django-cqrs

Release 2.4.4.dev2+g4c11092.d20221201

CloudBlue

Dec 01, 2022

CONTENTS:

django-cqrs is an Django application, that implements CQRS data synchronisation between several Django microservices.

CHAPTER

ONE

CQRS

In CloudBlue Connect we have a rather complex Domain Model. There are many microservices, that are decomposed by subdomain and which follow database-per-service pattern. These microservices have rich and consistent APIs. They are deployed in cloud k8s cluster and scale automatically under load. Many of these services aggregate data from other ones and usually API Composition is totally enough. But, some services are working too slowly with API JOINS, so another pattern needs to be applied.

The pattern, that solves this issue is called CQRS - Command Query Responsibility Segregation. Core idea behind this pattern is that view databases (replicas) are defined for efficient querying and DB joins. Applications keep their replicas up to date by subscribing to Domain events published by the service that owns the data. Data is eventually consistent and that's okay for non-critical business transactions.

1.1 Getting started

Note: This guide assumes that you have at least a single instance of RabbitMQ up and running. For other messaging brokers/transports please see *Transports*.

1.1.1 Requirements

django-cqrs works with Python 3.7 or later and has the following dependencies:

- Django >= 2.2
- pika >= 1.0.0
- kombu >= 4.6
- ujson >= 3.0.0
- django-model-utils >= 4.0.0
- python-dateutil >= 2.4

1.1.2 Install

django-cqrs can be installed from pypi.org with pip:

```
$ pip install django-cqrs
```

1.1.3 Master service

Configure master service

Add dj_cqrs to Django `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    ...
    'dj_cqrs',
    ...
]
```

and add the *django-cqrs* configuration:

```
CQRS = {
    'transport': 'dj_cqrs.transport.RabbitMQTransport',
    'url': 'amqp://guest:guest@rabbit:5672/'
}
```

Setup master models

To setup master models add the `dj_cqrs.mixins.MasterMixin` to your model.

For example:

```
from django.db import models

from dj_cqrs.mixins import MasterMixin


class MyMasterModel(MasterMixin, models.Model):

    CQRS_ID = 'my_model'  # each model must have its unique CQRS_ID

    my_field = models.CharField(max_length=100)
    ....
```

Create and run migrations for master

Since the `MasterMixin` adds the `cqrs_revision` and `cqrs_updated` fields to the model, you must create a new migration for it:

```
$ ./manage.py makemigrations
$ ./manage.py migrate
```

Run your django application

```
$ ./manage.py runserver
```

1.1.4 Replica service

Configure replica service

Add dj_cqrs to Django `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    ...
    'dj_cqrs',
    ...
]
```

and add the *django-cqrs* configuration:

```
CQRS = {
    'transport': 'dj_cqrs.transport.RabbitMQTransport',
    'url': 'amqp://guest:guest@rabbit:5672/',
    'queue': 'my_replica', # Each replica service must have a unique queue.
}
```

Setup replica models

To setup replica models add the `dj_cqrs.mixins.ReplicaMixin` to each model.

For example:

```
from django.db import models

from dj_cqrs.mixins import ReplicaMixin


class MyReplicaModel(ReplicaMixin, models.Model):

    CQRS_ID = 'my_model'

    my_field = models.CharField(max_length=100)
    ....
```

Create and run migrations for replica

Since the `ReplicaMixin` adds the `cqrs_revision` and `cqrs_updated` fields to the model, you must create a new migration for it:

```
$ ./manage.py makemigrations
$ ./manage.py migrate
```

Run consumer process

```
$ ./manage.py cqrs_consume -w 2
```

And that's all!

Now every time you modify your master model, changes are replicated to all services that have a replica model with the same CQRS_ID.

1.1.5 Use of customized meta data

The library allow us to send customized metadata from the Master models to the Replica ones.

Configuring the metadata for Master model

There are two ways to specify what we want to include in this metadata, overriding the master function or setting a default generic function that will be executed for all masters.

