
tento Documentation

Release 0.0.1

Kang Hyojun

September 22, 2014

1	tento — tento	3
1.1	tento.db — tento DB	3
1.2	tento.user —	3
1.3	tento.music —	4
1.4	tento.web — tento web	14
1.5	tento.crawler — crawl song’s lyrics from nmusic.	16
2	Indices and tables	19
	Python Module Index	21

Contents:

1.1 tento.db — tento DB

```
tento.db.ensure_shutdown_session(app)
    tento.web.app.app , tento.db.session .
```

```
tento.db.get_engine(app=None)
    DB .
```

Parameters `app` (*flask.Flask*) – flask.Flask

Returns sqlalchemy

Return type sqlalchemy.engine.Engine

```
tento.db.get_session(engine=None)
    sqlalchemy .
```

Parameters `engine` (*sqlalchemy.engine.Engine*) – sqlalchemy

Returns DB

Return type sqlalchemy.orm.session.Session

```
tento.db.get_alembic_config(engine)
    alembic .
```

Parameters

- `engine` – db sqlalchemy.engine.Engine
- `app` (*flask.Flask*) – flask.Flask

Returns alembic alembic.config.Config

Return type alembic.config.Config

1.2 tento.user — .

```
class tento.user.User(email, password, created_at=None)
```

```
confirm_pass
```

Parameters `password` (*str*) –

Returns

Return type bool

created_at

email

id

name

```
.  
>>> User(email='admire9@gmail.com', password='foobar')  
>>> _.name  
'admire9'
```

Returns

Return type str

password

1.3 `tento.music` — .

```
class tento.music.Album(**kwargs)                                artist_id  
  
    created_at  
    id  
        Album  
    musics  
        Music  
    name  
    year  
class tento.music.Artist(**kwargs)                                albums  
    Album  
    created_at  
    id  
        Artist  
    name  
class tento.music.Genre(**kwargs)                                created_at  
  
    id  
        Genre  
    name  
class tento.music.Music(**kwargs)                                album_id  
    (,)  
    created_at  
    disc_number  
    ()
```



```

genre
genre_id
    ()
id
    Music
name
track_number
    ()
class tento.music.Position (**kwargs)
    x,y,music_id
created_at
id
    Position
music
    tento.Music
music_id
    tento.Music.music_id
x
    x ()
y
    y (BPM )
tento.music.relationship (argument, secondary=None, primaryjoin=None, secondaryjoin=None,
    foreign_keys=None, uselist=None, order_by=False, back-
    ref=None, back_populates=None, post_update=False, cas-
    cade=False, extension=None, viewonly=False, lazy=True, collec-
    tion_class=None, passive_deletes=False, passive_updates=True,
    remote_side=None, enable_typechecks=True, join_depth=None,
    comparator_factory=None, single_parent=False, innerjoin=False,
    distinct_target_key=None, doc=None, active_history=False, cas-
    cade_backrefs=True, load_on_pending=False, strategy_class=None,
    _local_remote_pairs=None, query_class=None, info=None)

```

Provide a relationship between two mapped classes.

This corresponds to a parent-child or associative table relationship. The constructed class is an instance of `RelationshipProperty`.

A typical `relationship()`, used in a classical mapping:

```

mapper(Parent, properties={
    'children': relationship(Child)
})

```

Some arguments accepted by `relationship()` optionally accept a callable function, which when called produces the desired value. The callable is invoked by the parent `Mapper` at “mapper initialization” time, which happens only when mappers are first used, and is assumed to be after all mappings have been constructed. This can be used to resolve order-of-declaration and other dependency issues, such as if `Child` is declared below `Parent` in the same file:

```

mapper(Parent, properties={
    "children":relationship(lambda: Child,

```

```
        order_by=lambda: Child.id)
    })
```

When using the *declarative_toplevel* extension, the Declarative initializer allows string arguments to be passed to `relationship()`. These string arguments are converted into callables that evaluate the string as Python code, using the Declarative class-registry as a namespace. This allows the lookup of related classes to be automatic via their string name, and removes the need to import related classes at all into the local module space:

```
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    children = relationship("Child", order_by="Child.id")
```

See also:

relationship_config_toplevel - Full introductory and reference documentation for `relationship()`.

orm_tutorial_relationship - ORM tutorial introduction.

Parameters

- **argument** – a mapped class, or actual `Mapper` instance, representing the target of the relationship.

:paramref:~.relationship.argument may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

See also:

declarative_configuring_relationships - further detail on relationship configuration when using Declarative.

