goal of an API-first toolkit, we, have created and hosted a number of data-sets that can be accessed as a part of this project. From Troi you can call any API you'd like, including the MusicBrainz and ListenBrainz APIs. We have also created the following sites with more API endpoints to support Troi:

1. More stable APIs are hosted on our Labs API page. We work hard to ensure that these APIs stay up at all times, but we do not guarantee it. Best to not use for production.
2. More transient APIs that we do not guarantee to always be up can be found on our data sets page. Do not use for production!

The ListenBrainz project offers a number of data sets:

1. Collaborative filtered recordings that suggest what recordings a user should listen to based on their previous listening habits. See the recommended tracks for user rob.
2. User statistics that were derived from users recent listening habits.

We will continue to build and host more datasets as time passes. If an API endpoint becomes useful to a greater number of people we will elevate these API endpoints to officially supported endpoints that we ensure are up to date on online at all times.

CHAPTER

THREE

TRIVIA

The project is named after Deanna Troi, the empath on the TV series Star Trek: The Next Generation.

3.1 Installation

3.1.1 Installation for End Users

Troi is available for download via PyPi:

```
pip3 install troi
troi --help
```

3.1.2 Installation for Development

Linux and Mac

Use these command line arguments to install Troi on Linux and Mac:

```
virtualenv -p python3 .ve
source .ve/bin/activate
pip3 install .[tests]
python3 -m troi.cli --help
```

Windows

Use these commands to install on Windows:

```
virtualenv -p python .ve
.ve\Scripts\activate.bat
pip install .[tests]
python -m troi.cli --help
```

3.2 User Guide

3.2.1 Global Playlist generation

Once you have installed Troi, get familiar with how the command line works. To get the usage for Troi:

```
troi --help
```

To list all the patches that are currently available, do:

```
troi list
```

To generate a global (MBID based) playlist from a patch and display it in the terminal, do:

```
troi playlist --print daily-jams <user_name>
```

This will generate a playlist and print the list to the terminal.

To generate a global playlist from a patch and display it and then upload it to ListenBrainz:

```
troi playlist --print --upload --token <user-token> daily-jams <user_name>
```

This will generate a playlist and print the list to the terminal and then upload it to ListenBrainz. You can find your user token on your profile page at ListenBrainz.

From here on you can explore different *Patches* or read how Troi works *Playlist generation technical Introduction*

3.2.2 Local Playlist Generation

Local playlists are playlists that have been resolved against a local collection and are playable to a local player. In order for this to work, you will need to index your local collection using the database tools first.

IMPORTANT: Local playlist generation only works if you music collection is tagged with MusicBrainz tags. We recommend Picard for tagging your collection.

If you're unwilling to properly tag your collection, then please do not contact us to request that we remove this requirement. We can't. We won't. Please close this tab and move on.

Index your music collection

If you have your collection hosted on an app like Funkwhale, Navidrome or Gonic, who have a Subsonic API, you can generate playlists directly the web application. Alternatively, if you music collection isn't available via a Subsonic API, you can scan a local collection of files and make local m3u playlists.

Note: We recommend that you scan *either* a local filesystem collection or a subsonic API hosted collection. Doing both is going to result in erratic behaviour of the content resolver.

Setting up config.py

While it isn't strictly necessary to setup **config.py**, it makes using the troi content resolver easier:

```
cp config.py.sample config.py
```

Then edit **config.py** and set the location of where you're going to store your resolver database file into **DATABASE_FILE**. If you plan to use a Subsonic API, then fill out the Subsonic section as well.

If you decide not to use the **config.py** file, make sure to pass the path to the DB file with **-d** to each command. All further examples in this file assume you added the config file and will therefore omit the **-d** option.

You can also define a set of directories to be used by default for the scan command in **MUSIC_DIRECTORIES**.

Options for saving playlists

The playlist generation functions below print the generated playlist and nothig else. In order to save the playlists or upload them, please refer to *Command line options*

Scanning your local filesystem collection

Then prepare the index and scan a music collection. mp3, m4a, wma, OggVorbis, OggOpus and flac files are supported.

```
troi db create
troi db scan <one or more paths to directories containing audio files>
```

If you configured **MUSIC_DIRECTORIES** in config file, you can just call **troi db scan**. It should be noted paths passed on command line take precedence over this configuration.

If you remove tracks from your collection, use **cleanup** to remove references to those tracks:

```
troi db cleanup
```

Scan a Subsonic collection

To scan a subsonic collection, you'll need to setup a config.py file. See above.

```
resolve subsonic
```

This discovers the files present in the subsonic API hosted collection and adds a reference to the local DB.

