Flask-AppBuilder

Simple and rapid application development framework, built on top of Flask. Includes detailed security, auto CRUD generation for your models, google charts and much more.

Lots of examples and a live Demo (login has guest/welcome).

Fixes, bugs and contributions

You're welcome to report bugs, propose new features, or even better contribute to this project.

Issues, bugs and new features

Contribute

Contents:

Introduction

The main goal for this project is to provide a simple development framework that handles the main problems any web application or site encounters. It will help you adhere to the DRY (Don't repeat yourself) principle.

Keep in mind that it is possible to develop directly on Flask/Jinja2 for custom pages or flows, that painlessly integrate with the framework.

This framework goes further than an admin scaffolding package. It has builtin presentation and behaviour alternatives, and you can easily build your own. It's highly configurable, and ships with extra goodies.

It's intended to lower errors, bugs and project's time to deliver.

This package has some CSS and JS batteries included:

- Google charts CSS and JS
- BootStrap CSS and JS
- BootsWatch Themes
- Font-Awesome CSS and Fonts

– Includes

- **Database**
  - SQLAlchemy, multiple database support: sqlite, MySQL, ORACLE, MSSQL, DB2 etc.
  - MongoDB, using mongoEngine, still partial support (only normalized).
  - Multiple database connections support (Vertical partitioning).
  - Easy mixin audit to models (created/changed by user, and timestamps).
- **Security**
  - Automatic permissions lookup, based on exposed methods. It will grant all permissions to the Admin Role.
  - Inserts on the Database all the detailed permissions possible on your application.
  - Public (no authentication needed) and Private permissions.
  - Role based permissions.
  - Authentication support for OAuth, OpenID, Database, LDAP and REMOTE_USER environ var.
  - Support for self user registration.
- **Views and Widgets**
  - Automatic menu generation.
  - Automatic CRUD generation.
  - Multiple actions on db records.
  - Big variety of filters for your lists.
  - Various view widgets: lists, master-detail, list of thumbnails etc.
  - Select2, Datepicker, DateTimePicker.
  - Select2 related fields.
  - Google charts with automatic group by or direct values and filters.
- **Forms**
  - Automatic, Add, Edit and Show from Database Models
  - Labels and descriptions for each field.
  - Automatic base validators from model's definition.
  - Custom validators, extra fields, custom filters for related dropdown lists.

    - Image and File support for upload and database field association. It will handle everything for you.
    - Field sets for Form's (Django style).

- i18n
    - Support for multi-language via Babel.

- Bootstrap 3.3.1 CSS and js, with Select2 and DatePicker.
- Font-Awesome icons, for menu icons and actions.
- Bootswatch Themes.

Installation

Installation is straightforward, using the normal python package install. I do advise you to additionally install the base skeleton application so that you can immediately have a running application (without any models yet) and an easy to grow boilerplate.

Checkout installation video on YouTube

Using pip

- **Simple Install**

  You can install the framework simply by:

  ```
  $ pip install flask-appbuilder
  ```

- **Advised Virtual Environment Install**

Virtual env is highly advisable because the more projects you have, the more likely it is that you will be working with different versions of Python itself, or at least different versions of Python libraries. Let's face it: quite often libraries break backwards compatibility, and it's unlikely that any serious application will have zero dependencies. So what do you do if two or more of your projects have conflicting dependencies?

If you are on Mac OS X or Linux, chances are that one of the following two commands will work for you:

```
$ sudo easy_install virtualenv
```

or even better:

```
$ sudo pip install virtualenv
```

One of these will probably install virtualenv on your system. Maybe it's even in your package manager. If you use a debian system (like Ubuntu), try:

```
$ sudo apt-get install python-virtualenv
```

Next create a virtualenv:

```
$ virtualenv venv
New python executable in venv/bin/python
Installing distribute.............done.
$ . venv/bin/activate
(venv)$
```

Now install F.A.B on the virtual env, it will install all the dependencies and these will be isolated from your system's python packages

```
(venv)$ pip install flask-appbuilder
```

Once you have virtualenv installed, use **Flask** the command line tool to create your first app. So create a skeleton application and the first admin user:

```
```

```
(venv)$ flask fab create-app
Your new app name: first_app
Your engine type, SQLAlchemy or MongoEngine [SQLAlchemy]:
Downloaded the skeleton app, good coding!
(venv)$ cd first_app
(venv)$ export FLASK_APP=app
(venv)$ flask fab create-admin
Username [admin]:
User first name [admin]:
User last name [user]:
Email [admin@fab.org]:
Password:
Repeat for confirmation:
```

Note

There are two type of skeletons available you can choose from SQLAlchemy default or MongoEngine for MongoDB. **To use the MongoEngine skeleton you need to install flask-mongoengine extension.**

The framework will immediately insert all possible permissions on the database, these will be associated with the *Admin* role that belongs to the *admin* user you just created. Your ready to run:

```
(venv)$ flask run
```

This will start a web development server

You now have a running development server on http://localhost:8080.

The skeleton application is not actually needed for you to run AppBuilder, but it's a good way to start. This first application is SQLAlchemy based.

Initialization

When starting your application for the first time, all AppBuilder security tables will be created for you. All your models can easily be created too (optionally).

Note

Since version 1.3.0 no admin user is automatically created, you must use **flask fab cli** to do it. There are lot's of other useful options you can use with **flask fab** like reset user's password, list all your users and views, etc.

Installation Requirements

pip installs all the requirements for you.

Flask App Builder depends on

- flask : The web framework, this is what we're extending.
- flask-sqlalchemy : DB access (see SQLAlchemy).
- flask-login : Login, session on flask.
- flask-openid : Open ID authentication.
- flask-wtform : Web forms.
- flask-Babel : For internationalization.

If you plan to use Image processing or upload, you will need to install Pillow:

```
pip install Pillow
```

Python 2 and 3 Compatibility

The framework removed support for python 2 since version 1.13.X

For version 2.1.1, the minimum supported Python version is 3.6.

Command Line Manager

Since version 1.13.1 F.A.B. has a new command line manager, integrated with Flask cli. The old `fabmanager` command line is now deprecated and will be completely removed on 2.2.X. It's very easy to migrate to the new command line, all sub commands are still the same and use the same parameters.

To use the new commands integrated with **Flask cli** you must specify on to import your app. Take a look at Flask docs.:

```
# Using the default skeleton application
$ export FLASK_APP=app

# Using factory app pattern
$ FLASK_APP="app:create_app('config')"
```

FAB creates a **Flask** command group named `fab`, so all commands are issued like:

```
$ flask fab <command> <parameters>
```

To Run your application on development:

```
$ flask run --with-threads --reload
```

Take a quick look to the current possibilities. (The bold ones require app context)

- babel-compile - Babel, Compiles all translations
- babel-extract - Babel, Extracts and updates all messages.
- **create-admin** - Creates an admin user
- **create-user** - Create user with arbitrary role
- create-app - Create a Skeleton application (SQLAlchemy or MongoEngine).
- create-addon - Create a Skeleton AddOn.
- **create-db** - Create all your database objects (SQLAlchemy only)
- **collect-static** - Copies static files from flask-appbuilder to your static folder. Nice to have on certain deploys
- **export-roles** - Export roles with permissions and view menus to JSON file.
- **import-roles** - Import roles with permissions and view menus from JSON file.
- **list-users** - List all users on the database.
- **list-views** - List all registered views.
- **reset-password** - Resets a user's password.
- **security-cleanup** - Cleanup unused permissions from views and roles. Security
- **security-converge** - Converges all security view and permission names from all your roles. Security
- **upgrade-db** - Upgrade your database after F.A.B upgrade.
- **version** - Flask-AppBuilder package version.

Command Line uses the excellent click package, so you can have a detailed help for each command, for instance:

```
$ flask fab create-app --help
Usage: flask fab create-app [OPTIONS]

Create a Skeleton application

Options:
--name TEXT                     Your application name, directory will have
                               this name
--engine [SQLAlchemy|MongoEngine]
                               Write your engine type
--help                          Show this message and exit.
```

create-app - Create new Applications

To create a ready to dev skeleton application, you can use this command for SQLAlchemy engine and MongoEngine (MongoDB). This commands needs an internet connection to **github.com**, because it will download a zip version of the skeleton repos.

create-admin - Create an admin user

Use this to create your first **Admin** user, or additional ones. This admin user can be used to any type of authentication method configured, but *flask fab create-admin* will not check it, so it will always ask for a password (assumes AUTH_DB).

babel-extract - Babel, Extracts and updates all messages.

Use multi **-k** options separated by space to specify how to locate the strings you want to translate. Default values: **lazy_gettext, gettext, _, __**. For example:

```
flask fab babel-extract --target flask_appbuilder/translations/ -k _ -k __
```

create-addon - Create new AddOns

To create a ready to dev skeleton addon. This commands needs an internet connection to **github.com**, because it will download a zip version of the skeleton repos.

collect-static - Collect static files

Use this to copy all static files from flask-appbuilder package to your application static folder. Nice to have on certain deploys, if your web server is serving the static files directly.

upgrade-db - Upgrade your database after F.A.B. upgrade to 1.3.0

Will upgrade your database, necessary if you're already using F.A.B. Users now are able to have multiple roles. Take a look at Version Migration

reset-password - Resets a user's password.

Reset a user's password

Base Configuration

Configuration keys

Use config.py to configure the following parameters. By default it will use SQLLITE DB, and bootstrap's default theme:

Key	Description
SQLALCHEMY_DATABASE_URI	DB connection string (
MONGODB_SETTINGS	DB connection string (
AUTH_TYPE = 0 | 1 | 2 | 3 | 4 or AUTH_TYPE = AUTH_OID, AUTH_DB, AUTH_LDAP, AUTH_REMOTE AUTH_OAUTH	This is the authentication ty • 0 = Open ID • 1 = Database st • 2 = LDAP, use A • 3 = uses web serve REMOTE_US • 4 = USE ONE O
AUTH_USER_REGISTRATION = True|False	Set to True to enable u
AUTH_USERNAME_CI = True|False	Make auth login CI of r
AUTH_USER_REGISTRATION_ROLE	Set role name, to be as
AUTH_USER_REGISTRATION_ROLE_JMESPATH	The JMESPath express
AUTH_ROLES_SYNC_AT_LOGIN	Sets if user's roles are r
AUTH_ROLES_MAPPING	A mapping from LDAP/ See example under AU
AUTH_LDAP_SERVER	define your ldap server AUTH_TYPE = 2 AUTH_LDAP_SERVER For using LDAP over TI
AUTH_LDAP_USE_TLS	Require the use of STA
AUTH_LDAP_BIND_USER	Define the DN for the AUTH_LDAP_BIND_U
AUTH_LDAP_BIND_PASSWORD	Define password for th
AUTH_LDAP_TLS_DEMAND	Demands TLS peer cer
AUTH_LDAP_TLS_CACERTDIR	CA Certificate director
AUTH_LDAP_TLS_CACERTFILE	CA Certificate file to ch
AUTH_LDAP_TLS_CERTFILE	Certificate file for clien

Key	Description
AUTH_LDAP_TLS_KEYFILE	Certificate key file for c
AUTH_LDAP_SEARCH	Use search with self us AUTH_LDAP_SERVER AUTH_LDAP_SEARCH
AUTH_LDAP_SEARCH_FILTER	Filter or limit allowable AUTH_LDAP_SEARCH
AUTH_LDAP_UID_FIELD	if doing an indirect bin AUTH_TYPE = 2 AUTH_LDAP_SERVER AUTH_LDAP_SEARCH AUTH_LDAP_UID_FIE
AUTH_LDAP_GROUP_FIELD	sets the field in the lda AUTH_TYPE = 2 AUTH_LDAP_SERVER AUTH_LDAP_SEARCH AUTH_LDAP_GROUP_ AUTH_ROLES_MAPPING = “cn=User,ou=groups,dc [“User”] }
AUTH_LDAP_FIRSTNAME_FIELD	sets the field in the lda AUTH_TYPE = 2 AUTH_LDAP_SERVER AUTH_LDAP_SEARCH AUTH_LDAP_FIRSTNA

Key	Description
AUTH_LDAP_LASTNAME_FIELD	sets the field in the lda AUTH_TYPE = 2 AUTH_LDAP_SERVER AUTH_LDAP_SEARCH AUTH_LDAP_LASTNA
AUTH_LDAP_EMAIL_FIELD	sets the field in the lda AUTH_TYPE = 2 AUTH_LDAP_SERVER AUTH_LDAP_SEARCH AUTH_LDAP_EMAIL_F
AUTH_LDAP_ALLOW_SELF_SIGNED	Allow LDAP authentica
AUTH_LDAP_APPEND_DOMAIN	Append a domain to al AUTH_LDAP_APPEND And the user can login
AUTH_LDAP_USERNAME_FORMAT	It converts username t username = “userexam AUTH_LDAP_USERNA It authenticates with “f
AUTH_ROLE_ADMIN	Configure the name of
AUTH_ROLE_PUBLIC	Special Role that holds
AUTH_API_LOGIN_ALLOW_MULTIPLE_PROVIDERS True|False	Allow REST API login w
APP_NAME	The name of your appli
APP_THEME	Various themes for you
APP_ICON	path of your applicatio
ADDON_MANAGERS	A list of addon manage
UPLOAD_FOLDER	Files upload folder. Ma

Key	Description
FILE_ALLOWED_EXTENSIONS	Tuple with allower exte
IMG_UPLOAD_FOLDER	Image upload folder. M
IMG_UPLOAD_URL	Image relative URL. Ma
IMG_SIZE	tuple to define default
BABEL_DEFAULT_LOCALE	Babel's default languag
LANGUAGES	A dictionary mapping t
LOGOUT_REDIRECT_URL	The location to redirec
FAB_API_SHOW_STACKTRACE	Sends api stack trace o
FAB_API_MAX_PAGE_SIZE	Sets a limit for FAB Mc
FAB_API_SWAGGER_UI	Enables a Swagger UI v
FAB_API_SWAGGER_TEMPLATE	Path of your custom Sv
FAB_API_ALLOW_JSON_QS	Allow query string para
FAB_UPDATE_PERMS	Enables or disables upc
FAB_SECURITY_MANAGER_CLASS	Declare a new custom
FAB_ADD_SECURITY_API	[Beta] Adds a CRUD RI
FAB_ADD_SECURITY_VIEWS	Enables or disables reg
FAB_ADD_SECURITY_PERMISSION_VIEW	Enables or disables reg
FAB_ADD_SECURITY_VIEW_MENU_VIEW	Enables or disables reg
FAB_ADD_SECURITY_PERMISSION_VIEWS_VIEW	Enables or disables reg
FAB_ADD_OPENAPI_VIEWS	Enables or disables reg
FAB_OPENAPI_SERVERS	Used for setting OpenA
FAB_ROLES	Configure builtin roles
FAB_INDEX_VIEW	Path of your custom In
FAB_MENU	Path of your custom M
FAB_BASE_TEMPLATE	Path of your custom ba
FAB_STATIC_FOLDER	Path to override defaul
FAB_STATIC_URL_PATH	Path to override defaul
FAB_PASSWORD_COMPLEXITY_VALIDATOR	Hook for your own cus
FAB_PASSWORD_COMPLEXITY_ENABLED	Enables the password c

Using config.py

My favorite way, and the one I advise if you are building a medium to large size application is to place all your configuration keys on a config.py file

Next you only have to import them to the Flask app object, like this

```
app = Flask(__name__)
app.config.from_object('config')
```

Take a look at the skeleton config.py

Using JMESPath to map user registration role

If user self registration is enabled and `AUTH_USER_REGISTRATION_ROLE_JMESPATH` is set, it is used as a JMESPath expression to evalate user registration role. The input values is `userinfo` dict, returned by `get_oauth_user_info` function of Security Manager. Usage of JMESPath expressions requires jmespath package to be installed.

In case of Google OAuth, userinfo contains user's email that can be used to map some users as admins and rest of the domain users as read only users. For example, this expression: `contains(['user1@domain.com', 'user2@domain.com'], email) && 'Admin' || 'Viewer'` causes users 1 and 2 to be registered with role `Admin` and rest with the role `Viewer`.

JMESPath expression allow more groups to be evaluated:
`email == 'user1@domain.com' && 'Admin' || (email == 'user2@domain.com' && 'Op' || 'Viewer')`

For more example, see specification.

Base Views

Views are the base concept of F.A.B. They work like a class that represent a concept and present the views and methods to implement it.

Each view is a Flask blueprint that will be created for you automatically by the framework. This is a simple but powerful concept. You will map your methods to routing points, and each method will be registered as a possible security permission if you want.

So your methods will have automatic routing points much like Flask, but this time in a class. Additionally you can have granular security (method access security) that can be associated with a user's role (take a look at Security for more detail).

The views documented on this chapter are the building blocks of F.A.B, but the juicy part is

on the next chapter with ModelView, ChartView and others.

BaseView

All views inherit from this class. Its constructor will register your exposed urls on flask as a Blueprint, as well as all security permissions that need to be defined and protected.

You can use this kind of view to implement your own custom pages, attach it to a menu or link it from any point to your site.

Decorate your url routing methods with **@expose**. Additionally add **@has_access** decorator to tell flask that this is a security protected method.

Using the Flask-AppBuilder-Skeleton (take a look at the Installation chapter). Edit views.py file and add:

```
from flask_appbuilder import AppBuilder, expose, BaseView
from app import appbuilder

class MyView(BaseView):
    route_base = "/myview"

    @expose('/method1/<string:param1>')
    def method1(self, param1):
        # do something with param1
        # and return it
        return param1

    @expose('/method2/<string:param1>')
    def method2(self, param1):
        # do something with param1
        # and render it
        param1 = 'Hello %s' % (param1)
        return param1

appbuilder.add_view_no_menu(MyView())
```

You can find this example on SimpleView1 look at the file app/views.py

This simple example will register your view with two routing urls on:

- /myview/method1/<string:param1>
- /myview/method2/<string:param1>

No menu will be created for this and no security permissions will be created. If you want to enable detailed security access for your methods use the **@has_access** decorator.

Now run this example:

```
$ export FLASK_APP=app
$ flask run
```

You can test your methods using the following urls:

http://localhost:8080/myview/method1/john

http://localhost:8080/myview/method2/john

As you can see, those methods are public. So let's secure them. Change views.py to:

```
from flask_appbuilder import AppBuilder, BaseView, expose, has_access
from app import appbuilder


class MyView(BaseView):

    default_view = 'method1'

    @expose('/method1/')
    @has_access
    def method1(self):
        # do something with param1
        # and return to previous page or index
        return 'Hello'

    @expose('/method2/<string:param1>')
    @has_access
    def method2(self, param1):
        # do something with param1
        # and render template with param
        param1 = 'Goodbye %s' % (param1)
        return param1

appbuilder.add_view(MyView, "Method1", category='My View')
appbuilder.add_link("Method2", href='/myview/method2/john', category='My View')
```

You can find this example on SimpleView2. Take a look at their definition:

flask_appbuilder.baseviews.expose(*url='/', methods=('GET',)*) [source]

Use this decorator to expose views on your view classes.

Parameters

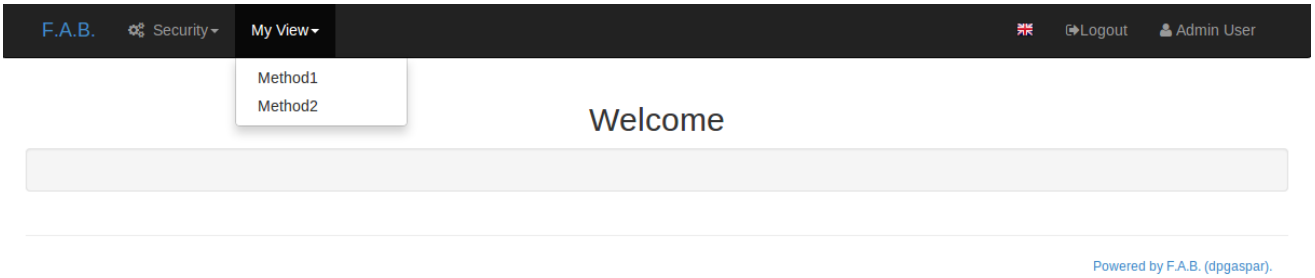
- **url** – Relative URL for the view
- **methods** – Allowed HTTP methods. By default only GET is allowed.

flask_appbuilder.security.decorators.has_access(*f*) [source]

Use this decorator to enable granular security permissions to your methods. Permissions will be associated to a role, and roles are associated to users.

By default the permission's name is the methods name.

This will create the following menu



Notice that these methods will render simple pages not integrated with F.A.B's look and feel. It's easy to render your method's response integrated with the app's look and feel, for this you have to create your own template. Under your project's directory and app folder create a folder named 'templates'. Inside it create a file name 'method3.html'

1 - Develop your template (on your <PROJECT_NAME>/app/templates/method3.html):

```
{% extends "appbuilder/base.html" %}
{% block content %}
    <h1>{{param1}}</h1>
{% endblock %}
```

2 - Add the following method on your *MyView* class:

```
from flask import render_template

@expose('/method3/<string:param1>')
@has_access
def method3(self, param1):
    # do something with param1
    # and render template with param
    param1 = 'Goodbye %s' % (param1)
    self.update_redirect()
    return self.render_template('method3.html',
                           param1 = param1)
```

3 - Create a menu link to your new method:

```
appbuilder.add_link("Method3", href='/myview/method3/john', category='My View')
```

As you can see you just have to extend "appbuilder/base.html" on your template and then override *block content*. You have many other *blocks* to override or extend things like css

includes, javascript, headers, tails etc... Next use **Flask render_template** to render your new template.

Note

Update redirect, on version 0.10.3, the redirect algorithm was reviewed, and uses session cookies to keep 5 records of navigation history. This is very useful to redirect back, keeping url arguments and improving UI experience. You must call *self.update_redirect()* to insert the current url into the navigation history. Sometimes you may want to skip the update, for example on form validation errors, so that the *back* operation won't send you to the same form, prior to the validation error.

Note

Since version 1.3.0, you must render all your views templates like *self.render_template* this is because the base_template (that can be overridden) and appbuilder are now always passed to the template.

Form Views

Subclass SimpleFormView or PublicFormView to provide base processing for your customized form views.

Usually you will need this kind of view to present forms that are not Database Model based, because when they do, F.A.B. can automatically generate them and you can add or remove fields to it, as well as custom validators. For this you can use ModelView instead.

To create a custom form view, first define your WTForm fields and inherit them from F.A.B. *DynamicForm*.

```
from wtforms import Form, StringField
from wtforms.validators import DataRequired
from flask_appbuilder.fieldwidgets import BS3TextFieldWidget
from flask_appbuilder.forms import DynamicForm


class MyForm(DynamicForm):
    field1 = StringField(('Field1'),
        description=('Your field number one!'),
        validators = [DataRequired()], widget=BS3TextFieldWidget())
    field2 = StringField(('Field2'),
        description=('Your field number two!'), widget=BS3TextFieldWidget())
```

Now define your form view to expose urls, create a menu entry, create security accesses, define pre and post processing.

Implement *form_get* and *form_post* to implement your form pre-processing and post-processing. You can use *form_get* to prefill the form with your data, and/or pre process something on your application, then use *form_post* to post process the form after the user submits it, you can save the data to database, send an email or any other action required.

On your form_post method, you can also return None, or a Flask response to render a custom template or redirect the user.

```
from flask import flash
from flask_appbuilder import SimpleFormView
from flask_babel import lazy_gettext as _


class MyFormView(SimpleFormView):
    form = MyForm
    form_title = 'This is my first form view'
    message = 'My form submitted'

    def form_get(self, form):
        form.field1.data = 'This was prefilled'

    def form_post(self, form):
        # post process form
        flash(self.message, 'info')

appbuilder.add_view(MyFormView, "My form View", icon="fa-group", label=_('My form
View'),
                    category="My Forms", category_icon="fa-cogs")
```

Notice that this class derives from *BaseView* so all properties from the parent class can be overridden. Notice also how label uses babel's lazy_gettext as _('text') function so that your menu items can be translated.

Most important Base Properties:

form_title

The title to be presented (this is mandatory)

form_columns

The form column names to include

form

Your form class (WTForm) (this is mandatory)

You can find this example on SimpleForm.

Model Views (Quick How to)

On this chapter we will create a very simple contacts application you can try a Live Demo (login with guest/welcome).

And the source code for this chapter on examples

The Base Skeleton Application

If you're working with the base skeleton application (take a look at the Installation chapter).

you now have the following directory structure:

```
<your project name>/
    config.py : All the application's configuration
    app/
        __init__.py : Application's initialization
        models.py : Declare your database models here
        views.py  : Implement your views here
```

It's very easy and fast to create an application out of the box, with detailed security.

Please take a look at github examples

Simple contacts application

Let's create a very simple contacts application. F.A.B uses the excellent SQLAlchemy ORM package, and its Flask extension. you should be familiar with its declarative syntax to define your database models on F.A.B.

Note

Since 1.3.0 there is partial support for **MongoDB** using MongoEngine. You can declare any *normalized* database schema, just like on SQLAlchemy, and use ModelView and CharView's exactly the same way. Next releases will gradually support non normalized schemas for MongoDB.

On our example application we are going to define two tables, a *Contacts* table that will hold the contact's detailed information, and a *ContactGroup* table to group our contacts or classify them. We could additionally define a *Gender* table, to serve the role of enumerated values for 'Male' and 'Female'.

Although you're not obliged to, I advise you to inherit your model classes from **Model** class. Model class is exactly the same as Flask-SQLALchemy **db.Model** but without the underlying connection. You can of course inherit from **db.Model** normal Flask-SQLAlchemy. The reason for this is that **Model** is on the same declarative space of F.A.B. and using it will allow you to

define relations to Users.

You can add automatic *Audit* triggered columns to your models, by inheriting them from *AuditMixin* also. (see API Reference)

So, first we are going to create a *ContactGroup* model, to group our contacts

Define your models (models.py)

The *ContactGroup* model.

```
from sqlalchemy import Column, Integer, String, ForeignKey, Date
from sqlalchemy.orm import relationship
from flask_appbuilder import Model

class ContactGroup(Model):
    id = Column(Integer, primary_key=True)
    name = Column(String(50), unique = True, nullable=False)

    def __repr__(self):
        return self.name
```

The *Contacts* table.

```
class Contact(Model):
    id = Column(Integer, primary_key=True)
    name =  Column(String(150), unique = True, nullable=False)
    address =  Column(String(564), default='Street ')
    birthday = Column(Date)
    personal_phone = Column(String(20))
    personal_cellphone = Column(String(20))
    contact_group_id = Column(Integer, ForeignKey('contact_group.id'))
    contact_group = relationship("ContactGroup")

    def __repr__(self):
        return self.name
```

Notice that SqlAlchemy properties used here like 'unique', 'nullable' and 'default', will have special treatment. In this case when adding a new *Contact* a query will be made to validate if someone with the same name already exists. Contacts with empty names will not be allowed. Column types are validated. The address field will contain 'Street ' as the default. You can add your own custom validations too, take a look at Advanced Configuration

Define your Views (views.py)

Now we are going to define our view for *ContactGroup* model. This view will setup functionality for create, remove, update and show primitives for your model's definition.

We inherit from the *ModelView* class, which inherits from *BaseCRUDView*, which itself inherits from *BaseModelView*, so you can override all their public properties to configure many details for your CRUD primitives. Take a look at Advanced Configuration.

```
from flask_appbuilder import ModelView
from flask_appbuilder.models.sqla.interface import SQLAInterface

class GroupModelView(ModelView):
    datamodel = SQLAInterface(ContactGroup)
    related_views = [ContactModelView]
```

I hope this was easy enough! Some questions may arise...

Required properties:

datamodel

is the db abstraction layer. Initialize it with your view's model.

Optional properties:

related_views

if you want a master/detail view on the show and edit. F.A.B. will relate 1/N relations automatically, it will display a show or edit view with tab (or accordion) with a list related record. You can relate charts also.

This is the most basic configuration (with an added related view).

But where is ContactModelView ? (that was a reference in *related_views* list)

Let's define it:

```
class ContactModelView(ModelView):
    datamodel = SQLAInterface(Contact)

    label_columns = {'contact_group':'Contacts Group'}
    list_columns = ['name','personal_cellphone','birthday','contact_group']

    show_fieldsets = [
        (
            'Summary',
            {'fields': ['name', 'address', 'contact_group']}
        ),
        (
            'Personal Info',
            {'fields': ['birthday', 'personal_phone', 'personal_cellphone'],
'expanded': False}
        ),
    ]
```

Some explanation:

label_columns

defines the labels for your columns. The framework will define the missing ones for you, with a pretty version of your column names.

show_fieldsets

A fieldset (Django style). You can use show_fieldsets, add_fieldsets, edit_fieldsets customize the show, add and edit views independently.

Additionally, you can customize which columns are displayed and their order on lists and forms. Remember you can include columns, relations or methods from a model's definition. If you have a long list of columns and want to exclude just a few from add/edit/show form you can use the exclude columns property:

class **flask_appbuilder.baseviews.BaseCRUDView(***kwargs*)** [source]

The base class for ModelView, all properties are inherited Customize ModelView overriding this properties

add_columns= *None*

A list of columns (or model's methods) to be displayed on the add form view. Use it to control the order of the display

add_exclude_columns= *None*

A list of columns to exclude from the add form. By default all columns are included.

edit_columns= *None*

A list of columns (or model's methods) to be displayed on the edit form view. Use it to control the order of the display

edit_exclude_columns= *None*

A list of columns to exclude from the edit form.

By default all columns are included.

list_columns= *None*

A list of columns (or model's methods) to be displayed on the list view. Use it to control the order of the display

show_columns= *None*

A list of columns (or model's methods) to be displayed on the show view. Use it to control the order of the display

show_exclude_columns= *None*

A list of columns to exclude from the show view. By default all columns are included.

You can also control which columns will be included on search, use the same logic for this:

class flask_appbuilder.baseviews.BaseModelView(***kwargs*) [source]

The base class of ModelView and ChartView, all properties are inherited Customize ModelView and ChartView overriding this properties

This class supports all the basics for query

search_columns= *None*

List with allowed search columns, if not provided all possible search columns will be used If you want to limit the search (*filter*) columns possibilities, define it with a list of column names from your model:

```
class MyView(ModelView):
    datamodel = SQLAInterface(MyTable)
    search_columns = ['name','address']
```

search_exclude_columns= *None*

List with columns to exclude from search. Search includes all possible columns by default

You can easily use builtin alternative look, using widgets take a look at the widgets example.

Note

Fields that reference relationships, will display the defined related model representation (on this case __repr__() methods on ContactGroup Model), so by default these fields can't be ordered. To enable order by on a list for relationship fields, you can (since 1.1.1) reference them using dotted notation. On this example you can reference them using 'contact_group.name'.

Register (views.py)

Register everything, to present the models and create the menu. Issue **create_all** to create your models also.:

```
db.create_all()
appbuilder.add_view(
    GroupModelView,
    "List Groups",
    icon = "fa-folder-open-o",
    category = "Contacts",
    category_icon = "fa-envelope"
)
appbuilder.add_view(
    ContactModelView,
    "List Contacts",
    icon = "fa-envelope",
    category = "Contacts"
)
```

Take a look at the API Reference for add_view method.

Security

FAB will create all possible permissions and add them to the `AUTH_ROLE_ADMIN` config key that defaults to **Admin**. you can completely override the default inferred permissions and reduce the level of granularity, for mode detail about this read the Security chapter.

Example and Live Demo

You can find this example at: https://github.com/dpgaspar/Flask-AppBuilder/tree/master/examples/quickhowto

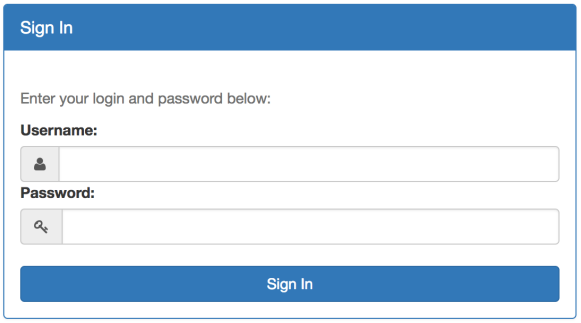
Live quickhowto Demo (login with guest/welcome).

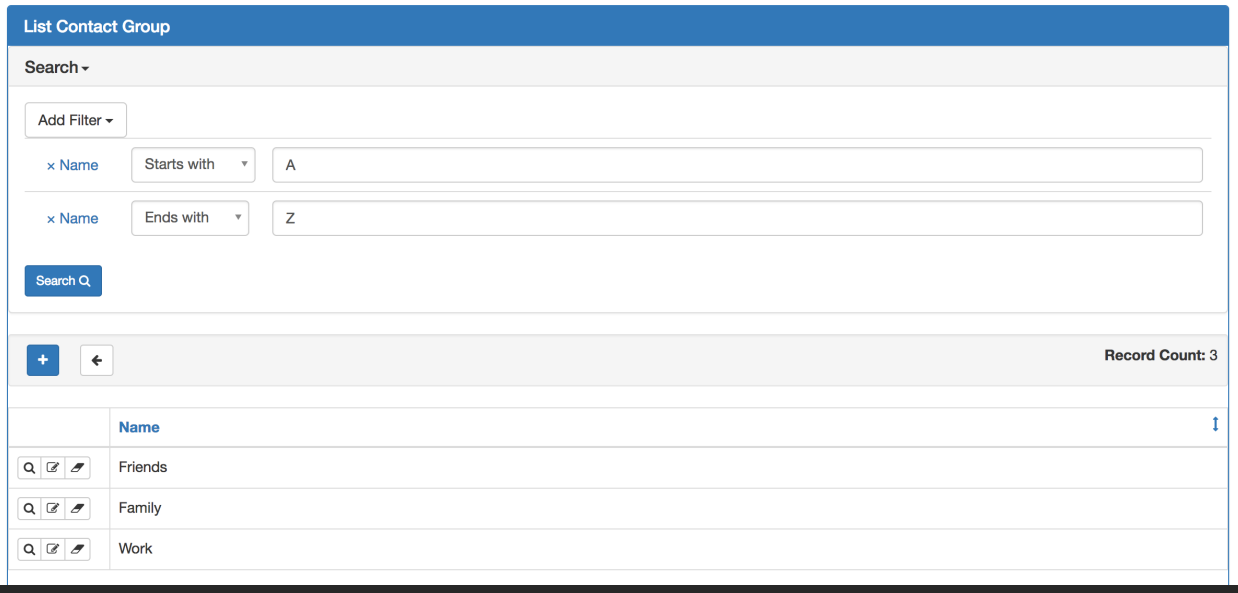
Note

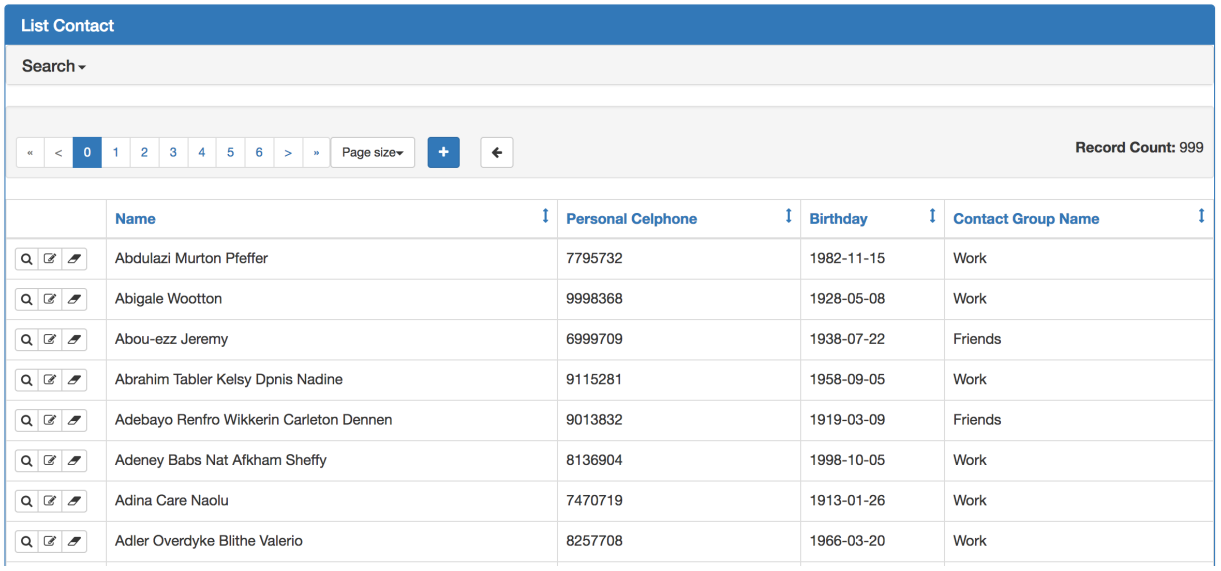
The icons for the menu on this example are from font-awesome, Checkout fontAwesome Icons names. Font-Awesome is already included and you can use any icon you like on menus and actions

With this very few lines of code (and could be fewer), you now have a web application with detailed security for each CRUD primitives and Menu options, authentication, and form field validation. Yet you can extensively change many details, add your own triggers before or after CRUD primitives, develop your own web views and integrate them.

Some images:







Exposed methods

Your **ModelView** classes expose the following methods as flask endpoints

- list
- show
- add
- edit

- delete
- download
- action
- API methods

This exposes a REST API (not completely strict). You also have an AJAX REST API. Each method has its own security permission, so you can control accesses at this level.

The API methods take the same arguments as list, show, add, edit and delete, but return JSON and HTTP return codes in case of success or errors. See the following table for a description of each method.

URL	Description
/api	Return the existing API URL's
/api/read	Queries models data, receives args as list
/api/column	Returns results for related column
/api/create	Receives a form as POST and creates record
/api/update	Receives a form as PUT and updates record
/api/column/add/<COL NAME/	Returns values for a related field
/api/column/add/<COL NAME/	Returns values for a related field
/api/readvalues	Queries models data, ready to use on select2 combos

REST API

note

This sort of automatic REST API is going to be deprecated, and will be completely removed in 2.3.X. Checkout the new API on REST API

URL=/api

The root of the API returns information about the available methods, like their URL's using url_for from Flask. The users permissions on this view, labels etc...

Let's take a close look at the returned JSON structure from this method. The returned object is a dictionary containing the following keys:

api_urls

Dictionary with All builtin CRUD methods and their URL's

`can_add`

User's permission on this view. Returns true or false.

`can_delete`

User's permission on this view. Returns true or false.

`can_edit`

User's permission on this view. Returns true or false.

`can_show`

User's permission on this view. Returns true or false.

`can_update`

User's permission on this view. Returns true or false.

`label_columns`

Dictionary for label_columns exactly equal as the ModelView property

`list_columns`

The columns to use when listing.

`modelview_name`

The name of the ModelView class.

