djangocms-moderation Documentation

Fidelity International

May 16, 2024

USER DOCUMENTATION:

CHAPTER

ONE

OVERVIEW

Moderation provides an approval workflow mechanism for organisations who need to ensure that content is approved before it is published. It is designed to extend and compliment the Versioning addon and has that as a dependency.

The general idea is that a draft version can be submitted for moderation. This involves adding that draft to a *Moderation Collection*, which can be thought of as a chapter, edition or batch of content that aims to all be published simultaneously. Various drafts can be added to the same *Moderation Collection*.

Drafts within the *Moderation Collection* can then be approved rejected by various parties according to role`s defined within the :ref:`workflow assigned to the *Moderation Collection*.

Once one or more items within the *Moderation Collection* have been approved, the *Moderation Collection* owner is able to select those items and publish them.

Comments can also be added to any of these entities *Moderation Collection*, *Moderation Request*, *Moderation Request Action*.

The Moderation addon makes use of the App Registry features provided as part of DjangoCMS 4.0 in order to register models for various content-types to be moderated. Thus it is possible to configure your CMS project so that some content-types are moderated whilst others are not. Any such model registered with Moderation must also be registered with Versioning.

CHAPTER

TWO

MODERATION COLLECTION

A Moderation Collection is primarily intended as a way of being able to group draft content versions together for: a) review and b) publishing

The rules for adding items to a Collection, removing items from a Collection and the actions that can be taken on items the Collection may vary by *Workflow*.

Publishing is a *djangocms-versioning* feature, thus *djangocms-moderation* depends on and extends the functionality made available by the Versioning addon.

Collections are stateful. The available states are:

- Collecting
- In review
- Archived
- Cancelled

Drafts can only be added to a Collection during the *Collecting* phase (see *Moderation Review Lock*)

2.1 Buttons

2.1.1 Add Collection

Those with permissions can create new collections. The author is auto-assigned as the current user. A *Workflow* must be selected. A name must be given to the collection. If there are already items in the Collection, these will be shown on the confirmation screen.

2.1.2 Edit Collection

The selected workflow can only be changed whilst the Collection is in *collecting* state.

2.1.3 Add draft to Collection ("Submit for moderation")

The CMSToolbar (for content-types with a preview end-point) will be modified to add the "Submit for moderation" button for draft versions. Doing so allows one to select which Collection to add the draft to.

2.2 Actions

2.2.1 Submit for review

Moves the collection state from *collecting* to *in review*. Only available whilst collection phase is *collecting*. Sends out notifications to the selected reviewers.

2.2.2 Cancel collection

Changes the collection state to *cancelled*.

2.2.3 Archive collection

Changes the collection state to *archived*. Only available if every *Moderation Request* in the Collection has been approved.

2.3 States

2.3.1 Collecting

Once a Collection is created it is in this initial state, which allows draft versions to be added to a Collection by its author only.

2.3.2 In Review

A collection is submitted for review by the collection author. Reviewers (see *Role*) are then able to act on versions in that Collection. Such actions are tracked as a *Moderation Request Action*. Drafts cannot be added to a Moderation Collection while that collection is *in review* and drafts that are already in the Collection have limited editing permissions (see *Moderation Review Lock*).

2.3.3 Archived

Once all items in a collection have either been removed or approved, the collection becomes archiveable. Archiving a collection is a manual process. The effect of archiving a collection is that it to facilitate list filtering. Archived collections cannot be modified in any way.

2.3.4 Cancelled

A collection can also be flagged as cancelled. This is similar to Archived except that it can be done at any stage.

2.4 Bulk Actions

These will appear in the Collection's action drop-down for each content-type registered with Moderation.

2.4.1 Remove from collection

Removes a draft from the collection.

2.4.2 Approve

Flags a draft as being ready for publishing.

2.4.3 Submit for rework (reject)

Flags a draft as being in need of further editing

2.4.4 Submit for review

Useful for items that have been flagged for rework - resubmits them for review, sending out notifications again.

