

README

Django_CMS

AutoFormation Django CMS

Sommaire Django CMS

- [!\[\]\(f15d3c54be60b4fd0ce1da9fb3f67256_img.jpg\) Installer django CMS](#)
 - [!\[\]\(7bf135d42c40a6430c927b2fd03d7659_img.jpg\) Prérequis](#)
 - [!\[\]\(2bcc37677ea6b96900e4d746ad300082_img.jpg\) Méthode 1 – Installation rapide avec Docker](#)
 - [!\[\]\(b62812e390f75b509ead0f847e76b4ce_img.jpg\) Méthode 2 – Installation manuelle \(sans Docker\)](#)
 - [!\[\]\(702f396a3c354a80d179cf62e75a5343_img.jpg\) Ajouter django CMS à un projet existant](#)
 - [!\[\]\(c4a9e26ffee79396bf5db4da66793f2a_img.jpg\) Templates et fichiers statiques](#)
 - [!\[\]\(05829f1dfede3fb516a7a7a32441dc04_img.jpg\) Plugins recommandés](#)
 - [!\[\]\(eacad74b03a8da6fc0adc9238f9330a0_img.jpg\) Vérification finale](#)
- [!\[\]\(ca70ba7dde2316a22dd28cb97da84711_img.jpg\) Templates & Placeholders](#)
 - [!\[\]\(382a2cf17fa1db45af06e74f73844f1f_img.jpg\) Templates](#)
 - [!\[\]\(4b753c12ef29eafd087d6c2a2eb37a94_img.jpg\) Placeholders](#)
 - [!\[\]\(4940600c19d9f5a6ca12f71ca256a02c_img.jpg\) Aliases statiques](#)
 - [!\[\]\(b27eee883a6f9eab3956f41ce94b8c6b_img.jpg\) Menus dynamiques](#)
- [!\[\]\(82a5a78ffb860d854c8968d7b3f2821c_img.jpg\) Intégration d'applications](#)
 - [!\[\]\(1d2c5ffcd9d1f10a1f957d002acac653_img.jpg\) Exemple : intégrer une app de sondages](#)
 - [!\[\]\(6267c5f5d55e7f998acf630695d96f53_img.jpg\) Améliorer l'intégration visuelle](#)
 - [!\[\]\(26921e147ab9064311149e569d874535_img.jpg\) Créer l'app d'intégration CMS](#)
- [!\[\]\(993ea680365d214ccbd13c2c3a1fc81a_img.jpg\) Créer un Plugin personnalisé](#)
 - [!\[\]\(7a7c5c49f0624306d3c0234618edd8c4_img.jpg\) Modèle du plugin](#)
 - [!\[\]\(37a3738ff2f6552570b7a35b8f3e5725_img.jpg\) Classe du plugin](#)
 - [!\[\]\(4c6c5a346262fd5bfaf4e087d6e6a61f_img.jpg\) Template du plugin](#)
 - [!\[\]\(9f8c4f920ed0d9093cea604653b450dc_img.jpg\) Tester le plugin](#)
- [!\[\]\(c345f5cfeda8fc89f45a930593114bb5_img.jpg\) Intégration via AppHook](#)
 - [!\[\]\(95d30c6ab4233c6607b82e3a89bdfe33_img.jpg\) Créer un AppHook](#)
 - [!\[\]\(54cfa8f0f85b101bebea84b65590f5b5_img.jpg\) Alternative : URLs manuelles](#)

-  [Nettoyer urls.py](#)
-  [Attacher l'AppHook à une page](#)
-  [Étendre la Toolbar](#)
 -  [Ajouter un menu Sondages](#)
 -  [Ajouter des boutons \(facultatif\)](#)
 -  [Limiter l'affichage aux pages pertinentes](#)
-  [Étendre le menu de navigation](#)
-  [Ajouter un Wizard de création](#)
-  [Guide général sur les Plugins](#)
-  [AppHooks – Rappel complet](#)
-  [Système de publication](#)
-  [Gestion multilingue](#)
-  [Internationalisation](#)
-  [Permissions et droits](#)
-  [Intégrer la recherche](#)
-  [Appareils tactiles et compatibilité](#)
-  [Schémas de couleurs \(clair/foncé\)](#)
-  [Fonctionnement du menu CMS](#)
-  [Intégration frontend](#)

Installer django CMS

Prérequis

Pas besoin d'être expert Django ou Python. Plusieurs options gratuites existent :

- **Divio Cloud** (simple, rapide, mais propriétaire)
 - **Docker** (recommandé pour tests ou déploiement local)
 - **Installation manuelle** (pour les devs qui veulent tout contrôler)
-

Méthode 1 – Installation rapide avec Docker

Étapes :

```
git clone https://github.com/django-cms/django-cms-quickstart.git
cd django-cms-quickstart
docker compose build web
docker compose up -d database_default
docker compose run web python manage.py migrate
```

```
docker compose run web python manage.py createsuperuser
docker compose up -d
```

Accède à `http://localhost:8000/admin` pour te connecter.

Créer et publier ta première page :

- Clique sur **Create** > **New Page** > Suivant
- Renseigne un titre > clique sur **Create**
- Clique ensuite sur **Publish** pour la rendre visible

💡 Pour activer la barre d'édition : ajoute `?toolbar_on` à l'URL

🔧 Méthode 2 – Installation manuelle (sans Docker)

1. Créer un environnement virtuel

```
python3 -m venv .venv
source .venv/bin/activate
pip install --upgrade pip
pip install django-cms
```

2. Générer le projet

```
djangoCMS mon_projet
cd mon_projet
python manage.py runserver
```

📄 Cela :

- Crée un projet Django préconfiguré
- Ajoute les plugins de base (ckeditor, frontend, filer...)
- Lance les migrations
- Te demande de créer un superuser
- Vérifie la config avec : `python manage.py cms check`

🔗 Ajouter django CMS à un projet Django existant

Modifie `settings.py` :

```
INSTALLED_APPS += [
    "django.contrib.sites",
    "cms",
    "menus",
    "treebeard",
```

```
"sekizai",  
]  
SITE_ID = 1
```

Optionnel mais conseillé :

```
pip install.djangocms-admin-style
```

Ajoute dans `INSTALLED_APPS` avant `django.contrib.admin` :

```
"djangocms_admin_style",
```

Configuration linguistique :

```
LANGUAGE_CODE = "fr"  
LANGUAGES = [("fr", "Français"), ("en", "Anglais")]
```

Middleware et context_processors :

Ajoute dans `MIDDLEWARE` :

```
"django.middleware.locale.LocaleMiddleware",  
"cms.middleware.user.CurrentUserMiddleware",  
"cms.middleware.page.CurrentPageMiddleware",  
"cms.middleware.toolbar.ToolbarMiddleware",  
"cms.middleware.language.LanguageCookieMiddleware",
```

