Wagtail Streamforms Documentation

Release 3.2

STuart George

Sep 06, 2018

Content

1 Backwards Compatibility 3

2 What else is included? 5

Allows you to build forms in the CMS admin area and add them to any StreamField in your site. You can add your own fields along with the vast array of default fields which include the likes of file fields. Form submissions are controlled by hooks that you can add that process the forms cleaned data. Templates can be created which will then appear as choices when you build your form, allowing you to display and submit a form however you want.

CHAPTER 1

Backwards Compatibility

Important: Please note that due to this package being virtually re-written for version 3, you cannot upgrade any existing older version of this package to version 3 and onwards. If you have an existing version installed less than 3 then you will need to completely remove it including tables and any migrations that were applied in the databases `django_migrations` table.

Older versions:

If you are using a version of wagtail 1.x, then the latest compatible version of this package is 1.6.3:

```
$ pip install wagtailstreamforms<2
```

Other wise you must install a version of this package from 2 onwards:

```
$ pip install wagtailstreamforms>=2
```

CHAPTER 2

What else is included?

- Each form is built using a StreamField.
- Customise things like success and error messages, post submit redirects and more.
- Forms are processed via a `before_page_serve` hook. Meaning there is no fuss like remembering to include a page mixin.
- The hook can easily be disabled to provide the ability to create your own.
- Form submissions are controlled via hooks meaning you can easily create things like emailing the submission which you can turn on and off on each form.
- Fields can easily be added to the form from your own code such as Recaptcha or a Regex Field.
- The default set of fields can easily be replaced to add things like widget attributes.
- Form submissions are also listed by their form which you can filter by date and are ordered by newest first.
- Files can also be submitted to the forms that are shown with the form submissions.
- A form and its fields can easily be copied to a new form.
- There is a template tag that can be used to render a form, in case you want it to appear outside a StreamField.

2.1 Installation

Wagtail Streamform is available on PyPI - to install it, just run:

```
pip install wagtailstreamforms
```

Once thats done you need to add the following to your `INSTALLED_APPS` settings:

```
INSTALLED_APPS = [
    ...
    'wagtail.contrib.modeladmin',
```

(continues on next page)

(continued from previous page)

```
    'wagtailstreamforms'
    ...
]
```

Run migrations:

```
python manage.py migrate
```

Go to your cms admin area and you will see the `Streamforms` section.

2.2 Basic Usage

Just add the `wagtailstreamforms.blocks.WagtailFormBlock()` in any of your streamfields:

```
body = StreamField([
    ...
    ('form', WagtailFormBlock())
    ...
])
```

And you are ready to go.

2.2.1 Using the template tag

There is also a template tag you can use outside of a streamfield, within a page. All this is doing is rendering the form using the same block as in the streamfield.

The tag takes three parameters:

- **slug** (`string`) - The slug of the form instance.
- **reference** (`string`) - This should be a unique string and needs to be persistent on refresh/reload. See note below.
- **action** (`string`) optional - The form action url.

Note: The reference is used when the form is being validated.

Because you can have any number of the same form on a page there needs to be a way of uniquely identifying the form beyond its `PK`. This is so that when the form has validation errors and it is passed back through the pages context, We know what form it is.

This reference **MUST** be persistent on page refresh or you will never see the errors.

Usage:

```
{% load streamforms_tags %}
{% streamforms_form slug="form-slug" reference="some-very-unique-reference" action="."
↪" %}
```

2.3 Templates

You can create your own form templates to use against any form in the system, providing a vast array of ways to create, style and submit your forms.

The default template located at `streamforms/form_block.html` can be seen below:

```
<h2>{{ value.form.title }}</h2>
<form{% if form.is_multipart %} enctype="multipart/form-data"{% endif %} action="{{
↪value.form_action }}" method="post" novalidate>
    {{ form.media }}
    {% csrf_token %}
    {% for hidden in form.hidden_fields %}{{ hidden }}{% endfor %}
    {% for field in form.visible_fields %}
        {% include 'streamforms/partials/form_field.html' %}
    {% endfor %}
    <input type="submit" value="{{ value.form.submit_button_text }}">
</form>
```

Note: It is important here to keep the hidden fields as the form will have some in order to process correctly.

Once you have created you own you will need to add it to the list of available templates within the form builder. This is as simple as adding it to the `WAGTAILSTREAMFORMS_FORM_TEMPLATES` in settings as below.

```
# this includes the default template in the package and an additional custom template.

WAGTAILSTREAMFORMS_FORM_TEMPLATES = (
    ('streamforms/form_block.html', 'Default Form Template'),  # default
    ('app/custom_form_template.html', 'Custom Form Template'),
)
```

You are not required to use the default template, its only there as a guideline to what is required and provide a fully working package out of the box. If you dont want it just remove it from the `WAGTAILSTREAMFORMS_FORM_TEMPLATES` setting.

2.3.1 Rendering your StreamField

It is important to ensure the request is in the context of your page to do this iterate over your StreamField block using wagtails `include_block` template tag.

```
{% load wagtailcore_tags %}

{% for block in page.body %}
    {% include_block block %}
{% endfor %}
```

DO NOT use the short form method of `{{ block }}` as described here as you will get CSRF verification failures.

2.3.2 Deleted forms

In the event of a form being deleted which is still in use in a streamfield the following template will be rendered in its place:

`streamforms/non_existent_form.html`

```
{% load i18n %}
<p>{% trans 'Sorry, this form has been deleted.' %}</p>
```

You can override this by putting a copy of the template in you own project using the same path under a templates directory ie `app/templates/streamforms/non_existent_form.html`. As long as the app is before `wagtailstreamforms` in `INSTALLED_APPS` it will use your template instead.

