
django-crispy-forms Documentation

Release 1.8.1

Miguel Araujo

Nov 22, 2019

Contents

1	User Guide	3
1.1	Installation	3
1.2	crispy filter	4
1.3	{% crispy %} tag with forms	5
1.4	FormHelper	12
1.5	Layouts	15
1.6	How to create your own template packs	23
1.7	{% crispy %} tag with formsets	25
1.8	Updating layouts on the go	27
1.9	Frequently Asked Questions	33
2	API documentation	37
2.1	API helpers	37
2.2	API Layout	38
2.3	API templatetags	41
3	Developer Guide	45
3.1	Contributing	45
	Python Module Index	49
	Index	51

django-crispy-forms provides you with a `|crispy` filter and `{% crispy %}` tag that will let you control the rendering behavior of your Django forms in a very elegant and DRY way. Have full control without writing custom form templates. All this without breaking the standard way of doing things in Django, so it plays nice with any other form application.

Get the most out of django-crispy-forms

1.1 Installation

1.1.1 Installing django-crispy-forms

Install latest stable version into your python environment using pip:

```
pip install django-crispy-forms
```

If you want to install development version (unstable), you can do so doing:

```
pip install git+git://github.com/django-crispy-forms/django-crispy-forms.git@dev  
↪#egg=django-crispy-forms
```

Or, if you'd like to install the development version as a git repository (so you can `git pull` updates), use the `-e` flag with `pip install`, like so:

```
pip install -e git+git://github.com/django-crispy-forms/django-crispy-forms.git@dev  
↪#egg=django-crispy-forms
```

Once installed add `crispy_forms` to your `INSTALLED_APPS` in `settings.py`:

```
INSTALLED_APPS = (  
    ...  
    'crispy_forms',  
)
```

In production environments, always activate Django template cache loader. This is available since Django 1.2 and what it does is basically load templates once, then cache the result for every subsequent render. This leads to a significant performance improvement. To see how to set it up refer to the fabulous [Django docs page](#).

1.1.2 Template packs

Since version 1.1.0, django-crispy-forms has built-in support for different CSS frameworks, known as template packs within django-crispy-forms:

- `bootstrap` [Bootstrap](#) is crispy-forms's default template pack, version 2 of the popular simple and flexible HTML, CSS, and Javascript for user interfaces from Twitter.
- `bootstrap3` [Twitter Bootstrap version 3](#).
- `bootstrap4` [support for Twitter Bootstrap version 4](#).
- `uni-form` [Uni-form](#) is a nice looking, well structured, highly customizable, accessible and usable forms.

In addition the following template packs are available through separately maintained projects.

- `foundation` [Foundation](#) In the creator's words, "The most advanced responsive front-end framework in the world." This template pack is available through [crispy-forms-foundation](#)

If your form CSS framework is not supported and it's open source, you can create a `crispy-forms-templatePackName` project. Please let me know, so I can link to it. Documentation on [How to create your own template packs](#) is provided.

You can set your default template pack for your project using the `CRISPY_TEMPLATE_PACK` Django settings variable:

```
CRISPY_TEMPLATE_PACK = 'uni_form'
```

Please check the documentation of your template pack package for the correct value of the `CRISPY_TEMPLATE_PACK` setting (there are packages which provide more than one template pack).

1.1.3 Setting media files

crispy-forms does not include static files. You will need to include the proper corresponding media files yourself depending on what CSS framework (Template pack) you are using. This might involve one or more CSS and JS files. Read CSS framework's docs for help on how to set it up.

1.2 crispy filter

Crispy filter lets you render a form or formset using django-crispy-forms elegantly div based fields. Let's see a usage example:

```
{% load crispy_forms_tags %}
<form method="post" class="uniForm">
  {{ my_formset|crispy }}
</form>
```

1. Add `{% load crispy_forms_tags %}` to the template.
2. Append the `|crispy` filter to your form or formset context variable.
3. If you are using `uni_form` template pack, don't forget to add the class 'uniForm' to your form.
4. Refresh and enjoy!

1.2.1 Using `{% crispy %}` tag because it rocks

As handy as the `lcrispy` filter is, think of it as the built-in methods: `as_table`, `as_ul` and `as_p`. You cannot tune up the output. The best way to make your forms crisp is using the `{% crispy %} tag with forms`. It will change how you do forms in Django.

1.3 `{% crispy %}` tag with forms

django-crispy-forms implements a class called `FormHelper` that defines the form rendering behavior. Helpers give you a way to control form attributes and its layout, doing this in a programmatic way using Python. This way you write as little HTML as possible, and all your logic stays in the forms and views files.

1.3.1 Fundamentals

For the rest of this document we will use the following example form for showing how to use a helper. This form is in charge of gathering some user information:

```
class ExampleForm(forms.Form):
    like_website = forms.TypedChoiceField(
        label = "Do you like this website?",
        choices = ((1, "Yes"), (0, "No")),
        coerce = lambda x: bool(int(x)),
        widget = forms.RadioSelect,
        initial = '1',
        required = True,
    )

    favorite_food = forms.CharField(
        label = "What is your favorite food?",
        max_length = 80,
        required = True,
    )

    favorite_color = forms.CharField(
        label = "What is your favorite color?",
        max_length = 80,
        required = True,
    )

    favorite_number = forms.IntegerField(
        label = "Favorite number",
        required = False,
    )

    notes = forms.CharField(
        label = "Additional notes or feedback",
        required = False,
    )
```

Let's see how helpers works step by step, with some examples explained. First you will need to import `FormHelper`:

```
from crispy_forms.helper import FormHelper
```

Your helper can be a class level variable or an instance level variable, if you don't know what this means you might want to read the article "[Be careful how you use static variables in forms](#)". As a rule of thumb, if you are not going

to manipulate a *FormHelper* in your code, like in a view, you should be using a static helper, otherwise you should be using an instance level helper. If you still don't understand the subtle differences between both, use an instance level helper, because you won't end up suffering side effects. As in the next steps I will show you how to manipulate the form helper, so we will create an instance level helper. This is how you would do it:

```
from crispy_forms.helper import FormHelper

class ExampleForm(forms.Form):
    [...]
    def __init__(self, *args, **kwargs):
        super(ExampleForm, self).__init__(*args, **kwargs)
        self.helper = FormHelper()
```

As you can see you need to call the base class constructor using `super` and override the constructor. This helper doesn't set any form attributes, so it's useless. Let's see how to set up some basic *FormHelper* attributes:

```
from crispy_forms.helper import FormHelper
from crispy_forms.layout import Submit

class ExampleForm(forms.Form):
    [...]
    def __init__(self, *args, **kwargs):
        super(ExampleForm, self).__init__(*args, **kwargs)
        self.helper = FormHelper()
        self.helper.form_id = 'id-exampleForm'
        self.helper.form_class = 'blueForms'
        self.helper.form_method = 'post'
        self.helper.form_action = 'submit_survey'

        self.helper.add_input(Submit('submit', 'Submit'))
```

Note that we are importing a class called `Submit` that is a layout object. We will see what layout objects are in detail later on, for now on let's just say that this adds a submit button to our form, so people can send their survey.

We've also done some helper magic. `FormHelper` has a list of attributes that can be set, that affect mainly form attributes. Our form will have as DOM id `id-exampleForm`, it will have as DOM CSS class `blueForms`, it will use `http POST` to send information and its action will be set to `reverse(submit_survey)`.

Let's see how to render the form in a template. Supposing we have the form in the template context as `example_form`, we would render it doing:

```
{% load crispy_forms_tags %}
{% crispy example_form example_form.helper %}
```

Notice that the `{% crispy %}` tags expects two parameters: first the form variable and then the helper. In this case we use the `FormHelper` attached to the form, but you could also create a `FormHelper` instance and pass it as a context variable. Most of the time, you will want to use the helper attached. Note that if you name your `FormHelper` attribute `helper` you will only need to do:

```
{% crispy form %}
```

This is exactly the html that you would get:

```
<form action="/submit/survey/" class="uniForm blueForms" method="post" id="id-
↪exampleForm">
  <div style='display:none'>
    <input type="hidden" name="csrfmiddlewaretoken" value=
↪"a643fab735d5ce6377ff456e73c4b1af" />
```

(continues on next page)

(continued from previous page)

```

</div>
<fieldset>
  <legend></legend>
  <div id="div_id_like_website" class="ctrlHolder">
    <label for="id_like_website" class="requiredField">¿Do you like this_
↪website?<span class="asteriskField">*</span></label>
    <ul>
      <li><label for="id_like_website_0"><input checked="checked" name=
↪"like_website" value="1" id="id_like_website_0" type="radio" class="radioselect" />_
↪Yes</label></li>
      <li><label for="id_like_website_1"><input value="0" type="radio" _
↪class="radioselect" name="like_website" id="id_like_website_1" /> No</label></li>
    </ul>
  </div>
  <div id="div_id_favorite_food" class="ctrlHolder">
    <label for="id_favorite_food" class="requiredField">What is you favorite_
↪food?<span class="asteriskField">*</span></label>
    <input id="id_favorite_food" class="textinput textInput" type="text" name=
↪"favorite_food" maxlength="80" required="required" />
  </div>
  <div id="div_id_favorite_color" class="ctrlHolder">
    <label for="id_favorite_color" class="requiredField">What is you favorite_
↪color?<span class="asteriskField">*</span></label>
    <input id="id_favorite_color" class="textinput textInput" type="text" _
↪name="favorite_color" maxlength="80" required="required" />
  </div>
  <div id="div_id_favorite_number" class="ctrlHolder">
    <label for="id_favorite_number">Favorite number</label>
    <input id="id_favorite_number" type="text" name="favorite_number" class=
↪"textinput textInput" />
  </div>
  <div id="div_id_notes" class="ctrlHolder">
    <label for="id_notes">Additional notes or feedback</label>
    <input id="id_notes" type="text" name="notes" class="textinput textInput" _
↪/>
  </div>
</fieldset>
<div class="buttonHolder">
  <input type="submit" name="submit" value="Submit" class="submit submitButton" _
↪id="submit-id-submit" />
</div>
</form>

```

What you'll get is the form rendered as HTML with awesome bits. Specifically...

