django-data-watcher

Vinta Serviços e Soluções Tecnológicas LTDA

Sep 28, 2022

USER GUIDE

Django Data Watcher is a library that will make easier to create/mantain side-effects of data operations in your django models.

It tries to fix some of *Django Signals*' problems, being reusable, giving visibility of the side-effects of doing data operations in a specific model, and also if some hook is triggered by a queryset operation it runs only once, giving you responsability of dealing with the queryset instead running once for each affected instance.

It's very practical to use and you can improve the readbility and performance of your data services.

Document version: 1.1.0

ONE

INSTALLATION

Django Data Watcher can be installed from PyPI with tools like pip:

```
pip install django-watcher
```

1.1 Requirements

Django Data Watcher is tested against all supported versions of Python ,

- **Python**: 3.6, 3.7, 3.8, 3.9

GETTING STARTED

Here you will understand how to use our WatcherMixins to decorate your Django Models and take advantage of the Hooks

1. Extend a base mixin, implementing the desired hook. check *Create Your Watcher* and *Available Mixins*.

2. Decorate you Model with **watched** decorator, check *Decorate Your Model*

2.1 The Hooks

Every basic data operation in Django can trigger a hook (Except read operations). And we have these hooks availables:

- **pre_create** called by: *CreateWatcherMixin*, and *SaveWatcherMixin*

- **pre_update** called by: *UpdateWatcherMixin*, and *SaveWatcherMixin*

- **pre_save** called by: *SaveWatcherMixin*

- **pre_delete** called by: *DeleteWatcherMixin*

- **post_create** called by: *CreateWatcherMixin*, and *SaveWatcherMixin*

- **post_update** called by: *UpdateWatcherMixin*, and *SaveWatcherMixin*

- **post_save** called by: *SaveWatcherMixin*

- **post_delete** called by: *DeleteWatcherMixin*

Each hook is a classmethod, it will always have the *target* param, update and create hooks will also have the *meta_params* param.

2.1.1 Target

The Target represents the objects afected by the current operation.

It can be a already filtered QuerySet or a List which instances.

Each hook signature will specify the type of the target, but you can infer thinking like: "Is possible to have a queryset here?" in pre_create hooks is not so you will receive a list of objects.

To check hook signature go to the specific mixin.

2.1.2 MetaParams

The Metaparams is a TypedDict which will inform you about the trigger of the current operation:

```
source: str  # "queryset" or "instance"
operation_params: dict  # is the kwargs of the trigger operation
instance_ref: optional[models.Model]  # in instance operations triggered by instances it
→will bring the reference to the instance that the operation was called
```

2.2 Create Your Watcher

The Watcher class is the core of our project, on that you will coordinate the hooks of your model.

We do give 3 basic mixins (*DeleteWatcherMixin*, *CreateWatcherMixin*, *UpdateWatcherMixin*) which will control your data flow.

Also, exists a 4th mixin that is a mix up of Create and Update mixins: *SaveWatcherMixin*.

These mixins will call the hooks in the approprieted order together with the desired operation everything inside a transaction, and it will Rollback if something goes wrong.

How to extend a basic mixins:

```python
# my_app.watchers.py

from __future__ import annotation

from typing import TYPE_CHECKING, List

from django_watcher.mixins import CreateWatcherMixin, DeleteWatcherMixin

from .tasks import send_deletion_email

if TYPE_CHECKING:
    from .models import MyModel


class MyModelWatcher(CreateWatcherMixin, DeleteWatcherMixin):
    @classmethod
    def post_delete(cls, undeleted_instances: List[MyModel]):
        send_deletion_email(undeleted_instances)

    @classmethod
    def pre_create(cls, target: List[MyModel], meta_params: dict):
        # do transformation, call functions, whatever you feel necessary
```

Usage of type hints is optional:

```python
# my_app.watchers.py

from django_watcher.mixins import CreateWatcherMixin, DeleteWatcherMixin

from .tasks import send_deletion_email
```

```python
class MyModelWatcher(CreateWatcherMixin, DeleteWatcherMixin):
    @classmethod
    def post_delete(cls, undeleted_instances):
        send_deletetion_email(undeleted_instances)

    @classmethod
    def pre_create(cls, target, meta_params):
        # do transformation, call functions, whatever you feel necessary
```

This section is only to show how easy is to use, but you can dive deep on the next section *Available Mixins* to check what are the available parameters of the hooks.

2.3 Available Mixins

2.3.1 DeleteWatcherMixin

The DeleteWatcherMixin extends our *AbstractWatcher* has the following hooks:

```python
@classmethod
def pre_delete(cls, target: models.QuerySet) -> None:
    ...

@classmethod
def post_delete(cls, undeleted_instances: List[D]) -> None:
    ...
```

2.3.2 CreateWatcherMixin

The CreateWatcherMixin extends our *AbstractWatcher* has the following hooks:

```python
@classmethod
def pre_create(cls, target: List['CreatedModel'], meta_params: MetaParams) -> None:
    ...

@classmethod
def post_create(cls, target: models.QuerySet, meta_params: MetaParams) -> None:
    ...
```

To understand what is *MetaParams*, click on the link.

2.3.3 UpdateWatcherMixin

The UpdateWatcherMixin extends our *AbstractWatcher* and has the following hooks:

```
@classmethod
def pre_update(cls, target: models.QuerySet, meta_params: MetaParams) -> None:
    ...

@classmethod
def post_update(cls, target: models.QuerySet, meta_params: MetaParams) -> None:
    ...
```

To understand what is *MetaParams*, click on the link.