CHAPTER

THREE

COMMENT

Comments may be added to various moderation entities:

- *Moderation Collection*
- *Moderation Request*
- *Moderation Request Action*

CHAPTER

FOUR

MODERATION REVIEW LOCK

As soon as a *Moderation Collection* status becomes in review then its drafts are automatically locked, in the sense that their content can no longer be edited (not at all, not by anyone, not even the collection author). Also once a collection is in Review then content versions cannot be added to the collection. This means that once you've clicked "Submit for review":

- Collection Lock: New drafts cannot be added to the *Moderation Collection*
- Version Lock: Drafts in the *Moderation Collection* cannot be edited unless rejected

Once a version is published the Moderation Version Lock is removed automatically.

CHAPTER

FIVE

MODERATION REQUEST

While the aim of a *Moderation Collection* is to group draft Version objects together. This is achieved via an intermediary model *Moderation Request* which allows meta-data such as approvals, comments, dates and actors to be associated with each draft as it goes through moderation.

Conceptually this entity can be thought of as a "request to publish" for a particular draft version. Thus the request tracks the meta-data associated with the moderation process for a particular draft.

Moderation Requests should not be confused with the standard Django request entity.

5.1 States

Moderation Requests do not track state directly, however they contain one or more instances of the *Moderation Request Action* entity, which is stateful and the Moderation Request state can thus be inferred from its *moderation request actions*. These latter also link to a draft version, which also has states. The inferred states for a request are:

5.1.1 Ready for review

Waiting for approval / rejection. I.e. (@TODO: ???)

5.1.2 Ready for rework

Waiting for editing and resubmission for review. I.e. contains one or more actions of the *rejected* state.

5.1.3 Approved

Ready to be published. I.e. contains only actions of the 'approved' state.

5.1.4 Published

I.e. refers to a Published version - no longer a draft.

CHAPTER

SIX

MODERATION REQUEST ACTION

Each *Workflow Step* must be assigned to a Role. This allows the moderation system to know the set of valid *Reviewers* for that step. Once that *Reviewer* acts on a given *Moderation Request*, their action is recorded as a *Moderation Request Action*.

CHAPTER

SEVEN

NOTIFICATIONS

7.1 Email notifications

Configure email notifications to fail silently by setting: `EMAIL_NOTIFICATIONS_FAIL_SILENTLY=True`

CHAPTER

EIGHT

ROLE

Understanding the Role model can be a bit tricky because it blurs the lines between the CMS permissions system and the custom permission system implemented by Moderation. So let's break this down a bit...

Firstly from a CMS permissions perspective - you can define whatever Groups you like to the various standard and customer model permissions for Moderation (e.g. *add_moderationcollection*). However, the recommendation is the following groups: Editor, Publisher and Reviewer. The Reviewer should have permission to view the *Moderation Collection* only (and should generally have very limited access to the CMS - no edit / create permissions for content-types in general. The Editor should have rights to view and edit a *Moderation Collection*. The Publisher should have rights to create, edit, cancel and view a *Moderation Collection*.

For the purpose of this explanation - *Role* (capitalised) and *role* (lowercase) - *Role* refers to the model, whereas *role* refers to the word "role" in the normal broad application of the English term.

Moderation has internal permissions logic which does not involve CMS permissions but rather which defines two *roles*, each which will have differing access to parts of the Moderation UI/UX. These *roles* are *Collection author* and *Reviewer*.

Collection author is defined simply by the *author* fk link to a user on the ModerationCollection model instance. It is always a single User. For this user, the following bulk actions will be enabled in the Moderation's change_list view , namely *cancel collection*, *publish* and *remove*.

