# Flask-Admin documentation

*Release 1.5.0*

**Serge S. Koval**

**Jun 21, 2017**

# Contents


**1 Introduction To Flask-Admin** **3**
- 1.1 Getting Started . . . 3
- 1.2 Authorization & Permissions . . . 4
- 1.3 Customizing Built-in Views . . . 5
- 1.4 Adding Your Own Views . . . 8
- 1.5 Working With the Built-in Templates . . . 9

**2 Advanced Functionality** **13**
- 2.1 Enabling CSRF Protection . . . 13
- 2.2 Localization With Flask-Babelex . . . 13
- 2.3 Managing Files & Folders . . . 14
- 2.4 Adding A Redis Console . . . 15
- 2.5 Replacing Individual Form Fields . . . 15
- 2.6 Managing Geographical Models . . . 16
- 2.7 Customising Builtin Forms Via Rendering Rules . . . 17
- 2.8 Using Different Database Backends . . . 18
- 2.9 Migrating From Django . . . 20
- 2.10 Overriding the Form Scaffolding . . . 21
- 2.11 Customizing Batch Actions . . . 21

**3 Adding A Model Backend** **23**
- 3.1 Extending BaseModelView . . . 23
- 3.2 Implementing filters . . . 25

**4 API** **27**
- 4.1 `flask_admin.base` . . . 27
- 4.2 `flask_admin.helpers` . . . 30
- 4.3 `flask_admin.model` . . . 31
- 4.4 `flask_admin.form` . . . 45
- 4.5 `flask_admin.form.rules` . . . 45
- 4.6 `flask_admin.form.fields` . . . 47
- 4.7 `flask_admin.form.upload` . . . 47
- 4.8 `flask_admin.tools` . . . 49
- 4.9 `flask_admin.actions` . . . 50
- 4.10 `flask_admin.contrib.sqla` . . . 51
- 4.11 `flask_admin.contrib.mongoengine` . . . 61
- 4.12 `flask_admin.contrib.mongoengine.fields` . . . 71

4.13 flask_admin.contrib.peewee . . . . . 71
4.14 flask_admin.contrib.pymongo . . . . . 79
4.15 flask_admin.contrib.fileadmin . . . . . 86
4.16 flask_admin.model.template . . . . . 87

**5 Changelog** **89**
5.1 1.5.0 . . . . . 89
5.2 1.4.2 . . . . . 89
5.3 1.4.1 . . . . . 90
5.4 1.4.0 . . . . . 90
5.5 1.3.0 . . . . . 90

**6 Support** **91**

**7 Indices And Tables** **93**

**Python Module Index** **95**

**Why Flask?** As a micro-framework, Flask lets you build web services with very little overhead. It offers freedom for you, the designer, to implement your project in a way that suits your particular application.

**Why Flask-Admin?** In a world of micro-services and APIs, Flask-Admin solves the boring problem of building an admin interface on top of an existing data model. With little effort, it lets you manage your web service's data through a user-friendly interface.

**How does it work?** The basic concept behind Flask-Admin, is that it lets you build complicated interfaces by grouping individual views together in classes: Each web page you see on the frontend, represents a method on a class that has explicitly been added to the interface.

These view classes are especially helpful when they are tied to particular database models, because they let you group together all of the usual *Create, Read, Update, Delete* (CRUD) view logic into a single, self-contained class for each of your models.

**What does it look like?** At http://examples.flask-admin.org/ you can see some examples of Flask-Admin in action, or browse through the *examples/* directory in the GitHub repository.

CHAPTER 1

# Introduction To Flask-Admin

## Getting Started

### Initialization

The first step is to initialize an empty admin interface for your Flask app:

```
from flask import Flask
from flask_admin import Admin

app = Flask(__name__)

admin = Admin(app, name='microblog', template_mode='bootstrap3')
# Add administrative views here

app.run()
```

Here, both the *name* and *template_mode* parameters are optional. Alternatively, you could use the *init_app()* method.

If you start this application and navigate to http://localhost:5000/admin/, you should see an empty page with a navigation bar on top.

### Adding Model Views

Model views allow you to add a dedicated set of admin pages for managing any model in your database. Do this by creating instances of the *ModelView* class, which you can import from one of Flask-Admin's built-in ORM backends. An example is the SQLAlchemy backend, which you can use as follows:

```
from flask_admin.contrib.sqla import ModelView

# Flask and Flask-SQLAlchemy initialization here

admin = Admin(app, name='microblog', template_mode='bootstrap3')
admin.add_view(ModelView(User, db.session))
admin.add_view(ModelView(Post, db.session))
```

Straight out of the box, this gives you a set of fully featured *CRUD* views for your model:

- A *list* view, with support for searching, sorting, filtering, and deleting records.
- A *create* view for adding new records.
- An *edit* view for updating existing records.
- An optional, read-only *details* view.

There are many options available for customizing the display and functionality of these built-in views. For more details on that, see *Customizing Built-in Views*. For more details on the other ORM backends that are available, see *Using Different Database Backends*.

### Adding Content to the Index Page

The first thing you'll notice when you visit http://localhost:5000/admin/ is that it's just an empty page with a navigation menu. To add some content to this page, save the following text as *admin/index.html* in your project's *templates* directory:

```
{% extends 'admin/master.html' %}

{% block body %}
  <p>Hello world</p>
{% endblock %}
```

This will override the default index template, but still give you the built-in navigation menu. So, now you can add any content to the index page, while maintaining a consistent user experience.

## Authorization & Permissions

---

When setting up an admin interface for your application, one of the first problems you'll want to solve is how to keep unwanted users out. With Flask-Admin there are a few different ways of approaching this.

### HTTP Basic Auth

The simplest form of authentication is HTTP Basic Auth. It doesn't interfere with your database models, and it doesn't require you to write any new view logic or template code. So it's great for when you're deploying something that's still under development, before you want the whole world to see it.

Have a look at Flask-BasicAuth to see just how easy it is to put your whole application behind HTTP Basic Auth.

Unfortunately, there is no easy way of applying HTTP Basic Auth just to your admin interface.

### Rolling Your Own

For a more flexible solution, Flask-Admin lets you define access control rules on each of your admin view classes by simply overriding the *is_accessible* method. How you implement the logic is up to you, but if you were to use a low-level library like Flask-Login, then restricting access could be as simple as:

```
class MicroBlogModelView(sqla.ModelView):

    def is_accessible(self):
        return login.current_user.is_authenticated

    def inaccessible_callback(self, name, **kwargs):
        # redirect to login page if user doesn't have access
        return redirect(url_for('login', next=request.url))
```

In the navigation menu, components that are not accessible to a particular user will not be displayed for that user. For an example of using Flask-Login with Flask-Admin, have a look at https://github.com/flask-admin/Flask-Admin/tree/master/examples/auth-flask-login.

The main drawback is that you still need to implement all of the relevant login, registration, and account management views yourself.

### Using Flask-Security

If you want a more polished solution, you could use Flask-Security, which is a higher-level library. It comes with lots of built-in views for doing common things like user registration, login, email address confirmation, password resets, etc.

The only complicated bit is making the built-in Flask-Security views integrate smoothly with the Flask-Admin templates to create a consistent user experience. To do this, you will need to override the built-in Flask-Security templates and have them extend the Flask-Admin base template by adding the following to the top of each file:

```
{% extends 'admin/master.html' %}
```

Now, you'll need to manually pass in some context variables for the Flask-Admin templates to render correctly when they're being called from the Flask-Security views. Defining a *security_context_processor* function will take care of this for you:

```
def security_context_processor():
    return dict(
        admin_base_template=admin.base_template,
        admin_view=admin.index_view,
        h=admin_helpers,
    )
```

For a working example of using Flask-Security with Flask-Admin, have a look at https://github.com/flask-admin/Flask-Admin/tree/master/examples/auth.

The example only uses the built-in *register* and *login* views, but you could follow the same approach for including the other views, like *forgot_password*, *send_confirmation*, etc.

## Customizing Built-in Views

The built-in *ModelView* class is great for getting started quickly. But, you'll want to configure its functionality to suit your particular models. This is done by setting values for the configuration attributes that are made available in the *ModelView* class.

To specify some global configuration parameters, you can subclass *ModelView* and use that subclass when adding your models to the interface:

```
from flask_admin.contrib.sqla import ModelView

# Flask and Flask-SQLAlchemy initialization here

class MicroBlogModelView(ModelView):
    can_delete = False  # disable model deletion
    page_size = 50  # the number of entries to display on the list view

admin.add_view(MicroBlogModelView(User, db.session))
admin.add_view(MicroBlogModelView(Post, db.session))
```

Or, in much the same way, you can specify options for a single model at a time:

```
class UserView(ModelView):
        can_delete = False  # disable model deletion

class PostView(ModelView):
        page_size = 50  # the number of entries to display on the list view

admin.add_view(UserView(User, db.session))
admin.add_view(PostView(Post, db.session))
```

## *ModelView* Configuration Attributes

For a complete list of the attributes that are defined, have a look at the API documentation for *BaseModelView()*. Here are some of the most commonly used attributes:

To **disable some of the CRUD operations**, set any of these boolean parameters:

```
can_create = False
can_edit = False
can_delete = False
```

If your model has too much data to display in the list view, you can **add a read-only details view** by setting:

```
can_view_details = True
```

**Removing columns** from the list view is easy, just pass a list of column names for the *column_excludes_list* parameter:

```
column_exclude_list = ['password', ]
```

To **make columns searchable**, or to use them for filtering, specify a list of column names:

```
column_searchable_list = ['name', 'email']
column_filters = ['country']
```

For a faster editing experience, enable **inline editing** in the list view:

```
column_editable_list = ['name', 'last_name']
```

Or, have the add & edit forms display inside a **modal window** on the list page, instead of the dedicated *create* & *edit* pages:

```
create_modal = True
edit_modal = True
```

You can restrict the possible values for a text-field by specifying a list of **select choices**:

```
form_choices = {
    'title': [
        ('MR', 'Mr'),
        ('MRS', 'Mrs'),
        ('MS', 'Ms'),
        ('DR', 'Dr'),
        ('PROF', 'Prof.')
    ]
}
```

To **remove fields** from the create and edit forms:

```
form_excluded_columns = ['last_name', 'email']
```

To specify **WTForms field arguments**:

```
form_args = {
    'name': {
        'label': 'First Name',
        'validators': [required()]
    }
}
```

Or, to specify arguments to the **WTForms widgets** used to render those fields:

```
form_widget_args = {
    'description': {
        'rows': 10,
        'style': 'color: black'
    }
}
```

When your forms contain foreign keys, have those **related models loaded via ajax**, using:

```
form_ajax_refs = {
    'user': {
        'fields': ['first_name', 'last_name', 'email'],
        'page_size': 10
    }
}
```

To filter the results that are loaded via ajax, you can use:

```
form_ajax_refs = {
    'active_user': QueryAjaxModelLoader('user', db.session, User,
                                        filters=["is_active=True", "id>1000"])
}
```

To **manage related models inline**:

```
inline_models = ['post', ]
```

These inline forms can be customized. Have a look at the API documentation for *inline_models()*.

To **enable csv export** of the model view:

```
can_export = True
```

This will add a button to the model view that exports records, truncating at *export_max_rows*.

# Adding Your Own Views

For situations where your requirements are really specific and you struggle to meet them with the built-in `ModelView` class, Flask-Admin makes it easy for you to take full control and add your own views to the interface.

## Standalone Views

A set of standalone views (not tied to any particular model) can be added by extending the *BaseView* class and defining your own view methods. For example, to add a page that displays some analytics data from a 3rd-party API:

```
from flask_admin import BaseView, expose

class AnalyticsView(BaseView):
    @expose('/')
    def index(self):
        return self.render('analytics_index.html')

admin.add_view(AnalyticsView(name='Analytics', endpoint='analytics'))
```

This will add a link to the navbar for your view. Notice that it is served at '/', the root URL. This is a restriction on standalone views: at the very minimum, each view class needs at least one method to serve a view at its root.

The *analytics_index.html* template for the example above, could look something like:

```
{% extends 'admin/master.html' %}
{% block body %}
  <p>Here I'm going to display some data.</p>
{% endblock %}
```

By extending the *admin/master.html* template, you can maintain a consistent user experience, even while having tight control over your page's content.

## Overriding the Built-in Views

There may be some scenarios where you want most of the built-in ModelView functionality, but you want to replace one of the default *create*, *edit*, or *list* views. For this you could override only the view in question, and all the links to it will still function as you would expect:

```
from flask_admin.contrib.sqla import ModelView

# Flask and Flask-SQLAlchemy initialization here

class UserView(ModelView):
    @expose('/new/', methods=('GET', 'POST'))
    def create_view(self):
    """
        Custom create view.
    """

    return self.render('create_user.html')
```

# Working With the Built-in Templates

Flask-Admin uses the Jinja2 templating engine.

## Extending the Built-in Templates

Rather than overriding the built-in templates completely, it's best to extend them. This will make it simpler for you to upgrade to new Flask-Admin versions in future.

Internally, the Flask-Admin templates are derived from the *admin/master.html* template. The three most interesting templates for you to extend are probably:

- *admin/model/list.html*
- *admin/model/create.html*
- *admin/model/edit.html*

To extend the default *edit* template with your own functionality, create a template in *templates/microblog_edit.html* to look something like:

```
{% extends 'admin/model/edit.html' %}

{% block body %}
    <h1>MicroBlog Edit View</h1>
    {{ super() }}
{% endblock %}
```

Now, to make your view classes use this template, set the appropriate class property:

```
class MicroBlogModelView(ModelView):
    edit_template = 'microblog_edit.html'
    # create_template = 'microblog_create.html'
    # list_template = 'microblog_list.html'
```

If you want to use your own base template, then pass the name of the template to the admin constructor during initialization:

```
admin = Admin(app, base_template='microblog_master.html')
```

## Overriding the Built-in Templates

To take full control over the style and layout of the admin interface, you can override all of the built-in templates. Just keep in mind that the templates will change slightly from one version of Flask-Admin to the next, so once you start overriding them, you need to take care when upgrading your package version.

To override any of the built-in templates, simply copy them from the Flask-Admin source into your project's *templates/admin/* directory. As long as the filenames stay the same, the templates in your project directory should automatically take precedence over the built-in ones.

### Available Template Blocks

Flask-Admin defines one *base* template at *admin/master.html* that all other admin templates are derived from. This template is a proxy which points to *admin/base.html*, which defines the following blocks:

| Block Name | Description |
|---|---|
| head_meta | Page metadata in the header |
| title | Page title |
| head_css | Various CSS includes in the header |
| head | Empty block in HTML head, in case you want to put something there |
| page_body | Page layout |
| brand | Logo in the menu bar |
| main_menu | Main menu |
| menu_links | Links menu |
| access_control | Section to the right of the menu (can be used to add login/logout buttons) |
| messages | Alerts and various messages |
| body | Content (that's where your view will be displayed) |
| tail | Empty area below content |

In addition to all of the blocks that are inherited from *admin/master.html*, the *admin/model/list.html* template also contains the following blocks:

| Block Name | Description |
|---|---|
| model_menu_bar | Menu bar |
| model_list_table | Table container |
| list_header | Table header row |
| list_row_actions_header | Actions header |
| list_row | Single row |
| list_row_actions | Row action cell with edit/remove/etc buttons |
| empty_list_message | Message that will be displayed if there are no models found |

Have a look at the *layout* example at https://github.com/flask-admin/flask-admin/tree/master/examples/layout to see how you can take full stylistic control over the admin interface.

## Environment Variables

While working in any of the templates that extend *admin/master.html*, you have access to a small number of environment variables:

| Variable Name | Description |
|---|---|
| admin_view | Current administrative view |
| admin_base_template | Base template name |
| _gettext | Babel gettext |
| _ngettext | Babel ngettext |
| h | Helpers from `helpers` module |

## Generating URLs

To generate the URL for a specific view, use *url_for* with a dot prefix:

```
from flask import url_for

class MyView(BaseView):
    @expose('/')
    def index(self):
        # Get URL for the test view method
        user_list_url = url_for('user.index_view')
        return self.render('index.html', user_list_url=user_list_url)
```

A specific record can also be referenced with:

```
# Edit View for record #1 (redirect back to index_view)
url_for('user.edit_view', id=1, url=url_for('user.index_view'))
```

When referencing ModelView instances, use the lowercase name of the model as the prefix when calling *url_for*. Other views can be referenced by specifying a unique endpoint for each, and using that as the prefix. So, you could use:

```
url_for('analytics.index')
```

If your view endpoint was defined like:

```
admin.add_view(CustomView(name='Analytics', endpoint='analytics'))
```

CHAPTER 2

# Advanced Functionality

## Enabling CSRF Protection

To add CSRF protection to the forms that are generated by *ModelView* instances, use the SecureForm class in your *ModelView* subclass by specifying the *form_base_class* parameter:

```
from flask_admin.form import SecureForm
from flask_admin.contrib.sqla import ModelView

class CarAdmin(ModelView):
    form_base_class = SecureForm
```

SecureForm requires WTForms 2 or greater. It uses the WTForms SessionCSRF class to generate and validate the tokens for you when the forms are submitted.

## Localization With Flask-Babelex

Flask-Admin comes with translations for several languages. Enabling localization is simple:

1. Install Flask-BabelEx to do the heavy lifting. It's a fork of the Flask-Babel package:

   ```
   pip install flask-babelex
   ```

2. Initialize Flask-BabelEx by creating instance of *Babel* class:

   ```
   from flask import app
   from flask_babelex import Babel

   app = Flask(__name__)
   babel = Babel(app)
   ```

3. Create a locale selector function:

```
@babel.localeselector
def get_locale():
    if request.args.get('lang'):
        session['lang'] = request.args.get('lang')
    return session.get('lang', 'en')
```

Now, you could try a French version of the application at: http://localhost:5000/admin/?lang=fr.

Go ahead and add your own logic to the locale selector function. The application can store locale in a user profile, cookie, session, etc. It can also use the *Accept-Language* header to make the selection automatically.

If the built-in translations are not enough, look at the Flask-BabelEx documentation to see how you can add your own.

# Managing Files & Folders

To manage static files instead of database records, Flask-Admin comes with the FileAdmin plug-in. It gives you the ability to upload, delete, rename, etc. You can use it by adding a FileAdmin view to your app:

```
from flask_admin.contrib.fileadmin import FileAdmin

import os.path as op

# Flask setup here

admin = Admin(app, name='microblog', template_mode='bootstrap3')

path = op.join(op.dirname(__file__), 'static')
admin.add_view(FileAdmin(path, '/static/', name='Static Files'))
```

FileAdmin also has out-of-the-box support for managing files located on a Amazon Simple Storage Service bucket. To add it to your app:

```
from flask_admin import Admin
from flask_admin.contrib.fileadmin.s3 import S3FileAdmin

admin = Admin()

admin.add_view(S3FileAdmin('files_bucket', 'us-east-1', 'key_id', 'secret_key')
```

You can disable uploads, disable file deletion, restrict file uploads to certain types, etc. Check *flask_admin.contrib.fileadmin* in the API documentation for more details.

