# django-floppyforms Documentation

***Release dev***

**Bruno Renié**

**Aug 06, 2017**

# Contents

**django-floppyforms** is an application that gives you full control of the output of forms rendering. The forms API and features are exactly the same as Django's, the key difference is that fields and widgets are rendered in templates instead of using string interpolation, giving you full control of the output using Django templates.

The widgets API allows you to customize and extend the widgets behaviour, making it very easy to define custom widgets. The default widgets are very similar to the default Django widgets, except that they implement some nice features of HTML5 forms, such as the `placeholder` and `required` attribute, as well as the new `<input>` types. For more information, read this if you haven't yet.

The form rendering API is a set of template tags that lets you render forms using custom layouts. This is very similar to Django's `as_p`, `as_ul` or `as_table`, except that you can customize and add layouts to your convenience.

The source code is hosted on github.

CHAPTER 1

# Installation

As a requirement of django-floppyforms, you will need to have Django in version 1.4 or higher installed and use Python 2.7 or newer. Python >=3.3 and PyPy are supported!

Two-step process to install django-floppyforms:

- `pip install django-floppyforms`
- Add `'floppyforms'` to your `INSTALLED_APPS`

When you're done you can jump to the *usage* section. For the impatient reader, there's also an *examples* section.

CHAPTER 2

# Using `django-floppyforms`

## Usage

### Forms

Floppyforms are supposed to work just like Django forms:

```
import floppyforms as forms

class ProfileForm(forms.Form):
    name = forms.CharField()
    email = forms.EmailField()
    url = forms.URLField()
```

With some template code:

```
<form method="post" action="/some-action/">
    {% csrf_token %}
    {{ form.as_p }}
    <p><input type="submit" value="Yay!"></p>
</form>
```

The form will be rendered using the `floppyforms/layouts/p.html` template. See the *documentation about layouts* for details.

Each field has a default widget and widgets are rendered using templates.

Default templates are provided and their output is relatively similar to Django widgets, with a few *minor differences*:

- HTML5 `<input>` types are supported: `url`, `email`, `date`, `datetime`, `time`, `number`, `range`, `search`, `color`, `tel`.
- The `required` and `placeholder` attributes are also supported.

Widgets are rendered with the following context variables:

- `hidden`: set to `True` if the field is hidden.
- `required`: set to `True` if the field is required.
- `type`: the input type. Can be `text`, `password`, etc. etc.
- `name`: the name of the input.
- `attrs`: the dictionnary passed as a keyword argument to the widget. It contains the `id` attribute of the widget by default.

Each widget has a `template_name` attribute which points to the template to use when rendering the widget. A basic template for an `<input>` widget may look like:

```
<input {% for key, val in attrs.items %}
         {{ key }}="{{ val }}"
       {% endfor %}
       type="{{ type }}"
       name="{{ name }}"
       {% if value %}value="{{ value }}"{% endif %}>
```

The default floppyforms template for an `<input>` widget is slightly more complex.

Some widgets may provide extra context variables and extra attributes:

| Widget | Extra context | Extra `attrs` |
|---|---|---|
| Textarea | | `rows`, `cols` |
| NumberInput | | `min`, `max`, `step` |
| RangeInput | | `min`, `max`, `step` |
| Select | `optgroups`, `multiple` | |
| RadioSelect | `optgroups`, `multiple` | |
| NullBooleanSelect | `optgroups`, `multiple` | |
| SelectMultiple | `optgroups`, `multiple` (`True`) | |
| CheckboxSelectMultiple | `optgroups`, `multiple` (`True`) | |

Furthermore, you can specify custom `attrs` during widget definition. For instance, with a field created this way:

```
bar = forms.EmailField(widget=forms.EmailInput(attrs={'placeholder': 'john@example.com
↪'}))
```

Then the `placeholder` variable is available in the `attrs` template variable.

## ModelForms

You can use `ModelForms` with floppyforms as you would use a ordinary django `ModelForm`. Here is an example showing it for a basic `Profile` model:

```
class Profile(models.Model):
    name = models.CharField(max_length=255)
    url = models.URLField()
```

Now create a `ModelForm` using floppyforms:

```
import floppyforms.__future__ as forms

class ProfileForm(forms.ModelForm):
    class Meta:
        model = Profile
        fields = ('name', 'url')
```

The `ProfileForm` will now have form fields for all the model fields. So there will be a `floppyforms.CharField` used for the `Profile.name` model field and a `floppyforms.URLField` for `Profile.url`.

---

**Note:** Please note that you have to import from `floppyforms.__future__` to use this feature. Here is why:

This behaviour changed in version 1.2 of **django-floppyforms**. Before, no alterations were made to the widgets of a `ModelForm`. So you had to take care of assigning the floppyforms widgets to the django form fields yourself to use the template based rendering provided by floppyforms. Here is an example of how you would have done it with django-floppyforms 1.1 and earlier:

```
import floppyforms as forms

class ProfileForm(forms.ModelForm):
    class Meta:
        model = Profile
        fields = ('name', 'url')
        widgets = {
            'name': forms.TextInput,
            'url': forms.URLInput,
        }
```

Since the change is backwards incompatible, we decided to provide a deprecation path. If you create a `ModelForm` with django-floppyforms 1.2 and use `import floppyforms as forms` as the import you will get the old behaviour and you will see a `DeprecationWarning`.

To use the new behaviour, you can use `import floppyforms.__future__ as forms` as the import.

Please make sure to test your code if your modelforms work still as expected with the new behaviour. The old version's behaviour will be removed completely with django-floppyforms 1.4.

---

# Provided widgets

## Default widgets for form fields

The first column represents the name of a `django.forms` field. FloppyForms aims to implement all the Django fields with the same class name, in the `floppyforms` namespace.

| Fields | Widgets | Specificities |
|---|---|---|
| BooleanField | CheckboxInput | |
| CharField | TextInput | |
| ComboField | TextInput | |
| ChoiceField | Select | |
| TypedChoiceField | Select | |
| FilePathField | Select | |
| ModelChoiceField | Select | |
| DateField | DateInput | <input type="date"> |
| DateTimeField | DateTimeInput | <input type="datetime"> |
| DecimalField | NumberInput | <input type="number"> |
| EmailField | EmailInput | <input type="email"> |
| FileField | ClearableFileInput | |
| FloatField | NumberInput | <input type="number"> |
| ImageField | ClearableFileInput | |
| IntegerField | NumberInput | <input type="number"> |
| MultipleChoiceField | SelectMultiple | |
| TypedMultipleChoiceField | SelectMultiple | |
| ModelMultipleChoiceField | SelectMultiple | |
| NullBooleanField | NullBooleanSelect | |
| TimeField | TimeInput | <input type="time"> |
| URLField | URLInput | <input type="url"> |
| SlugField | SlugInput | <input pattern="[-\w]+"> |
| RegexField | TextInput | <input [pattern=...]> |
| IPAddressField | IPAddressInput | <input pattern=...> |
| GenericIPAddressField | TextInput | |
| MultiValueField | None (*abstract*) | |
| SplitDateTimeField | SplitDateTimeWidget | |

**Note:** Textarea

The `Textarea` widget renders a `<textarea>` HTML element and is available with django-floppyforms. It doesn't appear on the table above since no field has it as a default widget.

**Note:** RegexField

In Django, `RegexField` takes a required `regex` argument. The version shipped in floppyforms also takes an optional `js_regex` argument, for client-side validation of the regex. The `js_regex` must be a regex written in javascript syntax. Example:

```
class RegexForm(forms.Form):
    re_field = forms.RegexField(r'^\d{3}-[a-z]+$',  # regex
                                '\d{3}-[a-z]+')     # js_regex
```

If you don't provide the `js_regex` argument, there will be no client-side validation of the field. Although the the two versions of the regex may be identical, the distinction allows you to pass compiled regexes as a `regex` argument.

## Extra widgets

Django provides "extra" widgets in `django.forms.extras.widgets`. In fact, a single extra widget is implemented: `SelectDateWidget`. The template-based version is available under the `floppyforms.`

`SelectDateWidget` name.

By defaut, this widget with split the date into three select (year, month and day). You can overload the template so that it is displayed in a different order or with 3 inputs:

```
<input type="text" name="{{ day_field }}" value="{{ day_val|stringformat:"02d" }}" id=
↪"{{ day_id }}"{% for attr in attrs.items %} {{ attr.0 }}="{{ attr.1 }}"{% endfor %}␣
↪/>
<input type="text" name="{{ month_field }}" value="{{ month_val|stringformat:"02d" }}
↪" id="{{ month_id }}"{% for attr in attrs.items %} {{ attr.0 }}="{{ attr.1 }}"{%␣
↪endfor %}/>
<input type="text" name="{{ year_field }}" value="{{ year_val|stringformat:"04d" }}"␣
↪id="{{ year_id }}"{% for attr in attrs.items %} {{ attr.0 }}="{{ attr.1 }}"{%␣
↪endfor %}/>
```

### Other (HTML5) widgets

Some HTML5 widgets are also provided, although browser support may not be there yet:

- `SearchInput`: a widget that renders `<input type="search">`.
- `ColorInput`: `<input type="color">` (currently only supported by Chrome 20+ and Opera 11+).
- `RangeInput`: `<input type="range">`, for sliders instead of spinboxes for numbers.
- `PhoneNumberInput`: `<input type="tel">`. For phone numbers.

You can easily check browser support for the various (HTML5) input types on caniuse.com.