- Opening and closing form tags, with id, class, action and method set as in the helper:

```

<form action="/submit/survey/" class="uniForm blueForms" method="post" id="id-
↪exampleForm">
  [...]
</form>

```

- Django's CSRF controls:

```

<div style='display:none'>
  <input type="hidden" name="csrfmiddlewaretoken" value=
↪"a643fab735d5ce6377ff456e73c4b1af" />

```

(continues on next page)

(continued from previous page)

</div>

- Submit button:

```
<div class="buttonHolder">
  <input type="submit" name="submit" value="Submit" class="submit submitButton"
  ↪id="submit-id-submit" />
</div>
```

1.3.2 Manipulating a helper in a view

Let's see how we could change any helper property in a view:

```
@login_required()
def inbox(request, template_name):
    example_form = ExampleForm()
    redirect_url = request.GET.get('next')

    # Form handling logic
    [...]

    if redirect_url is not None:
        example_form.helper.form_action = reverse('submit_survey') + '?next=' +
        ↪redirectUrl

    return render_to_response(template_name, {'example_form': example_form}, context_
    ↪instance=RequestContext(request))
```

We are changing `form_action` helper property in case the view was called with a `next` GET parameter.

1.3.3 Rendering several forms with helpers

Often we get asked: “How do you render two or more forms, with their respective helpers, using `{% crispy %}` tag, without having `<form>` tags rendered twice?” Easy, you need to set `form_tag` helper property to `False` in every helper:

```
self.helper.form_tag = False
```

Then you will have to write a little of html code surrounding the forms:

```
<form action="{% url 'submit_survey' %}" class="uniForm" method="post">
  {% crispy first_form %}
  {% crispy second_form %}
</form>
```

You can read a list of *Helper attributes you can set* and what they are for.

1.3.4 Change “*” required fields

If you don't like the use of `*` (asterisk) to denote required fields you have two options:

- Asterisks have an `asteriskField` class set. So you can hide it using CSS rule:

```
.asteriskField {
    display: none;
}
```

- Override `field.html` template with a custom one.

1.3.5 Make crispy-forms fail loud

By default when crispy-forms encounters errors, it fails silently, logs them and continues working if possible. A settings variable called `CRISPY_FAIL_SILENTLY` has been added so that you can control this behavior. If you want to raise exceptions instead of logging, telling you what's going on when you are developing in debug mode, you can set it to:

```
CRISPY_FAIL_SILENTLY = not DEBUG
```

1.3.6 Change crispy-forms `<input>` default classes

Django fields generate default classes, crispy-forms handles these and adds other classes for compatibility with CSS frameworks.

For example a CharField generates an `<input class="textinput" ...`. But in uniform we need the class to be `textInput` (with capital 'I'), so crispy-forms leaves it like `<input class="textinput textInput" ...`. All official template packs are handled automatically, but maybe you are integrating a new CSS framework, or your company's custom one, with crispy-forms and need to change the default conversions. For this you need to use a settings variable called `CRISPY_CLASS_CONVERTERS`, expected to be a Python dictionary:

```
CRISPY_CLASS_CONVERTERS = {'textinput': "textinput inputtext"}
```

For example this setting would generate `<input class="textinput inputtext" ...`. The key of the dictionary `textInput` is the Django's default class, the value is what you want it to be substituted with, in this case we are keeping `textInput`.

1.3.7 Render a form within Python code

Sometimes, it might be useful to render a form using crispy-forms within Python code, like a Django view, for that there is a nice helper `render_crispy_form`. The prototype of the method is `render_crispy_form(form, helper=None, context=None)`. You can use it like this. Remember to pass your CSRF token to the helper method using the context dictionary if you want the rendered form to be able to submit.

1.3.8 AJAX validation recipe

One easy way to validate a crispy-form through AJAX and re-render the resulting form errors if any is to set up a view, that validates the form and renders its html using `render_crispy_form` to finally return this html to the client AJAX request. Let's see an example.

Our server side code could be:

```
from crispy_forms.utils import render_crispy_form

@json_view
def save_example_form(request):
```

(continues on next page)

(continued from previous page)

```
form = ExampleForm(request.POST or None)
if form.is_valid():
    # You could actually save through AJAX and return a success code here
    form.save()
    return {'success': True}

# RequestContext ensures CSRF token is placed in newly rendered form_html
request_context = RequestContext(request)
form_html = render_crispy_form(form, context=request_context)
return {'success': False, 'form_html': form_html}
```

I'm using a jsonview decorator from [django-jsonview](#).

Note that in Django versions 1.8 and onwards, using `RequestContext` in this way will not work. Instead you can provide `render_crispy_form` with the necessary CSRF token with the following code:

```
from django.template.context_processors import csrf
ctx = {}
ctx.update(csrf(request))
form_html = render_crispy_form(form, context=ctx)
```

In our client side using jQuery would look like:

```
var example_form = '#example-form';

$.ajax({
  url: "{% url 'save_example_form' %}",
  type: "POST",
  data: $(example_form).serialize(),
  success: function(data) {
    if (!(data['success'])) {
      // Here we replace the form, for the
      $(example_form).replaceWith(data['form_html']);
    }
    else {
      // Here you can show the user a success message or do whatever you need
      $(example_form).find('.success-message').show();
    }
  },
  error: function () {
    $(example_form).find('.error-message').show()
  }
});
```

Obviously, you can adjust this snippets to your needs, or class based views or favorite frontend library.

Warning: When replacing form html, you need to bind events using `live` or `on` jQuery method.

1.3.9 Bootstrap3 horizontal forms

The way you do horizontal forms in Bootstrap version 3 is setting some `col-lg-X` classes in labels and divs wrapping fields. This would mean a lot of hassle updating your layout objects for settings these classes, too much verbosity. Instead some `FormHelper` attributes have been added to help you easily achieve this. You will need to set only three attributes:

```
helper.form_class = 'form-horizontal'
helper.label_class = 'col-lg-2'
helper.field_class = 'col-lg-8'
helper.layout = Layout(
    'email',
    'password',
    'remember_me',
    StrictButton('Sign in', css_class='btn-default'),
)
```

Of course you can set your widths as you like, it doesn't have to be exactly like this.

1.3.10 Bootstrap3 inline forms

The way you do inline forms in Bootstrap version 3 is:

```
helper.form_class = 'form-inline'
helper.field_template = 'bootstrap3/layout/inline_field.html'
helper.layout = Layout(
    'email',
    'password',
    'remember_me',
    StrictButton('Sign in', css_class='btn-default'),
)
```

If you need to set attributes in a field, you have to use `InlineField` instead of `Field`:

```
from crispy_forms.bootstrap import InlineField

helper.layout = Layout(
    InlineField('email', readonly=True),
    'password',
    [...]
)
```

1.4 FormHelper

What is a `FormHelper` and how to use it, is thoroughly explained in a previous section *{% crispy %} tag with forms*.

1.4.1 FormHelper with a form attached (Default layout)

Since version 1.2.0 `FormHelper` optionally can be passed an instance of a form. You would do it this way:

```
from crispy_forms.helper import FormHelper

class ExampleForm(forms.Form):
    def __init__(self, *args, **kwargs):
        super(ExampleForm, self).__init__(*args, **kwargs)
        self.helper = FormHelper(self)
```

When you do this `crispy-forms` builds a default layout using `form.fields` for you, so you don't have to manually list them all if your form is huge. If you later need to manipulate some bits of a big layout, using dynamic layouts is highly recommended, check *Updating layouts on the go*.

Also, now the helper is able to cross match the layout with the form instance, being able to search by widget type if you are using dynamic API.

1.4.2 Helper attributes you can set

template_pack Allows you to set what template pack you want to use at `FormHelper` level. This is useful for example when a website needs to render different styling forms for different use cases, like desktop website vs smartphone website.

template When set allows you to render a form/formset using a custom template. Default template is at `{{ TEMPLATE_PACK }}/whole_uni_form.html|whole_uni_formset.html`

field_template When set allows you to render a form/formset using a custom field template. Default template is at `{{ TEMPLATE_PACK }}/field.html`.

form_method = 'POST' Specifies form method attribute. You can see it to 'POST' or 'GET'. Defaults to 'POST'

form_action Applied to the form action attribute. Can be a named url in your URLconf that can be executed via the `{% url %}` template tag. Example: 'show_my_profile'. In your URLconf you could have something like:

```
url(r'^show/profile/$', 'show_my_profile_view', name='show_my_profile')
```

You can also point it to a URL `"/whatever/blabla/`.

Sometimes you may want to add arguments to the URL, for that you will have to do in your view:

```
from django.urls import reverse
form.helper.form_action = reverse('url_name', args=[event.id])
form.helper.form_action = reverse('url_name', kwargs={'book_id': book.id})
```

attrs Added in 1.2.0, a dictionary to set any kind of form attributes. Underscores in keys are translated into hyphens. The recommended way when you need to set several form attributes in order to keep your helper tidy:

```
``{'id': 'form-id', 'data_id': '/whatever'}``
<form id="form-id" data-id="/whatever" ...>
```

form_id Specifies form DOM id attribute. If no id provided then no id attribute is created on the form.

form_class String containing separated CSS classes to be applied to form class attribute. The form will always have by default ‘uniForm’ class.

form_tag = True It specifies if `<form></form>` tags should be rendered when using a Layout. If set to `False` it renders the form without the `<form></form>` tags. Defaults to `True`.

disable_csrf = False Disable CSRF token, when done, crispy-forms won’t use `{% csrf_token %}` tag. This is useful when rendering several forms using `{% crispy %}` tag and `form_tag = False` `csrf_token` gets rendered several times.

form_error_title If you are rendering a form using `{% crispy %}` tag and it has `non_field_errors` to display, they are rendered in a div. You can set the title of the div with this attribute. Example: “Form Errors”.

formset_error_title If you are rendering a formset using `{% crispy %}` tag and it has `non_form_errors` to display, they are rendered in a div. You can set the title of the div with this attribute. Example: “Formset Errors”.

form_style = ‘default’ Helper attribute for uni_form template pack. Uni-form has two different form styles built-in. You can choose which one to use, setting this variable to `default` or `inline`.

form_show_errors = True Default set to `True`. It decides whether to render or not form errors. If set to `False`, `form.errors` will not be visible even if they happen. You have to manually render them customizing your template. This allows you to customize error output.

render_unmentioned_fields = False By default django-crispy-forms renders the layout specified if it exists strictly, which means it only renders what the layout mentions, unless your form has `Meta.fields` and `Meta.exclude` defined, in that case it uses them. If you want to render unmentioned fields (all form fields), for example if you are worried about forgetting mentioning them you have to set this property to `True`. It defaults to `False`.

render_hidden_fields = False By default django-crispy-forms renders the layout specified if it exists strictly. Sometimes you might be interested in rendering all form’s hidden fields no matter if they are mentioned or not. It defaults to `False`.

render_required_fields = False By default django-crispy-forms renders the layout specified if it exists strictly. Sometimes you might be interested in rendering all form’s required fields no matter if they are mentioned or not. It defaults to `False`.

include_media = True By default django-crispy-forms renders all form media for you within the form. If you want to render form media yourself manually outside the form, set this to `False`. If you want to globally prevent rendering of form media, override the `FormHelper` class with this setting modified. It defaults to `True`.