## Adding new file backends

You can also implement your own storage backend by creating a class that implements the same methods defined in the *LocalFileStorage* class. Check *flask_admin.contrib.fileadmin* in the API documentation for details on the methods.

## Adding A Redis Console

Another plug-in that's available is the Redis Console. If you have a Redis instance running on the same machine as your app, you can:

```
from redis import Redis
from flask_admin.contrib import rediscli

# Flask setup here

admin = Admin(app, name='microblog', template_mode='bootstrap3')

admin.add_view(rediscli.RedisCli(Redis()))
```

## Replacing Individual Form Fields

The *form_overrides* attribute allows you to replace individual fields within a form. A common use-case for this would be to add a *What-You-See-Is-What-You-Get* (WYSIWIG) editor, or to handle file / image uploads that need to be tied to a field in your model.

### WYSIWIG Text Fields

To handle complicated text content, you can use CKEditor by subclassing some of the built-in WTForms classes as follows:

```
from wtforms import TextAreaField
from wtforms.widgets import TextArea

class CKTextAreaWidget(TextArea):
    def __call__(self, field, **kwargs):
        if kwargs.get('class'):
            kwargs['class'] += ' ckeditor'
        else:
            kwargs.setdefault('class', 'ckeditor')
        return super(CKTextAreaWidget, self).__call__(field, **kwargs)

class CKTextAreaField(TextAreaField):
    widget = CKTextAreaWidget()

class MessageAdmin(ModelView):
    extra_js = ['//cdn.ckeditor.com/4.6.0/standard/ckeditor.js']

    form_overrides = {
        'body': CKTextAreaField
    }
```

### File & Image Fields

Flask-Admin comes with a built-in *FileUploadField()* and *ImageUploadField()*. To make use of them, you'll need to specify an upload directory and add them to the forms in question. Image handling also requires you to have Pillow installed if you need to do any processing on the image files.

Have a look at the example at https://github.com/flask-admin/Flask-Admin/tree/master/examples/forms.

If you are using the MongoEngine backend, Flask-Admin supports GridFS-backed image and file uploads through WTForms fields. Documentation can be found at *flask_admin.contrib.mongoengine.fields*.

If you just want to manage static files in a directory, without tying them to a database model, then use the *File-Admin* plug-in.

## Managing Geographical Models

---

If you want to store spatial information in a GIS database, Flask-Admin has you covered. The GeoAlchemy backend extends the SQLAlchemy backend (just as GeoAlchemy extends SQLAlchemy) to give you a pretty and functional map-based editor for your admin pages.

Some notable features include:

- Maps are displayed using the amazing Leaflet Javascript library, with map data from Mapbox.
- Geographic information, including points, lines and polygons, can be edited interactively using Leaflet.Draw.
- Graceful fallback: GeoJSON data can be edited in a `<textarea>`, if the user has turned off Javascript.
- Works with a Geometry SQL field that is integrated with Shapely objects.

To get started, define some fields on your model using GeoAlchemy's *Geometry* field. Next, add model views to your interface using the ModelView class from the GeoAlchemy backend, rather than the usual SQLAlchemy backend:

```
from geoalchemy2 import Geometry
from flask_admin.contrib.geoa import ModelView

# .. flask initialization
db = SQLAlchemy(app)

class Location(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), unique=True)
    point = db.Column(Geometry("POINT"))
```

Some of the Geometry field types that are available include: "POINT", "MULTIPOINT", "POLYGON", "MULTIPOLYGON", "LINESTRING" and "MULTILINESTRING".

Have a look at https://github.com/flask-admin/flask-admin/tree/master/examples/geo_alchemy to get started.

### Loading Tiles From Mapbox

To have map data display correctly, you'll have to sign up for an account at https://www.mapbox.com/ and include some credentials in your application's config:

```
app = Flask(__name__)
app.config['MAPBOX_MAP_ID'] = "example.abc123"
app.config['MAPBOX_ACCESS_TOKEN'] = "pk.def456"
```

Leaflet supports loading map tiles from any arbitrary map tile provider, but at the moment, Flask-Admin only supports Mapbox. If you want to use other providers, make a pull request!

### Limitations

There's currently no way to sort, filter, or search on geometric fields in the admin. It's not clear that there's a good way to do so. If you have any ideas or suggestions, make a pull request!

## Customising Builtin Forms Via Rendering Rules

---

Before version 1.0.7, all model backends were rendering the *create* and *edit* forms using a special Jinja2 macro, which was looping over the fields of a WTForms form object and displaying them one by one. This works well, but it is difficult to customize.

Starting from version 1.0.7, Flask-Admin supports form rendering rules, to give you fine grained control of how the forms for your modules should be displayed.

The basic idea is pretty simple: the customizable rendering rules replace a static macro, so you can tell Flask-Admin how each form should be rendered. As an extension, however, the rendering rules also let you do a bit more: You can use them to output HTML, call Jinja2 macros, render fields, and so on.

Essentially, form rendering rules separate the form rendering from the form definition. For example, it no longer matters in which sequence your form fields are defined.

To start using the form rendering rules, put a list of form field names into the *form_create_rules* property one of your admin views:

```
class RuleView(sqla.ModelView):
    form_create_rules = ('email', 'first_name', 'last_name')
```

In this example, only three fields will be rendered and *email* field will be above other two fields.

Whenever Flask-Admin sees a string value in *form_create_rules*, it automatically assumes that it is a form field reference and creates a *flask_admin.form.rules.Field* class instance for that field.

Let's say we want to display some text between the *email* and *first_name* fields. This can be accomplished by using the *flask_admin.form.rules.Text* class:

```
from flask_admin.form import rules

class RuleView(sqla.ModelView):
    form_create_rules = ('email', rules.Text('Foobar'), 'first_name', 'last_name')
```

### Built-in Rules

Flask-Admin comes with few built-in rules that can be found in the *flask_admin.form.rules* module:

| Form Rendering Rule | Description |
|---|---|
| *flask_admin.form.rules.BaseRule* | All rules derive from this class |
| *flask_admin.form.rules.NestedRule* | Allows rule nesting, useful for HTML containers |
| *flask_admin.form.rules.Text* | Simple text rendering rule |
| *flask_admin.form.rules.HTML* | Same as *Text* rule, but does not escape the text |
| *flask_admin.form.rules.Macro* | Calls macro from current Jinja2 context |
| *flask_admin.form.rules.Container* | Wraps child rules into container rendered by macro |
| *flask_admin.form.rules.Field* | Renders single form field |
| *flask_admin.form.rules.Header* | Renders form header |
| *flask_admin.form.rules.FieldSet* | Renders form header and child rules |

# Using Different Database Backends

Other than SQLAlchemy... There are five different backends for you to choose from, depending on which database you would like to use for your application. If, however, you need to implement your own database backend, have a look at *Adding A Model Backend*.

If you don't know where to start, but you're familiar with relational databases, then you should probably look at using *SQLAlchemy*. It is a full-featured toolkit, with support for SQLite, PostgreSQL, MySQL, Oracle and MS-SQL amongst others. It really comes into its own once you have lots of data, and a fair amount of relations between your data models. If you want to track spatial data like latitude/longitude points, you should look into GeoAlchemy, as well.

## SQLAlchemy

Notable features:

- SQLAlchemy 0.6+ support
- Paging, sorting, filters
- Proper model relationship handling
- Inline editing of related models

**Multiple Primary Keys**

Flask-Admin has limited support for models with multiple primary keys. It only covers specific case when all but one primary keys are foreign keys to another model. For example, model inheritance following this convention.

Let's Model a car with its tyres:

```
class Car(db.Model):
    __tablename__ = 'cars'
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    desc = db.Column(db.String(50))

    def __unicode__(self):
        return self.desc

class Tyre(db.Model):
    __tablename__ = 'tyres'
    car_id = db.Column(db.Integer, db.ForeignKey('cars.id'), primary_key=True)
    tyre_id = db.Column(db.Integer, primary_key=True)
```

```
    car = db.relationship('Car', backref='tyres')
    desc = db.Column(db.String(50))
```

A specific tyre is identified by using the two primary key columns of the `Tyre` class, of which the `car_id` key is itself a foreign key to the class `Car`.

To be able to CRUD the `Tyre` class, you need to enumerate columns when defining the AdminView:

```
class TyreAdmin(sqla.ModelView):
    form_columns = ['car', 'tyre_id', 'desc']
```

The `form_columns` needs to be explicit, as per default only one primary key is displayed.

When having multiple primary keys, **no** validation for uniqueness *prior* to saving of the object will be done. Saving a model that violates a unique-constraint leads to an Sqlalchemy-Integrity-Error. In this case, `Flask-Admin` displays a proper error message and you can change the data in the form. When the application has been started with `debug=True` the `werkzeug` debugger will catch the exception and will display the stacktrace.

## MongoEngine

If you're looking for something simpler than SQLAlchemy, and your data models are reasonably self-contained, then MongoDB, a popular *NoSQL* database, could be a better option.

MongoEngine is a python wrapper for MongoDB. For an example of using MongoEngine with Flask-Admin, see https://github.com/flask-admin/flask-admin/tree/master/examples/mongoengine.

Features:

- MongoEngine 0.7+ support
- Paging, sorting, filters, etc
- Supports complex document structure (lists, subdocuments and so on)
- GridFS support for file and image uploads

In order to use MongoEngine integration, install the Flask-MongoEngine package. Flask-Admin uses form scaffolding from it.

Known issues:

- Search functionality can't split query into multiple terms due to MongoEngine query language limitations

For more, check the *`mongoengine`* API documentation.

## Peewee

Features:

- Peewee 2.x+ support;
- Paging, sorting, filters, etc;
- Inline editing of related models;

In order to use peewee integration, you need to install two additional Python packages: peewee and wtf-peewee.

Known issues:

- Many-to-Many model relations are not supported: there's no built-in way to express M2M relation in Peewee

For more, check the *peewee* API documentation. Or look at the Peewee example at https://github.com/flask-admin/flask-admin/tree/master/examples/peewee.

### PyMongo

The bare minimum you have to provide for Flask-Admin to work with PyMongo:

1. A list of columns by setting *column_list* property
2. Provide form to use by setting *form* property
3. When instantiating *flask_admin.contrib.pymongo.ModelView* class, you have to provide PyMongo collection object

This is minimal PyMongo view:

```
class UserForm(Form):
    name = StringField('Name')
    email = StringField('Email')

class UserView(ModelView):
    column_list = ('name', 'email')
    form = UserForm

if __name__ == '__main__':
    admin = Admin(app)

    # 'db' is PyMongo database object
    admin.add_view(UserView(db['users']))
```

On top of that you can add sortable columns, filters, text search, etc.

For more, check the `pymongoe` API documentation. Or look at the Peewee example at https://github.com/flask-admin/flask-admin/tree/master/examples/pymongo.

## Migrating From Django

If you are used to Django and the *django-admin* package, you will find Flask-Admin to work slightly different from what you would expect.

### Design Philosophy

In general, Django and *django-admin* strives to make life easier by implementing sensible defaults. So a developer will be able to get an application up in no time, but it will have to conform to most of the defaults. Of course it is possible to customize things, but this often requires a good understanding of what's going on behind the scenes, and it can be rather tricky and time-consuming.

The design philosophy behind Flask is slightly different. It embraces the diversity that one tends to find in web applications by not forcing design decisions onto the developer. Rather than making it very easy to build an application that *almost* solves your whole problem, and then letting you figure out the last bit, Flask aims to make it possible for you to build the *whole* application. It might take a little more effort to get started, but once you've got the hang of it, the sky is the limit... Even when your application is a little different from most other applications out there on the web.

Flask-Admin follows this same design philosophy. So even though it provides you with several tools for getting up & running quickly, it will be up to you, as a developer, to tell Flask-Admin what should be displayed and how. Even though it is easy to get started with a simple CRUD interface for each model in your application, Flask-Admin doesn't fix you to this approach, and you are free to define other ways of interacting with some, or all, of your models.

Due to Flask-Admin supporting more than one ORM (SQLAlchemy, MongoEngine, Peewee, raw pymongo), the developer is even free to mix different model types into one application by instantiating appropriate CRUD classes.

Here is a list of some of the configuration properties that are made available by Flask-Admin and the SQLAlchemy backend. You can also see which *django-admin* properties they correspond to:

| Django | Flask-Admin |
|---|---|
| actions | *actions* |
| exclude | *form_excluded_columns* |
| fields | *form_columns* |
| form | *form* |
| formfield_overrides | *form_args* |
| inlines | *inline_models* |
| list_display | *column_list* |
| list_filter | *column_filters* |
| list_per_page | *page_size* |
| search_fields | *column_searchable_list* |
| add_form_template | *create_template* |
| change_form_template | change_form_template |

You might want to check *BaseModelView* for basic model configuration options (reused by all model backends) and specific backend documentation, for example *ModelView*. There's much more than what is displayed in this table.

## Overriding the Form Scaffolding

If you don't want to the use the built-in Flask-Admin form scaffolding logic, you are free to roll your own by simply overriding `scaffold_form()`. For example, if you use WTForms-Alchemy, you could put your form generation code into a *scaffold_form* method in your *ModelView* class.

For SQLAlchemy, if the *synonym_property* does not return a SQLAlchemy field, then Flask-Admin won't be able to figure out what to do with it, so it won't generate a form field. In this case, you would need to manually contribute your own field:

```
class MyView(ModelView):
    def scaffold_form(self):
        form_class = super(UserView, self).scaffold_form()
        form_class.extra = StringField('Extra')
        return form_class
```

## Customizing Batch Actions

If you want to add other batch actions to the list view, besides the default delete action, then you can define a function that implements the desired logic and wrap it with the *@action* decorator.

The *action* decorator takes three parameters: *name*, *text* and *confirmation*. While the wrapped function should accept only one parameter - *ids*:

```
from flask_admin.actions import action

class UserView(ModelView):
    @action('approve', 'Approve', 'Are you sure you want to approve selected users?')
    def action_approve(self, ids):
        try:
            query = User.query.filter(User.id.in_(ids))

            count = 0
            for user in query.all():
                if user.approve():
                    count += 1

            flash(ngettext('User was successfully approved.',
                           '%(count)s users were successfully approved.',
                           count,
                           count=count))
        except Exception as ex:
            if not self.handle_view_exception(ex):
                raise

            flash(gettext('Failed to approve users. %(error)s', error=str(ex)), 'error
↪')
```

CHAPTER 3

# Adding A Model Backend

Flask-Admin makes a few assumptions about the database models that it works with. If you want to implement your own database backend, and still have Flask-Admin's model views work as expected, then you should take note of the following:

1. Each model must have one field which acts as a *primary key* to uniquely identify instances of that model. However, there are no restriction on the data type or the field name of the *primary key* field.
2. Models must make their data accessible as python properties.

If that is the case, then you can implement your own database backend by extending the *BaseModelView* class, and implementing the set of scaffolding methods listed below.

## Extending BaseModelView

Start off by defining a new class, which derives from from *BaseModelView*:

```
class MyDbModel(BaseModelView):
    pass
```

This class inherits BaseModelView's *__init__* method, which accepts a model class as first argument. The model class is stored as the attribute `self.model` so that other methods may access it.

Now, implement the following scaffolding methods for the new class:

1. *get_pk_value()*

   This method returns a primary key value from the model instance. In the SQLAlchemy backend, it gets the primary key from the model using *scaffold_pk()*, caches it and then returns the value from the model whenever requested.

   For example:

   ```
   class MyDbModel(BaseModelView):
       def get_pk_value(self, model):
           return self.model.id
   ```

2. *scaffold_list_columns()*

   Returns a list of columns to be displayed in a list view. For example:

```
class MyDbModel(BaseModelView):
    def scaffold_list_columns(self):
        columns = []

        for p in dir(self.model):
            attr = getattr(self.model, p)
            if isinstance(attr, MyDbColumn):
                columns.append(p)

        return columns
```

3. *scaffold_sortable_columns()*

   Returns a dictionary of sortable columns. The keys in the dictionary should correspond to the model's field names. The values should be those variables that will be used for sorting.

   For example, in the SQLAlchemy backend it is possible to sort by a foreign key field. So, if there is a field named *user*, which is a foreign key for the *Users* table, and the *Users* table also has a name field, then the key will be *user* and value will be *Users.name*.

   If your backend does not support sorting, return *None* or an empty dictionary.

4. *init_search()*

   Initialize search functionality. If your backend supports full-text search, do initializations and return *True*. If your backend does not support full-text search, return *False*.

   For example, SQLAlchemy backend reads value of the *self.searchable_columns* and verifies if all fields are of text type, if they're local to the current model (if not, it will add a join, etc) and caches this information for future use.

5. *scaffold_form()*

   Generate *WTForms* form class from the model.

   For example:

```
class MyDbModel(BaseModelView):
    def scaffold_form(self):
        class MyForm(Form):
            pass

        # Do something
        return MyForm
```

6. *get_list()*

   This method should return list of model instances with paging, sorting, etc applied.

   For SQLAlchemy backend it looks like:

   (a) If search was enabled and provided search value is not empty, generate LIKE statements for each field from *self.searchable_columns*

   (b) If filter values were passed, call *apply* method with values:

```
for flt, value in filters:
    query = self._filters[flt].apply(query, value)
```

(c) Execute query to get total number of rows in the database (count)

(d) If *sort_column* was passed, will do something like (with some extra FK logic which is omitted in this example):

```
if sort_desc:
    query = query.order_by(desc(sort_field))
else:
    query = query.order_by(sort_field)
```

(e) Apply paging

(f) Return count, list as a tuple

7. *get_one()*

   Return a model instance by its primary key.

8. *create_model()*

   Create a new instance of the model from the *Form* object.

9. *update_model()*

   Update the model instance with data from the form.

10. *delete_model()*

    Delete the specified model instance from the data store.

11. *is_valid_filter()*

    Verify whether the given object is a valid filter.

12. *scaffold_filters()*

    Return a list of filter objects for one model field.

    This method will be called once for each entry in the *self.column_filters* setting.

    If your backend does not know how to generate filters for the provided field, it should return *None*.

    For example:

```
class MyDbModel(BaseModelView):
    def scaffold_filters(self, name):
        attr = getattr(self.model, name)

        if isinstance(attr, MyDbTextField):
            return [MyEqualFilter(name, name)]
```

## Implementing filters

Each model backend should have its own set of filter implementations. It is not possible to use the filters from SQLAlchemy models in a non-SQLAlchemy backend. This also means that different backends might have different set of available filters.

The filter is a class derived from `BaseFilter` which implements at least two methods:

1. `apply()`
2. `operation()`

*apply* method accepts two parameters: *query* object and a value from the client. Here you can add filtering logic for the filter type.

Lets take SQLAlchemy model backend as an example:

All SQLAlchemy filters derive from `BaseSQLAFilter` class.