2.3.3 Messaging

When the `success` or `error` message options are completed in the form builder and upon submission of the form a message is sent to django's messaging framework.

You will need to add `django.contrib.messages` to your `INSTALLED_APPS` setting:

```
INSTALLED_APPS = [
    ...
    'django.contrib.messages'
    ...
]
```

To display these in your site you will need to include somewhere in your page's markup a snippet similar to the following:

```
{% if messages %}
<ul class="messages">
    {% for message in messages %}
    <li{% if message.tags %} class="{{ message.tags }}"{% endif %}>{{ message }}</li>
    {% endfor %}
</ul>
{% endif %}
```

Any message from the form will then be displayed.

2.4 Fields

Form fields are added to the form by the means of a StreamField. There are various default fields already defined as listed below:

- singleline
- multiline
- date
- datetime
- email
- url
- number
- dropdown
- multiselect
- radio

- checkboxes
- checkbox
- hidden
- singlefile
- multifile

The various default options for the fields are set when choosing that type of field within the StreamField. For example a dropdown includes options to set the `choices` and an additional `empty_label` as the first choice.

2.4.1 Adding new fields

You can also register your own fields which will be added to the form builders StreamField. First you need to create the file `wagtailstreamforms_fields.py` in the root of an app in your project and add the following as an example:

```
from django import forms
from wagtailstreamforms.fields import BaseField, register

@register('mytext')
class CustomTextField(BaseField):
    field_class = forms.CharField
```

This will add a simple single line charfield to the list of available fields with the type `mytext`.

The `BaseField` class also has some additional properties you can set as follows:

```
@register('mytextarea')
class CustomTextAreaField(BaseField):
    # the form field class
    field_class = forms.CharField
    # the widget for the form field
    widget = forms.widgets.Textarea
    # the icon in the streamfield
    icon = 'placeholder'
    # the label to show in the streamfield
    label = 'My text area'
```

2.4.2 Setting widget attributes

Setting widget attributes can be done on the `BaseField` class as follows:

```
@register('mytextarea')
class CustomTextAreaField(BaseField):
    field_class = forms.CharField
    widget = forms.widgets.Textarea(attrs={'rows': 10})
```

2.4.3 Setting field options

The `BaseField` class has a default dict of options set from the StreamField's StructValue:

```
class BaseField:
    def get_options(self, block_value):
        return {
            'label': block_value.get('label'),
            'help_text': block_value.get('help_text'),
            'required': block_value.get('required'),
            'initial': block_value.get('default_value')
        }
```

You can use this to provide additional options set either by passing them from the StreamField or manually setting them. The below adds django's slug validator to create a slug field:

```
from django.core import validators

@register('slug')
class SlugField(BaseField):
    field_class = forms.CharField

    def get_options(self, block_value):
        options = super().get_options(block_value)
        options.update({'validators': [validators.validate_slug]})
        return options
```

2.4.4 Editable field options

To be able to make the field options editable from within the StreamField you must override the `BaseField.get_form_block()` method with the additonal options you will require.

Consider that you need a max length on a `CharField` but want the length to be configurable on every instance of that field. Firstly you need to setup the field's StructBlock so that the additional options are available within the StreamField:

```
@register('maxlength')
class MaxLengthField(BaseField):
    field_class = forms.CharField
    label = 'Text field (max length)'

    def get_form_block(self):
        return blocks.StructBlock([
            ('label', blocks.CharBlock()),
            ('help_text', blocks.CharBlock(required=False)),
            ('required', blocks.BooleanBlock(required=False)),
            ('max_length', blocks.IntegerBlock(required=True)),
            ('default_value', blocks.CharBlock(required=False)),
        ], icon=self.icon, label=self.label)
```

and then pull that value into the fields options:

```
@register('maxlength')
class MaxLengthField(BaseField):
    field_class = forms.CharField
    label = 'Text field (max length)'

    def get_options(self, block_value):
        options = super().get_options(block_value)
        options.update({'max_length': block_value.get('max_length')})
```

(continues on next page)

(continued from previous page)

```
        return options

    def get_form_block(self):
        return blocks.StructBlock([
            ('label', blocks.CharBlock()),
            ('help_text', blocks.CharBlock(required=False)),
            ('required', blocks.BooleanBlock(required=False)),
            ('max_length', blocks.IntegerBlock(required=True)),
            ('default_value', blocks.CharBlock(required=False)),
        ], icon=self.icon, label=self.label)
```

2.4.5 Overriding an existing field

Important: When overriding an existing field make sure the app that has the `wagtailstreamforms_fields.py` file appears after `wagtailstreamforms` in your `INSTALLED_APPS` or the field will not be overridden.

You can replace one of the form fields by simply using an existing name in the `@register` decorator. Suppose you want to add a rows attribute to the textarea widget of the `multiline` field.

In your `wagtailstreamforms_fields.py` file:

```
@register('multiline')
class MultiLineTextField(BaseField):
    field_class = forms.CharField
    widget = forms.widgets.Textarea(attrs={'rows': 10})
```

2.4.6 Using file fields

To handle file fields correctly you must ensure the your form template has the correct enctype. you can automatically add this with a simple if statement to detect if the form is a multipart type form.

```
<form{% if form.is_multipart %} enctype="multipart/form-data"{% endif %} action="...
```

Files will be uploaded using your default storage class to the path `streamforms/` and are listed along with the form submissions. When a submission is deleted all files are also deleted from the storage.

2.4.7 Examples

Below are some examples of useful fields.

Model choice

An example model choice field of users.

```
from django import forms
from django.contrib.auth.models import User

from wagtail.core import blocks
from wagtailstreamforms.fields import BaseField, register
```

(continues on next page)

(continued from previous page)

```
@register('user')
class UserChoiceField(BaseField):
    field_class = forms.ModelChoiceField
    icon = 'arrow-down-big'
    label = 'User dropdown field'

    @staticmethod
    def get_queryset():
        return User.objects.all()

    def get_options(self, block_value):
        options = super().get_options(block_value)
        options.update({'queryset': self.get_queryset()})
        return options

    def get_form_block(self):
        return blocks.StructBlock([
            ('label', blocks.CharBlock()),
            ('help_text', blocks.CharBlock(required=False)),
            ('required', blocks.BooleanBlock(required=False)),
        ], icon=self.icon, label=self.label)
```

Regex validated

An example field that allows a selection of regex patterns with an option to set the invalid error message.

Taking this further you could provide the invalid error messages from code if they were always the same for any given regex pattern.

```
from django import forms

from wagtail.core import blocks
from wagtailstreamforms.fields import BaseField, register

@register('regex_validated')
class RegexValidatedField(BaseField):
    field_class = forms.RegexField
    label = 'Regex field'

    def get_options(self, block_value):
        options = super().get_options(block_value)
        options.update({
            'regex': block_value.get('regex'),
            'error_messages': {'invalid': block_value.get('error_message')}
        })
        return options

    def get_regex_choices(self):
        return (
            ('(.*?)', 'Any'),
            ('^[a-zA-Z0-9]+$', 'Letters and numbers only'),
        )
```

(continues on next page)

(continued from previous page)

```
    def get_form_block(self):
        return blocks.StructBlock([
            ('label', blocks.CharBlock()),
            ('help_text', blocks.CharBlock(required=False)),
            ('required', blocks.BooleanBlock(required=False)),
            ('regex', blocks.ChoiceBlock(choices=self.get_regex_choices())),
            ('error_message', blocks.CharBlock()),
            ('default_value', blocks.CharBlock(required=False)),
        ], icon=self.icon, label=self.label)
```

ReCAPTCHA

Adding a ReCAPTCHA field is as simple as follows.

Installing `django-recaptcha`:

```
pip install django-recaptcha
```

Django `settings.py` file:

```
INSTALLED_APPS = [
    ...
    'captcha'
    ...
]

# developer keys
RECAPTCHA_PUBLIC_KEY = '6LeIxAcTAAAAAJcZVRqyHh71UMIEGNQ_MXjiZKhI'
RECAPTCHA_PRIVATE_KEY = '6LeIxAcTAAAAAGG-vFI1TnRWxMZNFuojJ4WifJWe'
# enable no captcha
NOCAPTCHA = True
```

`wagtailstreamforms_fields.py` file:

```
from captcha.fields import ReCaptchaField
from wagtail.core import blocks
from wagtailstreamforms.fields import BaseField, register

@register('recaptcha')
class ReCaptchaField(BaseField):
    field_class = ReCaptchaField
    icon = 'success'
    label = 'ReCAPTCHA field'

    def get_options(self, block_value):
         options = super().get_options(block_value)
         options.update({
             'required': True
         })
         return options

    def get_form_block(self):
        return blocks.StructBlock([
            ('label', blocks.CharBlock()),
            ('help_text', blocks.CharBlock(required=False)),
        ], icon=self.icon, label=self.label)
```

2.4.8 Useful Resources

- Django Documentation - Form fields

2.4.9 Reference

class wagtailstreamforms.fields.**BaseField**
A base form field class, all form fields must inherit this class.

Usage:

```
@register('multiline')
class MultiLineTextField(BaseField):
    field_class = forms.CharField
    widget = forms.widgets.Textarea
    icon = 'placeholder'
    label = 'Text (multi line)'
```

get_form_block()
The StreamField StructBlock.

Override this to provide additional fields in the StreamField.

Returns The `wagtail.core.blocks.StructBlock` to be used in the StreamField

get_formfield(*block_value*)
Get the form field. Its unlikely you will need to override this.

Parameters **block_value** – The StreamValue for this field from the StreamField

Returns An instance of a form field class, ie `django.forms.CharField(**options)`

get_options(*block_value*)
The field options.

Override this to provide additional options such as `choices` for a dropdown.

Parameters **block_value** – The StreamValue for this field from the StreamField

Returns The options to be passed into the field, ie `django.forms.CharField(**options)`

2.5 Advanced Settings

Some times there is a requirement to save additional data for each form. Such as details of where to email the form submission. When this is needed we have provided the means to define your own model.

To enable this you need to declare a model that inherits from `wagtailstreamforms.models.AbstractFormSetting`:

```
from wagtailstreamforms.models.abstract import AbstractFormSetting

class AdvancedFormSetting(AbstractFormSetting):
    to_address = models.EmailField()
```

Once that's done you need to add a setting to point to that model:

```
# the model defined to save advanced form settings
# in the format of 'app_label.model_class'.
# Model must inherit from 'wagtailstreamforms.models.AbstractFormSetting'.
WAGTAILSTREAMFORMS_ADVANCED_SETTINGS_MODEL = 'myapp.AdvancedFormSetting'
```

A button will appear on the Streamforms listing view `Advanced` which will allow you to edit that model.

