django-crosswalk Documentation

Release 0.0.4

Jon McClure

Apr 13, 2019

Contents:


POLITICO

CHAPTER 1

Why this?

This package is made by journalists. We work with a lot of unstandardized datasets, and reconciling entities between them is a daily challenge.

Some methods of deduplication within large datasets can be very complex, for which there are tools like dedupe.io. But much more often, our record problems are less complex and can be addressed with a few simple record linkage techniques.

Django-crosswalk is an entity service that provides a few basic, but highly extensible tools to resolve entity IDs and create linked records of known aliases.

We use it to standardize IDs across datasets, to create a master crosswalk of all known aliases and as a reference library for entity metadata, augmenting libraries like us.

1.1 What's in it?

The project consists of two packages.

Django-crosswalk is a pluggable Django application that creates tables to house your entities. It also manages tokens to authenticate users who can query and modify records through a robust API.

Django-crosswalk-client is a tiny library to make interacting with the API easy to do inline in your code.

1.2 What can you do with it?

Create complex databases of entities and their aliases in django-crosswalk, then use fuzzy queries to help resolve entities' identities when working with unstandardized data.

The client library lets you easily integrate django-crosswalk's record linkage techniques in your code. Use it like a middleware between your raw data and your destination database. Scrape some entities from the web, then query your crosswalk tables and normalize your IDs before saving to your db.

Django-crosswalk will give your entities a canonical UUID you can use across databases or will persist one you've already created and return it whenever you query an alias.

Improve the precision of your queries by using the things you know about an entity. Create entities within a useful "domain" category so you don't confuse unrelated entities. Use whatever arbitrary attributes of an entity to create a custom blocking index. For example, you might use the state location and industry code to reduce the number of possible matches to a query based on a company's name.

Create records of known aliases and use that data to create a training set for higher order deduplication algorithms.

CHAPTER 2

Quickstart

2.1 Prerequisites

- Python 3.6
- Django 2.2
- PostgreSQL 9.4

2.2 Installation

1. Install django-crosswalk.

```
$ pip install django-crosswalk
```

2. Add the app and DRF to `INSTALLED_APPS` in your project settings.

```
# settings.py

INSTALLED_APPS = [
  # ...
  'rest_framework',
  'crosswalk',
]
```

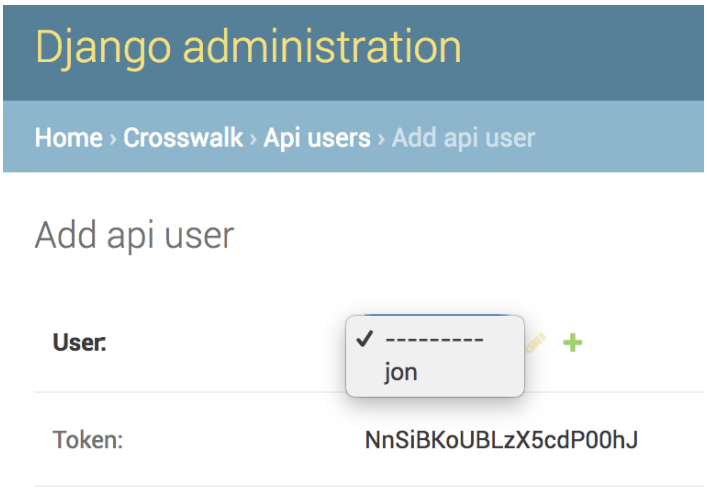
3. Include the app in your project's URLconf.

```
# urls.py
urlpatterns = [
    # ...
    path('crosswalk', include('crosswalk.urls')),
]
```

4. Migrate databases.

```
$ python manage.py migrate crosswalk
```

5. Open Django's admin and create your first API users on the crosswalk `ApiUser` model.



6. Install the client.

```
$ pip install django-crosswalk-client
```

7. Read through the rest of these docs to see how to interact with your database using the client.

CHAPTER 3

Concepts

3.1 Domains

A domain is an arbitrary category your entities belong to. For example, you may have a domain of `states`, which includes entities that are U.S. states.

Every entity you create **must belong** to one domain.

Domains may be nested with a parent-child hierarchy. For example, `states` may be a parent to `counties`.

A domain is the highest level blocking index in django-crosswalk. These indexes help greatly increase the precision of queries. For example, restricting your query to our `states` domain makes sure you won't match Texas County, Missouri when you mean to match Texas state.

3.2 Entities

Entities can be whatever you need them to be: people, places, organizations or even ideas. The only strict requirements in django-crosswalk are that an entity has one or more identifying string attributes and that it belong to a domain.

On creation, each entity in django-crosswalk is given a `uuid` attribute that you can use to match with records in other databases. You can also create entities with your own uuids.