Ajoute dans `TEMPLATES["OPTIONS"]["context_processors"]` :

```
"django.template.context_processors.i18n",  
"sekizai.context_processors.sekizai",  
"cms.context_processors.cms_settings",
```

Templates et fichiers statiques

Structure minimale `templates/home.html` :

```
{% load cms_tags sekizai_tags %}  
<html>  
<head>  
  <title>{% page_attribute "page_title" %}</title>  
  {% render_block "css" %}  
</head>  
<body>  
  {% cms_toolbar %}  
  {% placeholder "content" %}  
  {% render_block "js" %}
```

```
</body>
</html>
```

Ajoute dans `settings.py` :

```
CMS_TEMPLATES = [('home.html', 'Template accueil')]
TEMPLATES[0]['DIRS'] = ['templates']
```

Fichiers médias :

```
MEDIA_URL = "/media/"
MEDIA_ROOT = os.path.join(BASE_DIR, "media")
```

Dans `urls.py` :

```
from django.conf import settings
from django.conf.urls.static import static
urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

🔧 Plugins recommandés

Versioning & alias

```
pip install.djangocms-versioning.djangocms-alias
```

Ajoute à `INSTALLED_APPS` :

```
"djangocms_versioning",
"djangocms_alias",
```

Gestion des fichiers (filer)

```
pip install django-filer>=3.0
```

Ajoute à `INSTALLED_APPS` :

```
"filer",
"easy_thumbnails",
```

Et dans `settings.py` :

```
THUMBNAIL_HIGH_RESOLUTION = True
THUMBNAIL_PROCESSORS = (
    'easy_thumbnails.processors.colorspace',
    'easy_thumbnails.processors.autocrop',
    'filer.thumbnail_processors.scale_and_crop_with_subject_location',
    'easy_thumbnails.processors.filters'
)
```

CKEditor

```
pip install.djangocms-text-ckeditor
```

Ajoute à `INSTALLED_APPS`, puis :

```
python manage.py migrate
```

Frontend (Bootstrap 5)

```
pip install.djangocms-frontend.djangocms-link
```

Ajoute les sous-modules :

```
"djangocms_frontend",  
"djangocms_frontend.contrib.accordion",  
"djangocms_frontend.contrib.alert",  
...  
"djangocms_frontend.contrib.utilities",
```

Autres plugins utiles

```
pip install.djangocms-file.djangocms-picture.djangocms-video.djangocms-googlemap  
djangocms-snippet.djangocms-style
```

```
"djangocms_file",  
"djangocms_picture",  
"djangocms_video",  
"djangocms_googlemap",  
"djangocms_snippet",  
"djangocms_style",
```

✅ Vérification finale

```
python manage.py cms check
```

Cela analysera ta config et signalera les erreurs.



Templates & Placeholders dans django CMS



Templates

Les templates dans django CMS fonctionnent comme dans Django, avec l'héritage via `base.html`.

Un template pour django CMS doit inclure :

```
{% load cms_tags sekizai_tags %}
<head>
    {% render_block "css" %}
</head>
<body>
    {% cms_toolbar %}
    {% placeholder "content" %}
    {% render_block "js" %}
</body>
```

Exemple de configuration dans `settings.py`

```
CMS_TEMPLATES = [
    ('bootstrap5.html', 'Bootstrap 5'),
    ('minimal.html', 'Minimal'),
    ('whitenoise-static-files-demo.html', 'Démo Fichiers Statiques'),
]
```

 Les templates se trouvent dans `backend/templates/` dans le projet quickstart.

Placeholders

Un **placeholder** est une zone dynamique dans le HTML que le CMS remplit avec du contenu stocké en base de données.

```
{% block content %}
    {% placeholder "Feature" %}
    {% placeholder "Content" %}
    {% placeholder "Splashbox" %}
{% endblock %}
```

Pour voir les zones disponibles, passe en mode **Structure** via l'interface (en haut à droite).

Aliases statiques (ex : pied de page)

Les aliases statiques sont utiles pour afficher **le même contenu sur plusieurs pages** (comme un footer géré via l'interface).

Étapes :

1. Installer :

```
pip install.djangocms-alias
```

2. Ajouter à `INSTALLED_APPS` :

```
"djangocms_alias",
```


3. Dans `base.html`, insérer :

```
{% load djangocms_alias_tags %}

{% block content %}
    ...
    <footer>
        {% static_alias 'footer' %}
    </footer>
{% endblock %}

{% render_block "js" %}
```

4. Une fois ajouté, va dans le menu “Aliases...” pour l’éditer depuis le CMS.

 Tu peux créer ou modifier son contenu comme une page normale. Le contenu est partagé sur toutes les pages.

Menus dynamiques

Pour afficher le menu du CMS :

```
{% load menu_tags %}

<ul class="nav">
    {% show_menu 0 100 100 100 %}
</ul>
```

`show_menu` affiche la hiérarchie des pages. Les chiffres définissent la profondeur mais peuvent rester tels quels.

Intégration d’applications dans django CMS

Pourquoi intégrer une app dans django CMS ?

Intégrer une application ne signifie pas juste la faire cohabiter avec django CMS, mais **lier leur logique** pour créer un site cohérent, facilement gérable et extensible via le CMS.

 **Avantage** : tu n’as **pas besoin de modifier l’application** originale (utile pour les apps tierces).

Exemple : Intégrer une app de sondages (polls)

1. Installer l'app `polls`

```
pip install git+http://git@github.com:divio/django-polls.git#egg=polls
```

Ajoute `'polls'` à `INSTALLED_APPS` dans `settings.py`.

2. Ajouter les URLs

```
from django.urls import re_path, include
from django.conf.urls.i18n import i18n_patterns

urlpatterns = i18n_patterns(
    re_path(r'^admin/', include(admin.site.urls)),
    re_path(r'^polls/', include('polls.urls')),
    re_path(r'^$', include('cms.urls')),
)
```

 Important : les URLs de django CMS doivent être en dernier.

3. Migrer les données

```
python manage.py migrate polls
```

Va sur <http://localhost:8000/admin/> → onglet **Polls**.

Crée un sondage :

- Question : *Quel navigateur préférez-vous ?*
- Choix : Safari, Firefox, Chrome

Visite ensuite : <http://localhost:8000/en/polls/>

Améliorer l'intégration visuelle

Par défaut, les templates de `polls` sont minimaux. Fais-les hériter de `base.html` :

Crée ce fichier : `mysite/templates/polls/base.html` :

```
{% extends 'base.html' %}

{% block content %}
    {% block polls_content %}
    {% endblock %}
{% endblock %}
```

Recharge `/polls/` : la page utilise maintenant le même design que le reste du site CMS.

✂ Créer l'app d'intégration CMS

1. Créer l'app dédiée

```
python manage.py startapp polls_cms_integration
```

Structure obtenue :

