sir Documentation

Release v1.0.2

Wieland Hoffmann

Dec 29, 2020

Contents

Contents:

CHAPTER 1

Setup

1.1 Installation

1.1.1 Git

If you want the latest code or even feel like contributing, the code is available on Github.

You can easily clone the code with git:

```
git clone git://github.com/metabrainz/sir.git
```

Now you can install it system-wide:

```
python2 setup.py install
```

or start hacking on the code. To do that, you'll need to run at least:

```
python2 setup version
```

once to generate the file `sir/version.py` which the code needs. This file does not have to be added into the git repository because it only contains the hash of the current git commit, which changes after each commit operation.

1.1.2 Setup

The easiest way to run sir at the moment is to use a virtual environment. Once you have virtualenv for Python 2.7 installed, use the following to create the environment:

```
virtualenv venv
source venv/bin/activate
pip install -r requirements.txt
cp config.ini.example config.ini
```

Note: Environment variables can be used in *config.ini* with the syntaxes *$NAME* and *${NAME}*. Undefined variables will not be replaced at all. Escaping is not supported.

You can now use sir via:

```
python -m sir
```

1.2 AMQP Setup

1.2.1 RabbitMQ Server

To set up the exchanges and queues on your RabbitMQ server,

- install RabbitMQ (if you have not already done so)
- start RabbitMQ
- configure your AMQP access data in `config.ini`
- run `python -m sir amqp_setup` to configure the necessary exchanges and queues on your AMQP server.

The default values for the RabbitMQ configuration options can be found in the RabbitMQ documentation.

1.2.2 Database

Sir requires that you both install an extension into your MusicBrainz database and add triggers to it.

AMQP Extension

- Install pg_amqp.
- Check values for the following keys in the file `config.ini`:

Keys	Description
[database] user	Name of the PostgreSQL user the MusicBrainz Server uses
[rabbitmq] host	The hostname that's running your RabbitMQ server
[rabbitmq] user	The username with which to connect to your RabbitMQ server
[rabbitmq] password	The password with which to connect to your RabbitMQ server
[rabbitmq] vhost	The vhost on your RabbitMQ server

The default values for the RabbitMQ configuration options can be found in the RabbitMQ documentation.

- Run `python -m sir extension` once to generate the file `sql/CreateExtension.sql`.
- Connect to your database as a superuser with `psql` to execute from this file.

Triggers

In addition to the steps above, it is necessary to install functions and triggers into the database to send messages via AMQP after a change has been applied to the database. Those can be found in the `sql` directory and can be installed with

```
MB_SERVER_PATH=<mb_path> make installsql
```

where <mb_path> is the path to your clone of the MusicBrainz server.

1.3 Solr

Of course you'll need a Solr server somewhere to send the data to. The mbsssss repository contains instructions on how to add the MusicBrainz schemas to a Solr server.

1.4 Web Service Compatibility

If you have applications that are already able to parse search results from search.musicbrainz.org in the mmd-schema XML or the derived JSON format, you can enable the *wscompat* setting in the configuration file. This will store an mmd-compatible XML document in a field called *_store* for each Solr document. Installing mb-solrquerywriter on your Solr server will then allow you to retrieve responses as mmd-compatible XML or the derived JSON.

CHAPTER 2

Usage

As already mentioned in *Setup*, `python -m sir` is the entry point for the command line interface which provides several subcommands:

`reindex`
: This subcommand allows reindexing data for specific or all entity types (see *The import process* for more information).

`triggers`
: This subcommand regenerates the trigger files in the `sql/` directory.

`amqp_setup`
: This subcommand sets up AMQP exchanges and queues (see *AMQP Setup* for more information).

`amqp_watch`
: This subcommand starts a process that listens on the configured queues and regenerates the index data (see *Queues* for more information).