1.4.3 Bootstrap Helper attributes

There are currently some helper attributes that only have functionality for a specific template pack. This doesn’t necessarily mean that they won’t be supported for other template packs in the future.

help_text_inline = False Sets whether help texts should be rendered inline or block. If set to `True` help texts will be rendered using `help-inline` class, otherwise using `help-block`. By default text messages are rendered in block mode.

error_text_inline = True Sets whether to render error messages inline or block. If set to `True` errors will be rendered using `help-inline` class, otherwise using `help-block`. By default error messages are rendered in inline mode.

html5_required = False When set to `True` all required fields inputs will be rendered with HTML5 `required=required` attribute.

form_show_labels = True Default set to `True`. Determines whether or not to render the form’s field labels.

1.4.4 Bootstrap 3 Helper attributes

All previous, `bootstrap` (version 2) attributes are also settable in `bootstrap 3` template pack `FormHelpers`. Here are listed the ones, that are only available in `bootstrap3` template pack:

label_class = '' Default set to `''`. This class will be applied to every label, this is very useful to do horizontal forms. Set it for example like this `label_class = col-lg-2`.

field_class = '' Default set to `''`. This class will be applied to every `div controls` wrapping a field. This is useful for doing horizontal forms. Set it for example like this `field_class = col-lg-8`.

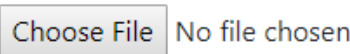
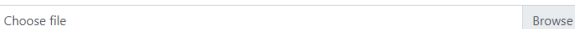
1.4.5 Bootstrap 4 Helper attributes

All previous, `bootstrap` (version 2 and 3) attributes are also settable in `bootstrap 4` template pack `FormHelpers`. Here are listed the ones, that are only available in `bootstrap4` template pack:

use_custom_control = True Enables the [optional UI customization](#) of the template pack for radio, checkbox and file field (non clearable only). Useful when you already have customization based on the default interpretation of the template pack. Setting to `False` results in the [standard bootstrap](#) classes being applied for radio and checkbox, and Django rendering for file field. See table below for examples.

The file field requires [additional JS](#) to enable its functionality, it is provided within the template pack as `vanilla JS`.

Defaults to `True`.

Standard	Optional
<p>Checkboxes*</p> <ul style="list-style-type: none"> <input checked="" type="checkbox"/> Option one is this and that be sure to include why it's great <input type="checkbox"/> Option two can also be checked and included in form results <input type="checkbox"/> Option three can yes, you guessed it also be checked and included in form results 	<p>Checkboxes*</p> <ul style="list-style-type: none"> <input checked="" type="checkbox"/> Option one is this and that be sure to include why it's great <input type="checkbox"/> Option two can also be checked and included in form results <input type="checkbox"/> Option three can yes, you guessed it also be checked and included in form results
<p>Radio buttons*</p> <ul style="list-style-type: none"> <input type="radio"/> Option one is this and that be sure to include why it's great <input checked="" type="radio"/> Option two can is something else and selecting it will deselect option one 	<p>Radio buttons*</p> <ul style="list-style-type: none"> <input type="radio"/> Option one is this and that be sure to include why it's great <input checked="" type="radio"/> Option two can is something else and selecting it will deselect option one
<p>File field*</p> 	<p>File field*</p> 

1.4.6 Custom Helper attributes

Maybe you would like `FormHelper` to do some extra thing that is not currently supported or maybe you have a very specific use case. The good thing is that you can add extra attributes and `crispy-forms` will automatically inject them within template context. Let's see an example, to make things clear.

We want some forms to have uppercase labels, and for that we would like to set a helper attribute name `labels_uppercase` to `True` or `False`. So we go and set in our helper:

```
helper.labels_uppercase = True
```

What will happen is `crispy-forms` will inject a Django template variable named `{{ labels_uppercase }}` with its corresponding value within its templates, including `field.html`, which is the template in charge of rendering a field when using `crispy-forms`. So we can go into that template and customize it. We will need to get familiar with it, but it's quite easy to follow; in the end it's just a Django template.

When we find where labels get rendered, this chunk of code to be more precise:

```
{% if field.label and not field|is_checkbox and form_show_labels %}
    <label for="{ field.id_for_label }" class="control-label {% if field.field.
↪required %}requiredField{% endif %}">
        {{ field.label|safe }}{% if field.field.required %}<span class="asteriskField
↪">*</span>{% endif %}
    </label>
{% endif %}
```

The line that we would change would end up like this:

```
{% if not labels_uppercase %}{{ field.label|safe }}{% else %}{{ field.
↪label|safe|upper }}{% endif %}{% if field.field.required %}
```

Now we only need to override field template, for that you may want to check section *Overriding layout objects templates*.

Warning: Be careful, depending on what you aim to do, sometimes using dynamic layouts is a better option, check section *Updating layouts on the go*.

1.5 Layouts

1.5.1 Fundamentals

Django-crispy-forms defines another powerful class called `Layout`, which allows you to change the way the form fields are rendered. This allows you to set the order of the fields, wrap them in divs or other structures, add html, set ids, classes or attributes to whatever you want, etc. And all that without writing a custom form template, using programmatic layouts. Just attach the layout to a helper, layouts are optional, but probably the most powerful thing django-crispy-forms has to offer.

A `Layout` is constructed by layout objects, which can be thought of as form components.

All these components are explained later in *Universal layout objects*, what you need to know now about them is that every component renders a different template and has a different purpose. Let's write a couple of different layouts for our form, continuing with our form class example (note that the full form is not shown again).

Some layout objects are specific to a template pack. For example `ButtonHolder` is for `uni_form` template_pack, while `FormActions` is for `bootstrap` template pack.

Let's add a layout to our helper:

```
from crispy_forms.helper import FormHelper
from crispy_forms.layout import Layout, Fieldset, ButtonHolder, Submit

class ExampleForm(forms.Form):
    [...]
    def __init__(self, *args, **kwargs):
        super(ExampleForm, self).__init__(*args, **kwargs)
        self.helper = FormHelper()
        self.helper.layout = Layout(
            Fieldset(
                'first arg is the legend of the fieldset',
                'like_website',
                'favorite_number',
```

(continues on next page)

(continued from previous page)

```

        'favorite_color',
        'favorite_food',
        'notes'
    ),
    ButtonHolder(
        Submit('submit', 'Submit', css_class='button white')
    )
)

```

When we render the form now using:

```

{% load crispy_forms_tags %}
{% crispy example_form %}

```

We will get the fields wrapped in a fieldset, whose legend will be set to ‘first arg is the legend of the fieldset’. The fields’ order will be: `like_website`, `favorite_number`, `favorite_color`, `favorite_food` and `notes`. We also get a submit button wrapped in a `<div class="buttonHolder">` which uni-form CSS positions in a nice way. That button has its CSS class set to `button white`.

This is just the tip of the iceberg: now imagine you want to add an explanation for what notes are, you can use HTML layout object:

```

Layout(
    Fieldset(
        'Tell us your favorite stuff {{ username }}',
        'like_website',
        'favorite_number',
        'favorite_color',
        'favorite_food',
        HTML("""
            <p>We use notes to get better, <strong>please help us {{ username }}</
→strong></p>
            """),
        'notes'
    )
)

```

As you’ll notice the fieldset legend is context aware and you can write it as if it were a chunk of a template where the form will be rendered. The HTML object will add a message before the notes input and it’s also context aware. Note how you can wrap layout objects into other layout objects. Layout objects `Fieldset`, `Div`, `MultiField` and `ButtonHolder` can hold other Layout objects within. Let’s do an alternative layout for the same form:

```

Layout(
    MultiField(
        'Tell us your favorite stuff {{ username }}',
        Div(
            'like_website',
            'favorite_number',
            css_id = 'special-fields'
        ),
        'favorite_color',
        'favorite_food',
        'notes'
    )
)

```

This time we are using a `MultiField`, which is a layout object that as a general rule can be used in the same places

as `Fieldset`. The main difference is that this renders all the fields wrapped in a `div` and when there are errors in the form submission, they are shown in a list instead of each one surrounding the field. Sometimes the best way to see what layout objects do, is just try them and play with them a little bit.

1.5.2 Layout objects attributes

All layout objects you can set kwargs that will be used as HTML attributes. For example if you want to turn autocomplete off for a field you can do:

```
Field('field_name', autocomplete='off')
```

If you want to set html attributes, with words separated by hyphens like `data-name`, as Python doesn't support hyphens in keyword arguments and hyphens are the usual notation in HTML, underscores will be translated into hyphens, so you would do:

```
Field('field_name', data_name="whatever")
```

As `class` is a reserved keyword in Python, for it you will have to use `css_class`. For example:

```
Field('field_name', css_class="black-fields")
```

And `id` attribute is set using `css_id`:

```
Field('field_name', css_id="custom_field_id")
```

1.5.3 Universal layout objects

These ones live in module `crispy_forms.layout`. These are layout objects that are not specific to a template pack. We'll go one by one, showing usage examples:

- **Div:** It wraps fields in a `<div>`:

```
Div('form_field_1', 'form_field_2', 'form_field_3', ...)
```

NOTE Mainly in all layout objects you can set kwargs that will be used as HTML attributes. As `class` is a reserved keyword in Python, for it you will have to use `css_class`. For example:

```
Div('form_field_1', style="background: white;", title="Explication title", css_class="bigdivs")
```

- **HTML:** A very powerful layout object. Use it to render pure html code. In fact it behaves as a Django template and it has access to the whole context of the page where the form is being rendered. This layout object doesn't accept any extra parameters than the html to render, you cannot set html attributes like in `Div`:

```
HTML("{% if success %} <p>Operation was successful</p> {% endif %}")
```

Warning: Beware that this is rendered in a standalone template, so if you are using custom templates or filters, don't forget to add your `{% load custom_tags %}`

- **Field:** Extremely useful layout object. You can use it to set attributes in a field or render a specific field with a custom template. This way you avoid having to explicitly override the field's widget and pass an ugly `attrs` dictionary:

```
Field('password', id="password-field", css_class="passwordfields", title=
↳"Explanation")
Field('slider', template="custom-slider.html")
```