```
mon-projet/  
├─ polls_cms_integration/  
│   └─ models.py, views.py, etc.
```

2. Ajouter à `INSTALLED_APPS`

Dans `settings.py` :

```
INSTALLED_APPS += ["polls_cms_integration"]
```

Cette app servira à **relier polls au CMS** via un plugin personnalisé. Elle contiendra le code d'intégration (plugins, apphooks...).

🔌 Créer un Plugin dans django CMS (intégration de sondage)

Nous allons créer un **plugin django CMS** qui affiche un sondage provenant de l'app `polls`.

🧬 Modèle du plugin

Dans `polls_cms_integration/models.py` :

```
from django.db import models  
from cms.models import CMSPlugin  
from polls.models import Poll  
  
class PollPluginModel(CMSPlugin):  
    poll = models.ForeignKey(Poll, on_delete=models.CASCADE)  
  
    def __str__(self):  
        return self.poll.question
```

📝 Ce modèle hérite de `CMSPlugin` et non de `models.Model`.

Ensuite, crée les migrations :

```
python manage.py makemigrations polls_cms_integration  
python manage.py migrate polls_cms_integration
```

⚙️ Classe du plugin

Crée un fichier `polls_cms_integration/cms_plugins.py` :

```
from cms.plugin_base import CMSPluginBase
from cms.plugin_pool import plugin_pool
from polls_cms_integration.models import PollPluginModel
from django.utils.translation import gettext as _

@plugin_pool.register_plugin
class PollPluginPublisher(CMSPluginBase):
    model = PollPluginModel
    module = _("Sondages")
    name = _("Plugin Sondage")
    render_template = "polls_cms_integration/poll_plugin.html"

    def render(self, context, instance, placeholder):
        context.update({"instance": instance})
        return context
```

💡 Convention de nommage :

- `PollPluginModel` → modèle
- `PollPluginPublisher` → classe du plugin

🖼️ Template du plugin

Crée le fichier :

`polls_cms_integration/templates/polls_cms_integration/poll_plugin.html`

Contenu :

```
<h1>{{ instance.poll.question }}</h1>

<form action="{% url 'polls:vote' instance.poll.id %}" method="post">
    {% csrf_token %}
    <div class="form-group">
        {% for choice in instance.poll.choice_set.all %}
            <div class="radio">
                <label>
                    <input type="radio" name="choice" value="{{ choice.id }}">
                    {{ choice.choice_text }}
                </label>
            </div>
        {% endfor %}
    </div>
```

```
        {% endfor %}
    </div>
    <input type="submit" value="Voter" />
</form>
```

✓ Tester le plugin

1. Redémarre le serveur Django (`runserver`)
2. Connecte-toi au CMS
3. Ajoute une instance du **Plugin Sondage** dans une page (via un placeholder)

Tu peux maintenant afficher un sondage Django directement dans une page CMS !

Intégration via AppHook dans django CMS

Nous allons connecter dynamiquement l'application `polls` à une page django CMS grâce à un **AppHook**, sans passer par `urls.py`.

Créer un AppHook

Dans l'app `polls_cms_integration`, crée un fichier `cms_apps.py` :

```
from cms.app_base import CMSApp
from cms.apphook_pool import apphook_pool

@apphook_pool.register
class PollsApphook(CMSApp):
    app_name = "polls"
    name = "Application Sondages"

    def get_urls(self, page=None, language=None, **kwargs):
        return ["polls.urls"]
```

`app_name` sert de namespace, `name` apparaît dans les réglages CMS de la page.

Alternative : URLs manuelles (si besoin)

Tu peux aussi déclarer les URLs directement :

```
from django.urls import path
from polls import views
```

```
def get_urls(self, page=None, language=None, **kwargs):
    return [
        path("<int:pk>/results/", views.ResultsView.as_view(), name="results"),
        path("<int:pk>/vote/", views.vote, name="vote"),
        path("<int:pk>/", views.DetailView.as_view(), name="detail"),
        path("", views.IndexView.as_view(), name="index"),
    ]
```

Nettoyer `urls.py`

✗ Supprime cette ligne dans `urls.py` :

```
path('polls/', include('polls.urls', namespace='polls'))
```

Sinon conflit : `URL namespace 'polls' isn't unique`.

Redémarrer le serveur

Redémarre le serveur pour que `cms_apps.py` soit pris en compte :


```
python manage.py runserver
```

💡 Astuce : pour éviter les redémarrages à chaque fois, ajoute dans `MIDDLEWARE` :

```
"cms.middleware.utils.ApphookReloadMiddleware",
```

Attacher l'AppHook à une page

1. Crée une **nouvelle page CMS**
2. Ouvre les **Paramètres avancés** de la page
3. Dans "Application", choisis **Application Sondages**
4. Enregistre

 Recharge la page CMS → l'app `polls` est maintenant disponible directement depuis cette page.

🚨 **Attention** : Ne crée **pas de sous-pages** sous une page avec AppHook → les URL sont redirigées vers l'app intégrée.

Étendre la barre d'outils (Toolbar) dans django CMS

Tu peux personnaliser la barre d'outils de django CMS pour intégrer des menus ou boutons liés à tes propres apps, ici `polls`.

Ajouter un menu "Sondages"

Dans l'app `polls_cms_integration`, crée un fichier `cms_toolbars.py` :

```
from cms.toolbar_base import CMSToolbar
from cms.toolbar_pool import toolbar_pool
from polls.models import Poll

class PollToolbar(CMSToolbar):

    def populate(self):
        self.toolbar.get_or_create_menu(
            'polls_cms_integration-polls', # identifiant unique
            'Sondages'                     # nom du menu dans l'UI
        )

toolbar_pool.register(PollToolbar)
```

Redémarre le serveur (`runserver`) pour voir le menu dans la toolbar sur toutes les pages.

Ajouter des éléments au menu

Ajoute des liens pour :

- afficher la liste des sondages
- créer un nouveau sondage

```
from cms.utils.urlutils import admin_reverse

class PollToolbar(CMSToolbar):

    def populate(self):
        menu = self.toolbar.get_or_create_menu('polls_cms_integration-polls',
        'Sondages')

        menu.add_sideframe_item(
            name='Liste des sondages',
            url=admin_reverse('polls_poll_changelist'),
        )

        menu.add_modal_item(
            name='Ajouter un sondage',
```

```
        url=admin_reverse('polls_poll_add'),
    )
```

Ajouter des boutons (facultatif)

Tu peux aussi ajouter des **boutons** dans la barre d'outils :

```
def populate(self):
    buttonlist = self.toolbar.add_button_list()

    buttonlist.add_sideframe_button(
        name='Liste des sondages',
        url=admin_reverse('polls_poll_changelist'),
    )

    buttonlist.add_modal_button(
        name='Ajouter un sondage',
        url=admin_reverse('polls_poll_add'),
    )
```

Limiter l'affichage aux pages pertinentes

Pour ne pas afficher le menu sur toutes les pages, ajoute ceci :