Reviewer is a more nebulous concept. A ModerationCollection may have a number of *Reviewers*. The *Workflow* has one or more *Workflow Step* each which have a single Role assigned to it. The Role links to either a single User or a Group of Users. This dynamically determines a set of users that are valid *Reviewers* for a given *Moderation Request* at a given *Workflow Step*. Once a valid *Reviewer* acts on a given *Moderation Request*, their action is recorded as a *Moderation Request Action*. A *Reviewer* has access to a compliment of bulk actions - specifically allowing a *Reviewer* to *accept* or *reject* a draft version.

A User may be both a *Reviewer* and a *Collection author* for a given *Moderation Collection*.

Thus, the Role model defines *the person/s who is responsible for reviewing a particular step of the workflow*. I.e. it defines the users that may review a *draft* version for a given *Moderation Request* for a given Collection.

In summary...

8.1 Reviewer

The Reviewer is responsible for approving / rejecting items in the collection and making comments. They have access to the *Approve* and *Submit for rework* *Moderation Collection* bulk actions.

8.2 Collection author

The collection author is responsible for creating, editing and (usually) publishing the collection. They have access to the *Submit for review* and *Publish* *Moderation Collection* bulk actions, as well as the various *Moderation Collection* buttons.

CHAPTER

NINE

REFERENCES

Moderation offers integration with the *djangocms-references* addon (djangocms-references). If this is enabled then on the confirmation screen when publishing, all content records that will be affected by the publish action will be listed for review before confirmation.

CHAPTER

TEN

WORKFLOW

The moderation workflow system is designed to be flexible enough that it can cater for a multi-step approval workflow. E.g. if your organisation has *marketing*, *legal* and *compliance* departments who each need to approve every request, you would add 3 steps to your workflow, each assigned to a different Role. The workflow would require each step to be approved before the *Moderation Request* could be published.

Workflows are designed to be extensible and customisable for developers.

CHAPTER

ELEVEN

WORKFLOW STEP

Each *Workflow* has at least one workflowstep. These are steps of review the moderation process needs to go through. For example, if an organisation had several different departments, each needing to approve each *Moderation Request*, then:

1. Each of those departments would be set up as a user Group
2. A workflow would be created to represent this
3. A step would be added for each department and the Group for that department would be assigned as the *Role* for that workflow step.

As a result, the draft could not be published without first being approved at each step in the *Workflow*

CHAPTER

TWELVE

INTRODUCTION

12.1 Integrating Moderation

Moderation depends on Versioning to be installed. The content-type models that should be moderated need to be registered. This can be done in *cms_config.py* file:

```
# blog/cms_config.py
from collections import OrderedDict
from cms.app_base import CMSAppConfig
from djangocms_versioning.datastructures import VersionableItem, default_copy
from .models import PostContent

def get_preview_url(obj):
    # generate url as required
    return obj.get_absolute_url()

def get_blog_additional_changelist_action(obj):
    return "Custom moderation action"

def get_blog_additional_changelist_field(obj):
    return "Custom moderation field"
get_poll_additional_changelist_field.short_description = "Custom Field"

class BlogCMSConfig(CMSAppConfig):
    djangocms_versioning_enabled = True  # -- 1
    djangocms_moderation_enabled = True  # -- 2
    versioning = [
        VersionableItem(   # -- 3
            content_model=PostContent,
            grouper_field_name='post',
            copy_function=default_copy,
            preview_url=get_preview_url,
        ),
    ]
    moderated_models = [   # -- 4
      PostContent,
    ]
    moderation_request_changelist_actions = [   # -- 5
        get_blog_additional_changelist_action
    ]
    moderation_request_changelist_fields = [   # -- 6
```

(continues on next page)

(continued from previous page)

```
        get_blog_additional_changelist_field
    ]
```

1. This must be set to True for Versioning to read app's CMS config.
2. This must be set to True for Moderation to read app's CMS config.
3. *versioning* attribute takes a list of *VersionableItem* objects. See *djangocms_versioning* documentation for details.
4. *moderated_models* attribute takes a list of moderatable model objects.
5. *moderation_request_changelist_actions* attribute takes a list of actions that are added to the action field in the Moderation Request Changelist admin view
6. *moderation_request_changelist_fields* attribute takes a list of admin fields that are added to the display list in the Moderation Request Changelist admin view

CHAPTER

THIRTEEN

INTERNALS

13.1 Admin Moderation

13.1.1 monkeypatch.py

Moderation monkeypatches some of Versioning's admin pages. *get_state_actions*:- this adds a "Submit for moderation" link next to draft versions in the Version table for a given content-type.