Each filter implements one simple filter SQL operation (like, not like, greater, etc) and accepts a column as input parameter.

Whenever model view wants to apply a filter to a query object, it will call *apply* method in a filter class with a query and value. Filter will then apply real filter operation.

For example:

```
class MyBaseFilter(BaseFilter):
    def __init__(self, column, name, options=None, data_type=None):
        super(MyBaseFilter, self).__init__(name, options, data_type)

        self.column = column

class MyEqualFilter(MyBaseFilter):
    def apply(self, query, value):
        return query.filter(self.column == value)

    def operation(self):
        return gettext('equals')

    # You can validate values. If value is not valid,
    # return `False`, so filter will be ignored.
    def validate(self, value):
        return True

    # You can "clean" values before they will be
    # passed to the your data access layer
    def clean(self, value):
        return value
```

Feel free ask questions if you have problems adding a new model backend. Also, if you get stuck, try taking a look at the SQLAlchemy model backend and use it as a reference.

CHAPTER 4

# API

## flask_admin.base

### Base View

**expose**(*url='/'*, *methods=('GET',)*)
Use this decorator to expose views in your view classes.

**Parameters**

- **url** – Relative URL for the view
- **methods** – Allowed HTTP methods. By default only GET is allowed.

**expose_plugview**(*url='/'*)
Decorator to expose Flask's pluggable view classes (`flask.views.View` or `flask.views.MethodView`).

**Parameters** **url** – Relative URL for the view

New in version 1.0.4.

**class BaseView**(*name=None*, *category=None*, *endpoint=None*, *url=None*, *static_folder=None*, *static_url_path=None*, *menu_class_name=None*, *menu_icon_type=None*, *menu_icon_value=None*)
Base administrative view.

Derive from this class to implement your administrative interface piece. For example:

```
from flask_admin import BaseView, expose
class MyView(BaseView):
    @expose('/')
    def index(self):
        return 'Hello World!'
```

Icons can be added to the menu by using *menu_icon_type* and *menu_icon_value*. For example:

```
admin.add_view(MyView(name='My View', menu_icon_type='glyph', menu_icon_value=
↪'glyphicon-home'))
```

**create_blueprint** (*admin*)
Create Flask blueprint.

**get_url** (*endpoint*, ***kwargs*)
Generate URL for the endpoint. If you want to customize URL generation logic (persist some query string argument, for example), this is right place to do it.

**Parameters**

- **endpoint** – Flask endpoint name
- **kwargs** – Arguments for *url_for*

**inaccessible_callback** (*name*, ***kwargs*)
Handle the response to inaccessible views.

By default, it throw HTTP 403 error. Override this method to customize the behaviour.

**is_accessible** ()
Override this method to add permission checks.

Flask-Admin does not make any assumptions about the authentication system used in your application, so it is up to you to implement it.

By default, it will allow access for everyone.

**is_visible** ()
Override this method if you want dynamically hide or show administrative views from Flask-Admin menu structure

By default, item is visible in menu.

Please note that item should be both visible and accessible to be displayed in menu.

**render** (*template*, ***kwargs*)
Render template

**Parameters**

- **template** – Template path to render
- **kwargs** – Template arguments

## Default view

**class AdminIndexView** (*name=None*, *category=None*, *endpoint=None*, *url=None*, *template='admin/index.html'*, *menu_class_name=None*, *menu_icon_type=None*, *menu_icon_value=None*)
Default administrative interface index page when visiting the `/admin/` URL.

It can be overridden by passing your own view class to the `Admin` constructor:

```
class MyHomeView(AdminIndexView):
    @expose('/')
    def index(self):
        arg1 = 'Hello'
        return self.render('admin/myhome.html', arg1=arg1)

admin = Admin(index_view=MyHomeView())
```

Also, you can change the root url from /admin to / with the following:

```
admin = Admin(
    app,
    index_view=AdminIndexView(
        name='Home',
        template='admin/myhome.html',
        url='/'
    )
)
```

Default values for the index page are:

- If a name is not provided, 'Home' will be used.
- If an endpoint is not provided, will default to `admin`
- Default URL route is `/admin`.
- Automatically associates with static folder.
- Default template is `admin/index.html`

## Admin

**class Admin**(*app=None*, *name=None*, *url=None*, *subdomain=None*, *index_view=None*, *translations_path=None*, *endpoint=None*, *static_url_path=None*, *base_template=None*, *template_mode=None*, *category_icon_classes=None*)

Collection of the admin views. Also manages menu structure.

**add_link**(*link*)

Add link to menu links collection.

**Parameters** **link** – Link to add.

**add_links**(**args*)

Add one or more links to the menu links collection.

Examples:

```
admin.add_links(link1)
admin.add_links(link1, link2, link3, link4)
admin.add_links(*my_list)
```

**Parameters** **args** – Argument list including the links to add.

**add_menu_item**(*menu_item*, *target_category=None*)

Add menu item to menu tree hierarchy.

**Parameters**

- **menu_item** – MenuItem class instance
- **target_category** – Target category name

**add_view**(*view*)

Add a view to the collection.

**Parameters** **view** – View to add.

**add_views**(**args*)
Add one or more views to the collection.

Examples:

```
admin.add_views(view1)
admin.add_views(view1, view2, view3, view4)
admin.add_views(*my_list)
```

**Parameters** **args** – Argument list including the views to add.

**init_app**(*app*, *index_view=None*, *endpoint=None*, *url=None*)
Register all views with the Flask application.

**Parameters** **app** – Flask application instance

**menu**()
Return the menu hierarchy.

**menu_links**()
Return menu links.

## flask_admin.helpers

**get_current_view**()
Get current administrative view.

Forms

**is_required_form_field**(*field*)
Check if form field has *DataRequired* or *InputRequired* validators.

**Parameters** **field** – WTForms field to check

**is_form_submitted**()
Check if current method is PUT or POST

**validate_form_on_submit**(*form*)
If current method is PUT or POST, validate form and return validation status.

**get_form_data**()
If current method is PUT or POST, return concatenated *request.form* with *request.files* or *None* otherwise.

**is_field_error**(*errors*)
Check if wtforms field has error without checking its children.

**Parameters** **errors** – Errors list.

Jinja2 helpers

**resolve_ctx**(*context*)
Resolve current Jinja2 context and store it for general consumption.

**get_render_ctx**()
Get view template context.

## flask_admin.model

**class BaseModelView**(*model*, *name=None*, *category=None*, *endpoint=None*, *url=None*, *static_folder=None*, *menu_class_name=None*, *menu_icon_type=None*, *menu_icon_value=None*)

Base model view.

This view does not make any assumptions on how models are stored or managed, but expects the following:

1. The provided model is an object
2. The model contains properties
3. Each model contains an attribute which uniquely identifies it (i.e. a primary key for a database model)
4. It is possible to retrieve a list of sorted models with pagination applied from a data source
5. You can get one model by its identifier from the data source

Essentially, if you want to support a new data store, all you have to do is:

1. Derive from the *BaseModelView* class
2. Implement various data-related methods (*get_list*, *get_one*, *create_model*, etc)
3. Implement automatic form generation from the model representation (*scaffold_form*)

**can_create = True**
Is model creation allowed

**can_edit = True**
Is model editing allowed

**can_delete = True**
Is model deletion allowed

**list_template = 'admin/model/list.html'**
Default list view template

**edit_template = 'admin/model/edit.html'**
Default edit template

**create_template = 'admin/model/create.html'**
Default create template

**column_list**
Collection of the model field names for the list view. If set to *None*, will get them from the model.

For example:

```
class MyModelView(BaseModelView):
    column_list = ('name', 'last_name', 'email')
```

(Added in 1.4.0) SQLAlchemy model attributes can be used instead of strings:

```
class MyModelView(BaseModelView):
    column_list = ('name', User.last_name)
```

**When using SQLAlchemy models, you can reference related columns like this::**

**class MyModelView(BaseModelView):** column_list = ('<relationship>.<related column name>',)

**column_exclude_list**
Collection of excluded list column names.

For example:

```
class MyModelView(BaseModelView):
    column_exclude_list = ('last_name', 'email')
```

**column_labels**
Dictionary where key is column name and value is string to display.

For example:

```
class MyModelView(BaseModelView):
    column_labels = dict(name='Name', last_name='Last Name')
```

**column_descriptions = None**
Dictionary where key is column name and value is description for *list view* column or add/edit form field.

For example:

```
class MyModelView(BaseModelView):
    column_descriptions = dict(
        full_name='First and Last name'
    )
```

**column_formatters**
Dictionary of list view column formatters.

For example, if you want to show price multiplied by two, you can do something like this:

```
class MyModelView(BaseModelView):
    column_formatters = dict(price=lambda v, c, m, p: m.price*2)
```

or using Jinja2 *macro* in template:

```
from flask_admin.model.template import macro

class MyModelView(BaseModelView):
    column_formatters = dict(price=macro('render_price'))

# in template
{% macro render_price(model, column) %}
    {{ model.price * 2 }}
{% endmacro %}
```

The Callback function has the prototype:

```
def formatter(view, context, model, name):
    # `view` is current administrative view
    # `context` is instance of jinja2.runtime.Context
    # `model` is model instance
    # `name` is property name
    pass
```

**column_type_formatters**
Dictionary of value type formatters to be used in the list view.

By default, three types are formatted:

1. `None` will be displayed as an empty string
2. `bool` will be displayed as a checkmark if it is `True`
3. `list` will be joined using ', '

If you don't like the default behavior and don't want any type formatters applied, just override this property with an empty dictionary:

```
class MyModelView(BaseModelView):
    column_type_formatters = dict()
```

If you want to display *NULL* instead of an empty string, you can do something like this. Also comes with bonus *date* formatter:

```
from datetime import date
from flask_admin.model import typefmt

def date_format(view, value):
    return value.strftime('%d.%m.%Y')

MY_DEFAULT_FORMATTERS = dict(typefmt.BASE_FORMATTERS)
MY_DEFAULT_FORMATTERS.update({
        type(None): typefmt.null_formatter,
        date: date_format
    })

class MyModelView(BaseModelView):
    column_type_formatters = MY_DEFAULT_FORMATTERS
```

Type formatters have lower priority than list column formatters.

The callback function has following prototype:

```
def type_formatter(view, value):
    # `view` is current administrative view
    # `value` value to format
    pass
```

**column_display_pk**
Controls if the primary key should be displayed in the list view.

**column_sortable_list**
Collection of the sortable columns for the list view. If set to *None*, will get them from the model.

For example:

```
class MyModelView(BaseModelView):
    column_sortable_list = ('name', 'last_name')
```

If you want to explicitly specify field/column to be used while sorting, you can use a tuple:

```
class MyModelView(BaseModelView):
    column_sortable_list = ('name', ('user', 'user.username'))
```

When using SQLAlchemy models, model attributes can be used instead of strings:

```
class MyModelView(BaseModelView):
    column_sortable_list = ('name', ('user', User.username))
```

**column_searchable_list**

A collection of the searchable columns. It is assumed that only text-only fields are searchable, but it is up to the model implementation to decide.

Example:

```
class MyModelView(BaseModelView):
    column_searchable_list = ('name', 'email')
```

**column_default_sort = None**

Default sort column if no sorting is applied.

Example:

```
class MyModelView(BaseModelView):
    column_default_sort = 'user'
```

You can use tuple to control ascending descending order. In following example, items will be sorted in descending order:

```
class MyModelView(BaseModelView):
    column_default_sort = ('user', True)
```

**column_choices = None**

Map choices to columns in list view

Example:

```
class MyModelView(BaseModelView):
    column_choices = {
        'my_column': [
            ('db_value', 'display_value'),
        ]
    }
```

**column_filters = None**

Collection of the column filters.

Can contain either field names or instances of `BaseFilter` classes.

Example:

```
class MyModelView(BaseModelView):
    column_filters = ('user', 'email')
```

**form = None**

Form class. Override if you want to use custom form for your model. Will completely disable form scaffolding functionality.

For example:

```
class MyForm(Form):
    name = StringField('Name')

class MyModelView(BaseModelView):
    form = MyForm
```

**form_base_class = <class 'flask_admin.form.BaseForm'>**

Base form class. Will be used by form scaffolding function when creating model form.

Useful if you want to have custom constructor or override some fields.

Example:

```
class MyBaseForm(Form):
    def do_something(self):
        pass

class MyModelView(BaseModelView):
    form_base_class = MyBaseForm
```

**form_columns = None**

Collection of the model field names for the form. If set to *None* will get them from the model.

Example:

```
class MyModelView(BaseModelView):
    form_columns = ('name', 'email')
```

(Added in 1.4.0) SQLAlchemy model attributes can be used instead of strings:

```
class MyModelView(BaseModelView):
    form_columns = ('name', User.last_name)
```

SQLA Note: Model attributes must be on the same model as your ModelView or you will need to use *inline_models*.

**form_excluded_columns**

Collection of excluded form field names.

For example:

```
class MyModelView(BaseModelView):
    form_excluded_columns = ('last_name', 'email')
```

**form_args = None**

Dictionary of form field arguments. Refer to WTForms documentation for list of possible options.

Example:

```
from wtforms.validators import DataRequired
class MyModelView(BaseModelView):
    form_args = dict(
        name=dict(label='First Name', validators=[DataRequired()])
    )
```

**form_overrides = None**

Dictionary of form column overrides.

Example:

```
class MyModelView(BaseModelView):
    form_overrides = dict(name=wtf.FileField)
```

**form_widget_args = None**

Dictionary of form widget rendering arguments. Use this to customize how widget is rendered without using custom template.

Example:

```
class MyModelView(BaseModelView):
    form_widget_args = {
        'description': {
            'rows': 10,
            'style': 'color: black'
        },
        'other_field': {
            'disabled': True
        }
    }
```

Changing the format of a DateTimeField will require changes to both form_widget_args and form_args.

Example:

```
form_args = dict(
    start=dict(format='%Y-%m-%d %I:%M %p') # changes how the input is parsed by strptime (12 hour time)
)
form_widget_args = dict(
    start={
        'data-date-format': u'yyyy-mm-dd HH:ii P',
        'data-show-meridian': 'True'
    } # changes how the DateTimeField displays the time
)
```

**form_extra_fields = None**

Dictionary of additional fields.

Example:

```
class MyModelView(BaseModelView):
    form_extra_fields = {
        'password': PasswordField('Password')
    }
```

You can control order of form fields using `form_columns` property. For example:

```
class MyModelView(BaseModelView):
    form_columns = ('name', 'email', 'password', 'secret')

    form_extra_fields = {
        'password': PasswordField('Password')
    }
```

In this case, password field will be put between email and secret fields that are autogenerated.

**form_ajax_refs = None**

Use AJAX for foreign key model loading.

Should contain dictionary, where key is field name and value is either a dictionary which configures AJAX lookups or backend-specific *AjaxModelLoader* class instance.

For example, it can look like:

```
class MyModelView(BaseModelView):
    form_ajax_refs = {
        'user': {
            'fields': ('first_name', 'last_name', 'email'),
```

```
            'page_size': 10
        }
    }
```

Or with SQLAlchemy backend like this:

```
class MyModelView(BaseModelView):
    form_ajax_refs = {
        'user': QueryAjaxModelLoader('user', db.session, User, fields=['email
↪'], page_size=10)
    }
```

If you need custom loading functionality, you can implement your custom loading behavior in your *AjaxModelLoader* class.

**form_create_rules = None**
Customized rules for the create form. Override *form_rules* if present.

**form_edit_rules = None**
Customized rules for the edit form. Override *form_rules* if present.

**action_disallowed_list**
Set of disallowed action names. For example, if you want to disable mass model deletion, do something like this:

**class MyModelView(BaseModelView):** action_disallowed_list = ['delete']

**page_size = 20**
Default page size for pagination.

**can_set_page_size = False**
Allows to select page size via dropdown list

**action_form**(*obj=None*)
Instantiate model action form and return it.

Override to implement custom behavior.

**action_view**(**args*, ***kwargs*)
Mass-model action view.

**after_model_change**(*form*, *model*, *is_created*)
Perform some actions after a model was created or updated and committed to the database.

Called from create_model after successful database commit.

By default does nothing.

**Parameters**

- **form** – Form used to create/update model
- **model** – Model that was created/updated
- **is_created** – True if model was created, False if model was updated

**after_model_delete**(*model*)
Perform some actions after a model was deleted and committed to the database.

Called from delete_model after successful database commit (if it has any meaning for a store backend).

By default does nothing.

**Parameters** **model** – Model that was deleted

**ajax_update**(**args*, ***kwargs*)
Edits a single column of a record in list view.

**can_export = False**
Is model list export allowed

**can_view_details = False**
Setting this to true will enable the details view. This is recommended when there are too many columns to display in the list_view.

**column_details_exclude_list = None**
Collection of fields excluded from the details view.

**column_details_list = None**
Collection of the field names included in the details view. If set to *None*, will get them from the model.

**column_display_actions = True**
Controls the display of the row actions (edit, delete, details, etc.) column in the list view.

Useful for preventing a blank column from displaying if your view does not use any build-in or custom row actions.

This column is not hidden automatically due to backwards compatibility.

Note: This only affects display and does not control whether the row actions endpoints are accessible.

**column_editable_list = None**
Collection of the columns which can be edited from the list view.

For example:

```
class MyModelView(BaseModelView):
    column_editable_list = ('name', 'last_name')
```

**column_export_exclude_list = None**
Collection of fields excluded from the export.

**column_export_list = None**
Collection of the field names included in the export. If set to *None*, will get them from the model.

**column_extra_row_actions = None**
List of row actions (instances of `BaseListRowAction`).

Flask-Admin will generate standard per-row actions (edit, delete, etc) and will append custom actions from this list right after them.

For example:

```
from flask_admin.model.template import EndpointLinkRowAction, LinkRowAction

class MyModelView(BaseModelView):
    column_extra_row_actions = [
        LinkRowAction('glyphicon glyphicon-off', 'http://direct.link/?id={row_
↪id}'),
        EndpointLinkRowAction('glyphicon glyphicon-test', 'my_view.index_view
↪')
    ]
```

**column_formatters_export = None**
Dictionary of list view column formatters to be used for export.

Defaults to column_formatters when set to None.

Functions the same way as column_formatters except that macros are not supported.

**column_type_formatters_export = None**
Dictionary of value type formatters to be used in the export.

By default, two types are formatted:

1. `None` will be displayed as an empty string
2. `list` will be joined using ', '

Functions the same way as column_type_formatters.

**create_form**(*obj=None*)
Instantiate model creation form and return it.

Override to implement custom behavior.

**create_modal = False**
Setting this to true will display the create_view as a modal dialog.

**create_modal_template = 'admin/model/modals/create.html'**
Default create modal template

**create_model**(*form*)
Create model from the form.

Returns the model instance if operation succeeded.

Must be implemented in the child class.