## Customization

### Override default templates

Widgets have a `template_name` attribute that points to the template that is used when rendering the form. Default templates are provided for all *built-in widgets*. In most cases the default implementation of these templates have no specific behaviour and simply inherit from `floppyforms/input.html`. They are provided mainly to give an easy way for a site-wide customization of how a specifig widget is rendered.

You can easily override these templates in your project-level `TEMPLATE_DIRS`, assuming they take precedence over app-level templates.

### Custom widgets with custom templates

If you want to override the rendering behaviour only for a few widgets, you can extend a `Widget` class from Floppy-Forms and override the `template_name` attribute:

```
import floppyforms as forms

class OtherEmailInput(forms.EmailInput):
    template_name = 'path/to/other_email.html'
```

Then, the output can be customized in `other_email.html`:

```
<input type="email"
       name="{{ name }}"
       id="{{ attrs.id }}"
       placeholder="john@example.com"
       {% if value %}value="{{ value }}"{% endif %}>
```

Here we have a hardcoded placeholder without needing to instantiate the widget with an `attrs` dictionary:

```
class EmailForm(forms.Form):
    email = forms.EmailField(widget=OtherEmailInput())
```

You can also customize the `template_name` without subclassing, by passing it as an argument when instantiating the widget:

```
class EmailForm(forms.Form):
    email = forms.EmailField(
        widget=forms.EmailInput(template_name='path/to/other_email.html'))
```

For advanced use, you can even customize the template used per-render, by passing a `template_name` argument to the widget's `render()` method.

## Adding more template variables

There is also a way to add extra context. This is done by subclassing the widget class and extending the `get_context()` method:

```
class OtherEmailInput(forms.EmailInput):
    template_name = 'path/to/other.html'

    def get_context(self, name, value, attrs):
        ctx = super(OtherEmailInput, self).get_context(name, value, attrs)
        ctx['foo'] = 'bar'
        return ctx
```

And then the `other.html` template can make use of the `{{ foo }}` context variable.

`get_context()` takes `name`, `value` and `attrs` as arguments, except for all `Select` widgets which take an additional `choices` argument.

In case you don't need the arguments passed to `get_context()`, you can extend `get_context_data()` which doesn't take any arguments:

```
class EmailInput(forms.EmailInput):
    def get_context_data(self):
        ctx = super(EmailInput, self).get_context_data()
        ctx.update({
            'placeholder': 'hello@example.com',
        })
        return ctx
```

## Altering the widget's `attrs`

All widget attibutes except for `type`, `name`, `value` and `required` are put in the `attrs` context variable, which you can extend in `get_context()`:

```
def get_context(self, name, value, attrs):
    ctx = super(MyWidget, self).get_context(name, value, attrs)
    ctx['attrs']['class'] = 'mywidget'
    return ctx
```

This will render the widget with an additional `class="mywidget"` attribute.

If you want only the attribute's key to be rendered, set it to `True`:

```
def get_context(self, name, value, attrs):
    ctx = super(MyWidget, self).get_context(name, value, attrs)
    ctx['attrs']['awesome'] = True
    return ctx
```

This will simply add `awesome` as a key-only attribute.

## Widgets reference

For each widgets, the default class attributes.

**class** `floppyforms.widgets.`**Input**

**datalist**
A list of possible values, which will be rendered as a `<datalist>` element tied to the input. Note that the list of options passed as `datalist` elements are only **suggestions** and are not related to form validation.

**template_name**
A path to a template that should be used to render this widget. You can change the template name per instance by passing in a keyword argument called `template_name`. This will override the default that is set by the widget class. You can also change the template used for rendering by an argument to the `Input.render()` method. See more about exchanging the templates in the *documentation about customization*.

**class** `floppyforms.widgets.`**TextInput**

**template_name**
`'floppyforms/text.html'`

**input_type**
`text`

**class** `floppyforms.widgets.`**PasswordInput**

**template_name**
`'floppyforms/password.html'`

**input_type**
`password`

**class** `floppyforms.widgets.`**HiddenInput**

**template_name**
`'floppyforms/hidden.html'`

**input_type**
hidden

**class** floppyforms.widgets.**SlugInput**

**template_name**
'floppyforms/slug.html'

**input_type**
text

An text input that renders as <input pattern="[-\w]+" ...> for client-side validation of the slug.

**class** floppyforms.widgets.**IPAddressInput**

**template_name**
'floppyforms/ipaddress.html'

**input_type**
text

An text input that renders as <input pattern="..." ...> for client-side validation. The pattern checks that the entered value is a valid IPv4 address.

**class** floppyforms.widgets.**FileInput**

**template_name**
'floppyforms/file.html'

**input_type**
file

**class** floppyforms.widgets.**ClearableFileInput**

**template_name**
'floppyforms/clearable_input.html'

**input_type**
file

**initial_text**
_('Currently')

**input_text**
_('Change')

**clear_checkbox_label**
_('Clear')

The initial_text, input_text and clear_checkbox_label attributes are provided in the template context.

**class** floppyforms.widgets.**EmailInput**

**template_name**
'floppyforms/email.html'

**input_type**
email

**class** floppyforms.widgets.**URLInput**

**template_name**
'floppyforms/url.html'

**input_type**
url

**class** floppyforms.widgets.**SearchInput**

**template_name**
'floppyforms/search.html'

**input_type**
search

**class** floppyforms.widgets.**ColorInput**

**template_name**
'floppyforms/color.html'

**input_type**
color

**class** floppyforms.widgets.**PhoneNumberInput**

**template_name**
'floppyforms/phonenumber.html'

**input_type**
tel

**class** floppyforms.widgets.**DateInput**

**template_name**
'floppyforms/date.html'

**input_type**
date

A widget that renders as `<input type="date" value="...">`. Value is rendered in ISO-8601 format (i.e. `YYYY-MM-DD`) regardless of localization settings.

**class** floppyforms.widgets.**DateTimeInput**

**template_name**
'floppyforms/datetime.html'

**input_type**
datetime

**class** floppyforms.widgets.**TimeInput**

**template_name**
'floppyforms/time.html'

**input_type**
time

**class** floppyforms.widgets.**NumberInput**

**template_name**
'floppyforms/number.html'

**input_type**
number

**min**
None

**max**
None

**step**
None

min, max and step are available in the attrs template variable if they are not None.

**class** floppyforms.widgets.**RangeInput**

NumberInput.**template_name**
'floppyforms/range.html'

**input_type**
range

**min**
None

**max**
None

**step**
None

min, max and step are available in the attrs template variable if they are not None.

**class** floppyforms.widgets.**Textarea**

**template_name**
'floppyforms/textarea.html'

**rows**
10

**cols**
40

rows and cols are available in the attrs variable.

**class** floppyforms.widgets.**CheckboxInput**

**template_name**
'floppyforms/checkbox.html'

**input_type**
checkbox

**class** floppyforms.widgets.**Select**

**template_name**
'floppyforms/select.html'

**class** floppyforms.widgets.**NullBooleanSelect**

**template_name**
'floppyforms/select.html'

**class** floppyforms.widgets.**RadioSelect**

**template_name**
'floppyforms/radio.html'

**class** floppyforms.widgets.**SelectMultiple**

**template_name**
'floppyforms/select_multiple.html'

**class** floppyforms.widgets.**CheckboxSelectMultiple**

**template_name**
'floppyforms/checkbox_select.html'

**class** floppyforms.widgets.**MultiWidget**
The same as django.forms.widgets.MultiWidget. The rendering can be customized by overriding format_output, which joins all the rendered widgets.

**class** floppyforms.widgets.**SplitDateTimeWidget**
Displays a DateInput and a TimeInput side by side.

**class** floppyforms.widgets.**MultipleHiddenInput**
A multiple <input type="hidden"> for fields that have several values.

**class** floppyforms.widgets.**SelectDateWidget**
A widget that displays three <select> boxes, for the year, the month and the date.

Available context:

- year_field: the name for the year's <select> box.
- month_field: the name for the month's <select> box.
- day_field: the name for the day's <select> box.

**template_name**
The template used to render the widget. Default: 'floppyforms/select_date.html'.

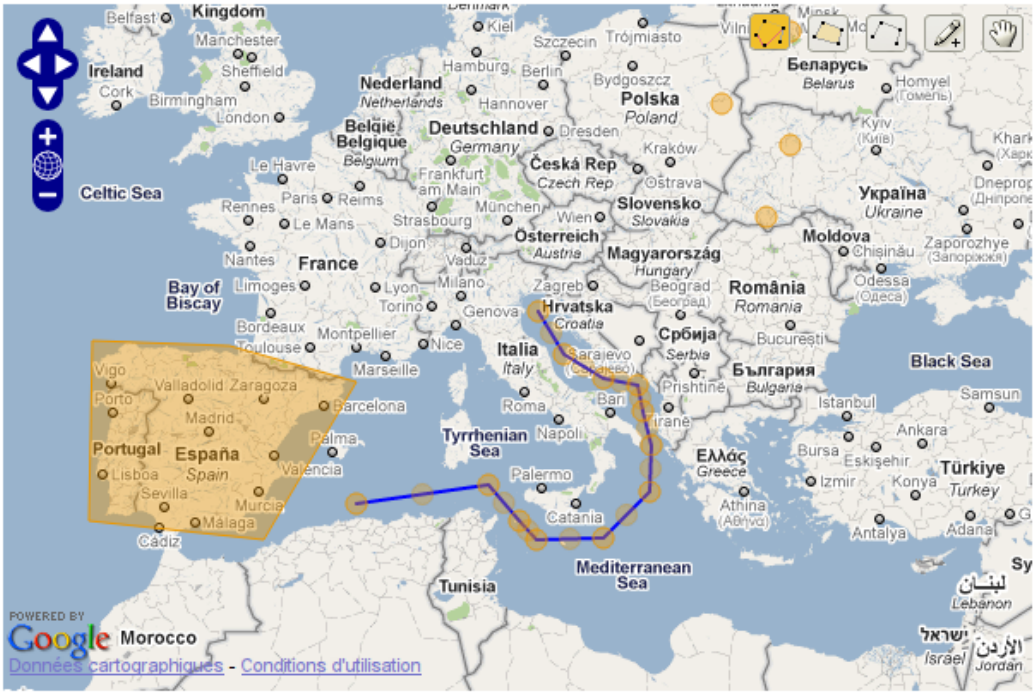
**none_value**
A tuple representing the value to display when there is no initial value. Default: (0, '---').