It also adds some checks to the *checks-framework* checks registered in Versioning, to prevent certain Versioning functions at certain stages of moderation.

13.1.2 admin.py

Aside from the usual, there are a number of bulk-action confirmation views that are generated here:- *delete_selected*, *approve*, *rework*, *publish*, *resubmit*. These each provide additional information whilst facilitating confirmation and the *admin_actions.py* redirect to these views.

The available bulk actions are also controlled by the internal permissions system within Moderation which links Users or Groups to Roles. Each *Moderation Request Action* within a *Workflow* has a single *Role* assigned to it. Each *Moderation Collection* has a *Workflow*. The result is that not all bulk-actions will be available to every user and some will appear only when the *Moderation Request* is in a particular inferred state.

13.1.3 cms_toolbars.py

Replaces the VersioningToolbar object with the ModerationToolbar object in order to show Versioning-related buttons at the correct part of the *Workflow*.

13.2 Tree Admin

13.2.1 Add Children To Collection

The CollectionItemsView class from *views.py* provides a way, when adding a Page to a *Moderation Collection* of also adding any drafts from other content-types that are included as plugins in any placeholders on that Page.

This poses certain UI / UX challenges. The default implementation meant that these drafts would simply be added to the Collection as part of the list of content objects in that collection. However there are several problems with this:

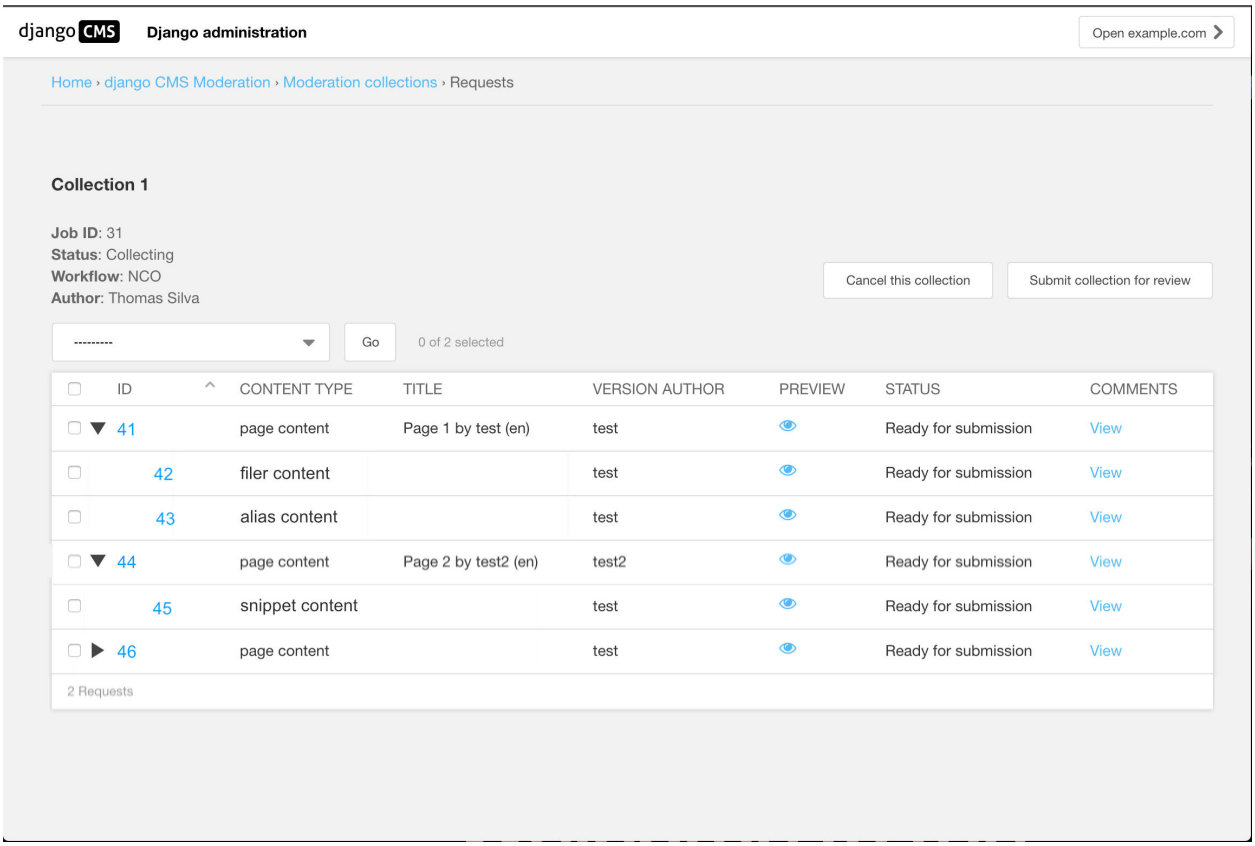
1. There's no obvious link between the listed items and the pages that contain them and they would thus be indistinguishable from content objects that may have been added without being part of any page.