Metadata Download

In order to use the LB Local Radio playlist generator you'll need to download more data for your MusicBrainz tagged music collection.

First, download tag and popularity data:

```
troi db metadata
```

ListenBrainz Radio Local

ListenBrainz's LB Radio feature generates global playlists that can be resolved to streaming services. Troi also supports a local version that resolved tracks against a local collection of music.

Currently artist and tag elements are supported for LB Radio Local, which means that playlists from these two elements are made from the local collection and thus will not need to be resolved. All other elements may generate playlists with tracks that are not availalble in your collection. In this case, the fuzzy search will attempt to make the missing tracks to your collection.

For a complete reference to LB Radio, see the *LB Radio Prompt Reference*

The playlist generator works with a given mode: "easy", "medium" and "hard". An easy playlist will generate data that more closely meets the prompt, which should translate into a playlist that should be easier and pleasent to listen to. Medium goes further and includes less popular and more far flung stuff, before hard digs at the bottom of the barrel.

This may not always feel very pronounced, especially if your collection isn't very suited for the prompt that was given.

Artist Element

```
troi lb-radio easy 'artist:(taylor swift, drake)'
```

Generates a playlist with music from Taylor Swift and artists similar to her and Drake, and artists similar to him.

Tag Element

```
troi lb-radio easy 'tag:(downtempo, trip hop)'
```

This will generate a playlist on easy mode for recordings that are tagged with "downtempo" AND "trip hop".

```
troi lb-radio medium 'tag:(downtempo, trip hop)::or'
```

This will generate a playlist on medium mode for recordings that are tagged with "downtempo" OR "trip hop", since the or option was specified at the end of the prompt.

You can include more than on tag query in a prompt:

```
troi lb-radio medium 'tag:(downtempo, trip hop)::or tag:(punk, ska)'
```

Stats, Collections, Playlists and Recommended recordings

There are more elements, but these are "global" elements that will need to have their results resolved to the local collection. The resolution process is always a bit tricky since its outcome heavily depends on the collection. The generator will do its best to generate a fitting playlist, but that doesn't always happen.

For the other elements, please refer to the *LB Radio Prompt Reference*

Resolve JSPF playlists to local collection

First, find a playlist on ListenBrainz that you'd like to resolve to a local collection:

```
https://listenbrainz.org/user/{your username}/playlists/
```

Then download the JSPF file:

```
curl "https://api.listenbrainz.org/1/playlist/<playlist MBID>" > playlist-test.jspf
```

Finally, resolve the playlist to local files:

```
troi resolve playlist-test.jspf playlist-test.m3u
```

Then open the m3u playlist with a local player.

Create Weekly-Jams Local Playlists

To create a weekly-jams recommendation playlist for a local collection run the weekly-jams command and give the ListenBrainz username for whom you wish to create a playlist for:

```
troi weekly-jams <LB user name>
```

3.3 Command line options

3.3.1 Playlist generation options

The full list of command line options for each of the commands is below:

3.3.2 Database / content resolution options

3.4 Playlist generation technical Introduction

3.4.1 Patches

A patch (data pipeline) is constructed by instantiating Element objects and then chaining them together with set_sources() methods. If you want to create a patch with two elements, the following code could be used:

```
element0 = MyElement()
element1 = MyOtherElement()
element1.set_sources(element0)
```

The most common data passed through a data are Recording objects that represent a MusicBrainz recording. But other data elements such as Users, Artists and Releases could easily be objects passed through a pipeline. For instance, to create a Recording object, you'll need to at least pass in a name of the Recording:

class troi.__init__.**Recording**(*name=None*, *mbid=None*, *msid=None*, *duration=None*, *artist_credit=None*, *release=None*, *ranking=None*, *year=None*, *spotify_id=None*, *musicbrainz=None*, *listenbrainz=None*, *acousticbrainz=None*)

The class that represents a recording.

Very often Recordings are created with name and mbid arguments and then the `RecordingLookupElement` is used to automatically lookup all the needed data (e.g. artist).

A patch never processes data on its own – a Patch returns a constructed pipeline of Elements that are chained together. The Elements are the classes that process data, but those are invoked only after the pipeline is constructed and when Troi begins to generate a playlist.

If is not uncommon to have a PlaylistRedundancyReducerElement at the end of the pipeline which returns a Playlist object of the desired length with a limit to the number of times a single artist's tracks can be included in the playlist.