This layout object can be used to easily extend Django's widgets. If you want to render a Django form field as hidden you can simply do:

```
Field('field_name', type="hidden")
```

If you need HTML5 attributes, you can easily do those using underscores `data_name` kwarg here will become into `data-name` in your generated html:

```
Field('field_name', data_name="special")
```

Fields in bootstrap are wrapped in a `<div class="control-group">`. You may want to set extra classes in this div, for that do:

```
Field('field_name', wrapper_class="extra-class")
```

- **Submit:** Used to create a submit button. First parameter is the name attribute of the button, second parameter is the value attribute:

```
Submit('search', 'SEARCH')
Submit('search', 'SEARCH')
```

Renders to:

```
<input type="submit" name="search" value="SEARCH" class="submit submitButton" id=
↳"submit-id-search" />
```

- **Hidden:** Used to create a hidden input:

```
Hidden('name', 'value')
```

- **Button:** Creates a button:

```
Button('name', 'value')
```

- **Reset:** Used to create a reset input:

```
reset = Reset('name', 'value')
```

- **Fieldset:** It wraps fields in a `<fieldset>`. The first parameter is the text for the fieldset legend, as we've said it behaves like a Django template:

```
Fieldset("Text for the legend {{ username }}",
    'form_field_1',
    'form_field_2'
)
```

1.5.4 Uni-form layout objects

These ones live in module `crispy_forms.layout`. Probably in the future they will be moved out to a `uni_form` module:

- **ButtonHolder:** It wraps fields in a `<div class="buttonHolder">`, which uni-form positions in a nice way. This is where form's submit buttons go in uni-form:

```
ButtonHolder(
    HTML("<span class='hidden'>✓ Saved data</span>"),
    Submit('save', 'Save')
)
```

- **MultiField:** It wraps fields in a `<div>` with a label on top. When there are errors in the form submission it renders them in a list instead of each one surrounding the field:

```
MultiField("Text for the label {{ username }}",
    'form_field_1',
    'form_field_2'
)
```

1.5.5 Bootstrap Layout objects

These ones live under module `crispy_forms.bootstrap`.

- **FormActions:** It wraps fields in a `<div class="form-actions">`. It is usually used to wrap form's buttons:

```
FormActions(
    Submit('save', 'Save changes'),
    Button('cancel', 'Cancel')
)
```



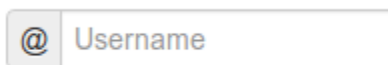
- **AppendedText:** It renders a bootstrap appended text input. The first parameter is the name of the field to add appended text to, then the appended text which can be HTML like. There is an optional parameter `active`, by default set to `False`, that you can set to a boolean to render appended text active:

```
AppendedText('field_name', 'appended text to show')
AppendedText('field_name', '$', active=True)
```



- **PrependedText:** It renders a bootstrap prepended text input. The first parameter is the name of the field to add prepended text to, then the prepended text which can be HTML like. There is an optional parameter `active`, by default set to `False`, that you can set to a boolean to render prepended text active:

```
PrependedText('field_name', '<b>Prepended text</b> to show')
PrependedText('field_name', '@', placeholder="username")
```



- **PrependedAppendedText:** It renders a combined prepended and appended text. The first parameter is the name of the field, then the prepended text and finally the appended text:

```
PrependedAppendedText('field_name', '$', '.00'),
```

- **InlineCheckboxes:** It renders a Django forms.MultipleChoiceField field using inline checkboxes:

```
InlineCheckboxes('field_name')
```

Option one Option two Option three

- **InlineRadios:** It renders a Django forms.ChoiceField field with its widget set to forms.RadioSelect using inline radio buttons:

```
InlineRadios('field_name')
```

Option one Option two

- **StrictButton:** It renders a button using <button> html, not input. By default type is set to button and class is set to btn:

```
StrictButton("Button's content", name="go", value="go", css_class="extra")
StrictButton('Success', css_class="btn-success")
```

Success

- **FieldWithButtons:** You can create an input connected with buttons:

```
FieldWithButtons('field_name', StrictButton("Go!"))
```

- **Tab & TabHolder:** Tab renders a tab, different tabs need to be wrapped in a TabHolder for automatic javascript functioning, also you will need bootstrap-tab.js included in your static files:

```
TabHolder(
    Tab('First Tab',
        'field_name_1',
        Div('field_name_2')
    ),
    Tab('Second Tab',
        Field('field_name_3', css_class="extra")
    )
)
```

First Tab **Second Tab**

Textarea

- **Accordion & AccordionGroup:** `AccordionGroup` renders an accordion pane, different groups need to be wrapped in an `Accordion` for automatic javascript functioning, also you will need `bootstrap-tab.js` included in your static files:

```
Accordion(
    AccordionGroup('First Group',
        'radio_buttons'
    ),
    AccordionGroup('Second Group',
        Field('field_name_3', css_class="extra")
    )
)
```

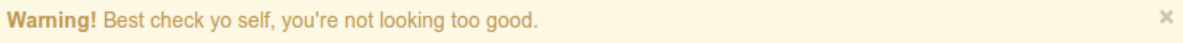
First Group

Radio buttons Option one is this and that be sure to include why it's great
 Option two can is something else and selecting it will deselect option one

Second Group

- **Alert:** `Alert` generates markup in the form of an alert dialog:

```
Alert(content='<strong>Warning!</strong> Best check yo self, you're not looking_
↳too good.')
```



- **UneditableField:** `UneditableField` renders a disabled field using the bootstrap `uneditable-input` class:

```
UneditableField('text_input', css_class='form-control-lg')
```

Text input*

1.5.6 Overriding layout objects templates

The mentioned set of *Universal layout objects* has been thoroughly designed to be flexible, standard compatible and support Django form features. Every Layout object is associated to a different template that lives in `templates/{ { TEMPLATE_PACK_NAME } }/layout/` directory.

Some advanced users may want to use their own templates, to adapt the layout objects to their use or necessities. There are three ways to override the template that a layout object uses.

- **Globally:** You override the template of the layout object, for all instances of that layout object you use:

```
from crispy_forms.layout import Div
Div.template = 'my_div_template.html'
```

- **Individually:** You can override the template for a specific layout object in a layout:

```
Layout (
    Div(
        'field1',
        'field2',
        template='my_div_template.html'
    )
)
```

- **Overriding templates directory:** This means mimicking crispy-forms directory structure into your project and then copying the templates that you want to override there, finally editing those copies. If you are using this approach it's better to just copy and edit templates you will customize instead of all.

1.5.7 Overriding project templates

You need to differentiate between layout objects' templates and django-crispy-forms templates. There are some templates that live in `templates/{{ TEMPLATE_PACK_NAME }}` that define the form/formset structure, how a field or errors are rendered, etc. They add very little logic and are pretty much basic wrappers for the rest of django-crispy-forms power. To override these ones you have two options:

- **template** and **field_template** attributes in `FormHelper`: Since version 1.3.0 you can override the form/formset template and the field template using helper attributes, see section *Helper attributes you can set*. With this you can change one specific form or all your project forms (creating a custom `FormHelper` base class for example).
- **Overriding templates directory:** This works the same as explained in section *Overriding layout objects templates*. If you are adapting crispy-forms templates to a popular open source template pack you use, submit it so more people can benefit from it.
- **Creating a TEMPLATE PACK:** You maybe want to use crispy-forms with you favorite CSS framework or your Company's CSS. For doing so, you will need to be quite familiar with crispy-forms, layout objects and their templates. You will probably want to start off with one of the existing template packs, probably `bootstrap`. Imagine your template pack is named `chocolate`, that means you probably want your root directory named the same way. For using your template pack, you will have to set `CRISPY_TEMPLATE_PACK = 'chocolate'` variable in your settings file and also set `CRISPY_ALLOWED_TEMPLATE_PACKS = ('bootstrap', 'chocolate')`. This way crispy-forms will know you want to use your own template pack, which is an allowed one and where to look for it.

1.5.8 Creating your own layout objects

The *Universal layout objects* bundled with django-crispy-forms are a set of the most seen components that build a form. You will probably be able to do anything you need combining them. Anyway, you may want to create your own components, for doing that, you will need a good grasp of django-crispy-forms. Every layout object must have a method called `render`. Its prototype should be:

```
def render(self, form, form_style, context):
```

The official layout objects live in `layout.py` and `bootstrap.py`, you may want to have a look at them to fully understand how to proceed. But in general terms, a layout object is a template rendered with some parameters passed.

If you come up with a good idea and design a layout object you think others could benefit from, please open an issue or send a pull request, so django-crispy-forms gets better.

1.5.9 Composing layouts

Imagine you have several forms that share a big chunk of the same layout. There is a easy way you can create a `Layout`, reuse and extend it. You can have a `Layout` as a component of another `Layout`. You can build that common chunk, different ways. As a separate class:

```
class CommonLayout(Layout):
    def __init__(self, *args, **kwargs):
        super(CommonLayout, self).__init__(
            MultiField("User data",
                'username',
                'lastname',
                'age'
            )
        )
```

Maybe an object instance is good enough:

```
common_layout = Layout(
    MultiField("User data",
        'username',
        'lastname',
        'age'
    )
)
```

Then you can do:

```
helper.layout = Layout(
    CommonLayout(),
    Div(
        'favorite_food',
        'favorite_bread',
        css_id = 'favorite-stuff'
    )
)
```

Or:

```
helper.layout = Layout(
    common_layout,
    Div(
        'professional_interests',
        'job_description',
    )
)
```

We have defined a layout and used it as a chunk of another layout, which means that those two layouts will start the same way and then extend the layout in different ways.

1.6 How to create your own template packs

First you will have to name your template pack, for this you can't use the name of one of the available template packs in `crispy-forms`, due to name collisions. For example, let's say in the company we work for, a designer has come up with a CSS bootstrap internally known as `chocolate`. The company has a Django project that needs to start using

chocolate, therefore we need to create a folder named `chocolate` within our templates directory. Check your `TEMPLATE_DIRS` setting in Django and pick your preferred path.