**day_field**
The way the day field's name is derived from the widget's name. Default: '%s_day'.

**month_field**
The way the month field's name is derived. Default: '%s_month'.

**year_field**
The way the year field's name is derived. Default: '%s_year'.

# GeoDjango widgets




django-floppyforms provides fields and rich widgets for easy manipulation of GEOS geometry fields. All geometry types are supported thanks to OpenLayers and a custom WKT parser/serializer implementing some Django-specific tweaks.

---

**Note:** Since GeoDjango doesn't provide any rich widget out of the box (except for the admin), the API described here is not trying to look like any existing API in GeoDjango.

The geographic fields and widgets are provided under the `floppyforms.gis` namespace.

---

## Setting up

To make sure you're ready to use the geographic widgets, follow the installation instructions for GeoDjango closely. You need to have `'django.contrib.gis'` in your `INSTALLED_APPS` setting.

Next, you need to serve the javascript library provided by django-floppyforms (located in `floppyforms/static/floppyforms/js/MapWidget.js`).

You might want to use `django.contrib.staticfiles`, so that the javascript library will be picked up automatically and gets served by the development server. Just make sure you run `manage.py collectstatic` once you deploy your project.

## Widget types

django-floppyforms provides **base widgets** and **geometry-specific widgets**:

- **base widgets** are in charge of rendering a map from a specific map provider (Metacarta, Google Maps, OpenStreetMap...). They are not aware of the type of geometry, they need to be complemented by geometry-specific widgets.
- **geometry-specific widgets** are here to make base widgets aware of the type of geometry to edit: is the geometry a point? A polygon? A collection? Geometry-specific widgets provides these information so that the corresponding controls are activated.

To get a fully working geometry widget, you need to define a class that inherits from a base widget (to specify the map provider) and a geometry-specific widget (to specify the type of geometry you want to create). Here is a quick example:

```
import floppyforms as forms

class PointWidget(forms.gis.PointWidget, forms.gis.BaseOsmWidget):
    pass
```

Here `BaseOsmWidget` is the base widget (i.e. I want to see an OpenStreetMap) and `PointWidget` is the geometry-specific widget (i.e. I want to draw a point on the map).

### Base Widgets

The following base widgets are provided:

- `BaseMetacartaWidget`: this base widget renders a map using the Vector Level 0 map from Metacarta.
- `BaseOsmWidget`: this base widget renders a map using OpenStreetMap.
- `BaseGMapWidget`: this base widget renders a map using the Google Maps API. It uses the v3 javascript API and requires an API Key (which can be obtained at Google Developers). Subclasses must set the attribute `google_maps_api_key`, otherwise the map will fail to load.

```
import floppyforms as forms

class PointWidget(forms.gis.PointWidget, forms.gis.BaseGMapWidget):
    google_maps_api_key = 'YOUR-GOOGLE-MAPS-API-KEY-HERE'
```

### Geometry-specific widgets

For each geographic model field, here are the corresponding form fields and form widgets provided by django-floppyforms:

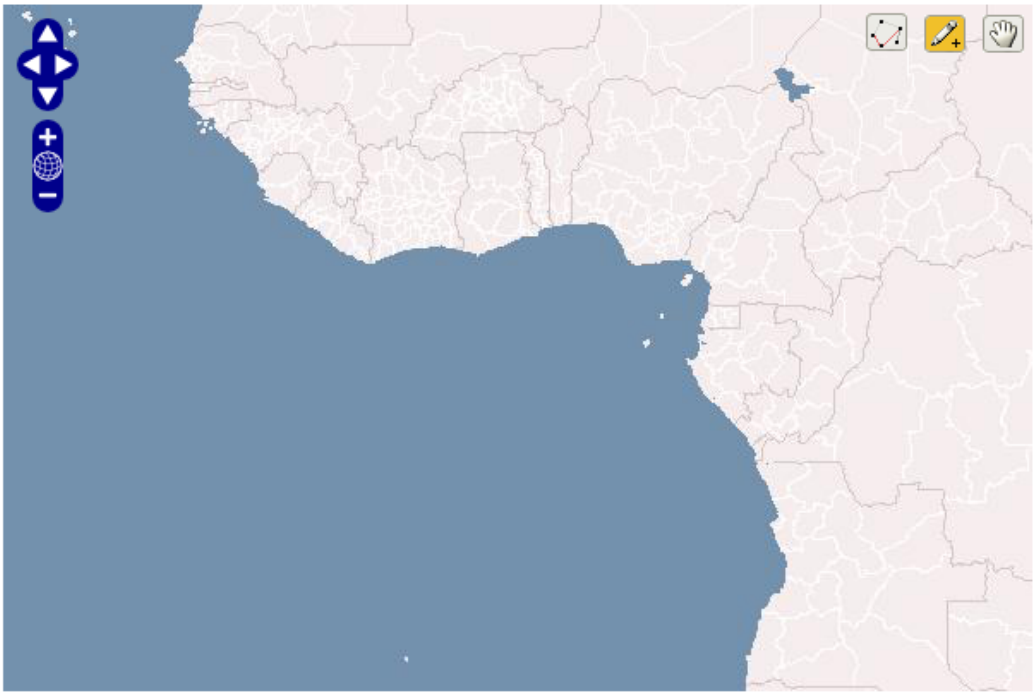
| GeoDjango model field | Floppyforms form field | Floppyforms form widget |
|---|---|---|
| PointField | PointField | PointWidget |
| MultiPointField | MultiPointField | MultiPointWidget |
| LineStringField | LineStringField | LineStringWidget |
| MultiLineStringField | MultiLineStringField | MultiLineStringWidget |
| PolygonField | PolygonField | PolygonWidget |
| MultiPolygonField | MultiPolygonField | MultiPolygonWidget |
| GeometryField | GeometryField | GeometryWidget |
| GeometryCollectionField | GeometryCollectionField | GeometryCollectionWidget |

Each form field has a default form widget, using the corresponding geometry-specific widget and the Metacarta base widget. A form defined using nothing more than floppyforms fields will be displayed using the Metacarta WMS map service. For instance:

```
# forms.py
import floppyforms as forms

class GeoForm(forms.Form):
    point = forms.gis.PointField()
```

```
{# template.html #}
<html>
  <head>
    {{ form.media }}
  </head>
  <body>
    <form method="post" action="/some-url/">
      {% csrf_token %}
      {{ form.as_p }}
      <p><input type="submit" value="Submit"></p>
    </form>
  </body>
</html>
```

And the result will looks like this:



Delete all Features

## Customization

The philosophy of this widgets library is to avoid building a complex layer of abstraction that would generate some javascript / OpenLayers code out of Python class attributes or methods. Everything that can be done in the template or JavaScript code should be done there.

Therefore there are few options to customize the map on the widget classes. Only basic customization can be made in python, the rest should be done in the templates using the JavaScript library.

### Widget attributes and arguments

The following attributes can be set on the widget class:

- `map_width`: the width of the map, in pixels. Default: 600.
- `map_height`: the height of the map, in pixels. Default: 400.
- `map_srid`: the SRID to use on the map. When existing geometries are edited, they are transformed to this SRID. The javascript code doesn't transform geometries so it's important to set this to the SRID used with your map provider. Default: 4326.
- `display_wkt`: whether to show the `textarea` in which the geometries are serialized. Usually useful for debugging. Default: `False`.

These options can be set as class attributes or passed into the `attrs` dictionnary used when instantiating a widget. The following snippets are equivalent:

```
import floppyforms as forms

class OsmPointWidget(forms.gis.PointWidget, forms.gis.BaseOsmWidget):
    pass

class CustomPointWidget(OsmPointWidget):
    map_width = 1000
    map_height = 700

class GeoForm(forms.Form):
    point = forms.gis.PointField(widget=CustomPointWidget)
```

and:

```
import floppyforms as forms

class OsmPointWidget(forms.gis.PointWidget, forms.gis.BaseOsmWidget):
    pass

class GeoForm(forms.Form):
    point = forms.gis.PointField(widget=OsmPointWidget(attrs={
        'map_width': 1000,
        'map_height': 700,
    }))
```

Of course, the traditional `template_name` class attribute is also supported.

### Template context

The following variables are available in the template context:

- `ADMIN_MEDIA_PREFIX`: this setting, yes. It's useful to display some icons that are missing in OpenLayers. **Deprecated, please switch to use the staticfiles machinery**
- `LANGUAGE_BIDI`: the current locale direction.
- `attrs`: the traditional attrs dictionnary. This is the attrs dict for a `textarea` widget, it contains the `id`, `cols` and `rows` attributes.
- `display_wkt`: the value from the widget class.
- `geom_type`: the OGR geometry type for the geometry being edited.