- **secondary** – for a many-to-many relationship, specifies the intermediary table, and is typically an instance of `Table`. In less common circumstances, the argument may also be specified as an `Alias` construct, or even a `Join` construct.

:paramref:~.relationship.secondary may also be passed as a callable function which is evaluated at mapper initialization time. When using Declarative, it may also be a string argument noting the name of a `Table` that is present in the `MetaData` collection associated with the parent-mapped `Table`.

The **:paramref:~.relationship.secondary** keyword argument is typically applied in the case where the intermediary `Table` is not otherwise expressed in any direct class mapping. If the “secondary” table is also explicitly mapped elsewhere (e.g. as in *association_pattern*), one should consider applying the **:paramref:~.relationship.viewonly** flag so that this `relationship()` is not used for persistence operations which may conflict with those of the association object pattern.

See also:

relationships_many_to_many - Reference example of “many to many”.

orm_tutorial_many_to_many - ORM tutorial introduction to many-to-many relationships.

self_referential_many_to_many - Specifics on using many-to-many in a self-referential case.

declarative_many_to_many - Additional options when using Declarative.

association_pattern - an alternative to **:paramref:‘~.relationship.secondary’** when composing association table relationships, allowing additional attributes to be specified on the association table.

composite_secondary_join - a lesser-used pattern which in some cases can enable complex `relationship()` SQL conditions to be used.

New in version 0.9.2: **:paramref:‘~.relationship.secondary’** works more effectively when referring to a `Join` instance.

- **active_history=False** – When `True`, indicates that the “previous” value for a many-to-one reference should be loaded when replaced, if not already loaded. Normally, history tracking logic for simple many-to-ones only needs to be aware of the “new” value in order to perform a flush. This flag is available for applications that make use of `attributes.get_history()` which also need to know the “previous” value of the attribute.
- **backref** – indicates the string name of a property to be placed on the related mapper’s class that will handle this relationship in the other direction. The other property will be created automatically when the mappers are configured. Can also be passed as a `backref()` object to control the configuration of the new relationship.

See also:

relationships_backref - Introductory documentation and examples.

:paramref:‘~.relationship.back_populates’ - alternative form of backref specification.

`backref()` - allows control over `relationship()` configuration when using **:paramref:‘~.relationship.backref’**.

- **back_populates** – Takes a string name and has the same meaning as **:paramref:‘~.relationship.backref’**, except the complementing property is **not** created automatically, and instead must be configured explicitly on the other mapper. The complementing property should also indicate **:paramref:‘~.relationship.back_populates’** to this relationship to ensure proper functioning.

See also:

relationships_backref - Introductory documentation and examples.

:paramref:‘~.relationship.backref’ - alternative form of backref specification.

- **cascade** – a comma-separated list of cascade rules which determines how Session operations should be “cascaded” from parent to child. This defaults to `False`, which means the default cascade should be used - this default cascade is `"save-update, merge"`.

The available cascades are `save-update`, `merge`, `expunge`, `delete`, `delete-orphan`, and `refresh-expire`. An additional option, `all` indicates shorthand for `"save-update, merge, refresh-expire, expunge, delete"`, and is often used as in `"all, delete-orphan"` to indicate that related objects should follow along with the parent object in all cases, and be deleted when de-associated.

See also:

unitofwork_cascades - Full detail on each of the available cascade options.

tutorial_delete_cascade - Tutorial example describing a delete cascade.

- **cascade_backrefs=True** – a boolean value indicating if the `save-update` cascade should operate along an assignment event intercepted by a backref. When set to `False`, the attribute managed by this relationship will not cascade an incoming transient object into the session of a persistent parent, if the event is received via backref.

See also:

backref_cascade - Full discussion and examples on how the **:paramref:‘~.relationship.cascade_backrefs’** option is used.

- **collection_class** – a class or callable that returns a new list-holding object. will be used in place of a plain list for storing elements.

See also:

custom_collections - Introductory documentation and examples.

- **comparator_factory** – a class which extends `RelationshipProperty.Comparator` which provides custom SQL clause generation for comparison operations.

See also:

PropComparator - some detail on redefining comparators at this level.

custom_comparators - Brief intro to this feature.

- **distinct_target_key=None** – Indicate if a “subquery” eager load should apply the `DISTINCT` keyword to the innermost `SELECT` statement. When left as `None`, the `DISTINCT` keyword will be applied in those cases when the target columns do not comprise the full primary key of the target table. When set to `True`, the `DISTINCT` keyword is applied to the innermost `SELECT` unconditionally.

It may be desirable to set this flag to `False` when the `DISTINCT` is reducing performance of the innermost subquery beyond that of what duplicate innermost rows may be causing.

New in version 0.8.3: - **:paramref:‘~.relationship.distinct_target_key’** allows the subquery eager loader to apply a `DISTINCT` modifier to the innermost `SELECT`.

Changed in version 0.9.0: - **:paramref:‘~.relationship.distinct_target_key’** now defaults to `None`, so that the feature enables itself automatically for those cases where the innermost query targets a non-unique key.

See also:

loading_toplevel - includes an introduction to subquery eager loading.

- **doc** – docstring which will be applied to the resulting descriptor.
- **extension** – an `AttributeExtension` instance, or list of extensions, which will be prepended to the list of attribute listeners for the resulting descriptor placed on the class.

Deprecated since version 0.7: Please see `AttributeEvents`.

- **foreign_keys** – a list of columns which are to be used as “foreign key” columns, or columns which refer to the value in a remote column, within the context of this `relationship()` object’s **:paramref:‘~.relationship.primaryjoin’** condition. That is, if the **:paramref:‘~.relationship.primaryjoin’** condition of this `relationship()` is `a.id == b.a_id`, and the values in `b.a_id` are required to be present in `a.id`, then the “foreign key” column of this `relationship()` is `b.a_id`.

In normal cases, the **:paramref:‘~.relationship.foreign_keys’** parameter is **not required**. `relationship()` will automatically determine which columns in the **:paramref:‘~.relationship.primaryjoin’** condition are to be considered “foreign key” columns based on those `Column` objects that specify `ForeignKey`, or are otherwise

listed as referencing columns in a `ForeignKeyConstraint` construct. **:param-ref:~.relationship.foreign_keys** is only needed when:

1. There is more than one way to construct a join from the local table to the remote table, as there are multiple foreign key references present. Setting `foreign_keys` will limit the `relationship()` to consider just those columns specified here as “foreign”.

Changed in version 0.8: A multiple-foreign key join ambiguity can be resolved by setting the **:param-ref:~.relationship.foreign_keys** parameter alone, without the need to explicitly set **:param-ref:~.relationship.primaryjoin** as well.

2. The Table being mapped does not actually have `ForeignKey` or `ForeignKeyConstraint` constructs present, often because the table was reflected from a database that does not support foreign key reflection (MySQL MyISAM).
3. The **:param-ref:~.relationship.primaryjoin** argument is used to construct a non-standard join condition, which makes use of columns or expressions that do not normally refer to their “parent” column, such as a join condition expressed by a complex comparison using a SQL function.

The `relationship()` construct will raise informative error messages that suggest the use of the **:param-ref:~.relationship.foreign_keys** parameter when presented with an ambiguous condition. In typical cases, if `relationship()` doesn’t raise any exceptions, the **:param-ref:~.relationship.foreign_keys** parameter is usually not needed.

:param-ref:~.relationship.foreign_keys may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

See also:

relationship_foreign_keys

relationship_custom_foreign

`foreign()` - allows direct annotation of the “foreign” columns within a **:param-ref:~.relationship.primaryjoin** condition.

New in version 0.8: The `foreign()` annotation can also be applied directly to the **:param-ref:~.relationship.primaryjoin** expression, which is an alternate, more specific system of describing which columns in a particular **:param-ref:~.relationship.primaryjoin** should be considered “foreign”.

- **info** – Optional data dictionary which will be populated into the `MapperProperty.info` attribute of this object.

New in version 0.8.

- **innerjoin=False** – when `True`, joined eager loads will use an inner join to join against related tables instead of an outer join. The purpose of this option is generally one of performance, as inner joins generally perform better than outer joins.

This flag can be set to `True` when the relationship references an object via many-to-one using local foreign keys that are not nullable, or when the reference is one-to-one or a collection that is guaranteed to have one or at least one entry.

If the joined-eager load is chained onto an existing `LEFT OUTER JOIN`, `innerjoin=True` will be bypassed and the join will continue to chain as `LEFT OUTER JOIN` so that the results don’t change. As an alternative, specify the value “nested”. This will instead nest the join on the right side, e.g. using the form “a `LEFT OUTER JOIN` (b `JOIN` c)”.

New in version 0.9.4: Added `innerjoin="nested"` option to support nesting of eager “inner” joins.

See also:

what_kind_of_loading - Discussion of some details of various loader options.

:paramref:‘`joinedload.innerjoin`‘ - loader option version

- **join_depth** – when non-None, an integer value indicating how many levels deep “eager” loaders should join on a self-referring or cyclical relationship. The number counts how many times the same Mapper shall be present in the loading condition along a particular join branch. When left at its default of None, eager loaders will stop chaining when they encounter a the same target mapper which is already higher up in the chain. This option applies both to joined- and subquery- eager loaders.

See also:

self_referential_eager_loading - Introductory documentation and examples.

- **lazy=‘select’** – specifies how the related items should be loaded. Default value is `select`. Values include:
 - `select` - items should be loaded lazily when the property is first accessed, using a separate SELECT statement, or identity map fetch for simple many-to-one references.
 - `immediate` - items should be loaded as the parents are loaded, using a separate SELECT statement, or identity map fetch for simple many-to-one references.
 - `joined` - items should be loaded “eagerly” in the same query as that of the parent, using a JOIN or LEFT OUTER JOIN. Whether the join is “outer” or not is determined by the **:paramref:‘`~.relationship.innerjoin`‘** parameter.
 - `subquery` - items should be loaded “eagerly” as the parents are loaded, using one additional SQL statement, which issues a JOIN to a subquery of the original statement, for each collection requested.
 - `noload` - no loading should occur at any time. This is to support “write-only” attributes, or attributes which are populated in some manner specific to the application.
 - `dynamic` - the attribute will return a pre-configured `Query` object for all read operations, onto which further filtering operations can be applied before iterating the results. See the section *dynamic_relationship* for more details.
 - `True` - a synonym for ‘select’
 - `False` - a synonym for ‘joined’
 - `None` - a synonym for ‘noload’

See also:

/orm/loading - Full documentation on relationship loader configuration.

dynamic_relationship - detail on the `dynamic` option.

- **load_on_pending=False** – Indicates loading behavior for transient or pending parent objects.

When set to `True`, causes the lazy-loader to issue a query for a parent object that is not persistent, meaning it has never been flushed. This may take effect for a pending object when autoflush is disabled, or for a transient object that has been “attached” to a `Session` but is not part of its pending collection.

The **:paramref:‘~.relationship.load_on_pending‘** flag does not improve behavior when the ORM is used normally - object references should be constructed at the object level, not at the foreign key level, so that they are present in an ordinary way before a flush proceeds. This flag is not intended for general use.

See also:

`Session.enable_relationship_loading()` - this method establishes “load on pending” behavior for the whole object, and also allows loading on objects that remain transient or detached.

- **order_by** – indicates the ordering that should be applied when loading these items. **:paramref:‘~.relationship.order_by‘** is expected to refer to one of the `Column` objects to which the target class is mapped, or the attribute itself bound to the target class which refers to the column.

:paramref:‘~.relationship.order_by‘ may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

- **passive_deletes=False** – Indicates loading behavior during delete operations.

A value of `True` indicates that unloaded child items should not be loaded during a delete operation on the parent. Normally, when a parent item is deleted, all child items are loaded so that they can either be marked as deleted, or have their foreign key to the parent set to `NULL`. Marking this flag as `True` usually implies an `ON DELETE <CASCADE|SET NULL>` rule is in place which will handle updating/deleting child rows on the database side.

Additionally, setting the flag to the string value ‘all’ will disable the “nulling out” of the child foreign keys, when there is no delete or delete-orphan cascade enabled. This is typically used when a triggering or error raise scenario is in place on the database side. Note that the foreign key attributes on in-session child objects will not be changed after a flush occurs so this is a very special use-case setting.

See also:

passive_deletes - Introductory documentation and examples.

- **passive_updates=True** – Indicates loading and `INSERT/UPDATE/DELETE` behavior when the source of a foreign key value changes (i.e. an “on update” cascade), which are typically the primary key columns of the source row.

When `True`, it is assumed that `ON UPDATE CASCADE` is configured on the foreign key in the database, and that the database will handle propagation of an `UPDATE` from a source column to dependent rows. Note that with databases which enforce referential integrity (i.e. PostgreSQL, MySQL with InnoDB tables), `ON UPDATE CASCADE` is required for this operation. The `relationship()` will update the value of the attribute on related items which are locally present in the session during a flush.

When `False`, it is assumed that the database does not enforce referential integrity and will not be issuing its own `CASCADE` operation for an update. The `relationship()` will issue the appropriate `UPDATE` statements to the database in response to the change of a referenced key, and items locally present in the session during a flush will also be refreshed.