3.2.1 Entity relationships

Entities can be related to each other in a couple of ways. An entity may be an **alias** for another entity or may be **superseded** by another entity. We make the distinction between the two based on whether the entity referenced is in the same domain.

By convention, an entity that is related to another entity in the same domain is an **alias**. For example, `George Bush, Jr.` could be an alias for `George W. Bush`, both within the domain of `politicians`.

On the model, `George Bush, Jr.` is foreign keyed to `George W. Bush`, which indicates that `George W. Bush` is the canonical representation of the entity within the `politicians` domain.

Let's say we also had a domain for `presidents`. We could say `George W. Bush` the politician is superseded by `George W. Bush` the president and represent that relationship with a foreign key. This is often exactly how we model entities that belong to multiple domains.

These are conventions and are not enforced at the database level. So it's up to you to use them in a way that suits your data. Please note, however, deleting an entity will also delete all of its aliases but not delete entites it supersedes.

On the `Entity` model, every object has a foreign key for both aliasing and superseding, which you can use to chain canonical references. Django-crosswalk will traverse that chain to the highest canonical alias entity when returning query results. It will not traverse superseding relationships.

3.3 Attributes

Entities are defined by their attributes. Django-crosswalk allows you to add any arbitrary attribute to an entity. For example, a state may have attributes like `postal_code`, `ap_abbreviation`, `fips` and `region` as well as `name`.

Attributes are stored in a single JSONField on the `Entity` model.

After you've created them, you can use attributes to create additional blocking indexes – i.e., a block of entities filtered by a set of attributes – to make your queries more precise. For example, you can query a state within a specific `region`.

Warning: An entity's attributes must be unique together within a domain.

Nested attributes are not allowed. Django-crosswalk is focused solely on entity resolution and record linkage, not in being a complete resource of all information about your entities. Complex data should be kept in other databases, probably linked by the UUID.

In general, we recommend snake-casing attribute names, though this is not enforced.

There are some reserved attribute names. Django-crosswalk will throw a validation error if you try to use them:

- `alias_for`
- `aliased`
- `attributes`
- `created`
- `domain`
- `entity`
- `match_score`
- `superseded_by`
- `uuid`

3.4 Scorers

Scorers are functions that compare strings and are used when querying entities. Django-crosswalk comes with four scorers, all using `process.exctractOne` from the fuzzywuzzy package:

- `fuzzywuzzy.default_process`
  - Uses fuzzywuzzy's simple ratio scorer.
- `fuzzywuzzy.partial_ratio_process`
  - Uses fuzzywuzzy's partial ratio scorer.
- `fuzzywuzzy.token_sort_ratio_process`
  - Uses fuzzywuzzy's token sort ratio scorer.
- `fuzzywuzzy.token_set_ratio_process`
  - Uses fuzzywuzzy's token set ratio scorer.

In django-crosswalk, all scorer functions have the same signature. They must accept a query string (`query_value`) and a list of strings to compare (`block_values`). They must return a tuple that contains a matched string from `block_values` and a normalized match score.

```
def your_custom_scorer(query_value, block_values):
    match, score = somefunc(query_value, block_values)
    return (match, score)
```

Feel free to submit new scorers to this project!

CHAPTER 4

Using the client

The django-crosswalk client lets you interact with your crosswalk database much like you would any standard library, albeit through an API.

Generally, we **do not** recommend interacting with django-crosswalk's API directly. Instead, use the methods built into the client, which have more verbose validation and error messages and are well tested.

4.1 Install

The client is maintained as a separate package, which you can install via pip.

```
$ pip install django-crosswalk-client
```

4.2 Client configuration

4.2.1 Creating a client instance

Create a client instance by passing your API token and the URL to the root of your hosted django-crosswalk API.

```
from crosswalk_client import Client

# Your API token, created in Django admin
token = "<TOKEN>"

# Address of django-crosswalk's API
service = "https://mysite.com/crosswalk/api/"

client = Client(token, service)
```

You can also instantiate a client with defaults.

```
client = Client(
    token,
    service,
    domain=None, # default
    scorer="fuzzywuzzy.default_process", # default
    threshold=80, # default
)
```

4.2.2 Set the default domain

In order to query, create or edit entities, you must specify a domain. You can set a default anytime:

```
# Using domain instance
client.set_domain(states)

# ... or a domain's slug
client.set_domain("states")
```

4.2.3 Set the default scorer

The string module path to a scorer function in `crosswalk.scorers`.

```
client.set_scorer("fuzzywuzzy.token_sort_ratio_process")
```

4.2.4 Set the default threshold

The default threshold is used when creating entities based on a match score. For all scorers, the match score should be an integer between 0 - 100.

```
client.set_threshold(90)
```