```
class PollToolbar(CMSToolbar):
    supported_apps = ['polls'] # l'app ciblée

    def populate(self):
        if not self.is_current_app:
            return
        ...
```

Code complet : cms_toolbars.py

```
from cms.utils.urlutils import admin_reverse
from cms.toolbar_base import CMSToolbar
from cms.toolbar_pool import toolbar_pool
from polls.models import Poll

class PollToolbar(CMSToolbar):
    supported_apps = ['polls']

    def populate(self):
        if not self.is_current_app:
```

```

        return

        menu = self.toolbar.get_or_create_menu('polls_cms_integration-polls',
'Sondages')

        menu.add_sideframe_item(
            name='Liste des sondages',
            url=admin_reverse('polls_poll_changelist'),
        )

        menu.add_modal_item(
            name='Ajouter un sondage',
            url=admin_reverse('polls_poll_add'),
        )

        buttonlist = self.toolbar.add_button_list()
        buttonlist.add_sideframe_button('Liste des sondages',
admin_reverse('polls_poll_changelist'))
        buttonlist.add_modal_button('Ajouter un sondage',
admin_reverse('polls_poll_add'))

        toolbar_pool.register(PollToolbar)

```

Étendre le menu de navigation dans django CMS

Actuellement, la navigation du site ne reflète que les **Pages CMS**.

Mais tu peux ajouter dynamiquement des entrées (nodes) dans le menu via le système de **CMSAttachMenu**.

Créer un menu personnalisé

Dans `polls_cms_integration`, crée le fichier `cms_menus.py` :

```

from django.urls import reverse
from django.utils.translation import gettext_lazy as _

from cms.menu_bases import CMSAttachMenu
from menus.base import NavigationNode
from menus.menu_pool import menu_pool

from polls.models import Poll

class PollsMenu(CMSAttachMenu):

```



```
name = _("Menu Sondages") # Nom visible dans l'admin
```

```
def get_nodes(self, request):
    nodes = []
    for poll in Poll.objects.all():
        node = NavigationNode(
            title=poll.question,
            url=reverse("polls:detail", args=(poll.pk,)),
            id=poll.pk,
        )
        nodes.append(node)
    return nodes
```


```
menu_pool.register_menu(PollsMenu)
```

Explication

- `CMSAttachMenu` : classe de base pour un menu attachable
- `get_nodes()` : retourne une liste de `NavigationNode` à afficher
- `menu_pool.register_menu()` : enregistre le menu

Lier ce menu à une page CMS

1. Va sur la page où l'**apphook polls** est déjà attaché
2. Ouvre les **Paramètres avancés**
3. Dans "**Menu attaché**", choisis **Menu Sondages**
4. Enregistre

 Tu peux forcer un apphook à ajouter automatiquement un menu via une méthode avancée (cf. doc officielle).

 Remarques :

- Si tu utilises des sous-pages, il faudra peut-être adapter le style du menu.
- Ce menu est illustratif : la page polls liste déjà tous les sondages.

Création de contenu avec un Wizard dans django CMS

Tu peux ajouter ton propre **assistant de création** (wizard) pour permettre aux utilisateurs de créer des objets via le bouton "Créer" de la barre d'outils du CMS.

Étape 1 : Formulaire pour le modèle

Dans `polls_cms_integration/forms.py` :

```
from django import forms
from polls.models import Poll

class PollWizardForm(forms.ModelForm):
    class Meta:
        model = Poll
        exclude = [] # Tous les champs sont inclus
```

Étape 2 : Enregistrer le wizard

Dans `polls_cms_integration/cms_wizards.py` :

```
from cms.wizards.wizard_base import Wizard
from cms.wizards.wizard_pool import wizard_pool
from polls_cms_integration.forms import PollWizardForm


class PollWizard(Wizard):
    pass

poll_wizard = PollWizard(
    title="Sondage",
    weight=200, # ordre d'affichage dans la liste
    form=PollWizardForm,
    description="Créer un nouveau sondage",
)

wizard_pool.register(poll_wizard)
```

Tester le wizard

1. Redémarre le serveur
2. Va sur une page CMS
3. Clique sur **Créer** dans la barre d'outils
4. Tu verras l'option **Sondage** apparaître : un formulaire simple s'ouvrira dans une fenêtre modale

 Remarque :

Dans cet exemple, seul l'objet `Poll` est créé.

Si tu veux gérer en même temps les **questions associées (ForeignKey)**, il faudrait un formulaire plus

complexe (inline, formset...), ce qui dépasse le cadre de ce tutoriel.

🧩 Guide général sur les Plugins dans django CMS

Les **CMS Plugins** permettent d'insérer des éléments dynamiques dans une page, un bloc ou un placeholder django CMS. Ils sont **réutilisables** et peuvent afficher automatiquement des données venant d'un autre modèle Django.

😬 Pourquoi créer un plugin ?

Tu as besoin d'un plugin si tu veux :

- Publier dynamiquement du contenu venant d'une autre app Django
- Éviter de modifier manuellement des pages statiques
- Réutiliser un même composant sur plusieurs pages avec des paramètres différents

Exemple : une maison de disques utilise un plugin "Dernières sorties" qui affiche les nouveaux albums depuis la base de données sans avoir à modifier la page.

🧱 Structure d'un plugin

Un plugin suit la logique **Modèle – Vue – Template** (MVT) de Django :

Composant	Rôle	Classe Django CMS
Modèle (facultatif)	Configuration du plugin	<code>CMSPlugin</code>
Vue	Logique métier et affichage	<code>CMSPluginBase</code> (hérite de <code>ModelAdmin</code>)
Template	Rendu HTML	Fichier HTML dans <code>templates/</code>

🚩 Le modèle est **optionnel**. Tu peux créer un plugin sans modèle si son comportement est unique et fixe.

🔧 Exemple concret

- Un plugin sans configuration : toujours le même contenu (ex. "Top des ventes 7 derniers jours").
 - Un plugin configurable : choix d'une catégorie, d'un artiste, ou d'un délai personnalisé (7, 30, 90 jours, etc.).
-

⚙️ Options disponibles depuis `ModelAdmin`

`CMSPluginBase` hérite de `ModelAdmin`, donc tu peux utiliser :

```
exclude
fields
fieldsets
form
inline
readonly_fields
```

Mais certaines options **n'ont aucun effet** (ex. pagination admin ou recherche) :