Once we have that folder created, we will have to create a concrete directory hierarchy so that crispy-forms can pick it up. This is what bootstrap template pack (v2) looks like:

```
.
├── accordion-group.html
├── accordion.html
├── betterform.html
├── display_form.html
├── errors.html
├── errors_formset.html
├── * field.html
├── layout
│   ├── alert.html
│   ├── * baseinput.html
│   ├── button.html
│   ├── checkboxselectmultiple.html
│   ├── checkboxselectmultiple_inline.html
│   ├── div.html
│   ├── field_errors.html
│   ├── field_errors_block.html
│   ├── field_with_buttons.html
│   ├── fieldset.html
│   ├── formactions.html
│   ├── help_text.html
│   ├── help_text_and_errors.html
│   ├── multifield.html
│   ├── prepended_appended_text.html
│   ├── radioselect.html
│   ├── radioselect_inline.html
│   ├── tab-link.html
│   ├── tab.html
│   └── uneditable_input.html
├── table_inline_formset.html
├── * uni_form.html
├── uni_formset.html
├── * whole_uni_form.html
└── whole_uni_formset.html
```

Take it easy, don't panic, we won't need this many templates for our template pack. Templates are also quite simple to follow if you understand what problem crispy-forms solves. The bare minimum would be the templates marked with an asterisk.

1.6.1 Fundamentals

First, since crispy-forms 1.5.0, template packs are self contained, you cannot reference a template from a different template pack.

crispy-forms has many features, but maybe you don't need your template pack to cover all of them. `{% crispy %}` templatetag renders forms using a global structure contained within `whole_uni_form.html`. However, `|crispy` filter uses `uni_form.html`. As you've probably guessed, the name of the templates comes from the old days of `django-uni-form`. Anyway, as an example, if we don't use `|crispy` filter, we don't really need to maintain a `uni_form.html` template within our template pack.

If we are planning on using formsets + `{% crispy %}` we will need a `whole_uni_formset.html`, instead if we use formsets + `|crispy` we will need `uni_formset.html`.

All of these templates use a tag named `{% crispy_field %}` that is loaded doing `{% load crispy_forms_field %}`, that generates the html for `<input>` using `field.html` template, but previously doing Python preparation beforehand. In case you wonder the code for this tag lives in `crispy_forms.templatetags.crispy_forms_field`, together with some other stuff.

So a template pack for a very basic example covering only forms and the usage of `{% crispy %}` tag, would need 2 templates: `whole_uni_form.html`, `field.html`. Well, that's not completely true, because every layout object has a template attached. So if we wanted to use `Div`, we would need `div.html`. Some are not that obvious, if you need `Submit`, you will need `baseinput.html`. Some layout objects, don't really have a template attached, like `HTML`.

In the previous template tree, there are some templates that are there for DRY purposes, they are not really compulsory or part of a layout object, so don't worry too much.

1.6.2 Starting

Now your best bet is probably start copying some or all of the templates under an existing crispy-forms template pack, such as `bootstrap3`, then drop the ones you don't need. Next step is edit those templates, and adjust the HTML/CSS making it align with `chocolate`, that sometimes means dropping/adding divs, classes and other stuff around. You can always create a form in your application, with a helper attached to that new template pack and start trying out your adaptation right away.

Currently, there is an existing template pack for crispy-forms that doesn't live in core, developed by David Thenon as an external pluggable application named `crispy-forms-foundation`, it's also a good reference to check out.

Beware that crispy-forms evolves and adds new `FormHelper.attributes`, if you want to use those in the future you will have to adapt your templates adding those variables and its handling.

1.7 {% crispy %} tag with formsets

`{% crispy %}` tag supports formsets rendering too, all kind of Django formsets: `formsets`, `modelformsets` and `inline formsets`. In this section, it's taken for granted that you are familiar with previous explained crispy-forms concepts in the docs, like `FormHelper`, how to set `FormHelper` attributes or render a simple form using `{% crispy %}` tag.

1.7.1 Formsets

It's not the purpose of this documentation to explain how formsets work in detail, for that you should check [Django official formset docs](#). Let's start creating a formset using the previous `ExampleForm` form:

```
from django.forms.models import formset_factory

ExampleFormSet = formset_factory(ExampleForm, extra=3)
formset = ExampleFormSet()
```

This is how you would render the formset using default rendering, no layouts or form helpers:

```
{% crispy formset %}
```

Of course, you can still use a helper, otherwise there would be nothing crispy in this. When using a `FormHelper` with a formset compared to when you use it with a form, the main difference is that helper attributes are applied to the form structure, while the layout is applied to the formset's forms. Let's create a helper for our `ExampleFormSet`:

```
class ExampleFormSetHelper (FormHelper):
    def __init__(self, *args, **kwargs):
        super(ExampleFormSetHelper, self).__init__(*args, **kwargs)
        self.form_method = 'post'
        self.layout = Layout(
            'favorite_color',
            'favorite_food',
        )
        self.render_required_fields = True
```

This helper is quite easy to follow. We want our form to use POST method, and we want `favorite_color` to be the first field, then `favorite_food` and finally we tell crispy to render all required fields after. Let's go and use it, when using `{% crispy %}` tag in a template there is one main difference when rendering formsets vs forms, in this case you need to specify the helper explicitly.

This would be part of an hypothetical function view:

```
formset = ExampleFormSet()
helper = ExampleFormSetHelper()
return render(request, 'template.html', {'formset': formset, 'helper': helper})
```

Then in `template.html` you would have to do:

```
{% crispy formset helper %}
```

There are two ways you can add submit buttons to a formset. Using `FormHelper.add_input` method:

```
helper.add_input(Submit("submit", "Save"))
```

Or you can set `FormHelper.form_tag` to `False` and control the formset outer structure at your will writing boring HTML:

```
<form action="{% url 'save_formset' %}" method="POST">
  {% crispy formset helper %}
  <div class="form-actions">
    <input type="submit" name="submit" value="Save" class="btn btn-primary" id=
    ↪ "submit-save">
  </div>
</form>
```

Finally, model formsets and inline formsets are rendered exactly the same as formsets, the only difference is how you build them in your Django code.

1.7.2 Extra context

Rendering any kind of formset with crispy injects some extra context in the layout rendering so that you can do things like:

```
HTML("{% if forloop.first %}Message displayed only in the first form of a formset_
↪ forms list{% endif %}",
Fieldset("Item {{ forloop.counter }}", 'field-1', [...])
```

Basically you can access a `forloop` Django node, as if you were rendering your formsets forms using a for loop.

1.7.3 Custom templates and table inline formsets

Default formset template will render your formset's form using divs, but many times people prefer tables for formsets. Don't worry, `crispy-forms`'s got you covered. `FormHelper` has an attribute named `template` that can be used to specify a custom template for rendering a form or formset, in this case a formset. Obviously when we specify a `template` attribute, we are making that helper only usable with forms or formsets.

The name of the template to use is `table_inline_formset.html` and you use it doing:

```
helper.template = 'bootstrap/table_inline_formset.html'
```

The best part is that if this template doesn't do exactly what you want, you can copy it into your templates folder, customize it and then link your helper to your alternative version. If you think what you are missing would be valuable to others, then please submit a pull request at github.

Warning: This template doesn't currently take into account any layout you have specified and only works with bootstrap template pack.

1.7.4 Formset forms with different layouts

By default `crispy-forms` formset rendering shares the same layout among all formset's forms. This is the case 99% of the times. But maybe you want to render your formset's forms using different layouts that you cannot achieve using the extra context injected, for that you will have to create and use a custom template. Most likely you will want to do:

```
{{ formset.management_form|crispy }}
{% for form in formset %}
    {% crispy form %}
{% endfor %}
```

Where every `form` has a `helper` attribute from which `crispy` will grab the layout. In your view you will need to change the layout or use a different help for every formset's form. Make sure that you have `form_tag` attribute set to `False`, otherwise you will get 3 individual forms rendered.

1.8 Updating layouts on the go

Layouts can be changed, adapted and generated programmatically.

The next sections will explain how to select parts of a layout and update them. We will use this API from the `FormHelper` instance and not the layout itself. This API's basic behavior consists of selecting the piece of the layout to manipulate and chaining methods that alter it after that.

1.8.1 Selecting layout objects with slices

You can get a slice of a layout using familiar `[]` Python operator:

```
form.helper[1:3]
form.helper[2]
form.helper[:-1]
```

You can basically do all kind of slices, the same ones supported by Python's lists. You can also concatenate them. If you had this layout:

```
Layout (
    Div('email')
)
```

You could access 'email' string doing:

```
form.helper[0][0]
```

1.8.2 wrap

One useful action you can apply on a slice is `wrap`, which wraps every selected field using a layout object type and parameters passed. Let's see an example. If We had this layout:

```
Layout (
    'field_1',
    'field_2',
    'field_3'
)
```

We could do:

```
form.helper[1:3].wrap(Field, css_class="hello")
```

We would end up having this layout:

```
Layout (
    'field_1',
    Field('field_2', css_class='hello'),
    Field('field_3', css_class='hello')
)
```

Note how `wrap` affects each layout object selected, if you would like to wrap `field_2` and `field_3` together in a `Field` layout object you will have to use `wrap_together`.

Beware that the slice `[1:3]` only looks in the first level of depth of the layout. So if the previous layout was this way:

```
Layout (
    'field_1',
    Div('field_2'),
    'field_3'
)
```

`helper[1:3]` would return this layout:

```
Layout (
    'field_1',
    Field(Div('field_2'), css_class="hello"),
    Field('field_3', css_class="hello")
)
```

Parameters passed to `wrap` or `wrap_together` will be used for creating the layout object that is wrapping selected fields. You can pass `args` and `kwargs`. If you are using a layout object like `Fieldset` which needs a string as compulsory first argument, `wrap` will not work as desired unless you provide the text of the legend as an argument to `wrap`. Let's see a valid example:


```
form.helper[1:3].wrap(Fieldset, "legend of the fieldset")
```

Also you can pass args and kwargs:

```
form.helper[1:3].wrap(Fieldset, "legend of the fieldset", css_class="fieldsets")
```

1.8.3 wrap_together

`wrap_together` wraps a whole slice within a layout object type with parameters passed. Let's see an example. If we had this layout:

```
Layout(
    'field_1',
    'field_2',
    'field_3'
)
```

We could do:

```
form.helper[0:3].wrap_together(Field, css_class="hello")
```

We would end up having this layout:

```
Layout(
    Field(
        'field_1',
        'field_2',
        'field_3',
        css_class='hello'
    )
)
```

1.8.4 update_attributes

Updates attributes of every layout object contained in a slice:

```
Layout(
    'field_1',
    Field('field_2'),
    Field('field_3')
)
```

We could do:

```
form.helper[0:3].update_attributes(css_class="hello")
```

Layout would turn into:

```
Layout(
    'field_1',
    Field('field_2', css_class='hello'),
    Field('field_3', css_class='hello')
)
```

We can also apply it to a field name wrapped in a layout object:

```
form.helper['field_2'].update_attributes(css_class="hello")
```

However, the following wouldn't be correct:

```
form.helper['field_1'].update_attributes(css_class="hello")
```

Because it would change Layout attrs. It's your job to have it wrapped correctly.