- `hidden`: set to `False`, textareas can't be hidden.
- `is_collection`: whether the geometry is a collection.
- `is_linestring`: whether the geometry is a line string.
- `is_point`: whether the geometry is a point.
- `is_polygon`: whether the geometry is a polygon.
- `map_width`: the width, from the class attribute.
- `map_height`: the height, from the class attribute.
- `map_srid`: the SRID, from the class attribute.
- `module`: the name to use for the javascript object that contains the map.
- `name`: the name of the field.
- `required`: True if the field is required.
- `type`: the input type, `None` in this case.
- `value`: the WKT serialization of the geometry, expressed in the projection defined by `map_srid`.

### Javascript library

The javascript library provided by django-floppyforms relies on OpenLayers. It creates a map container based on a series of options. A minimal widget can be created like this:

```
var options = {
    geom_type: OpenLayers.Geometry.Point,
    id: 'id_point',
    is_point: true,
    map_id: 'point_map',
    name: 'My awesome point'
};
var point_map = new MapWidget(options);
```

With these options, you need in your HTML code a `<textarea id="id_point">` and an empty `<div id="point_map">`. The size of the map can be set by styling the div with CSS.

Generally you don't have to touch the `geom_type`, `id`, `is_point`, `map_id` and `name` attributes: django-floppyforms generates them for you. However, the template structure makes it easy to specify some custom options. The base template defines a `map_options` and an `options` block. They can be altered like this (let's say we want to re-implement the Google Maps base widget):

```
# forms.py
from django.template.defaultfilters import safe
import floppyforms as forms

class BaseGMapWidget(forms.gis.BaseGeometryWidget):
    map_srid = 900913  # Use the google projection
    template_name = 'forms/google_map.html'

    class Media:
        js = (
            'http://openlayers.org/dev/OpenLayers.js',
            'floppyforms/js/MapWidget.js',

            # Needs safe() because the ampersand (&):
```

```
            safe('http://maps.google.com/maps/api/js?'
                 'v=3&key=YOUR-GOOGLE-MAPS-API-KEY-HERE'),
        )
```

Here we need the development version of OpenLayers because OpenLayers 2.10 doesn't implement version 3 of the Google Maps API. We also specify that we're using the google projection.

```
{# forms/google_map.html #}
{% extends "floppyforms/gis/openlayers.html" %}

{% block options %}
{{ block.super }}
options['base_layer'] = new OpenLayers.Layer.Google("Google Streets",
                                                    {numZoomLevels: 20,
                                                     units: 'm'});
options['point_zoom'] = 14;
{% endblock %}
```

Calling `block.super` generates the options dictionary with all the required options. We can then safely alter it at will. In this case we can directly add an OpenLayers.Layer instance to the map options and it will be picked up as a base layer.

The following options can be passed to the widget constructor:

- `base_layer`: an OpenLayers.Layer instance (or an instance of a subclass) that will be used as a base layer for the map. Default: Metacarta's base WMS layer.
- `color`: the color of the features drawn on the map. Default: `'ee9900'`.
- `default_lon`: the default longitude to center the map on if there is no feature. Default: 0.
- `default_lat`: the default latitude to center the map on if there is no feature. Default: 0.
- `default_zoom`: the default zoom level to use when there is no feature. Default: 4.
- `geom_type`: an OpenLayers.Geometry.* class name.
- `id`: the id of the textarea to whih the feature is serialized.
- `is_collection`: whether the feature to draw is a collection. Default: false.
- `is_linestring`: whether the feature to draw is a linestring. Default: false.
- `is_point`: whether the feature to draw is a point. Default: false.
- `is_polygon`: whether the feature to draw is a polygon. Default: false.
- `layerswitcher`: whether to show OpenLayers' layerswitcher control. Default: false.
- `map_id`: the id of the div containing the map.
- `map_options`: a dictionnary for the options passed to the OpenLayers.Map constructor. Default: {}.
- `map_srid`: the SRID to use for the map. Default: 4326.
- `modifiable`: whether the feature can be modifiable or not. Default: true.
- `mouse_position`: whether to show the coordinates of the mouse on the side of the map. Default: false.
- `name`: the name of the layer containing the feature to draw.
- `opacity`: the opacity of the inner parts of the drawn features (mostly, polygons). Default: 0.4.
- `point_zoomm`: the zoom level to set when a map is displayed with a single point on it. For other feature types, the map is focused automatically on the feature. Default: 12.

- `scale_text`: whether to show the scale information on the side of the map. Default: false.
- `scrollable`: if set to false, the user won't be able to scroll to zoom in and out.

There is also a `map_options` block that can be overridden. Its purpose is to declare a `map_options` dictionnary that can be passed to the `OpenLayers.Map` constructor. For instance:

```
{% block map_options %}
var map_options = {
    maxExtend: new OpenLayers.Bounds(-20037508,-20037508,20037508,20037508),
    maxResolution: 156543.0339,
    numZoomLevels: 20,
    units: 'm'
};
{% endblock %}
```

Here we don't need to call `block.super` since the base template only instantiates an empty dictionnary.

### Going further

If the options or the map options don't give you enough flexibility, you can, not necessarily in that order:

- Redefine the template structure, based on the default OpenLayers template.
- Extend the MapWidget javascript library.

In either way, digging into floppyforms' code (templates, widgets, javascript lib) is more than encouraged. Of course, if you end up implementing additional base widgets for new map providers, feel free to contribute them back!

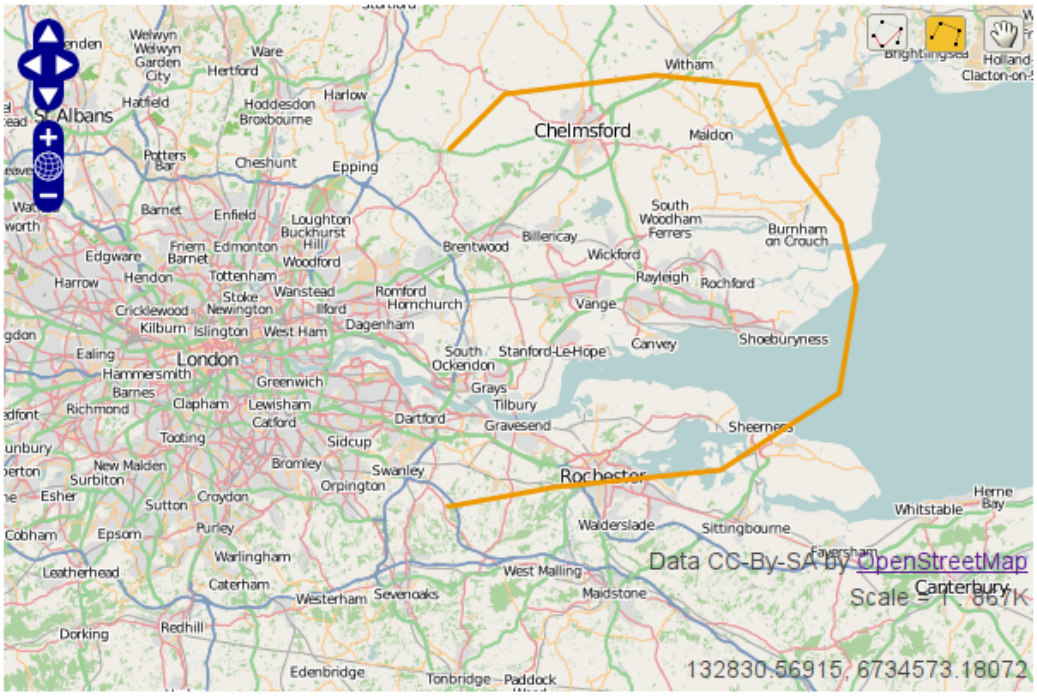
If you need a custom base widget, it is important to inherit from `floppyforms.gis.BaseGeometryWidget`: if you inherit from an existing base widget, you may end up with conflicting media files. `BaseGeometryWidget` doesn't specify any javascript file so get more control by subclassing it.

## Examples

### OpenStreetMap

```
# forms.py
import floppyforms as forms

class OsmLineStringWidget(forms.gis.BaseOsmWidget,
                          forms.gis.LineStringWidget):
    pass

class OsmForm(forms.Form):
    line = forms.gis.LineStringField(widget=OsmLineStringWidget)
```

Result:




Delete all Features

### Google Maps

```
# forms.py
import floppyforms as forms

class GMapPolygonWidget(forms.gis.BaseGMapWidget,
                        forms.gis.PolygonWidget):
    google_maps_api_key = 'YOUR-GOOGLE-MAPS-API-KEY-HERE'

class GmapForm(forms.Form):
    poly = forms.gis.PolygonField(widget=GMapPolygonWidget)
```

Result:


Delete all Features

# Form layouts

New in version 1.0.

## Using form layouts

django-floppyforms tries to make displaying Django forms in a template a bit easier by using the concept of a reusable form layout. A layout is basically just a single template that knows how to render a form into HTML. Here is a simple example demonstrating how to use a layout:

```
<form action="/contact/" method="post">{% csrf_token %}
    {% form contact_form using "floppyforms/layouts/p.html" %}
    <input type="submit" value="Submit" />
</form>
```

Usually a form layout doesn't include the surrounding `<form>` tags and the submit button. So you need to take care of that.