`modelview_urls`

Dictionary with the UI's URLS for Add, Edit and Show.

`order_columns`

List with the columns allowed to do order by commands.

`page_size`

The default page size.

`search_fields`

Dictionary with column names as keys, and WTForm html fields as values.

`search_filters`

Dictionary with column names as keys and a List with allowed operations for filters as values.

URL=/api/read

This is the read method of the API, will query your model with filter, ordering and paging

operations.

Let's take a close look at the returned JSON structure from this method. The returned object is a dictionary containing the following keys:

count

Returns an Int with the total number of records.

label_columns

Dictionary for label_columns exactly equal as the ModelView property

list_columns

The columns to use when listing.

modelview_name

The name of the ModelView class.

order_columns

List with the columns allowed to do order by commands.

page

Returns an Int, with the page on some page size where the result is located.

page_size

Returns an Int with the current page size.

pks

Returns a List with the results private keys.

result

Returns a List with a dictionary for each record.

This method accepts as parameters the following:

Set page size

psize<YOUR MODEL VIEW>=<PAGE SIZE>

Set page

page<YOUR MODEL VIEW>=<PAGE>

Order by column

_oc_<<YOUR MODEL VIEW>=<COLUMN NAME>

Order by direction

_od_<<YOUR MODEL VIEW>=<asc|desc>

Filters

flt<INDEX of the search operations for this column>_<COLUMN NANE>=<VALUE>
example: _flt_0_name=A

URL=/api/delete/<PK>

Deletes a record from the model only accepts HTTP DELETE operations. if you want to delete a record with 8 as primary key issue an HTTP DELETE to the following URL: htpp://localhost:8080/contactmodelview/delete/8

It will return a dictionary that on case of success will have the following keys (returns HTTP 200):

{ “message”: “Deleted Row”, “severity”: “success” }

In case of error (returns HTTP 500):

{ “message”: “General Error <class ‘sqlalchemy.orm.exc.UnmappedInstanceError’>”, “severity”: “danger” }

Extra Views

F.A.B. as some extra views like **ModelView** but with different behaviours. You can radically change the way a ModelView looks like using various approaches like changing CRUD templates or widgets, CSS, inserting or injecting your own HTML etc, take a look at Templates, Advanced Configuration, Customizing.

Yet the framework brings 3 extra subclasses from BaseCRUDView (**ModelView** is a subclass of **BaseCRUDView**, this means that it implements complete CRUD based on models as well as JSON exposure). This views implement alternative CRUD GUI.

For rendering multiple views (subclasses of BaseModelView) on the same page use **MultipleView**. Using our previous example you could render the Group list and Contact list on the same page, to do it add the following view after the definition of **GroupModelView** and **ContactModelView**:

First remember to import:

```
from flask_appbuilder import MultipleView
```

Then define your View:

```
class MultipleViewsExp(MultipleView):
    views = [GroupModelView, ContactModelView]
```

Then register the view with a menu:

```
appbuilder.add_view(
    MultipleViewsExp,
    "Multiple Views",
    icon="fa-envelope",
    category="Contacts"
)
```

You can render as many views on the same page as you want, this includes Chart type views also, take a look at Chart Views to learn about Chart views.

Another interesting alternative view is the **MasterDetailView** as the name implies it implements a master detail GUI, it will render a menu version of a chosen model and then relate with a previous defined BaseModelView subclass of you choice. Again using the Contact application example:

```
class GroupMasterView(MasterDetailView):
    datamodel = SQLAInterface(ContactGroup)
    related_views = [ContactModelView]
```

The datamodel is the master and the **related_views** property are the views to be filtered by the user's selection of the group. You can define as many detail views as you like and again you can even include Chart type views (they are subclasses of BaseModelView), remember there must be a model relation between the master and the details, and again the framework will figure out how to relate them by inspecting the backend defined relationships.

REST API

On this chapter we are going to describe how you can define a RESTful API using almost the same concept as defining your MVC views.

note

Follow this example on Flask-AppBuilder project ./examples/base_api/

First let's see a basic example on how you can define your own custom API endpoints:

```
from flask_appbuilder.api import BaseApi, expose
from . import appbuilder


class ExampleApi(BaseApi):
    @expose('/greeting')
    def greeting(self):
        return self.response(200, message="Hello")


appbuilder.add_api(ExampleApi)
```

On the previous example, we are exposing an HTTP GET endpoint, that returns the following JSON payload:

```
{
    "message": "Hello"
}
```

The `@expose` decorator registers your class method as a Flask route that is going to be associated with a Flask blueprint. A `BaseApi` class defines a blueprint that contains all exposed methods. By default the base route of the class blueprint is defined by:

`/api/v1/<LOWERCASE_CLASS_NAME>`

So we can make a request to our method using:

```
$ curl http://localhost:8080/api/v1/exampleapi/greeting
```

To override the base route class blueprint, override the `route_base` property, so on our previous example:

```
from flask_appbuilder.api import BaseApi, expose
from . import appbuilder


class ExampleApi(BaseApi):

    route_base = '/newapi/v2/nice'

    @expose('/greeting')
    def greeting(self):
        return self.response(200, message="Hello")


appbuilder.add_api(ExampleApi)
```

Now our endpoint will be:

```
$ curl http://localhost:8080/newapi/v2/nice/greeting
```

We can also just override the version and/or resource name, using `version` and `resource_name` properties:

```
from flask_appbuilder.api import BaseApi, expose
from . import appbuilder


class ExampleApi(BaseApi):

    resource_name = 'example'

    @expose('/greeting')
    def greeting(self):
        return self.response(200, message="Hello")


appbuilder.add_api(ExampleApi)
```

Now our endpoint will be:

```
$ curl http://localhost:8080/api/v1/example/greeting
```

The other HTTP methods (PUT, POST, DELETE, ...) can be defined just like a Flask route signature:

```
from flask import request
from flask_appbuilder.api import BaseApi, expose

class ExampleApi(BaseApi):

    ....

    @expose('/greeting2', methods=['POST', 'GET'])
    def greeting2(self):
        if request.method == 'GET':
            return self.response(200, message="Hello (GET)")
        return self.response(201, message="Hello (POST)")
```

The previous example will expose a new *greeting2* endpoint on HTTP GET and POST so we can request it by:

```
$ curl http://localhost:8080/api/v1/example/greeting2
{
    "message": "Hello (GET)"
}
$ curl -XPOST http://localhost:8080/api/v1/example/greeting2
{
    "message": "Hello (POST)"
}
```

Let's make our method a bit more interesting, and send our name on the HTTP GET method. You can optionally use a `@rison` decorator that will parse the HTTP URI arguments from a *Rison* structure to a python data structure. On this example it may seem a bit overboard but with *Rison* we can handle complex HTTP GET arguments in a human readable and predictable way. *Rison* is a slight variation of JSON that looks vastly superior after URI encoding. Rison still expresses exactly the same set of data structures as JSON, so data can be translated back and forth without loss or guesswork:

```
from flask_appbuilder.api import BaseApi, expose, rison

class ExampleApi(BaseApi):

    ...

    @expose('/greeting3')
    @rison()
    def greeting3(self, **kwargs):
        if 'name' in kwargs['rison']:
            return self.response(
                200,
                message=f"Hello {kwargs['rison']['name']}"
            )
        return self.response_400(message="Please send your name")
```

And to test our method:

```
$ curl 'http://localhost:8080/api/v1/example/greeting3?q=(name:daniel)'
{
    "message": "Hello daniel"
}
```

To test this concept let's create a new method where we send a somewhat complex data structure that will use numbers, booleans and lists, and send it back JSON formatted. First our data structure, let's first think JSON:

```
```

```
{
    "bool": true,
    "list": ["a", "b", "c"],
    "number": 777,
    "string": "string"
    "null": null
}
```

On *Rison* format:

```
(bool:!t,list:!(a,b,c),null:!n,number:777,string:'string')
```

Behind the scenes FAB is using *prison* a very nicely done fork developed by @betodealmeida
We can use this package, to help us dump or load python structures to Rison:

```
import prison
b = {
    "bool": True,
    "list": ["a", "b", "c"],
    "number": 777,
    "string": "string",
    "null": None
}

print(prison.dumps(b))
```

So to test our concept:

```
...

@expose('/risonjson')
@rison()
def rison_json(self, **kwargs):
    return self.response(200, result=kwargs['rison'])
```

Then call it:

```

```

```
$ curl 'http://localhost:8080/api/v1/example/risonjson?q=(bool:!t,list:!(a,b,c),null:!
n,number:777,string:'string')'
{
  "result": {
    "bool": true,
    "list": [
      "a",
      "b",
      "c"
    ],
    "null": null,
    "number": 777,
    "string": "string"
  }
}
```

Notice how the data types are preserved. Remember that we are building a Flask app so you can always use *normal* URI arguments using Flask's `request.args`

If we send an invalid *Rison* argument we get an error:

```
$ curl -v 'http://localhost:8080/api/v1/example/risonjson?q=(bool:!t'
...
< HTTP/1.0 400 BAD REQUEST
< Content-Type: application/json; charset=utf-8
...
{
  "message": "Not a valid rison argument"
}
```

You can additionally pass a JSON schema to validate your Rison arguments, this way you can implement a very strict API easily:

```
schema = {
    "type": "object",
    "properties": {
        "name": {
            "type": "string"
        }
    }
}
...

@expose('/greeting4')
@rison(schema)
def greeting4(self, **kwargs):
    return self.response(
        200,
        message=f"Hello {kwargs['rison']['name']}"
    )
```

Finally to properly handle all possible exceptions use the `safe` decorator, that will catch all uncaught exceptions for you and return a proper error response. You can enable or disable stack trace response using the `FAB_API_SHOW_STACKTRACE` configuration key:

```
from flask_appbuilder.api import BaseApi, expose, rison, safe

...

@expose('/error')
@safe
def error(self):
    raise Exception
```

OpenAPI spec

We can define an OpenAPI specification by using YAML on the docs section of our methods:

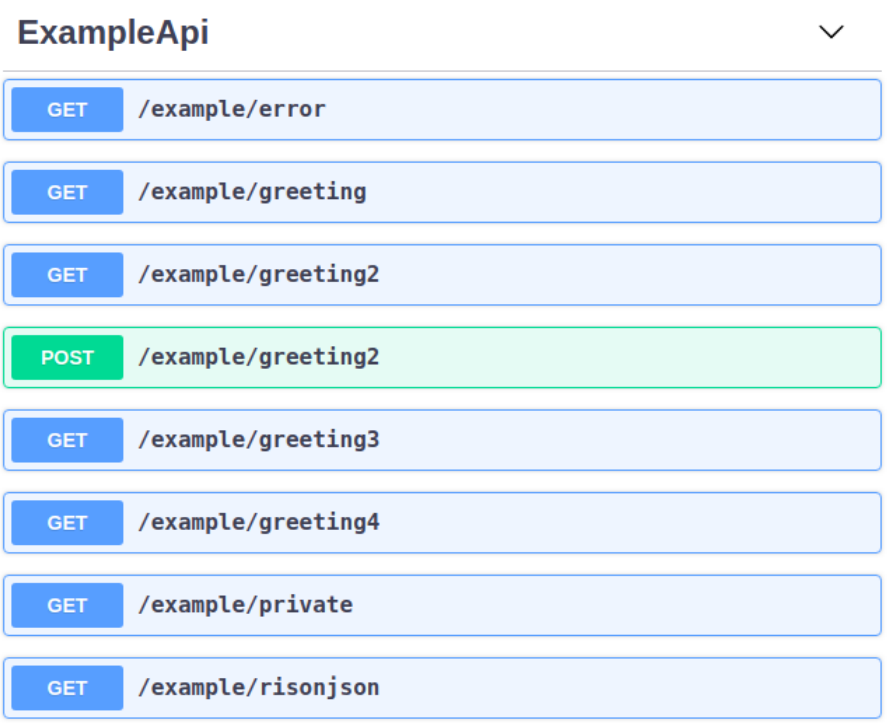
```
@expose('/greeting')
def greeting(self):
    """Send a greeting
    ---
    get:
      responses:
        200:
          description: Greet the user
          content:
            application/json:
              schema:
                type: object
                properties:
                  message:
                    type: string
    """
    return self.response(200, message="Hello")
```

We are defining that, our endpoint will respond to HTTP GET with a JSON object that contains a key `message` with values of type **string**. To access all our OpenAPI specifications request it on `/api/v1/_openapi`, this is a dynamic endpoint that will serve all specs from different API versions. So if we register an API for version **v2** we access it's spec on `/api/v2/_openapi`. Please note that OpenAPI specs are subject to authentication.

So our spec for a method that accepts two HTTP verbs:

```
@expose('/greeting2', methods=['POST', 'GET'])
def greeting2(self):
    """Send a greeting
    ---
    get:
      responses:
        200:
          description: Greet the user
          content:
            application/json:
              schema:
                type: object
                properties:
                  message:
                    type: string
    post:
      responses:
        201:
          description: Greet the user
          content:
            application/json:
              schema:
                type: object
                properties:
                  message:
                    type: string
    """
    if request.method == 'GET':
        return self.response(200, message="Hello (GET)")
    return self.response(201, message="Hello (POST)")
```

To access Swagger UI you must enable `FAB_API_SWAGGER_UI = True` on your config file then goto `http://localhost:8080/swagger/v1` for OpenAPI **v1** definitions On Swagger UI our example API looks like:



Notice the `get` and `put` structures, we should always detail all our possible responses. The

`BaseApi` class comes with some pre packaged HTTP responses we can use for the sake of brevity:

```
@expose('/error')
@protect()
@safe
def error(self):
    """Error 500
    ---
    get:
      responses:
        500:
          $ref: '#/components/responses/500'
    """
    raise Exception
```

A complete list of packaged responses you can use:

```
responses:
  400:
    $ref: '#/components/responses/400'
  401:
    $ref: '#/components/responses/401'
  404:
    $ref: '#/components/responses/404'
  422:
    $ref: '#/components/responses/422'
  500:
    $ref: '#/components/responses/500'
```

The automatic OpenAPI spec generation also supports **Rison** arguments and their json schema spec. Since both are compatible we can reuse our Json schema spec on OpenAPI. First we need to register our spec, using `apispec_parameter_schemas` dictionary:

```
class ExampleApi(BaseApi):

    resource_name = 'example'
    apispec_parameter_schemas = {
        "greeting_schema": greeting_schema
    }
```

FAB will register your schema on `/components/parameters`, so you can now easily reference them:

```
@expose('/greeting4')
@rison(greeting_schema)
def greeting4(self, **kwargs):
    """Get item from Model
    ---
    get:
      parameters:
      - $ref: '#/components/parameters/greeting_schema'
      responses:
        200:
          description: Greet the user
          content:
            application/json:
              schema:
                type: object
                properties:
                  message:
                    type: string
    """
    return self.response(
        200,
        message=f"Hello {kwargs['rison']['name']}"
    )
```

Security

FAB offers user management, several authentication backends and granular role base access so we can use these features on the API also. Default API authentication method is done using JSON Web Tokens (JWT).

tip

FAB's JWT authentication is done with flask-jwt-extended. Checkout it's documentation for custom configuration: https://flask-jwt-extended.readthedocs.io/en/latest/options.html

Next, let's see how to create a private method:

```
from flask import request
from flask_appbuilder.api import BaseApi, expose, rison
from flask_appbuilder.security.decorators import protect
from . import appbuilder


class ExampleApi(BaseApi):

    ...
    @expose('/private')
    @protect()
    def rison_json(self):
        """Say it's risonjson
        ---
        get:
          responses:
            200:
              description: Say it's private
              content:
                application/json:
                  schema:
                    type: object
            401:
              $ref: '#/components/responses/401'
        """
        return self.response(200, message="This is private")


appbuilder.add_api(ExampleApi)
```

Accessing this method as expected will return an HTTP 401 not authorized code and message:

```
$ curl -v 'http://localhost:8080/api/v1/example/private'
...
< HTTP/1.0 401 UNAUTHORIZED
< Content-Type: application/json
...
{
    "msg": "Missing Authorization Header"
}
```

So we need to first obtain our JSON Web token, for this, FAB registers a login endpoint. For this we POST request with a JSON payload using:

```
{
    "username": "<USERNAME>",
    "password": "<PASSWORD>",
    "provider": "db|ldap"
}
```

Notice the *provider* argument, FAB currently supports DB and LDAP authentication backends

for the Api. The login endpoint returns a fresh **access token** and optionally a **refresh token**. You can renew the **access token** using the **refresh token** but this time the returned token will not be fresh. To obtain a new non fresh access token use `refresh` endpoint with the **refresh token**. To obtain a **refresh token** on the login endpoint send the optional parameter **"refresh": true** on the JSON PUT payload.

Let's request our Token then:

```
# If not already, create an admin user
$ export FLASK_APP=app
$ flask fab create-admin
Username [admin]:
User first name [admin]:
User last name [user]:
Email [admin@fab.org]:
Password:
Repeat for confirmation:
...
Admin User admin created.

# Login to obtain a token
$ curl -XPOST http://localhost:8080/api/v1/security/login -d \
  '{"username": "admin", "password": "password", "provider": "db"}' \
  -H "Content-Type: application/json"
{
    "access_token": "<SOME TOKEN>"
}
# It's nice to use the Token as an env var
$ export TOKEN="<SOME TOKEN>"
```

Next we can use our token on protected endpoints:

```
$ curl 'http://localhost:8080/api/v1/example/private' -H "Authorization: Bearer $TOKEN"
{
    "message": "This is private"
}
```

As always FAB created a new **can_private** permission on the DB named "can private on ExampleApi" Note that you can protect all your methods and make them public or not by adding them to the *Public* Role.

Also to restrict the default permissions we can use `base_permissions` list property. This can be specially useful on `ModelRestApi` (up next) where we can restrict our Api resources to be read only, or only allow POST methods:

```
class ExampleApi(BaseApi):
    base_permissions = ['can_private']
```

You can create an alternate JWT user loader, this can be useful if you want to use an external Authentication provider and map the JWT identity to your user Model:

```
@appbuilder.sm.jwt_manager.user_loader_callback_loader
def alternate_user_loader(identity):
    # find the user by it's identity
    ...
    return user
```

Optionally you can enable signed cookie sessions (from flask-login) on the API. You can do it class or method wide:

```
class ExampleApi(BaseApi):
    allow_browser_login = True
```

The previous example will enable cookie sessions on the all class:

```
class ExampleApi(BaseApi):

    @expose('/private')
    @protect(allow_browser_login=True)
    def private(self)
        ....
```

On the previous example, we are enabling signed cookies on the `private` method. Note that even then a valid JWT is also accepted.

Model REST API

To automatically create a RESTfull CRUD Api from a database *Model*, use `ModelRestApi` class and define it almost like an MVC `ModelView`. This class will expose the following REST endpoints

note

Follow this example on Flask-AppBuilder project ./examples/crud_rest_api/

URL	Description	Permission Name
/_info	Returns info about the CRUD model and security	can_info
/	Queries models data, receives args as Rison	can_get

URL	Description	Permission Name
/<PK>	Returns a single model from it's primary key (id)	can_get
/	Receives a JSON payload as POST and creates record	can_post
/	Receives a JSON payload as PUT and updates record	can_put
/<PK>	Deletes a single model from it's primary key (id)	can_delete

For each `ModelRestApi` you will get 5 CRUD endpoints and an extra information method. All created CRUD endpoints have their OpenAPI spec accessible on `/api/<version>/_openapi`, each class is tagged so the CRUD endpoints get nicely grouped when using Swagger UI. Notice that `ModelRestApi` will generate a complete OpenAPI schema models for you data, so you can get free documentation for you API's.

FAB will create all possible permissions and add them to the `AUTH_ROLE_ADMIN` config key that defaults to **Admin**. you can completely override the default inferred permissions and reduce the level of granularity, for more detail about this read the Security chapter.

Let's dive into a simple example using the quickhowto. The quickhowto example has a Contact's Model and a Group Model, so each Contact belongs to a Group.

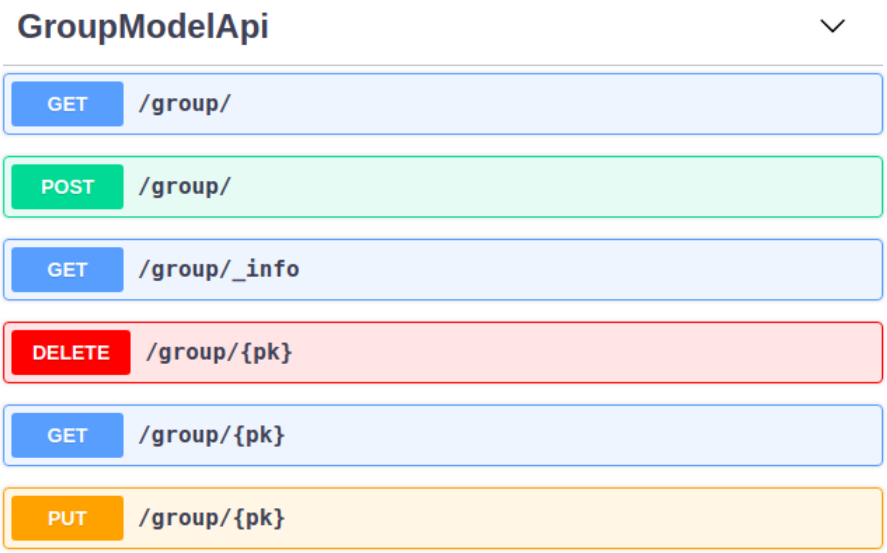
First let's define a CRUD REST Api for our Group model resource:

```
from flask_appbuilder.models.sqla.interface import SQLAInterface
from flask_appbuilder.api import ModelRestApi
from . import appbuilder


class GroupModelApi(ModelRestApi):
    resource_name = 'group'
    datamodel = SQLAInterface(ContactGroup)

appbuilder.add_api(GroupModelApi)
```

Behind the scenes FAB uses marshmallow-sqlalchemy to infer the Model to a Marshmallow Schema, that can be safely serialized and deserialized. Let's recall our Model definition for `ContactGroup`:

```
class ContactGroup(Model):
    id = Column(Integer, primary_key=True)
    name = Column(String(50), unique=True, nullable=False)

    def __repr__(self):
        return self.name
```

Swagger UI API representation for groups (http://localhost:8080/swagger/v1):



All endpoints are protected so we need to request a JWT and use it on our REST resource, like shown before we need to make a PUT request to the login API endpoint:

```
# Login to obtain a token
$ curl -XPOST http://localhost:8080/api/v1/security/login -d \
  '{"username": "admin", "password": "password", "provider": "db"}' \
  -H "Content-Type: application/json"
{
    "access_token": "<SOME TOKEN>"
}
# It's nice to use the Token as an env var
$ export TOKEN="<SOME TOKEN>"
```

First let's create a Group:

```
$ curl -XPOST http://localhost:8080/api/v1/group/ -d \
 '{"name": "Friends"}' \
 -H "Content-Type: application/json" \
 -H "Authorization: Bearer $TOKEN"
{
    "id": 1,
    "result": {
        "name": "Friends"
    }
}
```

We got back a response with the model id and result with the inserted data. Now let's query our newly created Group:

```
$ curl http://localhost:8080/api/v1/group/1 \
 -H "Content-Type: application/json" \
 -H "Authorization: Bearer $TOKEN"

{
    "description_columns": {},
    "show_title": "Show Contact Group",
    "show_columns": [
        "name"
    ],
    "label_columns": {
        "name": "Name"
    },
    "id": "1",
    "result": {
       "name": "Friends"
    }
}
```

As you can see, the API returns the model data, and extra meta data so you can properly render a page with labels, descriptions and defined column order. This way it should be possible to develop a React component (for example) that renders any model just by switching between HTTP endpoints. It's also possible to just ask for certain meta data keys, we will talk about this later.

Next let's change our newly created model (HTTP PUT):

```
$ curl -XPUT http://localhost:8080/api/v1/group/1 -d \
 '{"name": "Friends Changed"}' \
 -H "Content-Type: application/json" \
 -H "Authorization: Bearer $TOKEN"
{
    "result": {
        "name": "Friends Changed"
    }
}
```

And finally test the delete method (HTTP DELETE):

```
$ curl -XDELETE http://localhost:8080/api/v1/group/1 \
 -H "Content-Type: application/json" \
 -H "Authorization: Bearer $TOKEN"
{
    "message": "OK"
}
```

Let's check if it exists (HTTP GET):

```

```

```
$ curl http://localhost:8080/api/v1/group/1 \
 -H "Content-Type: application/json" \
 -H "Authorization: Bearer $TOKEN"
{
    "message": "Not found"
}
```

We get an HTTP 404 (Not found).

Information endpoint

This endpoint serves as a method to fetch meta information about our CRUD methods. Again the main purpose to serve meta data is to make possible for a frontend layer to be able to render dynamically:

- Search options
- Forms
- Enable/disable features based on permissions.

First a birds eye view from the output of the **_info** endpoint:

```
{
    "add_columns": [...],
    "edit_columns": [...],
    "add_title": "...",
    "edit_title": "...",
    "filters": {...},
    "permissions": [...]
}
```

Let's drill down this data structure, `add_columns` and `edit_columns` are similar and serve to aid on rendering forms for add and edit so their response contains the following data structure:

```
{
    "add_columns": [
        {
            "description": "<COL_DESCRIPTION>",
            "label": "<COL_LABEL>",
            "name": "<COL_NAME>",
            "required": true|false,
            "unique": true|false,
            "type": "String|Integer|Related|RelatedList|...",
            "validate": [ ... list of validation methods ... ]
            "count": <optional number>
            "values" : [ ... optional with all possible values for a related field ...
]
        },
        ...
    ]
}
```

Edit fields `edit_columns` is similar, but it's content may be different, since we can configure it in a distinct way

Next, filters, this returns all the necessary info to render all possible filters allowed by the backend database for each field on the model:

```
{
    "filters": {
        "<COL_NAME>": [
            {
                "name": "<HUMAN READABLE AND I18N>",
                "operator": "<OPERATION_NAME>"
            },
            ...
        ],
        ...
    }
}
```

Note that the **operator** value can be used to filter our list queries, more about this later.

Finally the permissions, this declares all allowed permissions for the current user. Remember that these can extend the automatic HTTP methods generated by `ModelRestApi` by just defining new methods and protecting them with the `protect` decorator:

```
{
    "permissions": ["can_get", "can_put", ... ]
}
```

On all GET HTTP methods we can select which meta data keys we want, this can be done using *Rison* URI arguments. So the **_info** endpoint is no exception. The across the board way

to filter meta data is to send a GET request using the following structure:

```
{
    "keys": [ ... LIST OF META DATA KEYS ... ]
}
```

That translates to the following in *Rison* for fetching just the permissions meta data:

```
(keys:!(permissions))
```

So, back to our example:

```
$ curl 'http://localhost:8080/api/v1/group/_info?q=(keys:!(permissions))' \
-H "Content-Type: application/json" \
-H "Authorization: Bearer $TOKEN"
{
    "permissions": [
        "can_get",
        "can_post",
        "can_put",
        "can_delete"
    ]
}
```

And to fetch the permissions and Add form fields info:

```
$ curl 'http://localhost:8080/api/v1/group/_info?q=(keys:!(permissions,add_columns))' \
-H "Content-Type: application/json" \
-H "Authorization: Bearer $TOKEN"
{
    "add_columns": [ ... ],
    "permissions": [
        "can_get",
        "can_post",
        "can_put",
        "can_delete"
    ]
}
```

To fetch meta data with internationalization use **_I_** URI key argument with i18n country code as the value. This will work on any HTTP GET endpoint:

```

```

```
$ curl 'http://localhost:8080/api/v1/group/_info?q=(keys:!
(permissions,add_columns))&_l_=pt' \
-H "Content-Type: application/json" \
-H "Authorization: Bearer $TOKEN"
{
    "add_columns": [ ... ],
    "permissions": [
        "can_get",
        "can_post",
        "can_put",
        "can_delete"
    ]
}
```

Render meta data with *Portuguese*, labels, description, filters

The `add_columns` and `edit_columns` keys also render all possible values from related fields, using our *quickhowto* example:

```
{
    "add_columns": [
        {
            "description": "",
            "label": "Gender",
            "name": "gender",
            "required": false,
            "unique": false,
            "type": "Related",
            "count": 2,
            "values": [
                {
                  "id": 1,
                  "value": "Male"
                },
                {
                  "id": 2,
                  "value": "Female"
                }
            ]
        },
        ...
    ]
}
```

These related field values can be filtered server side using the `add_query_rel_fields` or `edit_query_rel_fields`:

```
class ContactModelApi(ModelRestApi):
    resource_name = 'contact'
    datamodel = SQLAInterface(Contact)
    add_query_rel_fields = {
        'gender': [['name', FilterStartsWith, 'F']]
    }
```

You can also impose an order for these values server side using `order_rel_fields`:

```
class ContactModelApi(ModelRestApi):
    resource_name = 'contact'
    datamodel = SQLAInterface(Contact)
    order_rel_fields = {
        'contact_group': ('name', 'asc'),
        'gender': ('name', 'asc')
    }
```

The previous example will filter out only the **Female** gender from our list of possible values

Note that these related fields may render a long list of values, so pagination is available and subject to a max page size. You can paginate these values using the following Rison argument structure:

```
{
    "add_columns": {
        <COL_NAME> : {
            'page': int,
            'page_size': int
        }
    }
}
```

Using Rison example:

```
(add_columns:(contact_group:(page:0,page_size:10)))
```

We can also restrict server side the available fields for add and edit using `add_columns` and `edit_columns`. Additionally you can use `add_exclude_columns` and `edit_exclude_columns`:

```
class ContactModelApi(ModelRestApi):
    resource_name = 'contact'
    datamodel = SQLAInterface(Contact)
    add_columns = ['name']
```

Will only return the field *name* from our *Contact* model information endpoint for `add_fields`

Get Item

The get item endpoint is very simple, and was already covered to some extent. The response data structure is:

```
{
    "id": "<Primary Key>"
    "description_columnns": {},
    "label_columns": {},
    "show_columns": [],
    "show_title": "",
    "result": {}
}
```

Now we are going to cover the *Rison* arguments for custom fetching meta data keys or columns. This time the accepted arguments are slightly extended:

```
{
    "keys": [ ... List of meta data keys to return ... ],
    "columns": [ ... List of columns to return ... ]
}
```

So for fetching only the *name* and *address* for a certain *Contact*, using *Rison*:

```
(columns:!(name,address))
```

Our *curl* command will look like:

```
curl 'http://localhost:8080/api/v1/contact/1?q=(columns:!(name,address))' \
-H "Content-Type: application/json" \
-H "Authorization: Bearer $TOKEN"
{
  "description_columns": {},
  "id": "1",
  "show_columns": [
    "name",
    "address"
  ],
  "show_title": "Show Contact",
  "label_columns": {
    "address": "Address",
    "name": "Name"
  },
  "result": {
    "address": "Street phoung",
    "name": "Wilko Kamboh"
  }
}
```

And to only include the *label_columns* meta data, *Rison* data structure:

```
(columns:!(name,address),keys:!(label_columns))
```

Our *curl* command will look like:

```
curl 'http://localhost:8080/api/v1/contact/1?q=(columns:!(name,address),keys:!
(label_columns))' \
-H "Content-Type: application/json" \
-H "Authorization: Bearer $TOKEN"
{
  "id": "1",
  "label_columns": {
    "address": "Address",
    "name": "Name"
  },
  "result": {
    "address": "Street phoung",
    "name": "Wilko Kamboh"
  }
}
```

To discard completely all meta data use the special key `none`:

```
(columns:!(name,address),keys:!(none))
```

Our *curl* command will look like:

```
curl 'http://localhost:8080/api/v1/contact/1?q=(columns:!(name,address),keys:!(none))'
\
-H "Content-Type: application/json" \
-H "Authorization: Bearer $TOKEN"
{
  "id": "1",
  "result": {
    "address": "Street phoung",
    "name": "Wilko Kamboh"
  }
}
```

We can restrict or add fields for the get item endpoint using the `show_columns` property. This takes precedence from the *Rison* arguments:

```

```

```
class ContactModelApi(ModelRestApi):
    resource_name = 'contact'
    datamodel = SQLAInterface(Contact)
    show_columns = ['name']
```

By default FAB will issue a query containing the exact fields for *show_columns*, but these are also associated with the response object. Sometimes it's useful to distinguish between the query select columns and the response itself. Imagine the case you want to use a *@property* to further transform the output, and that transformation implies two model fields (concat or sum for example):

```
class ContactModelApi(ModelRestApi):
    resource_name = 'contact'
    datamodel = SQLAInterface(Contact)
    show_columns = ['name', 'age']
    show_select_columns = ['name', 'birthday']
```

The Model:

```
class Contact(Model):
    id = Column(Integer, primary_key=True)
    name = Column(String(150), unique=True, nullable=False)
    ...
    birthday = Column(Date, nullable=True)
    ...

    @property
    def age(self):
        return date.today().year - self.birthday.year
```

Note: The same logic is applied on *list_select_columns*

We can add fields that are python functions also, for this on the SQLAlchemy definition, let's add a new function:

```
class Contact(Model):
    id = Column(Integer, primary_key=True)
    name = Column(String(150), unique=True, nullable=False)
    address = Column(String(564))
    birthday = Column(Date, nullable=True)
    personal_phone = Column(String(20))
    personal_celphone = Column(String(20))
    contact_group_id = Column(Integer, ForeignKey('contact_group.id'), nullable=False)
    contact_group = relationship("ContactGroup")
    gender_id = Column(Integer, ForeignKey('gender.id'), nullable=False)
    gender = relationship("Gender")

    def __repr__(self):
        return self.name

    def some_function(self):
        return f"Hello {self.name}"
```

And then on the REST API:

```
class ContactModelApi(ModelRestApi):
    resource_name = 'contact'
    datamodel = SQLAInterface(Contact)
    show_columns = ['name', 'some_function']
```

The `show_columns` is also useful to impose an order on the columns. Again this is useful to develop a dynamic frontend show item page/component by using the *include_columns* meta data key.

Note that this can be done on the query list endpoint also using `list_columns`

Lists and Queries

Finally for our last HTTP endpoint, and the most feature rich. The response data structure is:

```
{
    "count": <RESULT_COUNT">
    "ids": [ ... List of PK's ordered by result ... ],
    "description_columns": {},
    "label_columns": {},
    "list_columns": [ ... An ordered list of columns ...],
    "order_columns": [ ... List of columns that can be ordered ... ],
    "list_title": "",
    "result": []
}
```

As before meta data can be chosen using *Rison* arguments:

```
(keys:!(label_columns))
```

Will only fetch the *label_columns* meta data key

And we can choose which columns to fetch:

```
(columns:!(name,address))
```

To reduce or extend the default inferred columns from our *Model*. On server side we can use the `list_columns` property, this takes precedence over *Rison* arguments:

```
class ContactModelApi(ModelRestApi):
    resource_name = 'contact'
    datamodel = SQLAInterface(Contact)
    list_columns = ['name', 'address']
```

FAB supports dotted notation (one level on GET methods only) so you can control what columns get rendered on related nested columns this applies with order by fields:

```
class ContactModelApi(ModelRestApi):
    resource_name = 'contact'
    datamodel = SQLAInterface(Contact)
    list_columns = ['name', 'address', 'contact_group.name']
```

By default related columns on this case `contact_group` will create a nested complete sub schema (on our example will return {“contact_group”: {“name”, “id”}}.

For ordering the results, the following will order contacts by name descending Z..A:

```
(order_column:name,order_direction:desc)
```

To set a default order server side use `base_order` tuple:

```
class ContactModelApi(ModelRestApi):
    resource_name = 'contact'
    datamodel = SQLAInterface(Contact)
    base_order = ('name', 'desc')
```

Pagination, get the second page using page size of two (just an example):

```
(page:2,page_size:2)
```

To set the default page size server side:

```
class ContactModelApi(ModelRestApi):
    resource_name = 'contact'
    datamodel = SQLAInterface(Contact)
    page_size = 20
```

And last, but not least, *filters*. The query *filters* data structure:

```
{
    "filters": [
        {
            "col": <COL_NAME>,
            "opr": <Operation type>,
            "value": <VALUE>
        },
        ...
    ]
}
```

All filters are **AND** operations. We can filter by several column names using different operations, so using *Rison*:

```
(filters:!((col:name,opr:sw,value:a),(col:name,opr:ew,value:z)))
```

The previous filter will query all contacts whose **name** starts with “a” and ends with “z”. The possible operations for each field can be obtained from the information endpoint. FAB can filter your models by any field type and all possible operations

Note that all *Rison* arguments can be used alone or in combination:

```
(filters:!((col:name,opr:sw,value:a),(col:name,opr:ew,value:z)),columns:!
(name),order_columns:name,order_direction:desc)
```

Will filter all contacts whose **name** starts with “a” and ends with “z”, using descending name order by, and just fetching the **name** column.

To impose base filters server side:

```
class ContactModelApi(ModelRestApi):
    resource_name = 'contact'
    datamodel = SQLAInterface(Contact)
    base_filters = [['name', FilterStartsWith, 'A']]
```

The filter will act on all HTTP endpoints, protecting delete, create, update and display operations

Simple example using doted notation, FAB will infer the necessary join operation:

```
class ContactModelApi(ModelRestApi):
    resource_name = 'contact'
    datamodel = SQLAInterface(Contact)
    base_filters = [['contact_group.name', FilterStartsWith, 'F']]
```

Locks all contacts, to groups whose name starts with “F”. Using the provided test data on the quickhowto example, limits the contacts to family and friends.

Updates and Partial Updates

PUT methods allow for changing a **Model**. Allowed changes are controlled by `edit_columns`:

```
class ContactModelApi(ModelRestApi):
    resource_name = 'contact'
    datamodel = SQLAInterface(Contact)
    edit_columns = ['name']
```

First let’s create a new contact:

```
curl -XPOST 'http://localhost:8080/api/v1/contact/' -H "Authorization: Bearer $TOKEN" -
d \
'{"name":"New Contact", "personal_celphone":"1234", "contact_group": 1, "gender":1}' \
-H "Content-Type: application/json"
{
   "id": 4,
   "result": {
       "address": null,
       "birthday": null,
       "contact_group": 1,
       "gender": 1,
       "name": "New Contact",
       "personal_celphone": "1234",
       "personal_phone": null
   }
}
```

So if you submit a change for `personal_celphone`:

```
$ curl -v XPUT http://localhost:8080/api/v1/contact/4 -d \
'{"name": "Change name", "personal_celphone": "this should not change"}' \
-H "Content-Type: application/json" \
-H "Authorization: Bearer $TOKEN"
{
    "result": {
        "name": "Change name"
    }
}
```

Let's confirm:

```
curl -XGET 'http://localhost:8080/api/v1/contact/4' -H "Authorization: Bearer $TOKEN"
{
    ....
    "id": "4",
    "result": {
        "address": null,
        "birthday": null,
        "contact_group": {
            "id": 1,
            "name": "Friends"
        },
        "gender": {
            "id": 1,
            "name": "Male"
        }
        "name": "Change name",
        "personal_celphone": "1234",
        "personal_phone": null
    }
}
```

The PUT method may also work like a PATCH method, remove the `edit_columns` from the API class and test a partial update:

```
$ curl -v XPUT http://localhost:8080/api/v1/contact/ -d \
'{"personal_celphone": "4321"}' \
-H "Content-Type: application/json" \
-H "Authorization: Bearer $TOKEN"
{
    "result": {
        "address": null,
        "birthday": null,
        "contact_group": 1
        "gender": 1,
        "name": "Change name",
        "personal_celphone": "4321",
        "personal_phone": null
    }
}
```

Validation and Custom Validation

Notice that by using marshmallow with SQLAlchemy, we are validating field size, type and required fields out of the box. This is done by marshmallow-sqlalchemy that automatically creates ModelSchema's inferred from our SQLAlchemy Models. But you can always use your own defined Marshmallow schemas independently for add, edit, list and show endpoints.

