# "djangocms-versioning" Documentation

Fidelity International

May 04, 2024

# QUICK START:


**1 Introduction** **1**
- 1.1 Version states . . . 1
- 1.2 Effect on the model's manager . . . 2

**2 Integrating Versioning** **3**
- 2.1 Change the model structure . . . 3
- 2.2 Register the model for versioning . . . 4
- 2.3 Accessing content model objects . . . 5
- 2.4 Implement a custom copy function . . . 6
- 2.5 Adding Versioning Entries to a Content Model Admin . . . 8
- 2.6 Adding status indicators to a versioned content model . . . 9
- 2.7 Adding Status Indicators *and* Versioning Entries to a versioned content model . . . 10
- 2.8 Adding Versioning Entries to a Grouper Model Admin . . . 10
- 2.9 Summary admin options . . . 11
- 2.10 Additional/advanced configuration . . . 11

**3 Permissions in djangocms-versioning** **13**
- 3.1 Understanding Permissions . . . 13
- 3.2 Conclusion . . . 14

**4 Locking versions** **15**
- 4.1 Explanation . . . 15
- 4.2 Activation . . . 15
- 4.3 Email notifications . . . 15

**5 Advanced configuration** **17**
- 5.1 Overriding how versioning handles core cms models . . . 17
- 5.2 Adding to the context of versioning admin views . . . 17
- 5.3 Additional options on the VersionableItem class . . . 18

**6 Signals** **21**
- 6.1 How to use the version publish and un-publish signal for a CMS Page . . . 21
- 6.2 Handling the effect of a (un-)publish to other items via signals . . . 22

**7 Customizing the Version Table Admin View** **23**
- 7.1 Changing breadcrumbs . . . 23
- 7.2 Changing the preview url . . . 23
- 7.3 Adding additional UI filters . . . 23

**8 Management command** **25**
- 8.1 create_versions . . . 25

**9 Settings for djangocms Versioning** **27**

**10 The Admin with Versioning** **29**
10.1 The content model admin . . . . . 29
10.2 The Version model admin . . . . . 29

**11 2.0.0 release notes (unreleased)** **31**
11.1 Django and Python compatibility . . . . . 31
11.2 Features . . . . . 31
11.3 Backwards incompatible changes in 2.0.0 . . . . . 32
11.4 Miscellaneous . . . . . 33
11.5 Bug fixes . . . . . 33

**12 Glossary** **35**

**Index** **37**

CHAPTER

ONE

# INTRODUCTION

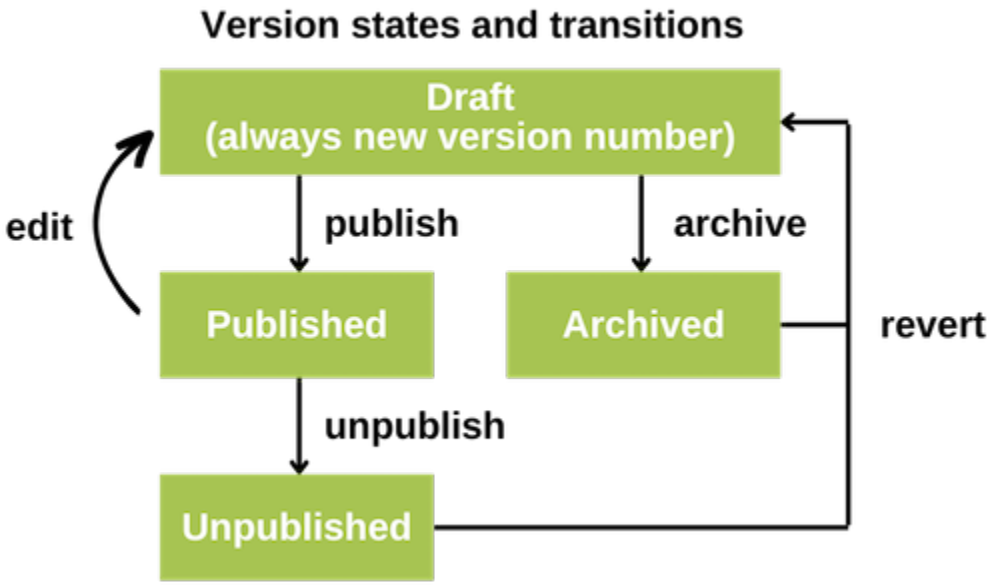
djangocms-versioning is a general purpose package that manages versions for page contents and other models within four categories: **published**, **draft**, **unpublished**, or **archived**, called "version states".

## 1.1 Version states

Each versioned object carries a version number, creation date, modification date, a reference to the user who created the version, and **version state**. The states are:

- **draft**: This is the version which currently can be edited. Only draft versions can be edited and only one draft version per language is allowed. Changes made to draft pages are not visible to the public.
- **published**: This is the version currently visible on the website to the public. Only one version per language can be public. It cannot be changed. If it needs to be changed a new draft is created based on a published page and the published page stays unchanged.
- **unpublished**: This is a version which was published at one time but now is not visible to the public any more. There can be many unpublished versions.
- **archived**: This is a version which has not been published and therefore has never been visible to the public. It represents a state which is intended to be used for later work (by reverting it to a draft state).

Each new draft version will generate a new version number.




When an object is published, it changes state to **published** and thereby becomes publicly visible. All other version states are invisible to the public.

## 1.2 Effect on the model's manager

When handling versioned models in code, you'll generally only "see" published objects:

```
MyModel.objects.filter(language="en")   # get all published English objects of MyModel
```

will return a queryset with published objects only. This is to ensure that no draft or unpublished versions leak or become visible to the public.