1.8.5 all

This method selects all first level of depth layout objects:

```
form.helper.all().wrap(Field, css_class="hello")
```

1.8.6 Selecting a field name

If you pass a string with the field name, this field name will be searched greedy throughout the whole Layout depth levels. Imagine we have this layout:

```
Layout (
    'field_1',
    Div(
        Div('password')
    ),
    'field_3'
)
```

If we do:

```
form.helper['password'].wrap(Field, css_class="hero")
```

Previous layout would become:

```
Layout (
    'field_1',
    Div(
        Div(
            Field('password', css_class="hero")
        )
    ),
    'field_3'
)
```

1.8.7 filter

This method will allow you to filter layout objects by its class type, applying actions to them:

```
form.helper.filter(basestring).wrap(Field, css_class="hello")
form.helper.filter(Div).wrap(Field, css_class="hello")
```

You can filter several layout objects types at the same time:

```
form.helper.filter(basestring, Div).wrap(Div, css_class="hello")
```

By default `filter` is not greedy, so it only searches first depth level. But you can tune it to search in different levels of depth with a kwarg `max_level` (By default set to 0). Let's see some examples, to clarify it. Imagine we have this layout:

```
Layout (
    'field_1',
    Div(
        Div('password')
    ),
    'field_3'
)
```

If we did:

```
form.helper.filter(basestring).wrap(Field, css_class="hello")
```

Only `field_1` and `field_3` would be wrapped, resulting into:

```
Layout (
    Field('field_1', css_class="hello"),
    Div(
        Div('password')
    ),
    Field('field_3', css_class="hello"),
)
```

If we wanted to search deeper, wrapping `password`, we would need to set `max_level` to 2 or more:

```
form.helper.filter(basestring, max_level=2).wrap(Field, css_class="hello")
```

In other words `max_level` indicates the number of jumps crispy-forms can do within a layout object for matching. In this case getting into the first `Div` would be one jump, and getting into the next `Div` would be the second jump, thus `max_level=2`.

We can turn `filter` greedy, making it search as deep as possible, setting `greedy` to `True`:

```
form.helper.filter(basestring, greedy=True).wrap(Div, css_class="hello")
```

Parameters:

- `max_level`: An integer representing the number of jumps that crispy-forms should do when filtering. Defaults to 0.
- `greedy`: A boolean that indicates whether to filter greedy or not. Defaults to `False`.

1.8.8 filter_by_widget

Matches all fields of a widget type. This method assumes you are using a helper with a form attached, see section *FormHelper with a form attached (Default layout)*, you could filter by widget type doing:

```
form.helper.filter_by_widget(forms.PasswordInput).wrap(Field, css_class="hero")
```

`filter_by_widget` is greedy by default, so it searches in depth. Let's see a use case example, imagine we have this Layout:

```
Layout (
    'username',
    Div('password1'),
    Div('password2')
)
```

Supposing `password1` and `password2` fields are using widget `PasswordInput`, would turn into:

```
Layout (
    'username',
    Div(Field('password1', css_class="hero")),
    Div(Field('password2', css_class="hero"))
)
```

An interesting real use case example here would be to wrap all `SelectInputs` with a custom made `ChosenField` that renders the field using a `chosenjs` compatible field.

1.8.9 `exclude_by_widget`

Excludes all fields of a widget type. This method assumes you are using a helper with a form attached, see section *FormHelper with a form attached (Default layout)*:

```
form.helper.exclude_by_widget(forms.PasswordInput).wrap(Field, css_class="hero")
```

`exclude_by_widget` is greedy by default, so it searches in depth. Let's see a use case example, imagine we have this `Layout`:

```
Layout (
    'username',
    Div('password1'),
    Div('password2')
)
```

Supposing `password1` and `password2` fields are using widget `PasswordInput`, would turn into:

```
Layout (
    Field('username', css_class="hero"),
    Div('password1'),
    Div('password2')
)
```

1.8.10 Manipulating a layout

Besides selecting layout objects and applying actions to them, you can also manipulate layouts themselves and layout objects easily, like if they were lists. We won't do this from the helper, but the layout and layout objects themselves. Consider this a lower level API.

All layout objects that can wrap others, contain a inner attribute `fields` which is a list, not a dictionary as in Django forms. You can apply any list methods on them easily. Beware that a `Layout` behaves itself like other layout objects such as `Div`, the only difference is that it is the root of the tree.

This is how you would replace a layout object for other:

```
layout[0][3][1] = Div('field_1')
```

This is how you would add one layout object at the end of the Layout:

```
layout.append(HTML("<p>whatever</p>"))
```

This is how you would add one layout object at the end of another layout object:

```
layout[0].append(HTML("<p>whatever</p>"))
```

This is how you would add several layout objects to a Layout:

```
layout.extend([
    HTML("<p>whatever</p>"),
    Div('add_field_on_the_go')
])
```

This is how you would add several layout objects to another layout object:

```
layout[0][2].extend([
    HTML("<p>whatever</p>"),
    Div('add_field_on_the_go')
])
```

This is how you would delete the second layout object within the Layout:

```
layout.pop(1)
```

This is how you would delete the second layout object within the second layout object:

```
layout[1].pop(1)
```

This is how you would insert a layout object in the second position of a Layout:

```
layout.insert(1, HTML("<p>whatever</p>"))
```

This is how you would insert a layout object in the second position of the second layout object:

```
layout[1].insert(1, HTML("<p>whatever</p>"))
```

Warning: Remember always that if you are going to manipulate a helper or layout in a view or any part of your code, you better use an instance level variable.

1.9 Frequently Asked Questions

- *Technical*
 - *Displaying columns in a bootstrap Layout*
- *General*
 - *Why use django-crispy-forms and not other app*
 - *How did this all get started?*

- *How fast is django-crispy-forms*
- *Which versions of Python does this support?*
- *Which versions of Django does this support?*

1.9.1 Technical

Displaying columns in a bootstrap Layout

Of course you can display form fields within columns, the same way you would do it in a standard bootstrap form, you can do it in a bootstrap layout, see an [example](#).

1.9.2 General

Why use django-crispy-forms and not other app

Well, I'm obviously biased for answering this question. But I once [answered it at StackOverflow](#).

How did this all get started?

In December 2008, while [Daniel Greenfeld](#) was working for NASA's Science Mission Directorate, his team began to use Django and Pinax. There was a necessity to make all the forms in Pinax [Section 508](#) compatible, and the thought of going through all of forms and rewriting `{{ form }}` as a block of `{% for field in form %}` with all the template logic seemed like way too much work.

So with the encouragement of [Katie Cunningham](#), [James Tauber](#) and [Jannis Leidel](#) Daniel took the Django docs on forms and combined it with Dragan Babic's excellent Uni-Form css/javascript library and created the ubiquitous `as_uni_form` filter. After that, fixing all the forms in Pinax to be section 508 compliant was trivial.

Not long before PyCon 2009 James Tauber suggested the `{% uni_form form helper %}` API, where one could trivially create forms without writing any HTML.

At PyCon 2009 Jannis Leidel helped Daniel through releasing the 0.3 release of django-uni-form on PyPI. It was also at that PyCon when the project moved from Google Code to Github.

Around January 2011 the project wasn't very active, Github issues and forks were stacking up. At that time [Miguel Araujo](#) found django-uni-form and loved the concept behind its architecture. He started working in a fork of the project, trying to gather some old submitted patches. Around march of 2011, after conversations with Daniel, he got commit powers in the project's repository, reactivating dev branch. Releases 0.8.0, 0.9.0 followed and the project more than doubled its watchers in Github.

By the end of 2011, Miguel and Daniel agreed on the necessity of renaming the project. As uni-form CSS framework was not anymore the only option available and the name was confusing the users. Thus django-crispy-forms was born, named by [Audrey Roy](#). The project is now actively maintained and leded by [Miguel Araujo](#).

How fast is django-crispy-forms

Performance in form rendering is normally a nigh moot issue in Django, because the majority of speed issues are fixable via appropriate use of Django's cache engine. Templates and especially form rendering are usually the last things to worry about when you try to increase performance.

However, because we do care about producing lean and fast code, work is being done to speed up and measure performance of this library. These are the average times of rendering 1000 forms with the latest django-crispy-forms code in Dell Latitude E6500 Intel Core 2 Duo @ 2.53GHz.

In production environments, you will want to activate template caching, see *Installing django-crispy-forms*.

Method	Time with template caching
Plain Django	0.915469169617 sec
crispy filter	4.23220916295 sec
{% crispy %} tag	4.53284406662 sec

Which versions of Python does this support?

Versions supported include Python 2.6.x, 2.7.x, Python 3.3.x. If you need greater backwards compatibility, django-crispy-forms below 1.3 supports 2.5.x, and django-uni-form 0.7.0 supports Python 2.4.x.

Which versions of Django does this support?

Versions supported include Django 1.3 or higher. Versions of django-crispy-forms below 1.3 support Django 1.2.x. If you need to support earlier versions you will need to use django-uni-form 0.7.0.

- See who's contributed to the project at [crispy-forms contributors](#)
- You can find a detailed history of the project in [Github's CHANGELOG](#)

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

2.1 API helpers

class `helper.FormHelper` (*form=None*)

This class controls the form rendering behavior of the form passed to the `{% crispy %}` tag. For doing so you will need to set its attributes and pass the corresponding helper object to the tag:

```
{% crispy form form.helper %}
```

Let's see what attributes you can set and what form behaviors they apply to:

form_method: Specifies form method attribute. You can set it to 'POST' or 'GET'. Defaults to 'POST'

form_action: Applied to the form action attribute:

- Can be a named url in your URLconf that can be executed via the `{% url %}` template tag. Example: 'show_my_profile'. In your URLconf you could have something like:

```
url(r'^show/profile/$', 'show_my_profile_view', name = 'show_my_
↳profile')
```

- It can simply point to a URL '/whatever/blahla/'.

form_id: Generates a form id for dom identification. If no id provided then no id attribute is created on the form.

form_class: String containing separated CSS classes to be applied to form class attribute. The form will always have by default 'uniForm' class.

form_group_wrapper_class: String containing separated CSS classes to be applied to each row of inputs.

form_tag: It specifies if `<form></form>` tags should be rendered when using a `Layout`. If set to `False` it renders the form without the `<form></form>` tags. Defaults to `True`.

form_error_title: If a form has *non_field_errors* to display, they are rendered in a div. You can set title's div with this attribute. Example: "Oooops!" or "Form Errors"

formset_error_title: If a formset has *non_form_errors* to display, they are rendered in a div. You can set title's div with this attribute.

form_style: Uni-form has two built in different form styles. You can choose your favorite. This can be set to "default" or "inline". Defaults to "default".

include_media: Whether to automatically include form media. Set to `False` if you want to manually include form media outside the form. Defaults to `True`.