`{% form myform using "floppyforms/layouts/p.html" %}` will output the form with each field and accompanying label wrapped in a paragraph and is meant as a replacement for django's `{{ myform.as_p }}` method. Here is the possible output for our example:

```
<form action="/contact/" method="post">
    <p>
        <label for="id_subject">Subject:</label>
        <input id="id_subject" type="text" name="subject" maxlength="100" />
    </p>
    <p>
        <label for="id_message">Message:</label>
```

```
        <input type="text" name="message" id="id_message" />
    </p>
    <p>
        <label for="id_sender">Sender:</label>
        <input type="text" name="sender" id="id_sender" />
    </p>
    <p>
        <label for="id_cc_myself">Cc myself:</label>
        <input type="checkbox" name="cc_myself" id="id_cc_myself" />
    </p>
    <input type="submit" value="Submit" />
</form>
```

You can also use `floppyforms/layouts/table.html` to output table rows (you'll need to provide your own `<table>` tags) and `floppyforms/layouts/ul.html` to output list items. See the *list of built-in form layouts* for more information.

## Customizing the layout template

If the default layouts are not to your taste, you can completely customize the way a form is presented using the Django template language. Extending the above example:

```
<form action="/contact/" method="post">
    {% form contact_form using "my_layout.html" %}
    <p><input type="submit" value="Send message" /></p>
</form>
```

`my_layout.html` is able to extend one of the built-in layouts, modifying the parts you want to change:

```
{% extends "floppyforms/layouts/table.html" %}

{% block errors %}
    <p>Following errors occurred that cannot be matched to a field:</p>
    {{ block.super }}
{% endblock %}
```

See the *form layout reference* for a detailed description on how you can structure your custom layouts.

You can also specify your form layout "inline" rather than in a separate template file, if you don't plan to reuse it. This is also done with the *form* tag:

```
<form action="/signup/" method="post">
    {% form signup_form using %}
    <div><label for="id_username">Username:</label>
        {% formfield form.username %}<div>
    <div><label for="id_password">Password:</label>
        {% formfield form.password %}</div>
    <div>
        <label for="id_firstname">First- and Lastname:</label><br />
        {% formfield form.firstname %}
        {% formfield form.lastname %}
    </div>
    {% endform %}
    <p><input type="submit" value="Send message" /></p>
</form>
```

Note that the `signup_form` variable will also be available as `form` inside the templatetag. This is for convenience and having always the same memorizable name makes using the same template a lot easier.

Something new in the example is also the *formfield* tag. It is used to render the *widget* of a form field so that you don't have to type out all the `<input />` tags yourself.

But just displaying the widget is not all that you need to take into account when you are creating your own design. You also need to take care where to display errors if a field's validation fails, how to display the help text that might be defined for a field, etc. Because of this it is in most cases easier to split out these *form rows* (containing one or more fields) into their own templates. They work just like form layouts but for a subset of fields and taking care of the errors, help text and other HTML that appears for every field. Here is how it might look like:

```
<form action="/signup/" method="post">
    {% form signup_form using %}
        {% formrow form.username using "div_row.html" %}
        {% formrow form.password using "div_row.html" %}
        {% formrow form.firstname form.lastname using "many_fields_div_row.html" with␣
↪label="First- and Lastname" %}
    {% endform %}
    <p><input type="submit" value="Sign up" /></p>
</form>
```

## Rendering multiple forms

Sometimes you want to render multiple forms at once, all with the same layout without repeating yourself. You can do that by passing either a list or multiple single forms into `{% form %}`:

```
<form action="" method="post">
    {% form myform1 myform2 using "floppyforms/layouts/p.html" %}
    <p><input type="submit" value="Submit" /></p>
</form>
```

For the built-in layouts, the output is the same as for:

```
<form action="" method="post">
    {% form myform1 using "floppyforms/layouts/p.html" %}
    {% form myform2 using "floppyforms/layouts/p.html" %}
    <p><input type="submit" value="Submit" /></p>
</form>
```

Your own layouts can change their behaviour depending on how many forms you have specified, like wrapping them in a fieldset and giving those unique ids etc.

### Using layouts with formsets

Here is how rendering a formset might look like:

```
<form action="" method="post">
    {{ formset.management_form }}
    {% form formset.forms %}
    <p><input type="submit" value="submit" /></p>
</form>
```

## Built-in layouts

django-floppyforms ships with three standard form layouts:

### Paragraph

Renders the form fields in `<p>` tags using the `floppyforms/layouts/p.html` template.

The **default row template** is `floppyforms/rows/p.html`.

The recommended way to use layouts is by using the *{% form %} templatetag*. However django-floppyforms will hook for your convenience into django's `as_*` methods so that they use templates and can be modified to your needs. The `p` layout will be used for all `{{ form.as_p }}`.

### Unordered list

Renders the form fields as `<li>` tags using the `floppyforms/layouts/ul.html` template. It does not display the surrounding `<ul>`. So infact you also can use it with a `<ol>`.

The **default row template** is `floppyforms/rows/li.html`.

This layout will be used for all `{{ form.as_ul }}`.

### Table

Renders the form fields as `<tr>` tags using the `floppyforms/layouts/table.html` template. It does not display the surrounding `<table>` or `<tbody>`. Please take care of that.

The **default row template** is `floppyforms/rows/tr.html`.

This layout will be used for all `{{ form.as_table }}`.

### Default template

django-floppyforms uses the default template layout `floppyforms/layouts/default.html` when calling `{% form myform %}` without the `using` parameter.

The actual code in the default layout looks like:

```
{% extends "floppyforms/layouts/table.html" %}
```

You can drop in your own default form layout, for use when no specific layout is defined, by placing a `floppyforms/layouts/default.html` in your templates directory.

The **default row template** is `floppyforms/rows/default.html`

This layout will be used as default for all `{{ form }}`.

## Create custom layouts

Sometimes the sample layouts mentioned above just don't meet your needs. In that case there are some possibilities to customize them.

The simplest way is to use Django's template inheritance to extend a built-in layout, only overwriting the bits you want to modify. In this case, use the layout that matches your needs best and customize it by overriding one of the following blocks:

- `formconfig`: In this block are all the *formconfig* templatetags that are used in the layout. The built-in layouts configure their row level template here.
- `forms`: This block wraps all the actual markup output. Use this to add markup before or after the rendered forms:

```
{% extends "floppyforms/layouts/p.html" %}

{% block forms %}
    <form action="" method="post">{% csrf_token %}
        {{ block.super }}
        <p><input type="submit" value="submit" /></p>
    </form>
{% endblock %}
```

  The preceding example shows a custom form layout that renders all elements in a paragraph based layout that also contains the necessary `<form>` tag and a submit button.
- `errors`: All non field errors and errors of hidden fields are rendered in this block (the default layouts render errors by including the `form/errors.html` template).
- `rows`: The `rows` block contains a for loop that iterates over all visible fields and displays them in the `row` block. Hidden fields are rendered in the last row.
- `row`: This block is wrapped around the `{% formrow %}` templatetag.

Alternatively it is of course possible to write your own form layout from scratch. Have a look at the existing ones to get an idea what is possible, what cases to take into account and how the template code could look like.

### Creating reusable layouts

When you try to create reusable layouts, it is in most cases usefull to provide some configuration options via arguments. In general the global template context is available to the layout as well as you can pass extra variables into the *form*:

```
{% form contact_form using "my_form_layout.html" with headline="Fill in your enquiry"
↪%}
```

Whereas `my_form_layout.html` could look like:

```
{% extends "floppyforms/layouts/p.html" %}

{% block forms %}
    {% if headline %}<h1>{{ headline }}</h1>{% endif %}
    {{ block.super }}
{% endblock %}
```

### Form rows

A vital part of any form layout is one or are many templates for form rows. A row can be used to render one or multiple fields in a repeating manner.

The built-in row templates render each passed in field in a separate row. You can extend and override these like you can with complete form layouts as described above. Use the following blocks to customize them to your needs:

- `row`: This is the most outer block and wraps all the generated HTML. Use it to wrap the row into additional markup.
- `field`: You can use this block to wrap every single field into additional markup.

- `errors`: Errors are displayed as a `<ul>` list. Override the `errors` block to customize their appearance.
- `label`: Change the label markup by overriding this block.
- `widget`: This one contains just the `{% formfield %}` templatetag that will render the field's widget.
- `help_text`: Change the help text markup by overriding this block.
- `hidden_fields`: The built-in row templates allow hidden fields to be passed into the row with the template variable named `hidden_fields`. The form layouts pass all the form's hidden fields into the last rendered form row.

## Template tags

To load the floppyforms template library you have to load it on top of your templates first:

```
{% load floppyforms %}
```

### form

New in version 1.0.

The `form` tag is used to render one or more form instances using a template.

```
{% form myform using "floppyforms/layouts/p.html" %}
{% form myform another_form form3 using "floppyforms/layouts/p.html" %}
```

django-floppyforms provides three built-in layouts:

- `floppyforms/layouts/p.html`: wraps each field in a `<p>` tag.
- `floppyforms/layouts/ul.html`: wraps each field in a `<li>` tag.
- `floppyforms/layouts/table.html`: wraps each form row with a `<tr>`, the label with a `<th>` and the widget with a `<td>` tag.

See the documentation on *layouts and how to customize them* for more details.

You can use a default layout by leaving the `using ...` out:

```
{% form myform %}
```

In this case the `floppyforms/layouts/default.html` template will be used, which by default is the same as `floppyforms/layouts/p.html`.