Override master function

Inside the Master model class you have to add the **get_cqrs_meta** function that will replace the default one (that returns an empty dict). For instance if you want to return the access of a given model instance inside the metadata you could do the following:

```
def get_cqrs_meta(self, **kwargs):
    meta = super().get_cqrs_meta(**kwargs)
    if self.is_owner():
        meta['access']['owner'] = True
        meta['access']['others'] = False
    else:
        meta['access']['owner'] = False
        meta['access']['others'] = True
    return meta
```

Setting a default generic function

In the django settings you could configure a function that will be executed everytime an event is emitted in any Master:

```
from ... import get_cqrs_meta

CQRS = {
    ...
    'master': {
        ...
        'meta_function': get_cqrs_meta,
    },
}
```

Retrieving the metadata from the Replica model

From the replica model you will now receive an additional parameter called **meta** that will contain all metadata set in the Master model. These data will be present in the following class functions: * cqrs_update * cqrs_create * cqrs_delete

For instance replacing the **cqrs_update** we could do something like:

```
def cqrs_update(self, sync, mapped_data, previous_data=None, meta=None):
    if meta and not meta['access']['owner']:
        # Call asynchronously external system to update some resource.
    else:
        # Call asynchronously internal system to update some resource.
    return super().cqrs_update(sync, mapped_data, previous_data, meta)
```

1.2 Django Admin

1.2.1 Synchronize items

Add action to synchronize master items from Django Admin page.

```
from django.db import models
from django.contrib import admin

from dj_cqrs.admin_mixins import CQRSAdminMasterSyncMixin


class AccountAdmin(CQRSAdminMasterSyncMixin, admin.ModelAdmin):
    pass


admin.site.register(models.Account, AccountAdmin)
```

- If necessary, override `_cqrs_sync_queryset` from `CQRSAdminMasterSyncMixin` to adjust the QuerySet and use it for synchronization.

1.3 Custom serialization

By default, *django-cqrs* serializes all the fields declared for the master model or the subset specified by the `CQRS_FIELDS` attribute.

Sometimes you want to customize how the master model will be serialized, for example including some other fields from related models.

Warning: When there are master models with related entities in CQRS_SERIALIZER, it's important to have operations within atomic transactions. CQRS sync will happen on transaction commit. Please, avoid saving master model within transaction more then once to reduce syncing and potential racing on replica side. Updating of related model won't trigger CQRS automatic synchronization for master model. This needs to be done manually.

1.3.1 Master service

In this case you can control how an instance of the master model is serialized providing a serializer class to be used for that:

```
class MyMasterModel(MasterMixin):
    CQRS_ID = 'my_model'
    CQRS_SERIALIZER = 'mymodule.serializers.MyMasterModelSerializer'

    @classmethod
    def relate_cqrs_serialization(cls, queryset):
        # Optimize related models fetching here
        return queryset
```

If you would to serialize fields from related models, you can optimize database access overriding the `relate_cqrs_serialization` method using the select_related and prefetch_related methods of the QuerySet object.

1.3.2 Replica service

If you provide a serializer to customize serialization, you must handle yourself deserialization for the replica model.

```
class MyReplicaModel(ReplicaMixin):
    CQRS_ID = 'my_model'
    CQRS_CUSTOM_SERIALIZATION = True # bypass default deserialization.

    @classmethod
    def cqrs_create(cls, sync, mapped_data, previous_data=None, meta=None):
        # Custom deserialization logic here
        pass

    def cqrs_update(self, sync, mapped_data, previous_data=None, meta=None):
        # Custom deserialization logic here
        pass
```

Note: A serializer class must follow these rules:

- The constructor must accept the model instance as the only positional argument
- Must have a `data` property that returns a python dictionary as the instance representation.

If your service exposes a RESTful API written using Django REST framework you can use your model serializers out of the box also for CQRS serialization.