2.3.4 SaveWatcherMixin

The SaveWatcherMixin extends *CreateWatcherMixin* and *UpdateWatcherMixin* has the same hooks of it supers and:

```
@classmethod
def pre_save(cls, target: Union[List['CreatedModel'], models.QuerySet], meta_params:␣
↪MetaParams) -> None:
    pass

@classmethod
def post_save(cls, target: models.QuerySet, meta_params: MetaParams) -> None:
    pass
```

pre_save and *post_save* hooks will **always** run.

Create hooks order:

1. **pre_save**
2. **pre_create**
3. **create**
4. **post_create**
5. **post_save**

Update hooks order:

1. **pre_save**
2. **pre_update**
3. **update**
4. **post_update**
5. **post_save**

To understand what is *MetaParams*, click on the link.

2.4 Decorate Your Model

Setting the Watcher on the model:

```python
# You will always decorate your model
from django_watcher.decorators import watched

# Approach #1 - Import the watcher locally
from my_app.whatchers import MyWatcher

@watched(MyWatcher)
class MyModel(models.Model):
    ...

# Approach #2 - Give a custom path
@watched('my_app.services.watchers.MyWatcher')
class MyModel(models.Model):
    ...

# Approach #3 - Give de module name + watcher name if is inside a `watchers.py` of the
→same django app
@watched('my_app.MyWatcher')
class MyModel(models.Model):
    ...
```

2.4.1 Using others than default django managers

Also if you have other managers (aside from *objects*) you can declarate it, on the seccond param of the *watched* decorator, default value is *['objects']*:

```python
from django_watcher.decorators import watched

@watched('my_app.MyWatcher', ['objects', 'deleted_objects'])
class MyModel(models.Model):
    ...
```

TUTORIAL

3.1 Migrating from Django Signals

Let's you have models and signals like this:

```python
# events.models.py

class Event(models.Model):
    name = models.CharField(max_length=255)
    starts_at = models.DateTimeField()
    duration = models.DurationField(default=timedelta(hours=1))


@receiver(pre_save, sender=Event)
def event_pre_save(sender, instance, **kwargs):
    if not instance.id:
        return

    old_instance = Event.objects.get(id=instance.id)
    if old_instance.starts_at != instance.starts_at or old_instance.duration != ␣
→instance.duration:
        event_tasks.resync_event_calendars.delay(event_id=instance.id)


class Enrollment(models.Model):

    PARTICIPANT = 'participant'
    SPEETCHER = 'speetcher'
    ORGANIZER = 'organizer'
    CO_ORGANIZER = 'co_organizer'

    ROLE_CHOICES = [
        (PARTICIPANT, 'Participant'),
        (SPEETCHER, 'Speetcher'),
        (ORGANIZER, 'Organizer'),
        (CO_ORGANIZER, 'Co-Organizer'),
    ]

    role = models.CharField(ax_length=255, choices=ROLE_CHOICES)
```

(continues on next page)

```
    user = models.ForeignKey("users.User", models.CASCADE, related_name="enrollments")
    event = models.ForeignKey("events.Event", models.CASCADE, related_name="enrollments")


@receiver(post_delete, sender=Enrollment)
def enrollment_post_save(sender, instance, **kwargs):
    event_tasks.remove_enrollment_calendar.delay(enrollment_id=instance.id)
```

Transforming it to a Watcher:

```
# events.watchers.py

from django_watchers.mixins import UpdateWatcherMixin, DeleteWatcherMixin


EventWatcher(UpdateWatcherMixin):
    @classmethod
    def pre_update(cls, target, meta_params):
        source = meta_params.get('source')

        if source == 'queryset':
            operation_params = meta_params.get('operation_params')
            resync = 'starts_at' in operation_params or 'duration' in operation_params
        else:
            old_instance = target.first()
            instance = meta_params.get('instance_ref')
            resync = old_instance.starts_at != instance.starts_at or old_instance.
→duration !=  instance.duration:

        if resync:
            event_tasks.resync_event_calendars.delay(event_ids=target.values_list('id'))


EnrollmentWatcher(DeleteWatcherMixin):
    @classmethod
    def post_delete(cls, target):
        event_tasks.remove_enrollment_calendar.delay(enrollment_ids=[enrollment.id for
→enrollment in target])

# events.models.py

from django_watcher.decorators import watched

from .watchers import EventWatcher, EnrollmentWatcher


@watched(EventWatcher)
class Event(models.Model):
    name = models.CharField(max_length=255)
    starts_at = models.DateTimeField()
    duration = models.DurationField(default=timedelta(hours=1))
```

```python
@watched(EnrollmentWatcher)
class Enrollment(models.Model):

    PARTICIPANT = 'participant'
    SPEETCHER = 'speetcher'
    ORGANIZER = 'organizer'
    CO_ORGANIZER = 'co_organizer'

    ROLE_CHOICES = [
        (PARTICIPANT, 'Participant'),
        (SPEETCHER, 'Speetcher'),
        (ORGANIZER, 'Organizer'),
        (CO_ORGANIZER, 'Co-Organizer'),
    ]

    role = models.CharField(ax_length=255, choices=ROLE_CHOICES)

    user = models.ForeignKey("users.User", models.CASCADE, related_name="enrollments")
    event = models.ForeignKey("events.Event", models.CASCADE, related_name="enrollments")
```

INDICES AND TABLES

- genindex
- modindex
- search