Public Methods:

add_input(input): You can add input buttons using this method. Inputs added using this method will be rendered at the end of the form/formset.

add_layout(layout): You can add a *Layout* object to *FormHelper*. The `Layout` specifies in a simple, clean and DRY way how the form fields should be rendered. You can wrap fields, order them, customize pretty much anything in the form.

Best way to add a helper to a form is adding a property named `helper` to the form that returns customized *FormHelper* object:

```
from crispy_forms.helper import FormHelper
from crispy_forms.layout import Submit

class MyForm(forms.Form):
    title = forms.CharField(_("Title"))

    @property
    def helper(self):
        helper = FormHelper()
        helper.form_id = 'this-form-rocks'
        helper.form_class = 'search'
        helper.add_input(Submit('save', 'save'))
        [...]
        return helper
```

You can use it in a template doing:

```
{% load crispy_forms_tags %}
{% crispy form %}
```

get_attributes (*template_pack=<SimpleLazyObject: 'bootstrap'>*)
Used by `crispy_forms_tags` to get helper attributes

render_layout (*form, context, template_pack=<SimpleLazyObject: 'bootstrap'>*)
Returns safe html of the rendering of the layout

2.2 API Layout

class `layout.BaseInput` (*name, value, **kwargs*)
A base class to reduce the amount of code in the `Input` classes.

render (*form, form_style, context, template_pack=<SimpleLazyObject: 'bootstrap'>, **kwargs*)
 Renders an `<input />` if container is used as a Layout object. Input button value can be a variable in context.

class `layout.Button` (**args, **kwargs*)
 Used to create a Submit input descriptor for the `{% crispy %}` template tag:

```
button = Button('Button 1', 'Press Me!')
```

Note: The first argument is also slugified and turned into the id for the button.

class `layout.ButtonHolder` (**fields, **kwargs*)
 Layout object. It wraps fields in a `<div class="buttonHolder">`

This is where you should put Layout objects that render to form buttons like Submit. It should only hold *HTML* and *BaseInput* inherited objects.

Example:

```
ButtonHolder(
    HTML(<span style="display: hidden;">Information Saved</span>),
    Submit('Save', 'Save')
)
```

class `layout.Column` (**fields, **kwargs*)
 Layout object. It wraps fields in a div so the wrapper can be used as a column. Example:

```
Column('form_field_1', 'form_field_2')
```

Depending on the template, css class associated to the div is `formColumn`, `row`, or nothing. For this last case, you must provide css classes. Example:

```
Column('form_field_1', 'form_field_2', css_class='col-xs-6',)
```

class `layout.Div` (**fields, **kwargs*)
 Layout object. It wraps fields in a `<div>`

You can set `css_id` for a DOM id and `css_class` for a DOM class. Example:

```
Div('form_field_1', 'form_field_2', css_id='div-example', css_class='divs')
```

class `layout.Field` (**args, **kwargs*)
 Layout object, It contains one field name, and you can add attributes to it easily. For setting class attributes, you need to use `css_class`, as `class` is a Python keyword.

Example:

```
Field('field_name', style="color: #333;", css_class="whatever", id="field_name")
```

class `layout.Fieldset` (*legend, *fields, **kwargs*)
 Layout object. It wraps fields in a `<fieldset>`

Example:

```
Fieldset("Text for the legend",
        'form_field_1',
        'form_field_2'
    )
```

The first parameter is the text for the fieldset legend. This text is context aware, so you can do things like:

```
Fieldset("Data for {{ user.username }}",
        'form_field_1',
        'form_field_2'
    )
```

class `layout.HTML` (*html*)

Layout object. It can contain pure HTML and it has access to the whole context of the page where the form is being rendered.

Examples:

```
HTML("{% if saved %}Data saved{% endif %}")
HTML('<input type="hidden" name="{{ step_field }}" value="{{ step0 }}" />')
```

class `layout.Hidden` (*name, value, **kwargs*)

Used to create a Hidden input descriptor for the `{% crispy %}` template tag.

class `layout.Layout` (**fields*)

Form Layout. It is conformed by Layout objects: *Fieldset*, *Row*, *Column*, *MultiField*, *HTML*, *ButtonHolder*, *Button*, *Hidden*, *Reset*, *Submit* and fields. Form fields have to be strings. Layout objects *Fieldset*, *Row*, *Column*, *MultiField* and *ButtonHolder* can hold other Layout objects within. Though *ButtonHolder* should only hold *HTML* and *BaseInput* inherited classes: *Button*, *Hidden*, *Reset* and *Submit*.

Example:

```
helper.layout = Layout(
    Fieldset('Company data',
            'is_company'
    ),
    Fieldset(_('Contact details'),
            'email',
            Row('password1', 'password2'),
            'first_name',
            'last_name',
            HTML(''),
            'company'
    ),
    ButtonHolder(
        Submit('Save', 'Save', css_class='button white'),
    ),
)
```

class `layout.MultiField` (*label, *fields, **kwargs*)

MultiField container. Renders to a MultiField `<div>`

class `layout.MultiWidgetField` (**args, **kwargs*)

Layout object. For fields with `MultiWidget` as *widget*, you can pass additional attributes to each widget.

Example:

```
MultiWidgetField(
    'multiwidget_field_name',
    attrs=(
        {'style': 'width: 30px;'},
        {'class': 'second_widget_class'}
    ),
)
```

Note: To override widget's css class use `class` not `css_class`.

class `layout.Reset` (**args, **kwargs*)

Used to create a Reset button input descriptor for the `{% crispy %}` template tag:

```
reset = Reset('Reset This Form', 'Revert Me!')
```

Note: The first argument is also slugified and turned into the id for the reset.

class `layout.Row` (**fields, **kwargs*)

Layout object. It wraps fields in a div whose default class is “formRow”. Example:

```
Row('form_field_1', 'form_field_2', 'form_field_3')
```

class `layout.Submit` (**args, **kwargs*)

Used to create a Submit button descriptor for the `{% crispy %}` template tag:

```
submit = Submit('Search the Site', 'search this site')
```

Note: The first argument is also slugified and turned into the id for the submit button.

2.3 API templatetags

class `templatetags.crispy_forms_tags.BasicNode` (*form, helper, template_pack=None*)

Basic Node object that we can rely on for Node objects in normal template tags. I created this because most of the tags we'll be using will need both the form object and the helper string. This handles both the form object and parses out the helper string into attributes that templates can easily handle.

get_render (*context*)

Returns a *Context* object with all the necessary stuff for rendering the form

Parameters `context` – *django.template.Context* variable holding the context for the node

self.form and *self.helper* are resolved into real Python objects resolving them from the *context*. The *actual_form* can be a form or a formset. If it's a formset *is_formset* is set to True. If the helper has a layout we use it, for rendering the form or the formset's forms.

get_response_dict (*helper, context, is_formset*)

Returns a dictionary with all the parameters necessary to render the form/formset in a template.

Parameters

- `context` – *django.template.Context* for the node

- **is_formset** – Boolean value. If set to True, indicates we are working with a formset.

class `templatetags.crispy_forms_tags.CrispyFormNode` (*form*, *helper*, *template_pack=None*)

render (*context*)

Return the node rendered as a string.

class `templatetags.crispy_forms_tags.ForLoopSimulator` (*formset*)

Simulates a forloop tag, precisely:

```
{% for form in formset.forms %}
```

If `{% crispy %}` is rendering a formset with a helper, We inject a `ForLoopSimulator` object in the context as `forloop` so that formset forms can do things like:

```
Fieldset("Item {{ forloop.counter }}", [...])
HTML("{% if forloop.first %}First form text{% endif %}")
```

iterate ()

Updates values as if we had iterated over the for

`templatetags.crispy_forms_tags.do_uni_form` (*parser*, *token*)

You need to pass in at least the form/formset object, and can also pass in the optional `crispy_forms.helpers.FormHelper` object.

helper (optional): A `crispy_forms.helper.FormHelper` object.

Usage:

```
{% load crispy_tags %}
{% crispy form form.helper %}
```

You can also provide the template pack as the third argument:

```
{% crispy form form.helper 'bootstrap' %}
```

If the `FormHelper` attribute is named `helper` you can simply do:

```
{% crispy form %}
{% crispy form 'bootstrap' %}
```

`templatetags.crispy_forms_tags.whole_uni_form_template`

`templatetags.crispy_forms_tags.whole_uni_formset_template`

`templatetags.crispy_forms_filters.as_crispy_errors` (*form*, *template_pack=<SimpleLazyObject: 'bootstrap'>*)

Renders only form errors the same way as django-crispy-forms:

```
{% load crispy_forms_tags %}
{{ form|as_crispy_errors }}
```

or:

```
{{ form|as_crispy_errors:"bootstrap" }}
```

templatetags.crispy_forms_filters.**as_crispy_field** (*field*, *template_pack=<SimpleLazyObject: 'bootstrap'>*, *label_class=""*, *field_class=""*)

Renders a form field like a django-crispy-forms field:

```
{% load crispy_forms_tags %}
{{ form.field|as_crispy_field }}
```

or:

```
{{ form.field|as_crispy_field:"bootstrap" }}
```

templatetags.crispy_forms_filters.**as_crispy_form** (*form*, *template_pack=<SimpleLazyObject: 'bootstrap'>*, *label_class=""*, *field_class=""*)

The original and still very useful way to generate a div elegant form/formset:

```
{% load crispy_forms_tags %}

<form class="uniForm" method="post">
  {% csrf_token %}
  {{ myform|crispy }}
</form>
```

or, if you want to explicitly set the template pack:

```
{{ myform|crispy:"bootstrap" }}
```