**Parameters** **form** – Form instance

**create_view**(**args*, ***kwargs*)
Create model view

**delete_form**()
Instantiate model delete form and return it.

Override to implement custom behavior.

The delete form originally used a GET request, so delete_form accepts both GET and POST request for backwards compatibility.

**delete_model**(*model*)
Delete model.

Returns *True* if operation succeeded.

Must be implemented in the child class.

**Parameters** **model** – Model instance

**delete_view**(**args*, ***kwargs*)
Delete model view. Only POST method is allowed.

**details_modal = False**
Setting this to true will display the details_view as a modal dialog.

**details_modal_template = 'admin/model/modals/details.html'**
Default details modal view template

**details_template = 'admin/model/details.html'**
Default details view template

**details_view**(**args*, ***kwargs*)
Details model view

**edit_form**(*obj=None*)
Instantiate model editing form and return it.

Override to implement custom behavior.

**edit_modal = False**
Setting this to true will display the edit_view as a modal dialog.

**edit_modal_template = 'admin/model/modals/edit.html'**
Default edit modal template

**edit_view**(**args*, ***kwargs*)
Edit model view

**export_max_rows = 0**
Maximum number of rows allowed for export.

Unlimited by default. Uses *page_size* if set to *None*.

**export_types = ['csv']**
A list of available export filetypes. *csv* only is default, but any filetypes supported by tablib can be used.

Check tablib for https://github.com/kennethreitz/tablib/blob/master/README.rst for supported types.

**form_rules = None**
List of rendering rules for model creation form.

This property changed default form rendering behavior and makes possible to rearrange order of rendered fields, add some text between fields, group them, etc. If not set, will use default Flask-Admin form rendering logic.

Here's simple example which illustrates how to use:

```
from flask_admin.form import rules

class MyModelView(ModelView):
    form_rules = [
        # Define field set with header text and four fields
        rules.FieldSet(('first_name', 'last_name', 'email', 'phone'), 'User'),
        # ... and it is just shortcut for:
        rules.Header('User'),
        rules.Field('first_name'),
        rules.Field('last_name'),
        # ...
        # It is possible to create custom rule blocks:
        MyBlock('Hello World'),
        # It is possible to call macros from current context
        rules.Macro('my_macro', foobar='baz')
    ]
```

**get_action_form**()
Create form class for a model action.

Override to implement customized behavior.

**get_column_name**(*field*)
Return a human-readable column name.

**Parameters** **field** – Model field name.

**get_column_names**(*only_columns*, *excluded_columns*)
Returns a list of tuples with the model field name and formatted field name.

**Parameters**

- **only_columns** – List of columns to include in the results. If not set, *scaffold_list_columns* will generate the list from the model.
- **excluded_columns** – List of columns to exclude from the results if *only_columns* is not set.

**get_create_form**()
Create form class for model creation view.

Override to implement customized behavior.

**get_delete_form**()
Create form class for model delete view.

Override to implement customized behavior.

**get_details_columns**()
Uses *get_column_names* to get a list of tuples with the model field name and formatted name for the columns in *column_details_list* and not in *column_details_exclude_list*. If *column_details_list* is not set, the columns from *scaffold_list_columns* will be used.

**get_edit_form**()
Create form class for model editing view.

Override to implement customized behavior.

**get_export_columns**()
Uses *get_column_names* to get a list of tuples with the model field name and formatted name for the columns in *column_export_list* and not in *column_export_exclude_list*. If *column_export_list* is not set, it will attempt to use the columns from *column_list* or finally the columns from *scaffold_list_columns* will be used.

**get_export_name**(*export_type='csv'*)

**Returns** The exported csv file name.

**get_export_value**(*model*, *name*)
Returns the value to be displayed in export. Allows export to use different (non HTML) formatters.

**Parameters**

- **model** – Model instance
- **name** – Field name

**get_filter_arg**(*index*, *flt*)
Given a filter *flt*, return a unique name for that filter in this view.

Does not include the *flt[n]_* portion of the filter name.

**Parameters**

- **index** – Filter index in _filters array
- **flt** – Filter instance

**get_filters**()
Return a list of filter objects.

If your model backend implementation does not support filters, override this method and return *None*.

**get_form**()
Get form class.

If `self.form` is set, will return it and will call `self.scaffold_form` otherwise.

Override to implement customized behavior.

**get_list**(*page*, *sort_field*, *sort_desc*, *search*, *filters*, *page_size=None*)
Return a paginated and sorted list of models from the data source.

Must be implemented in the child class.

**Parameters**

- **page** – Page number, 0 based. Can be set to None if it is first page.
- **sort_field** – Sort column name or None.
- **sort_desc** – If set to True, sorting is in descending order.
- **search** – Search query
- **filters** – List of filter tuples. First value in a tuple is a search index, second value is a search value.
- **page_size** – Number of results. Defaults to ModelView's page_size. Can be overriden to change the page_size limit. Removing the page_size limit requires setting page_size to 0 or False.

**get_list_columns**()
Uses *get_column_names* to get a list of tuples with the model field name and formatted name for the columns in *column_list* and not in *column_exclude_list*. If *column_list* is not set, the columns from *scaffold_list_columns* will be used.

**get_list_form**()
Get form class for the editable list view.

Uses only validators from *form_args* to build the form class.

Allows overriding the editable list view field/widget. For example:

```
from flask_admin.model.widgets import XEditableWidget

class CustomWidget(XEditableWidget):
    def get_kwargs(self, subfield, kwargs):
        if subfield.type == 'TextAreaField':
            kwargs['data-type'] = 'textarea'
            kwargs['data-rows'] = '20'
        # elif: kwargs for other fields

        return kwargs

class MyModelView(BaseModelView):
    def get_list_form(self):
        return self.scaffold_list_form(widget=CustomWidget)
```

**get_list_row_actions**()
Return list of row action objects, each is instance of `BaseListRowAction`

**get_list_value**(*context*, *model*, *name*)
Returns the value to be displayed in the list view

**Parameters**

- **context** – `jinja2.runtime.Context`
- **model** – Model instance

- **name** – Field name

**get_one**(*id*)
Return one model by its id.

Must be implemented in the child class.

**Parameters** **id** – Model id

**get_pk_value**(*model*)
Return PK value from a model object.

**get_save_return_url**(*model*, *is_created=False*)
Return url where user is redirected after successful form save.

**Parameters**

- **model** – Saved object
- **is_created** – Whether new object was created or existing one was updated

For example, redirect use to object details view after form save:

```
class MyModelView(ModelView):
    can_view_details = True

    def get_save_return_url(self, model, is_created):
        return self.get_url('.details_view', id=model.id)
```

**get_sortable_columns**()
Returns a dictionary of the sortable columns. Key is a model field name and value is sort column (for example - attribute).

If *column_sortable_list* is set, will use it. Otherwise, will call *scaffold_sortable_columns* to get them from the model.

**handle_filter**(*filter*)
Postprocess (add joins, etc) for a filter.

**Parameters** **filter** – Filter object to postprocess

**index_view**(**args*, ***kwargs*)
List view

**init_search**()
Initialize search. If data provider does not support search, *init_search* will return *False*.

**is_action_allowed**(*name*)
Override this method to allow or disallow actions based on some condition.

The default implementation only checks if the particular action is not in *action_disallowed_list*.

**is_editable**(*name*)
Verify if column is editable.

**Parameters** **name** – Column name.

**is_sortable**(*name*)
Verify if column is sortable.

Not case-sensitive.

**Parameters** **name** – Column name.

**is_valid_filter**(*filter*)
Verify that the provided filter object is valid.

Override in model backend implementation to verify if the provided filter type is allowed.

**Parameters** **filter** – Filter object to verify.

**list_form**(*obj=None*)
Instantiate model editing form for list view and return it.

Override to implement custom behavior.

**named_filter_urls = False**
Set to True to use human-readable names for filters in URL parameters.

False by default so as to be robust across translations.

Changing this parameter will break any existing URLs that have filters.

**on_form_prefill**(*form*, *id*)
Perform additional actions to pre-fill the edit form.

Called from edit_view, if the current action is rendering the form rather than receiving client side input, after default pre-filling has been performed.

By default does nothing.

You only need to override this if you have added custom fields that depend on the database contents in a way that Flask-admin can't figure out by itself. Fields that were added by name of a normal column or relationship should work out of the box.

**Parameters**

- **form** – Form instance
- **id** – id of the object that is going to be edited

**on_model_change**(*form*, *model*, *is_created*)
Perform some actions before a model is created or updated.

Called from create_model and update_model in the same transaction (if it has any meaning for a store backend).

By default does nothing.

**Parameters**

- **form** – Form used to create/update model
- **model** – Model that will be created/updated
- **is_created** – Will be set to True if model was created and to False if edited

**on_model_delete**(*model*)
Perform some actions before a model is deleted.

Called from delete_model in the same transaction (if it has any meaning for a store backend).

By default do nothing.

**scaffold_filters**(*name*)
Generate filter object for the given name

**Parameters** **name** – Name of the field

**scaffold_form**()
Create *form.BaseForm* inherited class from the model. Must be implemented in the child class.

**scaffold_list_columns**()
Return list of the model field names. Must be implemented in the child class.

Expected return format is list of tuples with field name and display text. For example:

```
['name', 'first_name', 'last_name']
```

**scaffold_list_form**(*widget=None*, *validators=None*)
Create form for the *index_view* using only the columns from *self.column_editable_list*.

**Parameters**

- **widget** – WTForms widget class. Defaults to *XEditableWidget*.
- **validators** – *form_args* dict with only validators {'name': {'validators': [DataRequired()]}}

Must be implemented in the child class.

**scaffold_sortable_columns**()
Returns dictionary of sortable columns. Must be implemented in the child class.

Expected return format is a dictionary, where keys are field names and values are property names.

**simple_list_pager = False**
Enable or disable simple list pager. If enabled, model interface would not run count query and will only show prev/next pager buttons.

**update_model**(*form*, *model*)
Update model from the form.

Returns *True* if operation succeeded.

Must be implemented in the child class.

**Parameters**

- **form** – Form instance
- **model** – Model instance

**validate_form**(*form*)
Validate the form on submit.

**Parameters** **form** – Form to validate

## flask_admin.form

**class BaseForm**(*formdata=None*, *obj=None*, *prefix=u''*, ***kwargs*)

## flask_admin.form.rules

**class BaseRule**
Base form rule. All form formatting rules should derive from *BaseRule*.

**class NestedRule**(*rules=[]*, *separator=''*)
Nested rule. Can contain child rules and render them.

**__init__**(*rules=[]*, *separator=''*)
Constructor.

**Parameters**

- **rules** – Child rule list
- **separator** – Default separator between rules when rendering them.

**class Text** (*text*, *escape=True*)
Render text (or HTML snippet) from string.

**__init__** (*text*, *escape=True*)
Constructor.

**Parameters**

- **text** – Text to render
- **escape** – Should text be escaped or not. Default is *True*.

**class HTML** (*html*)
Shortcut for *Text* rule with *escape* set to *False*.

**class Macro** (*macro_name*, ***kwargs*)
Render macro by its name from current Jinja2 context.

**__init__** (*macro_name*, ***kwargs*)
Constructor.

**Parameters**

- **macro_name** – Macro name
- **kwargs** – Default macro parameters

**class Container** (*macro_name*, *child_rule*, ***kwargs*)
Render container around child rule.

**__init__** (*macro_name*, *child_rule*, ***kwargs*)
Constructor.

**Parameters**

- **macro_name** – Macro name that will be used as a container
- **child_rule** – Child rule to be rendered inside of container
- **kwargs** – Container macro arguments

**class Field** (*field_name*, *render_field='lib.render_field'*)
Form field rule.

**__init__** (*field_name*, *render_field='lib.render_field'*)
Constructor.

**Parameters**

- **field_name** – Field name to render
- **render_field** – Macro that will be used to render the field.

**class Header** (*text*, *header_macro='lib.render_header'*)
Render header text.

**__init__** (*text*, *header_macro='lib.render_header'*)
Constructor.

**Parameters**

- **text** – Text to render

- **header_macro** – Header rendering macro

**class FieldSet**(*rules*, *header=None*, *separator=''*)
Field set with header.

**__init__**(*rules*, *header=None*, *separator=''*)
Constructor.

**Parameters**

- **rules** – Child rules
- **header** – Header text
- **separator** – Child rule separator

## flask_admin.form.fields

**class TimeField**(*label=None*, *validators=None*, *formats=None*, *default_format=None*, *widget_format=None*, ***kwargs*)
A text field which stores a *datetime.time* object. Accepts time string in multiple formats: 20:10, 20:10:00, 10:00 am, 9:30pm, etc.

**class Select2Field**(*label=None*, *validators=None*, *coerce=<type 'unicode'>*, *choices=None*, *allow_blank=False*, *blank_text=None*, ***kwargs*)
Select2 styled select widget.

You must include select2.js, form-x.x.x.js and select2 stylesheet for it to work.

**class Select2TagsField**(*label=None*, *validators=None*, *save_as_list=False*, *coerce=<type 'unicode'>*, ***kwargs*)
Select2 styled text field. You must include select2.js, form-x.x.x.js and select2 stylesheet for it to work.

## flask_admin.form.upload

**class FileUploadField**(*label=None*, *validators=None*, *base_path=None*, *relative_path=None*, *namegen=None*, *allowed_extensions=None*, *permission=438*, *allow_overwrite=True*, ***kwargs*)
Customizable file-upload field.

Saves file to configured path, handles updates and deletions. Inherits from *StringField*, resulting filename will be stored as string.

**__init__**(*label=None*, *validators=None*, *base_path=None*, *relative_path=None*, *namegen=None*, *allowed_extensions=None*, *permission=438*, *allow_overwrite=True*, ***kwargs*)
Constructor.

**Parameters**

- **label** – Display label
- **validators** – Validators
- **base_path** – Absolute path to the directory which will store files
- **relative_path** – Relative path from the directory. Will be prepended to the file name for uploaded files. Flask-Admin uses *urlparse.urljoin* to generate resulting filename, so make sure you have trailing slash.

- **namegen** – Function that will generate filename from the model and uploaded file object. Please note, that model is "dirty" model object, before it was committed to database.

  For example:

```
import os.path as op

def prefix_name(obj, file_data):
    parts = op.splitext(file_data.filename)
    return secure_filename('file-%s%s' % parts)

class MyForm(BaseForm):
    upload = FileUploadField('File', namegen=prefix_name)
```

- **allowed_extensions** – List of allowed extensions. If not provided, will allow any file.
- **allow_overwrite** – Whether to overwrite existing files in upload directory. Defaults to *True*.

New in version 1.1.1: The *allow_overwrite* parameter was added.

**class ImageUploadField**(*label=None*, *validators=None*, *base_path=None*, *relative_path=None*, *namegen=None*, *allowed_extensions=None*, *max_size=None*, *thumbgen=None*, *thumbnail_size=None*, *permission=438*, *url_relative_path=None*, *endpoint='static'*, ***kwargs*)

Image upload field.

Does image validation, thumbnail generation, updating and deleting images.

Requires PIL (or Pillow) to be installed.

**__init__**(*label=None*, *validators=None*, *base_path=None*, *relative_path=None*, *namegen=None*, *allowed_extensions=None*, *max_size=None*, *thumbgen=None*, *thumbnail_size=None*, *permission=438*, *url_relative_path=None*, *endpoint='static'*, ***kwargs*)

Constructor.

**Parameters**

- **label** – Display label
- **validators** – Validators
- **base_path** – Absolute path to the directory which will store files
- **relative_path** – Relative path from the directory. Will be prepended to the file name for uploaded files. Flask-Admin uses *urlparse.urljoin* to generate resulting filename, so make sure you have trailing slash.
- **namegen** – Function that will generate filename from the model and uploaded file object. Please note, that model is "dirty" model object, before it was committed to database.

  For example:

```
import os.path as op

def prefix_name(obj, file_data):
    parts = op.splitext(file_data.filename)
    return secure_filename('file-%s%s' % parts)

class MyForm(BaseForm):
    upload = FileUploadField('File', namegen=prefix_name)
```

- **allowed_extensions** – List of allowed extensions. If not provided, then gif, jpg, jpeg, png and tiff will be allowed.
- **max_size** – Tuple of (width, height, force) or None. If provided, Flask-Admin will resize image to the desired size.

  Width and height is in pixels. If *force* is set to *True*, will try to fit image into dimensions and keep aspect ratio, otherwise will just resize to target size.
- **thumbgen** – Thumbnail filename generation function. All thumbnails will be saved as JPEG files, so there's no need to keep original file extension.

  For example:

```
import os.path as op

def thumb_name(filename):
    name, _ = op.splitext(filename)
    return secure_filename('%s-thumb.jpg' % name)

class MyForm(BaseForm):
    upload = ImageUploadField('File', thumbgen=thumb_name)
```

- **thumbnail_size** – Tuple or (width, height, force) values. If not provided, thumbnail won't be created.

  Width and height is in pixels. If *force* is set to *True*, will try to fit image into dimensions and keep aspect ratio, otherwise will just resize to target size.
- **url_relative_path** – Relative path from the root of the static directory URL. Only gets used when generating preview image URLs.

  For example, your model might store just file names (*relative_path* set to *None*), but *base_path* is pointing to subdirectory.
- **endpoint** – Static endpoint for images. Used by widget to display previews. Defaults to 'static'.

**class FileUploadInput**
Renders a file input chooser field.

You can customize *empty_template* and *data_template* members to customize look and feel.

**class ImageUploadInput**
Renders a image input chooser field.

You can customize *empty_template* and *data_template* members to customize look and feel.

## flask_admin.tools

**import_module**(*name*, *required=True*)
Import module by name

**Parameters**

- **name** – Module name
- **required** – If set to *True* and module was not found - will throw exception. If set to *False* and module was not found - will return None. Default is *True*.

**import_attribute**(*name*)
Import attribute using string reference.

**Parameters** **name** – String reference.

Raises ImportError or AttributeError if module or attribute do not exist.

Example:

```
import_attribute('a.b.c.foo')
```

**module_not_found**(*additional_depth=0*)
Checks if ImportError was raised because module does not exist or something inside it raised ImportError

**Parameters** **additional_depth** – supply int of depth of your call if you're not doing import on the same level of code - f.e., if you call function, which is doing import, you should pass 1 for single additional level of depth

**rec_getattr**(*obj*, *attr*, *default=None*)
Recursive getattr.

**Parameters**

- **attr** – Dot delimited attribute name
- **default** – Default value

Example:

```
rec_getattr(obj, 'a.b.c')
```

## flask_admin.actions

**action**(*name*, *text*, *confirmation=None*)
Use this decorator to expose actions that span more than one entity (model, file, etc)

**Parameters**

- **name** – Action name
- **text** – Action text.
- **confirmation** – Confirmation text. If not provided, action will be executed unconditionally.

**class ActionsMixin**
Actions mixin.

In some cases, you might work with more than one "entity" (model, file, etc) in your admin view and will want to perform actions on a group of entities simultaneously.