3.4.2 Entities

The pre-defined elements User, Artist, Release, Recording all have elements of MBID and name. A recording also had an Artist element included for obvious reasons. The Artist element actually contains support for more than one artist, so its MBID element is actually plural MBIDS in order to accurately represent a MusicBrainz Artist Credit. Each of these objects contains two free form dicts: musicbrainz and listenbrainz. When you fetch data that doesn't fit into the predefined fields of the data object classes, that data can be stored in these free form dicts. For instance, a Recording element that was loaded from ListenBrainz statistics will have a listen_count field.

3.4.3 Elements

Troi uses a pipeline architecture comprised of Elements (a node in the pipeline) that can be chained togther. Each element `Element` can be a source of data and act like a filter that takes the incoming data, acts on it and passes output data. At the end of the pipeline, the last element should return a Playlist object that the Troi main function can then display or submit to ListenBrainz or other services.

The important methods of this class are:

class `troi.__init__.`**`Element`**(*patch=None*)

Base class for elements

static **`inputs`**()

Return a list of Artist, ArtistCredit, Release or Recording classes that define the type and number of input lists to this element. e.g. [Artist, ArtistCredit, Recording] means that this element expects a list of artist credits and a list of recordings for inputs.

static **`outputs`**()

Return a list of Artist, ArtistCreditm Release or Recording classes that define the type and number of output lists returned by this element. e.g. [Artist, ArtistCredit, Recording] means that this element returns a list of artist credits and a list of recordings.

`set_sources`(*sources*)

Set the source elements for this element.

Parameters

`sources` – A list of objects derived from the Element class, containing at least 1 class. When the pipeline is executed the output of the provided class(es) read function will become the input to this class.

Each class derived from Element must override these three functions.

3.5 Patches

We have a few patches that ship with Troi – please note that not all of them might be documented here. Some are still in development or for internal use; we're keeping them there as examples that might help others learn how to use Troi.

Patches can be found in the *troi/patches* directory.

3.5.1 Area Random Recordings

area-random-recordings

Given a geograpic area (defined in terms of an area in MusicBrainz) and start/end year, choose random tracks from this given country and return a playlist of these tracks.

3.5.2 Daily Jams

daily-jams

This is our first attempt to create recommended playlists for our users. Daily Jams is a playlist designed for the user to put on in the background and to "just have some good tunes without having to think". It should be all feel-good tracks with nothing new being introduced.

You can create your own daily jams from this patch.

3.5.3 Playlist from MBIDs

playlist-from-mbids

Given a list of MBIDs, make a playlist from it. Useful for converting a list of MBIDs and to a playlist and then submitting that playlist to ListenBrainz.

3.5.4 Recommendations to Playlist

recs-to-playlist

Fetch ListenBrainz recommended tracks and upload them to ListenBrainz as a playlist.

3.5.5 Weekly Flashback Jams

weekly-flashback-jams

Generate playlists for past decades based on your listening history.

3.5.6 World Trip

world-trip

Given a continent and whether to sort the playlist via longitude or latitude, generate a playlist with tracks from that continent.

3.6 Entities

Our entity classes are simple data containers that are laid out exactly as the entities in MusicBrainz are. The idea is that these classes are flexible and do not require a lot of data. Usually a name and or MBID are sufficient for creating an entity.

Each of our entities have two free form dicts intended to collect data that Troi might make use of in the data pipeline: musicbrainz and listenbrainz. If we fetch some top listened recordings from ListenBrainz, we might receive a listen_count for that Recording. We can simply store listen_count in the listenbrainz dict and then use it in any Element in the pipeline.

3.6.1 Artist

The Artist entity contains the name, the MBIDs and/or artist_credit_ids for a MusicBrainz artist. You can create an Artist object with the following parameters:

`class troi.__init__.`**`Artist`**(*name=None*, *mbid=None*, *artist_id=None*, *join_phrase=None*, *ranking=None*, *musicbrainz=None*, *listenbrainz=None*, *acousticbrainz=None*)

The class that represents an artist.

3.6.2 Release

The Release entity contains the name, the MBIDs and the Artist for a MusicBrainz artist. You can create an Release object with the following parameters:

`class troi.__init__.`**`Release`**(*name=None*, *mbid=None*, *artist_credit=None*, *ranking=None*, *caa_id=None*, *caa_release_mbid=None*, *musicbrainz=None*, *listenbrainz=None*, *acousticbrainz=None*)

The class that represents a release.