Sometimes it is necessary to pass additional template variables into the context of a form layout. This can be done in the same way and with the same syntax as django's include template tag:

```
{% form myform using "layout_with_title.html" with title="Please fill in the form"␣
↪only %}
```

The `only` keyword, as shown in the example above, acts also the same way as it does in the `include` tag. It prevents other, not explicitly specified, variables from being available in the layout's template context.

### Inline layouts

Inlining the form layout is also possible if you don't plan to reuse it somewhere else. This is done by not specifying a template name after the `using` keyword:

```
{% form myform using %}
    ... your form layout here ...
{% endform %}
```

## formconfig

New in version 1.0.

The `formconfig` tag can be used to configure some of the form template tags arguments upfront so that they don't need to be specified over and over again.

The first argument specifies which part of the form should be configured:

#### row

The `formrow` tag takes arguments to specify which template is used to render the row and whether additional variables are passed into this template. These parameters can be configured for multiple form rows with a `{% formconfig row ... %}` tag. The syntax is the same as with `formrow`:

```
{% formconfig row using "floppyforms/rows/p.html" %}
{% formconfig row using "my_form_layout.html" with hide_errors=1 only %}
```

Please note that form configurations will only be available in a form layout or wrapped by a `form` template tag. They also only apply to all the form tags that come after the `formconfig`. It is possible to overwrite already set options. Here is a valid example:

```
{% form myform using %}
<form action="" method="post" id="signup">{% csrf_token %}
    {% formconfig row using "floppyforms/rows/p.html" %}
    {% formrow form.username %}
    {% formrow form.password %}

    {% formconfig row using "floppyforms/rows/tr.html" %}
    <table>
    {% formrow form.firstname form.lastname %}
    {% formrow form.age %}
    {% formrow form.city form.street %}
    </table>

    <p><input type="submit" value="Signup!" /></p>
</form>
{% endform %}
```

However a configuration set with `formconfig` will only be available inside the `form` tag that it was specified in. This makes it possible to scope the configuration with an extra use of the `form` tag. See this example:

```
{% form myform using %}
<form action="" method="post" id="signup">{% csrf_token %}
    {# will use default row template #}
    {% formrow form.username %}
```

```
    {% form form using %}
        <ul>
            {# this config will not be available outside of the wrapping form tag #}
            {% formconfig row using "floppyforms/rows/li.html" %}

            {# will use configured li row template #}
            {% formrow form.password form.password2 %}
        </ul>
    {% endform %}

    {# will use default row template #}
    {% formrow form.firstname form.lastname %}

    <p><input type="submit" value="Signup!" /></p>
</form>
{% endform %}
```

### field

A form field takes the same arguments as a form row does, so the same configuration options are available here, in addition to a `for` keyword to limit which fields the specified configuration will apply to.

Using the `for` keyword allows you to limit the configuration to a specific field or a set of fields. After the `for` keyword, you can give:

- a form field, like `form.field_name`
- the name of a specific field, like `"username"`
- a class name of a form field, like `"CharField"`
- a class name of a widget, like `"Textarea"`

The configuration applied by `{% formconfig field ... %}` is then only available on those fields that match the given criteria.

Here is an example to clarify things. The `formconfig` in the snippet below will only affect the second `formfield` tag but the first one will be left untouched:

```
{% formconfig field using "input.html" with type="password" for userform.password %}
{% formfield userform.username %}
{% formfield userform.password %}
```

And some more examples showing the filtering applied on field names, field types and widget types:

```
{% formconfig field with placeholder="Type to search ..." for "search" %}
{% formfield myform.search %}

{% formconfig field using "forms/widgets/textarea.html" for "CharField" %}
{% formfield myform.comment %}

{% formconfig field using class="text_input" for "TextInput" %}
{% formfield myform.username %}
```

**Note:** Please note that the filterings that act on the field class name and widget class name (like `"CharField"`) also match on subclasses of those field. This means if your class inherits from `django.forms.fields.CharField`

it will also get the configuration applied specified by `{% formconfig field ... for "CharField" %}`.

## formfield

New in version 1.0.

Renders a form field using the associated widget. You can specify a widget template with the `using` keyword. Otherwise it will fall back to the *widget's default template*.

It also accepts `include`-like parameters:

```
{% formfield userform.password using "input.html" with type="password" %}
```

The `formfield` tag should only be used inside a form layout, usually in a row template.

## formrow

New in version 1.0.

The `formrow` tag is a quite similar to the `form` tag but acts on a set of form fields instead of complete forms. It takes one or more fields as arguments and a template which should be used to render those fields:

```
{% formrow userform.firstname userform.lastname using "floppyforms/rows/p.html" %}
```

It also accepts `include`-like parameters:

```
{% formrow myform.field using "my_row_layout.html" with hide_errors=1 only %}
```

The `formrow` tag is usually only used in form layouts.

See the documentation on *row templates and how they are customized* for more details.

## widget

New in version 1.0.

The `widget` tag lets you render a widget with the outer template context available. By default widgets are rendered using a completely isolated context. In some cases you might want to access the outer context, for instance for using floppyforms widgets with django-sekizai:

```
{% for field in form %}
    {% if not field.is_hidden %}
        {{ field.label_tag }}
        {% widget field %}
        {{ field.errors }}
    {% else %}
        {% widget field %}
    {% endif %}
{% endfor %}
```

You can safely use the `widget` tag with non-floppyforms widgets, they will be properly rendered. However, since they're not template-based, they won't be able to access any template context.

# Differences with django.forms

So, you have a project already using `django.forms`, and you're considering a switch to floppyforms? Here's what you need to know, assuming the only change you've made to your code is a simple change, from:

```
from django import forms
```

to:

```
import floppyforms as forms
```

---

**Note:** `django.forms.*` modules

Other modules contained by `django.forms`, such as `forms`, `utils` and `formsets` have not been aliased.

---

## HTML 5 forms!

Floppyforms adds a couple of HTML 5 features on top of the standard Django widgets: HTML syntax, more native widget types, the `required` attribute and client-side validation.

### HTML syntax instead of XHTML

Floppyforms uses an HTML syntax instead of Django's XHTML syntax. You will see `<input type="text" ... >` and not `<input type="text" />`.

### Native widget types

Floppyforms tries to use the native HTML5 widgets whenever it's possible. Thus some widgets which used to be simple `TextInputs` in `django.forms` are now specific input that will render as `<input type="...">` with the HTML5 types such as `url`, `email`. See *Default widgets for form fields* for a detailed list of specific widgets.

For instance, if you have declared a form using django.forms:

```
class ThisForm(forms.Form):
    date = forms.DateField()
```

The `date` field will be rendered as an `<input type="text">`. However, by just changing the forms library to floppyforms, the input will be an `<input type="date">`.

### Required attribute

In addition to the various input types, every required field has the `required` attribute set to `True` on its widget. That means that every `<input>` widget for a required field will be rendered as `<input type="..." ... required>`. This is used for client-side validation: for instance, Firefox 4 won't let the user submit the form unless he's filled the input. This saves HTTP requests but doesn't mean you can stop validating user input.

### Client-side validation

Like with the `required` attribute, the `pattern` attribute is especially interesting for slightly more complex client-side validation. The `SlugField` and the `IPAddressField` both have a pattern attached to the `<input>`.

However having these validations backed directly into the HTML and therefore allowing the browser to validate the user input might not always what you want to have. Sometimes you just want to have a form where it should be allowed to submit invalid data. In that case you can use the `novalidate` attribute on the `<form>` HTML tag or the `formnovalidate` attribute on the submit button:

```
<form action="" novalidate>
    This input will not be validated:
    <input type="text" required />
</form>

<form action="">
    Another way to not validate the form in the browser is using the
    formnovalidate attribute on the submit button:
    <input type="submit" value="cancel" formnovalidate>
</form>
```

Read the corresponding documentation for novalidate and formnovalidate on the Mozilla Developer Network if you want to know more.

## ModelForms

Prior to version 1.2 of django-floppyforms, you had to take some manual efforts to make your modelforms work with floppyforms. This is now done seemlesly, but since this was introduced a backwards incompatible change, it was necessary to provde a deprecation path.

So if you start out new with django-floppyforms just use `import floppyforms.__future__ as forms` as your import instead of `import floppyforms as forms` when you want to define modelforms.

For more information see the *section about modelforms in the usage documentation*.