A validation error for PUT and POST methods returns HTTP 422 and the following JSON data:

```
{
    "message": {
        "<COL_NAME>": [
            "<ERROR_MESSAGE>",
            ...
        ],
        ...
    }
}
```

Next we will test some basic validation, first the field type by sending a name that is a number:

```
$ curl XPOST http://localhost:8080/api/v1/group/ -d \
'{"name": 1234}' \
-H "Content-Type: application/json" \
-H "Authorization: Bearer $TOKEN"
{
  "message": {
    "name": [
      "Not a valid string."
    ]
  }
}
```

And we get an HTTP 422 (Unprocessable Entity).

How to add custom validation? On our next example we only allow group names that start with a capital “A”:

```
from flask_appbuilder.api.schemas import BaseModelSchema


def validate_name(n):
    if n[0] != 'A':
        raise ValidationError('Name must start with an A')

class GroupCustomSchema(BaseModelSchema):
    model_cls = ContactGroup
    name = fields.Str(validate=validate_name)
```

Note that *BaseModelSchema* extends marshmallow *Schema* class, to support automatic SQLAlchemy model creation and update, it’s a lighter version of marshmallow-sqlalchemy *ModelSchema*. Declare your SQLAlchemy model on *model_cls* so that a model is created on schema load.

Then on our Api class:

```
class GroupModelRestApi(ModelRestApi):
    resource_name = 'group'
    add_model_schema = GroupCustomSchema()
    edit_model_schema = GroupCustomSchema()
    datamodel = SQLAInterface(ContactGroup)
```

Let’s try it out:

```
$ curl -v XPOST http://localhost:8080/api/v1/group/ -d \
'{"name": "BOLA"}' \
-H "Content-Type: application/json" \
-H "Authorization: Bearer $TOKEN"
{
  "message": {
    "name": [
      "Name must start with an A"
    ]
  }
}
```

Overriding completely the marshmallow Schema gives you complete control but can become very cumbersome for **Models** with many attributes, there is a simpler way of doing this using `validators_columns` property:

```
class GroupModelRestApi(ModelRestApi):
    resource_name = 'group'
    datamodel = SQLAInterface(ContactGroup)
    validators_columns = {'name': validate_name}
```

Many to Many relations

Until now we have only tested one to many relations, let's see how to handle many to many relationships. First we need to change our models, on this example we are going to add **tags** to our **Contacts**:

```
class Tag(Model):
    id = Column(Integer, primary_key=True)
    name = Column(String(50), unique=True, nullable=False)

    def __repr__(self):
        return self.name


assoc_contact_tag = Table(
    "contact_tags",
    Model.metadata,
    Column("contact_id", Integer, ForeignKey("contact.id"), nullable=True),
    Column("tag_id", Integer, ForeignKey("tag.id"), nullable=True)
)
```

Then add a new field to the *Contact* Model:

```
class Contact(Model):
    id = Column(Integer, primary_key=True)
    ...
    tags = relationship(
        "Tag",
        secondary=assoc_contact_tag,
        backref="contact"
    )
```

By default M-M fields are not required, very simple REST API's to *Contact* and *Tag* Model would be:

```
class ContactApi(ModelRestApi):
    datamodel = SQLAInterface(Contact)
    resource_name = 'contact'

appbuilder.add_api(ContactApi)

class TagApi(ModelRestApi):
    datamodel = SQLAInterface(Tag)
    resource_name = 'tag'

appbuilder.add_api(TagApi)
```

First let create some tags (this example assumes that group and gender already contains data):

```
$ curl -XPOST http://localhost:8080/api/v1/tag/ -d \
$ '{"name": "T1"}' \
$ -H "Content-Type: application/json" -H "Authorization: Bearer $TOKEN"
{"id":1,"result":{"contact": [], "name":"T1"}}
$ curl -XPOST http://localhost:8080/api/v1/tag/ -d \
$ '{"name": "T2"}' \
$ -H "Content-Type: application/json" -H "Authorization: Bearer $TOKEN"
{"id":2,"result":{"contact": [], "name":"T2"}}
```

Notice the *contact* field on the *Tag* model, this is the *backref* and is not required by default also. To create a contact with some tags:

```
$ curl -XPOST http://localhost:8080/api/v1/contact/ -d \
$ '{"name": "C1", "contact_group": 1, "gender": 1, "tags": [1, 2]}' \
$ -H "Content-Type: application/json" -H "Authorization: Bearer $TOKEN"
{"id":1,"result":
{"address":null,"birthday":null,"contact_group":1,"gender":1,"name":"C1","personal_celpho
[1,2]}}
```

You can add a contact without any tags, if you want to enforce *tags* as a required field use the

info dict from SQLAlchemy:

```
class Contact(Model):
    id = Column(Integer, primary_key=True)
    ...
    tags = relationship(
        "Tag",
        secondary=assoc_contact_tag,
        backref="contact",
        info={"required": True}
    )
```

Pre and Post processing

`ModelRestApi` offers several methods that you can override to perform pre processing or post processing on all HTTP methods. These methods are nice places to change data before submission or retrieval:

class flask_appbuilder.api.ModelRestApi [source]

post_add(*item: Model*) → None [source]

Override this, will be called after update

post_delete(*item: Model*) → None [source]

Override this, will be called after delete

post_update(*item: Model*) → None [source]

Override this, will be called after update

pre_add(*item: Model*) → None [source]

Override this, will be called before add.

pre_delete(*item: Model*) → None [source]

Override this, will be called before delete

pre_get(*data: Dict[str, Any]*) → None [source]

Override this, will be called before data is sent to the requester on get item endpoint. You can use it to mutate the response sent. Note that any new field added will not be reflected on the OpenApi spec.

pre_get_list(*data: Dict[str, Any]*) → None [source]

Override this, will be called before data is sent to the requester on get list endpoint. You can use it to mutate the response sent Note that any new field added will not be

reflected on the OpenApi spec.

pre_update(*item: Model*) → None [source]

Override this, this method is called before the update takes place.

Excluding builtin generated routes

There may be the case where you want to leverage some of the auto generated endpoints but want to disable others. For example you may want to just expose the GET endpoints for fetching a single record or records. You can declare which methods don't get registered on the Flask blueprint for the class (no permissions are created also, since it's like the methods do not exist):

```
class ContactApi(ModelRestApi):
    datamodel = SQLAInterface(Contact)
    exclude_route_methods = ("put", "post", "delete", "info")

appbuilder.add_api(ContactApi)
```

On the previous example only the `get` and `get_list` methods are registered

Note that using by normal OOP, you can override any builtin methods or create new ones

Enum Fields

`ModelRestApi` offers support for **Enum** fields, you have to declare them on a specific way:

```
class GenderEnum(enum.Enum):
    male = 'Male'
    female = 'Female'


class Contact(Model):
    id = Column(Integer, primary_key=True)
    name = Column(String(150), unique=True, nullable=False)
    address = Column(String(564))
    birthday = Column(Date, nullable=True)
    personal_phone = Column(String(20))
    personal_celphone = Column(String(20))
    contact_group_id = Column(Integer, ForeignKey('contact_group.id'), nullable=False)
    contact_group = relationship("ContactGroup")
    gender = Column(Enum(GenderEnum), nullable=False, info={"enum_class": GenderEnum})
```

Notice the `info={"enum_class": GenderEnum}`

Model Views on MongoDB

Last chapter we created a very simple contacts application, we are going to do the same, this time using MongoDB. Remember you should use the correct app skeleton, the one for MongoDB, this way the security models will be created on the MongoDB and not on SQLLite by default, take a look at the way that AppBuilder is initialized.

And the source code for this chapter on examples

Initialization

Initialization with MongoDB is a bit different, we must tell F.A.B. to use a different SecurityManager.

On __init__.py:

```
import logging
from flask import Flask
from flask_appbuilder import AppBuilder
from flask_appbuilder.security.mongoengine.manager import SecurityManager
from flask_mongoengine import MongoEngine

logging.getLogger().setLevel(logging.DEBUG)

app = Flask(__name__)
app.config.from_object('config')
dbmongo = MongoEngine(app)
# The Flask-AppBuilder init
appbuilder = AppBuilder(app, security_manager_class=SecurityManager)

from app import models, views
```

AppBuilder is initialized with the *security_manager_class* parameter with a SecurityManager class for MongoDB. All security models are created on MongoDB. Notice also that no db.session is passed to AppBuilder there is no *session* on MongoDB.

Define your models (models.py)

We are going to define two extra models from the previous example, just for fun.

The *ContactGroup* model.

```
from mongoengine import Document
from mongoengine import DateTimeField, StringField, ReferenceField, ListField

class ContactGroup(Document):
    name = StringField(max_length=60, required=True, unique=True)

    def __unicode__(self):
        return self.name

    def __repr__(self):
        return self.name
```

The *Contacts Gender* and Tags models.

```
class Gender(Document):
    name = StringField(max_length=60, required=True, unique=True)

    def __unicode__(self):
        return self.name

    def __repr__(self):
        return self.name

    def __str__(self):
        return self.name


class Tags(Document):
    name = StringField(max_length=60, required=True, unique=True)

    def __unicode__(self):
        return self.name


class Contact(Document):
    name = StringField(max_length=60, required=True, unique=True)
    address = StringField(max_length=60)
    birthday = DateTimeField()
    personal_phone = StringField(max_length=20)
    personal_celphone = StringField(max_length=20)
    contact_group = ReferenceField(ContactGroup, required=True)
    gender = ReferenceField(Gender, required=True)
    tags = ListField(ReferenceField(Tags))
```

Notice how the relations many to one and many to many are made, the framework still only supports this kind of normalized schemas.

Define your Views (views.py)

Now we are going to define our view for *ContactGroup* model. This view will setup functionality for create, remove, update and show primitives for your model's definition.

Inherit from *ModelView* class that inherits from *BaseCRUDView* that inherits from

BaseModelView, so you can override all their public properties to configure many details for your CRUD primitives. take a look at Advanced Configuration.

```
from flask_appbuilder import ModelView
from flask_appbuilder.models.mongoengine.interface import MongoEngineInterface

class GroupModelView(ModelView):
    datamodel = MongoEngineInterface(ContactGroup)
    related_views = [ContactModelView]
```

The ContactModelView ? (that was a reference in *related_views* list)

Let's define it:

```
class ContactModelView(ModelView):
    datamodel = MongoEngineInterface(Contact)

    label_columns = {'contact_group':'Contacts Group'}
    list_columns = ['name','personal_celphone','birthday','contact_group']

    show_fieldsets = [
        ('Summary',{'fields':['name','address','contact_group']}),
        ('Personal Info',{'fields':
['birthday','personal_phone','personal_celphone'],'expanded':False}),
        ]
```

Register (views.py)

Register everything, to present the models and create the menu.

```
appbuilder.add_view(GroupModelView, "List Groups",icon = "fa-folder-open-o",category =
"Contacts",
                category_icon = "fa-envelope")
appbuilder.add_view(ContactModelView, "List Contacts",icon = "fa-envelope",category =
"Contacts")
```

Take a look at the API Reference for add_view method.

As you can see, you register and define your Views exactly the same way as with SQLAlchemy. You can even use both.

Chart Views

To implement views with google charts, use all inherited classes from BaseChartView, these are:

DirectChartView

Display direct data charts with multiple series, no group by is applied.

GroupByChartView

Displays grouped data with multiple series.

ChartView

(Deprecated) Display simple group by method charts.

TimeChartView

(Deprecated) Displays simple group by month and year charts.

You can experiment with some examples on a live Demo (login has guest/welcome).

Direct Data Charts

These charts can display multiple series, based on columns or methods defined on models. You can display multiple charts on the same view.

Let's create a simple model first, the gold is to display a chart showing the unemployment evolution versus the percentage of the population with higher education, our model will be:

```
class CountryStats(Model):
    id = Column(Integer, primary_key=True)
    stat_date = Column(Date, nullable=True)
    population = Column(Float)
    unemployed_perc = Column(Float)
    poor_perc = Column(Float)
    college = Column(Float)
```

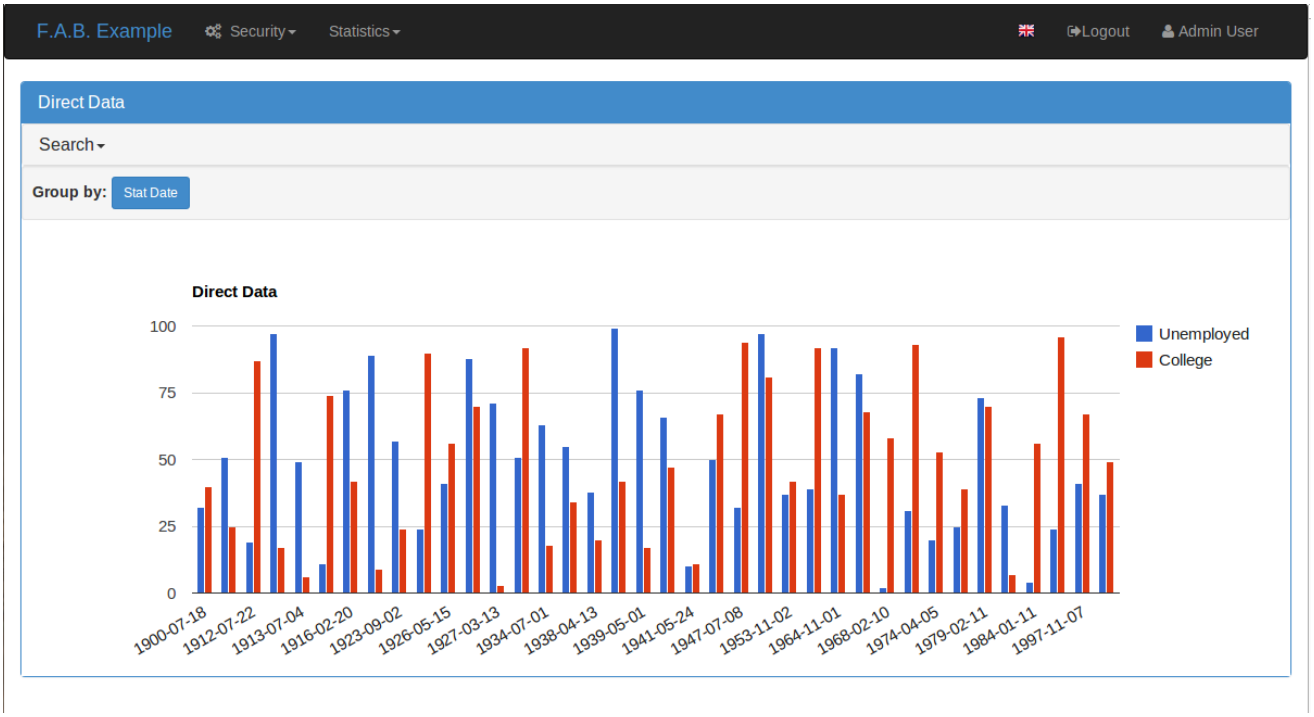
Suppose that the college field will have the total number of college students on some date. But the *unemployed_perc* field holds a percentage, we can't draw a chart with these two together, we must create a function to calculate the *college_perc*:

```
def college_perc(self):
    if self.population != 0:
        return (self.college*100)/self.population
    else:
        return 0.0
```

Now we are ready to define our view:

```
from flask_appbuilder.charts.views import DirectByChartView
from flask_appbuilder.models.sqla.interface import SQLAInterface

class CountryDirectChartView(DirectByChartView):
    datamodel = SQLAInterface(CountryStats)
    chart_title = 'Direct Data Example'

    definitions = [
    {
        'label': 'Unemployment',
        'group': 'stat_date',
        'series': ['unemployed_perc',
                   'college_perc']
    }
]
```

This view definition will produce this:



The **definitions** property respects the following grammar:

```
definitions = [
                {
                 'label': 'label for chart definition',
                 'group': '<COLNAME>'|'<MODEL FUNCNAME>',
                 'formatter': <FUNC FORMATTER FOR GROUP COL>,
                 'series': ['<COLNAME>'|'<MODEL FUNCNAME>',...]
                }, ...
               ]
```

Where ‘label’ and ‘formatter’ are optional parameters. So on the same view you can have multiple direct chart definitions, like this:

```
from flask_appbuilder.charts.views import DirectByChartView
from flask_appbuilder.models.sqla.interface import SQLAInterface

class CountryDirectChartView(DirectByChartView):
    datamodel = SQLAInterface(CountryStats)
    chart_title = 'Direct Data Example'

    definitions = [
    {
        'label': 'Unemployment',
        'group': 'stat_date',
        'series': ['unemployed_perc',
                   'college_perc']
    },
    {
        'label': 'Poor',
        'group': 'stat_date',
        'series': ['poor_perc',
                   'college_perc']
    }
]
```

Next register your view like this:

```
appbuilder.add_view(CountryDirectChartView, "Show Country Chart", icon="fa-dashboard",
category="Statistics")
```

This kind of chart inherits from **BaseChartView** that has some properties that you can configure these are:

chart_title

The Title of the chart (can be used with babel of course).

group_by_label

The label that will be displayed before the buttons for choosing the chart.

chart_type

The chart type PieChart, ColumnChart or LineChart

chart_3d

= True or false label like: 'true'

width

The charts width

height

The charts height

Additionally you can configure **BaseModelView** properties because **BaseChartView** is a child. The most interesting one is

`base_filters`

Defines the filters for data, this has precedence from all UI filters.

`label_columns`

Labeling for charts columns. If not provided the framework will generate a pretty version of the columns name.

Grouped Data Charts

These charts can display multiple series, based on columns from models or functions defined on the models. You can display multiple charts on the same view. This data can be grouped and aggregated has you like.

Let's create some simple models first, base on the prior example but this time lets make our models support has many countries has we like. The gold is to display a chart showing the unemployment versus the percentage of the population with higher education per country:

```
from flask_appbuilder import Model

class Country(Model):
    id = Column(Integer, primary_key=True)
    name = Column(String(50), unique = True, nullable=False)

    def __repr__(self):
        return self.name


class CountryStats(Model):
    id = Column(Integer, primary_key=True)
    stat_date = Column(Date, nullable=True)
    population = Column(Float)
    unemployed_perc = Column(Float)
    poor_perc = Column(Float)
    college = Column(Float)
    country_id = Column(Integer, ForeignKey('country.id'), nullable=False)
    country = relationship("Country")

    def college_perc(self):
        if self.population != 0:
            return (self.college*100)/self.population
        else:
            return 0.0

    def month_year(self):
        return datetime.datetime(self.stat_date.year, self.stat_date.month, 1)
```

Now we are ready to define our view:

```
from flask_appbuilder.charts.views import GroupByChartView
from flask_appbuilder.models.group import aggregate_count, aggregate_sum, aggregate_avg
from flask_appbuilder.models.sqla.interface import SQLAInterface


class CountryGroupByChartView(GroupByChartView):
    datamodel = SQLAInterface(CountryStats)
    chart_title = 'Statistics'

    definitions = [
        {
            'label': 'Country Stat',
            'group': 'country',
            'series': [(aggregate_avg, 'unemployed_perc'),
                       (aggregate_avg, 'population'),
                       (aggregate_avg, 'college_perc')
                      ]
        }
    ]
```

Next register your view like this:

```
appbuilder.add_view(CountryGroupByChartView, "Show Country Chart", icon="fa-dashboard",
category="Statistics")
```

F.A.B. has already some aggregation functions that you can use, for count, sum and average. On this example we are using average, this will display the historical average of unemployment and college formation, grouped by country.

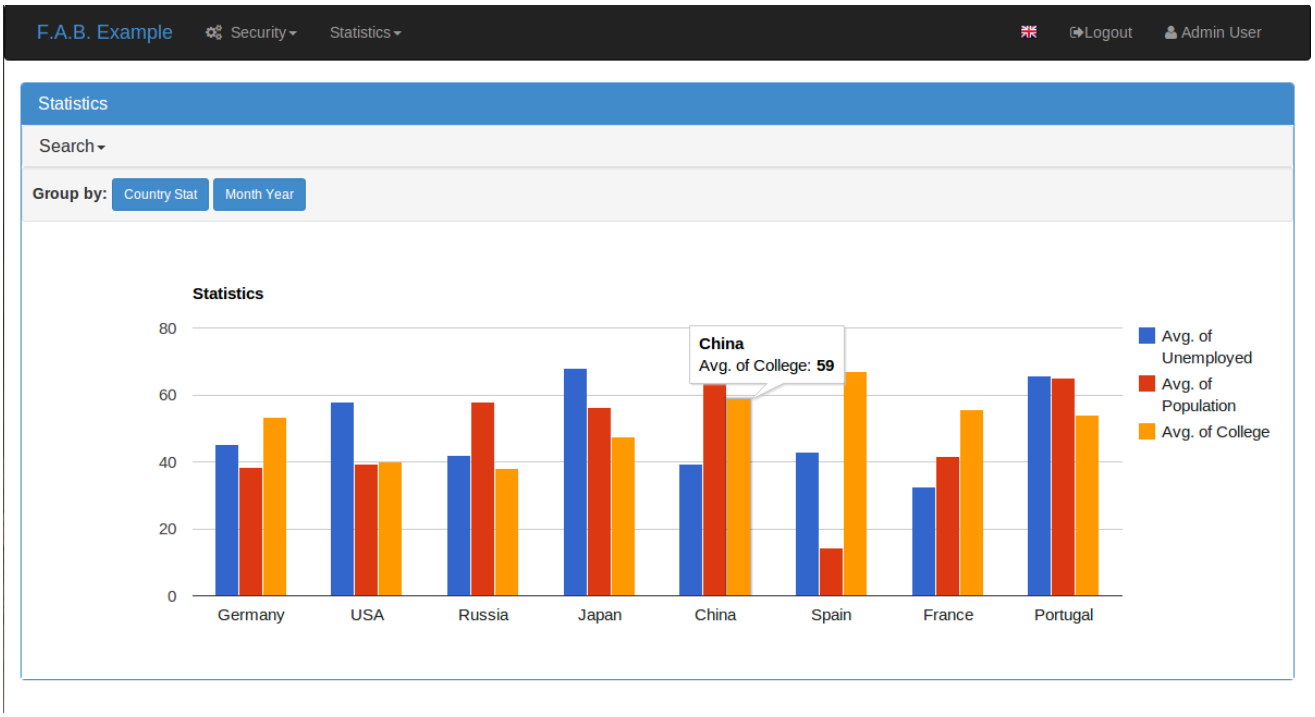
A different and interesting example is to group data monthly from all countries, this will show the use of **formater** property:

```
import calendar
from flask_appbuilder.charts.views import GroupByChartView
from flask_appbuilder.models.group import aggregate_count, aggregate_sum, aggregate_avg
from flask_appbuilder.models.sqla.interface import SQLAInterface

def pretty_month_year(value):
    return calendar.month_name[value.month] + ' ' + str(value.year)


class CountryGroupByChartView(GroupByChartView):
    datamodel = SQLAInterface(CountryStats)
    chart_title = 'Statistics'

    definitions = [
        {
            'group': 'month_year',
            'formatter': pretty_month_year,
            'series': [(aggregate_avg, 'unemployed_perc'),
                       (aggregate_avg, 'college_perc')
            ]
        }
    ]
```

This view will group data based on the model's method *month_year* that has the name says will group data by month and year, this grouping will be processed by averaging data from *unemployed_perc* and *college_perc*.

The group criteria will be formatted for display by *pretty_month_year* function that will change things like '1990-01' to 'January 1990'

This view definition will produce this:



You can create your own aggregation functions and *decorate* them for automatic labeling (and babel). Has an example let's look at F.A.B.'s code for *aggregate_sum*:

```
@aggregate(_('Count of'))
def aggregate_count(items, col):
    return len(list(items))
```

The label 'Count of' will be concatenated to your definition of *label_columns* or the pretty version generated by the framework of the columns them selfs.

(Deprecated) Define your Chart Views (views.py)

```
class ContactChartView(ChartView):
    search_columns = ['name','contact_group']
    datamodel = SQLAInterface(Contact)
    chart_title = 'Grouped contacts'
    label_columns = ContactModelView.label_columns
    group_by_columns = ['contact_group']
```

Notice that:

`label_columns`

Are the labels that will be displayed instead of the model's columns name. In this case they are the same labels from ContactModelView.

`group_by_columns`

Is a list of columns that you want to group.

this will produce a Pie chart, with the percentage of contacts by group. If you want a column chart just define:

```
chart_type = 'ColumnChart'
```

You can use 'BarChart', 'LineChart', 'AreaChart' the default is 'PieChart', take a look at the google charts documentation, the *chart_type* is the function on 'google.visualization' object

Let's define a chart grouped by a time frame?

```
class ContactTimeChartView(TimeChartView):
    search_columns = ['name','contact_group']
    chart_title = 'Grouped Birth contacts'
    label_columns = ContactModelView.label_columns
    group_by_columns = ['birthday']
    datamodel = SQLAInterface(Contact)
```

this will produce a column chart, with the number of contacts that were born on a particular month or year. Notice that the label_columns are from and already defined *ContactModelView* take a look at the Model Views (Quick How to)

Finally we will define a direct data chart

```
class StatsChartView(DirectChartView):
    datamodel = SQLAInterface(Stats)
    chart_title = lazy_gettext('Statistics')
    direct_columns = {'Some Stats': ('stat1', 'col1', 'col2'),
                      'Other Stats': ('stat2', 'col3')}
```

direct_columns is a dictionary you define to identify a label for your X column, and the Y columns (series) you want to include on the chart

This dictionary is composed by key and a tuple: {'KEY LABEL FOR X COL':('X COL','Y COL','Y2 COL',...),...}

Remember 'X COL', 'Ys COL' are identifying columns from the data model.

Take look at a more detailed example on quickcharts.

Register (views.py)

Register everything, to present your charts and create the menu:

```
appbuilder.add_view(ContactTimeChartView, "Contacts Birth Chart", icon="fa-envelope",
category="Contacts")
appbuilder.add_view(ContactChartView, "Contacts Chart", icon="fa-dashboard",
category="Contacts")
```

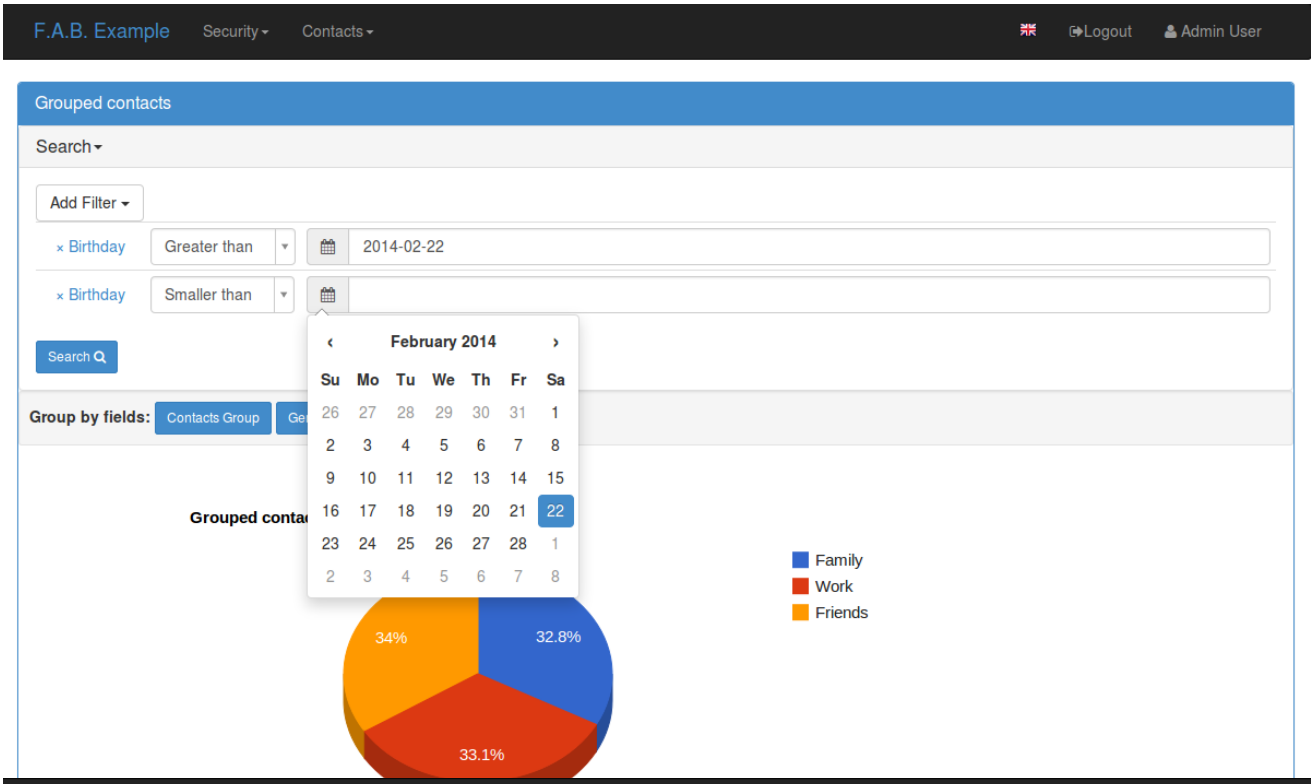
You can find this example at: https://github.com/dpgaspar/Flask-AppBuilder/tree/master/examples/quickhowto

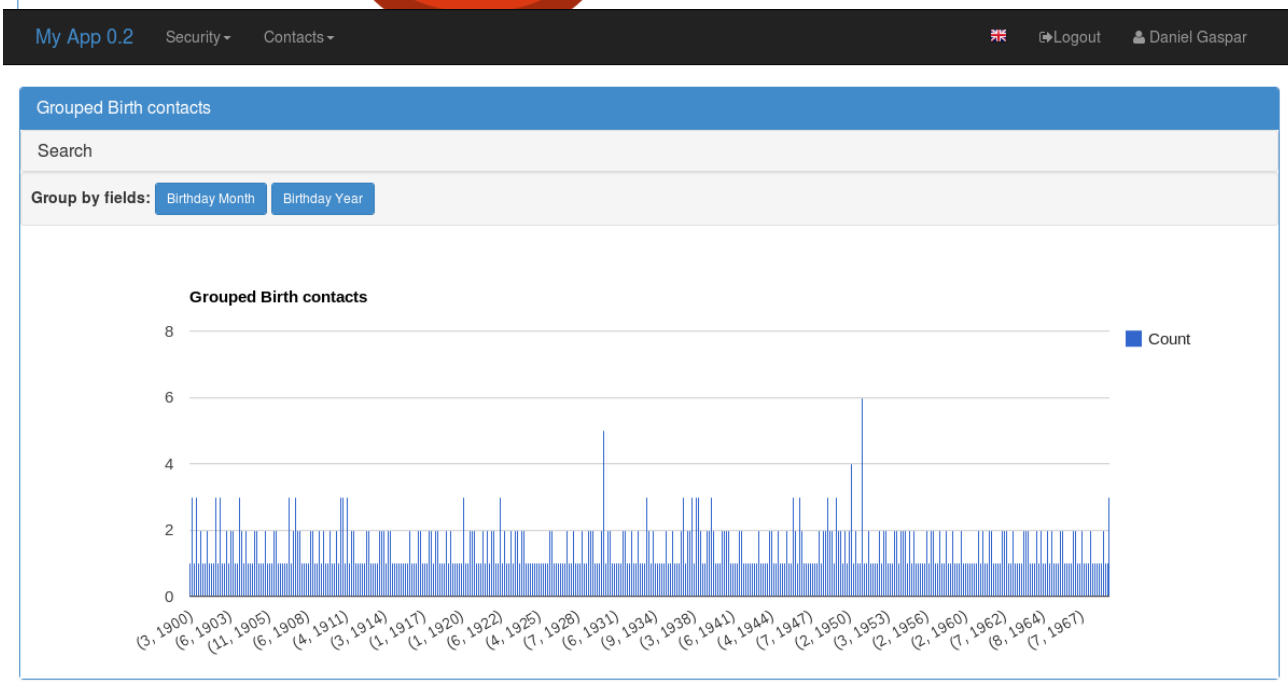
Take a look at the API Reference. For additional customization

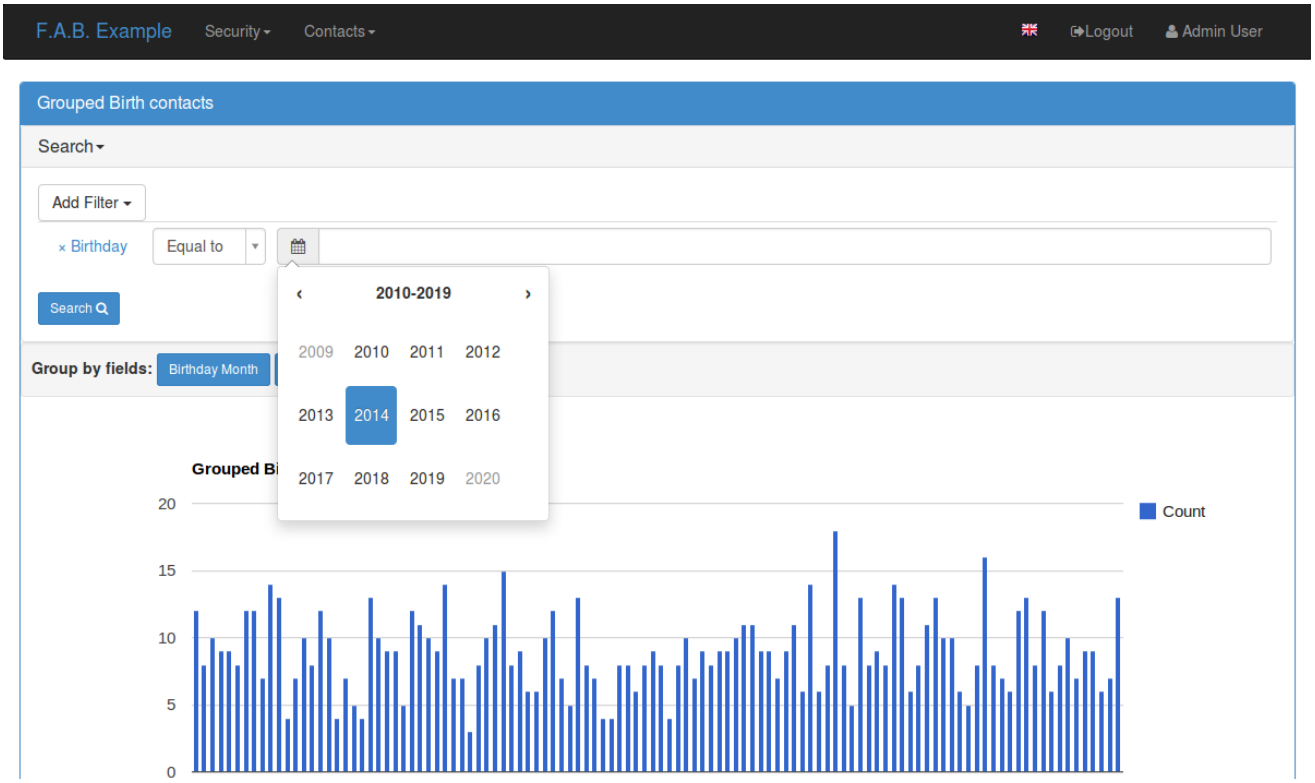
Note

You can use charts has related views also, just add them on your related_views properties.

Some images:


F.A.B. Example
Security
Contacts
Logout
Admin User
Grouped contacts
Search
Add Filter
× Birthday
Greater than
2014-02-22
× Birthday
Smaller than
February 2014
Su Mo Tu We Th Fr Sa
Search
Group by fields:
Contacts Group
Family
Work
Friends
34%
32.8%
33.1%


My App 0.2
Security
Contacts
Logout
Daniel Gaspar
Grouped Birth contacts
Search
Group by fields:
Birthday Month
Birthday Year
Grouped Birth contacts
Count
8
6
4
2
0
(3, 1900)
(6, 1903)
(11, 1905)
(6, 1908)
(4, 1911)
(3, 1914)
(1, 1917)
(1, 1920)
(6, 1922)
(4, 1925)
(7, 1928)
(6, 1931)
(9, 1934)
(3, 1938)
(6, 1941)
(4, 1944)
(7, 1947)
(2, 1950)
(3, 1953)
(2, 1956)
(2, 1960)
(7, 1962)
(8, 1964)
(7, 1967)



Model Views with Files and Images

You can implement views with images or files embedded on the model's definition. You can do it using SQLAlchemy or MongoDB (MongoEngine). When using SQLAlchemy, files and images are saved on the filesystem, on MongoDB on the db (GridFS).

Define your model (models.py)

```
from flask_appbuilder import Model
from flask_appbuilder.models.mixins import ImageColumn

class Person(Model):
    id = Column(Integer, primary_key=True)
    name = Column(String(150), unique = True, nullable=False)
    photo = Column(ImageColumn(size=(300, 300, True), thumbnail_size=(30, 30, True)))

    def photo_img(self):
        im = ImageManager()
        if self.photo:
            return Markup('<a href="' + url_for('PersonModelView.show',pk=str(self.id))
+\
             '" class="thumbnail"><img src="' + im.get_url(self.photo) +\
              '" alt="Photo" class="img-rounded img-responsive"></a>')
        else:
            return Markup('<a href="' + url_for('PersonModelView.show',pk=str(self.id))
+\
             '" class="thumbnail"><img src="//:0" alt="Photo" class="img-responsive"></
a>')

    def photo_img_thumbnail(self):
        im = ImageManager()
        if self.photo:
            return Markup('<a href="' + url_for('PersonModelView.show',pk=str(self.id))
+\
             '" class="thumbnail"><img src="' + im.get_url_thumbnail(self.photo) +\
              '" alt="Photo" class="img-rounded img-responsive"></a>')
        else:
            return Markup('<a href="' + url_for('PersonModelView.show',pk=str(self.id))
+\
             '" class="thumbnail"><img src="//:0" alt="Photo" class="img-responsive"></
a>')
```

Create two additional methods in this case *photo_img* and *photo_img_thumbnail*, to inject your own custom HTML, to show your saved images. In this example the customized method is showing the images, and linking them with the show view. Notice how the methods are calling *get_url* and *get_url_thumbnail* from ImageManager, these are returning the url for the images, each image is saved on the filesystem using the global config **IMG_UPLOAD_FOLDER**. Each image will have two files with different sizes, images are saved as <uuid>_sep_<filename>, and <uuid>_sep_<filename>_thumb

Note

The “ImageColumn” type, is an extended type from Flask-AppBuilder.