1.4 Keep track of changes to fields

In some circumstances, you want to keep track of changes made on some fields of the master model.

django-cqrs can send previous values of the tracked fields to replicas.

To do so, you can use the `CQRS_TRACKED_FIELDS` attribute to specify which fields to track:

```
class MyMasterModel(MasterMixin):

    CQRS_ID = 'my_model'
    CQRS_TRACKED_FIELDS = ('char_field', 'parent', 'status')


    char_field = models.CharField(max_length=100)
    status = models.CharField(max_length=15, choices=STATUSES)

    parent = models.ForeignKey(ParentMode, on_delete=models.CASCADE)
```

This way, you can override the `cqrs_save` and apply your persistence logic based o tracked fields before accessing your database:

```
class MyReplicaModel(ReplicaMixin):

    CQRS_ID = 'my_model'

    @classmethod
    def cqrs_save(cls, master_data, previous_data=None, sync=False):
        # Custom logic based on previous_data here.
        pass
```

Note: The fields tracking features honors the `CQRS_MAPPING` attribute.

Note: The fields tracking features relies on the FieldTracker utility class from the django-model-utils library.

1.5 Transports

django-cqrs ships with two transport that allow users to choose the messaging broker that best fit their needs.

1.5.1 RabbitMQ transport

The *dj_cqrs.transport.RabbitMQTransport* transport is based on the pika messaging library.

To configure the `RabbitMQTransport` you must provide the rabbitmq connection url:

```
CQRS = {
    'transport': 'dj_cqrs.transport.RabbitMQTransport',
    'url': 'amqp://guest:guest@rabbit:5672/'
}
```

Warning: Previous versions of the `RabbitMQTransport` use the attributes `host`, `port`, `user`, `password` to configure the connection with rabbitmq. These attributes are deprecated and will be removed in future versions of *django-cqrs*.

1.5.2 Kombu transport

The *dj_cqrs.transport.KombuTransport* transport is based on the kombu messaging library.

Kombu supports different messaging brokers like RabbitMQ, Redis, Amazon SQS etc.

To configure the `KombuTransport` you must provide the rabbitmq connection url:

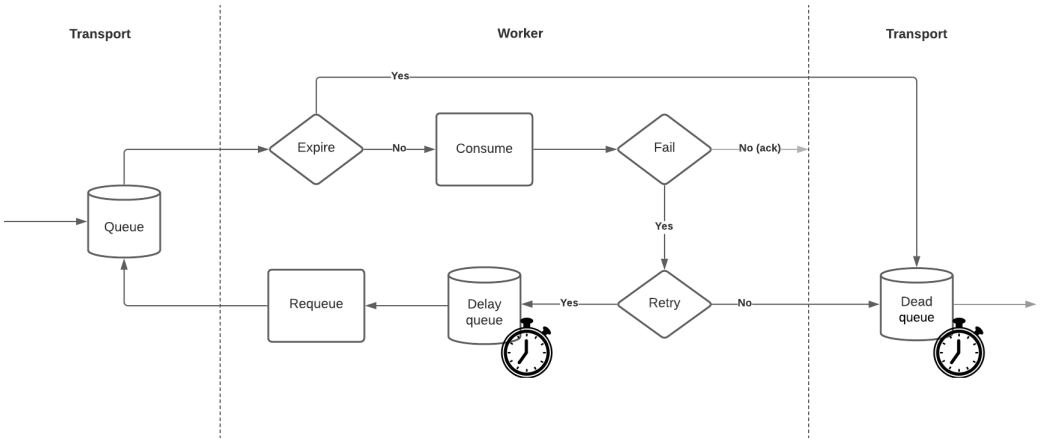
```
CQRS = {
    'transport': 'dj_cqrs.transport.KombuTransport',
    'url': 'redis://redis:6379/'
}
```

Please read Transport Comparison and URLs articles for Kombu to get more information on supported brokers and configuration urls.