Since often draft contents are the ones you interact with in the admin interface, or in draft mode in the CMS frontend, djangocms-versioning introduces an additional model manager for the versioned models **which may only be used on admin sites and admin forms**:

```
MyModel.admin_manager.filter(language="en")
```

will retrieve all objects of all versions. Alternativley, to get the current draft version you can to filter the `Version` object:

```
from djangocms_versioning.constants import DRAFT

MyModel.admin_manager.filter(language="en", versions__status==DRAFT)
```

Finally, there are instance where you want to access the "current" version of a page. This is either the current draft version or - there is no draft - the published version. You can easily achieve this by using:

```
MyModel.admin_manager.filter(language="en").current_content()
```

CHAPTER

TWO

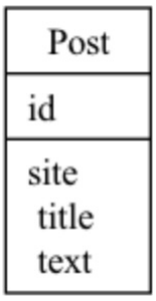
# INTEGRATING VERSIONING

Let's say we have an existing *blog* application. To make the *blog* app work with versioning, you would need to take the following steps:

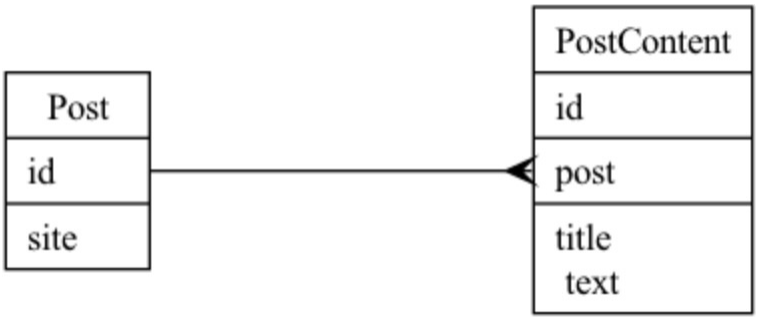
1. Change the model structure.
2. Register the *Post* model for versioning
3. (optionally as needed) Implement a custom copy function
4. (optionally as needed) Additional/advanced configuration

## 2.1 Change the model structure

Assuming that our *blog* app has one db table:




This would have to change to a db structure like this:




Or in python code, *models.py* would need to change from:

```
# blog/models.py
from django.db import models
```

(continues on next page)

(continued from previous page)

```
from django.contrib.sites.models import Site


class Post(models.Model):
    site = models.ForeignKey(Site, on_delete=models.CASCADE)
    title = models.CharField(max_length=100)
    text = models.TextField()
```

To:

```
# blog/models.py
from django.db import models
from django.contrib.sites.models import Site


class Post(models.Model):
    site = models.ForeignKey(Site, on_delete=models.CASCADE)


class PostContent(models.Model):
    post = models.ForeignKey(Post, on_delete=models.CASCADE)
    title = models.CharField(max_length=100)
    text = models.TextField()
```

*Post* becomes a *grouper model* and *PostContent* becomes a *content model*.

Keep in mind that it's not necessary to name the *content model* *PostContent*, it's just a naming convention. You could name the *content model* *Post* and have *PostGrouper* as the name of *grouper model* or come up with completely different naming.

Once the integration with versioning is complete, versioning will treat *Post* as the object being versioned and *PostContent* as a place to store data specific to each version. So every *Post* object will potentially have many *PostContent* objects referring to it via the *post* foreign key field. The states of the *PostContent* versions (whether they're published, drafts etc.) are represented in a separate model called *Version*, which has what is effectively a one2one relationship with *PostContent*.

Deciding which fields should be in the *content model* and which in the *grouper model* depends on which data should be versioned and which should not. In this example we're assuming that which site a blog post appears on cannot be changed, therefore we would not want to version it (it never changes so there's nothing to version!). But if your project assumes that the site can be changed and those changes should be versioned, we would put that field in the *PostContent* model.

## 2.2 Register the model for versioning

Now we need to make versioning aware of these models. So we have to register them in the *cms_config.py* file. A very basic configuration would look like this:

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

(continues on next page)

(continued from previous page)

```
class BlogCMSConfig(CMSAppConfig):
    djangocms_versioning_enabled = True
    versioning = [
        VersionableItem(
            content_model=PostContent,
            grouper_field_name='post',
            copy_function=default_copy,
        ),
    ]
```

In this configuration we must specify the *content model* (*PostContent*), the name of the field that is a foreign key to the *grouper model* (*post*) and a *copy function*. For simple model structures, the *default_copy* function which we have used is sufficient, but in many cases you might need to write your own custom *copy function* (more on that below).

Once a model is registered for versioning its behaviour changes:

1. It's default manager (`Model.objects`) only sees published versions of the model. See :term:`content model`.
2. It's `Model.objects.create` method now will not only create the *content model* but also a corresponding `Version` model. Since the `Version` model requires a `User` object to track who created which version the correct way of creating a versioned *content model* is:

   ```
   Model.objects.with_user(request.user).create(...)
   ```

   In certain situations, e.g., when implementing a *copy function*, this is not desirable. Use `Model._original_manager.create(...)` in such situations.

---

**Note:** If you want to allow using your models with and without versioning enabled we suggest to add dummy manager to your model that will swallow the `with_user()` syntax. This way you can always create objects with:

```
class ModelManager(models.Manager):
    def with_user(self, user):
        return self

class MyModel(models.Model):
    objects = ModelManager()

    ...
```

---

For more details on how *cms_config.py* integration works please check the documentation for django-cms>=4.0.

## 2.3 Accessing content model objects

Versioned content model objects have a customized `objects` manager which by default only creates querysets that return published versions of the content object. This will ensure that only published objects are visible to the public.

In some situations, namely when working in the admin, it is helpful to also have other content objects available, e.g. when linking to a not-yet-published object.