Later reference this method like it’s a column on your view.

To implement image or file support using GridFS from MongoDB is even easier, take a look at the example:

https://github.com/dpgaspar/Flask-AppBuilder/tree/master/examples/mongoimages

Define your Views (views.py)

```
from flask_appbuilder import ModelView
from flask_appbuilder.models.sqla.interface import SQLAInterface

class PersonModelView(ModelView):
    datamodel = SQLAInterface(Person)

    list_widget = ListThumbnail

    label_columns = {'name':'Name','photo':'Photo','photo_img':'Photo',
'photo_img_thumbnail':'Photo'}
    list_columns = ['photo_img_thumbnail', 'name']
    show_columns = ['photo_img','name']
```

We are overriding the *list_widget*, the widget that is normally used by ModelView. This will display a thumbnail list, excellent for displaying images.

We're not using the *image* column but the methods *photo_img* and *photo_img_thumbnail* we have created. These methods will display the images and link them to show view.

And that's it! images will be saved on the server. Their file names will result in the concatenation of UUID with their original name. They will be resized for optimization.

Note

You can define image resizing using configuration key *IMG_SIZE*

We are overriding the list_widget, the widget that is normally used by ModelView. This will display a thumbnail list excellent for displaying images.

And that's it! Images will be saved on the server with their filename concatenated by a UUID's. Aditionally will be resized for optimization.

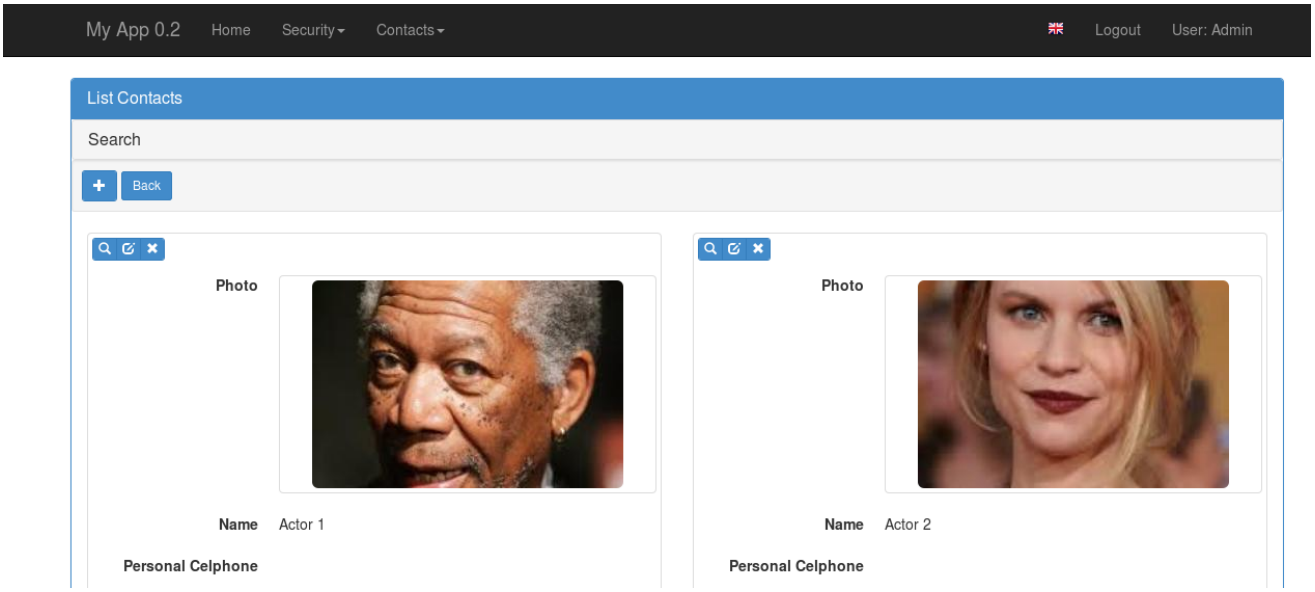
Next step

Take a look at the example:

https://github.com/dpgaspar/Flask-AppBuilder/tree/master/examples/quickimages

https://github.com/dpgaspar/Flask-AppBuilder/tree/master/examples/quickfiles

Some images:



Quick Minimal Application

How to setup a minimal Application

This is the most basic example, using the minimal code needed to setup a running application with F.A.B.

Will use sqlite for the database no need to install anything. Notice the SQLA class this is just a child class from flask.ext.SQLAlchemy that overrides the declarative base to F.A.B. You can use every configuration and method from flask extension except the model's direct query.

I do advise using the skeleton application as described on the Installation

```
import os
from flask import Flask
from flask_appbuilder import SQLA, AppBuilder

# init Flask
app = Flask(__name__)

# Basic config with security for forms and session cookie
basedir = os.path.abspath(os.path.dirname(__file__))
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///' + os.path.join(basedir, 'app.db')
app.config['CSRF_ENABLED'] = True
app.config['SECRET_KEY'] = 'thisismyscretkey'

# Init SQLAlchemy
db = SQLA(app)
# Init F.A.B.
appbuilder = AppBuilder(app, db.session)

# Run the development server
app.run(host='0.0.0.0', port=8080, debug=True)
```

If you run this, notice that your database will be created with two roles 'Admin' and 'Public', as well has all the security detailed permissions.

The default authentication method will be database, and you can initially login with **'admin'/'general'**. you can take a look at all your configuration options on Base Configuration

Take a look at this example on Github

Model Relations/Composite keys

On this chapter we are going to show how to setup model relationships and their view integration on the framework

And the source code for this chapter on examples

Many to One

First let's check the most simple relationship, already described on the quick how to with the contacts application.

Using a different (and slightly more complex) example. Let's assume we are building a human resources app. So we have an Employees table with some related data.

- Employee.
- Function.
- Department.

Each Employee belongs to a department and he/she has a particular function.

Let's define our models (models.py):

```
import datetime
from sqlalchemy import Column, Integer, String, ForeignKey, Date, Text
from sqlalchemy.orm import relationship
from flask_appbuilder import Model


class Department(Model):
    id = Column(Integer, primary_key=True)
    name = Column(String(50), unique=True, nullable=False)

    def __repr__(self):
        return self.name


class Function(Model):
    id = Column(Integer, primary_key=True)
    name = Column(String(50), unique=True, nullable=False)

    def __repr__(self):
        return self.name


def today():
    return datetime.datetime.today().strftime('%Y-%m-%d')


class Employee(Model):
    id = Column(Integer, primary_key=True)
    full_name = Column(String(150), nullable=False)
    address = Column(Text(250), nullable=False)
    fiscal_number = Column(Integer, nullable=False)
    employee_number = Column(Integer, nullable=False)
    department_id = Column(Integer, ForeignKey('department.id'), nullable=False)
    department = relationship("Department")
    function_id = Column(Integer, ForeignKey('function.id'), nullable=False)
    function = relationship("Function")
    begin_date = Column(Date, default=today, nullable=False)
    end_date = Column(Date, nullable=True)

    def __repr__(self):
        return self.full_name
```

This has two, one to many relations:

- One employee belongs to a department and a department has many employees
- One employee executes a function and a function is executed by many employees.

Now let's define ours views (views.py):

```
from flask_appbuilder import ModelView
from flask_appbuilder.models.sqla.interface import SQLAInterface
from .models import Employee,Department, Function, EmployeeHistory
from app import appbuilder


class EmployeeView(ModelView):
    datamodel = SQLAInterface(Employee)

    list_columns = ['full_name', 'department', 'employee_number']


class FunctionView(ModelView):
    datamodel = SQLAInterface(Function)
    related_views = [EmployeeView]


class DepartmentView(ModelView):
    datamodel = SQLAInterface(Department)
    related_views = [EmployeeView]
```

Has described on the Model Views (Quick How to) chapter the *related_views* property will tell F.A.B to add the defined **EmployeeView** filtered by the relation on the show and edit form for the departments and functions. So on the department show view you will have a tab with all the employees that belong to it, and of course on the function show view you will have a tab with all the employees that share this function.

Finally register everything to create the flask endpoints and automatic menu construction:

```
db.create_all()

appbuilder.add_view(EmployeeView, "Employees", icon="fa-folder-open-o",
category="Company")
appbuilder.add_separator("Company")
appbuilder.add_view(DepartmentView, "Departments", icon="fa-folder-open-o",
category="Company")
appbuilder.add_view(FunctionView, "Functions", icon="fa-folder-open-o",
category="Company")
```

Remember 'db.create_all()' will create all your models on the database if they do not exist already.

Many to Many

Our employees have benefits, and HR wants to track them. It's time to define a many to many relation.

On your model definition add the benefit model:

```
class Benefit(Model):
    id = Column(Integer, primary_key=True)
    name = Column(String(50), unique=True, nullable=False)

    def __repr__(self):
        return self.name
```

Then define the association table between Employee and Benefit, then add the relation to benefit on the Employee model:

```
assoc_benefits_employee = Table('benefits_employee', Model.metadata,
                                    Column('id', Integer, primary_key=True),
                                    Column('benefit_id', Integer,
ForeignKey('benefit.id')),
                                    Column('employee_id', Integer,
ForeignKey('employee.id'))
)


class Employee(Model):
    id = Column(Integer, primary_key=True)
    full_name = Column(String(150), nullable=False)
    address = Column(Text(250), nullable=False)
    fiscal_number = Column(Integer, nullable=False)
    employee_number = Column(Integer, nullable=False)
    department_id = Column(Integer, ForeignKey('department.id'), nullable=False)
    department = relationship("Department")
    function_id = Column(Integer, ForeignKey('function.id'), nullable=False)
    function = relationship("Function")
    benefits = relationship('Benefit', secondary=assoc_benefits_employee,
backref='employee')

    begin_date = Column(Date, default=today, nullable=False)
    end_date = Column(Date, nullable=True)

    def __repr__(self):
        return self.full_name
```

On your views (views.py) it would be nice to create a menu entry for benefits, so that HR can add the available benefits:

```
class BenefitView(ModelView):
    datamodel = SQLAInterface(Benefit)
    related_views = [EmployeeView]
    add_columns = ['name']
    edit_columns = ['name']
    show_columns = ['name']
    list_columns = ['name']
```

Then register your view:

```
appbuilder.add_view(BenefitView, "Benefits", icon="fa-folder-open-o",
category="Company")
```

F.A.B. will add a select2 widget for adding benefit tags to employees, when adding or editing an employee.

Many to Many with extra properties

Finally we are creating a history of the employee on the company, we want to record all his/her department changes and when did it occur. This can be done in different ways, this one is useful for our example on how to use a many to many relation with extra properties. So let's define our employee history model:

```
class EmployeeHistory(Model):
    id = Column(Integer, primary_key=True)
    department_id = Column(Integer, ForeignKey('department.id'), nullable=False)
    department = relationship("Department")
    employee_id = Column(Integer, ForeignKey('employee.id'), nullable=False)
    employee = relationship("Employee")
    begin_date = Column(Date, default=today)
    end_date = Column(Date)
```

As you can see, this model is related to departments and employees and it has a begin date and end date when he is/was allocated to it. It's a special kind of association table.

We want the history to be shown on the employee show/detail view, has a list history. for this we need to create a view for employee history and tell F.A.B to make a relation to it:

```
class EmployeeHistoryView(ModelView):
    datamodel = SQLAInterface(EmployeeHistory)
    list_columns = ['department', 'begin_date', 'end_date']
```

Then change the employee view, this time we do not want a tab to navigate to the relation, we want to show it on the same page cascading:

```
class EmployeeView(ModelView):
    datamodel = SQLAInterface(Employee)
    list_columns = ['full_name', 'department', 'employee_number']
    related_views = [EmployeeHistoryView]
    show_template = 'appbuilder/general/model/show_cascade.html'
```

We need to register the **EmployeeHistoryView** but without a menu, because it's history will be managed on the employee detail view:

```
appbuilder.add_view_no_menu(EmployeeHistoryView, "EmployeeHistoryView")
```

Take a look and run the example on Employees example It includes extra functionality like readonly fields, pre and post update logic, etc...

Composite Keys

Composite keys is supported for SQLAlchemy only, you can reference them using SQLAlchemy 'relationship', and use them on combo boxes and/or related views, take a look at the example

Notice the use of composite keys to prevent that and Item (server or whatever) can be on more then a Rack/Datacenter at the same time, and that a Datacenter can't have two racks with the same number

Note

This feature is only supported since 1.9.6

Actions

Define your view

You can setup your actions on records on the show or list views. This is a powerful feature, you can easily add custom functionality to your db records, like mass delete, sending emails with record information, special mass update etc.

Just use the @action decorator on your own functions. Here's an example

```
from flask_appbuilder.actions import action
from flask_appbuilder import ModelView
from flask_appbuilder.models.sqla.interface import SQLAInterface
from flask import redirect

class GroupModelView(ModelView):
    datamodel = SQLAInterface(Group)
    related_views = [ContactModelView]

    @action("myaction","Do something on this record","Do you really want to?","fa-
rocket")
    def myaction(self, item):
        """
            do something with the item record
        """
        return redirect(self.get_redirect())
```

This will create the necessary permissions for the item, so that you can include or remove them from a particular role.

You can easily implement a massive delete option on lists. Just add the following code to your view. This example will tell F.A.B. to implement the action just for list views and not show the option on the show view. You can do this by disabling the *single* or *multiple* parameters on the **@action** decorator.

```
@action("muldelete", "Delete", "Delete all Really?", "fa-rocket", single=False)
def muldelete(self, items):
    self.datamodel.delete_all(items)
    self.update_redirect()
    return redirect(self.get_redirect())
```

F.A.B will call your function with a list of record items if called from a list view. Or a single item if called from a show view. By default an action will be implemented on list views and show views so your method's should be prepared to handle a list of records or a single record:

```
@action("muldelete", "Delete", "Delete all Really?", "fa-rocket")
def muldelete(self, items):
    if isinstance(items, list):
        self.datamodel.delete_all(items)
        self.update_redirect()
    else:
        self.datamodel.delete(items)
    return redirect(self.get_redirect())
```

Advanced Configuration

Security

To block or set the allowed permissions on a view, just set the *base_permissions* property with the base permissions

```
class GroupModelView(ModelView):
    datamodel = SQLAInterface(Group)
    base_permissions = ['can_add','can_delete']
```

With this initial config, the framework will only create 'can_add' and 'can_delete' permissions on GroupModelView as the only allowed. So users and even the administrator of the application will not have the possibility to add list or show permissions on Group table view. Base available permission are: can_add, can_edit, can_delete, can_list, can_show. More detailed info on Security

Custom Fields

Custom Model properties can be used on lists. This is useful for formatting values like currencies, time or dates. or for custom HTML. This is very simple to do, first define your custom property on your Model and then use the **@renders** decorator to tell the framework to map you class method with a certain Model property:

```
from flask_appbuilder.models.decorators import renders

class MyModel(Model):
    id = Column(Integer, primary_key=True)
    name = Column(String(50), unique = True, nullable=False)
    custom = Column(Integer(20))

    @renders('custom')
    def my_custom(self):
    # will render this columns as bold on ListWidget
        return Markup('<b>' + self.custom + '</b>')
```

On your view reference your method as a column on list:

```
class MyModelView(ModelView):
    datamodel = SQLAInterface(MyTable)
    list_columns = ['name', 'my_custom']
```

Base Filtering

To filter a views data, just set the *base_filter* property with your base filters. These will allways be applied first on any search.

It's very flexible, you can apply multiple filters with static values, or values based on a function you define. On this next example we are filtering a view by the logged in user and with column *name* starting with "a"

base_filters is a list of lists with 3 values [['column name',FilterClass,'filter value],...]:

```
from flask import g
from flask_appbuilder import ModelView
from flask_appbuilder.models.sqla.interface import SQLAInterface
from flask_appbuilder.models.sqla.filters import FilterEqualFunction, FilterStartsWith
# If you're using Mongo Engine you should import filters like this, everything else is
exactly the same
# from flask_appbuilder.models.mongoengine.filters import FilterStartsWith,
FilterEqualFunction

from .models import MyTable


def get_user():
    return g.user


class MyView(ModelView):
    datamodel = SQLAInterface(MyTable)
    base_filters = [['created_by', FilterEqualFunction, get_user],
                    ['name', FilterStartsWith, 'a']]
```

Since version 1.5.0 you can use base_filter with dotted notation, necessary joins will be handled for you on the background. Study the following example to see how:

https://github.com/dpgaspar/Flask-AppBuilder/tree/master/examples/extendsecurity

Default Order

Use a default order on your lists, this can be overridden by the user on the UI. Data structure ('col_name':'asc|desc'):

```
class MyView(ModelView):
    datamodel = SQLAInterface(MyTable)
    base_order = ('my_col_to_be_ordered','asc')
```

Template Extra Arguments

You can pass extra Jinja2 arguments to your custom template, using extra_args property:

```
class MyView(ModelView):
    datamodel = SQLAInterface(MyTable)
    extra_args = {'my_extra_arg':'SOMEVALUE'}
    show_template = 'my_show_template.html'
```

Your overriding the 'show' template to handle your extra argument. You can still use F.A.B. show template using Jinja2 blocks, take a look at the Templates chapter

Forms - Override automatic form creation

Define your own Add, Edit forms using WTForms to override the automatic form creation:

```
class MyView(ModelView):
    datamodel = SQLAInterface(MyModel)
    add_form = AddFormWTF
```

Forms - Add or remove fields

Define what columns will be included on Add or Edit forms, for example if you have automatic fields like user or date, you can remove them from the Add Form:

```
class MyView(ModelView):
    datamodel = SQLAInterface(MyModel)
    add_columns = ['my_field1', 'my_field2']
    edit_columns = ['my_field1']
```

To contribute with any additional fields that are not on a table/model, for example a confirmation field:

```
class ContactModelView(ModelView):
    datamodel = SQLAInterface(Contact)
    add_form_extra_fields = {
        'extra': StringField(gettext('Extra Field'),
        description=gettext('Extra Field description'),
        widget=BS3TextFieldWidget())
    }
```

Forms - Readonly fields

Define/override readonly fields like this, first define a new **Readonly** field:

```
from flask_appbuilder.fieldwidgets import BS3TextFieldWidget


class BS3TextFieldROWidget(BS3TextFieldWidget):
    def __call__(self, field, **kwargs):
        kwargs['readonly'] = 'true'
        return super(BS3TextFieldROWidget, self).__call__(field, **kwargs)
```

Next override your field using your new widget:

```
class ExampleView(ModelView):
    datamodel = SQLAInterface(ExampleModel)
    edit_form_extra_fields = {
        'field2': StringField('field2', widget=BS3TextFieldROWidget())
    }
```

Readonly select fields are a special case, but it's solved in a simpler way:

```
# Define the field query
def department_query():
    return db.session.query(Department)


class EmployeeView(ModelView):
    datamodel = SQLAInterface(Employee)

    list_columns = ['employee_number', 'full_name', 'department']

    # override the 'department' field, to make it readonly on edit form
    edit_form_extra_fields = {
        'department':  QuerySelectField(
                            'Department',
                            query_func=department_query,
                            widget=Select2Widget(extra_classes="readonly")
                        )
    }
```

Forms - Custom validation rules

Contribute with your own additional form validations rules. Remember FAB will automatically validate any field that is defined on the database with *Not Null* (Required) or Unique constraints:

```
class MyView(ModelView):
    datamodel = SQLAInterface(MyModel)
    validators_columns = {
        'my_field1':[EqualTo('my_field2', message=gettext('fields must match'))]
    }
```

Forms - Custom query on related fields

You can create a custom query filter for all related columns like this:

```
from flask_appbuilder.models.sqla.filters import FilterStartsWith


class ContactModelView(ModelView):
    datamodel = SQLAInterface(Contact)
    add_form_query_rel_fields = {'group': [['name', FilterStartsWith, 'W']]}
```

This will filter list combo on Contact's model related with ContactGroup model. The combo will be filtered with entries that start with W. You can define individual filters for add,edit and search using **add_form_quey_rel_fields**, **edit_form_query_rel_fields**, **search_form_query_rel_fields** respectively. Take a look at the API Reference If you want to filter multiple related fields just add new keys to the dictionary, remember you can add multiple filters for each field also, take a look at the *base_filter* property:

```
class ContactModelView(ModelView):
    datamodel = SQLAInterface(Contact)
    add_form_query_rel_fields = {
        'group': [['name', FilterStartsWith, 'W']],
        'gender': [['name', FilterStartsWith, 'M']]
    }
```

Forms - Related fields

To use AJAX select2 (combo) fields and make use of the REST API, by default all fields are previously populated on the server. Here's a simple example:

```
class ContactModelView(ModelView):
    datamodel = SQLAInterface(Contact)

    add_form_extra_fields = {
        'contact_group': AJAXSelectField(
                            'contact_group',
                            description='This will be populated with AJAX',
                            datamodel=datamodel,
                            col_name='contact_group',
                            widget=Select2AJAXWidget(endpoint='/contactmodelview/api/
column/add/contact_group')
                           ),
    }
```

Even better you can (since 1.7.0) create related select2 fields, if you have two (or more) relationships that are related them self's, like a group and subgroup on a contact, when the user selects the group the second select2 combo will be populated with the subgroup values that belong to the group. Extending the previous example:

```
class ContactModelView(ModelView):
    datamodel = SQLAInterface(Contact)

    add_form_extra_fields = {
                    'contact_group': AJAXSelectField('contact_group',
                    description='This will be populated with AJAX',
                    datamodel=datamodel,
                    col_name='contact_group',
                    widget=Select2AJAXWidget(endpoint='/contactmodelview/api/column/
add/contact_group')),

                    'contact_sub_group': AJAXSelectField('Extra Field2',
                    description='Extra Field description',
                    datamodel=datamodel,
                    col_name='contact_sub_group',
                    widget=Select2SlaveAJAXWidget(master_id='contact_group',
                    endpoint='/contactmodelview/api/column/add/contact_sub_group?
_flt_0_contact_group_id={{ID}}'))
                    }
```

So as seen before add_form_extra_fields is a dictionary that expects keys as column names and values as WTF Fields.

AJAXSelectField is expecting the following parameters for the constructor: - label: A label for the column. - description: A description to render on the form. - datamodel: SQLAlchemy initialized with the model. - col_name: The column name. - widget: Use Select2AJAXWidget (for the master) and Select2SlaveAJAXWidget for the slave. - endpoint: The REST API that will be used to populate the select2.

You have 3 endpoint's API that will return data ready to use by this fields:

- /<YOUR MODELVIEW NAME>/api/column/add|edit/<COLUMN NAME> : you can append query string's to filter data. This will return all values of the related column on the model.
- /<YOUR MODELVIEW NAME>/api/readvalues: This will return all values on the modelview prepared to be used on a select2.

Customizing

You can override and customize almost everything on the UI, or use different templates and widgets already on the framework.

Even better you can develop your own widgets or templates and contribute to the project.

Changing themes

F.A.B comes with bootswatch themes ready to use, to change bootstrap default theme just change the APP_THEME key's value.

- On config.py (from flask-appbuilder-skeleton), using spacelab theme:

```
APP_THEME = "spacelab.css"
```

- Not using a config.py on your applications, set the key like this:

```
app.config['APP_THEME'] = "spacelab.css"
```

You can choose from the following themes

Changing the index

The index can be easily overridden by your own. You must develop your template, then define it in a IndexView and pass it to AppBuilder

The default index template is very simple, you can create your own like this:

1 - Develop your template (on your <PROJECT_NAME>/app/templates/my_index.html):

```
{% extends "appbuilder/base.html" %}
{% block content %}
<div class="jumbotron">
  <div class="container">
    <h1>{{_("My App on F.A.B.")}}</h1>
    <p>{{_("My first app using F.A.B, bla, bla, bla")}}</p>
  </div>
</div>
{% endblock %}
```

What happened here? We should always extend from “appbuilder/base.html” this is the base template that will include all CSS’s, Javascripts, and construct the menu based on the user’s security definition.

Next we will override the “content” block, we could override other areas like CSS, extend CSS, Javascript or extend javascript. We can even override the base.html completely

I’ve presented the text on the content like:

```
{{_("text to be translated")}}
```

So that we can use Babel to translate our index text

2 - Define an IndexView

Define a special and simple view inherit from IndexView, don't define this view on views.py, put it on a separate file like index.py:

```
from flask_appbuilder import IndexView


class MyIndexView(IndexView):
    index_template = 'my_index.html'
```

3 - Tell F.A.B to use your index view, when initializing AppBuilder:

```
from app.index import MyIndexView

app = Flask(__name__)
app.config.from_object('config')
db = SQLA(app)
appbuilder = AppBuilder(app, db.session, indexview=MyIndexView)
```

Of course you can use a more complex index view, you can use any kind of view (BaseView childs), you can even change relative url path to whatever you want, remember to set **default_view** to your function.

You can override **IndexView** index function to display a different view if a user is logged in or not.

Changing the Footer

The default footer can be easily changed by your own. You must develop your template, to override the existing one.

Develop your jinja2 template and place it on the following relative path to override the F.A.B footer.

./your_root_project_path/app/templates/appbuilder/footer.html

Actually you can override any given F.A.B. template.

Changing Menu Construction

You can change the way the menu is constructed adding your own links, separators and

changing the navbar reverse property.

By default menu is constructed based on your classes and in a reversed navbar. Let's take a quick look on how to easily change this

- Change the reversed navbar style, on AppBuilder initialization:

```
appbuilder = AppBuilder(app, db, menu=Menu(reverse=False))
```

- Add your own menu links, on a default reversed navbar:

```
# Register a view, rendering a top menu without icon
appbuilder.add_view(MyModelView, "My View")
# Register a view, a submenu "Other View" from "Other" with a phone icon
appbuilder.add_view(MyOtherModelView, "Other View", icon='fa-phone',
category="Others")
# Register a view, with label for babel support (internationalization), setup an
icon for the category.
appbuilder.add_view(MyOtherModelView, "Other View", icon='fa-phone',
label=lazy_gettext('Other View'),
                category="Others", category_label=lazy_gettext('Other'),
category_label='fa-envelope')
# Add a link
appbuilder.add_link("google", href="www.google.com", icon = "fa-google-plus")
```

- Add separators:

```
# Register a view, rendering a top menu without icon
appbuilder.add_view(MyModelView1, "My View 1", category="My Views")
appbuilder.add_view(MyModelView2, "My View 2", category="My Views")
appbuilder.add_separator("My Views")
appbuilder.add_view(MyModelView3, "My View 3", category="My Views")
```

Using *label* argument is optional for view name or category, but it's advised for internationalization, if you use it with Babel's *lazy_gettext* function it will automate translation's extraction.

Category icon and label can be setup only for the first time. Internally F.A.B. has already stored it, next references will be made by name.

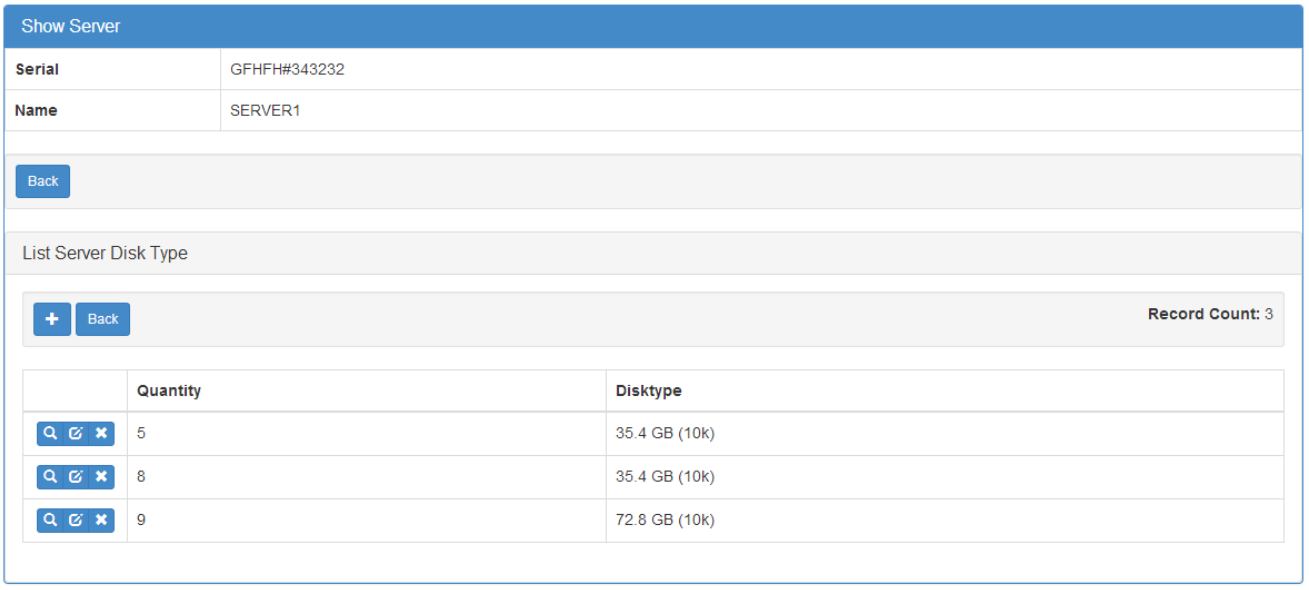
Changing Widgets and Templates

F.A.B. has a collection of widgets to change your views presentation, you can create your own and override, or (even better) create them and contribute to the project on git.

All views have templates that will display widgets in a certain layout. For example, on the edit or show view, you can display the related list (from *related_views*) on the same page, or as tab (default).:

```
class ServerDiskTypeModelView(ModelView):
    datamodel = SQLAInterface(ServerDiskType)
    list_columns = ['quantity', 'disktype']


class ServerModelView(ModelView):
    datamodel = SQLAInterface(Server)
    related_views = [ServerDiskTypeModelView]

    show_template = 'appbuilder/general/model/show_cascade.html'
    edit_template = 'appbuilder/general/model/edit_cascade.html'

    list_columns = ['name', 'serial']
    order_columns = ['name', 'serial']
    search_columns = ['name', 'serial']
```

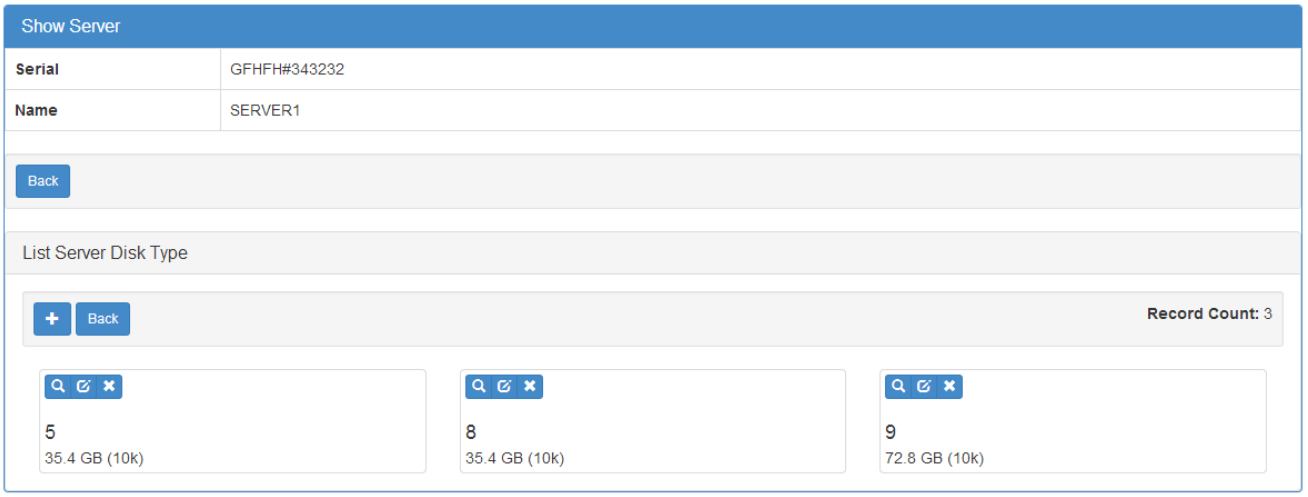
The above example will override the show and edit templates that will change the related lists layout presentation.



If you want to change the above example, and change the way the server disks are displayed has a list just use the available widgets:

```
class ServerDiskTypeModelView(ModelView):
    datamodel = SQLAInterface(ServerDiskType)
    list_columns = ['quantity', 'disktype']
    list_widget = ListBlock


class ServerModelView(ModelView):
    datamodel = SQLAInterface(Server)
    related_views = [ServerDiskTypeModelView]

    show_template = 'appbuilder/general/model/show_cascade.html'
    edit_template = 'appbuilder/general/model/edit_cascade.html'

    list_columns = ['name', 'serial']
    order_columns = ['name', 'serial']
    search_columns = ['name', 'serial']
```

We have overridden the list_widget property with the ListBlock Class. This will look like this.



You have the following widgets already available

- ListWidget (default)
- ListItem
- ListThumbnail
- ListBlock

If you want to develop your own widgets just look at the code

Read the docs for developing your own template widgets Templates

Implement your own and then create a very simple class like this one:

```
class MyWidgetList(ListWidget):
    template = '/widgets/my_widget_list.html'
```

Change Default View Behaviour

If you want to have Add, edit and list on the same page, this can be done. This could be very helpful on master/detail lists (inline) on views based on tables with very few columns.

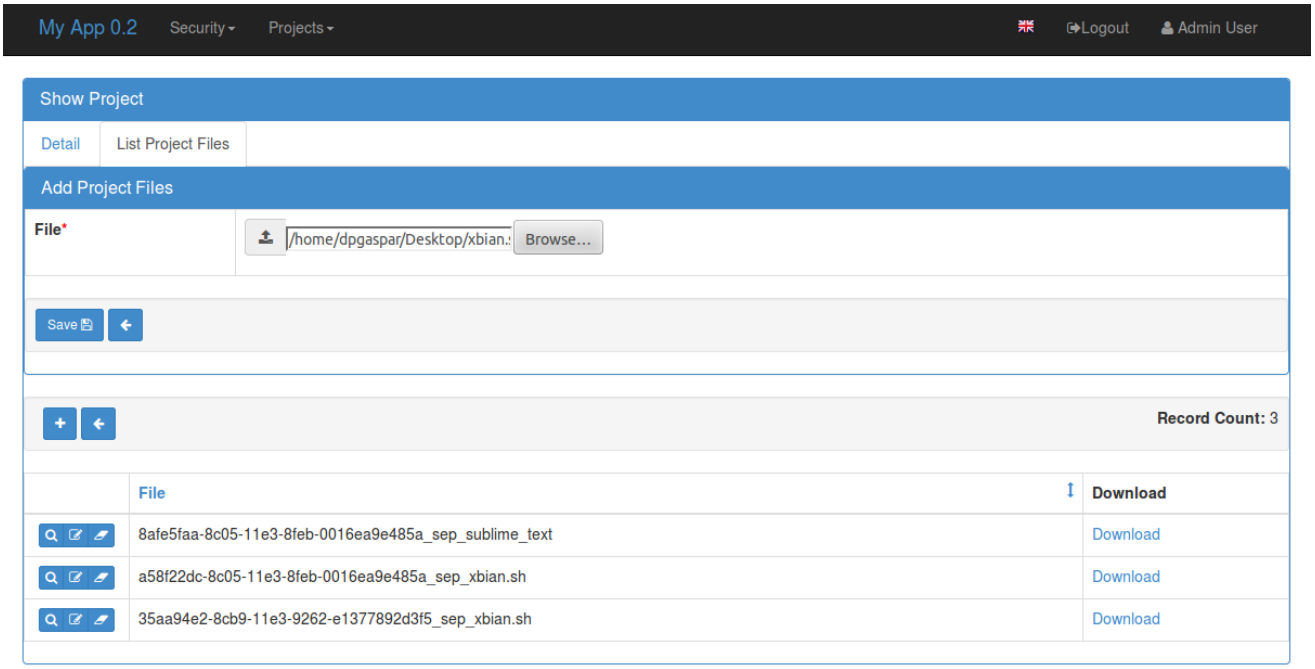
All you have to do is to mix *CompactCRUDMixin* class with the *ModelView* class.:

```
from flask_appbuilder.models.sqla.interface import SQLAInterface
from flask_appbuilder.views import ModelView, CompactCRUDMixin

from . import appbuilder
from .models import Project, ProjectFiles


class MyInlineView(CompactCRUDMixin, ModelView):
    datamodel = SQLAInterface(MyInlineTable)

class MyView(ModelView):
    datamodel = SQLAInterface(MyViewTable)
    related_views = [MyInlineView]

appbuilder.add_view(MyView, "List My View",icon = "fa-table", category = "My Views")
appbuilder.add_view_no_menu(MyInlineView)
```

Notice the class mixin, with this configuration you will have a *Master View* with the inline view *MyInlineView* where you can Add and Edit on the same page.

Of course you could use the mixin on *MyView* also, use it only on ModelView classes.

Take a look at the example: https://github.com/dpgaspar/Flask-appBuilder/tree/master/examples/quickfiles



Next we will take a look at a different view behaviour. A master detail style view, master is a view associated with a database table that is linked to the detail view.

Let's assume our quick how to example, a simple contacts applications. We have *Contact*

table related with *Group* table.