In this case, you can add this functionality by doing this: 1. Add this mixin to your administrative view class 2. Call *init_actions* in your class constructor 3. Expose actions view 4. Import *actions.html* library and add call library macros in your template

**get_actions_list**()
Return a list and a dictionary of allowed actions.

**handle_action**(*return_view=None*)
Handle action request.

**Parameters** **return_view** – Name of the view to return to after the request. If not provided, will return user to the return url in the form or the list view.

**init_actions**()
Initialize list of actions for the current administrative view.

**is_action_allowed**(*name*)
Verify if action with *name* is allowed.

**Parameters** **name** – Action name

## flask_admin.contrib.sqla

SQLAlchemy model backend implementation.

**class ModelView**(*model*, *session*, *name=None*, *category=None*, *endpoint=None*, *url=None*, *static_folder=None*, *menu_class_name=None*, *menu_icon_type=None*, *menu_icon_value=None*)
SQLAlchemy model view

Usage sample:

```
admin = Admin()
admin.add_view(ModelView(User, db.session))
```

Class inherits configuration options from *BaseModelView* and they're not displayed here.

**column_auto_select_related**
Enable automatic detection of displayed foreign keys in this view and perform automatic joined loading for related models to improve query performance.

Please note that detection is not recursive: if *__unicode__* method of related model uses another model to generate string representation, it will still make separate database call.

**column_select_related_list**
List of parameters for SQLAlchemy *subqueryload*. Overrides *column_auto_select_related* property.

For example:

```
class PostAdmin(ModelView):
    column_select_related_list = ('user', 'city')
```

You can also use properties:

```
class PostAdmin(ModelView):
    column_select_related_list = (Post.user, Post.city)
```

Please refer to the *subqueryload* on list of possible values.

**column_searchable_list**
Collection of the searchable columns.

Example:

```
class MyModelView(ModelView):
    column_searchable_list = ('name', 'email')
```

You can also pass columns:

```
class MyModelView(ModelView):
    column_searchable_list = (User.name, User.email)
```

The following search rules apply:

- If you enter `ZZZ` in the UI search field, it will generate `ILIKE '%ZZZ%'` statement against searchable columns.
- If you enter multiple words, each word will be searched separately, but only rows that contain all words will be displayed. For example, searching for `abc def` will find all rows that contain `abc` and `def` in one or more columns.
- If you prefix your search term with `^`, it will find all rows that start with `^`. So, if you entered `^ZZZ` then `ILIKE 'ZZZ%'` will be used.
- If you prefix your search term with =, it will perform an exact match. For example, if you entered `=ZZZ`, the statement `ILIKE 'ZZZ'` will be used.

**column_filters = None**

Collection of the column filters.

Can contain either field names or instances of `flask_admin.contrib.sqla.filters.BaseSQLAFilter` classes.

Filters will be grouped by name when displayed in the drop-down.

For example:

```
class MyModelView(BaseModelView):
    column_filters = ('user', 'email')
```

or:

```
from flask_admin.contrib.sqla.filters import BooleanEqualFilter

class MyModelView(BaseModelView):
    column_filters = (BooleanEqualFilter(column=User.name, name='Name'),)
```

or:

```
from flask_admin.contrib.sqla.filters import BaseSQLAFilter

class FilterLastNameBrown(BaseSQLAFilter):
    def apply(self, query, value, alias=None):
        if value == '1':
            return query.filter(self.column == "Brown")
        else:
            return query.filter(self.column != "Brown")

    def operation(self):
        return 'is Brown'

class MyModelView(BaseModelView):
    column_filters = [
        FilterLastNameBrown(
            User.last_name, 'Last Name', options=(('1', 'Yes'), ('0', 'No'))
        )
    ]
```

**filter_converter = <flask_admin.contrib.sqla.filters.FilterConverter object>**
Field to filter converter.

Override this attribute to use non-default converter.

**model_form_converter = <class 'flask_admin.contrib.sqla.form.AdminModelConverter'>**
Model form conversion class. Use this to implement custom field conversion logic.

For example:

```
class MyModelConverter(AdminModelConverter):
    pass


class MyAdminView(ModelView):
    model_form_converter = MyModelConverter
```

**inline_model_form_converter = <class 'flask_admin.contrib.sqla.form.InlineModelConverter'>**
Inline model conversion class. If you need some kind of post-processing for inline forms, you can customize behavior by doing something like this:

```
class MyInlineModelConverter(InlineModelConverter):
    def post_process(self, form_class, info):
        form_class.value = wtf.StringField('value')
        return form_class

class MyAdminView(ModelView):
    inline_model_form_converter = MyInlineModelConverter
```

**fast_mass_delete = False**
If set to *False* and user deletes more than one model using built in action, all models will be read from the database and then deleted one by one giving SQLAlchemy a chance to manually cleanup any dependencies (many-to-many relationships, etc).

If set to *True*, will run a `DELETE` statement which is somewhat faster, but may leave corrupted data if you forget to configure `DELETE CASCADE` for your model.

**inline_models = None**
Inline related-model editing for models with parent-child relations.

Accepts enumerable with one of the following possible values:

1.Child model class:

```
class MyModelView(ModelView):
    inline_models = (Post,)
```

2.Child model class and additional options:

```
class MyModelView(ModelView):
    inline_models = [(Post, dict(form_columns=['title']))]
```

3.Django-like `InlineFormAdmin` class instance:

```
from flask_admin.model.form import InlineFormAdmin

class MyInlineModelForm(InlineFormAdmin):
    form_columns = ('title', 'date')
```

```
class MyModelView(ModelView):
    inline_models = (MyInlineModelForm(MyInlineModel),)
```

You can customize the generated field name by:

1. Using the *form_name* property as a key to the options dictionary:

```
class MyModelView(ModelView):
    inline_models = ((Post, dict(form_label='Hello')))
```

2. Using forward relation name and *column_labels* property:

```
class Model1(Base):
    pass

class Model2(Base):
    # ...
    model1 = relation(Model1, backref='models')

class MyModel1View(Base):
    inline_models = (Model2,)
    column_labels = {'models': 'Hello'}
```

**form_choices = None**

Map choices to form fields

Example:

```
class MyModelView(BaseModelView):
    form_choices = {'my_form_field': [
        ('db_value', 'display_value'),
    ]}
```

**form_optional_types = (<class 'sqlalchemy.sql.sqltypes.Boolean'>,)**

List of field types that should be optional if column is not nullable.

Example:

```
class MyModelView(BaseModelView):
    form_optional_types = (Boolean, Unicode)
```

**action_form**(*obj=None*)

Instantiate model action form and return it.

Override to implement custom behavior.

**action_view**(**args*, ***kwargs*)

Mass-model action view.

**after_model_change**(*form*, *model*, *is_created*)

Perform some actions after a model was created or updated and committed to the database.

Called from create_model after successful database commit.

By default does nothing.

**Parameters**

- **form** – Form used to create/update model
- **model** – Model that was created/updated

- **is_created** – True if model was created, False if model was updated

**after_model_delete**(*model*)
Perform some actions after a model was deleted and committed to the database.

Called from delete_model after successful database commit (if it has any meaning for a store backend).

By default does nothing.

**Parameters** **model** – Model that was deleted

**ajax_update**(**args*, ***kwargs*)
Edits a single column of a record in list view.

**column_display_all_relations**
Controls if list view should display all relations, not only many-to-one.

**create_blueprint**(*admin*)
Create Flask blueprint.

**create_form**(*obj=None*)
Instantiate model creation form and return it.

Override to implement custom behavior.

**create_model**(*form*)
Create model from form.

**Parameters** **form** – Form instance

**create_view**(**args*, ***kwargs*)
Create model view

**delete_form**()
Instantiate model delete form and return it.

Override to implement custom behavior.

The delete form originally used a GET request, so delete_form accepts both GET and POST request for backwards compatibility.

**delete_model**(*model*)
Delete model.

**Parameters** **model** – Model to delete

**delete_view**(**args*, ***kwargs*)
Delete model view. Only POST method is allowed.

**details_view**(**args*, ***kwargs*)
Details model view

**edit_form**(*obj=None*)
Instantiate model editing form and return it.

Override to implement custom behavior.

**edit_view**(**args*, ***kwargs*)
Edit model view

**get_action_form**()
Create form class for a model action.

Override to implement customized behavior.

**get_actions_list**()
Return a list and a dictionary of allowed actions.

**get_column_name**(*field*)
Return a human-readable column name.

**Parameters** **field** – Model field name.

**get_column_names**(*only_columns*, *excluded_columns*)
Returns a list of tuples with the model field name and formatted field name.

Overridden to handle special columns like InstrumentedAttribute.

**Parameters**

- **only_columns** – List of columns to include in the results. If not set, *scaffold_list_columns* will generate the list from the model.
- **excluded_columns** – List of columns to exclude from the results.

**get_count_query**()
Return a the count query for the model type

A `query(self.model).count()` approach produces an excessive subquery, so `query(func.count('*'))` should be used instead.

See commit `#45a2723` for details.

**get_create_form**()
Create form class for model creation view.

Override to implement customized behavior.

**get_delete_form**()
Create form class for model delete view.

Override to implement customized behavior.

**get_details_columns**()
Uses *get_column_names* to get a list of tuples with the model field name and formatted name for the columns in *column_details_list* and not in *column_details_exclude_list*. If *column_details_list* is not set, the columns from *scaffold_list_columns* will be used.

**get_edit_form**()
Create form class for model editing view.

Override to implement customized behavior.

**get_export_columns**()
Uses *get_column_names* to get a list of tuples with the model field name and formatted name for the columns in *column_export_list* and not in *column_export_exclude_list*. If *column_export_list* is not set, it will attempt to use the columns from *column_list* or finally the columns from *scaffold_list_columns* will be used.

**get_export_name**(*export_type='csv'*)

**Returns** The exported csv file name.

**get_export_value**(*model*, *name*)
Returns the value to be displayed in export. Allows export to use different (non HTML) formatters.

**Parameters**

- **model** – Model instance
- **name** – Field name

**get_filter_arg**(*index*, *flt*)
Given a filter *flt*, return a unique name for that filter in this view.

Does not include the *flt[n]_* portion of the filter name.

**Parameters**

- **index** – Filter index in _filters array
- **flt** – Filter instance

**get_filters**()
Return a list of filter objects.

If your model backend implementation does not support filters, override this method and return *None*.

**get_form**()
Get form class.

If `self.form` is set, will return it and will call `self.scaffold_form` otherwise.

Override to implement customized behavior.

**get_list**(*page*, *sort_column*, *sort_desc*, *search*, *filters*, *execute=True*, *page_size=None*)
Return records from the database.

**Parameters**

- **page** – Page number
- **sort_column** – Sort column name
- **sort_desc** – Descending or ascending sort
- **search** – Search query
- **execute** – Execute query immediately? Default is *True*
- **filters** – List of filter tuples
- **page_size** – Number of results. Defaults to ModelView's page_size. Can be overriden to change the page_size limit. Removing the page_size limit requires setting page_size to 0 or False.

**get_list_columns**()
Uses *get_column_names* to get a list of tuples with the model field name and formatted name for the columns in *column_list* and not in *column_exclude_list*. If *column_list* is not set, the columns from *scaffold_list_columns* will be used.

**get_list_form**()
Get form class for the editable list view.

Uses only validators from *form_args* to build the form class.

Allows overriding the editable list view field/widget. For example:

```
from flask_admin.model.widgets import XEditableWidget

class CustomWidget(XEditableWidget):
    def get_kwargs(self, subfield, kwargs):
        if subfield.type == 'TextAreaField':
            kwargs['data-type'] = 'textarea'
            kwargs['data-rows'] = '20'
        # elif: kwargs for other fields

        return kwargs
```

```
class MyModelView(BaseModelView):
    def get_list_form(self):
        return self.scaffold_list_form(widget=CustomWidget)
```

**get_list_row_actions**()
Return list of row action objects, each is instance of `BaseListRowAction`

**get_list_value**(*context*, *model*, *name*)
Returns the value to be displayed in the list view

**Parameters**

- **context** – `jinja2.runtime.Context`
- **model** – Model instance
- **name** – Field name

**get_one**(*id*)
Return a single model by its id.

**Parameters** **id** – Model id

**get_pk_value**(*model*)
Return the primary key value from a model object. If there are multiple primary keys, they're encoded into string representation.

**get_query**()
Return a query for the model type.

If you override this method, don't forget to override *get_count_query* as well.

This method can be used to set a "persistent filter" on an index_view.

Example:

```
class MyView(ModelView):
    def get_query(self):
        return super(MyView, self).get_query().filter(User.username ==␣
↪current_user.username)
```

**get_save_return_url**(*model*, *is_created=False*)
Return url where user is redirected after successful form save.

**Parameters**

- **model** – Saved object
- **is_created** – Whether new object was created or existing one was updated

For example, redirect use to object details view after form save:

```
class MyModelView(ModelView):
    can_view_details = True

    def get_save_return_url(self, model, is_created):
        return self.get_url('.details_view', id=model.id)
```

**get_sortable_columns**()
Returns a dictionary of the sortable columns. Key is a model field name and value is sort column (for example - attribute).

If *column_sortable_list* is set, will use it. Otherwise, will call *scaffold_sortable_columns* to get them from the model.

**get_url**(*endpoint*, ***kwargs*)
Generate URL for the endpoint. If you want to customize URL generation logic (persist some query string argument, for example), this is right place to do it.

**Parameters**

- **endpoint** – Flask endpoint name
- **kwargs** – Arguments for *url_for*

**handle_action**(*return_view=None*)
Handle action request.

**Parameters** **return_view** – Name of the view to return to after the request. If not provided, will return user to the return url in the form or the list view.

**ignore_hidden = True**
Ignore field that starts with "_"

Example:

```
class MyModelView(BaseModelView):
    ignore_hidden = False
```

**inaccessible_callback**(*name*, ***kwargs*)
Handle the response to inaccessible views.

By default, it throw HTTP 403 error. Override this method to customize the behaviour.

**index_view**(**args*, ***kwargs*)
List view

**init_actions**()
Initialize list of actions for the current administrative view.

**init_search**()
Initialize search. Returns *True* if search is supported for this view.

For SQLAlchemy, this will initialize internal fields: list of column objects used for filtering, etc.

**is_accessible**()
Override this method to add permission checks.

Flask-Admin does not make any assumptions about the authentication system used in your application, so it is up to you to implement it.

By default, it will allow access for everyone.

**is_editable**(*name*)
Verify if column is editable.

**Parameters** **name** – Column name.

**is_sortable**(*name*)
Verify if column is sortable.

Not case-sensitive.

**Parameters** **name** – Column name.

**is_valid_filter**(*filter*)
Verify that the provided filter object is valid.

Override in model backend implementation to verify if the provided filter type is allowed.

**Parameters** **filter** – Filter object to verify.

**is_visible**()
Override this method if you want dynamically hide or show administrative views from Flask-Admin menu structure

By default, item is visible in menu.

Please note that item should be both visible and accessible to be displayed in menu.

**list_form**(*obj=None*)
Instantiate model editing form for list view and return it.

Override to implement custom behavior.

**on_form_prefill**(*form*, *id*)
Perform additional actions to pre-fill the edit form.

Called from edit_view, if the current action is rendering the form rather than receiving client side input, after default pre-filling has been performed.

By default does nothing.

You only need to override this if you have added custom fields that depend on the database contents in a way that Flask-admin can't figure out by itself. Fields that were added by name of a normal column or relationship should work out of the box.

**Parameters**

- **form** – Form instance
- **id** – id of the object that is going to be edited

**on_model_change**(*form*, *model*, *is_created*)
Perform some actions before a model is created or updated.

Called from create_model and update_model in the same transaction (if it has any meaning for a store backend).

By default does nothing.

**Parameters**

- **form** – Form used to create/update model
- **model** – Model that will be created/updated
- **is_created** – Will be set to True if model was created and to False if edited

**on_model_delete**(*model*)
Perform some actions before a model is deleted.

Called from delete_model in the same transaction (if it has any meaning for a store backend).

By default do nothing.

**render**(*template*, ***kwargs*)
Render template

**Parameters**

- **template** – Template path to render
- **kwargs** – Template arguments

**scaffold_auto_joins**()
Return a list of joined tables by going through the displayed columns.

**scaffold_filters**(*name*)
Return list of enabled filters

**scaffold_form**()
Create form from the model.

**scaffold_inline_form_models**(*form_class*)
Contribute inline models to the form

**Parameters** **form_class** – Form class

**scaffold_list_columns**()
Return a list of columns from the model.

**scaffold_list_form**(*widget=None*, *validators=None*)
Create form for the *index_view* using only the columns from *self.column_editable_list*.

**Parameters**

- **widget** – WTForms widget class. Defaults to *XEditableWidget*.
- **validators** – *form_args* dict with only validators {'name': {'validators': [required()]}}

**scaffold_pk**()
Return the primary key name(s) from a model If model has single primary key, will return a string and tuple otherwise

**scaffold_sortable_columns**()
Return a dictionary of sortable columns. Key is column name, value is sort column/field.

**update_model**(*form*, *model*)
Update model from form.

**Parameters**

- **form** – Form instance
- **model** – Model instance

**validate_form**(*form*)
Validate the form on submit.

**Parameters** **form** – Form to validate

## flask_admin.contrib.mongoengine

MongoEngine model backend implementation.

**class ModelView**(*model*, *name=None*, *category=None*, *endpoint=None*, *url=None*, *static_folder=None*, *menu_class_name=None*, *menu_icon_type=None*, *menu_icon_value=None*)
MongoEngine model scaffolding.

Class inherits configuration options from *BaseModelView* and they're not displayed here.

**column_filters = None**
Collection of the column filters.

Can contain either field names or instances of `flask_admin.contrib.mongoengine.filters.BaseMongoEngineFilter` classes.

Filters will be grouped by name when displayed in the drop-down.

For example:

```
class MyModelView(BaseModelView):
    column_filters = ('user', 'email')
```

or:

```
from flask_admin.contrib.mongoengine.filters import BooleanEqualFilter

class MyModelView(BaseModelView):
    column_filters = (BooleanEqualFilter(column=User.name, name='Name'),)
```

or:

```
from flask_admin.contrib.mongoengine.filters import BaseMongoEngineFilter

class FilterLastNameBrown(BaseMongoEngineFilter):
    def apply(self, query, value):
        if value == '1':
            return query.filter(self.column == "Brown")
        else:
            return query.filter(self.column != "Brown")

    def operation(self):
        return 'is Brown'

class MyModelView(BaseModelView):
    column_filters = [
        FilterLastNameBrown(
            column=User.last_name, name='Last Name',
            options=(('1', 'Yes'), ('0', 'No'))
        )
    ]
```

**column_type_formatters = {<type 'NoneType'>: <function empty_formatter>, <class 'mongoengine.base.datastruct**
Customized type formatters for MongoEngine backend

**filter_converter = <flask_admin.contrib.mongoengine.filters.FilterConverter object>**
Field to filter converter.

Override this attribute to use a non-default converter.