3.6.3 Recording

The Recording entity, which is the most used entity in Troi, contains the name, the MBIDs and the Artist and possible Release for a MusicBrainz artist. You can create an Recording object with the following parameters:

`class troi.__init__.`**`Recording`**(*name=None*, *mbid=None*, *msid=None*, *duration=None*, *artist_credit=None*, *release=None*, *ranking=None*, *year=None*, *spotify_id=None*, *musicbrainz=None*, *listenbrainz=None*, *acousticbrainz=None*)

The class that represents a recording.

3.6.4 User

The User entity represents a ListenBrainz user – this may not be very useful for anyone but the ListenBrainz team to run Troi to generate playlists for our users:

`class troi.__init__.User`(*user_name=None*, *user_id=None*)

The class that represents a ListenBrainz user.

3.7 Elements

3.7.1 troi.filters

Elements that are used as filter to remove some parts of the data passed into it:

`class troi.filters.ArtistCreditFilterElement`(*artist_credit_ids*, *include=False*)

Remove recordings if they do or do not belong to a given list of artists.

Parameters

- **`artist_credit_ids`** – A list of artist_credit_ids to remove/keep
- **`include`** – If true, include all tracks with the given artist_credit_id, otherwise remove them.

`class troi.filters.ArtistCreditLimiterElement`(*count=2*, *exclude_lower_ranked=True*)

This element examines there passed in recordings and if the count of recordings by any one artists exceeds the given limit, excessive recordigns are removed. If the flag exclude_lower_ranked is True, and each recording has a "ranked" key in the musicbrainz dict, then the lowest ranked recordings are removed, otherwise the highest ranked recordings are removed. Throws PipelineError is not all recordings have artist_credit_ids set.

Parameters

- **`count`** – The number of duplicate aritst_credits to allow in the output
- **`exclude_lower_ranked`** – Remove the lower ranked duplicates, if rankings are present.

`class troi.filters.DuplicateRecordingMBIDFilterElement`(*patch=None*)

This Element takes a list of recordings and removes any duplicate recordings based on the recording's MBID, preserving the input order.

`class troi.filters.DuplicateRecordingArtistCreditFilterElement`(*patch=None*)

This Element takes a list of recordings and removes any duplicate recordings based on the recording's name and artist_credit name, preserving the input order.

`class troi.filters.ConsecutiveRecordingFilterElement`(*patch=None*)

This Element takes a list of recordings and removes consecutive duplicate recordings based on the recording's MBID

For example, a sequence A, A, A, B, B, A, C will be reduced to A, B, A, C

`class troi.filters.EmptyRecordingFilterElement`(*patch=None*)

This Element takes a list of recordings and removes ones that have an empty name or artist.

`class troi.filters.YearRangeFilterElement`(*start_year*, *end_year=None*, *inverse=False*)

Filter a list of Recordings based on their year – the year must be between start_year and end_year, otherwise the recording will be filtered out. If no end_year is given, keep (or reject in case of inverse) tracks greater or equal to start_year. If inverse=True, then keep all Recordings that do no fit into the given year range.

Parameters

- **start_year** – The full start year to filter (inclusive).
- **end_year** – The full end year to filter (inclusive).
- **inverse** – If inverse is True, exclude everything in the year range.

class troi.filters.**GenreFilterElement**(*genre_list*)

Keep recorindgs that have at least one genre in commong from the list passed in when this class is created.

Parameters
genre_list – A list of genre trags to filter out.

class troi.filters.**LatestListenedAtFilterElement**(*min_number_of_days=14*)

Filter the recordings according to latest_listened_at field in the lb metadata. If that field is None, treat it as if the user hasn't listened to this track recently or at all and keep the track in the list.

Parameters
min_number_of_days – The number of tracks that must have passed for a track to be kept.

class troi.filters.**HatedRecordingsFilterElement**(*patch=None*)

Remove recordings that have been hated by the user

3.7.2 troi.loops

Elements useful for runnning Troi for many users:

class troi.loops.**ForLoopElement**(*patch_slugs*, *patch_args*)

An element that receives items from a pipeline and for each item in the pipeline it instantiates a new patch and executes that patch. A normal use case might include taking a list of users and running the specified patch for each user.

As of right now, only User objects can be processed with this element.