2. Reviewers aren't necessarily interested in all of that detail. They would want to simply be able to moderate the page, including all of it's content. So all of the detail may be undigestible for them.

The solution for these problems that has been implemented is to rework the admin as a TreeBeard admin view. This creates a tree for each collection which would map the relationships between items that are added as part of a page and that page.

The outcome of this is that a given content object may be appear repeatedly within the tree, even though it is only added to the Collection once. e.g. If a *child* that is added to a *Moderation Collection* as part of both *parent1* and *parent2* and individually via that *child* content-type's admin, it would then appear three times within the tree. Removing the *child* from the *Moderation Collection* would remove it from all/any parts of the tree it was part of.

The tree structure is shown visually using tabbed spacing to indicate nesting.



The *ModerationRequestTreeAdmin* class in *admin.py* replaces the original *ModerationRequestAdmin*, providing TreeBeard integration as described above.

The *delete_selected_view* function of that class ensures that removing an item from the *Moderation Collection* updates the tree correctly via means of a recursive function, *_traverse_moderation_nodes*.

CHAPTER

FOURTEEN

MANAGEMENT COMMANDS

The commands made available to developers should be used with caution, be sure that you know what you are doing.

14.1 moderation_fix_states

This command is to be used only when a version becomes un-editable due to inconsistencies with states that are controlled by moderation. It has been observed that the state can end up inconsistent in very rare scenarios. A *ModerationRequest* object should never have *is_active=True* when the item has been successfully published. The following states can cause a version to be locked from editing:

- *ModerationRequest.is_active=True*
- *ModerationRequest.version.state=published*
- *ModerationRequest.collection.state=Archived*

In this scenario a new Draft object cannot be created from the Published object due to version checks.

The command will first analyse and list any *ModerationRequest* objects that are in a broken / inconsistent state.

The fix will correctly set the is_active flag leaving the correct states:

- *ModerationRequest.is_active=False*
- *ModerationRequest.version.state=published*
- *ModerationRequest.collection.state=Archived*

14.1.1 Usage

To first run an analysis on whether any *ModerationRequest* objects have a broken / inconsistent state.

```
python manage.py moderation_fix_states
```

To execute and resolve any state inconsistencies, you can run the command with the *–perform-fix* flag set.

```
python manage.py moderation_fix_states --perform-fix
```

CHAPTER

FIFTEEN

SIGNALS

The *djangocms_moderation.signals* module defines a set of signals sent by Django CMS Moderation.