**model_form_converter = <class 'flask_admin.contrib.mongoengine.form.CustomModelConverter'>**
Model form conversion class. Use this to implement custom field conversion logic.

Custom class should be derived from the *flask_admin.contrib.mongoengine.form.CustomModelConverter*.

For example:

```
class MyModelConverter(AdminModelConverter):
    pass


class MyAdminView(ModelView):
    model_form_converter = MyModelConverter
```

**allowed_search_types = (<class 'mongoengine.fields.StringField'>, <class 'mongoengine.fields.URLField'>, <class '**
List of allowed search field types.

**form_subdocuments = None**

Subdocument configuration options.

This field accepts dictionary, where key is field name and value is either dictionary or instance of the *flask_admin.contrib.mongoengine.EmbeddedForm*.

Consider following example:

```
class Comment(db.EmbeddedDocument):
    name = db.StringField(max_length=20, required=True)
    value = db.StringField(max_length=20)

class Post(db.Document):
    text = db.StringField(max_length=30)
    data = db.EmbeddedDocumentField(Comment)

class MyAdmin(ModelView):
    form_subdocuments = {
        'data': {
            'form_columns': ('name',)
        }
    }
```

In this example, *Post* model has child *Comment* subdocument. When generating form for *Comment* embedded document, Flask-Admin will only create *name* field.

It is also possible to use class-based embedded document configuration:

```
class CommentEmbed(EmbeddedForm):
    form_columns = ('name',)

class MyAdmin(ModelView):
    form_subdocuments = {
        'data': CommentEmbed()
    }
```

Arbitrary depth nesting is supported:

```
class SomeEmbed(EmbeddedForm):
    form_excluded_columns = ('test',)

class CommentEmbed(EmbeddedForm):
    form_columns = ('name',)
    form_subdocuments = {
        'inner': SomeEmbed()
    }

class MyAdmin(ModelView):
    form_subdocuments = {
        'data': CommentEmbed()
    }
```

There's also support for forms embedded into *ListField*. All you have to do is to create nested rule with *None* as a name. Even though it is slightly confusing, but that's how Flask-MongoEngine creates form fields embedded into ListField:

```
class Comment(db.EmbeddedDocument):
    name = db.StringField(max_length=20, required=True)
    value = db.StringField(max_length=20)
```

```
class Post(db.Document):
    text = db.StringField(max_length=30)
    data = db.ListField(db.EmbeddedDocumentField(Comment))

class MyAdmin(ModelView):
    form_subdocuments = {
        'data': {
            'form_subdocuments': {
                None: {
                    'form_columns': ('name',)
                }
            }

        }
    }
```

**action_form**(*obj=None*)
Instantiate model action form and return it.

Override to implement custom behavior.

**action_view**(**args*, ***kwargs*)
Mass-model action view.

**after_model_change**(*form*, *model*, *is_created*)
Perform some actions after a model was created or updated and committed to the database.

Called from create_model after successful database commit.

By default does nothing.

**Parameters**

- **form** – Form used to create/update model
- **model** – Model that was created/updated
- **is_created** – True if model was created, False if model was updated

**after_model_delete**(*model*)
Perform some actions after a model was deleted and committed to the database.

Called from delete_model after successful database commit (if it has any meaning for a store backend).

By default does nothing.

**Parameters** **model** – Model that was deleted

**ajax_update**(**args*, ***kwargs*)
Edits a single column of a record in list view.

**allowed_search_types = (<class 'mongoengine.fields.StringField'>, <class 'mongoengine.fields.URLField'>, <class '**
List of allowed search field types.

**create_blueprint**(*admin*)
Create Flask blueprint.

**create_form**(*obj=None*)
Instantiate model creation form and return it.

Override to implement custom behavior.

**create_model**(*form*)
Create model helper

**Parameters** **form** – Form instance

**create_view**(**args*, ***kwargs*)
Create model view

**delete_form**()
Instantiate model delete form and return it.

Override to implement custom behavior.

The delete form originally used a GET request, so delete_form accepts both GET and POST request for backwards compatibility.

**delete_model**(*model*)
Delete model helper

**Parameters** **model** – Model instance

**delete_view**(**args*, ***kwargs*)
Delete model view. Only POST method is allowed.

**details_view**(**args*, ***kwargs*)
Details model view

**edit_form**(*obj=None*)
Instantiate model editing form and return it.

Override to implement custom behavior.

**edit_view**(**args*, ***kwargs*)
Edit model view

**get_action_form**()
Create form class for a model action.

Override to implement customized behavior.

**get_actions_list**()
Return a list and a dictionary of allowed actions.

**get_column_name**(*field*)
Return a human-readable column name.

**Parameters** **field** – Model field name.

**get_column_names**(*only_columns*, *excluded_columns*)
Returns a list of tuples with the model field name and formatted field name.

**Parameters**

- **only_columns** – List of columns to include in the results. If not set, *scaffold_list_columns* will generate the list from the model.
- **excluded_columns** – List of columns to exclude from the results if *only_columns* is not set.

**get_create_form**()
Create form class for model creation view.

Override to implement customized behavior.

**get_delete_form**()
Create form class for model delete view.

Override to implement customized behavior.

**get_details_columns**()
Uses *get_column_names* to get a list of tuples with the model field name and formatted name for the columns in *column_details_list* and not in *column_details_exclude_list*. If *column_details_list* is not set, the columns from *scaffold_list_columns* will be used.

**get_edit_form**()
Create form class for model editing view.

Override to implement customized behavior.

**get_export_columns**()
Uses *get_column_names* to get a list of tuples with the model field name and formatted name for the columns in *column_export_list* and not in *column_export_exclude_list*. If *column_export_list* is not set, it will attempt to use the columns from *column_list* or finally the columns from *scaffold_list_columns* will be used.

**get_export_name**(*export_type='csv'*)

**Returns** The exported csv file name.

**get_export_value**(*model*, *name*)
Returns the value to be displayed in export. Allows export to use different (non HTML) formatters.

**Parameters**

- **model** – Model instance
- **name** – Field name

**get_filter_arg**(*index*, *flt*)
Given a filter *flt*, return a unique name for that filter in this view.

Does not include the *flt[n]_* portion of the filter name.

**Parameters**

- **index** – Filter index in _filters array
- **flt** – Filter instance

**get_filters**()
Return a list of filter objects.

If your model backend implementation does not support filters, override this method and return *None*.

**get_form**()
Get form class.

If `self.form` is set, will return it and will call `self.scaffold_form` otherwise.

Override to implement customized behavior.

**get_list**(*page*, *sort_column*, *sort_desc*, *search*, *filters*, *execute=True*, *page_size=None*)
Get list of objects from MongoEngine

**Parameters**

- **page** – Page number
- **sort_column** – Sort column
- **sort_desc** – Sort descending

- **search** – Search criteria
- **filters** – List of applied filters
- **execute** – Run query immediately or not
- **page_size** – Number of results. Defaults to ModelView's page_size. Can be overriden to change the page_size limit. Removing the page_size limit requires setting page_size to 0 or False.

**get_list_columns**()
Uses *get_column_names* to get a list of tuples with the model field name and formatted name for the columns in *column_list* and not in *column_exclude_list*. If *column_list* is not set, the columns from *scaffold_list_columns* will be used.

**get_list_form**()
Get form class for the editable list view.

Uses only validators from *form_args* to build the form class.

Allows overriding the editable list view field/widget. For example:

```
from flask_admin.model.widgets import XEditableWidget

class CustomWidget(XEditableWidget):
    def get_kwargs(self, subfield, kwargs):
        if subfield.type == 'TextAreaField':
            kwargs['data-type'] = 'textarea'
            kwargs['data-rows'] = '20'
        # elif: kwargs for other fields

        return kwargs

class MyModelView(BaseModelView):
    def get_list_form(self):
        return self.scaffold_list_form(widget=CustomWidget)
```

**get_list_row_actions**()
Return list of row action objects, each is instance of `BaseListRowAction`

**get_list_value**(*context*, *model*, *name*)
Returns the value to be displayed in the list view

**Parameters**

- **context** – `jinja2.runtime.Context`
- **model** – Model instance
- **name** – Field name

**get_one**(*id*)
Return a single model instance by its ID

**Parameters** **id** – Model ID

**get_pk_value**(*model*)
Return the primary key value from the model instance

**Parameters** **model** – Model instance

**get_query**()
Returns the QuerySet for this view. By default, it returns all the objects for the current model.

**get_save_return_url**(*model*, *is_created=False*)
Return url where user is redirected after successful form save.

**Parameters**

- **model** – Saved object
- **is_created** – Whether new object was created or existing one was updated

For example, redirect use to object details view after form save:

```
class MyModelView(ModelView):
    can_view_details = True

    def get_save_return_url(self, model, is_created):
        return self.get_url('.details_view', id=model.id)
```

**get_sortable_columns**()
Returns a dictionary of the sortable columns. Key is a model field name and value is sort column (for example - attribute).

If *column_sortable_list* is set, will use it. Otherwise, will call *scaffold_sortable_columns* to get them from the model.

**get_url**(*endpoint*, ***kwargs*)
Generate URL for the endpoint. If you want to customize URL generation logic (persist some query string argument, for example), this is right place to do it.

**Parameters**

- **endpoint** – Flask endpoint name
- **kwargs** – Arguments for *url_for*

**handle_action**(*return_view=None*)
Handle action request.

**Parameters** **return_view** – Name of the view to return to after the request. If not provided, will return user to the return url in the form or the list view.

**handle_filter**(*filter*)
Postprocess (add joins, etc) for a filter.

**Parameters** **filter** – Filter object to postprocess

**inaccessible_callback**(*name*, ***kwargs*)
Handle the response to inaccessible views.

By default, it throw HTTP 403 error. Override this method to customize the behaviour.

**index_view**(**args*, ***kwargs*)
List view

**init_actions**()
Initialize list of actions for the current administrative view.

**init_search**()
Init search

**is_accessible**()
Override this method to add permission checks.

Flask-Admin does not make any assumptions about the authentication system used in your application, so it is up to you to implement it.

By default, it will allow access for everyone.

**is_editable**(*name*)
Verify if column is editable.

**Parameters** **name** – Column name.

**is_sortable**(*name*)
Verify if column is sortable.

Not case-sensitive.

**Parameters** **name** – Column name.

**is_valid_filter**(*filter*)
Validate if the provided filter is a valid MongoEngine filter

**Parameters** **filter** – Filter object

**is_visible**()
Override this method if you want dynamically hide or show administrative views from Flask-Admin menu structure

By default, item is visible in menu.

Please note that item should be both visible and accessible to be displayed in menu.

**list_form**(*obj=None*)
Instantiate model editing form for list view and return it.

Override to implement custom behavior.

**model_form_converter**
Model form conversion class. Use this to implement custom field conversion logic.

Custom class should be derived from the *flask_admin.contrib.mongoengine.form.CustomModelConverter*.

For example:

```
class MyModelConverter(AdminModelConverter):
    pass


class MyAdminView(ModelView):
    model_form_converter = MyModelConverter
```

alias of `CustomModelConverter`

**object_id_converter**
Mongodb `_id` value conversion function. Default is *bson.ObjectId*. Use this if you are using String, Binary and etc.

For example:

```
class MyModelView(BaseModelView):
    object_id_converter = int
```

or:

```
class MyModelView(BaseModelView):
    object_id_converter = str
```

alias of `ObjectId`

**on_form_prefill**(*form*, *id*)
Perform additional actions to pre-fill the edit form.

Called from edit_view, if the current action is rendering the form rather than receiving client side input, after default pre-filling has been performed.

By default does nothing.

You only need to override this if you have added custom fields that depend on the database contents in a way that Flask-admin can't figure out by itself. Fields that were added by name of a normal column or relationship should work out of the box.

**Parameters**

- **form** – Form instance
- **id** – id of the object that is going to be edited

**on_model_change**(*form*, *model*, *is_created*)
Perform some actions before a model is created or updated.

Called from create_model and update_model in the same transaction (if it has any meaning for a store backend).

By default does nothing.

**Parameters**

- **form** – Form used to create/update model
- **model** – Model that will be created/updated
- **is_created** – Will be set to True if model was created and to False if edited

**on_model_delete**(*model*)
Perform some actions before a model is deleted.

Called from delete_model in the same transaction (if it has any meaning for a store backend).

By default do nothing.

**render**(*template*, ***kwargs*)
Render template

**Parameters**

- **template** – Template path to render
- **kwargs** – Template arguments

**scaffold_filters**(*name*)
Return filter object(s) for the field

**Parameters** **name** – Either field name or field instance

**scaffold_form**()
Create form from the model.

**scaffold_list_columns**()
Scaffold list columns

**scaffold_list_form**(*widget=None*, *validators=None*)
Create form for the *index_view* using only the columns from *self.column_editable_list*.

**Parameters**

- **widget** – WTForms widget class. Defaults to *XEditableWidget*.

- **validators** – *form_args* dict with only validators {'name': {'validators': [required()]}}

**scaffold_sortable_columns**()
Return a dictionary of sortable columns (name, field)

**update_model**(*form*, *model*)
Update model helper

**Parameters**

- **form** – Form instance
- **model** – Model instance to update

**validate_form**(*form*)
Validate the form on submit.

**Parameters** **form** – Form to validate

## flask_admin.contrib.mongoengine.fields

**class ModelFormField**(*model*, *view*, *form_class*, *form_opts=None*, ***kwargs*)
Customized ModelFormField for MongoEngine EmbeddedDocuments.

**class MongoFileField**(*label=None*, *validators=None*, ***kwargs*)
GridFS file field.

**class MongoImageField**(*label=None*, *validators=None*, ***kwargs*)
GridFS image field.

## flask_admin.contrib.peewee

Peewee model backend implementation.

**class ModelView**(*model*, *name=None*, *category=None*, *endpoint=None*, *url=None*, *static_folder=None*, *menu_class_name=None*, *menu_icon_type=None*, *menu_icon_value=None*)
Class inherits configuration options from *BaseModelView* and they're not displayed here.

**column_filters = None**
Collection of the column filters.

Can contain either field names or instances of `flask_admin.contrib.peewee.filters.BasePeeweeFilter` classes.

Filters will be grouped by name when displayed in the drop-down.

For example:

```
class MyModelView(BaseModelView):
    column_filters = ('user', 'email')
```

or:

```
from flask_admin.contrib.peewee.filters import BooleanEqualFilter

class MyModelView(BaseModelView):
    column_filters = (BooleanEqualFilter(column=User.name, name='Name'),)
```

or:

```
from flask_admin.contrib.peewee.filters import BasePeeweeFilter

class FilterLastNameBrown(BasePeeweeFilter):
    def apply(self, query, value):
        if value == '1':
            return query.filter(self.column == "Brown")
        else:
            return query.filter(self.column != "Brown")

    def operation(self):
        return 'is Brown'

class MyModelView(BaseModelView):
    column_filters = [
        FilterLastNameBrown(
            column=User.last_name, name='Last Name',
            options=(('1', 'Yes'), ('0', 'No'))
        )
    ]
```

**filter_converter = <flask_admin.contrib.peewee.filters.FilterConverter object>**
Field to filter converter.

Override this attribute to use non-default converter.

**model_form_converter = <class 'flask_admin.contrib.peewee.form.CustomModelConverter'>**
Model form conversion class. Use this to implement custom field conversion logic.

For example:

```
class MyModelConverter(AdminModelConverter):
    pass


class MyAdminView(ModelView):
    model_form_converter = MyModelConverter
```

**inline_model_form_converter = <class 'flask_admin.contrib.peewee.form.InlineModelConverter'>**
Inline model conversion class. If you need some kind of post-processing for inline forms, you can customize behavior by doing something like this:

```
class MyInlineModelConverter(AdminModelConverter):
    def post_process(self, form_class, info):
        form_class.value = TextField('value')
        return form_class

class MyAdminView(ModelView):
    inline_model_form_converter = MyInlineModelConverter
```

**fast_mass_delete = False**
If set to *False* and user deletes more than one model using actions, all models will be read from the database and then deleted one by one giving Peewee chance to manually cleanup any dependencies (many-to-many relationships, etc).

If set to True, will run DELETE statement which is somewhat faster, but might leave corrupted data if you forget to configure DELETE CASCADE for your model.

**inline_models = None**

Inline related-model editing for models with parent to child relation.

Accept enumerable with one of the values:

1.Child model class:

```
class MyModelView(ModelView):
    inline_models = (Post,)
```

2.Child model class and additional options:

```
class MyModelView(ModelView):
    inline_models = [(Post, dict(form_columns=['title']))]
```

3.Django-like `InlineFormAdmin` class instance:

```
from flask_admin.model.form import InlineFormAdmin

class MyInlineModelForm(InlineFormAdmin):
    form_columns = ('title', 'date')

class MyModelView(ModelView):
    inline_models = (MyInlineModelForm(MyInlineModel),)
```

You can customize generated field name by:

1.Using *form_name* property as option:

**class MyModelView(ModelView):** inline_models = ((Post, dict(form_label='Hello')))

2.Using field's related_name:

**class Model1(Base):** # ... pass

**class Model2(Base):** # ... model1 = ForeignKeyField(related_name="model_twos")

**class MyModel1View(Base):** inline_models = (Model2,) column_labels = {'model_ones': 'Hello'}

**action_form**(*obj=None*)

Instantiate model action form and return it.

Override to implement custom behavior.

**action_view**(**args*, ***kwargs*)

Mass-model action view.

**after_model_change**(*form*, *model*, *is_created*)

Perform some actions after a model was created or updated and committed to the database.

Called from create_model after successful database commit.

By default does nothing.

**Parameters**

- **form** – Form used to create/update model
- **model** – Model that was created/updated
- **is_created** – True if model was created, False if model was updated

**after_model_delete**(*model*)
Perform some actions after a model was deleted and committed to the database.

Called from delete_model after successful database commit (if it has any meaning for a store backend).

By default does nothing.

**Parameters** **model** – Model that was deleted

**ajax_update**(**args*, ***kwargs*)
Edits a single column of a record in list view.

**create_blueprint**(*admin*)
Create Flask blueprint.

**create_form**(*obj=None*)
Instantiate model creation form and return it.

Override to implement custom behavior.

**create_view**(**args*, ***kwargs*)
Create model view

**delete_form**()
Instantiate model delete form and return it.

Override to implement custom behavior.

The delete form originally used a GET request, so delete_form accepts both GET and POST request for backwards compatibility.

**delete_view**(**args*, ***kwargs*)
Delete model view. Only POST method is allowed.

**details_view**(**args*, ***kwargs*)
Details model view

**edit_form**(*obj=None*)
Instantiate model editing form and return it.

Override to implement custom behavior.

**edit_view**(**args*, ***kwargs*)
Edit model view

**get_action_form**()
Create form class for a model action.

Override to implement customized behavior.

**get_actions_list**()
Return a list and a dictionary of allowed actions.

**get_column_name**(*field*)
Return a human-readable column name.

**Parameters** **field** – Model field name.

**get_column_names**(*only_columns*, *excluded_columns*)
Returns a list of tuples with the model field name and formatted field name.

**Parameters**

- **only_columns** – List of columns to include in the results. If not set, *scaffold_list_columns* will generate the list from the model.

- **excluded_columns** – List of columns to exclude from the results if *only_columns* is not set.

**get_create_form**()
Create form class for model creation view.

Override to implement customized behavior.

**get_delete_form**()
Create form class for model delete view.

Override to implement customized behavior.

**get_details_columns**()
Uses *get_column_names* to get a list of tuples with the model field name and formatted name for the columns in *column_details_list* and not in *column_details_exclude_list*. If *column_details_list* is not set, the columns from *scaffold_list_columns* will be used.

**get_edit_form**()
Create form class for model editing view.

Override to implement customized behavior.

**get_export_columns**()
Uses *get_column_names* to get a list of tuples with the model field name and formatted name for the columns in *column_export_list* and not in *column_export_exclude_list*. If *column_export_list* is not set, it will attempt to use the columns from *column_list* or finally the columns from *scaffold_list_columns* will be used.

**get_export_name**(*export_type='csv'*)

**Returns** The exported csv file name.

**get_export_value**(*model*, *name*)
Returns the value to be displayed in export. Allows export to use different (non HTML) formatters.

**Parameters**

- **model** – Model instance
- **name** – Field name

**get_filter_arg**(*index*, *flt*)
Given a filter *flt*, return a unique name for that filter in this view.

Does not include the *flt[n]_* portion of the filter name.

**Parameters**

- **index** – Filter index in _filters array
- **flt** – Filter instance

**get_filters**()
Return a list of filter objects.

If your model backend implementation does not support filters, override this method and return *None*.

**get_form**()
Get form class.

If `self.form` is set, will return it and will call `self.scaffold_form` otherwise.

Override to implement customized behavior.

**get_list**(*page*, *sort_column*, *sort_desc*, *search*, *filters*, *execute=True*, *page_size=None*)
Return records from the database.

**Parameters**

- **page** – Page number
- **sort_column** – Sort column name
- **sort_desc** – Descending or ascending sort
- **search** – Search query
- **filters** – List of filter tuples
- **execute** – Execute query immediately? Default is *True*
- **page_size** – Number of results. Defaults to ModelView's page_size. Can be overriden to change the page_size limit. Removing the page_size limit requires setting page_size to 0 or False.

**get_list_columns**()
Uses *get_column_names* to get a list of tuples with the model field name and formatted name for the columns in *column_list* and not in *column_exclude_list*. If *column_list* is not set, the columns from *scaffold_list_columns* will be used.

**get_list_form**()
Get form class for the editable list view.

Uses only validators from *form_args* to build the form class.

Allows overriding the editable list view field/widget. For example:

```
from flask_admin.model.widgets import XEditableWidget

class CustomWidget(XEditableWidget):
    def get_kwargs(self, subfield, kwargs):
        if subfield.type == 'TextAreaField':
            kwargs['data-type'] = 'textarea'
            kwargs['data-rows'] = '20'
        # elif: kwargs for other fields

        return kwargs

class MyModelView(BaseModelView):
    def get_list_form(self):
        return self.scaffold_list_form(widget=CustomWidget)
```

**get_list_row_actions**()
Return list of row action objects, each is instance of `BaseListRowAction`

**get_list_value**(*context*, *model*, *name*)
Returns the value to be displayed in the list view

**Parameters**

- **context** – `jinja2.runtime.Context`
- **model** – Model instance
- **name** – Field name

**get_save_return_url**(*model*, *is_created=False*)
Return url where user is redirected after successful form save.

**Parameters**

- **model** – Saved object
- **is_created** – Whether new object was created or existing one was updated

For example, redirect use to object details view after form save:

```
class MyModelView(ModelView):
    can_view_details = True

    def get_save_return_url(self, model, is_created):
        return self.get_url('.details_view', id=model.id)
```

**get_sortable_columns**()
Returns a dictionary of the sortable columns. Key is a model field name and value is sort column (for example - attribute).

If *column_sortable_list* is set, will use it. Otherwise, will call *scaffold_sortable_columns* to get them from the model.

**get_url**(*endpoint*, ***kwargs*)
Generate URL for the endpoint. If you want to customize URL generation logic (persist some query string argument, for example), this is right place to do it.

**Parameters**

- **endpoint** – Flask endpoint name
- **kwargs** – Arguments for *url_for*

**handle_action**(*return_view=None*)
Handle action request.

**Parameters** **return_view** – Name of the view to return to after the request. If not provided, will return user to the return url in the form or the list view.

**handle_filter**(*filter*)
Postprocess (add joins, etc) for a filter.

**Parameters** **filter** – Filter object to postprocess

**inaccessible_callback**(*name*, ***kwargs*)
Handle the response to inaccessible views.

By default, it throw HTTP 403 error. Override this method to customize the behaviour.

**index_view**(**args*, ***kwargs*)
List view

**init_actions**()
Initialize list of actions for the current administrative view.

**is_accessible**()
Override this method to add permission checks.

Flask-Admin does not make any assumptions about the authentication system used in your application, so it is up to you to implement it.

By default, it will allow access for everyone.

**is_editable**(*name*)
Verify if column is editable.

**Parameters** **name** – Column name.

**is_sortable**(*name*)
Verify if column is sortable.

Not case-sensitive.

**Parameters** **name** – Column name.

**is_visible**()
Override this method if you want dynamically hide or show administrative views from Flask-Admin menu structure

By default, item is visible in menu.

Please note that item should be both visible and accessible to be displayed in menu.

**list_form**(*obj=None*)
Instantiate model editing form for list view and return it.

Override to implement custom behavior.

**on_form_prefill**(*form*, *id*)
Perform additional actions to pre-fill the edit form.

Called from edit_view, if the current action is rendering the form rather than receiving client side input, after default pre-filling has been performed.

By default does nothing.

You only need to override this if you have added custom fields that depend on the database contents in a way that Flask-admin can't figure out by itself. Fields that were added by name of a normal column or relationship should work out of the box.

**Parameters**

- **form** – Form instance
- **id** – id of the object that is going to be edited

**on_model_change**(*form*, *model*, *is_created*)
Perform some actions before a model is created or updated.

Called from create_model and update_model in the same transaction (if it has any meaning for a store backend).

By default does nothing.

**Parameters**

- **form** – Form used to create/update model
- **model** – Model that will be created/updated
- **is_created** – Will be set to True if model was created and to False if edited

**on_model_delete**(*model*)
Perform some actions before a model is deleted.

Called from delete_model in the same transaction (if it has any meaning for a store backend).

By default do nothing.

**render**(*template*, ***kwargs*)
Render template

**Parameters**

- **template** – Template path to render

- **kwargs** – Template arguments

**scaffold_list_form**(*widget=None*, *validators=None*)
Create form for the *index_view* using only the columns from *self.column_editable_list*.

**Parameters**

- **widget** – WTForms widget class. Defaults to *XEditableWidget*.
- **validators** – *form_args* dict with only validators {'name': {'validators': [required()]}}

**validate_form**(*form*)
Validate the form on submit.

**Parameters** **form** – Form to validate

## flask_admin.contrib.pymongo

PyMongo model backend implementation.

**class ModelView**(*coll*, *name=None*, *category=None*, *endpoint=None*, *url=None*, *menu_class_name=None*, *menu_icon_type=None*, *menu_icon_value=None*)
MongoEngine model scaffolding.

Class inherits configuration options from *BaseModelView* and they're not displayed here.

**column_filters = None**
Collection of the column filters.

Should contain instances of `flask_admin.contrib.pymongo.filters.BasePyMongoFilter` classes.

Filters will be grouped by name when displayed in the drop-down.

For example:

```
from flask_admin.contrib.pymongo.filters import BooleanEqualFilter

class MyModelView(BaseModelView):
    column_filters = (BooleanEqualFilter(column=User.name, name='Name'),)
```

or:

```
from flask_admin.contrib.pymongo.filters import BasePyMongoFilter

class FilterLastNameBrown(BasePyMongoFilter):
    def apply(self, query, value):
        if value == '1':
            return query.filter(self.column == "Brown")
        else:
            return query.filter(self.column != "Brown")

    def operation(self):
        return 'is Brown'

class MyModelView(BaseModelView):
    column_filters = [
        FilterLastNameBrown(
            column=User.last_name, name='Last Name',
```

```
            options=(('1', 'Yes'), ('0', 'No'))
        )
    ]
```

**action_form**(*obj=None*)
Instantiate model action form and return it.

Override to implement custom behavior.

**action_view**(**args*, ***kwargs*)
Mass-model action view.

**after_model_change**(*form*, *model*, *is_created*)
Perform some actions after a model was created or updated and committed to the database.

Called from create_model after successful database commit.

By default does nothing.

**Parameters**

- **form** – Form used to create/update model
- **model** – Model that was created/updated
- **is_created** – True if model was created, False if model was updated

**after_model_delete**(*model*)
Perform some actions after a model was deleted and committed to the database.

Called from delete_model after successful database commit (if it has any meaning for a store backend).

By default does nothing.

**Parameters** **model** – Model that was deleted

**ajax_update**(**args*, ***kwargs*)
Edits a single column of a record in list view.

**create_blueprint**(*admin*)
Create Flask blueprint.

**create_form**(*obj=None*)
Instantiate model creation form and return it.

Override to implement custom behavior.

**create_model**(*form*)
Create model helper

**Parameters** **form** – Form instance

**create_view**(**args*, ***kwargs*)
Create model view

**delete_form**()
Instantiate model delete form and return it.

Override to implement custom behavior.

The delete form originally used a GET request, so delete_form accepts both GET and POST request for backwards compatibility.

**delete_model**(*model*)
Delete model helper

**Parameters** **model** – Model instance

**delete_view**(**args*, ***kwargs*)
Delete model view. Only POST method is allowed.

**details_view**(**args*, ***kwargs*)
Details model view

**edit_form**(*obj*)
Create edit form from the MongoDB document

**edit_view**(**args*, ***kwargs*)
Edit model view

**get_action_form**()
Create form class for a model action.

Override to implement customized behavior.

**get_actions_list**()
Return a list and a dictionary of allowed actions.

**get_column_name**(*field*)
Return a human-readable column name.

**Parameters** **field** – Model field name.

**get_column_names**(*only_columns*, *excluded_columns*)
Returns a list of tuples with the model field name and formatted field name.

**Parameters**

- **only_columns** – List of columns to include in the results. If not set, *scaffold_list_columns* will generate the list from the model.
- **excluded_columns** – List of columns to exclude from the results if *only_columns* is not set.

**get_create_form**()
Create form class for model creation view.

Override to implement customized behavior.

**get_delete_form**()
Create form class for model delete view.

Override to implement customized behavior.

**get_details_columns**()
Uses *get_column_names* to get a list of tuples with the model field name and formatted name for the columns in *column_details_list* and not in *column_details_exclude_list*. If *column_details_list* is not set, the columns from *scaffold_list_columns* will be used.

**get_edit_form**()
Create form class for model editing view.

Override to implement customized behavior.

**get_export_columns**()
Uses *get_column_names* to get a list of tuples with the model field name and formatted name for the columns in *column_export_list* and not in *column_export_exclude_list*. If *column_export_list* is not set, it will attempt to use the columns from *column_list* or finally the columns from *scaffold_list_columns* will be used.

**get_export_name**(*export_type='csv'*)

**Returns** The exported csv file name.

**get_export_value**(*model*, *name*)
Returns the value to be displayed in export. Allows export to use different (non HTML) formatters.

**Parameters**

- **model** – Model instance
- **name** – Field name

**get_filter_arg**(*index*, *flt*)
Given a filter *flt*, return a unique name for that filter in this view.

Does not include the *flt[n]_* portion of the filter name.

**Parameters**

- **index** – Filter index in _filters array
- **flt** – Filter instance

**get_filters**()
Return a list of filter objects.

If your model backend implementation does not support filters, override this method and return *None*.

**get_form**()
Get form class.

If `self.form` is set, will return it and will call `self.scaffold_form` otherwise.

Override to implement customized behavior.

**get_list**(*page*, *sort_column*, *sort_desc*, *search*, *filters*, *execute=True*, *page_size=None*)
Get list of objects from MongoEngine

**Parameters**

- **page** – Page number
- **sort_column** – Sort column
- **sort_desc** – Sort descending
- **search** – Search criteria
- **filters** – List of applied fiters
- **execute** – Run query immediately or not
- **page_size** – Number of results. Defaults to ModelView's page_size. Can be overriden to change the page_size limit. Removing the page_size limit requires setting page_size to 0 or False.

**get_list_columns**()
Uses *get_column_names* to get a list of tuples with the model field name and formatted name for the columns in *column_list* and not in *column_exclude_list*. If *column_list* is not set, the columns from *scaffold_list_columns* will be used.

**get_list_form**()
Get form class for the editable list view.

Uses only validators from *form_args* to build the form class.

Allows overriding the editable list view field/widget. For example:

```
from flask_admin.model.widgets import XEditableWidget

class CustomWidget(XEditableWidget):
    def get_kwargs(self, subfield, kwargs):
        if subfield.type == 'TextAreaField':
            kwargs['data-type'] = 'textarea'
            kwargs['data-rows'] = '20'
        # elif: kwargs for other fields

        return kwargs

class MyModelView(BaseModelView):
    def get_list_form(self):
        return self.scaffold_list_form(widget=CustomWidget)
```

**get_list_row_actions**()
Return list of row action objects, each is instance of `BaseListRowAction`

**get_list_value**(*context*, *model*, *name*)
Returns the value to be displayed in the list view

**Parameters**

- **context** – `jinja2.runtime.Context`
- **model** – Model instance
- **name** – Field name

**get_one**(*id*)
Return single model instance by ID

**Parameters** **id** – Model ID

**get_pk_value**(*model*)
Return primary key value from the model instance

**Parameters** **model** – Model instance

**get_save_return_url**(*model*, *is_created=False*)
Return url where user is redirected after successful form save.

**Parameters**

- **model** – Saved object
- **is_created** – Whether new object was created or existing one was updated

For example, redirect use to object details view after form save:

```
class MyModelView(ModelView):
    can_view_details = True

    def get_save_return_url(self, model, is_created):
        return self.get_url('.details_view', id=model.id)
```

**get_sortable_columns**()
Returns a dictionary of the sortable columns. Key is a model field name and value is sort column (for example - attribute).

If *column_sortable_list* is set, will use it. Otherwise, will call *scaffold_sortable_columns* to get them from the model.

**get_url**(*endpoint*, ***kwargs*)
Generate URL for the endpoint. If you want to customize URL generation logic (persist some query string argument, for example), this is right place to do it.

**Parameters**

- **endpoint** – Flask endpoint name
- **kwargs** – Arguments for *url_for*

**handle_action**(*return_view=None*)
Handle action request.

**Parameters** **return_view** – Name of the view to return to after the request. If not provided, will return user to the return url in the form or the list view.

**handle_filter**(*filter*)
Postprocess (add joins, etc) for a filter.

**Parameters** **filter** – Filter object to postprocess

**inaccessible_callback**(*name*, ***kwargs*)
Handle the response to inaccessible views.

By default, it throw HTTP 403 error. Override this method to customize the behaviour.

**index_view**(**args*, ***kwargs*)
List view

**init_actions**()
Initialize list of actions for the current administrative view.

**init_search**()
Init search

**is_accessible**()
Override this method to add permission checks.

Flask-Admin does not make any assumptions about the authentication system used in your application, so it is up to you to implement it.

By default, it will allow access for everyone.

**is_editable**(*name*)
Verify if column is editable.

**Parameters** **name** – Column name.

**is_sortable**(*name*)
Verify if column is sortable.

Not case-sensitive.

**Parameters** **name** – Column name.

**is_valid_filter**(*filter*)
Validate if it is valid MongoEngine filter

**Parameters** **filter** – Filter object

**is_visible**()
Override this method if you want dynamically hide or show administrative views from Flask-Admin menu structure

By default, item is visible in menu.

Please note that item should be both visible and accessible to be displayed in menu.

**list_form**(*obj=None*)
Instantiate model editing form for list view and return it.

Override to implement custom behavior.

**on_form_prefill**(*form*, *id*)
Perform additional actions to pre-fill the edit form.

Called from edit_view, if the current action is rendering the form rather than receiving client side input, after default pre-filling has been performed.

By default does nothing.

You only need to override this if you have added custom fields that depend on the database contents in a way that Flask-admin can't figure out by itself. Fields that were added by name of a normal column or relationship should work out of the box.

**Parameters**

- **form** – Form instance
- **id** – id of the object that is going to be edited

**on_model_change**(*form*, *model*, *is_created*)
Perform some actions before a model is created or updated.

Called from create_model and update_model in the same transaction (if it has any meaning for a store backend).

By default does nothing.

**Parameters**

- **form** – Form used to create/update model
- **model** – Model that will be created/updated
- **is_created** – Will be set to True if model was created and to False if edited

**on_model_delete**(*model*)
Perform some actions before a model is deleted.

Called from delete_model in the same transaction (if it has any meaning for a store backend).

By default do nothing.

**render**(*template*, ***kwargs*)
Render template

**Parameters**

- **template** – Template path to render
- **kwargs** – Template arguments

**scaffold_filters**(*attr*)
Return filter object(s) for the field

**Parameters** **name** – Either field name or field instance

**scaffold_list_columns**()
Scaffold list columns

**scaffold_list_form**(*widget=None*, *validators=None*)
Create form for the *index_view* using only the columns from *self.column_editable_list*.

**Parameters**

- **widget** – WTForms widget class. Defaults to *XEditableWidget*.
- **validators** – *form_args* dict with only validators {'name': {'validators': [DataRequired()]}}

Must be implemented in the child class.

**scaffold_sortable_columns**()
Return sortable columns dictionary (name, field)

**update_model**(*form*, *model*)
Update model helper

**Parameters**

- **form** – Form instance
- **model** – Model instance to update

**validate_form**(*form*)
Validate the form on submit.

**Parameters** **form** – Form to validate

## flask_admin.contrib.fileadmin

**class FileAdmin**(*base_path*, **args*, ***kwargs*)
Simple file-management interface.

**Parameters**

- **base_path** – Path to the directory which will be managed
- **base_url** – Optional base URL for the directory. Will be used to generate static links to the files. If not defined, a route will be created to serve uploaded files.