```
actions
list_display
ordering
search_fields
date_hierarchy
```

Résumé

- Un plugin = logique Python + rendu HTML + éventuellement un modèle
- Il s'insère dans n'importe quel **placeholder**
- Il permet de garder ton site à jour automatiquement
- Il est **réutilisable** avec des paramètres différents sur plusieurs pages



Application Hooks (AppHooks) dans django CMS

Un **AppHook** permet d'attacher une application Django à une page django CMS, pour une intégration **totale** : menu, URL, droits, publication, etc.

Pourquoi utiliser un AppHook ?

Prenons un exemple : tu as une application Django qui affiche des **records olympiques**.

Si tu l'ajoutes juste dans `urls.py` :

- Accessible via `/records/`
-  Fonctionne
-  Pas intégré au CMS :
 - Elle n'apparaît pas dans le menu
 - Le CMS peut créer une page `/records/` qui sera en conflit
 - Pas de gestion de publication, permissions, ou historique

Si tu utilises un **AppHook** :

- L'app est attachée à une **page CMS**

- Le CMS **gère l'URL et les sous-URLs** (ex: `/records/1984`)
 - Les pages sont **déplaçables dans l'arborescence CMS**
 - Tu peux **appliquer les workflows CMS** à l'app
-

Comment ça marche ?

1. Crée une classe AppHook (`CMSApp`)
 2. Enregistre-la avec `apphook_pool.register`
 3. Attache-la à une page dans les **Paramètres avancés**
 4. Publie la page → l'app est active
-

Plusieurs AppHooks pour la même app

Tu peux :

- Attacher **plusieurs fois** la même app sur des pages différentes
- Fournir **des configurations différentes** par page

Exemple :

- `/athlétisme/` affiche les résultats d'athlétisme
- `/cyclisme/` affiche ceux du cyclisme

Tu peux créer une **classe de configuration d'AppHook** avec des champs spécifiques, visibles dans l'admin.

Attention

- Un AppHook **n'est actif que s'il est attaché à une page publiée**
 - Il **englobe tous les chemins sous-jacents** (ex: `/records/**`)
 - Évite donc de créer des **sous-pages** manuelles sous une page apphookée
-

Système de publication dans django CMS

Comportement par défaut

Sans système de versioning, **toutes les pages sont publiées immédiatement** après enregistrement. Cela signifie que toute modification devient immédiatement visible sur le site public.

Ajouter un système de versioning

Pour contrôler le cycle de vie des pages (brouillon, publication, archivage...), on peut utiliser `django-cms-versioning`. Ce module permet de gérer plusieurs versions d'un même contenu avec

différents états.

💡 Bien que centré ici sur les pages, ce système peut aussi s'appliquer à d'autres objets comme les alias (via `djangocms-alias`).

📁 États de version

Chaque page (`Page`) contient un ou plusieurs objets `PageContent`, chacun représentant le contenu dans une langue spécifique. Le versioning s'applique à ces `PageContent`.

Voici les **états possibles** :

- `draft` (brouillon) :
 - Modifiable. Une seule version brouillon par langue.
 - Non visible au public.
- `published` (publiée) :
 - Version en ligne, visible par tous.
 - Non modifiable. Toute modification crée une nouvelle version brouillon.
- `unpublished` (dépubliée) :
 - Ancienne version retirée de la publication.
 - Plusieurs versions peuvent coexister.
- `archived` (archivée) :
 - Jamais publiée, mise de côté pour usage futur.
 - Peut être retransformée en brouillon.

🔴 Chaque brouillon génère un **nouveau numéro de version**.

👤 Accès aux contenus en code

Par défaut, seules les versions **publiées** sont accessibles :

```
PageContent.objects.filter(language="en") # Ne retourne que les versions publiées
```

Pour accéder aux versions non publiées, utilise le **manager admin** :

```
PageContent.admin_manager.filter(page=my_page, language="en") # Toutes les versions
```

🔍 Récupérer un brouillon en code :

```
from djangocms_versioning.constants import DRAFT
from djangocms_versioning.models import Version

version = Version.objects.get(content__page=my_page, content__language="en",
```

```
status=DRAFT)
draft_content = version.content
```

Accéder à la version “courante” (brouillon ou publiée)

```
for content in PageContent.admin_manager.filter(page=my_page).current_content():
    if content.versions.first().state == DRAFT:
        # Faire quelque chose avec le brouillon
```

Points importants

- Une page **publiée est visible** même si sa page parente ne l'est pas.
- Une **apphook** reliée à une page prend le contrôle de tous les chemins en dessous (`/ma-page/...`).
- La versioning ne fonctionne **qu'après publication** d'une page liée au contenu versionné.

Gestion multilingue avec django CMS

Concepts de base

django CMS prend en charge **plusieurs langues** de manière avancée. Il est capable d'afficher le contenu dans différentes langues, de proposer des **langues de repli** si une traduction est absente, et de **déterminer la langue préférée de l'utilisateur** de façon intelligente.

Comment django CMS détermine la langue préférée de l'utilisateur ?

Il s'appuie sur le comportement standard de Django, en suivant cet ordre :

1. **Préfixe de langue dans l'URL** (`/en/`, `/fr/`, etc.)
2. **Langue stockée en session**
3. **Cookie de langue** (`django_language`)
4. **Langue préférée du navigateur** (`Accept-Language` header)

Par défaut, **aucune session ni cookie de langue n'est utilisée**.

 Pour activer les cookies de langue :

Ajoute `cms.middleware.language.LanguageCookieMiddleware` dans `MIDDLEWARE` de ton `settings.py`.

Quelle langue est servie ?

Une fois la langue déterminée, django CMS vérifie si du contenu existe dans cette langue. Il suit ensuite la logique suivante :

- Si la langue est disponible :  le contenu est affiché dans cette langue.

- Si la langue n'est pas disponible :
 - Il consulte le fichier `CMS_LANGUAGES` pour chercher une **langue de repli** (`fallbacks`).
 - Il sert le contenu dans une des langues définies comme fallback, si disponible.
-

Et les menus ?

Le comportement du menu est influencé par la configuration suivante :

```
CMS_LANGUAGES = {
    'default': {
        'hide_untranslated': True,
        ...
    },
    ...
}
```

- Si `hide_untranslated = True` (valeur par défaut) :
 - Les pages **non traduites dans la langue actuelle** ne s'affichent **pas du tout** dans le menu.
 - Si tu veux afficher quand même les pages, même sans traduction, mets cette option à `False`.
-

Bonnes pratiques

- Traduis au moins la page d'accueil dans toutes les langues disponibles.
 - Active les cookies de langue pour offrir une meilleure expérience utilisateur.
 - Utilise `fallbacks` pour éviter les pages vides si une langue manque.
-

Internationalisation avec django CMS FR

django CMS est particulièrement performant dans la gestion des contenus multilingues. Il peut être configuré de manière fine pour répondre à des besoins variés en matière d'internationalisation (i18n) et de localisation (l10n).

Objectif

Permettre à un site de servir dynamiquement ses contenus dans plusieurs langues, selon les préférences de l'utilisateur et les traductions disponibles.

Configuration dans `CMS_LANGUAGES`

La clé `CMS_LANGUAGES` du fichier `settings.py` permet :

- de définir les langues disponibles,
- de configurer les langues par site,
- d'activer ou non les traductions par défaut,
- de définir des langues de repli (*fallbacks*).

Exemple :

```
CMS_LANGUAGES = {  
    1: [  
        {'code': 'fr', 'name': 'Français', 'fallbacks': ['en'], 'public': True},  
        {'code': 'en', 'name': 'English', 'fallbacks': ['fr'], 'public': True},  
    ]  
}
```

URLs multilingues

Pour que les URLs soient sensibles à la langue (`/en/`, `/fr/`, etc.), tu dois utiliser `i18n_patterns()` dans ton `urls.py` :

```
from django.conf.urls.i18n import i18n_patterns  
  
urlpatterns = i18n_patterns(  
    path('admin/', admin.site.urls),  
    path('', include('cms.urls')),  
)
```

Cela permet d'activer la prise en charge des préfixes linguistiques dans les routes.

Détection de la langue de l'utilisateur

django CMS suit l'ordre suivant pour détecter la langue à utiliser :

1. Le **code de langue dans l'URL** (`/en/`, `/fr/`)
2. La **langue stockée dans la session** (via `request.session`)
3. La **langue en cookie** (avec `LanguageCookieMiddleware`)
4. La **langue préférée du navigateur** (header `Accept-Language`)
5. La langue par défaut (`LANGUAGE_CODE` dans `settings.py`)

Ressources utiles

- [Django - Traduction et internationalisation](#)
 - [django CMS - Multilingue](#)
-

Permissions dans django CMS

Introduction

Le système de permissions de **django CMS** est **puissant, granulaire et multi-niveaux**. Il peut fonctionner en deux modes selon la configuration de `CMS_PERMISSION` :

- `False` : Mode simple – utilise uniquement les permissions standard de Django (Users et Groups).
 - `True` : Mode avancé – ajoute des permissions **au niveau des pages** (row-level) dans le CMS.
-

Permissions utilisateur clés

Dans l'admin Django > *Authentication and Authorization*, vous pouvez attribuer ces permissions :

Pour django CMS :

- `cms | plugin`
- `cms | page`
- `cms | placeholder`
- `cms | placeholder reference`
- ⚠ Permission spéciale : `cms | placeholder | Can use Structure mode`

Pour les paquets tiers :

- **djangoCMS-alias** :
 - `alias`, `alias content`, `category`
 - **djangoCMS-frontend** :
 - `UI item` (nécessite la commande `python manage.py frontend sync_permissions`)
 - **djangoCMS-text** :
 - `text`
 - **djangoCMS-versioning** :
 - `alias content version`, `page content version`, `version`
-

Permissions globales et locales

Avec `CMS_PERMISSION = True`, il faut aussi **donner des droits aux pages** (en plus des droits utilisateur).

Permissions globales


Attribuées à **toutes les pages** d'un site (ou sous-site).

Gérées dans l'admin : `django CMS > Pages global permissions`.

Permissions par page

Attribuées à une page spécifique (ou à ses enfants / descendants).

Gérées depuis la **toolbar** : `Page > Permissions`.

 Un utilisateur non superuser a besoin d'au moins une permission globale ou locale + les permissions Django correspondantes.

✓ Options de permission par page

Peuvent être attribuées à un **utilisateur** ou un **groupe** :

- `Can add`
- `Can edit`
- `Can delete`
- `Can publish`
- `Can change advanced settings`
- `Can change permissions`
- `Can move`

⚠ Pour modifier les plugins, les permissions Django standards sont également nécessaires :

- `cms | cms plugin | Can edit cms plugin`
-

🔧 Permissions spéciales

- `Login required` : page visible uniquement pour les utilisateurs authentifiés.
 - `Menu visibility` : contrôle l'apparition dans les menus (tous, connectés uniquement, anonymes uniquement).
 - `View restrictions` : restreint la visibilité à certains groupes/utilisateurs (hors éditeurs autorisés).
-

🧩 Modèles d'admin supplémentaires

Avec `CMS_PERMISSION = True`, deux nouveaux modèles apparaissent dans l'admin CMS :

- `User groups (page)`
- `Users (page)`

👉 Ceux-ci reflètent simplement les Groupes et Utilisateurs Django déjà existants.

🧬 Stratégies de permission

- **Utiliser des Groupes, pas des Utilisateurs individuels** : simplifie la gestion.

- **Composer les permissions avec des Groupes :**
 - Exemple : Groupe "Éditeur basique" = peut éditer, mais pas créer.
 - Groupe "Éditeur principal" = peut créer des pages.
- **Deux axes à gérer :**
 1. Ce qu'un utilisateur peut faire (publier, créer, modifier...)
 2. Où il peut le faire (ex. uniquement sur le sous-site Europe)

✓ Exemple concret

Vous pouvez avoir :

- Groupe `Europe` : assigné aux éditeurs qui gèrent les pages européennes.
- Groupe `Weblog` : donne accès au blog uniquement.
- Groupe `Administrateurs` : a tous les droits sur tout le site.

Ensuite, vous n'avez qu'à affecter les utilisateurs aux groupes adaptés.

🔍 Recherche dans django CMS

Introduction

Avec l'arrivée de django CMS 4.x, l'ancien système de recherche `aldryn-search` est devenu obsolète. À la place, on recommande d'utiliser des outils comme `django-haystack`, `djangoCMS-haystack`, ou d'autres solutions externes.

Mise en place rapide avec django-haystack

django-haystack permet d'indexer vos modèles et de les rendre accessibles via un moteur de recherche interne à votre projet.

Exemple minimal avec PageContent

```
from cms.models import PageContent
from haystack import indexes

class PageContentIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=False)
    title = indexes.CharField(indexed=False, stored=True)
    url = indexes.CharField(indexed=False, stored=True)


    def get_model(self):
        return PageContent
```

```

def index_queryset(self, using=None):
    return self.get_model().objects.filter(language=using)

def prepare(self, instance):
    data = super().prepare(instance)
    data["url"] = instance.page.get_absolute_url()
    data["title"] = instance.title
    data["text"] = data["title"] + (instance.meta_description or "")
    return data

```

 Place ce fichier dans `search_indexes.py` à la racine d'une app déclarée dans `INSTALLED_APPS`.

Configuration de Haystack

Ajoute la configuration suivante dans `settings.py` :

```

HAYSTACK_CONNECTIONS = {
    'default': {
        "ENGINE": "haystack.backends.whoosh_backend.WhooshEngine",
        "PATH": os.path.join(BASE_DIR, "search_index", "whoosh_index_default"),
    },
    "en": {
        "ENGINE": "haystack.backends.whoosh_backend.WhooshEngine",
        "PATH": os.path.join(BASE_DIR, "search_index", "whoosh_index_en"),
    },
    "de": {
        "ENGINE": "haystack.backends.whoosh_backend.WhooshEngine",
        "PATH": os.path.join(BASE_DIR, "search_index", "whoosh_index_de"),
    }
}

```

Rebuild de l'index

Lance cette commande pour générer ton index :

```
python manage.py rebuild_index
```

Tester l'index avec SearchQuerySet

```

from haystack.query import SearchQuerySet

qs = SearchQuerySet(using="en")
for result in qs.all():
    print(result.text)

```

Aller plus loin

Tu peux :

- Ajouter d'autres modèles personnalisés à l'index
- Créer des vues pour la recherche
- Créer un formulaire pour rechercher sur ton site

💡 **Astuce** : adapte la logique d'indexation à ton propre système de versioning (ex : `django-cms-versioning`), en filtrant les contenus publiés ou brouillons selon le contexte.

Utiliser des appareils tactiles avec django CMS

⚠ Important :

Ces remarques s'appliquent uniquement aux interfaces d'administration et d'édition de django CMS.

Le site publié (frontend visible par les visiteurs) est **indépendant** et doit être optimisé séparément par le développeur.

🧠 Comportement général

• Double-clic vs tactile :

django CMS repose fortement sur le double-clic (édition rapide), mais le tactile n'a pas d'équivalent exact.

→ Une tape (tap) est interprétée intelligemment comme une action contextuelle :

- soit **édition** (équivalent d'un double-clic),
- soit **sélection** (équivalent d'un clic simple).

• Déplacement ou défilement :

Par exemple dans la *liste des pages*, certaines zones permettent :

- le **glisser-déposer** (repositionnement),
- d'autres le **scroll** (navigation verticale).

• Bulles d'aide (tooltips) :

Sur les appareils tactiles, les infobulles dépendantes du survol (**hover**) sont souvent indisponibles.

📱 Appareils compatibles connus

⚠ Les petits écrans (ex. téléphones) ne sont **pas adaptés** pour une utilisation efficace.

Fonctionnent bien :

- **iOS** : iPad Air 1, iPad Mini 4

- **Android** : Sony Xperia Z2 Tablet, Galaxy Tab 4
 - **Windows** : Microsoft Surface
-

Intégration frontend

- Le **toolbar** de django CMS repose sur un HTML/CSS propre et responsive.
 - Si vous utilisez Bootstrap, Foundation, etc., évitez :
 - le CSS cassé ou mal indenté,
 - des éléments avec peu de **padding** (problème de tap detection).
-

Problèmes connus

Généraux

- Liens sans **padding suffisant** : difficilement éditables via tactile.
- Menus de navigation uniquement composés de liens : difficile à double-cliquer.
- Problèmes d'ajout de liens sur Android à cause du clavier.
- Utilisation en **navigation privée** : ralentissements (absence de local storage → sessions Django utilisées).

Problèmes avec CKEditor

- Sur mobile :
 - le clavier peut **mal se positionner**.
 - l'éditeur peut se déplacer ou afficher des **artefacts** visuels (Safari/iOS).
 - certains boutons semblent "manquants" mais sont encore **cliquables** (bug de rendu).

Problèmes Django Admin

- Dans l'arborescence des pages, le **focus automatique** sur la barre de recherche déclenche le clavier au premier clic → comportement gênant sur mobile.
-

Recommandations

- Préférez une **tablette récente** ou un **hybride PC/tactile**.
 - Ajoutez du **padding** aux éléments éditables.
 - Ne testez pas en navigation privée sur mobile.
 - Testez vos gabarits avec **la barre django CMS activée** sur un appareil tactile réel.
-

Schémas de couleurs (clair/foncé) avec Django CMS

Scope

⚠ **Important** : Ces options ne concernent que l'interface d'administration et d'édition de django CMS. Le site public n'est pas concerné.

Pour que le thème soit effectif, vous devez installer le package `djangoCMS-admin-style` (v3.2+). Sinon, c'est le thème de Django qui est utilisé (souvent en fonction des préférences du navigateur).

🎨 Changer le thème (CMS_COLOR_SCHEME)

Dans `settings.py`, ajoutez :

```
CMS_COLOR_SCHEME = "light" # par défaut
# CMS_COLOR_SCHEME = "dark"
# CMS_COLOR_SCHEME = "auto" # selon préférence navigateur/OS
```

💡 Astuce

Si vous forcez un thème clair/sombre, ajoutez `data-theme` à la balise HTML pour éviter le "flicker" au chargement :

```
<html data-theme="light">
```

NEW Depuis `django CMS 3.11.4`, on utilise `data-theme` (et plus `data-color-scheme`).

🌙 Bouton pour changer de thème (CMS_COLOR_SCHEME_TOGGLE)

Dans `settings.py` :

```
CMS_COLOR_SCHEME_TOGGLE = True # Affiche l'icône soleil/lune/auto dans la barre d'admin
```

✏ Adapter son CSS à la couleur

- Utilisez le moins possible `color` ou `background-color` en dur.
- Favorisez les variables CSS standard (de django CMS ou de Django).
- Exemple recommandé :

```
color: var(--dca-primary, var(--primary, #00bbff));
```

⚠ **Évitez les media queries** `@media (prefers-color-scheme: dark)` — elles contournent les réglages forcés.

🎨 Variables CSS disponibles

Variable	Couleur django CMS	Fallback Django	Hex par défaut
<code>--dca-white</code>	#ffffff	<code>--body-bg</code>	#ffffff
<code>--dca-gray</code>	#666	<code>--body-quiet-color</code>	#666
<code>--dca-gray-lightest</code>	#f2f2f2	<code>--darkened-bg</code>	#f8f8f8
<code>--dca-gray-lighter</code>	#ddd	<code>--border-color</code>	#ccc
<code>--dca-primary</code>	#00bbff	<code>--primary</code>	#79aec8
<code>--dca-black</code>	#000	<code>--body-fg</code>	#303030

✓ Recommandations synthétiques

- Blanc : `var(--dca-white, var(--body-bg, #fff))`
- Gris : `var(--dca-gray, var(--body-quiet-color, #666))`
- Primaire : `var(--dca-primary, var(--primary, #0bf))`
- Noir : `var(--dca-black, var(--body-fg, #000))`

🔗 Fonctionnement du système de menus dans django CMS

🌳 1. Concepts de base : les "Soft Roots"

◆ Qu'est-ce qu'un *Soft Root* ?

Un *soft root* est une page définie comme **point de départ local** d'un menu. Plutôt que d'afficher l'arborescence complète du site depuis la racine, le menu commence **à partir du soft root le plus proche**.

◆ Pourquoi c'est utile ?

Si ton site a une structure de navigation très profonde (plusieurs niveaux), cela évite d'afficher un menu trop chargé et difficile à lire.

🔧 Exemple :

Sans soft root :

Accueil > Faculté de Médecine > Département > Sous-département > Sujet > Sous-sujet

Avec un soft root défini au niveau du **Département**, le menu affiche simplement :

Département > Sujet > Sous-sujet

⚙️ 2. Architecture du système de menus

Le système de menus dans django CMS **n'est pas monolithique**. Il repose sur des composants indépendants qui collaborent :

- ♦ **Générateurs de menus** : construisent la liste initiale des nœuds du menu
- ♦ **Modificateurs** : ajustent dynamiquement ces nœuds selon des règles (ex. : cacher certains niveaux, supprimer des parties non pertinentes)

3. Les composants techniques

Les nœuds (`NavigationNode`)

Chaque élément de menu est un objet avec :

- un **titre**
- une **URL**
- un **parent**
- une **liste d'enfants**
- un attribut `attr` pour y stocker des données personnalisées

⚠ Tous les nœuds ne représentent **pas forcément** des pages django CMS.

4. Générateurs de menus

Les générateurs sont des classes héritées de `menus.base.Menu`.

Exemple : `cms.menu.CMSMenu`

Leur méthode principale est :

```
get_nodes()
```

Elle retourne la liste initiale des nœuds du menu, souvent basée sur les objets `Page` de la base de données.

5. Modificateurs de menus

Ce sont des classes héritant de `menus.base.Modifier`, qui modifient les nœuds selon des règles.

Exemples :

- `cms.menu.SoftRootCutter` : coupe les nœuds hors du sous-arbre d'un *soft root*
- `cms.menu.NavExtender` : ajoute des entrées de menus externes
- `menus.modifiers.AuthVisibility` : supprime les nœuds non accessibles à l'utilisateur
- `menus.modifiers.Level` : attribue le niveau (profondeur) à chaque nœud

Ils utilisent la méthode :

```
modify(request, nodes, namespace, root_id, post_cut, breadcrumb)
```

Certains modifient **avant** et d'autres **après** la découpe du menu (`post_cut = True/False`).

🧩 6. Cycle de vie d'un menu (ex: `{% show_menu %}`)

Quand tu utilises le tag `{% show_menu %}` dans un template :

1. Appel de `get_context()` → commence la construction du menu
2. Appel à `MenuPool.get_nodes()` → génère tous les nœuds disponibles
3. Coupe les niveaux inutiles (`cut_levels`)
4. Applique les **modificateurs** (`apply_modifiers`)
5. Retourne le menu dans le contexte de rendu

🧠 7. Ce qu'il faut retenir

- Le système est **modulaire et extensible**
- Il repose sur une **liste de nœuds navigables**
- Chaque niveau de transformation (générateur/modificateur) est **enregistré et appliqué dynamiquement**
- Les soft roots permettent d'**adapter le menu à son contexte**
- Tu peux facilement créer des menus personnalisés en héritant des classes de base

Frontend Integration avec django CMS

🏗️ Indépendance du frontend

Django CMS **ne dépend d'aucun framework frontend particulier**. Vous pouvez utiliser ce que vous voulez : Bootstrap, Tailwind, Foundation, etc.

Le CMS ne fait **aucune supposition sur votre code HTML, CSS ou JavaScript**.

⚙️ Exception : mode édition

Lorsque vous activez la barre d'édition (`toolbar`) pour modifier du contenu en ligne, django CMS **injecte son propre JavaScript et ses propres styles**. Ces éléments peuvent :

- perturber votre CSS si mal structuré ;
- ou **ne pas réinitialiser vos widgets JS** lors de modifications dynamiques.

Exemple concret :

Depuis la version 3.5, le système recharge dynamiquement certaines zones de contenu après qu'un plugin a été déplacé, ajouté ou supprimé. Cela signifie que **votre frontend doit être prêt à recharger dynamiquement ses propres composants** (carrousel, sliders, graphiques JS, etc.).

🔧 Exemple d'intégration avec Less.js

Si vous utilisez un préprocesseur comme **Less.js**, il se peut que vos modifications CSS **ne soient pas rechargées** après une opération sur un plugin.

🔧 Pour y remédier, vous pouvez écouter l'événement `cms-content-refresh` généré par django CMS :

```
{% if request.toolbar and request.toolbar.edit_mode_active %}
<script>
CMS.$(window).on('cms-content-refresh', function () {
    less.refresh(); // Recharge votre CSS Less dynamiquement
});
</script>
{% endif %}
```

✅ À placer juste après le JavaScript de la toolbar dans vos templates.

🎯 Conseils pratiques

- **N'utilisez pas de JavaScript inline** dans vos plugins sans gestion de rechargement.
- **Initialisez vos composants dans un gestionnaire** `cms-content-refresh` si besoin.
- Utilisez `CMS.$` (jQuery spécifique à django CMS) pour cibler les événements internes.



📌 En résumé

Élément	Comportement dans django CMS
HTML/CSS personnalisé	Totalement libre
Framework frontend imposé	Aucun
Impact en édition	Oui, si widgets ou CSS dynamiques
Solution	Écouter <code>cms-content-refresh</code>



🧠 **Astuce** : si vos composants ne se réinitialisent pas après un ajout/suppression de plugin, pensez à vérifier que leurs initialisations sont bien déclenchées **à chaque mise à jour du DOM**.

Sommaire du guide django CMS



Mise en place

-  [Prérequis](#)
-  [Installation rapide \(avec Docker\)](#)
-  [Installation manuelle \(sans Docker\)](#)




Ajouter django CMS à un projet existant

-  [Modifier settings.py](#)
-  [Gérer les templates et les fichiers statiques](#)





Extensions et modules recommandés

-  [Plugins à installer](#)
-  [Vérification de la configuration](#)



Mise en page avec django CMS

-  [Templates & Placeholders](#)
-  [Aliases statiques \(footer, header...\)](#)
-  [Menus dynamiques](#)

Intégrer une application Django

-  [Pourquoi intégrer une app ?](#)
-  [Exemple avec l'app de sondages](#)
-  [Créer un plugin CMS](#)
-  [AppHook : raccorder une app à une page](#)

Personnaliser l'administration

-  [Étendre la barre d'outils \(Toolbar\)](#)
-  [Ajouter un menu personnalisé](#)

Assistants de création (Wizard)

-  [Créer un Wizard personnalisé](#)

CMS Plugins – Guide général

-  [Pourquoi créer un plugin ?](#)



AppHooks (approfondi)

-  [À quoi servent les AppHooks ?](#)

Système de publication

-  [Gérer les brouillons, publications, archivages...](#)

Multilingue et internationalisation

-  [Activer les langues et fallback](#)
-  [Configuration avancée i18n/l10n](#)

Permissions et sécurité

-  [Attribuer les bons droits](#)

Intégrer un moteur de recherche

-  [Utiliser django-haystack](#)

Utilisation sur tablette / mobile

-  [Optimiser l'interface tactile](#)

Thème clair ou sombre

-  [Changer le thème de l'admin](#)

Comprendre le système de menus

-  [Soft Roots & Menu Extenders](#)

Intégration frontend

-  [Adapter vos scripts JS et styles](#)