2.8. Signals

New in version 0.5.

Note

Signal support is provided by the excellent blinker library. If you wish to enable signal support this library must be installed, though it is not required for MongoEngine to function.

2.8.1. Overview

Signals are found within the *mongoengine.signals* module. Unless specified signals receive no additional arguments beyond the *sender* class and *document* instance. Post-signals are only called if there were no exceptions raised during the processing of their related function.

Available signals include:

pre_init

Called during the creation of a new `Document` or `EmbeddedDocument` instance, after the constructor arguments have been collected but before any additional processing has been done to them. (I.e. assignment of default values.) Handlers for this signal are passed the dictionary of arguments using the *values* keyword argument and may modify this dictionary prior to returning.

post_init

Called after all processing of a new `Document` or `EmbeddedDocument` instance has been completed.

pre_save

Called within `save()` prior to performing any actions.

pre_save_post_validation

Called within `save()` after validation has taken place but before saving.

post_save

Called within `save()` after most actions (validation, insert/update, and cascades, but not clearing dirty flags) have completed successfully. Passed the additional boolean keyword argument *created* to indicate if the save was an insert or an update.

pre_delete

Called within `delete()` prior to attempting the delete operation.

post_delete

Called within `delete()` upon successful deletion of the record.

pre_bulk_insert

Called after validation of the documents to insert, but prior to any data being written. In this case, the *document* argument is replaced by a *documents* argument representing the list of documents being inserted.

post_bulk_insert

Called after a successful bulk insert operation. As per *pre_bulk_insert*, the *document* argument is omitted and replaced with a *documents* argument. An additional boolean argument, *loaded*, identifies the contents of *documents* as either `Document` instances when *True* or simply a list of primary key values for the inserted records if *False*.

2.8.2. Attaching Events

After writing a handler function like the following:

```
import logging
from datetime import datetime

from mongoengine import *
from mongoengine import signals

def update_modified(sender, document):
    document.modified = datetime.utcnow()
```

You attach the event handler to your `Document` or `EmbeddedDocument` subclass:

```
class Record(Document):
    modified = DateTimeField()

signals.pre_save.connect(update_modified)
```

While this is not the most elaborate document model, it does demonstrate the concepts involved. As a more complete demonstration you can also define your handlers within your subclass:

```
class Author(Document):
    name = StringField()

    @classmethod
    def pre_save(cls, sender, document, **kwargs):
        logging.debug("Pre Save: %s" % document.name)

    @classmethod
    def post_save(cls, sender, document, **kwargs):
        logging.debug("Post Save: %s" % document.name)
        if 'created' in kwargs:
            if kwargs['created']:
                logging.debug("Created")
            else:
                logging.debug("Updated")

signals.pre_save.connect(Author.pre_save, sender=Author)
signals.post_save.connect(Author.post_save, sender=Author)
```

Warning

Note that EmbeddedDocument only supports pre/post_init signals. pre/post_save, etc should be attached to Document's class only. Attaching pre_save to an EmbeddedDocument is ignored silently.

Finally, you can also use this small decorator to quickly create a number of signals and attach them to your `Document` or `EmbeddedDocument` subclasses as class decorators:

```
def handler(event):
    """Signal decorator to allow use of callback functions as class decorators."""

    def decorator(fn):
        def apply(cls):
            event.connect(fn, sender=cls)
            return cls

        fn.apply = apply
        return fn

    return decorator
```

Using the first example of updating a modification time the code is now much cleaner looking while still allowing manual execution of the callback:

```
@handler(signals.pre_save)
def update_modified(sender, document):
    document.modified = datetime.utcnow()

@update_modified.apply
class Record(Document):
    modified = DateTimeField()
```