1.6 Message lifecycle

Warning: Expiration, retrying and 'dead letters' queueing supported in `RabbitMQTransport` only (**on** by default).

django-cqrs since version 1.11 provides mechanism for reliable message delivery.



1.6.1 Expiration

Name	Default	Description
CQRS_MESSAGE_TTL	86400	Limits message lifetime in **seconds**, then it will be moved to 'dead letters' queue.

```
# settings.py

CQRS = {
```

(continues on next page)

(continued from previous page)

```
    ...
    'master': {
        'CQRS_MESSAGE_TTL': 86400, # 1 day
    },
}
```

1.6.2 Fail

Message assumed as failed when a consumer raises an exception or returns negative boolean value (*False*, *None*, etc).

```
# models.py

class Example(ReplicaMixin, models.Model):
    CQRS_ID = 'example'
    ...

    @classmethod
    def cqrs_create(cls, sync, mapped_data, previous_data=None, meta=None):
        raise Exception("Some issue during create") # exception could be caught at␣
↪should_retry_cqrs() method

    def cqrs_update(self, sync, mapped_data, previous_data=None, meta=None):
        return None # returning negative boolean for retrying
```

1.6.3 Retrying

Name	Default	Description
CQRS_MAX_RETRIES	30	Maximum number of retry attempts. Infinite if *None*, 0 to disable retries.
CQRS_RETRY_DELAY	2	Constant delay in **seconds** between message failure and requeueing.
CQRS_DELAY_QUEUE_MAX_SIZE	1000	Maximum number of delayed messages per worker. Infinite if *None*.

```
# settings.py

CQRS = {
    ...
    'replica': {
        'CQRS_MAX_RETRIES': 30, # attempts
        'CQRS_RETRY_DELAY': 2,  # seconds
        'CQRS_DELAY_QUEUE_MAX_SIZE': 1000,
    },
}
```

Customization

The *dj_cqrs.mixins.ReplicaMixin* allows to take full control on retrying.

```
# models.py

class Example(ReplicaMixin, models.Model):
    CQRS_ID = 'example'
    ...

    @classmethod
    def get_cqrs_retry_delay(cls, current_retry=0):
        # Linear delay growth
        return (current_retry + 1) * 60

    @classmethod
    def should_retry_cqrs(cls, current_retry, exception=None):
        # Retry 10 times or until we have troubles with database
        return (
            current_retry < 10
            or isinstance(exception, django.db.OperationalError)
        )
```

1.6.4 Dead letters

Expired or failed messages which should not be retried are moved to 'dead letters' queue.

Name	Default	Description
dead_letter_queue	'dead_letter_' + queue	Queue name for dead letters.
dead_message_ttl	864000	Expiration **seconds**. Infinite if *None*.

```
# settings.py

CQRS = {
    ...
    'queue': 'example',
    'replica': {
        ...
        'dead_letter_queue': 'dead_letter_example', # generated from CQRS.queue
        'dead_message_ttl': 864000, # 10 days
    },
}
```

Commands

Dump

Dumps all dead letters to stdout.

```
$ python manage.py cqrs_dead_letters dump
{"signal_type":"SAVE","cqrs_id":"example","instance_data":{"id":1,"cqrs_revision":0,
↪"cqrs_updated":"2021-04-30 11:50:05.164341+00:00"},"previous_data":null,"instance_pk
↪":135,"correlation_id":null,"retries":30,"expires":"2021-05-01T11:50:00+00:00"}
```

Retry

Retry all dead letters. Message body retries and expires fields are downgraded.

```
$ python manage.py cqrs_dead_letters retry
Total dead letters: 1
Retrying: 1/1
{"signal_type":"SAVE","cqrs_id":"example","instance_data":{"id":1,"cqrs_revision":0,
↪"cqrs_updated":"2021-04-30 11:50:05.164341+00:00"},"previous_data":null,"instance_pk
↪":135,"correlation_id":null,"retries":0,"expires":"2021-05-02T12:30:00+00:00"}
```

Purge

Removes all dead letters.

```
$ python manage.py cqrs_dead_letters purge
Total dead letters: 1
Purged
```

1.7 Utilities

1.7.1 Bulk synchronizer without transport

Usage example: it may be used for initial configuration and/or may be used at planned downtime.