Parameters

- **patch_slug** – the slug of the patch to run inside the for loop
- **pipeline_args** – the arguments passed to top-level pipeline, so that they apply to dynamically created pipelines inside the for loop

3.7.3 troi.operations

Elements that perform operations on the data pipeline, such as union, difference, intersection and uniqing.

class troi.operations.**UniqueElement**(*key='mbid'*)

Make this passed list of entities unique base on the passed key (must be one of name or mbid) and return the unique list. recordings also allow msid as key and artists allow artist_credit_id as key. Currently the order of the list is not preserved. This should be improved on…

Parameters
key – Which key to use for making the list unique. Defaults to "mbid".

class troi.operations.**UnionElement**

Combine both entities lists into one

class troi.operations.**IntersectionElement**(*key='mbid'*)

Return the list of entities that exist in both entities lists.

class troi.operations.**DifferenceElement**(*key='mbid'*)

Return the list of recordings in entities_0 minus those in entities_1

class troi.operations.**ZipperElement**

Given two or more inputs, pick recordings from each alternatingly

3.7.4 troi.sorts

Elements that sort the data in a pipeline:

class troi.sorts.**YearSortElement**(*reverse=False*)

Sort recordings by year ascending – recordings that have the same year will be returned in an undefined order. Recordings that have no year set will be returned at the end of the list. If reverse=True, sort descending and return tracks with no year first.

Parameters

reverse – Reverse the sort order.

3.8 MusicBrainz Elements

The following elements fetch data from MusicBrainz:

3.8.1 troi.musicbrainz.mbid_mapping

Look up a Recording in the ListenBrainz MBID mapper from only an Artist.artsit_credit_name and a Recording.name.

3.8.2 troi.musicbrainz.mbid_reader

Load MBIDs from a file and return a list of Recording elements:

3.8.3 troi.musicbrainz.recording_lookup

Retrieve metadata for Recordings that have their MBID set, but other metadata is missing. This Element is useful for taking a list of Recording MBIDs and turning them into a full set of Recording objects.

3.8.4 troi.musicbrainz.recording

Given a list of Recording objects, return them from the Element. This is useful if something has generated Recordings that will need to be processed by Troi.

class troi.musicbrainz.recording.**RecordingListElement**(*recordings*)

This element is used to pass a provided list of Recordings into the pipeline.

Parameters

recordings – The recordings to return from this Element.

3.8.5 troi.musicbrainz.year_lookup

Given a list of Recording objects fetch, fetch the year when they were released, using Recording.artist.name and Recording.name.

NOTE: This lookup does not use MBIDs!

3.9 ListenBrainz Elements

The following elements fetch data from ListenBrainz:

3.9.1 troi.listenbrainz.dataset_fetcher.DataSetFetcherElement

ListenBrainz has developed a tool called the dataset hoster which allows us to quickly host SQL queries on the web. This Element is a shortcut for fetching data from one of these endpoints and to return a list of Recordings.

3.9.2 troi.listenbrainz.feedback.ListensFeedbackLookup

Given a list of Recordings as input, fetch the feedback (like/hate) for a given user_name.

`class troi.listenbrainz.feedback.`**`ListensFeedbackLookup`**(*user_name*, *auth_token=None*)

Element to look up the user's feedback for the given Recordings.

Parameters

`user_name` – The ListenBrainz user_name for whom to fetch feedback information.

3.9.3 troi.listenbrainz.listens.RecentListensTimestampLookup

Given a list of Recordings, fetch the timestamp for when that Recording was listened to in the window specified by the days parameter.

`class troi.listenbrainz.listens.`**`RecentListensTimestampLookup`**(*user_name*, *days: int*, *auth_token=None*)

Element to look up the time when a user last listened given recordings in past X days. Note that the element is stateful and caches the recent listens lookup results!

Timestamps are stored in the listenbrainz dict, with key name "latest_listened_at".

Parameters

- **`user_name`** – The ListenBrainz user for whome to fetch recent listen timestamps.
- **`auth_token`** – a ListenBrainz auth token
- **`days`** – The number of days to check.

3.9.4 troi.listenbrainz.recs.UserRecordingRecommendationsElement

Given a user_name and artist_type, fetch Recordings for that user if they are available. artist_type must be one of "top" for top artist recommendations (for that user), "similar" for similar artist recommendations (also for that user) or "raw" recommendations that have not be filtered like the top/similar recommendations.

3.9.5 troi.listenbrainz.stats.UserArtistsElement

Given a user_name and a time_range, fetch the top artist statistics for that user and time_range. Available time_ranges are defined in the ListenBrainz Statistics API documentation.

3.9.6 troi.listenbrainz.stats.UserReleasesElement

Given a user_name and a time_range, fetch the top release statistics for that user and time_range. Available time_ranges are defined in the ListenBrainz Statistics API documentation.