## `help_text` values are autoescaped by default

If you use HTML in the `help_text` value for a Django form field and are not using django-floppyforms, then you will get the correct HTML rendered in the template. For example you have this form:

```
from django import forms

class DjangoForm(forms.Form):
    myfield = forms.CharField(help_text='A <strong>help</strong> text.')
```

When you now use this form with `{{ form.as_p }}` in the template, you will get the help text put in the template as it is, with no HTML escaping. That might imply a security risk if your help text contains content from untrusted sources. django-floppyforms applies autoescaping by default to the help text. So if you define:

```
import floppyforms as forms

class FloppyForm(forms.Form):
    myfield = forms.CharField(help_text='A <strong>help</strong> text.')
```

And then use `{{ form.as_p }}`, you will get an output that contains `A &lt;strong&gt;help&lt;/strong&gt; text.`. You can disable the autoescaping of the help text by using Django's `mark_safe` helper:

```
from django.utils.html import mark_safe
import floppyforms as forms

class FloppyForm(forms.Form):
    myfield = forms.CharField(help_text=mark_safe('A <strong>help</strong> text.'))
```

### `TEMPLATE_STRING_IF_INVALID` caveats

The use of a non-empty `TEMPLATE_STRING_IF_INVALID` setting can impact rendering. Missing template variables are rendered using the content of `TEMPLATE_STRING_IF_INVALID` but filters used on non-existing variables are not applied (see django's documentation on how invalid template variables are handled for more details).

django-floppyforms assumes in its predefined form layouts that all filters are applied. You can work around this by making your `TEMPLATE_STRING_IF_INVALID` evaluate to `False` but still keep its string representation. Here is an example how you could achieve this in your `settings.py`:

```
# on Python 2
class InvalidVariable(unicode):
    def __nonzero__(self):
        return False

# on Python 3
class InvalidVariable(str):
    def __bool__(self):
        return False

TEMPLATE_STRING_IF_INVALID = InvalidVariable(u'INVALID')
```

### Getting back Django's behaviour

If you need to get the same output as standard Django forms:

- Override `floppyforms/input.html`, `floppyforms/radio.html`, `floppyforms/clearable_input.html`, `floppyforms/textarea.html` and `floppyforms/checkbox_select.html` to use an XHTML syntax
- Remove the `required` attribute from the same templates, as well as `floppyforms/select.html`
- Make sure your fields which have HTML5 widgets by default get simple `TextInputs` instead:

```
class Foo(forms.Form):
    url = forms.URLField(widget=forms.TextInput)
```

## Example widgets

### A date picker

This snippet implements a rich date picker using the browser's date picker if the `date` input type is supported and falls back to a jQuery UI date picker.

```python
# forms.py
import floppyforms as forms


class DatePicker(forms.DateInput):
    template_name = 'datepicker.html'

    class Media:
        js = (
            'js/jquery.min.js',
            'js/jquery-ui.min.js',
        )
        css = {
            'all': (
                'css/jquery-ui.css',
            )
        }


class DateForm(forms.Form):
    date = forms.DateField(widget=DatePicker)
```

```
{# datepicker.html #}
{% include "floppyforms/input.html" %}

<script type="text/javascript">
  $(document).ready(function() {
    // Checking support for <input type="date"> using Modernizr:
    // http://modernizr.com/
    if (!Modernizr.inputtypes.date) {
      var options = {
        dateFormat: 'yy-mm-dd'
      };
      $('#{{ attrs.id }}').datepicker(options);
    }
  });
</script>
```

Here is how chromium renders it with its native (but sparse) date picker:


And here is the jQuery UI date picker as shown by Firefox:


## An autofocus input

A text input with the autofocus attribute and a fallback for browsers that doesn't support it.

```
# forms.py
import floppyforms as forms


class AutofocusInput(forms.TextInput):
    template_name = 'autofocus.html'

    def get_context_data(self):
        self.attrs['autofocus'] = True
        return super(AutofocusInput, self).get_context_data()


class AutofocusForm(forms.Form):
    text = forms.CharField(widget=AutofocusInput)
```

```
{# autofocus.html #}
{% include "floppyforms/input.html" %}

<script type="text/javascript">
  window.onload = function() {
    if (!("autofocus" in document.createElement("input"))) {
      document.getElementById("{{ attrs.id }}").focus();
    }
  };
</script>
```

## A slider

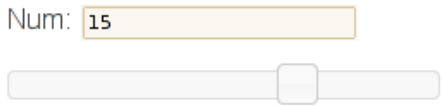
A `range` input that uses the browser implementation or falls back to jQuery UI.

```python
# forms.py
import floppyforms as forms


class Slider(forms.RangeInput):
    min = 5
    max = 20
    step = 5
    template_name = 'slider.html'

    class Media:
        js = (
            'js/jquery.min.js',
            'js/jquery-ui.min.js',
        )
        css = {
            'all': (
                'css/jquery-ui.css',
            )
        }


class SlideForm(forms.Form):
    num = forms.IntegerField(widget=Slider)

    def clean_num(self):
        num = self.cleaned_data['num']
        if not 5 <= num <= 20:
            raise forms.ValidationError("Enter a value between 5 and 20")

        if not num % 5 == 0:
            raise forms.ValidationError("Enter a multiple of 5")
        return num
```

```
{# slider.html #}
{% include "floppyforms/input.html" %}
<div id="{{ attrs.id }}-slider"></div>

<script type="text/javascript">
  $(document).ready(function() {
    var type = $('<input type="range" />').attr('type');
    if (type == 'text') { // No HTML5 support
      $('#{{ attrs.id }}').attr("readonly", true);
      $('#{{ attrs.id }}-slider').slider({
        {% if value %}value: {{ value }},{% endif %}
        min: {{ attrs.min }},
        max: {{ attrs.max }},
        step: {{ attrs.step }},
        slide: function(event, ui) {
          $('#{{ attrs.id }}').val(ui.value);
        }
      });
    }
  });
</script>
```

Here is how chromium renders it with its native slider:


And here is the jQuery UI slider as shown by Firefox:




## A placeholder with fallback

An `<input>` with a `placeholder` attribute and a javascript fallback for broader browser support.

```
# forms.py
import floppyforms as forms


class PlaceholderInput(forms.TextInput):
    template_name = 'placeholder_input.html'


class MyForm(forms.Form):
    text = forms.CharField(widget=PlaceholderInput(
        attrs={'placeholder': _('Some text here')},
    ))
```

```
{# placeholder_input.html #}

{% include "floppyforms/input.html" %}

<script type="text/javascript">
    window.onload = function() {
        if (!('placeholder' in document.createElement('input'))) {
            var input = document.getElementById('{{ attrs.id }}');
            input.value = '{{ attrs.placeholder }}';

            input.onblur = function() {
                if (this.value == '')
                    this.value='{{ attrs.placeholder }}';
            };

            input.onfocus = function() {
                if (this.value == '{{ attrs.placeholder }}')
                    this.value = '';
            };
        }
    };
</script>
```

## An image clearable input with thumbnail

If we have an image set for the field, display the image and propose to clear or to update.

```
# forms.py
import floppyforms as forms

class ImageThumbnailFileInput(forms.ClearableFileInput):
    template_name = 'floppyforms/image_thumbnail.html'


class ImageForm(forms.ModelForm):
    class Meta:
        model = Item
        fields = ('image',)
        widgets = {'image': ImageThumbnailFileInput}
```

```
{# image_thumbnail.html #}
{% load i18n %}
{% if value.url %}{% trans "Currently:" %} <a target="_blank" href="{{ value.url }}">
↪<img src="{{ value.url }}" alt="{{ value }}" height="120"/></a>
{% if not required %}
<p><input type="checkbox" name="{{ checkbox_name }}" id="{{ checkbox_id }}">
<label for="{{ checkbox_id }}">{% trans "Clear" %}</label></p>
    {% else %}<br/>
{% endif %}
{% trans "Change:" %}
{% endif %}
<input type="{{ type }}" name="{{ name }}"{% if required %} required{% endif %}{%␣
↪include "floppyforms/attrs.html" %}>
```

You now have your image:


## Layout example with Bootstrap

If you use Floppyforms with Bootstrap you might be interested in using a bootstrap layout for your form.

What you have to do is to create those two templates:

**floppyforms/templates/floppyforms/layouts/bootstrap.html**:

```
{% load floppyforms %}{% block formconfig %}{% formconfig row using "floppyforms/rows/
↪bootstrap.html" %}{% endblock %}

{% block forms %}{% for form in forms %}
{% block errors %}
    {% for error in form.non_field_errors %}
        <div class="alert alert-error">
            <a class="close" href="#" data-dismiss="alert">×</a>
            {{ error }}
        </div><!--- .alert.alert-error -->
    {% endfor %}
    {% for error in form|hidden_field_errors %}
        <div class="alert alert-error">
            <a class="close" href="#" data-dismiss="alert">×</a>
            {{ error }}
        </div><!--- .alert.alert-error -->
    {% endfor %}
{% endblock errors %}
{% block rows %}
        {% for field in form.visible_fields %}
            {% if forloop.last %}{% formconfig row with hidden_fields=form.hidden_
↪fields %}{% endif %}
            {% block row %}{% formrow field %}{% endblock %}
        {% endfor %}
        {% if not form.visible_fields %}{% for field in form.hidden_fields %}{%␣
↪formfield field %}{% endfor %}{% endif %}
{% endblock %}
{% endfor %}{% endblock %}
```

**floppyforms/templates/floppyforms/rows/bootstrap.html**:

```
{% load floppyforms %}{% block row %}{% for field in fields %}
<div class="control-group{% if field.errors %} error{% endif %}">
    {% with classes=field.css_classes label=label|default:field.label help_text=help_
↪text|default:field.help_text %}
    {% block label %}{% if field|id %}<label class="control-label" for="{{ field|id }}
↪">{% endif %}{{ label }}{% if field.field.required %} <span class="required">*</
↪span>{% endif %}{% if label|last not in ".:!?" %}:{% endif %}{% if field|id %}</
↪label>{% endif %}{% endblock %}
    {% block field %}
        <div class="controls {{ classes }} field-{{ field.name }}">
            {% block widget %}{% formfield field %}{% endblock %}
            {% block errors %}{% include "floppyforms/errors.html" with errors=field.
↪errors %}{% endblock %}
            {% block help_text %}{% if field.help_text %}
                <p class="help-block">{{ field.help_text }}</p>
            {% endif %}{% endblock %}
            {% block hidden_fields %}{% for field in hidden_fields %}{{ field.as_
↪hidden }}{% endfor %}{% endblock %}
        </div><!--- .controls -->
    {% endblock %}
    {% endwith %}
</div><!--- .control-group{% if field.errors %}.error{% endif %} -->
{% endfor %}{% endblock %}
```

You can also define this layout by default:

**floppyforms/templates/floppyforms/layouts/default.html**:

```
{% extends "floppyforms/layouts/bootstrap.html" %}
```

You can also make a change to the error display:

**floppyforms/templates/floppyforms/errors.html**:

```
{% if errors %}<span class="help-inline">{% for error in errors %}{{ error }}{% if
↪not forloop.last %}<br />{% endif %}{% endfor %}</span>{% endif %}
```

And that's it, you now have a perfect display for your form with bootstrap.