On master service:

```
$ python manage.py cqrs_bulk_dump --cqrs-id=author --output author.dump
```

On replica service:

```
$ python manage.py cqrs_bulk_load --input=author.dump
```

1.7.2 Filter synchronizer over transport

Usage example: sync some specific records to a given replica. Can be used dynamically.

To sync all replicas:

```
$ python manage.py cqrs_sync --cqrs-id=author --filter="{\"id__in\": [1, 2]}"
```

To sync all instances only with one replica:

```
$ python manage.py cqrs_sync --cqrs-id=author --filter="{}" --queue=replica
```

1.8 API Reference

1.8.1 Django Admin

class dj_cqrs.admin.**CQRSAdminMasterSyncMixin**

Mixin that includes a custom action in AdminModel. This action allows synchronizing master's model items from Django Admin page,

_cqrs_sync_queryset(*queryset*)

This function is used to adjust the QuerySet before sending the sync signal.

Parameters **queryset** (*Queryset*) – Original queryset

Returns Updated queryset

Return type Queryset

sync_items(*request*, *queryset*)

This method synchronizes selected items from the Admin Page. It is registered as a custom action in Django Admin

1.8.2 Mixins

class dj_cqrs.mixins.**RawMasterMixin**(**args*, ***kwargs*)

Base class for MasterMixin. **Users shouldn't use this class directly.**

CQRS_ID = None

Unique CQRS identifier for all microservices.

CQRS_PRODUCE = True

If false, no cqrs data is sent through the transport.

CQRS_FIELDS = '__all__'

List of fields to include in the CQRS payload. You can also set the fields attribute to the special value '__all__' to indicate that all fields in the model should be used.

CQRS_SERIALIZER = None

Optional serializer used to create the instance representation. Must be expressed as a module dotted path string like *mymodule.serializers.MasterModelSerializer*.

CQRS_TRACKED_FIELDS = None

List of fields of the main model for which you want to track the changes and send the previous values via transport. You can also set the field attribute to the special value "__all__" to indicate that all fields in the model must be used.

cqrs

Manager that adds needed CQRS queryset methods.

property cqrs_saves_count

Shows how many times this instance has been saved within the transaction.

property is_initial_cqrs_save

This flag is used to check if instance has already been registered for CQRS update.

reset_cqrs_saves_count()

This method is used to automatically reset instance CQRS counters on transaction commit. But this can also be used to control custom behaviour within transaction or in case of rollback, when several sequential transactions are used to change the same instance.

to_cqrs_dict(*using=None*, *sync=False*)
CQRS serialization for transport payload.

Parameters **using** (*str, optional*) – The using argument can be used to force the database to use, defaults to None

Returns The serialized instance data.

Return type dict

get_tracked_fields_data()
CQRS serialization for tracked fields to include in the transport payload.

Returns Previous values for tracked fields.

Return type dict

cqrs_sync(*using=None*, *queue=None*)
Manual instance synchronization.

Parameters

- **using** (*str, optional*) – The using argument can be used to force the database to use, defaults to None
- **queue** (*str, optional*) – Syncing can be executed just for a single queue, defaults to None (all queues)

Returns True if instance can be synced, False otherwise.

Return type bool

is_sync_instance()
This method can be overridden to apply syncing only to instances by some rules. For example, only objects with special status or after some creation date, etc.

Returns True if this instance needs to be synced, False otherwise

Return type bool

get_cqrs_meta(***kwargs*)
This method can be overridden to collect model/instance specific metadata.

Returns Metadata dictionary if it's provided.