3.9.7 troi.listenbrainz.stats.UserRecordingsElement

Given a user_name and a time_range, fetch the top recording statistics for that user and time_range. Available time_ranges are defined in the ListenBrainz Statistics API documentation.

3.9.8 troi.listenbrainz.user.UserListElement

Given a list of users, return those users as User objects.

class `troi.listenbrainz.user.`**`UserListElement`**(*user_list*)

This element is used to pass a provided list of users into the pipeline.

Parameters

`user_list` – A list of user_names

3.9.9 troi.listenbrainz.yim_user.YIMUserListElement

This Element is used for when we run the Year in Music report and create playlists. The elment fetches the list of users who should have YIM playlists generated and returns them.

3.10 LB Radio Prompt Reference

ListenBrainz Radio is a powerful playlist/radio generation tool that gives users a lot of power to automatically make playlists. The troi toolkit offers two types of LB Radio playlists: Global and local. Global playlists generated with the lb-radio patch via the playlist command are made from all the available recorings available in MusicBrainz and contain only MusicBrainz MBIDs. Normal music players cannot play these playlists. LB Radio Local creates local playlists that are fully resolved against a local playlist and when converted to the m3u format are playable with a local playlist.

To generate a playlist, the user will need to enter an "LB Radio prompt", which is a type of search query that specifies what music should be added to the playlist. A Radio prompt is composed of one or more terms, taking the form:

```
entity:values:weight:option
```

Each term generates a stream of recordings and recordings from each of the streams are then interleaved to make a single playlist. The entity must be either “artist” or “tag” currently. The optional weight argument allows the user to control how often this term will contribute to the final playlist. By default each term gets a value of 1, if the user didn’t specify a weight value. A term with a weight of 3 will contribute 3 times more recordings than a term with weight 1. The final part of each term are options, documented in the options section.

3.10.1 Entities

The LB Radio supports the following entities:

1. **artist**: Play tracks from this artists and similar artists
2. **tag**: Play tracks from one of more tags
3. **collection**: Use a MusicBrainz collection as a source of recordings. (mode does not apply to collections)
4. **playlist**: Use a ListenBrainz playlist as a source of recordings. (mode also does not apply to playlists)
5. **stats**: Use a ListenBrainz user’s statistics as a source of recordings.
6. **recs**: Use a ListenBrainz user’s recommended recordings as a source of recordings.
7. **country**: Select recordings from artists who are from the given country.

3.10.2 Options

All terms have the following options:

1. **easy**: Use easy mode for this term. See modes below.
2. **medium**: Use medium mode for this term.
3. **hard**: Use hard mode for this term.

For artist and tag terms, the following option applies:

1. **nosim**: Do not add similar artists/tags, only output recordings from the given artist/tag.

For tag queries, the following options exist:

1. **and**: For a tag query, if “and” is specified (the default) recordings will be chosen if all the given tags are applied to that recording.
2. **or**: For a tag query, if “or” is specified, then recordings will be chosen if any of the tags are applied to the recording.
3. **nosim**: Tag queries on medium and hard mode may include similar tags. Specifying nosim for a tag query ensures that no similar tags are used.

For the stats term, the following options apply:

1. **week**, **month**, **quarter**, **half_yearly**, **year**: Stats for the user for the past week, month, quarter, half_yearly, year, respectively.
2. **all_time**: Stats for all time, covering all listens for the user.
3. **this_week**, **this_month**, **this_year**: Stats for the user for the current week, month, year, respectively.

For the recs term, the following options apply:

1. **listened**: Fetch recommeded recordings that the user has listened to. Useful for making “safe” playlists.
2. **unlistened**: Fetch recommeded recordings that the user has not listened to. Useful for making “exploration” playlists.

3.10.3 Modes

Along with a prompt, the user will need to specify which mode they would like to use to generate the playlist: easy, medium or hard.

The core functionality of LB radio is to intelligently, yet sloppily, pick from vast lists of data to form pleasing playlists. Almost all of the data sources (similar artists, top recordings of an artist, user stats, etc) are ordered lists of data, with the most relevant data near the top and less revelvant data near the bottom. Broadly speaking, the three modes divide each of these datasets into three chunks: easy mode will focus on the most relevant data, medium on the middle relevant data and hard on the tail end.