Versioned objects therefore also have an additional manager `admin_manager` which can access all objects. To get all draft blog posts, you can write `PostContent.admin_manager.filter(versions__state=DRAFT)`. Since the `admin_manager` has access to non-public information it should only be used inside the Django admin (hence its name).

## 2.4 Implement a custom copy function

Whilst simple model structures should be fine using the *default_copy* function, you will most likely need to implement a custom copy function if your *content model* does any of the following:

- Contains any one2one or m2m fields.
- Contains a generic foreign key.
- Contains a foreign key that relates to an object that should be considered part of the version. For example if you're versioning a poll object, you might consider the answers in the poll as part of a version. If so, you will need to copy the answer objects, not just the poll object. On the other hand if a poll has an fk to a category model, you probably wouldn't consider category as part of the version. In this case the default copy function will take care of this.
- Other models have reverse relationships to your content model and should be considered part of the version

So let's get back to our example and complicate the model structure a little. Let's say our *blog* app supports the use of polls in posts and also our posts can be categorized. Now our *blog/models.py* now looks like this:

```
# blog/models.py
from django.db import models
from django.contrib.sites.models import Site


class Category(models.Model):
    name = models.CharField(max_length=100)


class Post(models.Model):
    site = models.ForeignKey(Site, on_delete=models.CASCADE)


class PostContent(models.Model):
    post = models.ForeignKey(Post, on_delete=models.CASCADE)
    title = models.CharField(max_length=100)
    text = models.TextField()
    category = models.ForeignKey(Category, on_delete=models.CASCADE)


class Poll(models.Model):
    post_content = models.ForeignKey(PostContent, on_delete=models.CASCADE)
    name = models.CharField(max_length=100)


class Answer(models.Model):
    poll = models.ForeignKey(Poll, on_delete=models.CASCADE)
    text = models.CharField(max_length=100)
```

If we were using the *default_copy* function on this model structure, versioning wouldn't necessarily do what you expect. Let's take a scenario like this:

1. A Post object has 2 versions - *version #1* which is archived and *version #2* which is published.
2. We revert to *version #1* which creates a draft *version #3*.

3. The PostContent data in *version #3* is a copy of what was in *version #1* (the version we reverted to), but the Poll and Answer data is what was there at the time of *version #2* (the latest version).
4. We edit both the PostContent, Poll and Answer data on *version #3*.
5. The PostContent data is now different in all three versions. However, the poll data is the same in all three versions. This means that the data edit we did on *version #3* (a draft) to Poll and Answer objects is now being displayed on the published site (*version #2* is published).

This is probably not how one would want things to work in this scenario, so to fix it, we need to implement a custom *copy function* like so:

```
# blog/cms_config.py
from cms.app_base import CMSAppConfig
from djangocms_versioning.datastructures import VersionableItem
from .models import PostContent, Poll, Answer


def custom_copy(original_content):
    content_fields = {
        field.name: getattr(original_content, field.name)
        for field in PostContent._meta.fields
        # don't copy pk because we're creating a new obj
        if PostContent._meta.pk.name != field.name
    }
    new_content = PostContent._original_manager.create(**content_fields)
    original_polls = Poll.objects.filter(post_content=original_content)
    for poll in original_polls:
        poll_fields = {
            field.name: getattr(poll, field.name)
            for field in Poll._meta.fields
            # don't copy pk because we're creating a new obj
            # don't copy post_content fk because we're assigning
            # the new PostContent object to it
            if field.name not in [Poll._meta.pk.name, 'post_content']
        }
        new_poll = Poll.objects.create(post_content=new_content, **poll_fields)
        for answer in poll.answer_set.all():
            answer_fields = {
                field.name: getattr(answer, field.name)
                for field in Answer._meta.fields
                # don't copy pk because we're creating a new obj
                # don't copy poll fk because we're assigning
                # the new Poll object to it
                if field.name not in [Answer._meta.pk.name, 'poll']
            }
            Answer.objects.create(poll=new_poll, **answer_fields)
    return new_content


class BlogCMSConfig(CMSAppConfig):
    djangocms_versioning_enabled = True
    versioning = [
        VersionableItem(
            content_model=PostContent,
```

(continues on next page)

(continued from previous page)

```
            grouper_field_name='post',
            copy_function=custom_copy,
        ),
    ]
```

As you can see from the example above the *copy function* takes one param (the content object of the version we're copying) and returns the copied content object. We have customized it to create not just a new PostContent object (which *default_copy* would have done), but also new Poll and Answer objects.

---

**Note:** A custom copy method will need to use the content model's `PostContent._original_manager` to create only a content model object and not also a Version object which the `PostContent.objects` manager would have done!

---

Notice that we have not created new Category objects in this example. This is because the default behaviour actually suits Category objects fine. If the name of a category changed, we would not want to revert the whole site to use the old name of the category when reverting a PostContent object.

## 2.5 Adding Versioning Entries to a Content Model Admin

Versioning provides a number of actions and fields through the ExtendedVersionAdminMixin, these function by extending the ModelAdmin list_display to add the fields:

- author
- modified date
- versioning state
- preview action
- edit action
- version list action

```
class PostContentAdmin(ExtendedVersionAdminMixin, admin.ModelAdmin):
    list_display = "title"
```

The ExtendedVersionAdminMixin also has functionality to alter fields from other apps. By adding the admin_field_modifiers to a given apps cms_config, in the form of a dictionary of {model_name: {field: method}}, the admin for the model, will alter the field, using the method provided.