So we are using master detail view, first we will define the detail view (this view can be customized like the examples above):

```
class ContactModelView(ModelView):
    datamodel = SQLAInterface(Contact)
```

Then we define the master detail view, where master is the one side of the 1-N relation:

```
class GroupMasterView(MasterDetailView):
    datamodel = SQLAInterface(Group)
    related_views = [ContactModelView]
```

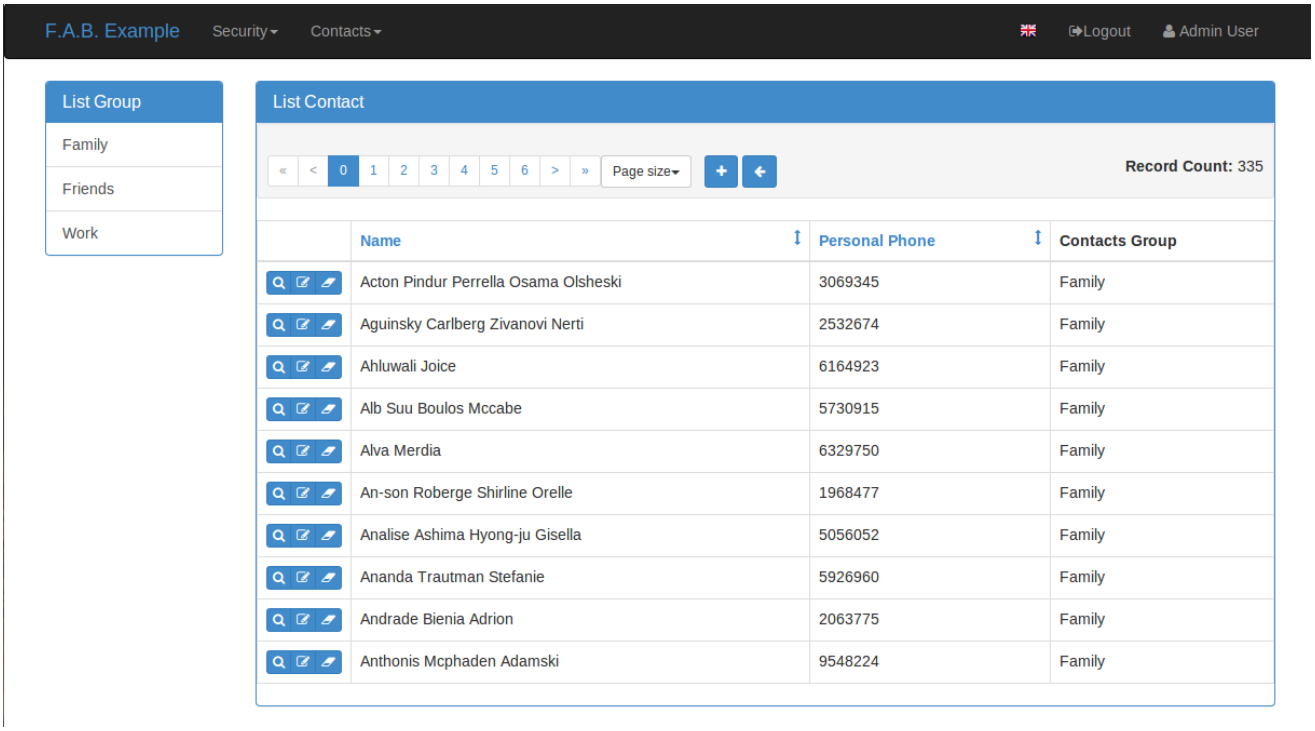
Remember you can use charts has related views, you can use it like this:

```
class ContactTimeChartView(TimeChartView):
    datamodel = SQLAInterface(Contact)
    chart_title = 'Grouped Birth contacts'
    chart_type = 'AreaChart'
    label_columns = ContactModelView.label_columns
    group_by_columns = ['birthday']


class GroupMasterView(MasterDetailView):
    datamodel = SQLAInterface(Group)
    related_views = [ContactModelView, ContactTimeChartView]
```

This will show a left side menu with the *groups* and a right side list with contacts, and a time chart with the number of birthdays during time by the selected group.

Finally register everything:

```
// if Using the above example with related chart
appbuilder.add_view_no_menu(ContactTimeChartView)

appbuilder.add_view(GroupMasterView, "List Groups", icon="fa-folder-open-o",
category="Contacts")
appbuilder.add_separator("Contacts")
appbuilder.add_view(ContactModelView, "List Contacts", icon="fa-envelope",
category="Contacts")
```



Templates

F.A.B. uses jinja2, all the framework templates can be overridden entirely or partially. This way you can add your own html on jinja2 templates. This can be done before or after defined blocks on the page, without the need of developing a template from scratch because you just want to add small changes on it. Next is a quick description on how you can do this

CSS and Javascript

To add your own CSS's or javascript application wide. You will need to tell the framework to use your own base *jinja2* template, this template is extended by all the templates. It's very simple: first create your own template in your **templates** directory.

On a simple application structure create *mybase.html* (or whatever name you want):

```
<my_project>
    <app>
        __init__.py
        models.py
        views.py
        <templates>
            **mybase.html**
```

Then on mybase.html add your js files and css files, use **head_css** for css's and **head_js** for javascript. These are *jinja2* blocks, F.A.B. uses them so that you can override or extend critical parts of the default templates, making it easy to change the UI, without having to develop your own from scratch:

```
{% extends 'appbuilder/baselayout.html' %}

{% block head_css %}
    {{ super() }}
    <link rel="stylesheet" href="{{url_for('static',filename='css/
your_css_file.css')}}"></link>
{% endblock %}

{% block head_js %}
    {{ super() }}
    <script src="{{url_for('static',filename='js/your_js_file.js')}}"></script>
{% endblock %}
```

If you want to import your javascript files at the end of the templates use **tail_js**:

```
{% block tail_js %}
    {{ super() }}
    <script src="{{url_for('static',filename='js/your_js_file.js')}}"></script>
{% endblock %}
```

Finally tell the framework to use it, instead of the default base template, when initializing on __init__.py use the *base_template* parameter:

```
appbuilder = AppBuilder(app, db.session, base_template='mybase.html')
```

You have an example that changes the way the menu is displayed on examples

This main structure of jinja2 on the baselayout template is:

```
{% block head_meta %}
    ... HTML Meta
{% endblock %}
{% block head_css %}
    ... CSS imports (bootstrap, fontAwesome, select2, fab specific etc...
{% endblock %}
{% block head_js %}
    ... JS imports (JQuery, fab specific)
{% endblock %}
{% block body %}
    {% block navbar %}
        ... The navigation bar (Menu)
    {% endblock %}
     {% block messages %}
        ... Where the flask flash messages are shown ("Added row", etc)
      {% endblock %}
      {% block content %}
        ... All the content goes here, forms, lists, index, charts etc..
      {% endblock %}
    {% block footer %}
        ... The footer, by default its almost empty.
    {% endblock %}
{% block tail_js %}
{% endblock %}
```

Navigation Bar

There is also the possibility to customize the navigation bar. You can completely override it, or just partially.

To completely override the navigation bar, implement your own base layout as described earlier and then extend the existing one and override the **navbar** block.

As an example, let's say you created your own base layout named **my_layout.html** in your **templates** folder:

```
{% extends 'appbuilder/baselayout.html' %}

{% block navbar %}
    <div class="navbar" role="navigation">
       <div class="container">
            <div class="navbar-header">
                    ....
            </div>
            <div class="navbar-collapse collapse">
                    ....
            </div>
       </div>
    </div>
{% endblock %}
```

Remember to tell Flask-Appbuilder to use your layout instead (previous chapter).

The best way to just override the navbar partially is to override the existing templates from the framework. You can always do this with any template. There are two good candidates for this:

`/templates/appbuilder/navbar_menu.html`

This will render the navbar menus.

`/templates/appbuilder/navbar_right.html`

This will render the right part of the navigation bar (locale and user).

List Templates

Using the contacts app example, we are going to see how to override or insert jinja2 on specific sections of F.A.B. list template. Remember that the framework uses templates with generated widgets, this widgets are big widgets, because they render entire sections of a page. On list's of records you will have two widgets: the search widget and the list widget. You will have a template with the following sections, where you can add your template sections over, before and after each block:

- `List template`
    - `Block "content"`
        - `Block "list_search"`
            - Search Widget
        - End Block "list_search"
        - `Block "list_list"`
            - List Widget
        - End Block "list_list"
    - End Block "content"

To insert your template section over a block, say "list_search" just do:

```
{% extends "appbuilder/general/model/list.html" %}

    {% block list_search scoped %}
        This Text will replace the search widget
    {% endblock %}
```

To insert your template section after a block do:

```
{% extends "appbuilder/general/model/list.html" %}

    {% block list_search scoped %}
        {{ super() }}
        This Text will show after the search widget
    {% endblock %}
```

I guess you get the general ideal, make use of {{ super() }} to render the block's original content. To use your templates override **list_template** to your templates relative path, on your ModelView's declaration.

If you have your template on ./your_project/app/templates/list_contacts.html

```
class ContactModelView(ModelView):
    datamodel = SQLAInterface(Contact)
    list_template = 'list_contacts.html'
```

In your template you can do something like this

```
{% extends "appbuilder/general/model/list.html" %}

{% block content %}
    Text on top of the page
    {{ super() }}
    {% block list_search scoped %}
        Text before the search section
        {{ super() }}
    {% endblock %}

    {% block list_list scoped %}
        Text before the list
        {{ super() }}
    {% endblock %}
{% endblock %}
```

Add Templates

In this section we will see how to override the add template form. You will have only one widget, the add form widget. So you will have a template with the following sections, where you can add your template sections over, before and after each block:

- Add template
    - Block "content"
        - Block "add_form"
            - Add Widget

        - End Block “add_form”

    - End Block “content”

To insert your template section before a block, say “add_form” just create your own template like this:

```
{% extends "appbuilder/general/model/add.html" %}

    {% block add_form %}
        This Text is before the add form widget
        {{ super() }}
    {% endblock %}
```

To use your template define you ModelView with **add_template** declaration to your templates relative path

If you have your template in ./your_project/app/templates/add_contacts.html

```
class ContactModelView(ModelView):
    datamodel = SQLAInterface(Contact)

    add_template = 'add_contacts.html'
```

Edit Templates

In this section we will see how to override the edit template form. You will have only one widget the edit form widget, so you will have a template with the following sections, where you can add your template sections over, before and after each block:

- Add template
    - Block “content”
        - Block “edit_form”
            - Edit Widget
        - End Block “edit_form”
    - End Block “content”

To insert your template section before the edit widget, just create your own template like this:

```
{% extends "appbuilder/general/model/edit.html" %}

    {% block add_form %}
        This Text is before the add form widget
        {{ super() }}
    {% endblock %}
```

To use your template define you ModelView with **edit_template** declaration to your templates relative path

If you have your template in ./your_project/app/templates/edit_contacts.html

```
class ContactModelView(ModelView):
    datamodel = SQLAInterface(Contact)

    edit_template = 'edit_contacts.html'
```

Show Templates

In this section we will see how to override the show template. You will have only one widget the show widget, so you will have a template with the following sections, where you can add your template sections over, before and after each block:

- Show template
    - Block “content”
        - Block “show_form”
            - Show Widget
        - End Block “show_form”
    - End Block “content”

To insert your template section before a block, say “show_form” just create your own template like this:

```
{% extends "appbuilder/general/model/show.html" %}

    {% block show_form %}
        This Text is before the show widget
        {{ super() }}
    {% endblock %}
```

To use your template define you ModelView with **show_template** declaration to your templates relative path

If you have your template in ./your_project/app/templates/show_contacts.html

```
class ContactModelView(ModelView):
    datamodel = SQLAInterface(Contact)

    show_template = 'show_contacts.html'
```

Edit/Show Cascade Templates

In cascade templates for related views the above rules apply, but you can use an extra block to insert your template code before, after or over the related view list widget. For show cascade templates you have the following structure:

- Show template
    - Block “content”
        - Block “show_form”
            - Show Widget
        - End Block “show_form”
        - Block “related_views”
            - Related Views Widgets
        - End Block “related_views”
    - End Block “content”

Widgets

Widgets are reusable, you can and should implement your own. Widgets are a special kind of jinja2 templates. They will be contained inside a python class, and rendered on a jinja2 template. So **list_template**, **add_template**, **edit_template**, **show_template** will work like layouts with widgets.

To create your own widgets follow the next recipe.

Example 1: Custom list widget

- Make your own widget template, we are going to create a very simple list widget. since

version 1.4.1 list widgets extend **base_list.html** this will make your life simpler, this base template declares the following blocks you should use, when implementing your own widget for lists:

```
{% block list_header scoped %}
    This is where the list controls are rendered, extend it to *inject* your own
controls.
{% endblock %}

{% block begin_content scoped %}
    Area next to the controls
{% endblock %}

{% block begin_loop_header scoped %}
    Nice place to render your list headers.
{% endblock %}

{% block begin_loop_values %}
    Make your loop and render the list itself.
{% endblock %}
```

Let's make a simple example:

```
{% import 'appbuilder/general/lib.html' as lib %}
{% extends 'appbuilder/general/widgets/base_list.html' %}

{% block list_header %}
   {{ super() }}
   <a href="url_for('Class.method for my control')" class="btn btn-sm btn-primary"
        <i class="fa fa-rocket"></i>
   </a>
{% endblock %}

{% block begin_loop_values %}
    {% for item in value_columns %}
        {% set pk = pks[loop.index-1] %}
        {% if actions %}
            <input id="{{pk}}" class="action_check" name="rowid" value="{{pk}}"
type="checkbox">
        {% endif %}
        {% if can_show or can_edit or can_delete %}
            {{ lib.btn_crud(can_show, can_edit, can_delete, pk, modelview_name,
filters) }}
        {% endif %}
        </div>

        {% for value in include_columns %}
            <p {{ item[value]|safe }}</p>
        {% endfor %}
    {% endfor %}
{% endblock %}
```

This example will just use two blocks **list_header** and **begin_loop_values**. On **list_header** we are rendering an extra button/link to a class method. Notice that first we call **super()** so that

our control will be placed next to pagination, add button and back button

> **Note**
>
> If you just want to add a new control next to the list controls and keep everything else from the predefined widget. extend your widget from {% extends 'appbuilder/general/widgets/list.html' %} and just implement **list_header** the way it's done in this example.

Next we will render the values of the list, so we will override the **begin_loop_values** block. Widgets have the following jinja2 vars that you should use:

- can_show: Boolean, if the user has access to the show view.
- can_edit: Boolean, if the user has access to the edit view.
- can_add: Boolean, if the user has access to the add view.
- can_delete: Boolean, if the user has access to delete records.
- value_columns: A list of Dicts with column names as keys and record values as values.
- include_columns: A list with columns to include in the list, and their order.
- order_columns: A list with the columns that can be ordered.
- pks: A list of primary key values.
- actions: A list of declared actions.
- modelview_name: The name of the ModelView class responsible for controlling this template.

Save your widget template in your templates folder. I advise you to create a subfolder named *widgets*. So in our example we will keep our template in */templates/widgets/my_list.html*.

- Next we must create our python class to contain our widget. In your **app** folder create a file named widgets.py:

```
from flask_appbuilder.widgets import ListWidget


class MyListWidget(ListWidget):
     template = 'widgets/my_list.html'
```

- Finally use your new widget on your views:

```
class MyModelView(ModelView):
    datamodel = SQLAInterface(MyModel)
    list_widget = MyListWidget
```

Example 2: Custom show widget

By default, Actions related buttons are located at the end of the detail page. If you now have a longer detail page, it can be cumbersome for your users to have to go to the bottom of the page to perform the actions. Let's just add a second set of buttons to the top of the page.

To do this, do the following (similar to the steps above):

- Create a template override file *<module>/templates/widgets/my_show.html*:

```
{% extends "appbuilder/general/widgets/show.html" %}
{% block columns %}
    <div class="well well-sm">
        {{ lib.render_action_links(actions, pk, modelview_name) }}
        {{ lib.lnk_back() }}
    </div>
    {{ super() }}
{% endblock %}
```

  Please note that we have just overridden the jinja block named `columns`, prepended our own HTML code and then called the original block (using `super()`).
- Create the custom ShowWidget class:

```
from flask_appbuilder.widgets import ShowWidget

class MyShowWidget(ShowWidget):
    template = 'widgets/show.html'
```

- And finally refer to your widget in your view:

```
class MyModelView(ModelView):
    datamodel = SQLAInterface(MyModel) show_widget = MyShowWidget
```

Other widget types

Flask-AppBuilder has already some widgets that you can choose from, try them out:

- ListWidget - The default for lists.
- ListLinkWidget - The default for lists.
- ListThumbnail - For lists, nice to use with photos.
- ListItem - Very simple list of items.
- ListBlock - For lists, similar to thumbnail.
- FormWidget - For add and edit.
- FormHorizontalWidget - For add and edit.
- FormInlineWidget - For add and edit

- ShowWidget - For show view.
- ShowBlockWidget - For show view.
- ShowVerticalWidget - For show view.

Take a look at the widgets example.

Library Functions

F.A.B. has the following library functions that you can use to render bootstrap 3 components easily. Using them will ease your productivity and help you introduce new html that shares the same look and feel of the framework.

- Panel component:

```
{{ lib.panel_begin("Panel's Title") }}
    Your html goes here
{{ lib.panel_end() }}
```

- Accordion (pass your view's name, or something that will serve as an id):

```
{% call lib.accordion_tag(view.__class__.__name__,"Accordion Title", False) %}
    Your HTML goes here
{% endcall %}
```

AddOn development

Using AddOn's with the framework it is a great way to develop your application and make public open source contributions to the community.

With it you can use a more modular design on your application, you can add functionality, views and models that you can build independently and install or uninstall (using different versions).

To start building your own AddOn's you can use issue the following command:

```
$ flask fab create-addon --name first
```

Your addon name will be prefixed by 'fab_addon_' so this addon would be called **fab_addon_first**. The create-addon will download a default skeleton addon for you to start more easily to code (much like the create-app command).

The structure of the default base addon:

- `<fab_addon_first>`
    - setup.py: Setup installer use it to install your addon, or upload it to Pypi when your ready to release.
    - config.py: Used internaly by setup.py, this will make your setup more generic.
    - `<fab_addon_first>`
        - __init__.py: empty
        - models.py: Declare your addon's models (if any) just like on a normal app.
        - views.py: Declare your addon's views but don't register them here.
        - manager.py: This is where your addon manager will reside, It's your manager that will be imported by appbuilder.
        - version.py: Declare your addon version here, write your name (author), a small description and your email.

Your can use your addon much like a regular F.A.B. app, just don't instantiate anything (appbuilder, flask, SQLAlchemy etc...) notice, __init__.py module is empty. So if you or anyone (if you upload your addon to pypi or make it public somewhere like github) want to use your addon they just have to install it and declare it using the ADDON_MANAGERS key, this key is a list of addon manager's.

So what is a manager? Manager is a class you declare that subclasses appbuilder BaseManager, and you have 4 important methods you can override, there are:

`__init__(self, appbuilder)`

Manager's constructor. Good place to check for your addon's specific keys. For custom configuration

`register_views(self)`

Use it to register all your views and setup a menu for them (if you want to).

`pre_processs`

Will be called before register_views. Good place to insert data into your models for example.

`post_process`

Will be called after register_views.

A very simple manager would look something like this:

```
import logging

from flask_appbuilder.basemanager import BaseManager
from flask_babel import lazy_gettext as _

from .model import MyModel
from .views import FirstModelView1


log = logging.getLogger(__name__)


class FirstAddOnManager(BaseManager):

    def __init__(self, appbuilder):
        """
             Use the constructor to setup any config keys specific for your app.
        """
        super(FirstAddOnManager, self).__init__(appbuilder)

    def register_views(self):
        """
            This method is called by AppBuilder when initializing, use it to add you
views
        """
        self.appbuilder.add_view(FirstModelView1, "First View1",icon = "fa-
user",category = "First AddOn")

    def pre_process(self):
        stuff = self.appbuilder.get_session.query(MyModel).filter(name ==
'something').all()
        # process stuff

    def post_process(self):
        pass
```

How can you or someone use your AddOn? On the app config.py add this key:

```
ADDON_MANAGERS = ['fab_addon_first.manager.FirstAddOnManager']
```

And thats it.

I've just added a simple audit modelViews's addon to start contributions and to serve as an example.

you can install it using:

```
$ pip install fab_addon_audit
```

The source code is pretty simple, use it as an example to write your own:

https://github.com/dpgaspar/fab_addon_audit

Generic Data Sources

This feature is still beta, but you can already use it, it allows you to use alternative/generic datasources. With it you can use python libraries, systems commands or whatever with the framework as if they were SQLAlchemy models.

PS Command example

Already on the framework, and intended to be an example, is a data source that holds the output from the linux 'ps -ef' command, and shows it as if it were a SQLA model.

Your own generic data source must subclass from **GenericSession** and implement at least the **all** method

The **GenericSession** mimics a subset of SQLA **Session** class and it's query feature, so if you override the all method you will implement the data generation at it's heart.

On our example you must first define the **Model** you will represent:

```
from flask_appbuilder.models.generic import GenericModel, GenericSession, GenericColumn

class PSModel(GenericModel):
    UID = GenericColumn(str)
    PID = GenericColumn(int, primary_key=True)
    PPID = GenericColumn(int)
    C = GenericColumn(int)
    STIME = GenericColumn(str)
    TTY = GenericColumn(str)
    TIME = GenericColumn(str)
    CMD = GenericColumn(str)
```

As you can see, we are subclassing from **GenericModel** and use **GenericColumn** much like SQLAlchemy. except type are really python types. No type obligation is implemented, but you should respect it when implementing your own data generation

For your data generation, and regarding our example:

```
class PSSession(GenericSession):
    regexp = "(\w+) +(\w+) +(\w+) +(\w+) +(\w+:\w+|\w+) (\?|tty\w+) +(\w+:\w+:\w+) +(.
+)\n"

    def _add_object(self, line):
        import re

        group = re.findall(self.regexp, line)
        if group:
            model = PSModel()
            model.UID = group[0][0]
            model.PID = int(group[0][1])
            model.PPID = int(group[0][2])
            model.C = int(group[0][3])
            model.STIME = group[0][4]
            model.TTY = group[0][5]
            model.TIME = group[0][6]
            model.CMD = group[0][7]
            self.add(model)

    def get(self, pk):
        self.delete_all(PSModel())
        out = os.popen('ps -p {0} -f'.format(pk))
        for line in out.readlines():
            self._add_object(line)
        return super(PSSession, self).get(pk)


    def all(self):
        self.delete_all(PSModel())
        out = os.popen('ps -ef')
        for line in out.readlines():
            self._add_object(line)
        return super(PSSession, self).all()
```

So each time the framework queries the data source, it will **delete_all** records, and call ‘ps -ef’ for a query all records, or ‘ps -p <PID>’ for a single record.

The **GenericSession** class will implement by itself the Filters and order by methods to be applied prior to your *all* method. So that everything works much like SQLAlchemy.

I implemented this feature out of the necessity of representing LDAP queries, but of course you can use it to wherever your imagination/necessity drives you.

Finally you can use it on the framework like this:

```
sess = PSSession()


class PSView(ModelView):
    datamodel = GenericInterface(PSModel, sess)
    base_permissions = ['can_list', 'can_show']
    list_columns = ['UID', 'C', 'CMD', 'TIME']
    search_columns = ['UID', 'C', 'CMD']
```

And then register it like a normal ModelView.

You can try this example on *quickhowto2 example <https://github.com/dpgaspar/Flask-AppBuilder/tree/master/examples/quickhowto2>*

I know this is still a short doc for such a complex feature, any doubts you may have just open an issue.

Multiple Databases

Because you can use Flask-SQLAlchemy (using the framework SQLA class) multiple databases is supported.

You can configure them the following way, first setup config.py:

```
SQLALCHEMY_DATABASE_URI = 'sqlite:///' + os.path.join(basedir, 'app.db')

SQLALCHEMY_BINDS = {
    'my_sql1': 'mysql://root:password@localhost/quickhowto',
    'my_sql2': 'mysql://root:password@externalserver.domain.com/quickhowto2'
}
```

The **SQLALCHEMY_DATABASE_URI** is the default connection this is where the framework's security tables will be created. The **SQLALCHEMY_BINDS** are the extra binds.

Now you can configure which models reside on which database using the __bind_key__ property

```
class Model1(Model):
    __bind_key__ = 'my_sql1'
    id = Column(Integer, primary_key=True)
    name =  Column(String(150), unique = True, nullable=False)


class Model2(Model):
    __bind_key__ = 'my_sql2'
    id = Column(Integer, primary_key=True)
    name =  Column(String(150), unique = True, nullable=False)


class Model3(Model):
    id = Column(Integer, primary_key=True)
    name =  Column(String(150), unique = True, nullable=False)
```

On this example:

- Model1 will be on the local MySql instance with db 'quickhowto'.

- Model2 will be on the externalserver.domain.com MySql instance with db 'quickhowto2'.
- Model3 will be on the default connection using sqlite.

i18n Translations

Introduction

F.A.B. has support for 15 languages (planning for some more):

- Chinese
- Dutch
- English
- French
- German
- Japanese
- Polish
- Portuguese
- Portuguese Brazil
- Russian
- Slovenian
- Spanish
- Greek
- Korean
- Italian

This means that all messages, builtin on the framework are translated to these languages.

You can add your own translations for your application, using Flask-Babel.

You can add your own translations, and your own language support. Take a look at Flask-Babel for setup an babel initial configuration.

Initial Configuration

On you projects root create a directory named babel, then create and edit a file named babel.cfg with the following content (this configuration is already made on the base skeleton application):

```
[python: **.py]
[jinja2: **/templates/**.html]
encoding = utf-8
```

First, create your translations, for example to portuguese, spanish and german, execute on you projects root:

```
pybabel init -i ./babel/messages.pot -d app/translations -l pt
pybabel init -i ./babel/messages.pot -d app/translations -l es
pybabel init -i ./babel/messages.pot -d app/translations -l de
```

Next extract your strings to be translated, execute on you projects root:

```
$ flask fab babel-extract
```

If you want to, or if you're using a version prior to 1.3.0 you can use:

```
pybabel extract -F ./babel/babel.cfg -k lazy_gettext -o ./babel/messages.pot .
```

Quick How to

Let's work with the contacts application example, so you want to add translations for the menus "List Groups" and "List Contacts".

```
from flask_babel import lazy_gettext as _

class GroupModelView(ModelView):
    datamodel = SQLAInterface(ContactGroup)
    related_views = [ContactModelView]
    label_columns = {'name':_('Name')}

genapp.add_view(GroupModelView(), "List Groups",icon = "th-large", label=_('List
Groups'),
                    category = "Contacts", category_icon='fa-envelope',
category_label=_('Contacts'))
genapp.add_view(ContactModelView(), "List Contacts",icon = "earphone", label=_('List
Contacts'),
                    category = "Contacts")
```

1 - Run the extraction, from the root directory of your project:

```
$ flask fab babel-extract
```

If you want to, or if you're using a version prior to 1.3.0 you can use:

```
pybabel extract -F ./babel/babel.cfg -k lazy_gettext -o ./babel/messages.pot .
```

2 - Make your translations

- On app/translations/pt/LC_MESSAGES/messages.po you will find the messages you added to translate:

```
msgid "Name"
msgstr ""

msgid "Contacts"
msgstr ""

msgid "List Groups"
msgstr ""

msgid "List Contacts"
msgstr ""
```

- Translate them:

```
msgid "Name"
msgstr "Nome"

msgid "Contacts"
msgstr "Contactos"

msgid "List Groups"
msgstr "Lista de Grupos"

msgid "List Contacts"
msgstr "Lista de Contactos"
```

3 - Compile your translations, from the root directory of your project:

```
$ flask fab babel-compile
```

4 - Add your language support to the framework

- On config tell the framework the languages you support. With this you will render a menu with the corresponding country flags. use the config var 'LANGUAGES' with a dict whose first key is a string with the corresponding babel language code, the value is another dict with two keys 'flag' and 'name', with the country flag code, and text to be displayed:

```
LANGUAGES = {
    'en': {'flag':'gb', 'name':'English'},
    'pt': {'flag':'pt', 'name':'Portuguese'}
}
```

And that's it!

Security

Responsible disclosure

We want to keep Flask-AppBuilder safe for everyone. If you've discovered a security vulnerability please report to danielvazgaspar@gmail.com.

Supported Authentication Types

Database

username and password style that is queried from the database to match. Passwords are kept hashed on the database.

Open ID

Uses the user's email field to authenticate on Gmail, Yahoo etc...

LDAP

Authentication against an LDAP server, like Microsoft Active Directory.

REMOTE_USER

Reads the *REMOTE_USER* web server environ var, and verifies if it's authorized with the framework users table. It's the web server responsibility to authenticate the user, useful for intranet sites, when the server (Apache, Nginx) is configured to use kerberos, no need for the user to login with username and password on F.A.B.

OAUTH

Authentication using OAUTH (v1 or v2). You need to install authlib.

Configure the authentication type on config.py, take a look at Base Configuration

The session is preserved and encrypted using Flask-Login, OpenID requires Flask-OpenID.

Authentication Methods

You can choose one from 5 authentication methods. Configure the method to be used on the **config.py** (when using the create-app, or following the proposed app structure). First the configuration imports the constants for the authentication methods:

```
from flask_appbuilder.security.manager import (
    AUTH_DB,
    AUTH_LDAP,
    AUTH_OAUTH,
    AUTH_OID,
    AUTH_REMOTE_USER
)
```

Next you will use the **AUTH_TYPE** key to choose the type:

```
AUTH_TYPE = AUTH_DB
```

Additionally you can customize the name of the builtin roles for Admin and Public accesses:

```
AUTH_ROLE_ADMIN = 'My Admin Role Name'
AUTH_ROLE_PUBLIC = 'My Public Role Name'
```

Finally you can allow users to self register (take a look at the following chapters for further detail):

```
AUTH_USER_REGISTRATION = True
AUTH_USER_REGISTRATION_ROLE = "My Public Role Name"
```

These settings can apply to all the authentication methods. When you create your first admin user using **flask fab** command line, this user will be authenticated using the authentication method defined on your **config.py**.

Authentication: Database

The database authentication type is the most *simple* one, it authenticates users against an username and hashed password field kept on your database.

Administrators can create users with passwords, and users can change their passwords. This

is all done using the UI. (You can override and extend the default UI as we'll see on *Your Custom Security*)

Authentication: OpenID

This authentication method uses Flask-OpenID. All configuration is done on **config.py** using OPENID_PROVIDERS key, just add or remove from the list the providers you want to enable:

```
AUTH_TYPE = AUTH_OID
OPENID_PROVIDERS = [
    { 'name': 'Yahoo', 'url': 'https://me.yahoo.com' },
    { 'name': 'AOL', 'url': 'http://openid.aol.com/<username>' },
    { 'name': 'Flickr', 'url': 'http://www.flickr.com/<username>' },
    { 'name': 'MyOpenID', 'url': 'https://www.myopenid.com' }
]
```

Each list entry is a dict with a readable OpenID name and it's url, if the url needs an username just add it using <username>. The login template for this method will provide a text box for the user to fillout his/her username.

F.A.B. will ask for the 'email' from OpenID, and if this email belongs to some user on your application he/she will login successfully.

Authentication: LDAP

This method will authenticate the user's credentials against an LDAP server.

WARNING: To use LDAP you need to install python-ldap.

For a typical Microsoft AD setup (where all users can preform LDAP searches):

```
AUTH_TYPE = AUTH_LDAP
AUTH_LDAP_SERVER = "ldap://ldap.example.com"
AUTH_LDAP_USE_TLS = False

# registration configs
AUTH_USER_REGISTRATION = True  # allow users who are not already in the FAB DB
AUTH_USER_REGISTRATION_ROLE = "Public"  # this role will be given in addition to any
AUTH_ROLES_MAPPING
AUTH_LDAP_FIRSTNAME_FIELD = "givenName"
AUTH_LDAP_LASTNAME_FIELD = "sn"
AUTH_LDAP_EMAIL_FIELD = "mail"  # if null in LDAP, email is set to: "{username}
@email.notfound"

# bind username (for password validation)
AUTH_LDAP_USERNAME_FORMAT = "uid=%s,ou=users,dc=example,dc=com"  # %s is replaced with
the provided username
# AUTH_LDAP_APPEND_DOMAIN = "example.com"  # bind usernames will look like: {USERNAME}
@example.com

# search configs
AUTH_LDAP_SEARCH = "ou=users,dc=example,dc=com"  # the LDAP search base (if non-empty,
a search will ALWAYS happen)
AUTH_LDAP_UID_FIELD = "uid"  # the username field
```

For a typical OpenLDAP setup (where LDAP searches require a special account):

```
AUTH_TYPE = AUTH_LDAP
AUTH_LDAP_SERVER = "ldap://ldap.example.com"
AUTH_LDAP_USE_TLS = False

# registration configs
AUTH_USER_REGISTRATION = True  # allow users who are not already in the FAB DB
AUTH_USER_REGISTRATION_ROLE = "Public"  # this role will be given in addition to any
AUTH_ROLES_MAPPING
AUTH_LDAP_FIRSTNAME_FIELD = "givenName"
AUTH_LDAP_LASTNAME_FIELD = "sn"
AUTH_LDAP_EMAIL_FIELD = "mail"  # if null in LDAP, email is set to: "{username}
@email.notfound"

# search configs
AUTH_LDAP_SEARCH = "ou=users,dc=example,dc=com"  # the LDAP search base
AUTH_LDAP_UID_FIELD = "uid"  # the username field
AUTH_LDAP_BIND_USER = "uid=admin,ou=users,dc=example,dc=com"  # the special bind
username for search
AUTH_LDAP_BIND_PASSWORD = "admin_password"  # the special bind password for search
```

You can limit the LDAP search scope by configuring:

```
# only allow users with memberOf="cn=myTeam,ou=teams,dc=example,dc=com"
AUTH_LDAP_SEARCH_FILTER = "(memberOf=cn=myTeam,ou=teams,dc=example,dc=com)"
```

You can give FlaskAppBuilder roles based on LDAP roles (note, this requires

AUTH_LDAP_SEARCH to be set):

```
# a mapping from LDAP DN to a list of FAB roles
AUTH_ROLES_MAPPING = {
    "cn=fab_users,ou=groups,dc=example,dc=com": ["User"],
    "cn=fab_admins,ou=groups,dc=example,dc=com": ["Admin"],
}

# the LDAP user attribute which has their role DNs
AUTH_LDAP_GROUP_FIELD = "memberOf"

# if we should replace ALL the user's roles each login, or only on registration
AUTH_ROLES_SYNC_AT_LOGIN = True

# force users to re-auth after 30min of inactivity (to keep roles in sync)
PERMANENT_SESSION_LIFETIME = 1800
```

TLS

For STARTTLS, configure an *ldap://* server and set *AUTH_LDAP_USE_TLS* to *True*:

```
AUTH_LDAP_SERVER = "ldap://ldap.example.com"
AUTH_LDAP_USE_TLS = True
```

For LDAP over TLS (ldaps), configure the server with the *ldaps://* scheme and set *AUTH_LDAP_USE_TLS* to *False*:

```
AUTH_LDAP_SERVER = "ldaps://ldap.example.com"
AUTH_LDAP_USE_TLS = False
```

Additional LDAP/TLS Options, including CA certificate settings and client authentication, can be found in the Base Configuration.

Authentication: OAuth

This method will authenticate the user's credentials against an OAuth provider.

Note

To use OAuth you need to install Python AuthLib.