This flag should probably be set to `False` if primary key changes are expected and the database in use doesn’t support `CASCADE` (i.e. SQLite, MySQL MyISAM tables).

See also:

passive_updates - Introductory documentation and examples.

:paramref:‘.mapper.passive_updates‘ - a similar flag which takes effect for joined-table inheritance mappings.

- **post_update** – this indicates that the relationship should be handled by a second UPDATE statement after an INSERT or before a DELETE. Currently, it also will issue an UPDATE after the instance was UPDATED as well, although this technically should be improved. This flag is used to handle saving bi-directional dependencies between two individual rows (i.e. each row references the other), where it would otherwise be impossible to INSERT or DELETE both rows fully since one row exists before the other. Use this flag when a particular mapping arrangement will incur two rows that are dependent on each other, such as a table that has a one-to-many relationship to a set of child rows, and also has a column that references a single child row within that list (i.e. both tables contain a foreign key to each other). If a flush operation returns an error that a “cyclical dependency” was detected, this is a cue that you might want to use **:paramref:‘~.relationship.post_update‘** to “break” the cycle.

See also:

post_update - Introductory documentation and examples.

- **primaryjoin** – a SQL expression that will be used as the primary join of this child object against the parent object, or in a many-to-many relationship the join of the primary object to the association table. By default, this value is computed based on the foreign key relationships of the parent and child tables (or association table).

:paramref:‘~.relationship.primaryjoin‘ may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

See also:

relationship_primaryjoin

- **remote_side** – used for self-referential relationships, indicates the column or list of columns that form the “remote side” of the relationship.

:paramref:‘.relationship.remote_side‘ may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

Changed in version 0.8: The `remote()` annotation can also be applied directly to the `primaryjoin` expression, which is an alternate, more specific system of describing which columns in a particular `primaryjoin` should be considered “remote”.

See also:

self_referential - in-depth explanation of how **:paramref:‘~.relationship.remote_side‘** is used to configure self-referential relationships.

`remote()` - an annotation function that accomplishes the same purpose as **:paramref:‘~.relationship.remote_side‘**, typically when a custom **:paramref:‘~.relationship.primaryjoin‘** condition is used.

- **query_class** – a `Query` subclass that will be used as the base of the “appender query” returned by a “dynamic” relationship, that is, a relationship that specifies `lazy="dynamic"` or was otherwise constructed using the `orm.dynamic_loader()` function.

See also:

dynamic_relationship - Introduction to “dynamic” relationship loaders.

- **secondaryjoin** – a SQL expression that will be used as the join of an association table to the child object. By default, this value is computed based on the foreign key relationships of the association and child tables.

:paramref:‘~.relationship.secondaryjoin‘ may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

See also:

relationship_primaryjoin

- **single_parent** – when True, installs a validator which will prevent objects from being associated with more than one parent at a time. This is used for many-to-one or many-to-many relationships that should be treated either as one-to-one or one-to-many. Its usage is optional, except for `relationship()` constructs which are many-to-one or many-to-many and also specify the `delete-orphan` cascade option. The `relationship()` construct itself will raise an error instructing when this option is required.

See also:

unitofwork_cascades - includes detail on when the **:paramref:‘~.relationship.single_parent‘** flag may be appropriate.

- **uselist** – a boolean that indicates if this property should be loaded as a list or a scalar. In most cases, this value is determined automatically by `relationship()` at mapper configuration time, based on the type and direction of the relationship - one to many forms a list, many to one forms a scalar, many to many is a list. If a scalar is desired where normally a list would be present, such as a bi-directional one-to-one relationship, set **:paramref:‘~.relationship.uselist‘** to False.

The **:paramref:‘~.relationship.uselist‘** flag is also available on an existing `relationship()` construct as a read-only attribute, which can be used to determine if this `relationship()` deals with collections or scalar attributes:

```
>>> User.addresses.property.uselist
True
```

See also:

relationships_one_to_one - Introduction to the “one to one” relationship pattern, which is typically when the **:paramref:‘~.relationship.uselist‘** flag is needed.

- **viewonly=False** – when set to True, the relationship is used only for loading objects, and not for any persistence operation. A `relationship()` which specifies **:paramref:‘~.relationship.viewonly‘** can work with a wider range of SQL operations within the **:paramref:‘~.relationship.primaryjoin‘** condition, including operations that feature the use of a variety of comparison operators as well as SQL functions such as `cast()`. The **:paramref:‘~.relationship.viewonly‘** flag is also of general use when defining any kind of `relationship()` that doesn’t represent the full set of related objects, to prevent modifications of the collection from resulting in persistence operations.