For almost all of the source entities (see above), this applies in a pretty staightforward manner: Whenever an ordered list of data exists, we use the modes to inform which section of data we look at. However, the tag element is an entirely different beast. Roughly speaking, easy mode attempts to fetch recordings tagged with the given tag, medium mode picks tags from release/release-group tags and hard mode picks tagged recordings from artists. In reality there are a lot more nuances in this process. What if there aren't enough tracks to make a reliable easy playlist? Then don't make one and let the user know they could try again on medium mode and that they would get a playlist. There are other heuristics baked into the tag query that are not easy to describe and quite likely will change in the near future as we respond to community feedback. Once we're comfortable that the tag entities is working well, we will improve these docs.

This idea of modes comes from video games, where players can choose how hard the game should be to play. In the context of LB Radio, the resultant playlist will also be more work to listen to the harder the mode. Which mode to use is entirely up to the user – easy is likely going to create a playlist with familiar music, and a hard playlist may expose you to less familiar music.

3.10.4 Syntax Notes

Artist and tag names are the tricky bits to specify in a prompt, so they must be enclosed with ():

```
artist:(Blümchen)
tag:(deep house)
artist:( )
```

Furthermore, artist names must be spelled exactly as their appear in MusicBrainz. If you have difficulty specifying the correct artist, you can use an artist MBID to be very precise.

Tags have similar restrictions. If a tag you'd like to specify has no spaces or non-latin unicode characters you may use:

```
tag:(punk)
#punk
```

But with spaces or non-latin unicode characters, wrap it in () and use the full tag element name:

```
tag:(hip hop)
```

```
tag:()
```

3.10.5 Simple examples

```
Rick Astley
```

Create a single stream, from artist Rick Astley and similar artists. Artist names must be spelled here exactly as they are spelled in MusicBrainz. If for some reason the artist name is not recognized, specify an MBID instead. See below.

```
#punk
```

The # shorthand notation allows user to quickly specify a tag radio, but it only works for one tag and the tag cannot contain spaces. For more advanced prompts, use the full notation described above.

```
tag:(rock,pop)::or
```

This prompt generates a playlist with recordings that have been tagged with either the “rock” OR “pop” tags. The weight can be omitted and will be assumed to be 1.

```
tag:(rock) tag:(pop)
```

Create two streams, one from tag “rock” contributing 3 parts of the recordings and one from tag “pop” contibuting 2 parts of the recordings.

```
tag:(trip hop)
```

Tags that have a space in them must be enclosed in (). Specifying multiple tags requires the tags to be enclosed in () as well as comma separated.

```
tag:(trip hop, downtempo)
```

If LB-radio does not find your artist, you can specify an artist using an Artist MBID:

```
artist:(8f6bd1e4-fbe1-4f50-aa9b-94c450ec0f11)
```

LB-radio also supports MusicBrainz collections as sources:

```
collection:(8be1a919-a386-45f3-8cc2-0d9249b02aa4)
```

Will select random recordings from a MusicBrainz recording collection – the modes wont have any affect on collections, since collections have no inherent ranking that could be used to select recordings according to mode. :(

```
playlist:(8be1a919-a386-45f3-8cc2-0d9249b02aa4)
```

Will select random recordings from a ListenBrainz playlist – the modes wont have any affect on collections, since plylists have no inherent ranking that could be used to select recordings according to mode. :(

```
stats:lucifer::all_time
```

Will select random recordings from the ListenBrainz user lucifer recordings statistics for all time.

```
recs:mr_monkey::unlistened
```

Will select random recordings from the ListenBrainz user mr_monkey’s recommended recordings that mr_monkey hasn’t listened to.

```
country:(Mali)
```

Will select random recordings from artists who are from the given country. While this features generally represents music from that selected country, some artists leave their home country and don't perform music representative of their country, so this element may not always be 100% on point. But it can still create some very interesting playlists!

3.10.6 More complex examples

```
artist:(pretty lights):3:easy tag:(trip hop):2 artist:(morcheeba)::nosim
```

This prompt will play 3 parts from artist "Pretty Lights", 2 parts from the tag "trip hop" and 1 part from the artist "Morcheeba" with no tracks from similar artists.

```
tag:(deep house):2:medium tag:(metal):1:hard artist:(blümchen):2:easy
```

This will play 2 parts from tag "deep house" on medium mode, 1 part from tag "metal" on hard mode and 2 parts from artists "Blümchen" on easy mode.

3.11 Developer Documentation

The documentation in this section describes the internal workings of Troi. Developers wishing to use Troi shouldn't need to know about these modules, but if you wish to extend the core functionality of Troi, this documentation is for you!

Note: There are some modules in troi directory that are not covered here – those are important for end users, so thos are defined in the main section of our docs.

3.11.1 troi.cli

This module is the entry point for the troi command line interface.