Specify a list of OAUTH_PROVIDERS in **config.py** that you want to allow for your users:

```
```

```
AUTH_TYPE = AUTH_OAUTH

# registration configs
AUTH_USER_REGISTRATION = True  # allow users who are not already in the FAB DB
AUTH_USER_REGISTRATION_ROLE = "Public"  # this role will be given in addition to any
AUTH_ROLES_MAPPING

# the list of providers which the user can choose from
OAUTH_PROVIDERS = [
    {
        "name": "twitter",
        "icon": "fa-twitter",
        "token_key": "oauth_token",
        "remote_app": {
            "client_id": "TWITTER_KEY",
            "client_secret": "TWITTER_SECRET",
            "api_base_url": "https://api.twitter.com/1.1/",
            "request_token_url": "https://api.twitter.com/oauth/request_token",
            "access_token_url": "https://api.twitter.com/oauth/access_token",
            "authorize_url": "https://api.twitter.com/oauth/authenticate",
        },
    },
    {
        "name": "google",
        "icon": "fa-google",
        "token_key": "access_token",
        "remote_app": {
            "client_id": "GOOGLE_KEY",
            "client_secret": "GOOGLE_SECRET",
            "api_base_url": "https://www.googleapis.com/oauth2/v2/",
            "client_kwargs": {"scope": "email profile"},
            "request_token_url": None,
            "access_token_url": "https://accounts.google.com/o/oauth2/token",
            "authorize_url": "https://accounts.google.com/o/oauth2/auth",
            "jwks_uri": "https://www.googleapis.com/oauth2/v3/certs",
        },
    },
    {
        "name": "openshift",
        "icon": "fa-circle-o",
        "token_key": "access_token",
        "remote_app": {
            "client_id": "system:serviceaccount:mynamespace:mysa",
            "client_secret": "<mysa serviceaccount token here>",
            "api_base_url": "https://openshift.default.svc.cluster.local:443",
            "client_kwargs": {"scope": "user:info"},
            "redirect_uri": "https://myapp-mynamespace.apps.<cluster_domain>",
            "access_token_url": "https://oauth-openshift.apps.<cluster_domain>/oauth/
token",
            "authorize_url": "https://oauth-openshift.apps.<cluster_domain>/oauth/
authorize",
            "token_endpoint_auth_method": "client_secret_post",
        },
    },
    {
        "name": "okta",
        "icon": "fa-circle-o",
        "token_key": "access_token",
        "remote_app": {
            "client_id": "OKTA_KEY",
            "client_secret": "OKTA_SECRET",
            "api_base_url": "https://OKTA_DOMAIN.okta.com/oauth2/v1/",
            "client_kwargs": {"scope": "openid profile email groups"},
```

```
            "access_token_url": "https://OKTA_DOMAIN.okta.com/oauth2/v1/token",
            "authorize_url": "https://OKTA_DOMAIN.okta.com/oauth2/v1/authorize",
            "server_metadata_url": f"https://OKTA_DOMAIN.okta.com/.well-known/openid-
configuration",
        },
    },
    {
        "name": "aws_cognito",
        "icon": "fa-amazon",
        "token_key": "access_token",
        "remote_app": {
            "client_id": "COGNITO_CLIENT_ID",
            "client_secret": "COGNITO_CLIENT_SECRET",
            "api_base_url": "https://COGNITO_APP.auth.REGION.amazoncognito.com/",
            "client_kwargs": {"scope": "openid email aws.cognito.signin.user.admin"},
            "access_token_url": "https://COGNITO_APP.auth.REGION.amazoncognito.com/
token",
            "authorize_url": "https://COGNITO_APP.auth.REGION.amazoncognito.com/
authorize",
        },
    },
    {
        "name": "keycloak",
        "icon": "fa-key",
        "token_key": "access_token",
        "remote_app": {
            "client_id": "KEYCLOAK_CLIENT_ID",
            "client_secret": "KEYCLOAK_CLIENT_SECRET",
            "api_base_url": "https://KEYCLOAK_DOMAIN/realms/master/protocol/openid-
connect",
            "client_kwargs": {
                "scope": "email profile"
            },
            "access_token_url": "KEYCLOAK_DOMAIN/realms/master/protocol/openid-connect/
token",
            "authorize_url": "KEYCLOAK_DOMAIN/realms/master/protocol/openid-connect/
auth",
            "request_token_url": None,
        },
    },
    {
        "name": "keycloak_before_17",
        "icon": "fa-key",
        "token_key": "access_token",
        "remote_app": {
            "client_id": "KEYCLOAK_CLIENT_ID",
            "client_secret": "KEYCLOAK_CLIENT_SECRET",
            "api_base_url": "https://KEYCLOAK_DOMAIN/auth/realms/master/protocol/
openid-connect",
            "client_kwargs": {
                "scope": "email profile"
            },
            "access_token_url": "KEYCLOAK_DOMAIN/auth/realms/master/protocol/openid-
connect/token",
            "authorize_url": "KEYCLOAK_DOMAIN/auth/realms/master/protocol/openid-
connect/auth",
            "request_token_url": None,
        },
    },
    {
        "name": "azure",
        "icon": "fa-windows",
        "token_key": "access_token",
        "remote_app": {
```

```
            "client_id": "AZURE_APPLICATION_ID",
            "client_secret": "AZURE_SECRET",
            "api_base_url": "https://login.microsoftonline.com/AZURE_TENANT_ID/oauth2",
            "client_kwargs": {
                "scope": "User.read name preferred_username email profile upn",
                "resource": "AZURE_APPLICATION_ID",
            },
            "request_token_url": None,
            "access_token_url": "https://login.microsoftonline.com/AZURE_TENANT_ID/
oauth2/token",
            "authorize_url": "https://login.microsoftonline.com/AZURE_TENANT_ID/oauth2/
authorize",
        },
    },
]
```

This needs a small explanation, you basically have five special keys:

name

the name of the provider: you can choose whatever you want, but FAB has builtin logic in *BaseSecurityManager.get_oauth_user_info()* for: ‘azure’, ‘github’, ‘google’, ‘keycloak’, ‘keycloak_before_17’, ‘linkedin’, ‘okta’, ‘openshift’, ‘twitter’

icon

the font-awesome icon for this provider

token_key

the token key name that the provider uses, default is *‘oauth_token’*

token_secret

the token secret key name, default is *‘oauth_token_secret’*

remote_app

the actual configs for the provider API

You can give FlaskAppBuilder roles based on Oauth groups:

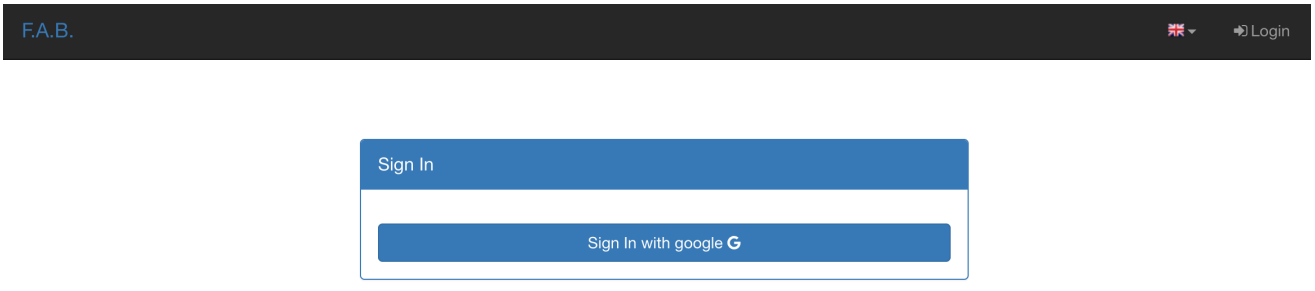
```
# note, this is only natively supported in `azure` and `okta` currently,
# however, if you customize userinfo retrieval to include 'role_keys', this will work
for other providers

# a mapping from the values of `userinfo["role_keys"]` to a list of FAB roles
AUTH_ROLES_MAPPING = {
    "FAB_USERS": ["User"],
    "FAB_ADMINS": ["Admin"],
}

# if we should replace ALL the user's roles each login, or only on registration
AUTH_ROLES_SYNC_AT_LOGIN = True

# force users to re-auth after 30min of inactivity (to keep roles in sync)
PERMANENT_SESSION_LIFETIME = 1800
```

To customize the userinfo retrieval, you can create your own method like this:

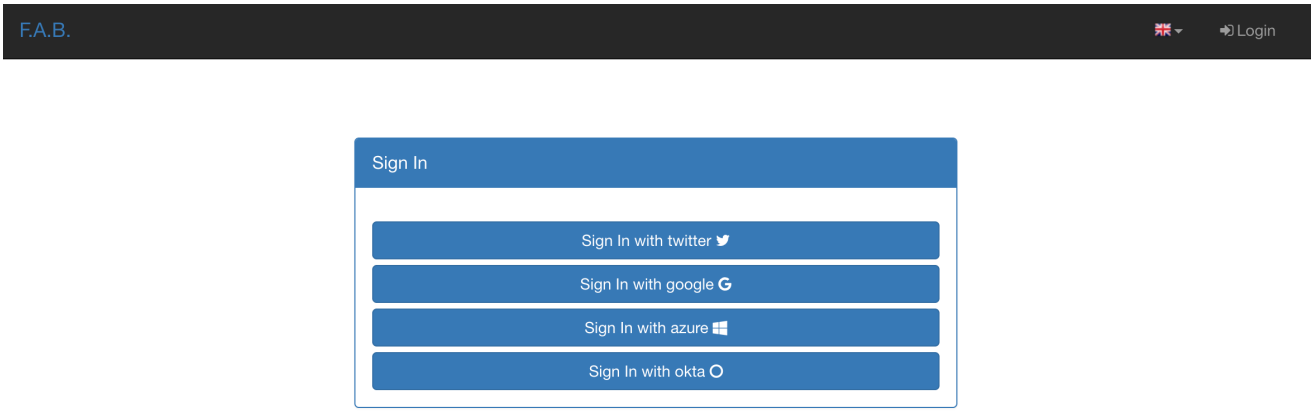
```
@appbuilder.sm.oauth_user_info_getter
def my_user_info_getter(sm, provider, response=None):
    if provider == "okta":
        me = sm.oauth_remotes[provider].get("userinfo")
        log.debug("User info from Okta: {0}".format(me.data))
        return {
            "username": "okta_" + me.data.get("sub", ""),
            "first_name": me.data.get("given_name", ""),
            "last_name": me.data.get("family_name", ""),
            "email": me.data.get("email", ""),
            "role_keys": me.data.get("groups", []),
        }
    if provider == "aws_cognito":
        me = self.appbuilder.sm.oauth_remotes[provider].get("userInfo")
        return {
            "username": me.json().get("username"),
            "email": me.json().get("email"),
            "first_name": me.json().get("given_name", ""),
            "last_name": me.json().get("family_name", ""),
            "id": me.json().get("sub", ""),
            "role_keys": ["User"], # set AUTH_ROLES_SYNC_AT_LOGIN = False
        }
    else:
        return {}
```

On Flask-AppBuilder 3.4.0 the login page has changed.

With one provider:



With multiple providers:



Note that on 3.3.X the user would automatically be sent to the provider allow page.

Decorate your method with the SecurityManager **oauth_user_info_getter** decorator. Your method should return a dictionary with the userinfo, with the keys having the same column names as the User Model. Your method will be called after the user authorizes your application on the OAuth provider. Take a look at the example

You can also use the OAuth provider APIs. Therefore, you can send tweets, post on the users Facebook, retrieve the user's LinkedIn profile etc. Take a look at the example to get an idea of a simple use for this.

Authentication: Rate limiting

To prevent brute-forcing of credentials, you can apply rate limits to AuthViews in 4.2.0, so that only 10 POST requests can be made every 20 seconds. This can be enabled by setting `AUTH_RATE_LIMITED` and `RATELIMIT_ENABLED` to `True`. The rate can be changed by adjusting `AUTH_RATE_LIMIT` to, for example, `1 per 10 seconds`. Take a look at the documentation of Flask-Limiter for more options and examples.

Role based

Each user may have multiple roles, and a role holds permissions on views/API and menus, so a user has permissions on views/API and menus.

Roles can be user defined (backed by the backend) and builtin readonly. Builtin readonly roles support regex for views/API and permissions, this simplifies security management and improve performance since the many to many permissions between a role and it's permissions does not need to be fetched from the backend.

Builtin roles are defined on the config using `FAB_ROLES` key and respect the following data structure:

```
FAB_ROLES = {
    "<ROLE NAME>": [
        ["<VIEW/MENU/API NAME>", "PERMISSION NAME"],
        ....
    ],
    ...
}
```

So for example a **Read Only** role might look like:

```
FAB_ROLES = {
    "ReadOnly": [
        [".*", "can_list"],
        [".*", "can_show"],
        [".*", "menu_access"],
        [".*", "can_get"],
        [".*", "can_info"]
    ]
}
```

These roles are inserted automatically to the database (only their name is added), and can be associated to users just like a “normal”/user defined role.

If you want to later on change the name of these roles, you can map these roles by their backend id:

```
FAB_ROLES = {
    "ReadOnly_Altered": [
        [".*", "can_list"],
        [".*", "can_show"],
        [".*", "menu_access"],
        [".*", "can_get"],
        [".*", "can_info"]
    ]
}

FAB_ROLES_MAPPING = {
    1: "ReadOnly_Altered"
}
```

There are two special roles, you can define their names on the Base Configuration

Admin Role

Special builtin read only Role, will have full access.

Public Role

This is a special role for non authenticated users, you can assign all the permissions on

views and menus to this role, and everyone will access specific parts of you application.

Of course you can create any additional role you want and configure them as you like.

Permissions

The framework automatically creates for you all the possible existing permissions on your views, API or menus, by “inspecting” your code.

Each time you create a new view based on a model (inherit from ModelView) it will create the following permissions:

- can list
- can show
- can add
- can edit
- can delete
- can download

In the case of CRUD REST API:

- can get
- can put
- can post
- can delete
- can info

These base permissions will be associated to your view or API, so if you create a view named `MyModelView` you can assign to any role the following permissions:

- can list on MyModelView
- can show on MyModelView
- can add on MyModelView
- can edit on MyModelView
- can delete on MyModelView
- can download on MyModelView

In case your developing a backend REST API subclassing `ModelRestApi` with a class named `MyApi` will generate the following permissions:

- can get on MyApi
- can put on MyApi
- can post on MyApi
- can delete on MyApi

- can info on MyApi

If you extend your view with some exposed method via the `@expose` decorator and you want to protect it use the `@has_access` decorator:

```
class MyModelView(ModelView):
    datamodel = SQLAInterface(Group)

    @has_access
    @expose('/mymethod/')
    def mymethod(self):
        # do something
        pass
```

The framework will create the following access, based on your method's name:

- can mymethod on MyModelView

You can aggregate some of your method's on a single permission, this can simplify the security configuration if there is no need for granular permissions on a group of methods, for this use `@permission_name` decorator.

You can use the `@permission_name` to override the permission's name to whatever you like.

Take a look at API Reference

Permission Customization

The default view/menu, permissions are highly granular, this is a good default since it enables a high level of customization, but on medium to large application the amount of permission pairs generated can get a bit daunting. You can fully customize the generated permission names generated and if you wish aggregate them:

```
class OneApi(ModelRestApi):
    datamodel = SQLAInterface(Contact)
    class_permission_name = "api"


class TwoApi(ModelRestApi):
    datamodel = SQLAInterface(Contact)
    class_permission_name = "api"
```

The previous example will generate half the default permissions, by just creating the following:

- can get on api
- can put on api
- can post on api
- can delete on api
- can info on api

The `class_permission_name` property is available also on BaseViews and their children `ModelView`, `MultipleView`, `MasterDetailView`, `FormView`, etc.

You can also aggregate method permissions by using `method_permission_name` attribute. Use the following `Dict` structure:

```
method_permission_name = {
    "<METHOD_NAME>": "<PERMISSION_NAME>",
    ...
}
```

Example:

```
class OneApi(ModelRestApi):
    datamodel = SQLAInterface(Contact)
    class_permission_name = "api"
    method_permission_name = {
        "get_list": "access",
        "get": "access",
        "post": "access",
        "put": "access",
        "delete": "access",
        "info": "access"
    }


class TwoApi(ModelRestApi):
    datamodel = SQLAInterface(Contact)
    class_permission_name = "api"
    method_permission_name = {
        "get_list": "access",
        "get": "access",
        "post": "access",
        "put": "access",
        "delete": "access",
        "info": "access"
    }
```

Now FAB will only generate one permission pair:

- can access on api

If you want to revert back your permission names override, or change just them again, you

need to hint FAB about what were your last permissions, so that the security converge procedure knows what to do:

```
class OneApi(ModelRestApi):
    datamodel = SQLAInterface(Contact)
    class_permission_name = "OneApi"
    previous_class_permission_name = "api"
    method_permission_name = {
        "get_list": "get",
        "get": "get",
        "post": "post",
        "put": "put",
        "delete": "delete",
        "info": "info"
    }
    previous_method_permission_name = {
        "get_list": "access",
        "get": "access",
        "post": "access",
        "put": "access",
        "delete": "access",
        "info": "access"
    }
```

An example for compressing permissions using MVC Model Views:

```
class OneView(ModelView):
    datamodel = SQLAInterface(Contact)
    class_permission_name = "view"
    method_permission_name = {
        'add': 'write',
        'delete': 'write',
        'download': 'write',
        'edit': 'write',
        'list': 'read',
        'muldelete': 'write',
        'show': 'read',
        'api': 'read',
        'api_column_add': 'write',
        'api_column_edit': 'write',
        'api_create': 'write',
        'api_delete': 'write',
        'api_get': 'read',
        'api_read': 'read',
        'api_readvalues': 'read',
        'api_update': 'write'
    }
```

Note that if your changing an already existing application, you need to migrate the old permission names to the new ones. Before doing that you should disable the boot automatic create/delete permissions, so set `FAB_UPDATE_PERMS = False`. Then run the following FAB cli command:

```
$ flask fab security-converge
```

Security converge will migrate all your permissions from the previous names to the current names, and also change all your roles, so you can migrate smoothly to your new security naming. After converging you can delete all your `previous_*` attributes if you have set them.

You can also migrate back by switching `previous_*` attributes to their target, ie switch `previous_method_permission_name` by `method_permission_name` and `previous_class_permission_name` by `class_permission_name`. Then run security converge will expand back all permissions on all your Roles.

note

You should backup your production database before migrating your permissions. Also note that you can run `flask fab security-converge --dry-run` to get a list of operations the converge will perform.

Automatic Cleanup

All your permissions and views are added automatically to the backend and associated with the 'Admin' *role*. The same applies to removing them. But, if you change the name of a view or menu, the framework will add the new *Views* and *Menus* names to the backend, but will not delete the old ones. It will generate unwanted names on the security models, basically *garbage*. To clean them, use the *security_cleanup* method.

Using security_cleanup is not always necessary, but using it after code rework, will guarantee that the permissions, and associated permissions to menus and views are exactly what exists on your app. It will prevent orphaned permission names and associations.

Use the cleanup after you have registered all your views.

```
appbuilder.add_view(GroupModelView, "List Groups", category="Contacts")
appbuilder.add_view(ContactModelView, "List Contacts", category="Contacts")
appbuilder.add_separator("Contacts")
appbuilder.add_view(ContactChartView, "Contacts Chart", category="Contacts")
appbuilder.add_view(ContactTimeChartView, "Contacts Birth Chart", category="Contacts")

appbuilder.security_cleanup()
```

You can always use it and everything will be painlessly automatic. But if you use it only when needed (change class name, add *security_cleanup* to your code, the *garbage* names are removed, then remove the method) no overhead is added when starting your site.

Auditing

All user's creation and modification are audited. On the show detail for each user you can check who created the user and when and who has last changed it.

You can check also, a total login count (successful login), and the last failed logins (these are reset if a successful login occurred).

If you're using SQLAlchemy you can mix auditing to your models in a simple way. Mix AuditMixin class to your models:

```
from flask_appbuilder.models.mixins import AuditMixin
from flask_appbuilder import Model
from sqlalchemy import Column, Integer, String


class Project(AuditMixin, Model):
    id = Column(Integer, primary_key=True)
    name = Column(String(150), unique=True, nullable=False)
```

This will add the following columns to your model:

- created_on: The date and time of the record creation.
- changed_on: The last date and time of record update.
- created_by: Who created the record.
- changed_by: Who last modified the record.

These columns will be automatically updated by the framework upon creation or update of records. So you should exclude them from add and edit form. Using our example you will define our view like this:

```
class ProjectModelView(ModelView):
    datamodel = SQLAInterface(Project)
    add_columns = ['name']
    edit_columns = ['name']
```

Password complexity validation

This feature only makes sense when using AUTH database. By default you can enable password complexity validation by setting *FAB_PASSWORD_COMPLEXITY_ENABLED = True*.

This default enforces:

- At least 2 Uppercase letters

- At least 3 Lowercase letters
- At least 1 special character
- At least 2 numeric digits
- At least 10 total characters

If you want to set your own password complexity validation, you can write your own validation function:

Example on your config:

```
from flask_appbuilder.exceptions import PasswordComplexityValidationError
...

def custom_password_validator(password: str) -> None:
    """
    A simplistic example for a password validator
    """
    if len(password) < 8:
        raise PasswordComplexityValidationError("Must have at least 8 characters")

FAB_PASSWORD_COMPLEXITY_VALIDATOR = custom_password_validator
FAB_PASSWORD_COMPLEXITY_ENABLED = True
```

Your Custom Security

If you want to alter the security views, or authentication methods since (1.0.1) you can do it in a simple way. The **AppBuilder** has a new optional initialization parameter where you pass your own custom **SecurityManager** If you want to add, for example, actions to the list of users you can do it in a simple way.

First i advise you to create security.py and add the following to it:

```
from flask import redirect
from flask_appbuilder.security.views import UserDBModelView
from flask_appbuilder.security.sqla.manager import SecurityManager
from flask_appbuilder.actions import action


class MyUserDBView(UserDBModelView):
    @action("muldelete", "Delete", "Delete all Really?", "fa-rocket", single=False)
    def muldelete(self, items):
        self.datamodel.delete_all(items)
        self.update_redirect()
        return redirect(self.get_redirect())


class MySecurityManager(SecurityManager):
    userdbmodelview = MyUserDBView
```

Then on the __init__.py initialize AppBuilder with you own security class:

```
appbuilder = AppBuilder(app, db.session, security_manager_class=MySecurityManager)
```

Alternatively since 1.13.1 you can declare your custom **SecurityManager** on the config. This is a must have if your using the factory app pattern, on the config declare you class the following way:

```
FAB_SECURITY_MANAGER_CLASS='app.security.MySecurityManager'
```

F.A.B. uses a different user view for each authentication method

UserDBModelView

For database auth method

UserOIDModelView

For Open ID auth method

UserLDAPModelView

For LDAP auth method

You can extend or create from scratch your own, and then tell F.A.B. to use them instead, by overriding their correspondent lower case properties on **SecurityManager** (just like on the given example).

Take a look and run the example on Employees example

Study the source code of BaseSecurityManager

Extending the User Model

If you want to extend the **User** Model with extra columns specific to your application (since 1.3.0) you can easily do it. Use the same type of approach as explained earlier.

First extend the User Model (create a sec_models.py file):

```
from flask_appbuilder.security.sqla.models import User
from sqlalchemy import Column, Integer, ForeignKey, String, Sequence, Table
from sqlalchemy.orm import relationship, backref
from flask_appbuilder import Model

class MyUser(User):
    __tablename__ = 'ab_user'
    extra = Column(String(256))
```

Next define a new User view, just like the default User view but with the extra column (create a sec_views.py) If you're using:

AUTH_DB

Extend UserDBModelView

AUTH_LDAP

Extend UserLDAPModelView

AUTH_REMOTE_USER

Extend UserRemoteUserModelView

AUTH_OID

Extend UserOIDModelView

AUTH_OAUTH

Extend UserOAuthModelView

So using AUTH_DB:

```
from flask_appbuilder.security.views import UserDBModelView
from flask_babel import lazy_gettext

class MyUserDBModelView(UserDBModelView):
    """
        View that add DB specifics to User view.
        Override to implement your own custom view.
        Then override userdbmodelview property on SecurityManager
    """

    show_fieldsets = [
        (lazy_gettext('User info'),
         {'fields': ['username', 'active', 'roles', 'login_count', 'extra']}),
        (lazy_gettext('Personal Info'),
         {'fields': ['first_name', 'last_name', 'email'], 'expanded': True}),
        (lazy_gettext('Audit Info'),
         {'fields': ['last_login', 'fail_login_count', 'created_on',
                     'created_by', 'changed_on', 'changed_by'], 'expanded': False}),
    ]

    user_show_fieldsets = [
        (lazy_gettext('User info'),
         {'fields': ['username', 'active', 'roles', 'login_count', 'extra']}),
        (lazy_gettext('Personal Info'),
         {'fields': ['first_name', 'last_name', 'email'], 'expanded': True}),
    ]

    add_columns = [
        'first_name',
        'last_name',
        'username',
        'active',
        'email',
        'roles',
        'extra',
        'password',
        'conf_password'
    ]
    list_columns = [
        'first_name',
        'last_name',
        'username',
        'email',
        'active',
        'roles'
    ]
    edit_columns = [
        'first_name',
        'last_name',
        'username',
        'active',
        'email',
        'roles',
        'extra'
    ]
```

Next create your own SecurityManager class, overriding your model and view for User (create a sec.py):

```
from flask_appbuilder.security.sqla.manager import SecurityManager
from .sec_models import MyUser
from .sec_views import MyUserDBModelView

class MySecurityManager(SecurityManager):
    user_model = MyUser
    userdbmodelview = MyUserDBModelView
```

Note that this is for AUTH_DB, so if you're using:

AUTH_DB

Override userdbmodelview

AUTH_LDAP

Override userldapmodelview

AUTH_REMOTE_USER

Override userremoteusermodelview

AUTH_OID

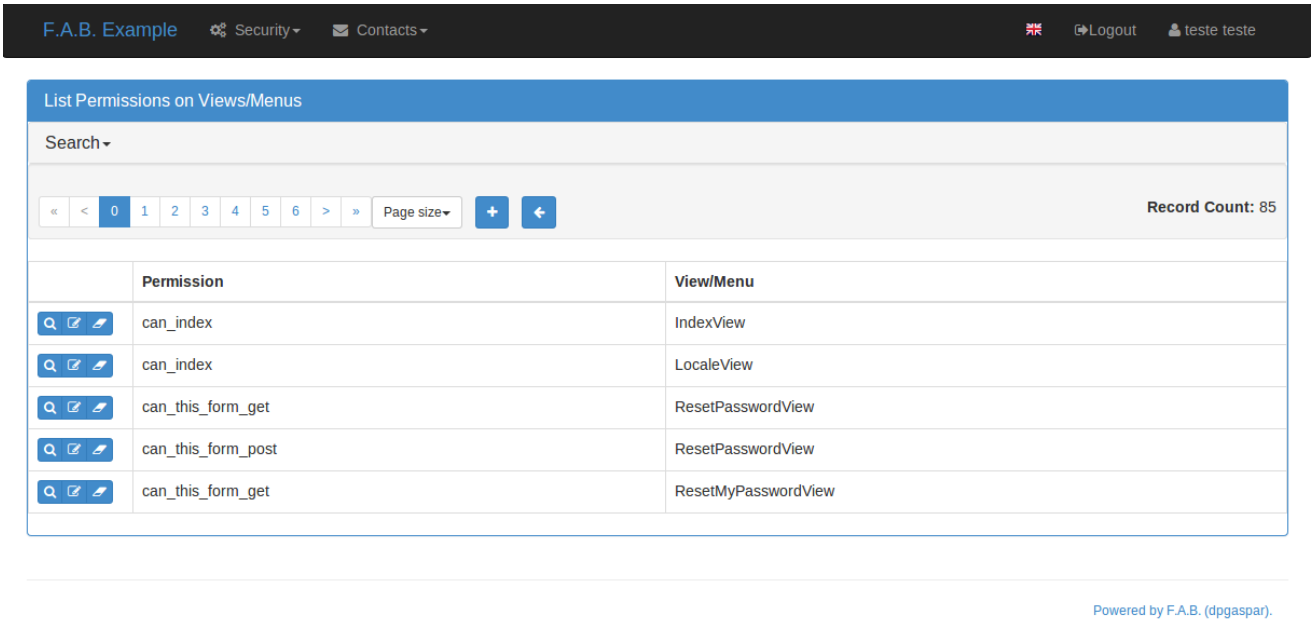
Override useroidmodelview

Finally (as shown on the previous example) tell F.A.B. to use your SecurityManager class, so when initializing **AppBuilder** (on __init__.py):

```
from flask import Flask
from flask_appbuilder import SQLA, AppBuilder
from flask_appbuilder.menu import Menu
from .sec import MySecurityManager

app = Flask(__name__)
app.config.from_object('config')
db = SQLA(app)
appbuilder = AppBuilder(app, db.session, menu=Menu(reverse=False),
security_manager_class=MySecurityManager)

from app import views
```

Now you'll have your extended User model as the authenticated user, *g.user* will have your model with the extra col.

Some images:



User Registration

Allows users to register themselves as users, will behave differently according to the authentication method.

Database Authentication

Using database authentication (auth db) the login screen will present a new 'Register' option where the user is directed to a form where he/she fill's a form with the necessary login/user information. The form includes a Recaptcha field to ensure a human is filling the form. After the form is correctly filled by the user an email is sent to the user with a link with an URL containing a hash belonging to his/her registration.

If the URL is accessed the user is inserted into the F.A.B user model and activated.

This behaviour can be easily configured or completely altered. By overriding the **RegisterUserDBView** properties. or implementing an all new class. **RegisterUserDBView** inherits from BaseRegisterUser that hold some handy base methods and properties.

Note that the process required for sending email's uses the excellent flask-mail package so make sure you installed it first.

Enabling and using the default implementation is easy just configure the following global config keys on config.py:

```
AUTH_TYPE = AUTH_DB
AUTH_USER_REGISTRATION = True
AUTH_USER_REGISTRATION_ROLE = 'Public'
# Config for Flask-WTF Recaptcha necessary for user registration
RECAPTCHA_PUBLIC_KEY = 'GOOGLE PUBLIC KEY FOR RECAPTCHA'
RECAPTCHA_PRIVATE_KEY = 'GOOGLE PRIVATE KEY FOR RECAPTCHA'
# Config for Flask-Mail necessary for user registration
MAIL_SERVER = 'smtp.gmail.com'
MAIL_USE_TLS = True
MAIL_USERNAME = 'yourappemail@gmail.com'
MAIL_PASSWORD = 'passwordformail'
MAIL_DEFAULT_SENDER = 'fabtest10@gmail.com'
```

OpenID Authentication

Registering a user when using OpenID authentication is very similar to database authentication, but this time all the basic necessary information is fetched from the provider and presented to the user to alter it (or not) and submit.

LDAP Authentication

LDAP user self registration is automatic, no register user option is shown. All users are registered, and the required information is fetched from the LDAP server.

Configuration

You can configure the default behaviour and UI on many different ways. The easiest one is making your own RegisterUser class and inherit from RegisterUserDBView (when using auth db). Let's take a look at a practical example:

```
from flask_appbuilder.security.registerviews import RegisterUserDBView

class MyRegisterUserDBView(RegisterUserDBView):
    email_template = 'register_mail.html'
    email_subject = lazy_gettext('Your Account activation')
    activation_template = 'activation.html'
    form_title = lazy_gettext('Fill out the registration form')
    error_message = lazy_gettext('Not possible to register you at the moment, try again
later')
    message = lazy_gettext('Registration sent to your email')
```

This class will override:

- The template used to generate the email sent by the user. Take a look at the default template to get a simple starting point Mail template. Your template will receive the following parameters:
    - first_name
    - last_name
    - username
    - url
- The email subject
- The activation template. This the page shown to the user when he/she finishes the activation. Take a look at the default template to get a simple starting point Activation Template.
- The form title. The title that is presented on the registration form.
- Message is the success message presented to the user when an email was successfully sent to him and his registration was recorded.

After defining your own class, override SecurityManager class and set the **registeruserdbview** property with your own class:

```
class MySecurityManager(SecurityManager):
    registeruserdbview = MyRegisterUserDBView
```

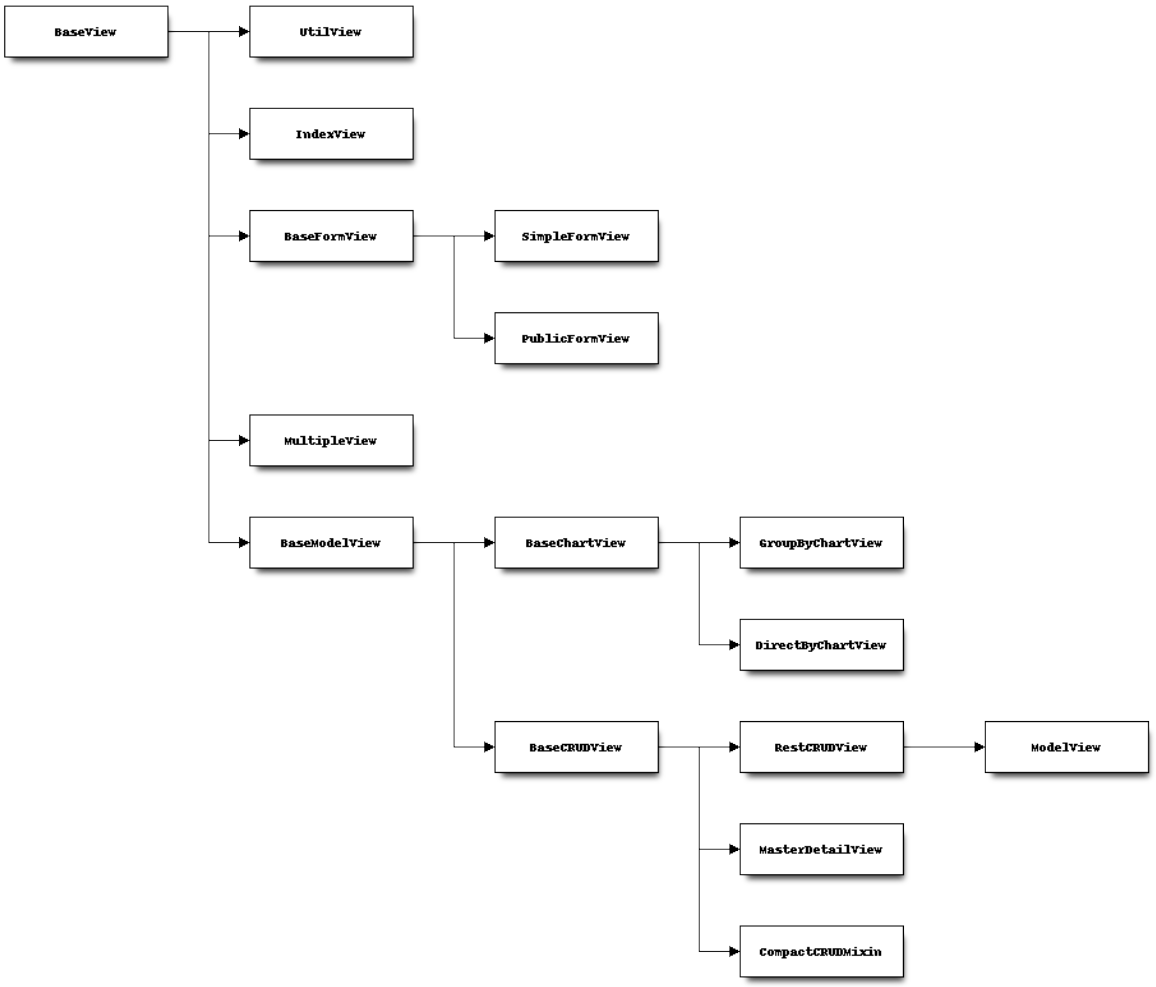
Then tell F.A.B. to use your security manager class, take a look at the Security on how to do it.

Diagrams

This page will show various diagrams about the framework structure.

Class View Diagram Tree

All class views tree reflect functionality each layer is responsible for a certain goal. You will be essentially using BaseViews, IndexViews and the leafs ModelView, chart views and form views.



Next is a summary explanation for each class:

BaseView

Collects all the exposed methods, creates the Flask blueprint and registers the URLs, initializes base permissions.

UtilView

Implements exposes **back** for special back UI functionality.

IndexView

Special view for rendering the index page.

SimpleFormView

Subclass it to render WTForms.

PublicFormView

Same as SimpleFormView but with public access only.

BaseModelView

Class responsible for an initial datamodel layer, implements search form and filters.

BaseChartView

Basic chart view functionality.

GroupByChartView

Subclass it to render Google charts with group by queries.

DirectByChartView

Subclass it to render Google charts with queries.

BaseCRUDView

Implement base functionality for add, edit, delete, creates all forms.

RestCRUDView

Exposes the JSON REST API for CRUD methods and more.

ModelView

Subclass it to render your views based on models, with complete CRUD UI functionality.

MasterDetailView

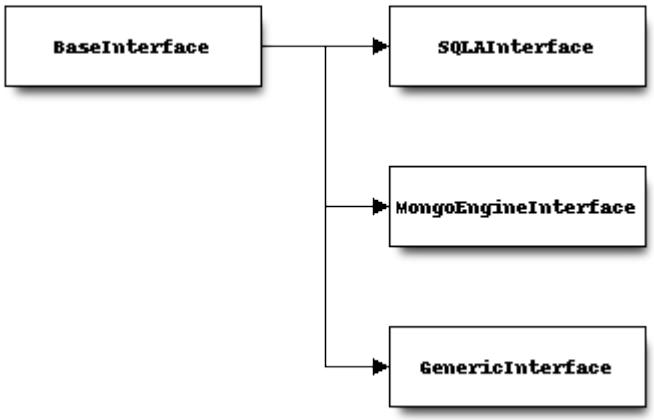
Renders a master ModelView and multiple detail ModelViews thar are database related.

MultipleView

Renders multiple views on the same page (ex: ModelView and GroupByChartView)

Class Data Diagram Tree

All classes for data access aim for abstracting the backend.



BaseInterface

Interface class, imposes a unique API layer for data access.

SQLAInterface

Data access for SQLAlchemy.

MongoEngineInterface

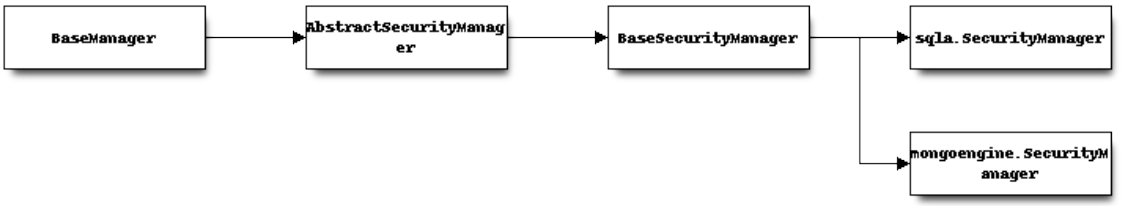
Data access for MongoEngine (MongoDB).

GenericInterface

Data access for custom data structures.

Class Security Diagram Tree

Classes that are involved in implementing security. Register security views, implement various methods of authentication manage permissions (insert/remove all permission on the backend).



BaseManager

Base class for all Manager classes, holds AppBuilder class.

AbstractSecurityManager

Abstract class for Security managers, defines the must have methods.

BaseSecurityManager

Base class for security, registers security views, implements authentication, inserts/removes all permission on the database, manages roles/users and views.

sqla.SecurityManager

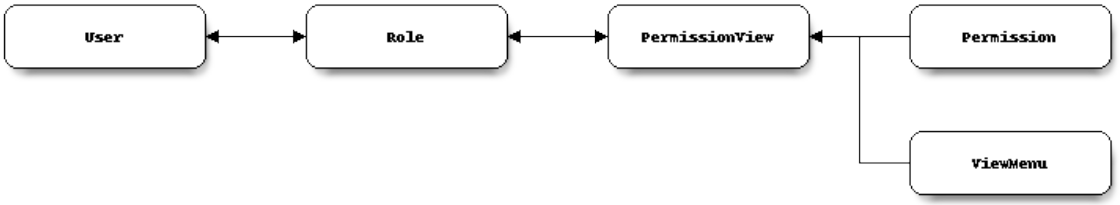
Implements BaseSecurityManager for SQAlchemy.

mongoengine.SecurityManager

Implements BaseSecurityManager for MongoEngine.

Security Models ERD

This is the ERD of the frameworks security models.



API Reference

flask_appbuilder

AppBuilder

class flask_appbuilder.base.AppBuilder(*app: Optional[Flask] = None, session: Optional[Session] = None, menu: Optional[Menu] = None, indexview: Optional[Type[AbstractViewApi]] = None, base_template: str = 'appbuilder/baselayout.html', static_folder: str = 'static/appbuilder', static_url_path: str = '/appbuilder', security_manager_class: Optional[Type[BaseSecurityManager]] = None, update_perms: bool = True*)
[source]

This is the base class for all the framework. This is where you will register all your views and create the menu structure. Will hold your flask app object, all your views, and security classes.

initialize your application like this for SQLAlchemy:

```
from flask import Flask
from flask_appbuilder import SQLA, AppBuilder

app = Flask(__name__)
app.config.from_object('config')
db = SQLA(app)
appbuilder = AppBuilder(app, db.session)
```

When using MongoEngine:

```
from flask import Flask
from flask_appbuilder import AppBuilder
from flask_appbuilder.security.mongoengine.manager import SecurityManager
from flask_mongoengine import MongoEngine

app = Flask(__name__)
app.config.from_object('config')
dbmongo = MongoEngine(app)
appbuilder = AppBuilder(app, security_manager_class=SecurityManager)
```

You can also create everything as an application factory.

__init__(*app: Optional[Flask] = None, session: Optional[Session] = None, menu: Optional[Menu] = None, indexview: Optional[Type[AbstractViewApi]] = None, base_template: str = 'appbuilder/baselayout.html', static_folder: str = 'static/appbuilder', static_url_path: str = '/appbuilder',*

security_manager_class: Optional[Type[BaseSecurityManager]] = None, update_perms: bool = True) → None [source]

AppBuilder init

Parameters

- **app** – The flask app object
- **session** – The SQLAlchemy session object
- **menu** – optional, a previous contructed menu
- **indexview** – optional, your customized indexview
- **static_folder** – optional, your override for the global static folder
- **static_url_path** – optional, your override for the global static url path
- **security_manager_class** – optional, pass your own security manager class
- **update_perms** – optional, update permissions flag (Boolean) you can use FAB_UPDATE_PERMS config key also

add_api(*baseview: Type[AbstractViewApi]*) → AbstractViewApi [source]

Add a BaseApi class or child to AppBuilder

Parameters

baseview – A BaseApi type class

Returns

The instantiated base view

add_link(*name: str, href: str, icon: str = '', label: str = '', category: str = '', category_icon: str = '', category_label: str = '', baseview: Optional[AbstractViewApi] = None, cond: Optional[Callable[[...], bool]] = None*) → None [source]

Add your own links to menu using this method

Parameters

- **baseview** –
- **name** – The string name that identifies the menu.
- **href** – Override the generated href for the menu. You can use an url string or an endpoint name
- **icon** – Font-Awesome icon name, optional.
- **label** – The label that will be displayed on the menu, if absent param name will be used
- **category** – The menu category where the menu will be included, if non

provided the view will be accessible as a top menu.