# Changelog

## 1.8.0 (in development)

- #176: Fix HTML validation for hidden textarea used with GIS widgets.
- #191: Support for Django 1.10. Thanks to MrJmad for the patch.
- #194: Remove official support for Python 2.6 and Python 3.2.

## 1.7.0

- #171: Fix path to GIS widget images in `openlayers.html` template. The files coming with Django admin where used, but the naming changed in 1.9. We vendor these know to have better control over it.
- #174: Support for setting your own Google Maps key in the `BaseGMapWidget`. See the documentation for details

## 1.6.2

- #169: Use the attributes `ClearableFileInput.initial_text`, `ClearableFileInput.input_text`, `ClearableFileInput.clear_checkbox_label` to determine the used text in the template. This was inconsistent so far with Django's behaviour.

## 1.6.1

- #167: Fix django-floppyforms' `CheckboxInput.value_from_datadict` which was inconsistent with Django's behaviour.

## 1.6.0

- #160: Django 1.9 support! Thanks to Jonas Haag for the patch.

## 1.5.2

- #156: The `min`, `max`, `step` attributes for `DecimalField` and `FloatField` were localized which can result in invalid values (rendering `0.01` as `0,01` in respective locales). Those attributes won't get localized anymore. Thanks to Yannick Chabbert for the fix.

### 1.5.1

- *FloatField*‘ now fills in `min`, `max`, and `step` attributes to match the behaviour of *DecimalField*. Leaving out the `step` attribute would result in widgets that only allow integers to be filled in (HTML 5 default for `step` is `1`).

### 1.5.0

- #148: Added support for custom `label_suffix` arguments in forms and fields.
- The contents in `floppyforms/input.html` is now wrapped in a `{% block content %}` for easier extending.
- #70: *DecimalField*‘ now fills in `min`, `max`, and `step` attributes for better client side validation. Use the `novalidate` attribute on your `<form>` tag to disable HTML5 input validation in the browser. Thanks to caacree for the patch.

### 1.4.1

- Fixed source distribution to include all files in `floppyforms/static/floppyforms/openlayers`.

### 1.4.0

- Every widget is now using its own template. Previously all widgets that are based on the HTML `<input>` tag used the generic `floppyforms/input.html` template. Now the widgets each have a custom element for easier customisation. For example `CheckboxInput` now uses `floppyforms/checkbox.html` instead of `floppyforms/input.html`. See Widgets reference for a complete list of available widgets and which templates they use.
- Adjusting the SRIDs used in the GeoDjango widgets to conform with Django 1.7. Thanks to Tyler Tipton for the patch.
- Python 3.2 is now officially supported.
- Django 1.8 is now officially supported. django-floppyforms no longers triggers Django deprecation warnings.
- Adding OpenLayers distribution to django-floppyforms static files in order to better support HTTPS setups when GIS widgets are used (See #15 for more details).
- Fix: `python setup.py bdist_rpm` failed because of wrong string encodings in setup.py. Thanks to Yuki Izumi for the fix.
- Fix: The `CheckboxInput` widget did detect different values in Python 2 when given `'False'` and `u'False'` as data. Thanks to @artscoop for the patch.
- Fix: `MultipleChoiceField` can now correctly be rendered as hidden field by using the `as_hidden` helper in the template. That was not working previously as there was no value set for `MultipleChoiceField.hidden_widget`.

### 1.3.0

- DateInput widget renders hardcoded “%Y-%m-%d” format. We don’t allow custom formats there since the “%Y-%m-%d” format is what browsers are submitting with HTML5 date input fields. Thanks to Bojan Mihelac for the patch.

- Adding `supports_microseconds` attribute to all relevant widget classes. Thanks to Stephen Burrows for the patch.
- Using a property for `Widget.is_hidden` attribute on widgets to be in conformance with Django 1.7 default widget implementation.
- The docs mentioned that the current `ModelForm` behaviour in `floppyforms.__future__` will become the default in 1.3. This is postpone for one release and will be part of 1.4.

## 1.2.0

- Subclasses of `floppyforms.models.ModelForm` did not convert widgets of form fields that were automatically created for the existing model fields into the floppyform variants. This is now changed, thanks to a patch by Stephen Burrows.

  Previously you had to set the widgets your self in a model form. For example you would write:

```
import floppyforms as forms

class ProfileForm(forms.ModelForm):
    class Meta:
        model = Profile
        widgets = {
            'name': forms.TextInput,
            'url': forms.URLInput,
            ...
        }
```

  Now this is done automatically. But since this is a kind-of backwardsincompatible change, you need to use a special import:

```
import floppyforms.__future__ as forms

class ProfileForm(forms.ModelForm):
    class Meta:
        model = Profile
```

  This feature will become the default behaviour in floppyforms 2.0.

  See the documentation for more information: http://django-floppyforms.readthedocs.org/en/latest/usage.html#modelforms
- If you added an attribute with value 1 to the attrs kwargs (e.g. `{'value': 1}`, you would get no attribute value in the rendered html (e.g. `value` instead of `value="1"`). That's fixed now, thanks to Viktor Ershov for the report.
- All floppyform widget classes now take a `template_name` argument in the `__init__` and `render` method. Thanks to Carl Meyer for the patch.

## 1.1.1

- Fix for Django 1.6
- Fix for GIS widgets on Django 1.4 and some versions of GEOS.

### 1.1

- Added GenericIPAddressField.
- Django 1.5 and Python 3.3 support added.
- Django 1.3 support dropped.
- GIS widgets switched to stable OpenLayers release instead of a dev build.
- Fixed `Textarea` widget template to work with a non-empty `TEMPLATE_STRING_IF_INVALID` setting. Thanks to Leon Matthews for the report.
- Fixed context handling in widget rendering. It didn't take care of popping the context as often as it was pushed onto. This could cause strange behaviour in the template by leaking variables into outer scopes. Thanks to David Danier for the report.
- Added missing empty choice for selectboxes in `SelectDateWidget`. Thanks fsx999 for the report.
- `IntegerField` now automatically passes its `min_value` and `max_value` (if provided) to the `NumberInput` widget.
- Added basic support for `<datalist>` elements for suggestions in `Input` widgets.
- `date`, `datetime` and `time` inputs are not localized anymore. The HTML5 spec requires the rendered values to be RFC3339-compliant and the browsers are in charge of localization. If you still want localized date/time inputs, use those provided by Django or override the `_format_value()` method of the relevant widgets.

### 1.0

- cleaned up the behaviour of `attrs`
- compatible with Django 1.3 and 1.4
- `<optgroup>` support in select widgets
- `Select` widgets: renamed `choices` context variable to `optgroups`. This is **backwards-incompatible**: if you have custom templates for `Select` widgets, they need to be updated.
- `get_context()` is more reliable
- Added `form`, `formrow`, `formfield`, `formconfig` and `widget` template tags.
- Added template-based form layout system.
- Added ability to render widgets with the broader page context, for instance for django-sekizai compatibility.

### 0.4

- All widgets from Django have their floppyforms equivalent
- Added widgets for GeoDjango

CHAPTER 3

# Getting help

Feel free to join the `#django-floppyforms` IRC channel on freenode.

CHAPTER 4

# Why the name?

- There aren't enough packages with silly names in the Django community. So, here's one more.
- The name reflects the idea that a widget can take any kind of shape, if that makes any sense.

CHAPTER 5

# Performance

Each time a widget is rendered, there is a template inclusion. To what extent does it affect performance? You can try with this little script:

```
import timeit

django = """from django import forms

class DjangoForm(forms.Form):
    text = forms.CharField()
    slug = forms.SlugField()
    some_bool = forms.BooleanField()
    email = forms.EmailField()
    date = forms.DateTimeField()
    file_ = forms.FileField()

rendered = DjangoForm().as_p()"""

flop = """import floppyforms as forms

class FloppyForm(forms.Form):
    text = forms.CharField()
    slug = forms.SlugField()
    some_bool = forms.BooleanField()
    email = forms.EmailField()
    date = forms.DateTimeField()
    file_ = forms.FileField()

rendered = FloppyForm().as_p()"""

def time(stmt):
    t = timeit.Timer(stmt=stmt)
    return t.timeit(number=1000)

print "Plain django:", time(django)
print "django-floppyforms:", time(flop)
```

The result varies if you're doing template caching or not. To put it simply, here is the average time for a single iteration on a MacBookPro @ 2.53GHz.

| Method | Time without template caching | Time with template caching |
|---|---|---|
| Plain Django | 1.63973999023 msec | 1.6320669651 msec |
| django-floppyforms | 9.05481505394 msec | 3.0161819458 msec |

Even with template caching, the rendering time is doubled. However the impact is probably not noticeable since rendering the form above takes 3 milliseconds instead of 1.6: **it still takes no time :)**. The use of template caching in production is, of course, encouraged.

# Python Module Index

## f

# Index