Return type dict

classmethod relate_cqrs_serialization(*queryset*)
This method shoud be overriden to optimize database access for example using *select_related* and *prefetch_related* when related models must be included into the master model representation.

Parameters **queryset** (*django.db.models.QuerySet*) – The initial queryset.

Returns The optimized queryset.

Return type django.db.models.QuerySet

get_custom_cqrs_delete_data()
This method should be overridden when additional data is needed in DELETE payload.

classmethod call_post_bulk_create(*instances*, *using=None*)
Post bulk create signal caller (django doesn't support it by default).

```
# Used automatically by cqrs.bulk_create()
instances = model.cqrs.bulk_create(instances)
```

classmethod call_post_update(*instances*, *using=None*)
Post bulk update signal caller (django doesn't support it by default).

```
# Used automatically by cqrs.bulk_update()
qs = model.objects.filter(k1=v1)
model.cqrs.bulk_update(qs, k2=v2)
```

class dj_cqrs.mixins.**MasterMixin**(**args*, ***kwargs*)
Bases: *dj_cqrs.mixins.RawMasterMixin*

Mixin for the master CQRS model, that will send data updates to it's replicas.

class dj_cqrs.mixins.**ReplicaMixin**(**args*, ***kwargs*)
Mixin for the replica CQRS model, that will receive data updates from master. Models, using this mixin should be readonly, but this is not enforced (f.e. for admin).

CQRS_ID = None
Unique CQRS identifier for all microservices.

CQRS_MAPPING = None
Mapping of master data field name to replica model field name.

CQRS_CUSTOM_SERIALIZATION = False
Set it to True to skip default data check.

CQRS_SELECT_FOR_UPDATE = False
Set it to True to acquire lock on instance creation/update.

CQRS_NO_DB_OPERATIONS = False
Set it to True to disable any default DB operations for this model.

CQRS_META = False
Set it to True to receive meta data for this model.

cqrs
Manager that adds needed CQRS queryset methods.

classmethod cqrs_save(*master_data*, *previous_data=None*, *sync=False*, *meta=None*)
This method saves (creates or updates) model instance from CQRS master instance data. This method must not be overridden. Otherwise, sync checks need to be implemented manually.

Parameters

- **master_data** (*dict*) – CQRS master instance data.
- **previous_data** (*dict*) – Previous values for tracked fields.
- **sync** (*bool*) – Sync package flag.
- **or None meta** (*dict*) – Payload metadata, if exists.

Returns Model instance.

Return type django.db.models.Model

classmethod cqrs_create(*sync*, *mapped_data*, *previous_data=None*, *meta=None*)
This method creates model instance from CQRS mapped instance data. It must be overridden by replicas of master models with custom serialization.

Parameters

- **sync** (*bool*) – Sync package flag.
- **mapped_data** (*dict*) – CQRS mapped instance data.

- **previous_data** (*dict*) – Previous mapped values for tracked fields.
- **or None meta** (*dict*) – Payload metadata, if exists.

Returns Model instance.

Return type django.db.models.Model

cqrs_update(*sync*, *mapped_data*, *previous_data=None*, *meta=None*)
This method updates model instance from CQRS mapped instance data. It must be overridden by replicas of master models with custom serialization.

Parameters

- **sync** (*bool*) – Sync package flag.
- **mapped_data** (*dict*) – CQRS mapped instance data.
- **previous_data** (*dict*) – Previous mapped values for tracked fields.
- **or None meta** (*dict*) – Payload metadata, if exists.

Returns Model instance.

Return type django.db.models.Model

classmethod cqrs_delete(*master_data*, *meta=None*)
This method deletes model instance from mapped CQRS master instance data.

Parameters

- **master_data** (*dict*) – CQRS master instance data.
- **or None meta** (*dict*) – Payload metadata, if exists.

Returns Flag, if delete operation is successful (even if nothing was deleted).