3.11.2 troi.core

Troi core module that creates and executes the patches:

3.11.3 troi.patch

Troi patch class definition:

class troi.patch.**Patch**(*args*)

> **abstract create**(*input_args*)
>
> The function creates the data pipeline and then returns it.
>
> **Params:**
> input_args: the arguments passed to the patch.
>
> **static description**()
>
> Return the description for this patch – this short description (not more than a paragraph) should give the user an idea as to what the patch does.
>
> e.g "Generate a list of random recordings from a given area."

`generate_playlist()`

Generate a playlist

The args parameter is a dict and may containt the following keys:

- quiet: Do not print out anything
- save: The save option causes the generated playlist to be saved to disk.
- token: Auth token to use when using the LB API. Required for submitting playlists to the server. See https://listenbrainz.org/profile to get your user token.
- upload: Whether or not to submit the finished playlist to the LB server. Token must be set for this to work.
- created-for: If this option is specified, it must give a valid user name and the TOKEN argument must specify a user who is whitelisted as a playlist bot at listenbrainz.org .
- name: Override the algorithms that generate a playlist name and use this name instead.
- desc: Override the algorithms that generate a playlist description and use this description instead.
- min-recordings: The minimum number of recordings that must be present in a playlist to consider it complete. If it doesn't have sufficient numbers of tracks, ignore the playlist and don't submit it. Default: Off, a playlist with at least one track will be considere complete.
- spotify: if present, attempt to submit the playlist to spotify as well. should be a dict and contain the spotify user id, spotify auth token with appropriate permissions, whether the playlist should be public, private or collaborative. it can also optionally have the existing urls to update playlists instead of creating new ones.

Parameters

args – the arguments to pass to the patch, may contain one of more of the following keys:

`get_service(`*slug*`)`

Given a service slug, return the class registered for this service.

Raises IndexError if no such service is registered.

`static inputs()`

This function should return a list of dicts that defined the type (argument or option), args, and kwargs to be passed to the click function. MusicBrainz entities and python base types can all be used. The documentation of the method is used as the help returned by the command. Example:

```
[
    {
        "type" : "argument",
        "args": ["num_recordings"],
        "kwargs": {
            "optional": true
        }
    }
]
```

`is_local()`

If this function returns True, it means that the patch expects to use the local database, so Troi should setup the local database before running this patch. Returns False unless overridden by a deriving patch.

`post_process()`

This function is called once the pipeline has produced its playlist, just before the Playlist object is created. This function could be used to inspect data in patch local storage to create the detailed playlist name and descriptionm which may not be available when the pipeline is constructed.

`register_service(`*service*`)`

Register a new service that can provide services to troi patches.

Only one service can be registered for any given service slug at a time. The most recently registered service will be use for the next playlist generation.

`static slug()`

Return the slug for this patch – this is a URL friendly short identifier that can be used to invoke this patch via an HTTP call.

e.g area-random-recordings

`user_feedback()`

Call this function to retrieve a list of strings that give the user feedback about the playlist that was generated, if any.

3.11.4 troi.playlist

Troi playlist class that creates and serializes playlists:

3.11.5 troi.print_recording

Debugging module for printing out the information associated with recordings:

`class troi.print_recording.PrintRecordingList`

Print a list of recordings in a sane matter intended to fit on a reasonably sized screen. It prints recording name and artist name always, and year, bpm, listen_count or moods if they are found in the first recording.

`print(`*entity*`)`

Print out a list(Recording) or list(Playlist).

3.11.6 troi.utils

Misc functions needed to run troi:

`troi.utils.discover_patches()`

Attempt to load patches from the installed patches dir as well as any patches directory in the current dir.

`troi.utils.discover_patches_from_dir(`*module_path*, *patch_dir*, *add_dot=False*`)`

Load patches given the appropriate python module path and then file system path. If add_dot = True, add . to the sys.path and then remove it before this function exists.

`troi.utils.interleave(`*lists*`)`

Return a list with all items from the given lists.

`troi.utils.`**`recursively_update_dict`**(*source*, *overrides*)

Updates the *source* dictionary in place and in a recursive fashion. That is unlike dict1.update(dict2) which would simply replace values of keys even in case one of the values is dict, this method will attempt to merge the nested dicts.

Eg: dict1 - {“a”: {“b”: 1}}, dict2 - {“a”: {“c”: 2}} dict1.update(dict2) - {“a”: {“c”: 2}} recursively_update_dict(dict1, dict2) - {“a”: {“b”: 1, “c”: 2}}

CHAPTER FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX