

---

# MapEntity Documentation

*Release 8.0.1*

**Makina Corpus**

**Apr 15, 2022**



# CONTENTS

<b>1</b>	<b>Installation</b>	<b>3</b>
1.1	System prerequisites . . . . .	3
1.2	Manual installation With a PostGIS database . . . . .	3
<b>2</b>	<b>Getting started</b>	<b>5</b>
2.1	Settings . . . . .	5
2.2	Model . . . . .	6
2.3	Admin . . . . .	7
2.4	URLs . . . . .	7
2.5	Initialize the database . . . . .	7
2.6	Start the app . . . . .	8
2.7	Done! . . . . .	8
<b>3</b>	<b>Customization</b>	<b>9</b>
3.1	Views . . . . .	9
3.2	Filters . . . . .	10
3.3	Forms . . . . .	10
3.4	Templates . . . . .	11
3.5	Exports . . . . .	11
3.6	Settings . . . . .	12
<b>4</b>	<b>Development</b>	<b>13</b>
4.1	Release . . . . .	13
<b>5</b>	<b>Indices and tables</b>	<b>15</b>



MapEntity is a CRUD interface for geospatial entities built with Django.



## INSTALLATION

### 1.1 System prerequisites

For GeoDjango to work, your system must meet the following requirements:

```
$ sudo apt install binutils libproj-dev gdal-bin
```

For weasyprint and PDF generation, you need:

```
$ sudo apt install libjpeg62 libjpeg62-dev zlib1g-dev libcairo2 libpango-1.0-0_
↳ libpangocairo-1.0-0 libgdk-pixbuf2.0-0 libffi-dev shared-mime-info
```

If you use spatialite, you will need:

```
$ sudo apt install libsqlite3-mod-spatialite
```

Else, if you use PostGIS, you will need:

```
$ sudo apt install libpq-dev
```

### 1.2 Manual installation With a PostGIS database

In order to use MapEntity you'll need to create a geospatial database. Feel free to skip this section if you already know how to do this. Here is how you can create a PostGIS database:

As user `postgres`, create a new user and database:

```
$ createuser -PSRD dbuser
Enter password for new role:
Enter it again:
$ createdb --owner=dbuser spatialdb
```

Now enable PostGIS extension for your new database:

```
$ psql -q spatialdb
spatialdb=# CREATE EXTENSION postgis;
```

Create a *virtualenv*, and activate it:

```
virtualenv env/
source env/bin/activate
```

Then install the Python packages:

```
$ pip install django-mapentity
```

Since you will PostgreSQL, also install its python library:

```
$ pip install psycopg2
```



## GETTING STARTED

In this short tutorial, we'll see how to create an app to manage museum locations.

### 2.1 Settings

Create your django Project and your main app:

```
$ django-admin.py startproject museum
$ cd museum/
$ python3 manage.py startapp main
```

Edit your Django settings to point to your PostGIS database:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.contrib.gis.db.backends.postgis',
        'NAME': 'spatialdb',
        'USER': 'dbuser',
        'PASSWORD': 's3cr3t',
        'HOST': 'localhost',
        'PORT': '',
    }
}
```

Add these entries to your `INSTALLED_APPS`:

```
'paperclip',
'djgeojson',
'compressor',
'easy_thumbnails',
'crispy_forms',
'rest_framework',
'embed_video',
'modeltranslation'
'mapentity', # Make sure mapentity settings are loaded before leaflet ones
'leaflet',
'main', # the app you just created
```

Add `django.middleware.locale.LocaleMiddleware` to your `MIDDLEWARE` classes.

Setup your list of supported languages:

```
LANGUAGES = (  
    ('en', 'English'),  
    ('fr', 'French'),  
)
```

Specify a media URL:

```
MEDIA_URL = '/media/'
```

Specify a static root:

```
import os  
BASE_DIR = os.path.dirname(os.path.abspath(__file__))  
STATIC_ROOT = os.path.join(BASE_DIR, 'static')
```

Add MapEntity and request context processors to the list of default context processors:

```
TEMPLATES = [  
    {  
        ...  
        'OPTIONS': {  
            ...  
            'context_processors': [  
                ...  
                "django.core.context_processors.request",  
                "mapentity.context_processors.settings",  
            ]  
        }  
    }  
]
```

## 2.2 Model

Create a GeoDjango model which also inherits from `MapEntityMixin`. Note that you'll need to specify the *GeoDjango* manager, as below:

```
from django.contrib.gis.db import models  
from mapentity.models import MapEntityMixin  
  
class Museum(MapEntityMixin, models.Model):  
    geom = models.PointField()  
    name = models.CharField(max_length=80)
```

## 2.3 Admin

Create a file `admin.py` in the main directory and register your model against the admin registry:

```
from django.contrib import admin
from leaflet.admin import LeafletGeoAdmin

from .models import Museum

admin.site.register(Museum, LeafletGeoAdmin)
```

## 2.4 URLs

Register your MapEntity views in `main/urls.py`:

```
from main.models import Museum
from mapentity import registry

urlpatterns = registry.register(Museum)
```

Then glue everything together in your project's `urls.py`:

```
from django.conf.urls import patterns, include, url
from django.contrib import admin

admin.autodiscover()

urlpatterns = [
    '',
    path('', 'main.views.home', name='home'),
    path('login/', 'django.contrib.auth.views.login', name='login'),
    path('logout/', 'django.contrib.auth.views.logout', name='logout'),
    path('', include('mapentity.urls')),
    path('paperclip/', include('paperclip.urls')),
    path('admin', admin.site.urls),
]
```

## 2.5 Initialize the database

Create a database schema based on your models:

```
$ python manage.py migrate
```

Create all permission objects with this command:

```
$ python manage.py update_permissions_mapentity
```

## 2.6 Start the app

```
$ python manage.py runserver
```

## 2.7 Done!

Now you should be able to visit <http://127.0.0.1:8000/admin> and add a museum with a name (if you can't see a map, make sure you're using Django 1.6).

Then visit <http://127.0.0.1:8000/museum/list/> and you should be able to see your museum listed.

## CUSTOMIZATION

### 3.1 Views

Create a set of class-based views. You can define only some of them. Then you can override CBV methods as usual:

```
from django.shortcuts import redirect
from mapentity.views.generic import (
    MapEntityList, MapEntityLayer, MapEntityDetail,
    MapEntityFormat, MapEntityCreate, MapEntityUpdate, MapEntityDocument,
    MapEntityDelete, MapEntityViewSet)
from .models import Museum
from .serializers import MuseumSerializer

def home(request):
    return redirect('museum_list')

class MuseumList(MapEntityList):
    model = Museum
    columns = ['id', 'name']

class MuseumLayer(MapEntityLayer):
    model = Museum

class MuseumDetail(MapEntityDetail):
    model = Museum

class MuseumFormat(MapEntityFormat, MuseumList):
    pass

class MuseumCreate(MapEntityCreate):
    model = Museum

class MuseumUpdate(MapEntityUpdate):
```

(continues on next page)

(continued from previous page)

```
model = Museum

class MuseumDocument(MapEntityDocument):
    model = Museum

class MuseumDelete(MapEntityDelete):
    model = Museum

class MuseumViewSet(MapEntityViewSet):
    model = Museum
    serializer_class = MuseumSerializer
```

## 3.2 Filters

MapEntity allows you to define a set of filters which will be used to lookup geographical data. Create a file `filters.py` in your app:

```
from .models import Museum
from mapentity.filters import MapEntityFilterSet

class MuseumFilter(MapEntityFilterSet):
    class Meta:
        model = Museum
        fields = ('name', )
```

Then update `views.py` to use your custom filter in your custom views:

```
from .filters import MuseumFilter

class MuseumList(MapEntityList):
    model = Museum
    filterform = MuseumFilter
    columns = ['id', 'name']
```

## 3.3 Forms

Create a form for your Museum model:

```
from mapentity.forms import MapEntityForm
from .models import Museum

class MuseumForm(MapEntityForm):
    class Meta:
        model = Museum
        fields = ('name', )
```

Then update `views.py` to use your custom form in your custom views:

```
from .forms import MuseumForm

class MuseumCreate(MapEntityCreate):
    model = Museum
    form_class = MuseumForm

class MuseumUpdate(MapEntityUpdate):
    model = Museum
    form_class = MuseumForm
```

## 3.4 Templates

To display information accordingly to your Museum model, you can create a template in `main/templates/main`.

`museum_detail_attributes.html` can contain:

```
{% extends "mapentity/mapentity_detail_attributes.html" %}
{% load i18n mapentity_tags %}

{% block attributes %}
    <table class="table-striped table-bordered table">
        <tr>
            <th>{{ object|verbose:"name" }}</th>
            <td>{{ object.name }}</td>
        </tr>
    </table>
    {{ block.super }}
{% endblock attributes %}
```

You can override the detail view template for your Museum model by creating a `museum_detail.html` in the same directory as before.

## 3.5 Exports

There is another export system in MapEntity which use *Weasyprint* (<http://weasyprint.org/>).

Instead of using ODT templates, Weasyprint use HTML/CSS and export to PDF. Do not use this system if you need an ODT or DOC export.

**Although Weasyprint export only to PDF, there are multiple advantages to it, such as :**

- Use the power of HTML/CSS to generate your pages (far simpler than the ODT template)
- Use the Django template system to generate PDF content
- No longer need an instance of `convertit` to convert ODT to PDF and svg to png

To use MapEntity with Weasyprint, you just need to activate it in the `settings.py` of MapEntity.

Replace:

```
'MAPENTITY_WEASYPRINT': False,
```

by:

```
'MAPENTITY_WEASYPRINT': True,
```

If you want to include images that are not SVG or PNG, you will need to install GDK-PixBuf

```
sudo apt-get install libgdk-pixbuf2.0-dev
```

Now, you can customize the templates used to export your model in two different ways.

First one is to create a template for a model only.

In your museum project, you can override the CSS used to style the export by creating a file named `museum_detail_pdf.css` in `main/templates/main`. Refer to the CSS documentation and `mapentity_detail_pdf.css`.

Note that, in the `mapentity_detail_pdf.html`, the CSS file is included instead of linked to take advantage of the Django template generation.

Same as the CSS, you can override `mapentity_detail_pdf.html` by creating a file named `musuem_detail_pdf.html`. Again, refer to `mapentity_detail_pdf.html`.

If you create another model and need to override his template, the template should be of the form `templates/appname/modelname_detail_pdf.html` with `appname` the name of your Django app and `modelname` the name of your model.

The second way overrides these templates for all your models.

you need to create a sub-directory named `mapentity` in `main/templates`. Then you can create a file named `override_detail_pdf.html` (or `.css`) and it will be used for all your models if a specific template is not provided.

## 3.6 Settings

Attached files are downloaded by default by browser, with the following line, files will be opened in the browser :

```
MAPENTITY_CONFIG['SERVE_MEDIA_AS_ATTACHMENT'] = False
```

All layers colors can be customized from the settings. See [Leaflet reference](#) for vectorial layer style.

The styles are loaded in leaflet map in js and can be use with `window.SETTINGS.map.styles`

```
MAPENTITY_CONFIG['MAP_STYLES'][key] = {'color': 'red', 'weight': 5}
```

Or change just one parameter (the opacity for example) :

```
MAPENTITY_CONFIG['MAP_STYLES'][key]['opacity'] = 0.8
```

Paperclip medias (under `/paperclip/<app>_<model>/<pk>/<name>.*`) are protected by mapentity. We use `easy_thumbnail` to generate thumbnails of pictures. These files are generated with a new name with all the characteristics of the thumbnail generated (crop or not, width, height, etc...). These files need to be protected as the parent picture. We use a regex to find the parent's picture and all the permissions on this picture.

You can change the regex, for example if you need to add other behaviour with `easy_thumbnail` :

```
MAPENTITY_CONFIG['REGEX_PATH_ATTACHMENTS'] = r'\.\\d+x\\d+_q\\d+(_crop)?\\.(jpg|png|jpeg)$'
```



## DEVELOPMENT

Follow installation procedure, and then install development packages:

```
$ pip install -r dev-requirements.txt
```

### 4.1 Release

Set mapentity/VERSION

Set Changelog

Draft a new release on github



## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`