- **category_icon** – Font-Awesome icon name for the category, optional.
- **category_label** – The label that will be displayed on the menu, if absent param name will be used
- **cond** – If a callable, `cond` will be invoked when constructing the menu items. If it returns `True`, then this link will be a part of the menu. Otherwise, it will not be included in the menu items. Defaults to `None`, meaning the item will always be present.

add_separator(*category: str*, *cond: Optional[Callable[[...], bool]] = None*) → None [source]

Add a separator to the menu, you will sequentially create the menu

Parameters

- **category** – The menu category where the separator will be included.
- **cond** – If a callable, `cond` will be invoked when constructing the menu items. If it returns `True`, then this separator will be a part of the menu. Otherwise, it will not be included in the menu items. Defaults to `None`, meaning the separator will always be present.

add_view(*baseview: Union[Type[AbstractViewApi], AbstractViewApi]*, *name: str*, *href: str = ''*, *icon: str = ''*, *label: str = ''*, *category: str = ''*, *category_icon: str = ''*, *category_label: str = ''*, *menu_cond: Optional[Callable[[...], bool]] = None*) → AbstractViewApi [source]

Add your views associated with menus using this method.

Parameters

- **baseview** – A BaseView type class instantiated or not. This method will instantiate the class for you if needed.
- **name** – The string name that identifies the menu.
- **href** – Override the generated href for the menu. You can use an url string or an endpoint name if non provided default_view from view will be set as href.
- **icon** – Font-Awesome icon name, optional.
- **label** – The label that will be displayed on the menu, if absent param name will be used
- **category** – The menu category where the menu will be included, if non provided the view will be acessible as a top menu.
- **category_icon** – Font-Awesome icon name for the category, optional.
- **category_label** – The label that will be displayed on the menu, if absent param name will be used

- **menu_cond** – If a callable, `menu_cond` will be invoked when constructing the menu items. If it returns `True`, then this link will be a part of the menu. Otherwise, it will not be included in the menu items. Defaults to `None`, meaning the item will always be present.

Examples:

```
appbuilder = AppBuilder(app, db)
# Register a view, rendering a top menu without icon.
appbuilder.add_view(MyModelView(), "My View")
# or not instantiated
appbuilder.add_view(MyModelView, "My View")
# Register a view, a submenu "Other View" from "Other" with a phone icon.
appbuilder.add_view(
    MyOtherModelView,
    "Other View",
    icon='fa-phone',
    category="Others"
)
# Register a view, with category icon and translation.
appbuilder.add_view(
    YetOtherModelView,
    "Other View",
    icon='fa-phone',
    label=_('Other View'),
    category="Others",
    category_icon='fa-envelop',
    category_label=_('Other View')
)
# Register a view whose menu item will be conditionally displayed
appbuilder.add_view(
    YourFeatureView,
    "Your Feature",
    icon='fa-feature',
    label=_('Your Feature'),
    menu_cond=lambda: is_feature_enabled("your-feature"),
)
# Add a link
appbuilder.add_link("google", href="www.google.com", icon = "fa-google-plus")
```

add_view_no_menu(*baseview: Union[Type[AbstractViewApi], AbstractViewApi]*, *endpoint: Optional[str] = None*, *static_folder: Optional[str] = None*) → AbstractViewApi [source]

Add your views without creating a menu.

Parameters

- **baseview** – A BaseView type class instantiated.
- **endpoint** – The endpoint path for the Flask blueprint
- **static_folder** – The static folder for the Flask blueprint

property **app_icon: *str***

Get the App icon location

Returns

String with relative app icon location

property **app_name: *str***

Get the App name

Returns

String with app name

property **app_theme: *str***

Get the App theme name

Returns

String app theme name

property **get_app: *Flask***

Get current or configured flask app

Returns

Flask App

property **get_session: *Session***

Get the current sqlalchemy session.

Returns

SQLAlchemy Session

init_app(*app: Flask*, *session: Session*) → None [source]

Will initialize the Flask app, supporting the app factory pattern.

Parameters

- **app** –
- **session** – The SQLAlchemy session

security_cleanup() → None [source]

This method is useful if you have changed the name of your menus or classes, changing them will leave behind permissions that are not associated with anything.

You can use it always or just sometimes to perform a security cleanup. Warning this will delete any permission that is no longer part of any registered view or menu.

Remember invoke ONLY AFTER YOU HAVE REGISTERED ALL VIEWS

security_converge(*dry: bool = False*) → Dict[str, Any] [source]

This method is useful when you use:

- *class_permission_name*
- *previous_class_permission_name*

- *method_permission_name*
- *previous_method_permission_name*

migrates all permissions to the new names on all the Roles

Parameters

dry – If True will not change DB

Returns

Dict with all computed necessary operations

property **version: *str***

Get the current F.A.B. version

Returns

String with the current F.A.B. version

flask_appbuilder.security.decorators

flask_appbuilder.security.decorators.protect(*allow_browser_login=False*) [source]

Use this decorator to enable granular security permissions to your API methods (BaseApi and child classes). Permissions will be associated to a role, and roles are associated to users.

allow_browser_login will accept signed cookies obtained from the normal MVC app:

```
class MyApi(BaseApi):
    @expose('/dosonmething', methods=['GET'])
    @protect(allow_browser_login=True)
    @safe
    def do_something(self):
        ....

    @expose('/dosonmethingelse', methods=['GET'])
    @protect()
    @safe
    def do_something_else(self):
        ....
```

By default the permission's name is the methods name.

flask_appbuilder.security.decorators.has_access(*f*) [source]

Use this decorator to enable granular security permissions to your methods. Permissions will be associated to a role, and roles are associated to users.

By default the permission's name is the methods name.

flask_appbuilder.security.decorators.permission_name(*name*) [source]

Use this decorator to override the name of the permission. has_access will use the methods name has the permission name if you want to override this add this decorator to your methods. This is useful if you want to aggregate methods to permissions

It will add '_permission_name' attribute to your method that will be inspected by BaseView to collect your view's permissions.

Note that you should use @has_access to execute after @permission_name like on the following example.

Use it like this to aggregate permissions for your methods:

```
class MyModelView(ModelView):
    datamodel = SQLAInterface(MyModel)

    @has_access
    @permission_name('GeneralXPTO_Permission')
    @expose(url='/xpto')
    def xpto(self):
        return "Your on xpto"

    @has_access
    @permission_name('GeneralXPTO_Permission')
    @expose(url='/xpto2')
    def xpto2(self):
        return "Your on xpto2"
```

Parameters

name – The name of the permission to override

flask_appbuilder.models.decorators

flask_appbuilder.models.decorators.renders(*col_name*) [source]

Use this decorator to map your custom Model properties to actual Model db properties. As an example:

```
class MyModel(Model):
    id = Column(Integer, primary_key=True)
    name = Column(String(50), unique = True, nullable=False)
    custom = Column(Integer(20))

    @renders('custom')
    def my_custom(self):
        # will render this columns as bold on ListWidget
        return Markup('<b>' + self.custom + '</b>')

class MyModelView(ModelView):
    datamodel = SQLAInterface(MyTable)
    list_columns = ['name', 'my_custom']
```

flask_appbuilder.hooks

flask_appbuilder.hooks.before_request(*hook: Optional[Callable[[], Any]] = None*, *only: Optional[List[str]] = None*) → Callable[[...], Any] [source]

This decorator provides a way to hook into the request lifecycle by enqueueing methods to be invoked before each handler in the view. If the method returns a value other than `None`, then that value will be returned to the client. If invoked with the `only` kwarg, the hook will only be invoked for the given list of handler methods.

Examples:

```
class MyFeature(ModelView)

    @before_request
    def ensure_feature_is_enabled(self):
        if self.feature_is_disabled:
            return self.response_404()
        return None

    # etc...


class MyView(ModelRestAPI):

    @before_request(only=["create", "update", "delete"])
    def ensure_write_mode_enabled(self):
        if self.read_only:
            return self.response_400()
        return None

    # etc...
```

Parameters

- **hook** – A callable to be invoked before handlers in the class. If the hook returns `None`, then the request proceeds and the handler is invoked. If it returns something other than `None`, then execution halts and that value is returned to the

client.

- **only** – An optional list of the names of handler methods. If present, `hook` will only be invoked before the handlers specified in the list. If absent, `hook` will be invoked for before all handlers in the class.

flask_appbuilder.api

flask_appbuilder.api.expose(*url: str = '/', methods: Tuple[str] = ('GET',)*) → Callable[[...], Any] [source]

Use this decorator to expose API endpoints on your API classes.

Parameters

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

flask_appbuilder.api.rison(*schema: Optional[Dict[str, Any]] = None*) → Callable[[Callable[[...], Any]], Callable[[...], Any]] [source]

Use this decorator to parse URI *Rison* arguments to a python data structure, your method gets the data structure on kwargs['rison']. Response is HTTP 400 if *Rison* is not correct:

```
class ExampleApi(BaseApi):
        @expose('/risonjson')
        @rison()
        def rison_json(self, **kwargs):
            return self.response(200, result=kwargs['rison'])
```

You can additionally pass a JSON schema to validate Rison arguments:

```
schema = {
    "type": "object",
    "properties": {
        "arg1": {
            "type": "integer"
        }
    }
}

class ExampleApi(BaseApi):
        @expose('/risonjson')
        @rison(schema)
        def rison_json(self, **kwargs):
            return self.response(200, result=kwargs['rison'])
```

flask_appbuilder.api.safe(*f: Callable[[...], Any]*) → Callable[[...], Any] [source]

A decorator that catches uncaught exceptions and return the response in JSON format

(inspired on Superset code)

BaseApi

class flask_appbuilder.api.BaseApi [source]

All apis inherit from this class. it's constructor will register your exposed urls on flask as a Blueprint.

This class does not expose any urls, but provides a common base for all APIS.

allow_browser_login= *False*

Will allow flask-login cookie authorization on the API default is False.

apispec_parameter_schemas: *Optional[Dict[str, Dict[str, Any]]]* = *None*

Set your custom Rison parameter schemas here so that they get registered on the OpenApi spec:

```
custom_parameter = {
    "type": "object"
    "properties": {
        "name": {
            "type": "string"
        }
    }
}

class CustomApi(BaseApi):
    apispec_parameter_schemas = {
        "custom_parameter": custom_parameter
    }
```

base_permissions: *Optional[List[str]]* = *None*

A list of allowed base permissions:

```
class ExampleApi(BaseApi):
    base_permissions = ['can_get']
```

class_permission_name: *Optional[str]* = *None*

Override class permission name default fallback to self.__class__.__name__

csrf_exempt= *True*

If using flask-wtf CSRFProtect exempt the API from check

exclude_route_methods: *Set[str]* = *{}*

Does not register routes for a set of builtin ModelRestApi functions. example:

```
class ContactModelView(ModelRestApi):
    datamodel = SQLAInterface(Contact)
    exclude_route_methods = {"info", "get_list", "get"}
```

The previous examples will only register the *put*, *post* and *delete* routes

get_init_inner_views() → List[AbstractViewApi] [source]

Sets initialized inner views

get_method_permission(*method_name: str*) → str [source]

Returns the permission name for a method

get_uninit_inner_views() → List[Type[AbstractViewApi]] [source]

Will return a list with views that need to be initialized. Normally related_views from ModelView

include_route_methods: *Optional[Set[str]]* = *None*

If defined will assume a white list setup, where all endpoints are excluded except those define on this attribute example:

```
class ContactModelView(ModelRestApi):
    datamodel = SQLAInterface(Contact)
    include_route_methods = {"list"}
```

The previous example will exclude all endpoints except the *list* endpoint

limits: *Optional[List[Limit]]* = *None*

List of limits for this api.

Use it like this if you want to restrict the rate of requests to a view:

```
class MyView(ModelView):
    limits = [Limit("2 per 5 second")]
```

or use the decorator @limit.

method_permission_name: *Optional[Dict[str, str]]* = *None*

Override method permission names, example:

```
method_permissions_name = {
    'get_list': 'read',
    'get': 'read',
    'put': 'write',
    'post': 'write',
    'delete': 'write'
}
```

openapi_spec_component_schemas: ***Tuple[Type[Schema], ...] = ()***

A Tuple containing marshmallow schemas to be registered on the OpenAPI spec has component schemas, these can be referenced by the endpoint's spec like: *$ref: '#/components/schemas/MyCustomSchema'* Where MyCustomSchema is the marshmallow schema class name.

To set your own OpenAPI schema component name, declare your schemas with: __component_name__

class Schema1(Schema):

__component_name__ = "MyCustomSchema" id = fields.Integer() ...

openapi_spec_methods: ***Dict[str, Any] = {}***

Merge OpenAPI spec defined on the method's doc. For example to merge/override *get_list*:

```
class GreetingApi(BaseApi):
    resource_name = "greeting"
    openapi_spec_methods = {
        "greeting": {
            "get": {
               "description": "Override description",
            }
        }
    }
```

openapi_spec_tag: ***Optional[str] = None***

By default all endpoints will be tagged (grouped) to their class name. Use this attribute to override the tag name

operation_helper(*path: Optional[str] = None, operations: Optional[Dict[str, Any]] = None, methods: Optional[List[str]] = None, func: Optional[Callable[[...], Response]] = None, **kwargs: Any***) → None**

[source]

May mutate operations. :param str path: Path to the resource :param dict operations: A *dict* mapping HTTP methods to operation object. :param list methods: A list of HTTP methods registered for this path

path_helper(*path: Optional[str] = None*, *operations: Optional[Dict[str, Dict]] = None*, ***kwargs: Any*) → str [source]

Works like an apispec plugin May return a path as string and mutate operations dict.

Parameters

- **path** (*str*) – Path to the resource
- **operations** (*dict*) – A *dict* mapping HTTP methods to operation object. See https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.2.md#operationObject
- **kwargs** –

Returns

Return value should be a string or None. If a string is returned, it

is set as the path.

previous_class_permission_name: *Optional[str]* = *None*

If set security converge will replace all permissions tuples with this name by the class_permission_name or self.__class__.__name__

previous_method_permission_name: *Optional[Dict[str, str]]* = *None*

Use same structure as method_permission_name. If set security converge will replace all method permissions by the new ones

resource_name: *Optional[str]* = *None*

Defines a custom resource name, overrides the inferred from Class name makes no sense to use it with route base

static response(*code: int*, ***kwargs: Any*) → Response [source]

Generic HTTP JSON response method

Parameters

- **code** – HTTP code (int)
- **kwargs** – Data structure for response (dict)

Returns

HTTP Json response

response_400(*message: Optional[str] = None*) → Response [source]

Helper method for HTTP 400 response

Parameters

message – Error message (str)

Returns

HTTP Json response

response_401() → Response [source]

Helper method for HTTP 401 response

Parameters

message – Error message (str)

Returns

HTTP Json response

response_403() → Response [source]

Helper method for HTTP 403 response

Parameters

message – Error message (str)

Returns

HTTP Json response

response_404() → Response [source]

Helper method for HTTP 404 response

Parameters

message – Error message (str)

Returns

HTTP Json response

response_422(*message: Optional[str] = None*) → Response [source]

Helper method for HTTP 422 response

Parameters

message – Error message (str)

Returns

HTTP Json response

response_500(*message: Optional[str] = None*) → Response [source]

Helper method for HTTP 500 response

Parameters

message – Error message (str)

Returns

HTTP Json response

responses
= {'400': {'content': {'application/json': {'schema': {'properties': {'message': {'type': 'string'}}, 'type': 'object'}}}, 'description': 'Bad request'}, '401': {'content': {'application/json': {'schema': {'properties': {'message': {'type': 'string'}}, 'type': 'object'}}}, 'description': 'Unauthorized'}, '403': {'content': {'application/json': {'schema': {'properties': {'message': {'type': 'string'}}, 'type': 'object'}}}, 'description': 'Forbidden'}, '404': {'content': {'application/json': {'schema': {'properties': {'message': {'type': 'string'}}, 'type': 'object'}}}, 'description': 'Not found'}, '422': {'content': {'application/json': {'schema': {'properties': {'message': {'type': 'string'}}, 'type': 'object'}}}, 'description': 'Could not process entity'}, '500': {'content': {'application/json': {'schema': {'properties': {'message': {'type': 'string'}}, 'type': 'object'}}}, 'description': 'Fatal error'}}

Override custom OpenApi responses

route_base: *Optional[str] = None*

Define the route base where all methods will suffix from

version: *Optional[str] = 'v1'*

Define the Api version for this resource/class

ModelRestApi

class flask_appbuilder.api.ModelRestApi [source]

add_columns: *Optional[List[str]] = None*

A list of columns (or model's methods) to be allowed to post

add_exclude_columns: *Optional[List[str]] = None*

A list of columns to exclude from the add endpoint. By default all columns are included.

add_model_schema: *Optional[Schema] = None*

Override to provide your own marshmallow Schema for JSON to SQLA dumps

add_query_rel_fields= *None*

Add Customized query for related add fields. Assign a dictionary where the keys are

the column names of the related models to filter, the value for each key, is a list of lists with the same format as base_filter {'relation col name':[['Related model col',FilterClass,'Filter Value'],...],...} Add a custom filter to form related fields:

```
class ContactModelView(ModelRestApi):
    datamodel = SQLAModel(Contact)
    add_query_rel_fields = {'group':[['name',FilterStartsWith,'W']]}
```

add_title: *Optional[str]* = ''

Add Title , if not configured the default is 'Add ' with pretty model name

delete(*pk: Union[str, int]***) → Response** [source]

Delete item from Model — delete:

parameters: - in: path

schema:

type: integer

name: pk

responses:

200:

description: Item deleted content:

application/json:

schema:

type: object properties:

message:

type: string

404:

$ref: '#/components/responses/404'

422:

$ref: '#/components/responses/422'

500:

$ref: '#/components/responses/500'

delete_headless(*pk: Union[str, int]***) → Response** [source]

Delete item from Model

description_columns: *Optional[Dict[str, str]] = None*

Dictionary with column descriptions that will be shown on the forms:

```
class MyView(ModelView):
    datamodel = SQLAModel(MyTable, db.session)

    description_columns = {'name':'your models name column',
                           'address':'the address column'}
```

edit_columns: *Optional[List[str]] = None*

A list of columns (or model's methods) to be allowed to update

edit_exclude_columns: *Optional[List[str]] = None*

A list of columns to exclude from the edit endpoint. By default all columns are included.

edit_model_schema: *Optional[Schema] = None*

Override to provide your own marshmallow Schema for JSON to SQLA dumps

edit_query_rel_fields= *None*

Add Customized query for related edit fields. Assign a dictionary where the keys are the column names of the related models to filter, the value for each key, is a list of lists with the same format as base_filter {'relation col name':[['Related model col','FilterClass','Filter Value'],...],...} Add a custom filter to form related fields:

```
class ContactModelView(ModelRestApi):
    datamodel = SQLAModel(Contact, db.session)
    edit_query_rel_fields = {'group':[['name',FilterStartsWith,'W']]}
```

edit_title: *Optional[str] = ''*

Edit Title , if not configured the default is 'Edit ' with pretty model name

get(*pk: Union[str, int]*, ***kwargs: Any*) → Response [source]

Get item from Model — get:

description: >-

Get an item model

parameters: - in: path

schema:

type: integer

name: pk

- in: query name: q content:

application/json:

schema:

$ref: '#/components/schemas/get_item_schema'

responses:

200:

description: Item from Model content:

application/json:

schema:

type: object properties:

label_columns:

type: object properties:

column_name:

description: >-

The label for the column name. Will be translated by babel

example: A Nice label for the column type: string

show_columns:

description: >-

A list of columns

type: array items:

type: string

description_columns:

type: object properties:

column_name:

description: >-

The description for the column name. Will be translated by babel

example: A Nice description for the column type: string

show_title:

description: >-

A title to render. Will be translated by babel

example: Show Item Details type: string

id:

description: The item id type: string

result:

$ref: '#/components/schemas/{{self.__class__.__name__}}.get'

400:

$ref: '#/components/responses/400'

401:

$ref: '#/components/responses/401'

404:

$ref: '#/components/responses/404'

422:

$ref: '#/components/responses/422'

500:

$ref: '#/components/responses/500'

get_headless(*pk: Union[str, int]*, ***kwargs: Any*) → Response [source]

Get an item from Model

Parameters

- **pk** – Item primary key
- **kwargs** – Query string parameter arguments

Returns

HTTP Response

get_list(***kwargs: Any*) → Response [source]

Get list of items from Model — get:

description: >-

Get a list of models

parameters: - in: query

name: q content:

application/json:

schema:

$ref: '#/components/schemas/get_list_schema'

responses:

200:

description: Items from Model content:

application/json:

schema:

type: object properties:

label_columns:

type: object properties:

column_name:

description: >-

The label for the column name. Will be translated by babel

example: A Nice label for the column type: string

list_columns:

description: >-

A list of columns

type: array items:

type: string

description_columns:

type: object properties:

column_name:

description: >-

The description for the column name. Will be translated by babel

example: A Nice description for the column type: string

list_title:

description: >-

A title to render. Will be translated by babel

example: List Items type: string

ids:

description: >-

A list of item ids, useful when you don’t know the column id

type: array items:

type: string

count:

description: >-

The total record count on the backend

type: number

order_columns:

description: >-

A list of allowed columns to sort

type: array items:

type: string

result:

description: >-

The result from the get list query

type: array items:

$ref: '#/components/schemas/
{{self.__class__.__name__}}.get_list' # noqa

400:

$ref: '#/components/responses/400'

401:

$ref: '#/components/responses/401'

422:

$ref: '#/components/responses/422'

500:

$ref: '#/components/responses/500'

get_list_headless(kwargs: Any) → Response** [source]

Get list of items from Model

info(kwargs: Any) → Response** [source]

Endpoint that renders a response for CRUD REST meta data — get:

description: >-

Get metadata information about this API resource

parameters: - in: query

name: q content:

application/json:

schema:

$ref: '#/components/schemas/get_info_schema'

responses:

200:

description: Item from Model content:

application/json:

schema:

type: object properties:

add_columns:

type: object

edit_columns:

type: object

filters:

type: object properties:

column_name:

type: array items:

type: object properties:

name:

description: >-

The filter name. Will be translated by babel

type: string

operator:

description: >-

The filter operation key to use on list filters

type: string

permissions:

description: The user permissions for this API resource
type: array items:

type: string

400:

$ref: '#/components/responses/400'

401:

$ref: '#/components/responses/401'

422:

$ref: '#/components/responses/422'

500:

$ref: '#/components/responses/500'

info_headless(***kwargs: Any*) → Response [source]

response for CRUD REST meta data

list_columns: *Optional[List[str]] = None*

A list of columns (or model's methods) to be displayed on the list view. Use it to control the order of the display

list_exclude_columns: *Optional[List[str]] = None*

A list of columns to exclude from the get list endpoint. By default all columns are included.

list_model_schema: *Optional[Schema] = None*

Override to provide your own marshmallow Schema for JSON to SQLA dumps

list_outer_default_load= *False*

If True, the default load for outer joins will be applied on the get item endpoint. This is useful for when you want to control the load of the many-to-many and many-to-one relationships at the model level. Will apply:

https://docs.sqlalchemy.org/en/14/orm/loading_relationships.html#sqlalchemy.orm.Load.defaultload

list_select_columns: *Optional[List[str]] = None*

A List of column names that will be included on the SQL select. This is useful for including all necessary columns that are referenced by properties listed on *list_columns* without generating N+1 queries.

list_title= ''

List Title, if not configured the default is 'List ' with pretty model name

max_page_size: *Optional[int] = None*

class override for the FAB_API_MAX_SIZE, use special -1 to allow for any page size

model2schemaconverter

Override to use your own Model2SchemaConverter (inherit from

BaseModel2SchemaConverter)

alias of `Model2SchemaConverter`

order_columns: *Optional[List[str]]* = *None*

Allowed order columns

order_rel_fields= *None*

Impose order on related fields. assign a dictionary where the keys are the related column names:

```
class ContactModelView(ModelRestApi):
    datamodel = SQLAModel(Contact)
    order_rel_fields = {
        'group': ('name', 'asc')
        'gender': ('name', 'asc')
    }
```

page_size= *20*

Use this property to change default page size

post() → Response [source]

POST item to Model — post:

requestBody:

description: Model schema required: true content:

application/json:

schema:

$ref: '#/components/schemas/{{self.__class__.__name__}}.post'

responses:

201:

description: Item inserted content:

application/json:

schema:

type: object properties:

id:

type: string

result:

$ref: '#/components/schemas/
{{self.__class__.__name__}}.post'

400:

$ref: '#/components/responses/400'

401:

$ref: '#/components/responses/401'

422:

$ref: '#/components/responses/422'

500:

$ref: '#/components/responses/500'

post_add(*item: Model*) → None [source]

Override this, will be called after update

post_delete(*item: Model*) → None [source]

Override this, will be called after delete

post_headless() → Response [source]

POST/Add item to Model

post_update(*item: Model*) → None [source]

Override this, will be called after update

pre_add(*item: Model*) → None [source]

Override this, will be called before add.

pre_delete(*item: Model*) → None [source]

Override this, will be called before delete

pre_get(*data: Dict[str, Any]*) → None [source]

Override this, will be called before data is sent to the requester on get item endpoint. You can use it to mutate the response sent. Note that any new field added will not be reflected on the OpenApi spec.

pre_get_list(*data: Dict[str, Any]*) → None [source]

Override this, will be called before data is sent to the requester on get list endpoint. You can use it to mutate the response sent Note that any new field added will not be reflected on the OpenApi spec.

pre_update(*item: Model*) → None [source]

Override this, this method is called before the update takes place.

put(*pk: Union[str, int]*) → Response [source]

PUT item to Model — put:

parameters: - in: path

schema:

type: integer

name: pk

requestBody:

description: Model schema required: true content:

application/json:

schema:

$ref: '#/components/schemas/{{self.__class__.__name__}}.put'

responses:

200:

description: Item changed content:

application/json:

schema:

type: object properties:

result:

$ref: '#/components/schemas/
{{self.__class__.__name__}}.put'

400:

$ref: '#/components/responses/400'

401:

$ref: '#/components/responses/401'

404:

$ref: '#/components/responses/404'

422:

$ref: '#/components/responses/422'

500:

$ref: '#/components/responses/500'

put_headless(*pk: Union[str, int]*) → Response [source]

PUT/Edit item to Model

show_columns: *Optional[List[str]] = None*

A list of columns (or model's methods) for the get item endpoint. Use it to control the order of the results

show_exclude_columns: *Optional[List[str]] = None*

A list of columns to exclude from the get item endpoint. By default all columns are included.

show_model_schema: *Optional[Schema] = None*

Override to provide your own marshmallow Schema for JSON to SQLA dumps

show_outer_default_load= *False*

If True, the default load for outer joins will be applied on the get item endpoint. This is useful for when you want to control the load of the many-to-many and many-to-one relationships at the model level. Will apply:

https://docs.sqlalchemy.org/en/14/orm/loading_relationships.html#sqlalchemy.orm.Load.defaultload

show_select_columns: *Optional[List[str]] = None*

A List of column names that will be included on the SQL select. This is useful for including all necessary columns that are referenced by properties listed on *show_columns* without generating N+1 queries.

show_title: *Optional[str] = ''*

Show Title , if not configured the default is 'Show ' with pretty model name

validators_columns: *Optional[Dict[str, Callable]] = None*

Dictionary to add your own marshmallow validators

flask_appbuilder.baseviews

flask_appbuilder.baseviews.expose(*url='/', methods=('GET',)*) [source]

Use this decorator to expose views on your view classes.

Parameters

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

BaseView

class flask_appbuilder.baseviews.BaseView [source]

All views inherit from this class. it's constructor will register your exposed urls on flask as a Blueprint.

This class does not expose any urls, but provides a common base for all views.

Extend this class if you want to expose methods for your own templates

base_permissions: *Optional[List[str]]* = *None*

List with allowed base permission. Use it like this if you want to restrict your view to readonly:

```
class MyView(ModelView):
    base_permissions = ['can_list','can_show']
```

class_permission_name: *str* = *None*

Override class permission name default fallback to self.__class__.__name__

create_blueprint(*appbuilder*, *endpoint=None*, *static_folder=None*) [source]

Create Flask blueprint. You will generally not use it

Parameters

- **appbuilder** – the AppBuilder object
- **endpoint** – endpoint override for this blueprint, will assume class name if not provided
- **static_folder** – the relative override for static folder, if omitted application will use the appbuilder static

default_view: *str* = *'list'*

the default view for this BaseView, to be used with url_for (method name)

exclude_route_methods= *{}*

Does not register routes for a set of builtin ModelView functions. example:

```
class ContactModelView(ModelView):
    datamodel = SQLAInterface(Contact)
    exclude_route_methods = {"delete", "edit"}
```

extra_args= *None*

dictionary for injecting extra arguments into template

classmethod **get_default_url(***kwargs*)** [source]

Returns the url for this class default endpoint

get_init_inner_views() [source]

Sets initialized inner views

get_method_permission(*method_name: str*) → str [source]

Returns the permission name for a method

get_redirect() [source]

Returns the previous url.

get_uninit_inner_views() [source]

Will return a list with views that need to be initialized. Normally related_views from ModelView

include_route_methods= *None*

If defined will assume a white list setup, where all endpoints are excluded except those define on this attribute example:

```
class ContactModelView(ModelView):
    datamodel = SQLAInterface(Contact)
    include_route_methods = {"list"}
```

The previous example will exclude all endpoints except the *list* endpoint

limits= *None*

List of limits for this view.

Use it like this if you want to restrict the rate of requests to a view:

class MyView(ModelView):

limits = [Limit("2 per 5 second")]

or use the decorator @limit.

method_permission_name= *None*

Override method permission names, example:

```
method_permissions_name = {
    'get_list': 'read',
    'get': 'read',
    'put': 'write',
    'post': 'write',
    'delete': 'write'
}
```

previous_class_permission_name= *None*

If set security cleanup will remove all permissions tuples with this name

previous_method_permission_name= *None*

Use same structure as method_permission_name. If set security converge will replace all method permissions by the new ones

render_template(*template*, ***kwargs*) [source]

Use this method on your own endpoints, will pass the extra_args to the templates.

Parameters

- **template** – The template relative path
- **kwargs** – arguments to be passed to the template

route_base= *None*

Override this if you want to define your own relative url

static_folder= *'static'*

The static folder relative location

template_folder= *'templates'*

The template folder relative location

update_redirect() [source]

Call it on your own endpoint's to update the back history navigation. If you bypass it, the next submit or back will go over it.

BaseFormView

class flask_appbuilder.baseviews.BaseFormView [source]

Base class FormView's

default_view: *str* = *'this_form_get'*

The form view default entry endpoint

edit_widget

Form widget to override

alias of `FormWidget`

form= *None*

The WTF form to render

form_columns= *None*

The form columns to include, if empty will include all

form_fieldsets= *None*

Form field sets

form_get(*form*) [source]

Override this method to implement your form processing

form_post(*form*) [source]

Override this method to implement your form processing

Parameters

form – WTForm form

Return None or a flask response to render a custom template or redirect the user

form_title= ''

The form title to be displayed

BaseModelView

class flask_appbuilder.baseviews.BaseModelView(***kwargs*) [source]

The base class of ModelView and ChartView, all properties are inherited Customize ModelView and ChartView overriding this properties

This class supports all the basics for query

base_filters= *None*

Filter the view use: [['column_name',BaseFilter,'value'],]

example:

```
def get_user():
    return g.user

class MyView(ModelView):
    datamodel = SQLAInterface(MyTable)
    base_filters = [['created_by', FilterEqualFunction, get_user],
                    ['name', FilterStartsWith, 'a']]
```

base_order= *None*

Use this property to set default ordering for lists ('col_name','asc|desc'):

```
class MyView(ModelView):
    datamodel = SQLAInterface(MyTable)
    base_order = ('my_column_name','asc')
```

datamodel= *None*

Your sqla model you must initialize it like:

```
class MyView(ModelView):
    datamodel = SQLAInterface(MyTable)
```

label_columns= *None*

Dictionary of labels for your columns, override this if you want different pretify labels

example (will just override the label for name column):

```
class MyView(ModelView):
    datamodel = SQLAInterface(MyTable)
    label_columns = {'name':'My Name Label Override'}
```

search_columns= *None*

List with allowed search columns, if not provided all possible search columns will be used If you want to limit the search (*filter*) columns possibilities, define it with a list of column names from your model:

```
class MyView(ModelView):
    datamodel = SQLAInterface(MyTable)
    search_columns = ['name','address']
```

search_exclude_columns= *None*

List with columns to exclude from search. Search includes all possible columns by default

search_form= *None*

To implement your own add WTF form for Search

search_form_extra_fields= *None*

A dictionary containing column names and a WTForm Form fields to be added to the search form, these fields do not exist on the model itself ex:

search_form_extra_fields = {'some_col':BooleanField('Some Col', default=False)}

search_form_query_rel_fields= *None*

Add Customized query for related fields on search form. Assign a dictionary where the keys are the column names of the related models to filter, the value for each key, is a list of lists with the same format as base_filter {'relation col name':[['Related model col',FilterClass,'Filter Value'],...],...} Add a custom filter to form related fields:

```
class ContactModelView(ModelView):
    datamodel = SQLAModel(Contact, db.session)
    search_form_query_rel_fields = {'group':[['name',FilterStartsWith,'W']]}
```

search_widget

Search widget you can override with your own

alias of `SearchWidget`

BaseCRUDView

class flask_appbuilder.baseviews.BaseCRUDView(***kwargs*) [source]

The base class for ModelView, all properties are inherited Customize ModelView overriding this properties

add_columns= *None*

A list of columns (or model's methods) to be displayed on the add form view. Use it to control the order of the display

add_exclude_columns= *None*

A list of columns to exclude from the add form. By default all columns are included.

add_fieldsets= *None*

add fieldsets django style (look at show_fieldsets for an example)

add_form= *None*

To implement your own, assign WTF form for Add

add_form_extra_fields= *None*

A dictionary containing column names and a WTForm Form fields to be added to the Add form, these fields do not exist on the model itself ex:

add_form_extra_fields = {'some_col':BooleanField('Some Col', default=False)}

add_form_query_rel_fields= *None*

Add Customized query for related fields to add form. Assign a dictionary where the keys are the column names of the related models to filter, the value for each key, is a list of lists with the same format as base_filter {

'relation col name':

[['Related model col', FilterClass, 'Filter Value'],...],...

} Add a custom filter to form related fields:

```
class ContactModelView(ModelView):
    datamodel = SQLAModel(Contact, db.session)
    add_form_query_rel_fields = {'group': [['name', FilterStartsWith, 'W']]}
```

add_template= *'appbuilder/general/model/add.html'*

Your own add jinja2 template for add

add_title= ''

Add Title , if not configured the default is 'Add ' with pretty model name

add_widget

Add widget override

alias of `FormWidget`

description_columns= *None*

Dictionary with column descriptions that will be shown on the forms:

```
class MyView(ModelView):
    datamodel = SQLAModel(MyTable, db.session)

    description_columns = {
        'name': 'your models name column',
        'address': 'the address column'
    }
```

edit_columns= *None*

A list of columns (or model's methods) to be displayed on the edit form view. Use it to control the order of the display

edit_exclude_columns= *None*

A list of columns to exclude from the edit form.

By default all columns are included.

edit_fieldsets= *None*

edit fieldsets django style (look at show_fieldsets for an example)

edit_form= *None*

To implement your own, assign WTF form for Edit

edit_form_extra_fields= *None*

Dictionary to add extra fields to the Edit form using this property

edit_form_query_rel_fields= *None*

Add Customized query for related fields to edit form. Assign a dictionary where the keys are the column names of the related models to filter, the value for each key, is a list of lists with the same format as base_filter {

'relation col name':

[['Related model col', FilterClass, 'Filter Value'],...],...

} Add a custom filter to form related fields:

```
class ContactModelView(ModelView):
    datamodel = SQLAModel(Contact, db.session)
    edit_form_query_rel_fields = {'group':[['name',FilterStartsWith,'W']]}
```

edit_template= *'appbuilder/general/model/edit.html'*

Your own add jinja2 template for edit

edit_title= ''

Edit Title , if not configured the default is ‘Edit ‘ with pretty model name

edit_widget

Edit widget override

alias of `FormWidget`

formatters_columns= *None*

Dictionary of formatter used to format the display of columns

formatters_columns = {‘some_date_col’: lambda x: x.isoformat() }

get_init_inner_views() [source]

Get the list of related ModelViews after they have been initialized

get_uninit_inner_views() [source]

Will return a list with views that need to be initialized. Normally related_views from ModelView

is_get_mutation_allowed() → bool [source]

Check is mutations on HTTP GET methods are allowed. Always called on a request

list_columns= *None*

A list of columns (or model’s methods) to be displayed on the list view. Use it to control the order of the display

list_template= *'appbuilder/general/model/list.html'*

Your own add jinja2 template for list

list_title= ''

List Title, if not configured the default is ‘List ‘ with pretty model name

list_widget

List widget override

alias of `ListWidget`

order_columns= *None*

Allowed order columns

page_size= *25*

Use this property to change default page size

post_add(*item*) [source]

Override this, will be called after update

post_delete(*item*) [source]

Override this, will be called after delete

post_update(*item*) [source]

Override this, will be called after update

pre_add(*item*) [source]

Override this, will be called before add. If an exception is raised by this method, the message is shown to the user and the add operation is aborted.

pre_delete(*item*) [source]

Override this, will be called before delete If an exception is raised by this method, the message is shown to the user and the delete operation is aborted. Because of this behavior, it can be used as a way to implement more complex logic around deletes. For instance allowing only the original creator of the object to delete it.

pre_update(*item*) [source]

Override this, this method is called before the update takes place. If an exception is raised by this method, the message is shown to the user and the update operation is aborted. Because of this behavior, it can be used as a way to implement more complex logic around updates. For instance allowing only the original creator of the object to update it.

prefill_form(*form*, *pk*) [source]

Override this, will be called only if the current action is rendering an edit form (a GET request), and is used to perform additional action to prefill the form.

This is useful when you have added custom fields that depend on the database contents. Fields that were added by name of a normal column or relationship should work out of the box.

example:

```
def prefill_form(self, form, pk):
    if form.email.data:
        form.email_confirmation.data = form.email.data
```

process_form(*form*, *is_created*) [source]

Override this, will be called only if the current action is submitting a create/edit form (a POST request), and is used to perform additional action before the form is used to populate the item.