In bootstrap3 or bootstrap4 for horizontal forms you can do:

```
{{ myform|label_class:"col-lg-2",field_class:"col-lg-8" }}
```

templatetags.crispy_forms_filters.**flatatt_filter** (*attrs*)

templatetags.crispy_forms_filters.**uni_form_template**

templatetags.crispy_forms_filters.**uni_formset_template**

class templatetags.crispy_forms_field.**CrispyFieldNode** (*field*, *attrs*)

render (*context*)

Return the node rendered as a string.

templatetags.crispy_forms_field.**classes** (*field*)

Returns CSS classes of a field

templatetags.crispy_forms_field.**crispy_addon** (*field*, *append=""*, *prepend=""*, *form_show_labels=True*)

Renders a form field using bootstrap's prepended or appended text:

```
{% crispy_addon form.my_field prepend="$" append=".00" %}
```

You can also just prepend or append like so

```
{% crispy_addon form.my_field prepend="$" %} {% crispy_addon form.my_field append=".00" %}
```

templatetags.crispy_forms_field.**crispy_field** (*parser*, *token*)

```
{% crispy_field field attrs %}
```

`templatetags.crispy_forms_field.css_class` (*field*)

Returns widgets class name in lowercase

`templatetags.crispy_forms_field.is_checkbox` (*field*)

`templatetags.crispy_forms_field.is_checkboxselectmultiple` (*field*)

`templatetags.crispy_forms_field.is_clearable_file` (*field*)

`templatetags.crispy_forms_field.is_file` (*field*)

`templatetags.crispy_forms_field.is_multivalue` (*field*)

`templatetags.crispy_forms_field.is_password` (*field*)

`templatetags.crispy_forms_field.is_radioselect` (*field*)

`templatetags.crispy_forms_field.is_select` (*field*)

`templatetags.crispy_forms_field.pairwise` (*iterable*)

s -> (s0,s1), (s2,s3), (s4, s5), ...

Think this is awesome and want to make it better? Read our contribution page, make it better, and you will be added to the [contributors](#) list!

3.1 Contributing

3.1.1 Setup

Fork on github

Before you do anything else, login/signup on Github.com and fork django-crispy-forms from <https://github.com/django-crispy-forms/django-crispy-forms>.

Clone your fork locally

If you have git-scm installed, you now clone your git repo using the following command-line argument where <my-github-name> is your account name on github:

```
git clone git@github.com:<my-github-name>/django-crispy-forms.git
```

3.1.2 Setting up topic branches and generating pull requests

While it's handy to provide useful code snippets in an issue, it is better for you as a developer to submit pull requests. By submitting pull request your contribution to django-crispy-forms will be recorded by Github.

In git it is best to isolate each topic or feature into a “topic branch”. While individual commits allow you control over how small individual changes are made to the code, branches are a great way to group a set of commits all related to one feature together, or to isolate different efforts when you might be working on multiple topics at the same time.

While it takes some experience to get the right feel about how to break up commits, a topic branch **must** be limited in scope to a single `issue` as submitted to an issue tracker.

Also since github pegs and syncs a pull request to a specific branch, it is the **ONLY** way that you can submit more than one fix at a time. If you submit a pull from your master branch, you can't make any more commits to your master without those getting added to the pull.

To create a topic branch, its easiest to use the convenient `-b` argument to `git checkout`:

```
git checkout -b fix-broken-thing
Switched to a new branch 'fix-broken-thing'
```

You should use a verbose enough name for your branch so it is clear what it is about. Now you can commit your changes and regularly merge in the upstream master as described below.

When you are ready to generate a pull request, either for preliminary review, or for consideration of merging into the project you must first push your local topic branch back up to github:

```
git push origin fix-broken-thing
```

Now when you go to your fork on github, you will see this branch listed under the “Source” tab where it says “Switch Branches”. Go ahead and select your topic branch from this list, and then click the “Pull request” button.

Here you can add a comment about your branch. If this in response to a submitted issue, it is good to put a link to that issue in this initial comment. The repo managers will be notified of your pull request and it will be reviewed (see below for best practices). Note that you can continue to add commits to your topic branch (and push them up to github) either if you see something that needs changing, or in response to a reviewer's comments. If a reviewer asks for changes, you do not need to close the pull and reissue it after making changes. Just make the changes locally, push them to github, then add a comment to the discussion section of the pull request.

3.1.3 Pull upstream changes into your fork regularly

django-crispy-forms is worked on by a lot of people. It is therefore critical that you pull upstream changes from trunk into your fork on a regular basis. Nothing is worse than putting in days of hard work into a pull request only to have it rejected because it has diverged too far from trunk.

To pull in upstream changes:

```
git remote add trunk git://github.com/django-crispy-forms/django-crispy-forms.git
git fetch trunk
```

Check the log to be sure that you actually want the changes, before merging:

```
git log ..django-crispy-forms/master
```

Then merge the changes that you fetched:

```
git merge django-crispy-forms/master
```

For more info, see <http://help.github.com/fork-a-repo/>

3.1.4 How to get your Pull Request accepted

We want your submission. But we also want to provide a stable experience for our users and the community. Follow these rules and you should succeed without a problem!

Run the tests!

Before you submit a pull request, please run the entire django-crispy-forms test suite via:

```
make test
```

If you don't have make installed the test suite can also be run via:

```
pytest --ds=crispy_forms.tests.test_settings --cov=crispy_forms
```

The first thing the core committers will do is run this command. Any pull request that fails this test suite will be **rejected**.

It's always good to add tests!

We've learned the hard way that code without tests is undependable. If your pull request comes with tests, it's got a greater chance to be included. Otherwise the lead will ask you to code them or will help you doing so.

We use the py.test.

Also, keep your tests as simple as possible. Complex tests end up requiring their own tests. We would rather see duplicated assertions across test methods than cunning utility methods that magically determine which assertions are needed at a particular stage. Remember: *Explicit is better than implicit*.

Don't mix code changes with whitespace cleanup

If you change two lines of code and correct 200 lines of whitespace issues in a file the diff on that pull request is functionally unreadable and will be **rejected**. Whitespace cleanups need to be in their own pull request.

Keep your pull requests limited to a single issue

django-crispy-forms pull requests should be as small/atomic as possible. Large, wide-sweeping changes in a pull request will be **rejected**, with comments to isolate the specific code in your pull request. Some examples:

1. If you are fixing a bug in one helper class don't '*cleanup*' unrelated helpers. That cleanup belongs in another pull request.
2. Changing permissions on a file should be in its own pull request with explicit reasons why.

Keep your code simple!

Memorize the Zen of Python:

```
>>> python -c 'import this'
```

Please keep your code as clean and straightforward as possible. When we see more than one or two functions/methods starting with *__my_special_function* or things like *__builtins__.object = str* we start to get worried. Rather than try and figure out your brilliant work we'll just **reject** it and send along a request for simplification.

Furthermore, the pixel shortage is over. We want to see:

- *helper* instead of *hpr*
- *django-crispy-forms* instead of *dcf*
- *my_function_that_does_things* instead of *mftdt*

h

`helper`, 37

l

`layout`, 38

t

`templatetags.crispy_forms_field`, 43

`templatetags.crispy_forms_filters`, 42

`templatetags.crispy_forms_tags`, 41

A

as_crispy_errors() (in module *template-tags.crispy_forms_filters*), 42
 as_crispy_field() (in module *template-tags.crispy_forms_filters*), 42
 as_crispy_form() (in module *template-tags.crispy_forms_filters*), 43

B

BaseInput (class in layout), 38
 BasicNode (class in *templatetags.crispy_forms_tags*), 41
 Button (class in layout), 39
 ButtonHolder (class in layout), 39

C

classes() (in module *template-tags.crispy_forms_field*), 43
 Column (class in layout), 39
 crispy_addon() (in module *template-tags.crispy_forms_field*), 43
 crispy_field() (in module *template-tags.crispy_forms_field*), 43
 CrispyFieldNode (class in *template-tags.crispy_forms_field*), 43
 CrispyFormNode (class in *templatetags.crispy_forms_tags*), 42
 css_class() (in module *template-tags.crispy_forms_field*), 44

D

Div (class in layout), 39
 do_uni_form() (in module *template-tags.crispy_forms_tags*), 42

F

Field (class in layout), 39
 Fieldset (class in layout), 39

flatatt_filter() (in module *template-tags.crispy_forms_filters*), 43
 ForLoopSimulator (class in *template-tags.crispy_forms_tags*), 42
 FormHelper (class in helper), 37

G

get_attributes() (helper.*FormHelper* method), 38
 get_render() (template-*tags.crispy_forms_tags.BasicNode* method), 41
 get_response_dict() (template-*tags.crispy_forms_tags.BasicNode* method), 41

H

helper (module), 37
 Hidden (class in layout), 40
 HTML (class in layout), 40

I

is_checkbox() (in module *template-tags.crispy_forms_field*), 44
 is_checkboxselectmultiple() (in module *templatetags.crispy_forms_field*), 44
 is_clearable_file() (in module *template-tags.crispy_forms_field*), 44
 is_file() (in module *template-tags.crispy_forms_field*), 44
 is_multivalue() (in module *template-tags.crispy_forms_field*), 44
 is_password() (in module *template-tags.crispy_forms_field*), 44
 is_radioselect() (in module *template-tags.crispy_forms_field*), 44
 is_select() (in module *template-tags.crispy_forms_field*), 44
 iterate() (template-*tags.crispy_forms_tags.ForLoopSimulator* method), 42

L

Layout (*class in layout*), 40
layout (*module*), 38

M

MultiField (*class in layout*), 40
MultiWidgetField (*class in layout*), 40

P

pairwise() (*in module template-tags.crispy_forms_field*), 44

R

render() (*layout.BaseInput method*), 38
render() (*templatetags.crispy_forms_field.CrispyFieldNode method*), 43
render() (*templatetags.crispy_forms_tags.CrispyFormNode method*), 42
render_layout() (*helper.FormHelper method*), 38
Reset (*class in layout*), 41
Row (*class in layout*), 41

S

Submit (*class in layout*), 41

T

templatetags.crispy_forms_field (*module*), 43
templatetags.crispy_forms_filters (*module*), 42
templatetags.crispy_forms_tags (*module*), 41

U

uni_form_template (*in module template-tags.crispy_forms_filters*), 43
uni_formset_template (*in module template-tags.crispy_forms_filters*), 43

W

whole_uni_form_template (*in module template-tags.crispy_forms_tags*), 42
whole_uni_formset_template (*in module templatetags.crispy_forms_tags*), 42