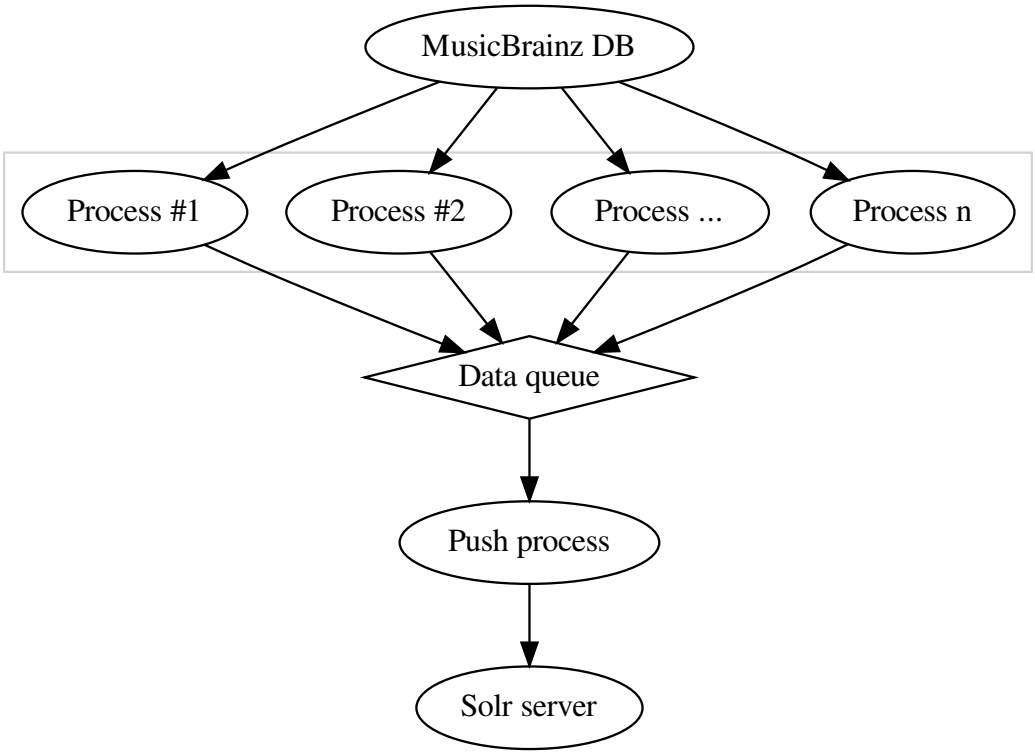
All of them support the `--help` option that prints further information about the available options.

CHAPTER 3

The import process

The process to import data into Solr is relatively straightforward. There's a *SearchEntity* object for each entity type that can be imported which keeps track of the indexable fields and the model in mbdata for that entity type.

Once its known which entity types will be imported, *sir.indexing._multiprocessed_import()* will successivey spawn multiprocessing.Process es via multiprocessing.pool . Each of the processes will retrieve one batch of entities from the database via a query built from *build_entity_query()* and convert them into regular dicts via *query_result_to_dict()*. The result of the conversion will be passed into a multiprocessing.Queue. On the other end of the queue, another process running *sir.indexing.queue_to_solr()* will send them to Solr in batches.



3.1 Paths

Each *SearchEntity* is assigned a declarative via its **model** attribute and a collection of *SearchField* objects, each corresponding to a field in the entities Solr core. Those fields each have one or more paths that "lead" to the values that will be put into the field in Solr. *iterate_path_values()* is a method that returns an iterator over all values for a specific field from an instance of a declarative class and its docstring describes how that works, so here's a verbatim copy of it:

querying.**iterate_path_values**(*obj*)

Return an iterator over all values for *path* on *obj*, an instance of a declarative class by first splitting the path into its elements by splitting it on the dots, resulting in a list of path elements. Then, for each element, a call to `getattr()` is made - the arguments will be the current model (which initially is the **model** assigned to the *SearchEntity*) and the current path element. After doing this, there are two cases:

1. The path element is not the last one in the path. In this case, the `getattr()` call returns one or more objects of another model which will replace the current one.
2. The path element is the last one in the path. In this case, the value returned by the `getattr()` call will be returned and added to the list of values for this field.