15.1 submitted_for_review

djangocms_moderation.**submitted_for_review**

Sent when a *Moderation Collection* is submitted for review, or when select *Moderation Requests* are resubmitted after being rejected.

Arguments sent with this signal:

sender
: djangocms_moderation.models.ModerationCollection class

collection
: A djangocms_moderation.models.ModerationCollection instance which was submitted for review

moderation_requests
: A list of djangocms_moderation.models.ModerationRequest instances which were submitted for review

---

Note: It's possible for this list to contain only some of the requests belonging to the collection being moderated, because only some of the requests required rework.

This case is only possible for resubmitting after a rework.

---

user
: A django.contrib.auth.models.User instance which triggered the submission

rework
: A bool value specifying if this was the first time the collection was submitted, or a rework of its moderation requests

15.2 published

djangocms_moderation.**published**

Sent when a *Moderation Collection* is being published

Arguments sent with this signal:

sender
djangocms_moderation.models.ModerationCollection class

collection
A djangocms_moderation.models.ModerationCollection instance which was submitted to be published.

moderator
A django.contrib.auth.models.User associated with the collection which is the moderator of the collection.

moderation_requests
A list of djangocms_moderation.models.ModerationRequest instances which were published.

> **Note:** It's possible for this list to contain only some of the requests belonging to the collection being moderated, because only some of the requests were published.

workflow
An instance of djangocms_moderation.models.Workflow which was used for this collection.

15.3 unpublished

djangocms_moderation.**unpublished**

Sent when a *Moderation Collection* is being unpublished

Arguments sent with this signal:

sender
djangocms_moderation.models.ModerationCollection class

collection
A djangocms_moderation.models.ModerationCollection instance which was submitted to be unpublished.

moderator
A django.contrib.auth.models.User associated with the collection which is the moderator of the collection.

moderation_requests
A list of djangocms_moderation.models.ModerationRequest instances which were unpublished.

> **Note:** It's possible for this list to contain only some of the requests belonging to the collection being moderated, because only some of the requests were unpublished.

workflow
An instance of djangocms_moderation.models.Workflow which was used for this collection.

15.4 How to use the moderation publish signal for a collection

The CMS used to provide page publish and unpublish signals which have since been removed in DjangoCMS 4.0. You can instead use the signals provided above to replace these.

Djangocms-moderation provides a way to take further actions once a collection has been published. The *published* event is the last event executed for a moderation.

```
from django.dispatch import receiver

from cms.models import PageContent

from djangocm_moderation.signals import published


@receiver(published)
def do_something_on_publish_event(*args, **kwargs):
    # all keyword arguments can be found in kwargs
    # pass
```

CHAPTER

SIXTEEN

GLOSSARY

Moderation
: A process by which a draft version (see docs for djangocms-versioning) goes through an approval process before it can be published.

Moderation Collection
: A collection (or batch) of drafts ready for moderation.

Moderation Request
: Each draft in a *Moderation Collection* is wrapped as a *Moderation Request* in order to associate additional *Workflow* -related data with that draft. Each request may also have comments added to it and may send out notifications

Workflow
: Each *Moderation Collection* is associated with a *Workflow*. The workflow determines through what steps the moderation process needs to go and may provide a differing moderation UX for each Workflow.

WorkflowStep
: Each *Workflow* has at least one *Workflow Step*.

Moderation Request Action
: Each *Moderation Request* will have a number of actions associated with it. The number of these is defined as part of the *Workflow*. A *Moderation Request Action* is the action taken by an actor who is part of the moderation process. E.g. "mark as approved", "request rework", "publish".

Role
: Each *Moderation Request Action* step in a *Workflow* is associated with a Role. The Role consists either of a single User or a single Group. The users associated with that Role are required to act at that stage of the *Workflow*.

CHAPTER

SEVENTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX