omniforms Documentation

Release 0.4.0

Omni Digital

Dec 18, 2018

Contents

Omni forms is a simple form builder written in Python and built on the Django web framework.

Omniforms ships with integrations for the Django and WagtailCMS admin interfaces allowing you to easily create and manage user facing forms for your django website.

CHAPTER 1

Project Aims

The Omniforms application aims to provide functionality by which user facing forms can be built and maintained through administration interfaces provided by the Django and Wagtail projects.

The application aims to be user friendly, developer friendly, extensible and pragmatic. All forms generated using this application are subclasses of either `django.forms.Form` or `django.forms.ModelForm` meaning developers are ultimately working with a forms library that is both familiar and predictable.

CHAPTER 2

Overview

The Omniforms application ships with 2 concrete `OmniForm` model classes:

- `omniforms.models.OmniForm`
- `omniforms.models.OmniModelForm`

Each of these models contains a `title` field (used for administration) and references to zero of more `OmniField` and `OmniFormHandler` model instances. In addition to these fields the `OmniModelForm` model holds a reference to the `ContentType` that the form manages.

Each of the concrete `OmniForm` models provides a `get_form_class` instance method which will generate and return an appropriate form class. This form classes fields will be built from all of the associated `OmniField` instances. In addition the form will be constructed in such a way that all associated `OmniFormHandler` instances will be run when the form instances `handle` method is called.

CHAPTER 3

Usage

3.1 Basic form example:

```
from django import forms
from omniforms.models import OmniForm

# Get an OmniForm model instance
omniform_instance = OmniForm.objects.get(pk=1)

# Generate a django form class from the OmniForm instance
form_class = omniform_instance.get_form_class()
assert issubclass(form_class, forms.Form)

# Work with the form class as per any other django form
form = form_class(request.POST)
if form.is_valid():
   # Call the forms 'handle' method (runs defined form handlers)
   form.handle()
```

3.2 Model form example:

```
from django import forms
from omniforms.models import OmniModelForm

# Get an OmniModelForm model instance
omniform_instance = OmniModelForm.objects.get(pk=1)

# Generate a django form class from the OmniModelForm instance
form_class = omniform_instance.get_form_class()
assert issubclass(form_class, forms.ModelForm)
```

(continues on next page)

(continued from previous page)

```python
# Work with the form class as per any other django form
form = form_class(request.POST)
if form.is_valid():
   # Call the forms 'handle' method (runs defined form handlers)
   form.handle()
```

The library does not intend to dictate how generated forms should be *used*. This is left as an exercise for developers.

CHAPTER 4

Compatibility

Omniforms is compatible with:

- Django 1.11
- Wagtail 1.11, 1.12, 1.13

Below are some useful links to help you get started with Omniforms.

CHAPTER 5

Index

5.1 Getting started

5.1.1 Installation

Omniforms is built on the Django web framework. This document assumes that you already have this installed. If not, you will need to install it:

Once you have django installed, the quickest way to install OmniForms is:

```
$ pip install omniforms
```

(`sudo` may be required if installing system-wide or without virtualenv)

Once Omniforms is installed into your python environment, you will need to add the omniforms library to the `INSTALLED_APPS` in your django projects settings file.

```
INSTALLED_APPS += ['omniforms']
```

Once you've done this you will need to run the database migrations that come bundled with the omniforms app.

```
python manage.py migrate
```

You should now be able to create and manage forms using the django admin interface.

5.2 Configuration

The OmniForms application can be configured in a number of different ways:

5.2.1 OmniModelForm permitted content types

You may not want administrators to be able to create forms for all different types of content in your database.

There are 2 different ways of restricting the types of content that can be associated with model forms created through the admin interface:

OMNI_FORMS_CONTENT_TYPES

It is possible to define specific apps and/or models which can be used by the omniforms app using the `OMNI_FORMS_CONTENT_TYPES` setting.

The following configuration would allow _any_ of the models in the app `foo`, and the `modelone` and `modeltwo` models within the `bar` app, to be used.

```
OMNI_FORMS_CONTENT_TYPES = [
    {'app_label': 'foo'},
    {'app_label': 'bar', 'model': 'modelone'},
    {'app_label': 'bar', 'model': 'modeltwo'},
]
```

If the `OMNI_FORMS_CONTENT_TYPES` setting is not defined it will default to None and the `OMNI_FORMS_EXCLUDED_CONTENT_TYPES` setting will be used instead.

OMNI_FORMS_EXCLUDED_CONTENT_TYPES

It is possible to prevent model forms from being created for specific apps or specific models using the `OMNI_FORMS_EXCLUDED_CONTENT_TYPES` setting.

The following configuration would prevent forms being created for *any* of the models in the app `foo`, and for the `modelone` and `modeltwo` models within the `bar` app.

```
OMNI_FORMS_EXCLUDED_CONTENT_TYPES = [
    {'app_label': 'foo'},
    {'app_label': 'bar', 'model': 'modelone'},
    {'app_label': 'bar', 'model': 'modeltwo'},
]
```

This setting defaults to the following if not overridden:

```
OMNI_FORMS_EXCLUDED_CONTENT_TYPES = [
    {'app_label': 'omniforms'}
]
```

This will prevent administrators from creating forms for managing omniforms. It's worth mentioning that allowing administrators to do this represents a potential security risk and should be avoided. As such, if you need to define your own `OMNI_FORMS_EXCLUDED_CONTENT_TYPES` setting it would be wise to exclude all `omniforms` models as shown above.

OMNI_FORMS_CUSTOM_FIELD_MAPPING

Although the omniforms app accounts for the majority of use cases, you may have models that use custom model fields. Omniforms will not be able to map these model fields to their corresponding form fields. As such you will need to provide a custom field mapping dictionary using the `OMNI_FORMS_CUSTOM_FIELD_MAPPING` setting.

- Each key within the mapping dictionary must be a string (python dotted import path) to a model field class.

- Each value within the mapping dictionary must be a string (python dotted import path) to an OmniField subclass.

For example, you can map a TagField to an OmniCharField model instance using the following configuration:

```
OMNI_FORMS_CUSTOM_FIELD_MAPPING = {
    'taggit.TagField': 'omniforms.models.OmniCharField',
}
```

With this configuration any instances of `taggit.TagField` on your models will be represented as `django.forms.CharField` instances in `OmniModelForm` instances created via the admin.

This mechanism also allows custom `OmniField` model classes to be defined and used on a per-model-field basis.

```
OMNI_FORMS_CUSTOM_FIELD_MAPPING = {
    'taggit.TagField': 'my_app.MySuperOmniField',
}
```

It is important to note that the dictionary values defined within the `OMNI_FORMS_CUSTOM_FIELD_MAPPING` **MUST** be subclasses of `omniforms.models.OmniField`. If you attempt to register fields that do not subclass `omniforms.models.OmniField` an `ImproperlyConfigured` exception will be raised by the application.

5.3 Bundled Handlers

Omniforms currently ships with 3 form handlers for use in your application.

5.3.1 Send Static Email

This form handler allows administrators to send an email to a set of predefined email addresses on successful form submission. In addition to the standard fields that every handler has, this handler allows the administrator to specify the following fields:

- `subject`: The subject for outgoing emails
- `recipients`: A list of email addresses the email will be sent to (one address per line)
- `template`: The email template. This template string will be rendered using djangos template rendering library. The forms cleaned data will be made available to the template rendering context meaning it is possible to render submitted form data in the template.

It is also worth noting that any files uploaded via the form will be attached to the outbound emails.

5.3.2 Send Email Confirmation

This form handler works in an almost identical way to the `Send Static Email` handler. However, rather than allowing a static list of recipients to be defined, the handler allows the administrator to select an `OmniEmailField` instance from the associated form that holds the recipient email address.

For example, a job application form may hold a field for the applicant to enter their email address. This handler would allow you to select that field and send a confirmation email to the applicant on form submission.

5.3.3 Save Data

This form handler is designed to emulate a model forms `save` method and allows submitted form data to be persisted to the database if valid.

This handler may only be attached to forms that:

- Are `OmniModelForm` instances;
- Have all of the models `required` fields configured correctly

5.4 Extending

It is possible to add custom Field and Handler models to your application, therefore allowing you to extend the basic functionality provided by the `OmniForms` library. Once you have done so, the omniforms library should automatically find these models and make it possible to create and associate these fields and handlers with your OmniForm model instances.

5.4.1 Fields

It is possible to add custom field types to your application that can be used by the omniforms library. Doing so should be as simple as adding a model class that subclasses `omniforms.models.OmniField` and provides a few class attributes used by the omniforms library.

At the very minimum, a custom Field model might look something like this:

```
from omniforms.models import OmniField

class MyCustomOmniField(OmniField):
    """
    Custom OmniField model
    """
    FIELD_CLASS = 'django.forms.CharField'
    FORM_WIDGETS = (
        'django.forms.widgets.TextInput',
        'django.forms.widgets.Textarea',
        'django.forms.widgets.PasswordInput',
        'myapp.widgets.MyCustomFormWidget',
    )
```

Of course, it is also possible to implement more complex `OmniField` models. If you are interested in doing so it may be wise to look at `omniforms.models.OmniChoiceField` and `omniforms.models.OmniMultipleChoiceField`.

The `FIELD_CLASS` attribute must be a python dotted import path to the form field class that this OmniField *will* use in the generated form. An OmniField modelo may only have one `FIELD_CLASS` (i.e. you cannot assign it a list or tuple of field classes) for the user to pick from.

The `FORM_WIDGETS` attribute must be a list or tuple containing a series of python dotted import paths to potential form widget classes that this OmniField *could* use in generated forms. This list or tuple must contain at least one potential widget but could have many. If there is more than one widget listed for a given OmniField model, the administrator should be able to select the type of widget they would like to use for a given field at the point of creation within the admin environment. If only one type of widget is provided, this option is effectively removed from the form and pre-selected for the admin user.

5.4.2 Handlers

It is possible to add custom form handler types to your application that can be used by the omniforms library. Doing so should be as simple as adding a model class that subclasses `omniforms.models.OmniFormHandler` and implements a `handle` method.

The handle method should accept one positional argument, form, which will be the valid form instance.

At the very minimum, a custom Handler model might look something like this:

```
from omniforms.models import OmniFormHandler

class MyCustomOmniFormHandler(OmniFormHandler):
    """
    Custom OmniFormHandler model
    """
    def handle(self, form):
        """
        Method for handling the valid form action

        :param form: The validated form instance this handler is attached to
        """
        do_something_with(form.cleaned_data)
```

It is worth noting that you should never call the forms `handle` or `save` (for model forms) methods within the `OmniFormHandler.handle` method. Doing so will cause the forms handlers to be run repeatedly until python reaches its recursion limit.

Omniforms ships with a handler - `OmniFormSaveInstanceHandler` - (to only be used with `OmniModelForm` instances) for saving model instances. This handler does not call the forms `save` method directly. Instead it calls the `django.forms.models.save_instance` function which ensures that the form data is persisted to the database correctly, but avoids the issue of the forms handlers being run repeatedly.

This approach allows us to set up a series of form handlers that will run one after the other when the forms `handle` or `save` method is called. For example, it is theoretically possible to implement and configure handlers to do the following:

- Create a NewsLetterSubscription (model instance);
- Send an email to the marketing department containing the subscribers data;
- Post the users data to a CRM application;
- Send an email to the subscriber confirming their subscription;

5.5 Wagtail integration

5.5.1 Installation

In addition to the steps outlined in the 'Getting started' section you will need to add the `omniforms.wagtail` app to your projects `INSTALLED_APPS` setting.

```
INSTALLED_APPS += ['omniforms', 'omniforms.wagtail']
```

Once you've done this you will need to run the database migrations that come bundled with the omniforms app.

```
python manage.py migrate
```

You should now be able to create and manage forms using the wagtail admin interface.

5.5.2 Custom admin forms

In some cases it may be desirable to customize the form class that wagtail uses in the admin for managing a custom related model type (i.e. an `OmniForm` Field or `OmniFormHandler` model). If this is desirable, the custom form class needs to be assigned to a `base_form_class` property on the model. e.g.

```
class MyOmniFieldForm(forms.ModelForm):
    def clean_name(self, value):
        if value.contains("something"):
            raise ValidationError("The field name cannot contain the word 'something'
↪")
        return value

class MyOmniField(OmniField):
    base_form_class = MyOmniFieldForm
```

It is worth noting that the `base_form_class` *must* subclass `django.forms.ModelForm`. You do *not* need to specify the model that the form is for (using the forms meta class) as this will be generated dynamically when the form class is created.

5.5.3 Locking forms

It may be desirable for forms to be locked under certain conditions. For example, if a form has been set up for data collection, and has already collected data, you may want to prevent the form from being modified or deleted. For this purpose we have added a custom wagtail hook which can be used to implement logic to prevent the form from being edited further.

The hook name is `omniform_permission_check` and is registered like any other wagtail hook. The hook takes 2 positional arguments:

- `action`: The type of action being performed on the form (clone, update, delete);
- `instance`: The form instance

Example

```
from wagtail.wagtailcore import hooks

@hooks.register('omniform_permission_check')
def lock_form(action, form):
    if action in ['update', 'delete'] and form.some_relationship.count() > 0:
        raise PermissionDenied
```