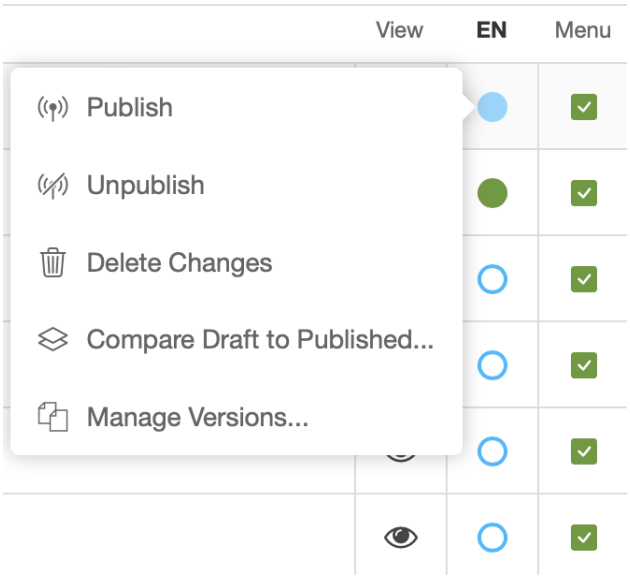
```
# cms_config.py
def post_modifier(obj, field):
    return obj.get(field) + " extra field text!"

class PostCMSConfig(CMSAppConfig):
    # Other versioning configurations...
    admin_field_modifiers = [
        {PostContent: {"title": post_modifier}},
    ]
```

Given the code sample above, "This is how we add" would be displayed as "this is how we add extra field text!" in the changelist of PostAdmin.

## 2.6 Adding status indicators to a versioned content model

djangocms-versioning provides status indicators for django CMS' content models, you may know them from the page tree in django-cms:




You can use these on your content model's changelist view admin by adding the following fixin to the model's Admin class:

```
import json
from djangocms_versioning.admin import StateIndicatorMixin


class MyContentModelAdmin(StateIndicatorMixin, admin.ModelAdmin):
    # Adds "indicator" to the list_items
     list_items = [..., "state_indicator", ...]
```

**Note:** For grouper models the mixin expects that the admin instances has properties defined for each extra grouping field, e.g., `self.language` if language is an extra grouping field. If you derive your admin class from `GrouperModelAdmin`, this behaviour is automatically observed.

Otherwise, this is typically set in the `get_changelist_instance` method, e.g., by getting the language from the request. The page tree, for example, keeps its extra grouping field (language) as a get parameter to avoid mixing language of the user interface and language that is changed.

```
def get_changelist_instance(self, request):
    """Set language property and remove language from changelist_filter_params"""
    if request.method == "GET":
        request.GET = request.GET.copy()
        for field in versionables.for_grouper(self.model).extra_grouping_fields:
            value = request.GET.pop(field, [None])[0]
            # Validation is recommended: Add clean_language etc. to your Admin class!
            if hasattr(self, f"clean_{field}"):
                value = getattr(self, f"clean_{field}")(value):
            setattr(self, field) = value
```

(continues on next page)

(continued from previous page)

```
        # Grouping field-specific cache needs to be cleared when they are changed
        self._content_cache = {}
    instance = super().get_changelist_instance(request)
    # Remove grouping fields from filters
    if request.method == "GET":
        for field in versionables.for_grouper(self.model).extra_grouping_fields:
            if field in instance.params:
                del instance.params[field]
    return instance
```

## 2.7 Adding Status Indicators *and* Versioning Entries to a versioned content model

Both mixins can be easily combined. If you want both, state indicators and the additional author, modified date, preview action, and edit action, you can simpliy use the `ExtendedIndicatorVersionAdminMixin`:

```
class MyContentModelAdmin(ExtendedIndicatorVersionAdminMixin, admin.ModelAdmin):
    ...
```

The versioning state and version list action are replaced by the status indicator and its context menu, respectively.

Add additional actions by overwriting the `self.get_list_actions()` method and calling `super()`.

## 2.8 Adding Versioning Entries to a Grouper Model Admin

Django CMS 4.1 and above provide the `GrouperModelAdmin` as to creat model admins for grouper models. To add version admin fields, use the `ExtendedGrouperVersionAdminMixin`:

```
class PostAdmin(ExtendedGrouperVersionAdminMixin, GrouperModelAdmin):
    list_display = ["title", "get_author", "get_modified_date", "get_versioning_state"]
```

`ExtendedGrouperVersionAdminMixin` also observes the admin_field_modifiers.

**Note:** Compared to the ExtendedVersionAdminMixin, the ExtendedGrouperVersionAdminMixin does not automatically add the new fields to the `list_display`.

The difference has compatibility reasons.

To also add state indicators, just add the `StateIndicatorMixin`:

```
class PostAdmin(ExtendedGrouperVersionAdminMixin, StateIndicatorMixin,␣
↪GrouperModelAdmin):
    list_display = ["title", "get_author", "get_modified_date", "state_indicator"]
```

## 2.9 Summary admin options

Table 1: Overview on versioning admin options: Grouper models

| Versioning state | Grouper Model Admin |
|---|---|
| Text, no interaction | `class GrouperAdmin(`<br>`  ExtendedGrouperVersionAdminMixin,`<br>`  GrouperModelAdmin`<br>`)`<br>`  list_display = ...` |
| Indicators, drop down menu | `class GrouperAdmin(`<br>`  ExtendedGrouperVersionAdminMixin,`<br>`  StateIndicatorMixin,`<br>`  GrouperModelAdmin`<br>`)`<br>`  list_display = ...` |

Table 2: Overview on versioning admin options: Content models

| Versioning state | **Content Model Admin** |
|---|---|
| Text, no interaction | `class ContentAdmin(`<br>`  ExtendedVersionAdminMixin,`<br>`  admin.ModelAdmin`<br>`)` |
| Indicators, drop down menu | `class ContentAdmin(`<br>`  ExtendedIndicatorVersionAdminMixin,`<br>`  admin.ModelAdmin,`<br>`)` |

## 2.10 Additional/advanced configuration

The above should be enough configuration for most cases, but versioning has a lot more configuration options. See the advanced_configuration page for details.