4.3 Client domain methods

4.3.1 Create a domain

```
states = client.create_domain("U.S. states")

states.name == "U.S. states"
states.slug == "u-s-states" # Name of domain is always slugified!

# Create with a parent domain instance
client.create_domain("counties", parent=states)

# ... or a parent domain's slug
client.create_domain("cities", parent="u-s-states")
```

4.3.2 Get a domain

```
# Use a domain's slug
states = client.get_domain("u-s-states")

states.name == "U.S. states"
```

4.3.3 Get all domains

```
states = client.get_domains()[0]

states.slug == "u-s-states"

# Filter domains by a parent domain instance
client.get_domains(parent=states)

# ... or parent domain's slug
client.get_domains(parent="u-s-states")
```

4.3.4 Update a domain

```
# Using the domain's slug
states = client.update_domain("u-s-states", {"parent": "countries"})

# ... or the domain instance
client.update_domain(states, {"parent": "country"})
```

4.3.5 Delete a domain

```
# Using domain's slug
client.delete_domain('u-s-states')

# ... or the domain instance
client.delete_domain(states)
```

4.4 Client entity methods

4.4.1 Create entities

Create a single entity as a shallow dictionary.

```
entities = client.create({"name": "Kansas", "postal_code": "KS"}, domain=states)
```

Create a list of shallow dictionaries for each entity you'd like to create. This method uses Django's `bulk_create` method.

```
import us

state_entities = [
    {
        "name": state.name,
        "fips": state.fips,
        "postal_code": state.abbr,
    } for state in us.states.STATES
]

entities = client.bulk_create(state_entities, domain=states)
```

Note: Django-crosswalk will create UUIDs for any new entities, which are automatically serialized and deserialized by the client.

You can also create entities with your own UUIDs. For example:

```
from uuid import uuid4()

uuid = uuid4()

entities = [
    {
      "uuid": uuid,
      "name": "some entity",
    }
]

entity = client.bulk_create(entities)[0]

entity.uuid == uuid
# True
```

Warning: You can't re-run a bulk create. If your script needs the equivalent of `get_or_create` or `update_or_create`, use the `match` or `match_or_create` methods and then update if needed it using the built-in entity `update` method.

4.4.2 Get entities in a domain

```
entities = client.get_entities(domain=states)

entities[0].name
# Alabama
```

Pass a dictionary of block attributes to filter entities in the domain.

```
entities = client.get_entities(
  domain=states,
  block_attrs={"postal_code": "KS"}
)
```

(continues on next page)

(continued from previous page)

```
entities[0].name
# Kansas
```

4.4.3 Find an entity

Pass a query dictionary to find an entity that *exactly* matches.

```
client.match({"name": "Missouri"}, domain=states)

# Pass block attributes to filter possible matches
client.match(
  {"name": "Texas"},
  block_attrs={"postal_code": "TX"},
  domain=states
)
```

You can also fuzzy match on your query dictionary and return the entity that *best* matches.

```
entity = client.best_match({"name": "Kalifornia"}, domain=states)

# Pass block attributes to filter possible matches
entity = client.best_match(
  {"name": "Kalifornia"},
  block_attrs={"postal_code": "CA"},
  domain=states
)

entity.name == "California"
```

Note: If the match for your query is an alias of another entity, this method will return the canonical entity with `entity.aliased = True`. To ignore aliased entities, set `return_canonical=False` and the method will return the best match for your query, regardless of whether it is an alias for another entity.

```
client.best_match(
  {"name": "Misouri"},
  return_canonical=False
)
```

4.4.4 Find a match or create a new entity

You can create a new entity if an *exact* match isn't found.

```
entity = client.match_or_create({"name": "Narnia"})


entity.created
# True
```

Or use a fuzzy matcher to find the *best* match. If one isn't found above a match threshold returned by your scorer, create a new entity.

```
entity = client.best_match_or_create({"name": "Narnia"})

entity.created
# True

# Set a custom threshold for the match scorer instead of using the default
entity = client.best_match_or_create(
    {"name": "Narnia"},
    threshold=80,
)
```

Note: If the best match for your query is an alias of another entity and is above your match threshold, this method will return the canonical entity with `entity.aliased = True`. To ignore aliased entities, set `return_canonical=False`.

```
client.best_match_or_create(
    {"name": "Misouri"},
    return_canonical=False,
)
```

Pass a dictionary of block attributes to filter match candidates.

```
entity = client.match_or_create(
    {"name": "Narnia"},
    block_attrs={"postal_code": "NA"},
)

entity = client.best_match_or_create(
    {"name": "Narnia"},
    block_attrs={"postal_code": "NA"},
)
```

If a sufficient match is not found, you can pass a dictionary of attributes to create your entity with. These will be combined with your query when creating a new entity.

```
import uuid

id = uuid.uuid4()

entity = client.match_or_create(
    {"name": "Xanadu"},
    create_attrs={"uuid": id},
)

entity = client.best_match_or_create(
    {"name": "Xanadu"},
    create_attrs={"uuid": id},
)

entity.name
# Xanadu
entity.uuid == id
# True
entity.created
# True
```