2.5.1 Usage

The data saved can be used in *Submission Hooks* on the `instance.advanced_settings` property.

```
@register('process_form_submission')
def email_submission(instance, form):
    send_mail(
        ..
        recipient_list=[instance.advanced_settings.to_address]
    )
```

2.6 Submission Methods

Form submissions are handled by the means of a wagtail `before_serve_page` hook. The built in hook at `wagtailstreamforms.wagtail_hooks.process_form` looks for a form in the post request, and either:

- processes it redirecting back to the current page or defined page in the form setup.
- or renders the current page with any validation error.

If no form was posted then the page serves in the usual manner.

Note: Currently the hook expects the form to be posting to the same page it exists on.

2.6.1 Providing your own submission method

If you do not want the current hook to be used you need to disable it by setting the `WAGTAILSTREAMFORMS_ENABLE_FORM_PROCESSING` to `False` in your settings:

```
WAGTAILSTREAMFORMS_ENABLE_FORM_PROCESSING = False
```

With this set no forms will be processed of any kind and you are free to process them how you feel fit.

A basic hook example

```
@hooks.register('before_serve_page')
def process_form(page, request, *args, **kwargs):
    """ Process the form if there is one, if not just continue. """

    # only process if settings.WAGTAILSTREAMFORMS_ENABLE_FORM_PROCESSING is True
    if not get_setting('ENABLE_FORM_PROCESSING'):
        return
```

(continues on next page)

(continued from previous page)

```
    if request.method == 'POST':
        form_def = get_form_instance_from_request(request)

        if form_def:
            form = form_def.get_form(request.POST, request.FILES, page=page,␣
↪user=request.user)
            context = page.get_context(request, *args, **kwargs)

            if form.is_valid():
                # process the form submission
                form_def.process_form_submission(form)

                # create success message
                if form_def.success_message:
                    messages.success(request, form_def.success_message, fail_
↪silently=True)

                # redirect to the page defined in the form
                # or the current page as a fallback - this will avoid refreshing and␣
↪submitting again
                redirect_page = form_def.post_redirect_page or page

                return redirect(redirect_page.get_url(request), context=context)

            else:
                # update the context with the invalid form and serve the page
                context.update({
                    'invalid_stream_form_reference': form.data.get('form_reference'),
                    'invalid_stream_form': form
                })

                # create error message
                if form_def.error_message:
                    messages.error(request, form_def.error_message, fail_
↪silently=True)

                return TemplateResponse(
                    request,
                    page.get_template(request, *args, **kwargs),
                    context
                )
```

Supporting ajax requests

The only addition here from the basic example is just the `if request.is_ajax:` and the `JsonResponse` parts.

We are just making it respond with this if the request was ajax.

```
@hooks.register('before_serve_page')
def process_form(page, request, *args, **kwargs):
    """ Process the form if there is one, if not just continue. """

    if request.method == 'POST':
        form_def = get_form_instance_from_request(request)
```

(continues on next page)

(continued from previous page)

```
        if form_def:
            form = form_def.get_form(request.POST, request.FILES, page=page,␣
↪user=request.user)
            context = page.get_context(request, *args, **kwargs)

            if form.is_valid():
                # process the form submission
                form_def.process_form_submission(form)

                # if the request is_ajax then just return a success message
                if request.is_ajax():
                    return JsonResponse({'message': form_def.success_message or
↪'success'})

                # insert code to serve page if not ajax (as original)

            else:
                # if the request is_ajax then return an error message and the form␣
↪errors
                if request.is_ajax():
                    return JsonResponse({
                        'message': form_def.error_message or 'error',
                        'errors': form.errors
                    })

                # insert code to serve page if not ajax (as original)
```

Add some javascript somewhere to process the form via ajax:

```
<form>...</form>
<script>
    $("#id_streamforms_{{ form.initial.form_id }}").submit(function(e) {
        $.ajax({
            type: "POST",
            url: ".",
            data: $(this).serialize(),
            success: function(data) {
                // do something with data
                console.log(data);
            },
            error: function(data) {
                // do something with data
                console.log(data);
            }
        });
        e.preventDefault();
    });
</script>
```

2.7 Submission Hooks

Form submission hooks are used to process the cleaned_data of the form after a successful post. The only defined one is that to save the form submission data.

```
@register('process_form_submission')
def save_form_submission_data(instance, form):
    """ saves the form submission data """

    # copy the cleaned_data so we dont mess with the original
    submission_data = deepcopy(form.cleaned_data)

    # change the submission data to a count of the files
    for field in form.files.keys():
        count = len(form.files.getlist(field))
        submission_data[field] = '{} file{}'.format(count, pluralize(count))

    # save the submission data
    submission = instance.get_submission_class().objects.create(
        form_data=json.dumps(submission_data, cls=FormSubmissionSerializer),
        form=instance
    )

    # save the form files
    for field in form.files:
        for file in form.files.getlist(field):
            FormSubmissionFile.objects.create(
                submission=submission,
                field=field,
                file=file
            )
```

2.7.1 Create your own hook

You can easily define additional hooks to perform a vast array of actions like

- send a mail
- save the data to a db
- reply to the sender
- etc

Here is a simple example to send an email with the submission data.

Create a `wagtailstreamforms_hooks.py` in the root of one of your apps and add the following.

```
from django.conf import settings
from django.core.files.uploadedfile import InMemoryUploadedFile
from django.core.mail import EmailMessage
from django.template.defaultfilters import pluralize

from wagtailstreamforms.hooks import register

@register('process_form_submission')
def email_submission(instance, form):
    """ Send an email with the submission. """

    addresses = ['to@example.com']
    content = ['Please see below submission\n', ]
    from_address = settings.DEFAULT_FROM_EMAIL
    subject = 'New Form Submission : %s' % instance.title
```

(continues on next page)

(continued from previous page)

```
    # build up the email content
    for field, value in form.cleaned_data.items():
        if isinstance(value, InMemoryUploadedFile):
            count = len(form.files.getlist(field))
            value = '{} file{}'.format(count, pluralize(count))
        elif isinstance(value, list):
            value = ', '.join(value)
        content.append('{}: {}'.format(field, value))
    content = '\n'.join(content)

    # create the email message
    email = EmailMessage(
        subject=subject,
        body=content,
        from_email=from_address,
        to=addresses
    )

    # attach any files submitted
    for field in form.files:
        for file in form.files.getlist(field):
            email.attach(file.name, file.file.getvalue(), file.content_type)

    # finally send the email
    email.send(fail_silently=True)
```

A new option will appear in the setup of the forms to run the above hook. The name of the option is taken from the function name so keep them unique to avoid confusion. The `instance` is the form class instance, the `form` is the processed valid form in the request.

2.8 Permissions

Setting the level of access to administer your forms is the same as it is for any page. The permissions will appear in the groups section of the wagtail admin > settings area.

Here you can assign the usual add, change and delete permissions.

Note: Its worth noting here that if you do delete a form it will also delete all submissions for that form.

2.8.1 Form submission permissions

Because the form submission model is not listed in the admin area the following statement applies.

Important: If you can either add, change or delete a form then you can view all of its submissions. However to be able to delete the submissions, it requires that you can delete the form.

2.9 Housekeeping

2.9.1 Removing old form submissions

There is a management command you can use to remove submissions that are older than the supplied number of days to keep.

```
python manage.py prunesubmissions 30
```

Where `30` is the number of days to keep before today. Passing `0` will keep today's submissions only.

Or to run the command from code:

```
from django.core.management import call_command

call_command('prunesubmissions', 30)
```

2.10 Settings

Any settings with their defaults are listed below for quick reference.

```
# the label of the forms area in the admin sidebar
WAGTAILSTREAMFORMS_ADMIN_MENU_LABEL = 'Streamforms'

# the order of the forms area in the admin sidebar
WAGTAILSTREAMFORMS_ADMIN_MENU_ORDER = None

# the model defined to save advanced form settings
# in the format of 'app_label.model_class'.
# Model must inherit from 'wagtailstreamforms.models.AbstractFormSetting'.
WAGTAILSTREAMFORMS_ADVANCED_SETTINGS_MODEL = None

# enable the built in hook to process form submissions
WAGTAILSTREAMFORMS_ENABLE_FORM_PROCESSING = True

# the default form template choices
WAGTAILSTREAMFORMS_FORM_TEMPLATES = (
    ('streamforms/form_block.html', 'Default Form Template'),
)
```

2.11 Contributors

People that have helped in any way shape or form to get to where we are, many thanks.

2.11.1 In our team

- Dave Fuller
- Stuart George
- Tim Donhou

2.11.2 In the community

- José Luis
- Nathan Victor
- Tom Dyson

2.12 Changelog

2.12.1 3.2.0

- fix template that inherited from wagtailforms to wagtailadmin

2.12.2 3.1.0

Support for Wagtail 2.2

2.12.3 3.0.0

Version 3 is a major re-write and direction change and therefor any version prior to this needs to be removed in its entirety first.

Whats New:

- Update to Wagtail 2.1
- The concept of creating a custom form class to add functionality has been removed.
- Along with the concept of custom form submission classes.
- Fields are now added via a StreamField and you can define your own like ReCAPTCHA or RegexFields.
- You can easily overwrite fields to add things like widget attributes.
- You can define a model that will allow you to save additional settings for each form.
- The form submission is processed via hooks instead of baked into the models.
- You can create as many form submission hooks as you like to process, email etc the data as you wish. These will be available to all forms that you can enable/disable at will.
- Files can now be uploaded and are stored along with the submission using the default storage.
- There is a management command to easily remove old submission data.

2.12.4 2.0.1

- Fixed migration issue #70

2.12.5 2.0.0

- Added support for wagtail 2.

2.12.6 1.6.3

- Fix issue where js was not in final package

2.12.7 1.6.2

- Added javascript to auto populate the form slug from the name

2.12.8 1.6.1

- Small tidy up in form code

2.12.9 1.6.0

- Stable Release

2.12.10 1.5.2

- Added `AbstractEmailForm` to more easily allow creating additional form types.

2.12.11 1.5.1

- Fix migrations being regenerated when template choices change

2.12.12 1.5.0

- Removed all project dependencies except wagtail and recapcha
- The urls no longer need to be specified in your `urls.py` and can be removed.

2.12.13 1.4.4

- The template tag now has the full page context incase u need a reference to the user or page

2.12.14 1.4.3

- Fixed bug where messages are not available in the template tags context

2.12.15 1.4.2

- Removed label value from recapcha field
- Added setting to set order of menu item in cms admin

2.12.16 1.4.1

- Added an optional error message to display if the forms have errors

2.12.17 1.4.0

- Added a template tag that can be used to render a form. Incase you want it to appear outside a streamfield

2.12.18 1.3.0

- A form and it's fields can easily be copied to a new form from within the admin area

2.12.19 1.2.3

- Fix paginator on submission list not remembering date filters

2.12.20 1.2.2

- Form submission viewing and deleting permissions have been implemented

2.12.21 1.2.1

- On the event that a form is deleted that is still referenced in a streamfield, we are rendering a generic template that can be overridden to warn the end user

2.12.22 1.2.0

- In the form builder you can now specify a page to redirect to upon successful submission of the form
- The page mixin StreamFormPageMixin that needed to be included in every page has now been replaced by a wagtail before_serve_page hook so you will need to remove this mixin

2.12.23 1.1.1

- Fixed bug where multiple forms of same type in a streamfield were both showing validation errors when one submitted

2.13 Screenshots

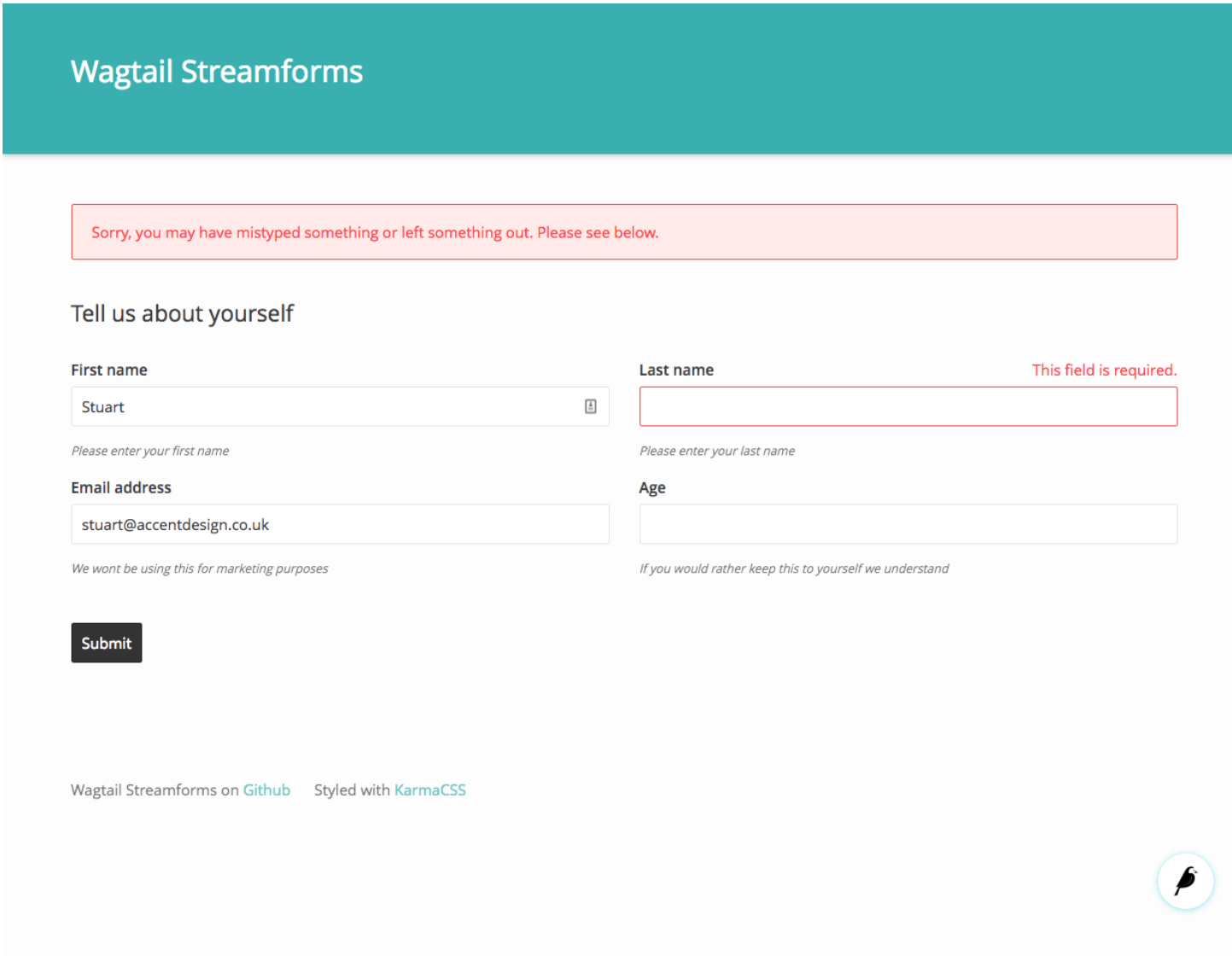


Fig. 1: Example Front End

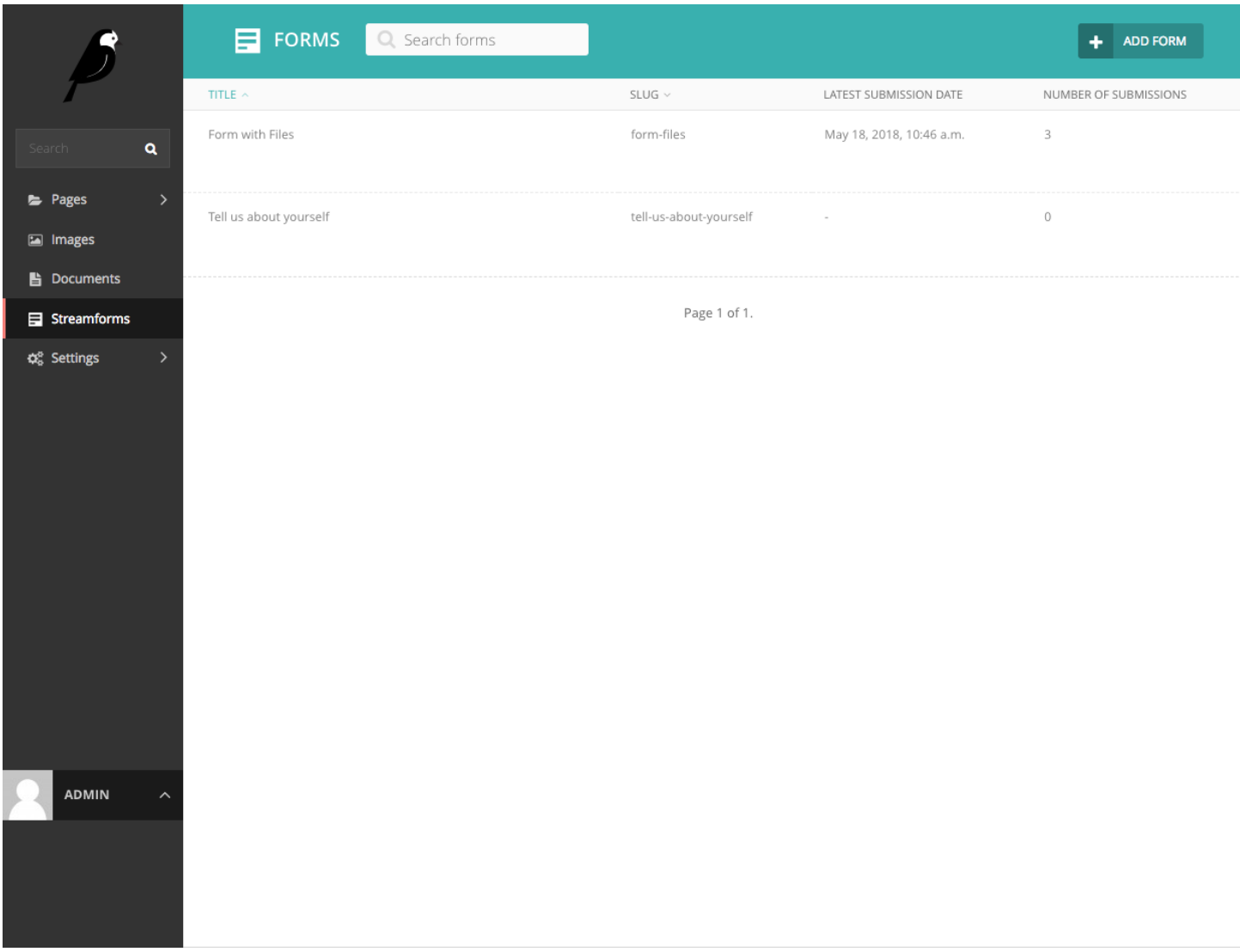


Fig. 2: Form Listing

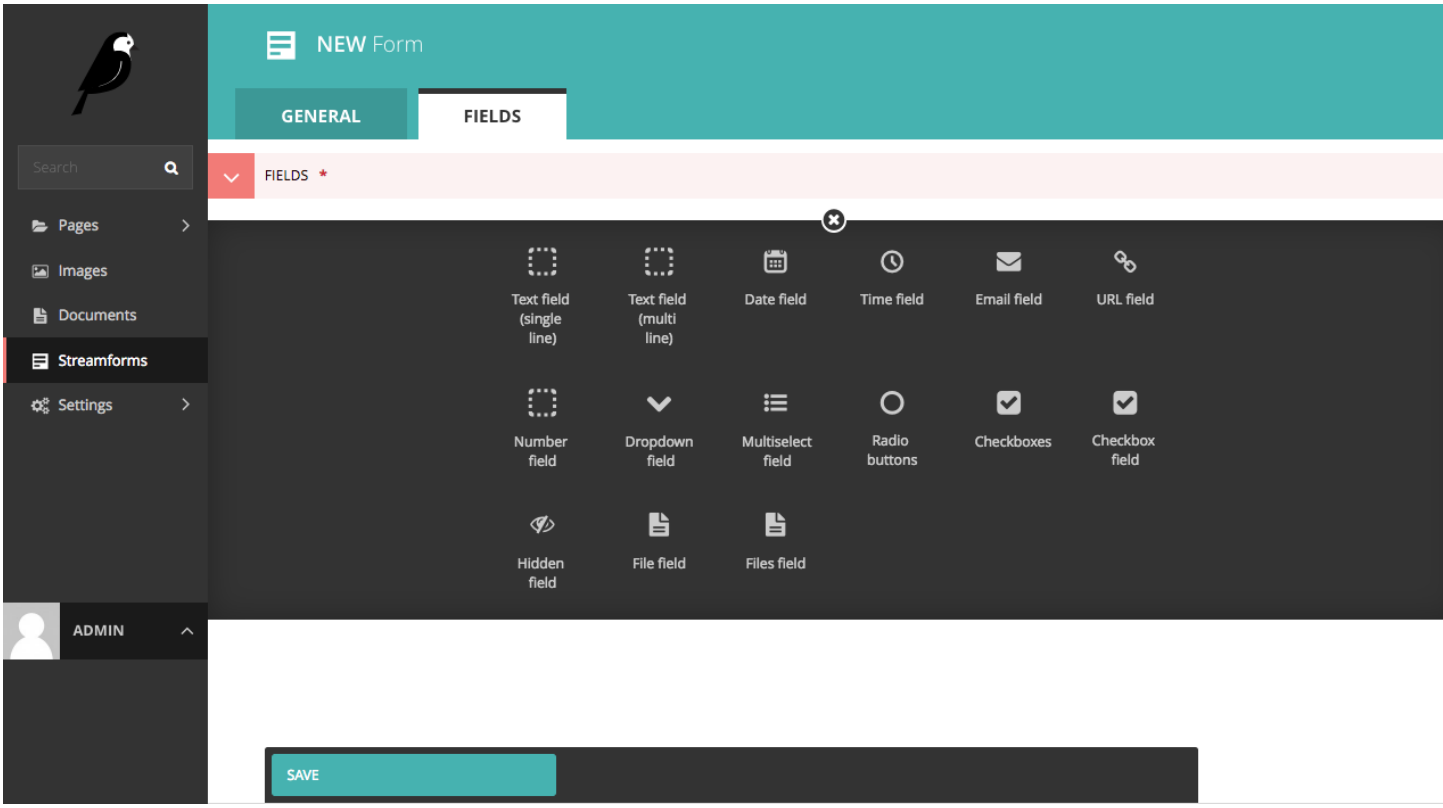


Fig. 3: Form Fields Selection

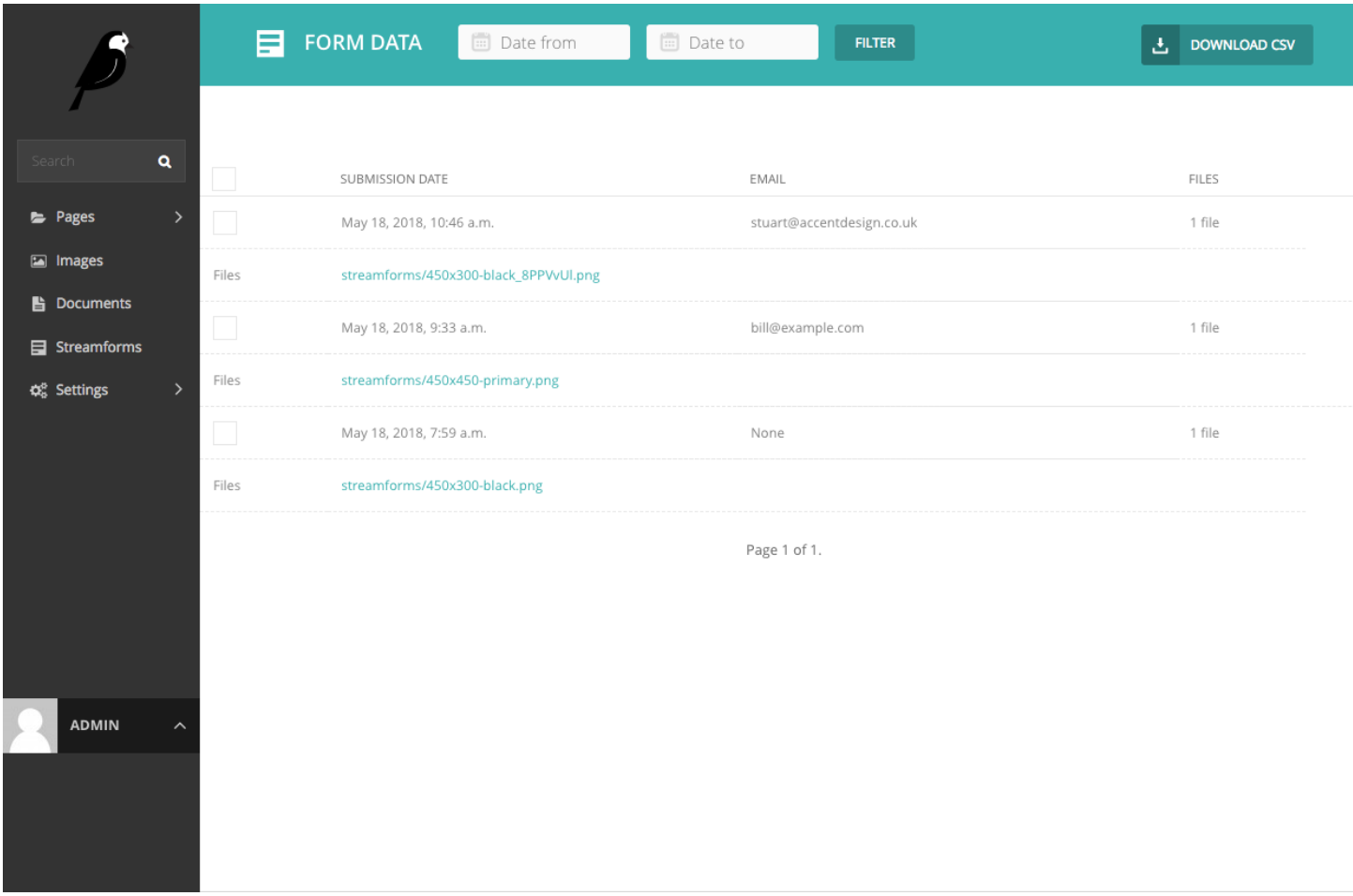


Fig. 4: Submission Listing

Index

B

BaseField (class in wagtailstreamforms.fields), 14

G

get_form_block() (wagtailstreamforms.fields.BaseField method), 14
get_formfield() (wagtailstreamforms.fields.BaseField method), 14
get_options() (wagtailstreamforms.fields.BaseField method), 14