CHAPTER

THREE

# PERMISSIONS IN DJANGOCMS-VERSIONING

This documentation covers the permissions system introduced for publishing and unpublishing content in djangocms-versioning. This system allows for fine-grained control over who can publish and unpublish or otherwise manage versions of content.

## 3.1 Understanding Permissions

Permissions are set at the content object level, allowing for detailed access control based on the user's roles and permissions. The system checks for specific methods within the **content object**, e.g. `PageContent` to determine if a user has the necessary permissions.

- **Specific publish permission** (only for publish/unpublish action): To check if a user has the permission to publish content, the system looks for a method named `has_publish_permission` on the content object. If this method is present, it will be called to determine whether the user is allowed to publish the content.

  Example:

```
def has_publish_permission(self, user):
    if user.is_superuser:
        # Superusers typically have permission to publish
        return True
    # Custom logic to determine if the user can publish
    return user_has_permission
```

- **Change Permission** (and first fallback for `has_publish_permission`): If the content object has a method named `has_change_permission`, this method will be called to assess if a user has the permission to change the content. This is a general permission check that is not specific to publishing or unpublishing actions.

  Example:

```
def has_change_permission(self, user):
    if user.is_superuser:
        # Superusers typically have permission to publish
        return True
    # Custom logic to determine if the user can change the content
    return user_has_permission
```

- **First Fallback Placeholder Change Permission**: For content objects that involve placeholders, such as PageContent objects, a method named `has_placeholder_change_permission` is checked. This method should determine if the user has the permission to change placeholders within the content.

  Example:

```python
def has_placeholder_change_permission(self, user):
    if user.is_superuser:
        # Superusers typically have permission to publish
        return True
    # Custom logic to determine if the user can change placeholders
    return user_has_permission
```

- **Last resort Django permissions:** If none of the above methods are present on the content object, the system falls back to checking if the user has a generic Django permission to change `Version` objects. This ensures that there is always a permission check in place, even if specific methods are not implemented for the content object. By default, the Django permissions are set on a user or group level and include all instances of the content object.

---

**Note:** It is highly recommended to implement the specific permission methods on your content objects for more granular control over user actions.

---

## 3.2 Conclusion

The permissions system introduced in djangocms-versioning for publishing and unpublishing content provides a flexible and powerful way to manage access to content. By defining custom permission logic within your content objects, you can ensure that only authorized users are able to perform these actions.

CHAPTER

FOUR

# LOCKING VERSIONS

## 4.1 Explanation

The lock versions setting is intended to modify the way djangocms-versioning works in the following way:

- A version is **locked to its author** when a draft is created.
- The lock prevents editing of the draft by anyone other than the author.
- That version becomes automatically unlocked again once it is published.
- Locks can be removed by a user with the correct permission (`delete_versionlock`)
- Unlocking an item sends an email notification to the author to which it was locked.
- Manually unlocking a version does not lock it to the unlocking user, nor does it change the author.
- The Version admin view for each versioned content-type shows lock icons and offers unlock actions

## 4.2 Activation

In your project's `settings.py` add:

```
DJANGOCMS_VERSIONING_LOCK_VERSIONS = True
```

## 4.3 Email notifications

Configure email notifications to fail silently by setting:

```
EMAIL_NOTIFICATIONS_FAIL_SILENTLY = True
```

CHAPTER

FIVE

# ADVANCED CONFIGURATION

For the most important configuration options see versioning_integration. Below are additional configuration options built into versioning.

## 5.1 Overriding how versioning handles core cms models

By default django-cms models will be registered with versioning automatically. If you do not want that to happen set `VERSIONING_CMS_MODELS_ENABLED` in *settings.py* to False. You could also set that setting to False and register the django-cms models yourself with different options.

## 5.2 Adding to the context of versioning admin views

Currently versioning supports adding context variables to the unpublish confirmation view. Wider support for adding context variables is planned, but at the moment only the unpublish confirmation view is supported. This is how one would configure this in *cms_config.py*:

```
# blog/cms_config.py
from collections import OrderedDict
from cms.app_base import CMSAppConfig


def stories_about_intelligent_cats(request, version, *args, **kwargs):
    return version.content.cat_stories


 class SomeConfig(CMSAppConfig):
    djangocms_versioning_enabled = True
    versioning_add_to_confirmation_context = {
        'unpublish': OrderedDict([('cat_stories', stories_about_intelligent_cats)]),
    }
```

Any context variable added to this setting will be displayed on the unpublish confirmation page automatically, but if you wish to change where on the page it displays, you will need to override the *djangocms_versioning/admin/unpublish_confirmation.html* template.

## 5.3 Additional options on the VersionableItem class

The three mandatory attributes of *VersionableItem* are described in detail on the versioning_integration page. Below are additional options you might want to set.

### 5.3.1 preview_url

This will define the url that will be used for each version on the version list table.

```
# some_app/cms_config.py
from django.urls import reverse
from cms.app_base import CMSAppConfig
from djangocms_versioning.datastructures import VersionableItem


def get_preview_url(obj):
    return reverse('some_interesting_url', args=(obj.pk,))


 class SomeCMSConfig(CMSAppConfig):
    djangocms_versioning_enabled = True
    versioning = [
        VersionableItem(
            ....,
            preview_url=get_preview_url,
        ),
    ]
```

### 5.3.2 extra_grouping_fields

Defines one or more *extra grouping fields*. This will add a UI filter to the version list table enabling filtering by that field.

```
# some_app/cms_config.py
from django.urls import reverse
from cms.app_base import CMSAppConfig
from djangocms_versioning.datastructures import VersionableItem


 class SomeCMSConfig(CMSAppConfig):
    djangocms_versioning_enabled = True
    versioning = [
        VersionableItem(
            ....,
            extra_grouping_fields=["language"],
        ),
    ]
```

### 5.3.3 version_list_filter_lookups

Must be defined if the *extra_grouping_fields* option has been set. This will let the UI filter know what values it should allow filtering by.

```
# some_app/cms_config.py
from django.urls import reverse
from cms.app_base import CMSAppConfig
from cms.utils.i18n import get_language_tuple
from djangocms_versioning.datastructures import VersionableItem


 class SomeCMSConfig(CMSAppConfig):
    djangocms_versioning_enabled = True
    versioning = [
        VersionableItem(
            ....,
            version_list_filter_lookups={"language": get_language_tuple},
        ),
    ]
```

### 5.3.4 grouper_selector_option_label

If the version table link is specified without a grouper param, a form with a dropdown of grouper objects will display. This setting defines how the labels of those groupers will display on the dropdown.

```
# some_app/cms_config.py
from django.urls import reverse
from cms.app_base import CMSAppConfig
from djangocms_versioning.datastructures import VersionableItem


def grouper_label(obj, language):
    return "{title} ({language})".format(title=obj.title, language=language)


 class SomeCMSConfig(CMSAppConfig):
    djangocms_versioning_enabled = True
    versioning = [
        VersionableItem(
            ....,
            grouper_selector_option_label=grouper_label,
        ),
    ]
```

### 5.3.5 content_admin_mixin

Versioning modifies how the admin of the *content model* works with *VersioningAdminMixin*. But you can modify this mixin with this setting.

```
# some_app/cms_config.py
from django.urls import reverse
from cms.app_base import CMSAppConfig
from djangocms_versioning.datastructures import VersionableItem


class SomeContentAdminMixin(VersioningAdminMixin):
    # override any standard django ModelAdmin attributes and methods
    # in this class

    def has_add_permission(self, request):
        return False


 class SomeCMSConfig(CMSAppConfig):
    djangocms_versioning_enabled = True
    versioning = [
        VersionableItem(
            ....,
            content_admin_mixin=SomeContentAdminMixin,
        ),
    ]
```

### 5.3.6 extended_admin_field_modifiers

These allow for the alteration of how a field is displayed, by providing a method, when the admin menu containing it uses the ExtendedVersionAdminMixin.

This can be provided as a dictionary of {model: {field: method}}.

model - the model which is registered with an admin that inherits ExtendedVersionAdminMixin field - field to be modified method - the method used to modify the field

```
# some_app/cms_config.py
from cms.app_base import CMSAppConfig

from .models import SomeModel

def transform_text_field(obj, field):
    return obj.field + " Extra Value!"

 class SomeCMSConfig(CMSAppConfig):
    djangocms_versioning_enabled = True
    ...
    extended_admin_field_modifiers = {SomeModel: {"text": transform_text_field}}
```

CHAPTER

SIX

# SIGNALS

**Signals are fired before and after the following events which can be found in the file 'constants.py':**

- When a version is created the operation sent is 'operation_draft'
- When a version is archived the operation sent is 'operation_archive'
- When a version is published the operation emitted is 'operation_publish'
- When a version is un-published the operation emitted is 'operation_unpublish'

A token is emitted in the signals that will allow the pre and post signals to be tied together, this could be of use if multiple transactions occur at the same time, allowing a token to match the pre and post signals that belong together.

## 6.1 How to use the version publish and un-publish signal for a CMS Page

The CMS used to provide page publish and unpublish signals which have since been removed in DjangoCMS 4.0. To replicate the behaviour you can listen to changes on the cms model PageContent as shown in the example below:

```
from django.dispatch import receiver

from cms.models import PageContent

from djangocms_versioning import constants
from djangocms_versioning.signals import post_version_operation


@receiver(post_version_operation, sender=PageContent)
def do_something_on_page_publish_unpublsh(*args, **kwargs):

    if (kwargs['operation'] == constants.OPERATION_PUBLISH or
       kwargs['operation'] == constants.OPERATION_UNPUBLISH):
        # ... do something
```

## 6.2 Handling the effect of a (un-)publish to other items via signals

Events often times do not happen in isolation. A publish signal indicates a publish of an item but it also means that potentially other items are unpublished as part of the same action, also triggering unpublish signals. To be able to react accordingly, information is added to the publish signal which other items were potentially unpublished as part of this action (*unpublished*) and information is also added to the unpublish singal which other items are going to get published (*to_be_published*). This information allows you to differentiate if an item is published for the first time - because nothing is unpublished - or if it is just a new version of an existing item.

**For example, the differentiation can be benefitial if you integrate with other services like Elasticsearch and you want to update the Elasticsearch index via signals. You can get in the following situations:**

- Publish signal with no unpublished item results in a new entry in the index.
- Publish signal with at least one unpublished item results in an update of an existing entry in the index.
- Unpublish singal with no to be published items results in the removal of the entry from the index.
- Unpublish signal with a to be published item results in the update on an existing entry in the index but will be handled in the corresponding publish signal and can be ignored.

All those situations are distinct, require different information, and can be handled according to requirements.

CHAPTER

SEVEN

# CUSTOMIZING THE VERSION TABLE ADMIN VIEW

## 7.1 Changing breadcrumbs

To override how breadcrumbs look on the version table page, you can create a template with a path that follows this pattern:

`templates/admin/djangocms_versioning/<app_label>/<model>/versioning_breadcrumbs.html`

This will override the breadcrumbs for the model specified.

In addition to the context vars which are present as standard in the django admin changelist view, you can also access the following in the template:

- `{{ grouper }}` - this is the grouper instance for the versions being displayed
- `{{ latest_content }}` - this is the content instance for the latest version of those displayed
- `{{ breadcrumb_opts }}` - like `{{ opts }}` (which is present in the django admin template context as standard), but for the content model

## 7.2 Changing the preview url

You can configure versioning to use a different preview url for versions in the table. See *preview_url* for details.

## 7.3 Adding additional UI filters

If you need to be able to filter the versions by fields on the *content model* (for example by language), the best way of doing so is to use the configuration options *extra_grouping_fields* and *version_list_filter_lookups*.

CHAPTER

# EIGHT

# MANAGEMENT COMMAND

## 8.1 create_versions

`create_versions` creates `Version` objects for versioned content that does not have a version assigned. This happens if djangocms-versioning is added to content models after content already has been created. It can also be used as a recovery tool if - for whatever reason - some or all `Version` objects have not been created for a grouper.

By default, the existing content is assigned the draft status. If a draft version already exists the content will be given the archived state.

Each version is assigned a user who created the version. When this command is run, either

- the user is taken from the `DJANGOCMS_VERSIONING_DEFAULT_USER` setting which must contain the primary key (pk) of the user, or
- one of the options `--userid` or `--username`

If `DJANGOCMS_VERSIONING_DEFAULT_USER` is set it cannot be overridden by a command line option.

```
usage: manage.py create_versions [-h] [--state {draft,published,archived}]
                                 [--username USERNAME] [--userid USERID] [--dry-run]
                                 [--version] [-v {0,1,2,3}] [--settings SETTINGS]
                                 [--pythonpath PYTHONPATH] [--traceback] [--no-color]
                                 [--force-color] [--skip-checks]

Creates Version objects for versioned models lacking one. If the
DJANGOCMS_VERSIONING_DEFAULT_USER setting is not populated you will have to provide
either the --userid or --username option for each Version object needs to be assigned
to a user. If multiple content objects for a grouper model are found only the newest
(by primary key) is assigned the state, older versions are marked as "archived".

optional arguments:
  -h, --help            show this help message and exit
  --state {draft,published,archived}
                        state of newly created version object (defaults to draft)
  --username USERNAME   Username of user to create the missing Version objects
  --userid USERID       User id of user to create the missing Version objects
  --dry-run             Do not change the database
```

CHAPTER

NINE

# SETTINGS FOR DJANGOCMS VERSIONING

**DJANGOCMS_VERSIONING_ALLOW_DELETING_VERSIONS**

Defaults to `False`

This setting controls if the `source` field of a `Version` object is protected. It is protected by default which implies that Django will not allow a user to delete a version object which itself is a source for another version object. This implies that the corresponding content and grouper objects cannot be deleted either.

This is to protect the record of how different versions have come about.

If set to `True` users can delete version objects if the have the appropriate rights. Set this to `True` if you want users to be able to delete versioned objects and you do not need a full history of versions, e.g. for documentation purposes.

The latest version (which is not a source of a newer version) can always be deleted (if the user has the appropriate rights).

**DJANGOCMS_VERSIONING_ENABLE_MENU_REGISTRATION**

Defaults to `True` (for django CMS <= 4.1.0) and `False` (for django CMS > 4.1.0)

This settings specifies if djangocms-versioning should register its own versioned CMS menu. This is necessary for CMS <= 4.1.0. For CMS > 4.1.0, the django CMS core comes with a version-ready menu.

The versioned CMS menu also shows draft content in edit and preview mode.

**DJANGOCMS_VERSIONING_LOCK_VERSIONS**

Defaults to `False`

Added in version 2.0: Before version 2.0 version locking was part of a separate package.

This setting controls if draft versions are locked. If they are, only the user who created the draft can change the draft. See *Locking versions* for more details.

**DJANGOCMS_VERSIONING_USERNAME_FIELD**

Defaults to `"username"`

Adjust this settings if your custom `User` model does contain a username field which has a different name.

**DJANGOCMS_VERSIONING_DEFAULT_USER**

Defaults to `None`

Creating versions require a user. For management commands (including migrations) either a user can be provided or this default user is used. If not set and no user is specified for the management command, it will fail.

**DJANGOCMS_VERSIONING_ON_PUBLISH_REDIRECT**

Defaults to `"published"`

Added in version 2.0: Before version 2.0 the behavior was always `"versions"`.

This setting determines what happens after publication/unpublication of a content object. Three options exist:

- `"versions"`: The user will be redirected to a version overview of the current object. This is particularly useful for advanced users who need to keep a regular overview on the existing versions.
- `"published"`: The user will be redirected to the content object on the site. Its URL is determined by calling `.get_absolute_url()` on the content object. If does not have an absolute url or the object was unpublished the user is redirected to the object's preview endpoint. This is particularly useful if users only want to interact with versions if necessary.
- `"preview"`: The user will be redirected to the content object's preview endpoint.

CHAPTER

TEN

# THE ADMIN WITH VERSIONING

## 10.1 The content model admin

Versioning modifies the admin for each *content model*. This is because versioning duplicates content model records every time a new version is created (since content models hold the version data that's content type specific). Versioning therefore needs to limit the queryset in the content model admin to include only the records for the latest versions.

### 10.1.1 Extended Mixin

The ExtendedVersionAdminMixin provides fields related to versioning (such as author, state, last modified) as well as a number of actions (preview, edit and versions list) to prevent the need to re-implement on each *content model* admin. It is used in the same way as any other admin mixin.

## 10.2 The Version model admin

### 10.2.1 Proxy models

Versioning generates a proxy model of `djangocms_versioning.models.Version` for each registered *content model*. These proxy models are then registered in the admin. This allows a clear separation of the versions of each *content model* registered and means the version table can be customized for each *content model*, for example by adding custom filtering (see below).

### 10.2.2 UI filters

Versioning generates `FakeFilter` classes (inheriting from django's `admin.SimpleListFilter`) for each *extra grouping field*. The purpose of these is to make the django admin display the filter in the UI. But these `FakeFilter` classes don't actually do any filtering as this is actually handled by `VersionChangeList.get_grouping_field_filters`.

CHAPTER

ELEVEN

# 2.0.0 RELEASE NOTES (UNRELEASED)

*October 2023*

Welcome to django CMS versioning 2.0.0!

These release notes cover the new features, as well as some backwards incompatible changes you'll want to be aware of when upgrading from django CMS versioning 1.x.

## 11.1 Django and Python compatibility

django CMS supports **Django 3.2, 4.0, and 4.1**. We highly recommend and only support the latest release of each series.

It supports **Python 3.8, 3.9, 3.10, and 3.11**. As for Django we highly recommend and only support the latest release of each series.

## 11.2 Features

### 11.2.1 Status indicators in page tree

- Status indicators are shown in the page tree as of django CMS 4.1+
- For a more consistent user experience djangocms-versioning uses icons provided by django CMS 4.1+ and does not provide its own icons any more.
- If `djangocms_admin_style` is listed in the `INSTALLED_APPS` setting make sure that at least version 3.2.1 is installed. Older versions contain a bug that interferes with djangocms-versioning's icons.

### 11.2.2 Status indicators for custom versioned models

- The new `StateIndicatorMixin` allows to add state indicators to a grouper or content model's admin changelist view.
- The new `ExtendedIndicatorVersionAdminMixin` combines the `ExtendedVersionAdminMixin` and the `StateIndicatorMixin`, where the version state is replaced by the indicator and the versioning actions are part of the indicator drop down menu.

### 11.2.3 Deletion protection

By default `Version` objects which are sources for later versions are protected from deletion. This implies that neither the corresponding content object nor the grouper object can be deleted. To allow deletion of `Version` objects set `DJANGOCMS_VERSIONING_ALLOW_DELETING_VERSIONS` to `True` in the project's `settings.py`.

### 11.2.4 Version-locking

Previously a separate package, djangocms-version-locking has now been included in djangocms-versioning. Upon setting `DJANGOCMS_VERSIONING_LOCK_VERSIONS` to `True`, draft versions will be locked by default and can only be edited by the person who created the draft. This is to avoid conflicts in certain editorial situations.

## 11.3 Backwards incompatible changes in 2.0.0

### 11.3.1 Monkey patching

- Version 2.0.0 uses new configuration possibilities of django CMS 4.1+ and therefor is incompatible with versions 4.0.x
- As a result monkey patching has been removed from djangocms-versioning and is discouraged

### 11.3.2 Accessing helper functions

- Direct imports from djangocms_versioning are discouraged. They block drop-in replacements of djangocms_versioning.
- `djangocms_verisoning.helpers.remove_published_where` has been removed. Use the `admin_manager` of a verisoned content object instead.

### 11.3.3 Title Extension

As of django CMS 4.1 `TitleExtension` in `cms.extensions.models` has been renamed to `PageContentExtension` to keep a consistent language in the page models. This change is reflected in djangocms-versioning 2.0.0.

See this PR.

### 11.3.4 Icon use

Djangocms-versioning now uses icons from the core which are only available as of django CMS v4.1+.

## 11.4 Miscellaneous

- Adds compatibility for User models with no username field (see this PR): Adds the possibility to configure which field of the User model uniquely identifies the User. Default is username.

## 11.5 Bug fixes

- Adjust migrations to ensure MySql compatibility (see this PR)

CHAPTER

# TWELVE

# GLOSSARY

**version model**
: A model that stores information such as state (draft, published etc.), author, created and modified dates etc. about each version.

**content model**
: A model with a one2one relationship with the *version model*, which stores version data specific to the content type that is being versioned. It can have relationships with other models which could also store version data (for example in the case of a poll with many answers, the answers would be kept in a separate model, but would also be part of the version).

**grouper model**
: A model with a one2many relationship with the *content model*. An instance of the grouper model groups all the versions of one object. It is in effect the object being versioned. It also stores data that is not version-specific.

**extra grouping field**
: The *content model* must always have a foreign key to the *grouper model*. However, the content model can also have additional grouping fields. This is how versioning is implemented for the `cms.PageContent` model, where `PageContent.language` is defined as an extra grouping field. This supports filtering of objects by both its grouper object and its extra grouping fields in the admin and in any other implementations (in the page example, this ensures that the latest version of a German alias would not be displayed on an English page).

**copy function**
: When creating a new draft version, versioning will usually copy an existing version. By default it will copy the current published version, but when reverting to an old version, a specific unpublished or archived version will be used. A customizable copy function is used for this.

# INDEX


## C

content model, 35
copy function, 35

## D

DJANGOCMS_VERSIONING_ALLOW_DELETING_VERSIONS, 27
DJANGOCMS_VERSIONING_DEFAULT_USER, 27
DJANGOCMS_VERSIONING_ENABLE_MENU_REGISTRATION, 27
DJANGOCMS_VERSIONING_LOCK_VERSIONS, 27
DJANGOCMS_VERSIONING_ON_PUBLISH_REDIRECT, 27
DJANGOCMS_VERSIONING_USERNAME_FIELD, 27

## E

extra grouping field, 35

## G

grouper model, 35

## V

version model, 35