4.4.5 Create an alias or create a new entity

Create an alias if an entity above a certain match score threshold is found or create a new entity. Method returns the aliased entity.

```
client.set_domain('states')

entity = client.alias_or_create({"name": "Kalifornia"}, threshold=85)

entity.name
# California
entity.aliased
# True

entity = client.alias_or_create(
  {"name": "Alderaan"},
  create_attrs={"galaxy": "Far, far away"}
  threshold=90
)

entity.name
# Alderaan
entity.aliased
# False
```

Note: If the best match for your query is an alias of another entity, this method will return the canonical entity with `entity.aliased = True`. To ignore aliased entities, set `return_canonical=False` and the method will return the best match for your query, regardless of whether it is an alias for another entity.

```
client.alias_or_create(
  {"name": "Missouri"},
  return_canonical=False
)
```

4.4.6 Get an entity by ID

Use the entity's UUID to retrieve it.

```
entity = client.get_entity(uuid)
```

4.4.7 Update an entity by ID

```
entity = client.best_match({"name": "Kansas"})
entity = client.update_by_id(
    entity.uuid,
    {"capital": "Topeka"}
)

entity.capital
# Topeka
```

4.4.8 Update a matched entity

```
entity = client.update_match(
    {"name": "Missouri"},
    update_attrs={"capital": "Jefferson City"},
    domain=states
)

entity.capital
# Jefferson City

entity = client.update_match(
    {"name": "Texas", "postal_code": "TX"},
    update_attrs={"capital": "Austin"},
    domain=states
)

entity.capital
# Jefferson City
```

Note: If your block attributes return more than one matched entity to be updated, an `UnspecificQueryError` will be raised and no entities will be updated.

4.4.9 Delete an entity by ID

```
entity = client.match({"name": "New York"})
deleted = client.delete_by_id(entity.uuid)

deleted
# True
```

4.4.10 Delete a matched entity

```
deleted = client.delete_match({"name": "Xanadu"})

deleted
# True

deleted = client.delete_match({"name": "Narnia", "postal_code": "NA"})

deleted
# True
```

Note: If your block attributes return more than one matched entity to be deleted, an `UnspecificQueryError` will be raised and no entities will be deleted.

4.5 Domain object methods

4.5.1 Update a domain

```
domain = client.get_domain('u-s-states')

domain.update({"parent": "countries"})
```

4.5.2 Set a parent domain

```
parent_domain = client.get_domain('countries')
domain = client.get_domain('u-s-states')

domain.set_parent(parent_domain)
```

4.5.3 Remove a parent domain

```
domain = client.get_domain('u-s-states')

domain.remove_parent()

domain.parent
# None
```

4.5.4 Delete a domain

```
domain = client.get_domain('u-s-states')

domain.delete()

domain.deleted
# True
```

4.5.5 Get domain's entities

```
domain = client.get_domain('u-s-states')

# Get all states
domain.get_entities()

# Filter entities using block attributes
entities = domain.get_entities({"postal_code": "KS"})
entities[0].name == "Kansas"
```

4.6 Entity object methods

4.6.1 Access an entity's attributes

```
entity = client.match({"name": "Texas"})

# See what user-defined attributes are set
entity.attrs() == ["fips", "name", "postal_code", "uuid"]

# Access a specific attribute
entity.attrs("postal_code") == "TX"
entity.postal_code == "TX"

# Raise AttributeError if undefined
entity.attrs("undefined_attr")
entity.undefined_attr
```

4.6.2 Update an entity

```
entity = client.best_match({"name": "Texas"})

entity.update({"capitol": "Austin"})
```

4.6.3 Alias entities

```
entity = client.best_match({"name": "Missouri"})
alias = client.best_match({"name": "Show me state"})

alias.set_alias_for(entity)

alias.alias_for == entity.uuid
# True
```

4.6.4 Remove an alias

```
alias = client.best_match({"name": "Show me state"})

alias.remove_alias_for()

alias.alias_for
# None
```

4.6.5 Set a superseding entity

```
superseded = client.best_match({"name": "George W. Bush"}, domain="politicians")
entity = client.best_match({"name": "George W. Bush"}, domain="presidents")

superseded.set_superseded_by(entity)
```

(continues on next page)

(continued from previous page)

```
superseded.superseded_by == entity.uuid
# True
```

4.6.6 Remove a superseding entity

```
superseded = client.best_match({"name": "George W. Bush"}, domain="politicians")

superseded.remove_superseded_by()

superseded.superseded_by
# None
```

4.6.7 Delete an entity

```
entity = client.best_match({"name": "Texas"})

entity.delete()

entity.deleted
# True
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search