To give an example, lets presume the object we're starting with is an instance of *Artist* and the path is "begin_area.name". The first `getattr()` call will be:

```
getattr(obj, "begin_area")
```

which returns an *Area* object, on which the call:

```
getattr(obj, "name")
```

will return the final value:

```
>>> from mbdata.models import Artist, Area
>>> artist = Artist(name="Johan de Meij")
>>> area = Area(name="Netherlands")
>>> artist.begin_area = area
>>> list(iterate_path_values("begin_area.name", artist))
['Netherlands']
```

One-to-many relationships will of course be handled as well:

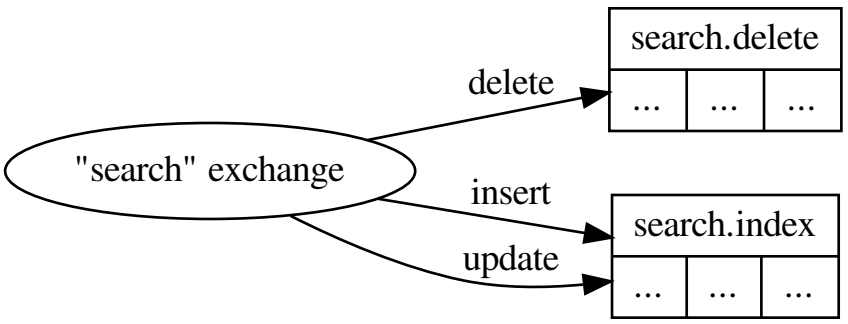
```
>>> from mbdata.models import Recording, ISRC
>>> recording = Recording(name="Fortuna Imperatrix Mundi: O Fortuna")
>>> recording.isrcs.append(ISRC(isrc="DEF056730100"))
>>> recording.isrcs.append(ISRC(isrc="DEF056730101"))
>>> list(iterate_path_values("isrcs.isrc", recording))
['DEF056730100', 'DEF056730101']
```

sir.schema.SCHEMA is a dictionary mapping core names to *SearchEntity* objects.

CHAPTER 4

Queues

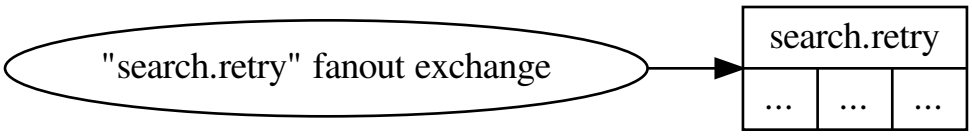
The queue setup is similar to the one used by the CAA indexer:



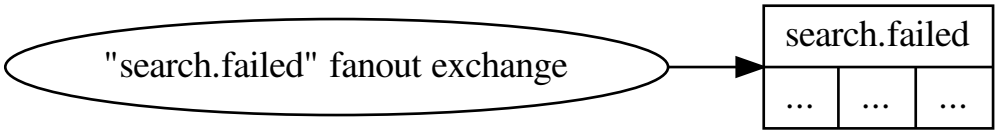
The `search` exchange is the entry point for new messages. It will route them to either the `search.delete` queue or the `search.index` one.

Messages in `search.delete` are used to delete documents from the Solr index without any additional queries by simply calling `solr.Solr.delete_many()` with the ids contained in the message.

For messages in `search.index`, additional queries have to be made to update the data.



If processing any message failed, it will be sent to the `search.retry` queue, which automatically dead-letters them back to `search` after 4 hours for another try.



If processing a message failed too often, it will be put into `search.failed` for manual inspection and intervention.

CHAPTER 5

API

5.1 Indexing

sir.indexing.**reindex**(*args*)
Reindexes all entity types in args["entity_type"].

If no types are specified, all known entities will be reindexed.

Parameters **args** (*dict*) – A dictionary with a key named entities.

sir.indexing.**index_entity**(*entity_name*, *bounds*, *data_queue*)
Retrieve rows for a single entity type identified by entity_name, convert them to a dict with sir.indexing.query_result_to_dict() and put the dicts into queue.

Parameters

- **entity_name** (*str*) –
- **bounds** ((*int*, *int*)) –
- **data_queue** (*Queue.Queue*) –

sir.indexing.**queue_to_solr**(*queue*, *batch_size*, *solr_connection*)
Read dict objects from queue and send them to the Solr server behind solr_connection in batches of batch_size.

Parameters

- **queue** (*multiprocessing.Queue*) –
- **batch_size** (*int*) –
- **solr_connection** (*solr.Solr*) –

sir.indexing.**send_data_to_solr**(*solr_connection*, *data*)
Sends data through solr_connection.

Parameters

- **solr_connection** (*solr.Solr*) –

- **data** (*[dict]*) –

Raises solr.SolrException

sir.indexing.**_multiprocessed_import**(*entity_names*, *live=False*, *entities=None*)
Does the real work to import all entities with entity_name in multiple processes via the multiprocessing module.

When live is True, it means, we are live indexing documents with ids in the entities dict, otherwise it reindexes the entire table for entities in entity_names.

Parameters

- **entity_names** (*[str]*) –
- **live** (*bool*) –
- **entities** (*dict(set(int))*) –

sir.indexing.**_index_entity_process_wrapper**(*args*, *live=False*)
Calls *sir.indexing.index_entity()* with args unpacked.

Parameters **live** (*bool*) –

Return type None or an Exception

sir.indexing.**live_index**(*entities*)

Reindex all documents in‘‘entities‘‘ in multiple processes via the

multiprocessing module.

Parameters **entities** (*dict(set(int))*) –

sir.indexing.**live_index_entity**(*entity_name*, *ids*, *data_queue*)
Retrieve rows for a single entity type identified by entity_name, convert them to a dict with sir.indexing.query_result_to_dict() and put the dicts into queue.

Parameters

- **entity_name** (*str*) –
- **ids** (*[int]*) –
- **data_queue** (*Queue.Queue*) –

5.2 AMQP

sir.amqp.setup.**setup_rabbitmq**(*args*)
Set up the AMQP server.

Parameters **args** – will be ignored

sir.amqp.handler.**callback_wrapper**(*f*)
Common wrapper for a message callback function that provides basic sanity checking for messages and provides exception handling for a function it wraps.

The following wrapper function is returned:

sir.amqp.handler.**wrapper**(*self*, *msg*, *queue*)

Parameters

- **self** (sir.amqp.handler.Handler) – Handler object that is processing a message.

- **msg** (*amqp.basic_message.Message*) – Message itself.
- **queue** (*str*) – Name of the queue that the message has originated from.

Calls `f` with `self` and an instance of *Message*. If an exception gets raised by `f`, it will be caught and the message will be `rejected` and sent to the `search.failed` queue (cf. *Queues*). Then the exception will not be reraised.

If no exception is raised, the message will be `acknowledged`.

`sir.amqp.handler.`**watch**(*args*)
Watch AMQP queues for messages.

Parameters **args** – will be ignored

class `sir.amqp.handler.`**Handler**
Bases: `object`

This class is used to provide callbacks for AMQP messages and access to Solr cores.

ack_message(*msg*, **args*, ***kwargs*)

connect_to_rabbitmq(*reconnect=False*)

delete_callback(*msg*, *queue*)
Callback for processing *delete* messages.

Messages for deletion have the following format:

<table name>, <id or gid>

First value is a table name for an entity that has been deleted. Second is GID or ID of the row in that table. For example:

{"_table": "release", "gid": "90d7709d-feba-47e6-a2d1-8770da3c3d9c"}

This callback function is expected to receive messages only from entity tables all of which have a *gid* column on them except the ones in *_ID_DELETE_TABLE_NAMES* which are deleted via their *id*.

Parameters **parsed_message** (`sir.amqp.message.Message`) – Message parsed by the *callback_wrapper*.

index_callback(*msg*, *queue*)
Callback for processing *index* messages.

Messages for indexing have the following format:

<table name>, keys{<column name>, <value>}

First value is a table name, followed by primary key values for that table. These are then used to lookup values that need to be updated. For example:

{"_table": "artist_credit_name", "position": 0, "artist_credit": 1}

In this handler we are doing a selection with joins which follow a "path" from a table that the trigger was received from to an entity (later "core", https://wiki.apache.org/solr/SolrTerminology). To know which data to retrieve we are using PK(s) of a table that was updated. *update_map* provides us with a view of dependencies between entities (cores) and all the tables. So if data in some table has been updated, we know which entities store this data in the index and need to be refreshed.

Parameters **parsed_message** (`sir.amqp.message.Message`) – Message parsed by the *callback_wrapper*.

process_messages()

reject_message(*msg*, **args*, ***kwargs*)

requeue_message(*msg*, **args*, ***kwargs*)

sir.amqp.handler.**_DEFAULT_MB_RETRIES = 4**
The number of times we'll try to process a message.

sir.amqp.handler.**_RETRY_WAIT_SECS = 30**
The number of seconds between each connection attempt to the AMQP server.

This module contains functions and classes to parse and represent the content of an AMQP message.

exception sir.amqp.message.**InvalidMessageContentException**
Bases: exceptions.ValueError

Exception indicating an error with the content of an AMQP message.

class sir.amqp.message.**MESSAGE_TYPES**
Bases: enum.Enum

delete = 1

index = 2

class sir.amqp.message.**Message**(*message_type*, *table_name*, *columns*, *operation*)
Bases: object

A parsed message from AMQP.

Construct a new message object.

A message contains a set of columns (dict) which can be used to determine which row has been updated. In case of messages from the *index* queue it will be a set of PK columns, and *gid* column for *delete* queue messages.

Parameters

- **message_type** – Type of the message. A member of *MESSAGE_TYPES*.
- **table_name** (*str*) – Name of the table the message is associated with.
- **columns** (*dict*) – Dictionary mapping columns of the table to their values.

classmethod from_amqp_message(*queue_name*, *amqp_message*)
Parses an AMQP message.

Parameters

- **queue_name** (*str*) – Name of the queue where the message originated from.
- **amqp_message** (*amqp.basic_message.Message*) – Message object from the queue.

Return type *sir.amqp.message.Message*

5.3 Querying

sir.querying.**iter_bounds**(*db_session*, *column*, *batch_size*, *importlimit*)
Return a list of (lower bound, upper bound) tuples which contain row ids to iterate through a table in batches of batch_size. If importlimit is greater than zero, return only enough tuples to contain importlimit rows. The second element of the last tuple in the returned list may be None. This happens if the last batch will contain less than batch_size rows.

Parameters

- **db_session** (*sqlalchemy.orm.session.Session*) –

- **column** (*sqlalchemy.Column*) –
- **batch_size** (*int*) –
- **importlimit** (*int*) –

Return type [(int, int)]

sir.querying.**iterate_path_values**(*path*, *obj*)

Return an iterator over all values for *path* on *obj*, an instance of a declarative class by first splitting the path into its elements by splitting it on the dots, resulting in a list of path elements. Then, for each element, a call to `getattr()` is made - the arguments will be the current model (which initially is the **model** assigned to the *SearchEntity*) and the current path element. After doing this, there are two cases:

1. The path element is not the last one in the path. In this case, the `getattr()` call returns one or more objects of another model which will replace the current one.
2. The path element is the last one in the path. In this case, the value returned by the `getattr()` call will be returned and added to the list of values for this field.

To give an example, lets presume the object we're starting with is an instance of *Artist* and the path is "begin_area.name". The first `getattr()` call will be:

```
getattr(obj, "begin_area")
```

which returns an *Area* object, on which the call:

```
getattr(obj, "name")
```

will return the final value:

```
>>> from mbdata.models import Artist, Area
>>> artist = Artist(name="Johan de Meij")
>>> area = Area(name="Netherlands")
>>> artist.begin_area = area
>>> list(iterate_path_values("begin_area.name", artist))
['Netherlands']
```

One-to-many relationships will of course be handled as well:

```
>>> from mbdata.models import Recording, ISRC
>>> recording = Recording(name="Fortuna Imperatrix Mundi: O Fortuna")
>>> recording.isrcs.append(ISRC(isrc="DEF056730100"))
>>> recording.isrcs.append(ISRC(isrc="DEF056730101"))
>>> list(iterate_path_values("isrcs.isrc", recording))
['DEF056730100', 'DEF056730101']
```

5.4 Trigger Generation

sir.trigger_generation.**generate**(*trigger_filename*, *function_filename*, *broker_id*)

Generates SQL queries that create and remove triggers for the MusicBrainz database.

Generation works in the following way:

1. **Determine which tables need to have triggers on them:**
   - Entity tables themselves
   - Tables in every path of entity's fields

2. **Generate triggers (for inserts, updates, and deletions) for each table (model in mbdata):** 2.1. Get a list of PKs 2.2. Write triggers that would send messages into appropriate RabbitMQ queues ("search.index"

   queue for INSERT and UPDATE queries, "search.delete" for DELETE queries):

   <table name>, PKs{<PK row name>, <PK value>}

3. Write generated triggers into SQL scripts to be run on the MusicBrainz database

Since table might have multiple primary keys, we need to explicitly specify their row names and values.

sir.trigger_generation.**generate_func**(*args*)
This is the entry point for this trigger_generation module. This function gets called from `main()`.

sir.trigger_generation.**get_trigger_tables**()
Determines which tables need to have triggers set on them.

Returns a dictionary of table names (key) with a dictionary (value) that provides additional information about a table:

- list of primary keys for each table.
- whether it's an entity table

sir.trigger_generation.**write_footer**(*f*)
Write an SQL "footer" into a file.

Adds a statement to commit a transaction. Should be written at the end of each SQL script that wrote a header (see *write_header* function).

Parameters **f** (*file*) – File to write the footer into.

sir.trigger_generation.**write_header**(*f*)
Write an SQL "header" into a file.

Adds a note about editing, sets command line options, and begins a transaction. Should be written at the beginning of each SQL script.

Parameters **f** (*file*) – File to write the header into.

sir.trigger_generation.**write_triggers**(*trigger_file*, *function_file*, *model*, *is_direct*, *has_gid*, ***generator_args*)

Parameters

- **file trigger_file** (*str*) – File where triggers will be written.
- **file function_file** (*str*) – File where functions will be written.
- **model** – A declarative class.
- **is_direct** (*bool*) – Whether this is an entity table or not.

sir.trigger_generation.**write_triggers_to_file**(*generators*, *trigger_file*, *function_file*, ***generator_args*)
Write SQL for creation of triggers (for deletion, insertion, and updates) and associated functions into files.

Parameters

- **generators** (*list*) – A set of generator classes (based on''TriggerGenerator'') to use for creating SQL statements.
- **trigger_file** (*file*) – File into which commands for creating triggers will be written.
- **function_file** (*file*) – File into which commands for creating trigger functions will be written.

class sir.trigger_generation.sql_generator.**TriggerGenerator**(*table_name*, *pk_columns*, *fk_columns*, *broker_id=1*, ***kwargs*)

Bases: object

Base generator class for triggers and corresponding function that would go into the MusicBrainz database.

Parameters

- **table_name** (*str*) – The table on which to generate the trigger.
- **pk_columns** – List of primary key column names for a table that this trigger is being generated for.
- **broker_id** (*int*) – ID of the AMQP broker row in a database.

op = None
The operation (*INSERT*, *UPDATE*, or *DELETE*)

trigger()
The CREATE TRIGGER statement for this trigger.

Return type str

function()
The CREATE FUNCTION statement for this trigger.

https://www.postgresql.org/docs/9.0/static/plpgsql-structure.html

We use https://github.com/omniti-labs/pg_amqp to publish messages to an AMQP broker.

Return type str

trigger_name
The name of this trigger and its function.

Return type str

class sir.trigger_generation.sql_generator.**InsertTriggerGenerator**(*table_name*, *pk_columns*, *fk_columns*, *broker_id=1*, ***kwargs*)

Bases: *sir.trigger_generation.sql_generator.TriggerGenerator*

A trigger generator for INSERT operations.

Parameters

- **table_name** (*str*) – The table on which to generate the trigger.
- **pk_columns** – List of primary key column names for a table that this trigger is being generated for.
- **broker_id** (*int*) – ID of the AMQP broker row in a database.

class sir.trigger_generation.sql_generator.**UpdateTriggerGenerator**(***gen_args*)

Bases: *sir.trigger_generation.sql_generator.TriggerGenerator*

A trigger generator for UPDATE operations.

trigger()
The CREATE TRIGGER statement for this trigger.

Return type str

class sir.trigger_generation.sql_generator.**DeleteTriggerGenerator**(*table_name*, *pk_columns*, *fk_columns*, *broker_id=1*, ***kwargs*)

Bases: *sir.trigger_generation.sql_generator.TriggerGenerator*

A trigger generator for DELETE operations.

Parameters

- **table_name** (*str*) – The table on which to generate the trigger.
- **pk_columns** – List of primary key column names for a table that this trigger is being generated for.
- **broker_id** (*int*) – ID of the AMQP broker row in a database.

class sir.trigger_generation.sql_generator.**GIDDeleteTriggerGenerator**(**args*, ***kwargs*)

Bases: *sir.trigger_generation.sql_generator.DeleteTriggerGenerator*

This trigger generator produces DELETE statements that selects just *gid* row and ignores primary keys.

It should be used for entity tables themselves (in "direct" triggers) for tables like "artist", "release_group", "recording", and the rest.

class sir.trigger_generation.sql_generator.**ReferencedDeleteTriggerGenerator**(*table_name*, *pk_columns*, *fk_columns*, *broker_id=1*, ***kwargs*)

Bases: *sir.trigger_generation.sql_generator.DeleteTriggerGenerator*

A trigger generator for DELETE operations for tables which are referenced in *SearchEntity* tables. Delete operations in such tables cause the main *SearchEntity* tables to be updated.

Parameters

- **table_name** (*str*) – The table on which to generate the trigger.
- **pk_columns** – List of primary key column names for a table that this trigger is being generated for.
- **broker_id** (*int*) – ID of the AMQP broker row in a database.

5.5 Schema

This package contains core entities that are used in the search index and various tools for working with them.

sir.schema.**SCHEMA = {'annotation': <sir.schema.searchentities.SearchEntity object>, 'area**
Maps core names to *SearchEntity* objects.

sir.schema.**generate_update_map**()
Generates mapping from tables to Solr cores (entities) that depend on these tables and the columns of those tables. In addition provides a path along which data of an entity can be retrieved by performing a set of JOINs and a map of table names to SQLAlchemy ORM models and other useful mappings.

Uses paths to determine the dependency.

:rtype (dict, dict, dict, dict)

class sir.schema.searchentities.**SearchEntity**(*model*, *fields*, *version*, *compatconverter=None*, *extrapaths=None*, *extraquery=None*)

Bases: object

An entity with searchable fields.

Parameters

- **model** – A declarative class.
- **fields** (*list*) – A list of *SearchField* objects.
- **version** (*float*) – The supported schema version of this entity.
- **compatconverter** – A function to convert this object into an XML document compliant with the MMD schema version 2
- **extrapaths** (*[str]*) – A list of paths that don't correspond to any field but are used by the compatibility conversion
- **extraquery** – A function to apply to the object returned by *query()*.

build_entity_query()

Builds a sqlalchemy.orm.query.Query object for this entity (an instance of *sir.schema.searchentities.SearchEntity*) that eagerly loads the values of all search fields.

Return type sqlalchemy.orm.query.Query

query

See *build_entity_query()*.

query_result_to_dict(*obj*)

Converts the result of single query result into a dictionary via the field specification of this entity.

Parameters **obj** – A declarative object.

Return type dict

class sir.schema.searchentities.**SearchField**(*name*, *paths*, *transformfunc=None*, *trigger=True*)

Bases: object

Represents a searchable field.

Each search field has a name and a set of paths. Name is used to reference a field in search queries. Path indicates where the value of that field can be found.

Paths are structured in the following way:

[<one or multiple dot-delimited relationships>.]<column name>

These paths can then be mapped to actual relationships and columns defined in the MusicBrainz schema (see *sir.schema* package and *mbdata* module).

For example, path "areas.area.gid", when bound to the *CustomAnnotation* model would be expanded in the following way:

1. *areas* relationship from the *CustomAnnotation* class
2. *area* relationship from the *AreaAnnotation* class (model)
3. *gid* column from the *Area* class (model)

Parameters

- **name** (*str*) – The name of the field.
- **paths** (*[str]*) – A dot-delimited path (or a list of them) along which the value of this field can be found, beginning at an instance of the model class this field is bound to. See class documentation for more details.
- **transformfunc** (*method*) – An optional function to transform the value before sending it to Solr.

sir.schema.searchentities.**defer_everything_but**(*mapper*, *load*, **columns*)

sir.schema.searchentities.**is_composite_column**(*model*, *colname*)
Checks if a models attribute is a composite column.

Parameters

- **model** – A declarative class.
- **colname** (*str*) – The column name.

Return type bool

sir.schema.searchentities.**merge_paths**(*field_paths*)
Given a list of paths as `field_paths`, return a dict that, for each level of the path, includes a dictionary whose keys are the columns to load and the values are other dictionaries of the described structure.

Parameters **field_paths** (*[[str]]*) –

Return type dict

5.6 Config

sir.config.**CFG = None**
A *SafeExpandingConfigParser* instance holding the configuration data.

exception sir.config.**ConfigError**
Bases: exceptions.Exception

class sir.config.**SafeExpandingConfigParser**(*defaults=None*, *dict_type=<class 'collections.OrderedDict'>*, *allow_no_value=False*)
Bases: ConfigParser.SafeConfigParser, object

sir.config.**read_config**()
Read config files from all possible locations and set *sir.config.CFG* to a *SafeExpandingConfigParser* instance.

5.7 Utilities

exception sir.util.**SIR_EXIT**
Bases: exceptions.Exception

exception sir.util.**VersionMismatchException**(*core*, *expected*, *actual*)
Bases: exceptions.Exception

sir.util.**check_solr_cores_version**(*cores*)
Checks multiple Solr cores for version compatibility

Parameters **cores** (*[str]*) – The names of the cores

Raises ***sir.util.VersionMismatchException*** – If the version in Solr is different from the supported one

sir.util.**create_amqp_connection**()
Creates a connection to an AMQP server.

Return type amqp.connection.Connection

sir.util.**db_session**()
Creates a new sqlalchemy.orm.session.sessionmaker.

Return type sqlalchemy.orm.session.sessionmaker

sir.util.**db_session_ctx**(**args*, ***kwds*)
A context manager yielding a database session.

Parameters **Session** (*sqlalchemy.orm.session.sessionmaker*) –

sir.util.**solr_connection**(*core*)
Creates a solr.Solr connection for the core core.

Parameters **core** (*str*) –

Raises **urllib2.URLError** – if a ping to the cores ping handler doesn't succeed

Return type solr.Solr

sir.util.**solr_version_check**(*core*)
Checks that the version of the Solr core core matches the one in the schema.

Parameters **core** (*str*) –

Raises

- **urllib2.URLError** – If the Solr core can't be reached
- ***sir.util.VersionMismatchException*** – If the version in Solr is different from the supported one

5.8 Examples

class mbdata.models.**Artist**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

aliases

area

area_id

begin_area

begin_area_id

begin_date

begin_date_day

begin_date_month

begin_date_year

comment

edits_pending

end_area

end_area_id

end_date

end_date_day

end_date_month

end_date_year

ended

gender

gender_id

gid

id

ipis

isnis

last_updated

meta

name

sort_name

type

type_id

class mbdata.models.**Area**(***kwargs*)

Bases: sqlalchemy.ext.declarative.api.Base

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

begin_date

begin_date_day

begin_date_month

begin_date_year

comment

edits_pending

end_date

```
end_date_day
end_date_month
end_date_year
ended
gid
id
iso_3166_1_codes
iso_3166_2_codes
iso_3166_3_codes
last_updated
name
type
type_id
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W