1.4 tento.web — tento web

1.4.1 tento.web.auth — tento

1.4.2 tento.web.user — tento user API

1.4.3 tento.web.login — tento login API

1.4.4 tento.web.music — tento API

```
tento.web.music.create()  
    tento.music.Artist,tento.music.Album,tento.music.Music .
```

```
POST /musics/ HTTP/1.1  
Content-Type: application/json  
Accept: application/json  
Host: tento.com
```

```
{  
  "music_name": "",  
  "music_track_number": 1,  
  "music_disc_number": 1,  
  "artist_name": "leeSA",  
  "album_name": "",  
  "album_release_year": 2010,  
  "genre": " > , > "  
}
```

```
HTTP/1.1 201 created  
Content-Type: application/json
```

Returns tento.music.Artist,tento.music.Album,tento.music.Music json .

Statuscode 201 .

Statuscode 400 .

Statuscode 500

```
tento.web.music.create_chunks()  
    tento.music.Artist,tento.music.Album,tento.music.Music .
```

```
POST /musics/chunk/ HTTP/1.1  
Content-Type: application/json  
Accept: application/json  
Host: tento.com
```

```
{  
  'musics': [  
    {  
      "music_name": "",  
      "music_track_number": 1,  
      "music_disc_number": 1,  
      "artist_name": "leeSA",  
      "album_name": "",  
      "album_release_year": 2010,  
      "genre": " > , > "  
    }  
  ]  
}
```

```

        },
        ...
    ]
}

HTTP/1.1 201 created
Content-Type: application/json

{
  'musics': [
    {'id': 1, 'name': ''},
    ...
  ]
}

```

Returns `tento.music.Artist`, `tento.music.Album`, `tento.music.Music` json .

Statuscode 201 .

Statuscode 400 .

Statuscode 500 .

```

tento.web.music.find_all_positions()
:class:'tento.music.Position' .

GET /musics/positions/?music_ids=1,2,3,4
Accept: application/json
Host: tento.com

HTTP/1.1 201 created
Content-Type: application/json
{
  "positions": [
    {
      "x": 0,
      "y": 0,
      "music_id": 1
    },
    ...
  ]
}

```

Query string music_ids `tento.music.Music.id` , .

Returns `:py:class:'tento.music.Position'` json

Statuscode 200 .

Statuscode 400 , music_ids

Statuscode 500 .

```

tento.web.music.find_position(id_)

Parameters id - :class:'tento.music.Position' .

```

```

GET /musics/:id/position/
Accept: application/json
Host: tento.com

```

```
HTTP/1.1 201 created
Content-Type: application/json
```

```
{
  "x": 0,
  "y": 0,
  "music_id": :id
}
```

Parameters `id` – `tento.music.Music` `tento.music.Music.id`

Returns :py:class:‘tento.music.Position‘ json

Statuscode 200 .

Statuscode 404 .

param `id_` `tento.music.Position`

Statuscode 500 .

```
tento.web.music.position(id_)
: class: ‘tento.music.Position‘ .
```

```
POST /musics/:id/position/
Content-Type: application/json
Accept: application/json
Host: tento.com
```

```
{
  "x": 10,
  "y": 9,
  "music_id": 1
}
```

..sourcecode:: http HTTP/1.1 201 created Content-Type: application/json

Parameters `id` – `tento.music.Music` `tento.music.Music.id`

Returns :py:class:‘tento.music.Position‘ json

Statuscode 201 .

Statuscode 400 .

Statuscode 500 .

1.5 `tento.crawler` — crawl song’s lyrics from nmusic.

```
class tento.crawler.NmusicCrawler
    Crawl nmusic’s information.
```

```
    lyrics(nmusic_id)
        Get lyrics.
```

Parameters `nmusic_id` (*int*) – naver music id.

Returns lyrics of :param nmusic_id: .

Return type str

lyrics_url ()

lyrics url

read_doc (*url*)

Read html document from given :param url: .

Parameters **url** – url want to be readed.

Returns html document.

search (*word*)

Search naver music track id.

Parameters **word** (*str*) – song's name or artist name want to search.

Returns list of result.

Return type list

url ()

search url

Indices and tables

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

t

tento, 1
tento.crawler, 16
tento.db, 3
tento.music, 4
tento.user, 3
tento.web, 13
tento.web.app, 14
tento.web.auth, 14
tento.web.login, 14
tento.web.music, 14
tento.web.user, 14