Return type bool

1.8.3 Managers

class dj_cqrs.managers.**MasterManager**(**args*, ***kwargs*)

bulk_create(*objs*, ***kwargs*)
Custom bulk create method to support sending of create signals. This can be used only in cases, when IDs are generated on client or DB returns IDs.

Parameters

- **objs** (*django.db.models.Model*) – List of objects for creation
- **kwargs** – Bulk create kwargs

bulk_update(*queryset*, ***kwargs*)
Custom update method to support sending of update signals.

Parameters

- **queryset** (*django.db.models.QuerySet*) – Django Queryset (f.e. filter)
- **kwargs** – Update kwargs

class dj_cqrs.managers.**ReplicaManager**(**args*, ***kwargs*)

save_instance(*master_data*, *previous_data=None*, *sync=False*, *meta=None*)
This method saves (creates or updates) model instance from CQRS master instance data.

Parameters

- **master_data** (*dict*) – CQRS master instance data.
- **previous_data** (*dict*) – Previous values for tracked fields.
- **sync** (*bool*) – Sync package flag.
- **or None meta** (*dict*) – Payload metadata, if exists.

Returns Model instance.

Return type django.db.models.Model

create_instance(*mapped_data*, *previous_data=None*, *sync=False*, *meta=None*)
This method creates model instance from mapped CQRS master instance data.

Parameters

- **mapped_data** (*dict*) – Mapped CQRS master instance data.
- **previous_data** (*dict*) – Previous values for tracked fields.
- **sync** (*bool*) – Sync package flag.
- **or None meta** (*dict*) – Payload metadata, if exists.

Returns ReplicaMixin model instance.

Return type django.db.models.Model

update_instance(*instance*, *mapped_data*, *previous_data=None*, *sync=False*, *meta=None*)
This method updates model instance from mapped CQRS master instance data.

Parameters

- **instance** (*django.db.models.Model*) – ReplicaMixin model instance.
- **mapped_data** (*dict*) – Mapped CQRS master instance data.
- **previous_data** (*dict*) – Previous values for tracked fields.
- **or None meta** (*dict*) – Payload metadata, if exists.
- **sync** (*bool*) – Sync package flag.

Returns ReplicaMixin model instance.

Return type django.db.models.Model

delete_instance(*master_data*)
This method deletes model instance from mapped CQRS master instance data.

Parameters **master_data** (*dict*) – CQRS master instance data.

Returns Flag, if delete operation is successful (even if nothing was deleted).

Return type bool

1.8.4 Signals

dj_cqrs.signals.**post_bulk_create = <django.dispatch.dispatcher.Signal object>**
Signal sent after a bulk create. See dj_cqrs.mixins.RawMasterMixin.call_post_bulk_create.

dj_cqrs.signals.**post_update = <django.dispatch.dispatcher.Signal object>**
Signal sent after a bulk update. See dj_cqrs.mixins.RawMasterMixin.call_post_update.

class dj_cqrs.signals.**MasterSignals**
Signals registry and handlers for CQRS master models.

classmethod register_model(*model_cls*)
Registers signals for a model.

Parameters **model_cls** (dj_cqrs.mixins.MasterMixin) – Model class inherited from CQRS MasterMixin.

classmethod post_save(*sender*, ***kwargs*)

Parameters **sender** (dj_cqrs.mixins.MasterMixin) – Class or instance inherited from CQRS MasterMixin.

classmethod post_delete(*sender*, ***kwargs*)

Parameters **sender** (dj_cqrs.mixins.MasterMixin) – Class or instance inherited from CQRS MasterMixin.

classmethod post_bulk_create(*sender*, ***kwargs*)

Parameters **sender** (dj_cqrs.mixins.MasterMixin) – Class or instance inherited from CQRS MasterMixin.

classmethod post_bulk_update(*sender*, ***kwargs*)

Parameters **sender** (dj_cqrs.mixins.MasterMixin) – Class or instance inherited from CQRS MasterMixin.

1.8.5 Transports

class dj_cqrs.transport.**BaseTransport**
CQRS pattern can be implemented over any transport (AMQP, HTTP, etc.) All transports need to inherit from this base class. Transport must be set in Django settings:

```
CQRS = {
    'transport': 'dj_cqrs.transport.rabbit_mq.RabbitMQTransport',
}
```

static produce(*payload*)
Send data from master model to replicas.

Parameters **payload** (dj_cqrs.dataclasses.TransportPayload) – Transport payload from master model.

static consume(**args*, ***kwargs*)
Receive data from master model.

static clean_connection(**args*, ***kwargs*)
Clean transport connection. Here you can close all connections that you have

class dj_cqrs.transport.**RabbitMQTransport**

classmethod clean_connection()
Clean transport connection. Here you can close all connections that you have

classmethod consume(*cqrs_ids=None*)
Receive data from master model.

classmethod produce(*payload*)
Send data from master model to replicas.

Parameters **payload** (dj_cqrs.dataclasses.TransportPayload) – Transport payload from master model.

class dj_cqrs.transport.**KombuTransport**

classmethod clean_connection()
Nothing to do here

classmethod consume(*cqrs_ids=None*)
Receive data from master model.

classmethod produce(*payload*)
Send data from master model to replicas.

Parameters **payload** (dj_cqrs.dataclasses.TransportPayload) – Transport payload from master model.

class dj_cqrs.constants.**SignalType**
Type of signal that generates this event.

SAVE = 'SAVE'
The master model has been saved.

DELETE = 'DELETE'
The master model has been deleted.

SYNC = 'SYNC'
The master model needs syncronization.

class dj_cqrs.dataclasses.**TransportPayload**(*signal_type*, *cqrs_id*, *instance_data*, *instance_pk*, *queue=None*, *previous_data=None*, *correlation_id=None*, *expires=None*, *retries=0*, *meta=None*)
Transport message payload.

Parameters

- **signal_type** (dj_cqrs.constants.SignalType) – Type of the signal for this message.
- **cqrs_id** (*str*) – The unique CQRS identifier of the model.
- **instance_data** (*dict*) – Serialized data of the instance that generates the event.
- **instance_pk** – Primary key of the instance.
- **queue** (*str, optional*) – Queue to synchronize, defaults to None.
- **previous_data** (*dict, optional*) – Previous values for fields tracked for changes, defaults to None.
- **correlation_id** (*str, optional*) – Correlation ID of process, where this payload is used.
- **retries** (*int, optional*) – Current number of message retries.

- **expires** (*datetime, optional*) – Message expiration datetime, infinite if None
- **meta** (*dict, optional*) – Payload metadata

classmethod from_message(*dct*)
Builds payload from message data.

Parameters **dct** (*dict*) – Deserialized message body data.

Returns TransportPayload instance.

Return type *TransportPayload*

to_dict()
Return the payload as a dictionary.

Returns This payload.

Return type dict

is_expired()
Checks if this payload is expired.

Returns True if payload is expired, False otherwise.

Return type bool

1.8.6 Registries

class dj_cqrs.registries.**MasterRegistry**

classmethod register_model(*model_cls*)
Registration of CQRS model identifiers.

classmethod get_model_by_cqrs_id(*cqrs_id*)
Returns the model class given its CQRS_ID.

Parameters **cqrs_id** (*str*) – The CQRS_ID of the model to be retrieved.

Returns The model that correspond to the given CQRS_ID or None if it has not been registered.

Return type django.db.models.Model

class dj_cqrs.registries.**ReplicaRegistry**

classmethod register_model(*model_cls*)
Registration of CQRS model identifiers.

classmethod get_model_by_cqrs_id(*cqrs_id*)
Returns the model class given its CQRS_ID.

Parameters **cqrs_id** (*str*) – The CQRS_ID of the model to be retrieved.

Returns The model that correspond to the given CQRS_ID or None if it has not been registered.

Return type django.db.models.Model

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

Symbols

B

C

D