Sample usage:

```
import os.path as op

from flask_admin import Admin
from flask_admin.contrib.fileadmin import FileAdmin

admin = Admin()

path = op.join(op.dirname(__file__), 'static')
admin.add_view(FileAdmin(path, '/static/', name='Static Files'))
```

**can_upload = True**

**can_delete = True**

**can_delete_dirs = True**

**can_mkdir = True**

**can_rename = True**

**allowed_extensions = None**

**editable_extensions = ()**

**list_template = 'admin/file/list.html'**

**upload_template = 'admin/file/form.html'**

**mkdir_template = 'admin/file/form.html'**

**rename_template = 'admin/file/form.html'**

**edit_template = 'admin/file/form.html'**

**delete**(**args*, ***kwargs*)
Delete view method

**download**(**args*, ***kwargs*)
Download view method.

**Parameters** **path** – File path.

**edit**(**args*, ***kwargs*)
Edit view method

**index_view**(**args*, ***kwargs*)
Index view method

**Parameters** **path** – Optional directory path. If not provided, will use the base directory

**mkdir**(**args*, ***kwargs*)
Directory creation view method

**Parameters** **path** – Optional directory path. If not provided, will use the base directory

**rename**(**args*, ***kwargs*)
Rename view method

**upload**(**args*, ***kwargs*)
Upload view method

**Parameters** **path** – Optional directory path. If not provided, will use the base directory

## flask_admin.model.template

**macro**(*name*)
Jinja2 macro list column formatter.

**Parameters** **name** – Macro name in the current template

CHAPTER 5

# Changelog

## 1.5.0

- Fixed CSRF generation logic for multi-process deployments
- Added WTForms >= 3.0 support
- Flask-Admin would not recursively save inline models, allowing arbitrary nesting
- Added configuration properties that allow injection of additional CSS and JS dependencies into templates without overriding them
- SQLAlchemy backend - Updated hybrid property detection using new SQLAlchemy APIs - Added support for association proxies - Added support for remote hybrid properties filters - Added support for ARRAY column type
- Localization-related fixes
- MongoEngine backend is now properly formats model labels
- Improved Google App Engine support: - Added TextProperty, KeyProperty and SelectField support - Added support for form_args, excluded_columns, page_size and after_model_update
- Fixed URL generation with localized named filters
- FileAdmin has Bootstrap 2 support now
- Geoalchemy fixes - Use Google Places (by default) for place search
- Updated translations
- Bug fixes

## 1.4.2

- Small bug fix release. Fixes regression that prevented usage of "virtual" columns with a custom formatter.

## 1.4.1

- Official Python 3.5 support
- Customizable row actions
- Tablib support (exporting to XLS, XLSX, CSV, etc)
- Updated external dependencies (jQuery, x-editable, etc)
- Added settings that allows exceptions to be raised on view errors
- Bug fixes

## 1.4.0

- Updated and reworked documentation
- FileAdmin went through minor refactoring and now supports remote file systems. Comes with the new, optional, AWS S3 file management interface
- Configurable CSV export for model views
- Added overridable URL generation logic. Allows using custom URLs with parameters for administrative views
- Added column_display_actions to ModelView control visibility of the action column without overriding the template
- Added support for the latest MongoEngine
- New SecureForm base class for easier CSRF validation
- Lots of translation-related fixes and updated translations
- Bug fixes

## 1.3.0

- New feature: Edit models in the list view in a popup
- New feature: Read-only model details view
- Fixed XSS in column_editable_list values
- Improved navigation consistency in model create and edit views
- Ability to choose page size in model list view
- Updated client-side dependencies (jQuery, Select2, etc)
- Updated documentation and examples
- Updated translations
- Bug fixes

CHAPTER 6

# Support

Python 2.6 - 2.7 and 3.3 - 3.4.

CHAPTER 7

# Indices And Tables

- genindex
- modindex
- search

# Python Module Index

## f

flask_admin.actions, 50
flask_admin.base, 27
flask_admin.contrib.fileadmin, 86
flask_admin.contrib.mongoengine, 61
flask_admin.contrib.mongoengine.fields, 71
flask_admin.contrib.peewee, 71
flask_admin.contrib.pymongo, 79
flask_admin.contrib.sqla, 51
flask_admin.form, 45
flask_admin.form.fields, 47
flask_admin.form.rules, 45
flask_admin.form.upload, 47
flask_admin.helpers, 30
flask_admin.model, 31
flask_admin.model.template, 87
flask_admin.tools, 49

# Index

## Symbols


__init__() (Container method), 46
__init__() (Field method), 46
__init__() (FieldSet method), 47
__init__() (FileUploadField method), 47
__init__() (Header method), 46
__init__() (ImageUploadField method), 48
__init__() (Macro method), 46
__init__() (NestedRule method), 45
__init__() (Text method), 46


## A


action() (in module flask_admin.actions), 50
action_disallowed_list (BaseModelView attribute), 37
action_form() (BaseModelView method), 37
action_form() (ModelView method), 54, 64, 73, 80
action_view() (BaseModelView method), 37
action_view() (ModelView method), 54, 64, 73, 80
ActionsMixin (class in flask_admin.actions), 50
add_link() (Admin method), 29
add_links() (Admin method), 29
add_menu_item() (Admin method), 29
add_view() (Admin method), 29
add_views() (Admin method), 29
Admin (class in flask_admin.base), 29
AdminIndexView (class in flask_admin.base), 28
after_model_change() (BaseModelView method), 37
after_model_change() (ModelView method), 54, 64, 73, 80
after_model_delete() (BaseModelView method), 37
after_model_delete() (ModelView method), 55, 64, 73, 80
ajax_update() (BaseModelView method), 37
ajax_update() (ModelView method), 55, 64, 74, 80
allowed_extensions (FileAdmin attribute), 86
allowed_search_types (ModelView attribute), 62, 64


## B


BaseForm (class in flask_admin.form), 45
BaseModelView (class in flask_admin.model), 31
BaseRule (class in flask_admin.form.rules), 45
BaseView (class in flask_admin.base), 27


## C


can_create (BaseModelView attribute), 31
can_delete (BaseModelView attribute), 31
can_delete (FileAdmin attribute), 86
can_delete_dirs (FileAdmin attribute), 86
can_edit (BaseModelView attribute), 31
can_export (BaseModelView attribute), 38
can_mkdir (FileAdmin attribute), 86
can_rename (FileAdmin attribute), 86
can_set_page_size (BaseModelView attribute), 37
can_upload (FileAdmin attribute), 86
can_view_details (BaseModelView attribute), 38
column_auto_select_related (ModelView attribute), 51
column_choices (BaseModelView attribute), 34
column_default_sort (BaseModelView attribute), 34
column_descriptions (BaseModelView attribute), 32
column_details_exclude_list (BaseModelView attribute), 38
column_details_list (BaseModelView attribute), 38
column_display_actions (BaseModelView attribute), 38
column_display_all_relations (ModelView attribute), 55
column_display_pk (BaseModelView attribute), 33
column_editable_list (BaseModelView attribute), 38
column_exclude_list (BaseModelView attribute), 31
column_export_exclude_list (BaseModelView attribute), 38
column_export_list (BaseModelView attribute), 38
column_extra_row_actions (BaseModelView attribute), 38
column_filters (BaseModelView attribute), 34
column_filters (ModelView attribute), 52, 61, 71, 79
column_formatters (BaseModelView attribute), 32
column_formatters_export (BaseModelView attribute), 38
column_labels (BaseModelView attribute), 32
column_list (BaseModelView attribute), 31
column_searchable_list (BaseModelView attribute), 33

column_searchable_list (ModelView attribute), 51
column_select_related_list (ModelView attribute), 51
column_sortable_list (BaseModelView attribute), 33
column_type_formatters (BaseModelView attribute), 32
column_type_formatters (ModelView attribute), 62
column_type_formatters_export (BaseModelView attribute), 39
Container (class in flask_admin.form.rules), 46
create_blueprint() (BaseView method), 28
create_blueprint() (ModelView method), 55, 64, 74, 80
create_form() (BaseModelView method), 39
create_form() (ModelView method), 55, 64, 74, 80
create_modal (BaseModelView attribute), 39
create_modal_template (BaseModelView attribute), 39
create_model() (BaseModelView method), 39
create_model() (ModelView method), 55, 64, 80
create_template (BaseModelView attribute), 31
create_view() (BaseModelView method), 39
create_view() (ModelView method), 55, 65, 74, 80

## D

delete() (FileAdmin method), 87
delete_form() (BaseModelView method), 39
delete_form() (ModelView method), 55, 65, 74, 80
delete_model() (BaseModelView method), 39
delete_model() (ModelView method), 55, 65, 80
delete_view() (BaseModelView method), 39
delete_view() (ModelView method), 55, 65, 74, 81
details_modal (BaseModelView attribute), 39
details_modal_template (BaseModelView attribute), 39
details_template (BaseModelView attribute), 39
details_view() (BaseModelView method), 39
details_view() (ModelView method), 55, 65, 74, 81
download() (FileAdmin method), 87

## E

edit() (FileAdmin method), 87
edit_form() (BaseModelView method), 39
edit_form() (ModelView method), 55, 65, 74, 81
edit_modal (BaseModelView attribute), 40
edit_modal_template (BaseModelView attribute), 40
edit_template (BaseModelView attribute), 31
edit_template (FileAdmin attribute), 87
edit_view() (BaseModelView method), 40
edit_view() (ModelView method), 55, 65, 74, 81
editable_extensions (FileAdmin attribute), 86
export_max_rows (BaseModelView attribute), 40
export_types (BaseModelView attribute), 40
expose() (in module flask_admin.base), 27
expose_plugview() (in module flask_admin.base), 27

## F

fast_mass_delete (ModelView attribute), 53, 72
Field (class in flask_admin.form.rules), 46
FieldSet (class in flask_admin.form.rules), 47
FileAdmin (class in flask_admin.contrib.fileadmin), 86
FileUploadField (class in flask_admin.form.upload), 47
FileUploadInput (class in flask_admin.form.upload), 49
filter_converter (ModelView attribute), 52, 62, 72
flask_admin.actions (module), 50
flask_admin.base (module), 27
flask_admin.contrib.fileadmin (module), 86
flask_admin.contrib.mongoengine (module), 61
flask_admin.contrib.mongoengine.fields (module), 71
flask_admin.contrib.peewee (module), 71
flask_admin.contrib.pymongo (module), 79
flask_admin.contrib.sqla (module), 51
flask_admin.form (module), 45
flask_admin.form.fields (module), 47
flask_admin.form.rules (module), 45
flask_admin.form.upload (module), 47
flask_admin.helpers (module), 30
flask_admin.model (module), 31
flask_admin.model.template (module), 87
flask_admin.tools (module), 49
form (BaseModelView attribute), 34
form_ajax_refs (BaseModelView attribute), 36
form_args (BaseModelView attribute), 35
form_base_class (BaseModelView attribute), 34
form_choices (ModelView attribute), 54
form_columns (BaseModelView attribute), 35
form_create_rules (BaseModelView attribute), 37
form_edit_rules (BaseModelView attribute), 37
form_excluded_columns (BaseModelView attribute), 35
form_extra_fields (BaseModelView attribute), 36
form_optional_types (ModelView attribute), 54
form_overrides (BaseModelView attribute), 35
form_rules (BaseModelView attribute), 40
form_subdocuments (ModelView attribute), 62
form_widget_args (BaseModelView attribute), 35

## G

get_action_form() (BaseModelView method), 40
get_action_form() (ModelView method), 55, 65, 74, 81
get_actions_list() (ActionsMixin method), 50
get_actions_list() (ModelView method), 55, 65, 74, 81
get_column_name() (BaseModelView method), 40
get_column_name() (ModelView method), 56, 65, 74, 81
get_column_names() (BaseModelView method), 40
get_column_names() (ModelView method), 56, 65, 74, 81
get_count_query() (ModelView method), 56
get_create_form() (BaseModelView method), 41
get_create_form() (ModelView method), 56, 65, 75, 81
get_current_view() (in module flask_admin.helpers), 30
get_delete_form() (BaseModelView method), 41
get_delete_form() (ModelView method), 56, 65, 75, 81
get_details_columns() (BaseModelView method), 41

get_details_columns() (ModelView method), 56, 66, 75, 81
get_edit_form() (BaseModelView method), 41
get_edit_form() (ModelView method), 56, 66, 75, 81
get_export_columns() (BaseModelView method), 41
get_export_columns() (ModelView method), 56, 66, 75, 81
get_export_name() (BaseModelView method), 41
get_export_name() (ModelView method), 56, 66, 75, 81
get_export_value() (BaseModelView method), 41
get_export_value() (ModelView method), 56, 66, 75, 82
get_filter_arg() (BaseModelView method), 41
get_filter_arg() (ModelView method), 56, 66, 75, 82
get_filters() (BaseModelView method), 41
get_filters() (ModelView method), 57, 66, 75, 82
get_form() (BaseModelView method), 41
get_form() (ModelView method), 57, 66, 75, 82
get_form_data() (in module flask_admin.helpers), 30
get_list() (BaseModelView method), 42
get_list() (ModelView method), 57, 66, 75, 82
get_list_columns() (BaseModelView method), 42
get_list_columns() (ModelView method), 57, 67, 76, 82
get_list_form() (BaseModelView method), 42
get_list_form() (ModelView method), 57, 67, 76, 82
get_list_row_actions() (BaseModelView method), 42
get_list_row_actions() (ModelView method), 58, 67, 76, 83
get_list_value() (BaseModelView method), 42
get_list_value() (ModelView method), 58, 67, 76, 83
get_one() (BaseModelView method), 43
get_one() (ModelView method), 58, 67, 83
get_pk_value() (BaseModelView method), 43
get_pk_value() (ModelView method), 58, 67, 83
get_query() (ModelView method), 58, 67
get_render_ctx() (in module flask_admin.helpers), 30
get_save_return_url() (BaseModelView method), 43
get_save_return_url() (ModelView method), 58, 67, 76, 83
get_sortable_columns() (BaseModelView method), 43
get_sortable_columns() (ModelView method), 58, 68, 77, 83
get_url() (BaseView method), 28
get_url() (ModelView method), 59, 68, 77, 83

## H

handle_action() (ActionsMixin method), 50
handle_action() (ModelView method), 59, 68, 77, 84
handle_filter() (BaseModelView method), 43
handle_filter() (ModelView method), 68, 77, 84
Header (class in flask_admin.form.rules), 46
HTML (class in flask_admin.form.rules), 46

## I

ignore_hidden (ModelView attribute), 59
ImageUploadField (class in flask_admin.form.upload), 48
ImageUploadInput (class in flask_admin.form.upload), 49
import_attribute() (in module flask_admin.tools), 49
import_module() (in module flask_admin.tools), 49
inaccessible_callback() (BaseView method), 28
inaccessible_callback() (ModelView method), 59, 68, 77, 84
index_view() (BaseModelView method), 43
index_view() (FileAdmin method), 87
index_view() (ModelView method), 59, 68, 77, 84
init_actions() (ActionsMixin method), 51
init_actions() (ModelView method), 59, 68, 77, 84
init_app() (Admin method), 30
init_search() (BaseModelView method), 43
init_search() (ModelView method), 59, 68, 84
inline_model_form_converter (ModelView attribute), 53, 72
inline_models (ModelView attribute), 53, 72
is_accessible() (BaseView method), 28
is_accessible() (ModelView method), 59, 68, 77, 84
is_action_allowed() (ActionsMixin method), 51
is_action_allowed() (BaseModelView method), 43
is_editable() (BaseModelView method), 43
is_editable() (ModelView method), 59, 69, 77, 84
is_field_error() (in module flask_admin.helpers), 30
is_form_submitted() (in module flask_admin.helpers), 30
is_required_form_field() (in module flask_admin.helpers), 30
is_sortable() (BaseModelView method), 43
is_sortable() (ModelView method), 59, 69, 77, 84
is_valid_filter() (BaseModelView method), 43
is_valid_filter() (ModelView method), 59, 69, 84
is_visible() (BaseView method), 28
is_visible() (ModelView method), 60, 69, 78, 84

## L

list_form() (BaseModelView method), 44
list_form() (ModelView method), 60, 69, 78, 85
list_template (BaseModelView attribute), 31
list_template (FileAdmin attribute), 86

## M

Macro (class in flask_admin.form.rules), 46
macro() (in module flask_admin.model.template), 87
menu() (Admin method), 30
menu_links() (Admin method), 30
mkdir() (FileAdmin method), 87
mkdir_template (FileAdmin attribute), 87
model_form_converter (ModelView attribute), 53, 62, 69, 72
ModelFormField (class in flask_admin.contrib.mongoengine.fields), 71

ModelView (class in flask_admin.contrib.mongoengine), 61
ModelView (class in flask_admin.contrib.peewee), 71
ModelView (class in flask_admin.contrib.pymongo), 79
ModelView (class in flask_admin.contrib.sqla), 51
module_not_found() (in module flask_admin.tools), 50
MongoFileField (class in flask_admin.contrib.mongoengine.fields), 71
MongoImageField (class in flask_admin.contrib.mongoengine.fields), 71

## N

named_filter_urls (BaseModelView attribute), 44
NestedRule (class in flask_admin.form.rules), 45

## O

object_id_converter (ModelView attribute), 69
on_form_prefill() (BaseModelView method), 44
on_form_prefill() (ModelView method), 60, 69, 78, 85
on_model_change() (BaseModelView method), 44
on_model_change() (ModelView method), 60, 70, 78, 85
on_model_delete() (BaseModelView method), 44
on_model_delete() (ModelView method), 60, 70, 78, 85

## P

page_size (BaseModelView attribute), 37

## R

rec_getattr() (in module flask_admin.tools), 50
rename() (FileAdmin method), 87
rename_template (FileAdmin attribute), 87
render() (BaseView method), 28
render() (ModelView method), 60, 70, 78, 85
resolve_ctx() (in module flask_admin.helpers), 30

## S

scaffold_auto_joins() (ModelView method), 60
scaffold_filters() (BaseModelView method), 44
scaffold_filters() (ModelView method), 61, 70, 85
scaffold_form() (BaseModelView method), 44
scaffold_form() (ModelView method), 61, 70
scaffold_inline_form_models() (ModelView method), 61
scaffold_list_columns() (BaseModelView method), 44
scaffold_list_columns() (ModelView method), 61, 70, 85
scaffold_list_form() (BaseModelView method), 45
scaffold_list_form() (ModelView method), 61, 70, 79, 85
scaffold_pk() (ModelView method), 61
scaffold_sortable_columns() (BaseModelView method), 45
scaffold_sortable_columns() (ModelView method), 61, 71, 86
Select2Field (class in flask_admin.form.fields), 47
Select2TagsField (class in flask_admin.form.fields), 47
simple_list_pager (BaseModelView attribute), 45

## T

Text (class in flask_admin.form.rules), 46
TimeField (class in flask_admin.form.fields), 47

## U

update_model() (BaseModelView method), 45
update_model() (ModelView method), 61, 71, 86
upload() (FileAdmin method), 87
upload_template (FileAdmin attribute), 87

## V

validate_form() (BaseModelView method), 45
validate_form() (ModelView method), 61, 71, 79, 86
validate_form_on_submit() (in module flask_admin.helpers), 30