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 Flask
   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')
```

v: latest

```
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()
```

v: latest

```
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-files-images.

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:

v: latest

```
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

v: latest

Form Rendering Rule	Description
`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 v: latest
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
```

v: latest

```
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`

v: latest

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')
```

v: latest