By default does nothing.

example:

```
def process_form(self, form, is_created):
    if not form.owner:
        form.owner.data = 'n/a'
```

related_views= *None*

List with ModelView classes Will be displayed related with this one using relationship sqlalchemy property:

```
class MyView(ModelView):
    datamodel = SQLAModel(Group, db.session)
    related_views = [MyOtherRelatedView]
```

show_columns= *None*

A list of columns (or model's methods) to be displayed on the show view. Use it to control the order of the display

show_exclude_columns= *None*

A list of columns to exclude from the show view. By default all columns are included.

show_fieldsets= *None*

show fieldsets django style [(<'TITLE'|None>, {'fields':[<F1>,<F2>,...]}),....]

```
class MyView(ModelView):
    datamodel = SQLAModel(MyTable, db.session)

    show_fieldsets = [
        ('Summary', {
            'fields': [
                'name',
                'address',
                'group'
                ]
            }
        ),
        ('Personal Info', {
            'fields': [
                'birthday',
                'personal_phone'
                ],
            'expanded':False
            }
        ),
    ]
```

show_template= *'appbuilder/general/model/show.html'*

Your own add jinja2 template for show

show_title= ''

Show Title , if not configured the default is 'Show ' with pretty model name

show_widget

Show widget override

alias of `ShowWidget`

validators_columns= *None*

Dictionary to add your own validators for forms

flask_appbuilder.views

IndexView

class flask_appbuilder.views.IndexView [source]

A simple view that implements the index for the site

default_view: *str* = *'index'*

the default view for this BaseView, to be used with url_for (method name)

route_base= ''

Override this if you want to define your own relative url

SimpleFormView

class flask_appbuilder.views.SimpleFormView [source]

View for presenting your own forms Inherit from this view to provide some base processing for your customized form views.

Notice that this class inherits from BaseView so all properties from the parent class can be overridden also.

Implement form_get and form_post to implement your form pre-processing and post-processing

PublicFormView

class flask_appbuilder.views.PublicFormView [source]

View for presenting your own forms Inherit from this view to provide some base processing for your customized form views.

Notice that this class inherits from BaseView so all properties from the parent class can be overridden also.

Implement form_get and form_post to implement your form pre-processing and post-processing

ModelView

class flask_appbuilder.views.ModelView(***kwargs*) [source]

This is the CRUD generic view. If you want to automatically implement create, edit, delete, show, and list from your database tables, inherit your views from this class.

Notice that this class inherits from BaseCRUDView and BaseModelView so all properties from the parent class can be overridden.

action(*name*, *pk*) [source]

Action method to handle actions from a show view

action_post() [source]

Action method to handle multiple records selected from a list view

get_action_permission_name(*name: str*) → str [source]

Get the permission name of an action name

post_add_redirect() [source]

Override this function to control the redirect after add endpoint is called.

post_delete_redirect() [source]

Override this function to control the redirect after edit endpoint is called.

post_edit_redirect() [source]

Override this function to control the redirect after edit endpoint is called.

MultipleView

class flask_appbuilder.views.MultipleView(***kwargs*) [source]

Use this view to render multiple views on the same page, exposed on the list endpoint.

example (after defining GroupModelView and ContactModelView):

```
class MultipleViewsExp(MultipleView):
    views = [GroupModelView, ContactModelView]
```

get_init_inner_views() [source]

Sets initialized inner views

get_uninit_inner_views() [source]

Will return a list with views that need to be initialized. Normally related_views from ModelView

list_template= *'appbuilder/general/model/multiple_views.html'*

Override this to implement your own template for the list endpoint

views= *None*

A list of ModelView's to render on the same page

MasterDetailView

class flask_appbuilder.views.MasterDetailView(***kwargs*) [source]

Implements behaviour for controlling two CRUD views linked by PK and FK, in a master/detail type with two lists.

Master view will behave like a left menu:

```
class DetailView(ModelView):
    datamodel = SQLAInterface(DetailTable, db.session)

class MasterView(MasterDetailView):
    datamodel = SQLAInterface(MasterTable, db.session)
    related_views = [DetailView]
```

list_template= *'appbuilder/general/model/left_master_detail.html'*

Your own add jinja2 template for list

list_widget

alias of `ListMasterWidget`

master_div_width= *2*

Set to configure bootstrap class for master grid size

CompactCRUDMixin

class flask_appbuilder.views.CompactCRUDMixin(***kwargs*) [source]

Mix with ModelView to implement a list with add and edit on the same page.

classmethod del_key(*k*) [source]

Matching get method for `set_key`

classmethod get_key(*k*, *default=None*) [source]

Matching get method for `set_key`

classmethod set_key(*k*, *v*) [source]

Allows attaching stateless information to the class using the flask session dict

flask_appbuilder.actions

flask_appbuilder.actions.action(*name*, *text*, *confirmation=None*, *icon=None*, *multiple=True*, *single=True*) [source]

Use this decorator to expose actions

Parameters

- **name** – Action name
- **text** – Action text.

- **confirmation** – Confirmation text. If not provided, action will be executed unconditionally.
- **icon** – Font Awesome icon name
- **multiple** – If true will display action on list view
- **single** – If true will display action on show view

flask_appbuilder.security

BaseSecurityManager

class flask_appbuilder.security.manager.BaseSecurityManager(*appbuilder*) [source]

add_permission(*name*) [source]

Adds a permission to the backend, model permission

Parameters

name – name of the permission: ‘can_add’,’can_edit’ etc...

add_permission_role(*role*, *perm_view*) [source]

Add permission-ViewMenu object to Role

Parameters

- **role** – The role object
- **perm_view** – The PermissionViewMenu object

add_permission_view_menu(*permission_name*, *view_menu_name*) [source]

Adds a permission on a view or menu to the backend

Parameters

- **permission_name** – name of the permission to add: ‘can_add’,’can_edit’ etc...
- **view_menu_name** – name of the view menu to add

add_permissions_menu(*view_menu_name*) [source]

Adds menu_access to menu on permission_view_menu

Parameters

view_menu_name – The menu name

add_permissions_view(*base_permissions*, *view_menu*) [source]

Adds a permission on a view menu to the backend

Parameters

- **base_permissions** –

  list of permissions from view (all exposed methods):

  'can_add','can_edit' etc...

- **view_menu** – name of the view or menu to add

add_register_user(*username*, *first_name*, *last_name*, *email*, *password=''*, *hashed_password=''*) [source]

Generic function to add user registration

add_user(*username*, *first_name*, *last_name*, *email*, *role*, *password=''*) [source]

Generic function to create user

add_view_menu(*name*) [source]

Adds a view or menu to the backend, model view_menu param name:

name of the view menu to add

auth_user_db(*username*, *password*) [source]

Method for authenticating user, auth db style

Parameters

- **username** – The username or registered email address
- **password** – The password, will be tested against hashed password on db

auth_user_ldap(*username*, *password*) [source]

Method for authenticating user with LDAP.

NOTE: this depends on python-ldap module

Parameters

- **username** – the username
- **password** – the password

auth_user_oauth(*userinfo*) [source]

Method for authenticating user with OAuth.

Userinfo

dict with user information (keys are the same as User model columns)

auth_user_oid(*email*) [source]

OpenID user Authentication

Parameters

email – user's email to authenticate

auth_user_remote_user(*username*) [source]

REMOTE_USER user Authentication

Parameters

username – user's username for remote auth

auth_view= *None*

The obj instance for authentication view

authdbview

Override if you want your own Authentication DB view

alias of `AuthDBView`

authldapview

Override if you want your own Authentication LDAP view

alias of `AuthLDAPView`

authoauthview

Override if you want your own Authentication OAuth view

alias of `AuthOAuthView`

authoidview

Override if you want your own Authentication OID view

alias of `AuthOIDView`

authremoteuserview

Override if you want your own Authentication REMOTE_USER view

alias of `AuthRemoteUserView`

count_users() [source]

Generic function to count the existing users

create_db() [source]

Setups the DB, creates admin and public roles if they don't exist.

create_jwt_manager(*app*) → JWTManager [source]

Override to implement your custom JWT manager instance

Parameters

app – Flask app

create_login_manager(*app*) → LoginManager [source]

Override to implement your custom login manager instance

Parameters

app – Flask app

create_state_transitions(*baseviews: List*, *menus: Optional[List[Any]]*) → Dict [source]

Creates a Dict with all the necessary vm/permission transitions

Dict: {

"add": {(<VM>, <PERM>): ((<VM>, PERM), ...)} "del_role_pvm": ((<VM>, <PERM>), ...) "del_views": (<VM>, ...) "del_perms": (<PERM>, ...)

}

Parameters

- **baseviews** – List with all the registered BaseView, BaseApi
- **menus** – List with all the menu entries

Returns

Dict with state transitions

del_permission(*name*) [source]

Deletes a permission from the backend, model permission

Parameters

name – name of the permission: 'can_add','can_edit' etc...

del_permission_role(*role*, *perm_view*) [source]

Remove permission-ViewMenu object to Role

Parameters

- **role** – The role object
- **perm_view** – The PermissionViewMenu object

del_register_user(*register_user*) [source]

Generic function to delete user registration

del_view_menu(*name*) [source]

Deletes a ViewMenu from the backend

Parameters

name – name of the ViewMenu

exist_permission_on_roles(*view_name: str*, *permission_name: str*, *role_ids: List[int]*) → bool [source]

Finds and returns permission views for a group of roles

export_roles(*path: Optional[str] = None*, *indent: Optional[Union[int, str]] = None*) → None [source]

Exports roles to JSON file.

find_permission(*name*) [source]

Finds and returns a Permission by name

find_permission_view_menu(*permission_name*, *view_menu_name*) [source]

Finds and returns a PermissionView by names

find_permissions_view_menu(*view_menu*) [source]

Finds all permissions from ViewMenu, returns list of PermissionView

Parameters

view_menu – ViewMenu object

Returns

list of PermissionView objects

find_register_user(*registration_hash*) [source]

Generic function to return user registration

find_user(*username=None*, *email=None*) [source]

Generic function find a user by it's username or email

find_view_menu(*name*) [source]

Finds and returns a ViewMenu by name

get_all_users() [source]

Generic function that returns all existing users

get_db_role_permissions(*role_id: int*) → List[object] [source]

Get all DB permissions from a role id

get_oauth_token_key_name(*provider*) [source]

Returns the token_key name for the oauth provider if none is configured defaults to oauth_token this is configured using OAUTH_PROVIDERS and token_key key.

get_oauth_token_secret_name(*provider*) [source]

Returns the token_secret name for the oauth provider if none is configured defaults to oauth_secret this is configured using OAUTH_PROVIDERS and token_secret

get_oauth_user_info(*provider*, *resp*) [source]

Since there are different OAuth API's with different ways to retrieve user info

get_public_permissions() [source]

returns all permissions from public role

get_public_role() [source]

returns all permissions from public role

get_role_permissions(*role*) → Set[Tuple[str, str]] [source]

Get all permissions for a certain role

get_roles_from_keys(*role_keys: List[str]*) → Set[None] [source]

Construct a list of FAB role objects, from a list of keys.

NOTE: - keys are things like: "LDAP group DNs" or "OAUTH group names" - we use AUTH_ROLES_MAPPING to map from keys, to FAB role names

Parameters

role_keys – the list of FAB role keys

Returns

a list of RoleModelView

get_user_by_id(*pk*) [source]

Generic function to return user by it's id (pk)

get_user_permissions(*user*) → Set[Tuple[str, str]] [source]

Get all permissions from the current user

get_user_roles(*user*) → List[object] [source]

Get current user roles, if user is not authenticated returns the public role

get_user_roles_permissions(*user*) → Dict[str, List[Tuple[str, str]]] [source]

Utility method just implemented for SQLAlchemy. Take a look to: flask_appbuilder.security.sqla.manager :param user: :return:

has_access(*permission_name: str*, *view_name: str*) → bool [source]

Check if current user or public has access to view or menu

import_roles(*path: str*) → None [source]

Imports roles from JSON file.

is_item_public(*permission_name*, *view_name*) [source]

Check if view has public permissions

Parameters

- **permission_name** – the permission: can_show, can_edit...
- **view_name** – the name of the class view (child of BaseView)

jwt_manager= *None*

Flask-JWT-Extended

lm= *None*

Flask-Login LoginManager

oauth= *None*

Flask-OAuth

oauth_remotes= *None*

OAuth email whitelists

oauth_tokengetter()

OAuth tokengetter function override to implement your own tokengetter method

oauth_user_info_getter(*f*) [source]

Decorator function to be the OAuth user info getter for all the providers, receives provider and response return a dict with the information returned from the provider. The returned user info dict should have it's keys with the same name as the User Model.

Use it like this an example for GitHub

```
@appbuilder.sm.oauth_user_info_getter
def my_oauth_user_info(sm, provider, response=None):
    if provider == 'github':
        me = sm.oauth_remotes[provider].get('user')
        return {'username': me.data.get('login')}
    else:
        return {}
```

oauth_whitelists= *{}*

Initialized (remote_app) providers dict {'provider_name', OBJ }

oid= *None*

Flask-OpenID OpenID

permission_model= *None*

Override to set your own Permission Model

permissionmodelview

alias of `PermissionModelView`

permissionview_model= *None*

Override to set your own PermissionView Model

permissionviewmodelview

alias of `PermissionViewModelView`

register_views() [source]

Generic function to create the security views

registeruser_model= *None*

Override to set your own RegisterUser Model

registeruser_view= *None*

The obj instance for registering user view

registeruserdbview

Override if you want your own register user db view

alias of `RegisterUserDBView`

registerusermodelview

alias of `RegisterUserModelView`

registeruseroauthview

Override if you want your own register user OAuth view

alias of `RegisterUserOAuthView`

registeruseroidview

Override if you want your own register user OpenID view

alias of `RegisterUserOIDView`

reset_password(*userid*, *password*) [source]

Change/Reset a user's password for authdb. Password will be hashed and saved.

Parameters

- **userid** – the user.id to reset the password
- **password** – The clear text password to reset and save hashed on the db

resetmypasswordview

Override if you want your own reset my password view

alias of `ResetMyPasswordView`

resetpasswordview

Override if you want your own reset password view

alias of `ResetPasswordView`

role_model= *None*

Override to set your own Role Model

rolemodelview

alias of `RoleModelView`

security_api

Override if you want your own Security API login endpoint

alias of `SecurityApi`

security_cleanup(*baseviews*, *menus*) [source]

Will cleanup all unused permissions from the database

Parameters

- **baseviews** – A list of BaseViews class
- **menus** – Menu class

security_converge(*baseviews: List*, *menus: Optional[List[Any]]*, *dry=False*) → Dict [source]

Converges overridden permissions on all registered views/api will compute all necessary operations from *class_permissions_name*, *previous_class_permission_name*, method_permission_name`, *previous_method_permission_name* class attributes.

Parameters

- **baseviews** – List of registered views/apis
- **menus** – List of menu items
- **dry** – If True will not change DB

Returns

Dict with the necessary operations (state_transitions)

set_oauth_session(*provider*, *oauth_response*) [source]

Set the current session with OAuth user secrets

update_user(*user*) [source]

Generic function to update user

Parameters

user – User model to update to database

update_user_auth_stat(*user*, *success=True*) [source]

Update user authentication stats upon successful/unsuccessful authentication attempts.

Parameters

- **user** – The identified (but possibly not successfully authenticated) user model
- **success** (*bool or None Defaults to true, if true increments login_count, updates last_login, and resets fail_login_count to 0, if false increments fail_login_count on user model.*) –

user_model= *None*

Override to set your own User Model

user_view= *None*

The obj instance for user view

userdbmodelview

Override if you want your own user db view

alias of `UserDBModelView`

userinfoeditview

Override if you want your own User information edit view

alias of `UserInfoEditView`

userldapmodelview

Override if you want your own user ldap view

alias of `UserLDAPModelView`

useroauthmodelview

Override if you want your own user OAuth view

alias of `UserOAuthModelView`

useroidmodelview

Override if you want your own user OID view

alias of `UserOIDModelView`

userremoteusermodelview

Override if you want your own user REMOTE_USER view

alias of `UserRemoteUserModelView`

userstatschartview

alias of `UserStatsChartView`

viewmenu_model= *None*

Override to set your own ViewMenu Model

viewmenumodelview

alias of `ViewMenuModelView`

BaseRegisterUser

class flask_appbuilder.security.registerviews.BaseRegisterUser [source]

Make your own user registration view and inherit from this class if you want to implement a completely different registration process. If not, just inherit from RegisterUserDBView or RegisterUserOIDView depending on your authentication method. then override SecurityManager property that defines the class to use:

```
from flask_appbuilder.security.registerviews import RegisterUserDBView

class MyRegisterUserDBView(BaseRegisterUser):
    email_template = 'register_mail.html'
    ...


class MySecurityManager(SecurityManager):
    registeruserdbview = MyRegisterUserDBView
```

When instantiating AppBuilder set your own SecurityManager class:

```
appbuilder = AppBuilder(
    app,
    db.session,
     security_manager_class=MySecurityManager
)
```

activation(*activation_hash*) [source]

Endpoint to expose an activation url, this url is sent to the user by email, when accessed the user is inserted and activated

activation_template= *'appbuilder/general/security/activation.html'*

The activation template, shown when the user is activated

add_registration(*username*, *first_name*, *last_name*, *email*, *password=''*) [source]

Add a registration request for the user.

:rtype : RegisterUser

email_subject= *l'Account activation'*

The email subject sent to the user

email_template= *'appbuilder/general/security/register_mail.html'*

The template used to generate the email sent to the user

error_message= *l'Not possible to register you at the moment, try again later'*

The message shown on an unsuccessful registration

false_error_message= *l'Registration not found'*

The message shown on an unsuccessful registration

form_title= *l'Fill out the registration form'*

The form title

message= *l'Registration sent to your email'*

The message shown on a successful registration

route_base= *'/register'*

Override this if you want to define your own relative url

send_email(*register_user*) [source]

Method for sending the registration Email to the user

flask_appbuilder.filemanager

flask_appbuilder.filemanager.get_file_original_name(*name*) [source]

Use this function to get the user's original filename. Filename is concatenated with <UUID>_sep_<FILE NAME>, to avoid collisions. Use this function on your models on an additional function

```
class ProjectFiles(Base):
    id = Column(Integer, primary_key=True)
    file = Column(FileColumn, nullable=False)

    def file_name(self):
        return get_file_original_name(str(self.file))
```

Parameters

name – The file name from model

Returns

Returns the user's original filename removes <UUID>_sep_

Aggr Functions for Group By Charts

flask_appbuilder.models.group.aggregate_count(*items*, *col*) [source]

Function to use on Group by Charts. accepts a list and returns the count of the list's items

flask_appbuilder.models.group.aggregate_avg(*items*, *col*) [source]

Function to use on Group by Charts. accepts a list and returns the average of the list's items

flask_appbuilder.models.group.aggregate_sum(*items*, *col*) [source]

Function to use on Group by Charts. accepts a list and returns the sum of the list's items

flask_appbuilder.charts.views

BaseChartView

class flask_appbuilder.charts.views.BaseChartView(***kwargs*) [source]

This is the base class for all chart views. Use DirectByChartView or GroupByChartView, override their properties

and their base classes

(BaseView, BaseModelView, BaseChartView) to customise your charts

chart_3d= *'true'*

Will display in 3D?

chart_template= *'appbuilder/general/charts/chart.html'*

The chart template, override to implement your own

chart_title= *'Chart'*

A title to be displayed on the chart

chart_type= *'PieChart'*

The chart type PieChart, ColumnChart, LineChart

chart_widget

Chart widget override to implement your own

alias of `ChartWidget`

default_view: *str* = *'chart'*

the default view for this BaseView, to be used with url_for (method name)

group_by_label= *l'Group by'*

The label that is displayed for the chart selection

group_bys= *{}*

New for 0.6.4, on test, don't use yet

search_widget

Search widget override to implement your own

alias of `SearchWidget`

width= *400*

The width

DirectByChartView

class flask_appbuilder.charts.views.DirectByChartView(***kwargs*) [source]

Use this class to display charts with multiple series, based on columns or methods defined on models. You can display multiple charts on the same view.

Default routing point is '/chart'

Setup definitions property to configure the chart

Label

(optional) String label to display on chart selection.

Group

String with the column name or method from model.

Formatter

(optional) function that formats the output of 'group' key

Series

A list of tuples with the aggregation function and the column name to apply the

aggregation

The **definitions** property respects the following grammar:

```
definitions = [
        {
         'label': 'label for chart definition',
         'group': '<COLNAME>'|'<MODEL FUNCNAME>',
         'formatter': <FUNC FORMATTER FOR GROUP COL>,
         'series': ['<COLNAME>'|'<MODEL FUNCNAME>',...]
        }, ...
      ]
```

example:

```
class CountryDirectChartView(DirectByChartView):
    datamodel = SQLAInterface(CountryStats)
    chart_title = 'Direct Data Example'

    definitions = [
        {
            'label': 'Unemployment',
            'group': 'stat_date',
            'series': ['unemployed_perc',
                'college_perc']
        }
    ]
```

ProcessClass

alias of `DirectProcessData`

GroupByChartView

class flask_appbuilder.charts.views.GroupByChartView(***kwargs*) [source]

ProcessClass

alias of `GroupByProcessData`

chart_template= *'appbuilder/general/charts/jsonchart.html'*

The chart template, override to implement your own

chart_type= *'ColumnChart'*

The chart type PieChart, ColumnChart, LineChart

chart_widget

alias of `DirectChartWidget`

definitions= *[]*

These charts can display multiple series, based on columns or methods defined on models. You can display multiple charts on the same view. This data can be grouped and aggregated has you like.

Label

(optional) String label to display on chart selection.

Group

String with the column name or method from model.

Formatter

(optional) function that formats the output of 'group' key

Series

A list of tuples with the aggregation function and the column name to apply the aggregation

```
[{
    'label': 'String',
    'group': '<COLNAME>'|'<FUNCNAME>'
    'formatter: <FUNC>
    'series': [(<AGGR FUNC>, <COLNAME>|'<FUNCNAME>'),...]
    }
]
```

example:

```
class CountryGroupByChartView(GroupByChartView):
    datamodel = SQLAInterface(CountryStats)
    chart_title = 'Statistics'

definitions = [
    {
        'label': 'Country Stat',
        'group': 'country',
        'series': [(aggregate_avg, 'unemployed_perc'),
               (aggregate_avg, 'population'),
               (aggregate_avg, 'college_perc')
              ]
    }
]
```

get_group_by_class(*definition*) [source]

intantiates the processing class (Direct or Grouped) and returns it.

(Deprecated) ChartView

class flask_appbuilder.charts.views.ChartView(***kwargs*) [source]

DEPRECATED

Provides a simple (and hopefully nice) way to draw charts on your application.

This will show Google Charts based on group by of your tables.

(Deprecated) TimeChartView

class flask_appbuilder.charts.views.TimeChartView(***kwargs*) [source]

DEPRECATED

Provides a simple way to draw some time charts on your application.

This will show Google Charts based on count and group by month and year for your tables.

chart_template= *'appbuilder/general/charts/chart_time.html'*

The chart template, override to implement your own

chart_type= *'ColumnChart'*

The chart type PieChart, ColumnChart, LineChart

(Deprecated) DirectChartView

class flask_appbuilder.charts.views.DirectChartView(***kwargs*) [source]

DEPRECATED

This class is responsible for displaying a Google chart with direct model values. Chart widget uses json. No group by is processed, example:

```
class StatsChartView(DirectChartView):
    datamodel = SQLAInterface(Stats)
    chart_title = lazy_gettext('Statistics')
    direct_columns = {'Some Stats': ('X_col_1', 'stat_col_1', 'stat_col_2'),
                      'Other Stats': ('X_col2', 'stat_col_3')}
```

chart_type= *'ColumnChart'*

The chart type PieChart, ColumnChart, LineChart

chart_widget

alias of `DirectChartWidget`

flask_appbuilder.models.mixins

class flask_appbuilder.models.mixins.BaseMixin [source]

class flask_appbuilder.models.mixins.AuditMixin [source]

Mixin for models, adds 4 columns to stamp, time and user on creation and modification will create the following columns:

Created on

Changed on

Created by

Changed by

Extra Columns

class flask_appbuilder.models.mixins.FileColumn(**args*, ***kwargs*) [source]

Extends SQLAlchemy to support and mostly identify a File Column

impl

alias of `Text`

class flask_appbuilder.models.mixins.ImageColumn(*thumbnail_size=(20, 20, True)*, *size=(100, 100, True)*, ***kw*) [source]

Extends SQLAlchemy to support and mostly identify an Image Column

impl

alias of `Text`

Generic Data Source (Beta)

flask_appbuilder.models.generic

class flask_appbuilder.models.generic.GenericColumn(*col_type*, *primary_key=False*, *unique=False*, *nullable=False*) [source]

class flask_appbuilder.models.generic.GenericModel(***kwargs*) [source]

Generic Model class to define generic purpose models to use with the framework.

Use GenericSession much like SQLAlchemy's Session Class. Extend GenericSession to implement specific engine features.

Define your models like:

```
class MyGenericModel(GenericModel):
    id = GenericColumn(int, primary_key=True)
    age = GenericColumn(int)
    name = GenericColumn(str)
```

class flask_appbuilder.models.generic.GenericSession [source]

This class is a base, you should subclass it to implement your own generic data source.

Override at least the **all** method.

GenericSession will implement filter and orders based on your data generation on the **all** method.

all() [source]

SQLA like 'all' method, will populate all rows and apply all filters and orders to it.

clear() [source]

Deletes the entire store

delete_all(*model_cls*) [source]

Deletes all objects of type model_cls

get(*pk*) [source]

Returns the object for the key Override it for efficiency.

query(*model_cls*) [source]

SQLAlchemy query like method

Version Migration

Migrating to 1.9.0

If you are using OAuth for authentication, this release will break your logins. This break is due to two reasons

One:

There was a security issue when using the default builtin information getter for the providers (see github: Prevent masquerade attacks through oauth providers #472) This fix will prepend the provider to the user id. So your usernames will look like 'google_<USER_ID>'

Two:

For google OAuth we migrated from the old and deprecated google plus API to OAuth2/v2, the old User.username field was based on the Google Plus display name, and now is based on a Google user_id.

In order to upgrade without breaking, you can override the current default OAuth information getter using something like this:

```
@appbuilder.sm.oauth_user_info_getter
def get_oauth_user_info(sm, provider, response=None):
# for GITHUB
    if provider == 'github' or provider == 'githublocal':
        me = sm.oauth_remotes[provider].get('user')
        return {'username': me.data.get('login')}
    # for twitter
    if provider == 'twitter':
        me = sm.oauth_remotes[provider].get('account/settings.json')
        return {'username': me.data.get('screen_name', '')}
    # for linkedin
    if provider == 'linkedin':
        me = sm.oauth_remotes[provider].get('people/~:(id,email-address,first-
name,last-name)?format=json')
        return {'username': me.data.get('id', ''),
                'email': me.data.get('email-address', ''),
                'first_name': me.data.get('firstName', ''),
                'last_name': me.data.get('lastName', '')}
    # for Google
    if provider == 'google':
        me = sm.oauth_remotes[provider].get('userinfo')
        return {'username': me.data.get('id', ''),
                'first_name': me.data.get('given_name', ''),
                'last_name': me.data.get('family_name', ''),
                'email': me.data.get('email', '')}
```

There was a Fix for the **oauth_user_info_getter** decorator also, now it will obey the doc definition.

Any help you need feel free to submit an Issue!

Migrating to 1.8.0

On this release flask-appbuilder supports python 3.5, and returned to flask-babel original package (stopped using the fork flask-babelpkg for multiple translation directories).

You can and should, uninstall flask-babelpkg from your package list and change all your

imports from:

```
from flask_babelpkg import ...
```

To:

```
from flask_babel import ...
```

Migrating from 1.2.X to 1.3.X

There are some breaking features:

1 - Security models have changed, users can have multiple roles, not just one. So you have to upgrade your db.

- The security models schema have changed.

    If you are using sqlite, mysql, pgsql, mssql or oracle, use the following procedure:

    1 - *Backup your DB*.

    2 - If you haven't already, upgrade to flask-appbuilder 1.3.0.

    3 - Issue the following commands, on your project folder where config.py exists:

    ```
    $ cd /your-main-project-folder/
    $ fabmanager upgrade-db
    ```

    4 - Test and Run (if you have a run.py for development)

    ```
    $ fabmanager run
    ```

    For **sqlite** you'll have to drop role_id columns and FK yourself. follow the script instructions to finish the upgrade.

2 - Security. If you were already extending security, this is even more encouraged from now on, but internally many things have changed. So, modules have changes and changed place, each backend engine will have it's SecurityManager, and views are common to all of them.

Change:

from:

```
from flask_appbuilder.security.sqla.views import UserDBModelView
from flask_appbuilder.security.manager import SecurityManager
```

to:

```
from flask_appbuilder.security.views import UserDBModelView
from flask_appbuilder.security.sqla.manager import SecurityManager
```

3 - SQLAInteface, SQLAModel. If you were importing like the following, change:

from:

```
from flask_appbuilder.models import SQLAInterface
```

to:

```
from flask_appbuilder.models.sqla.interface import SQLAInterface
```

4 - Filters, filters import moved:

to:

```
from flask_appbuilder.models.sqla.filters import FilterStartsWith, FilterEqualFunction,
FilterEqual
```

5 - Filters, filtering relationship fields (rendered with select2) changed:

from:

```
edit_form_query_rel_fields = [('group',
                               SQLAModel(Model1, self.db.session),
                               [['field_string', FilterEqual, 'G2']]
                              )
                             ]
```

to:

```
edit_form_query_rel_fields = {'group':[['field_string', FilterEqual, 'G2']]}
```

Migrating from 1.1.X to 1.2.X

There is a breaking feature, change your filters imports like this:

from:

```
flask_appbuilder.models.base import Filters, BaseFilter, BaseFilterConverter
flask_appbuilder.models.filters import FilterEqual, FilterRelation ....
```

to:

```
flask_appbuilder.models.filters import Filters, BaseFilter, BaseFilterConverter
flask_appbuilder.models.sqla.filter import FilterEqual, FilterRelation ....
```

Migrating from 0.9.X to 0.10.X

This new version has NO breaking features, all your code will work, unless you are hacking directly onto SQLAModel, Filters, DataModel etc.

But, to keep up with the changes, you should change these:

```
from flask_appbuilder.models.datamodel import SQLAModel
from flask_appbuilder.models.filters import FilterEqual, FilterContains
```

to:

```
from flask_appbuilder.models.sqla.interface import SQLAInterface
from flask_appbuilder.models.sqla.filters import FilterEqual, FilterContains
```

Migrating from 0.8.X to 0.9.X

This new version has a breaking feature, the way you initialize AppBuilder (former BaseApp)

has changed. internal retro compatibility was created, but many things have changed

1 - Initialization of AppBuilder (BaseApp) has changed, pass session not SQLAlchemy *db* object. this is the breaking feature.

from (__init__.py)

```
BaseApp(app, db)
```

to (__init__.py)

```
AppBuilder(app, db.session)
```

2 - 'BaseApp' changed to 'AppBuilder'. Has you already noticed on 1.

3 - BaseApp or now AppBuilder will not automatically create your models, after declaring them just invoke create_db method:

```
appbuilder.create_db()
```

4 - Change your models inheritance

from:

```
class MyModel(Model):
    id = Column(Integer, primary_key=True)
    first_name = Column(String(64), nullable=False)
```

to:

```
class MyModel(Model):
    id = Column(Integer, primary_key=True)
    first_name = Column(String(64), nullable=False)
```

5 - Although you're not obligated, you should not directly use your flask.ext.sqlalchemy class SQLAlchemy. Use F.A.B. SQLA class instead, read the docs to know why.

from (__init__.py):

```
from flask import Flask
from flask.ext.sqlalchemy import SQLAlchemy
from flask_appbuilder.baseapp import BaseApp


app = Flask(__name__)
app.config.from_object('config')
db = SQLAlchemy(app)
baseapp = BaseApp(app, db)
```

to (__init__.py):

```
from flask import Flask
from flask_appbuilder import SQLA, AppBuilder

app = Flask(__name__)
app.config.from_object('config')
db = SQLA(app)
appbuilder = AppBuilder(app, db.session)
```

Migrating from 0.6.X to 0.7.X

This new version has some breaking features. You don't have to change any code, main breaking changes are:

- The security models schema have changed.

If you are using sqlite, mysql or pgsql, use the following procedure:

1 - *Backup your DB*.

2 - If you haven't already, upgrade to flask-appbuilder 0.7.0.

3 - Issue the following commands, on your project folder where config.py exists:

```
cd /your-main-project-folder/
wget https://raw.github.com/dpgaspar/Flask-AppBuilder/master/bin/
migrate_db_0.7.py
python migrate_db_0.7.py
wget https://raw.github.com/dpgaspar/Flask-AppBuilder/master/bin/
hash_db_password.py
python hash_db_password.py
```

4 - Test and Run (if you have a run.py for development)

```
python run.py
```

If not (DB is not sqlite, mysql or pgsql), you will have to alter the schema yourself. use the following procedure:

1 - *Backup your DB*.

2 - If you haven't already, upgrade to flask-appbuilder 0.7.0.

3 - issue the corresponding DDL commands to:

ALTER TABLE ab_user MODIFY COLUMN password VARCHAR(256)

ALTER TABLE ab_user ADD COLUMN login_count INTEGER

ALTER TABLE ab_user ADD COLUMN created_on DATETIME

ALTER TABLE ab_user ADD COLUMN changed_on DATETIME

ALTER TABLE ab_user ADD COLUMN created_by_fk INTEGER

ALTER TABLE ab_user ADD COLUMN changed_by_fk INTEGER

ALTER TABLE ab_user ADD COLUMN last_login DATETIME

ALTER TABLE ab_user ADD COLUMN fail_login_count INTEGER

4 - Then hash your passwords:

```
wget https://raw.github.com/dpgaspar/Flask-AppBuilder/master/bin/
hash_db_password.py
python hash_db_password.py
```

- All passwords are kept on the database hashed, so all your passwords will be hashed by the framework.
- **Please *backup* your DB before altering the schema, if you feel lost please post an issue on github**

  https://github.com/dpgaspar/Flask-AppBuilder/issues?state=open

Migrating from 0.5.X to 0.6.X

This new version has some breaking features, that I hope will be easily changeable on your code.

If you feel lost please post an issue on github: https://github.com/dpgaspar/Flask-AppBuilder/issues?state=open

If you're using the **related_views** attribute on ModelView classes, you must not instantiate

the related classes. This is the correct form, it will be less memory and cpu resource consuming.

From this:

```
class MyView(GeneralView):
    datamodel = SQLAModel(Group, db.session)
    related_views = [MyOtherView()]
```

Change to this:

```
class MyView(GeneralView):
    datamodel = SQLAModel(Group, db.session)
    related_views = [MyOtherView]
```

Migrating from 0.2.X to 0.3.X

This new version (0.3.X) has many internal changes, if you feel lost please post an issue on github https://github.com/dpgaspar/Flask-AppBuilder/issues?state=open

All direct imports from your 'app' directory were removed, so there is no obligation in using the base AppBuilder-Skeleton.

Security tables have changed their names, AppBuilder will automatically migrate all your data to the new tables.

1 - Change your BaseApp initialization (views.py)

From this:

```
baseapp = BaseApp(app)
```

Change to this:

```
baseapp = BaseApp(app, db)
```

2 - Remove from OpenID and Login initialization (__init__.py)

From this:

```
app = Flask(__name__)
app.config.from_object('config')
db = SQLAlchemy(app)
babel = Babel(app)
lm = LoginManager()
lm.init_app(app)
lm.login_view = 'login'
oid = OpenID(app, os.path.join(basedir, 'tmp'))

from app import models, views
```

Change to this:

```
app = Flask(__name__)
app.config.from_object('config')
db = SQLAlchemy(app)

from app import models, views
```

Migrating from 0.1.X to 0.2.X

It's very simple, change this:

```
baseapp = BaseApp(app)
baseapp.add_view(GroupGeneralView, "List Groups","/groups/list","th-large","Contacts")
baseapp.add_view(PersonGeneralView, "List Contacts","/persons/
list","earphone","Contacts")
baseapp.add_view(PersonChartView, "Contacts Chart","/persons/
chart","earphone","Contacts")
```

To this:

```
baseapp = BaseApp(app)
baseapp.add_view(GroupGeneralView(), "List Groups","/groups/list","th-
large","Contacts")
baseapp.add_view(PersonGeneralView(), "List Contacts","/persons/
list","earphone","Contacts")
baseapp.add_view(PersonChartView(), "Contacts Chart","/persons/
chart","earphone","Contacts")
```

Small change, you just have to instantiate your classes.

BREAKING CHANGES

Version 4.0.0

- Drops python 3.6 support
- Removed config key *AUTH_STRICT_RESPONSE_CODES*, it's always strict now.
- Removes *Flask-OpenID* dependency (you can install it has an extra dependency *pip install flask-appbuilder[openid]*)
- Major version bumps on following packages

Flask from 1.X to 2.X

Breaking changes: https://flask.palletsprojects.com/en/2.0.x/changes/#version-2-0-0

flask-jwt-extended 3.X to 4.X:

Breaking changes: https://flask-jwt-extended.readthedocs.io/en/stable/v4_upgrade_guide/

Jinja2 2.X to 3.X

Breaking changes: https://jinja.palletsprojects.com/en/3.0.x/changes/#version-3-0-0

Werkzeug 1.X to 2.X

https://werkzeug.palletsprojects.com/en/2.0.x/changes/#version-2-0-0

The following packages are probably not impactful to you:

pyJWT 1.X to 2.X:

Breaking changes: https://pyjwt.readthedocs.io/en/stable/changelog.html#v2-0-0

Click 7.X to 8.X:

Breaking changes: https://click.palletsprojects.com/en/8.0.x/changes/#version-8-0-0

itsdangerous 1.X to 2.X

Breaking changes: https://github.com/pallets/itsdangerous/blob/main/CHANGES.rst#version-200

Version 3.0.0 (OAuth)

Major version 3, changed it's **OAuth** dependency from flask-oauth to authlib, due to this OAuth configuration changed:

Before:

```
```

```
OAUTH_PROVIDERS = [
    {'name':'google', 'icon':'fa-google', 'token_key':'access_token',
        'remote_app': {
            'consumer_key':'GOOGLE KEY',
            'consumer_secret':'GOOGLE SECRET',
            'base_url':'https://www.googleapis.com/oauth2/v2/',
            'request_token_params':{
              'scope': 'email profile'
            },
            'request_token_url':None,
            'access_token_url':'https://accounts.google.com/o/oauth2/token',
            'authorize_url':'https://accounts.google.com/o/oauth2/auth'}
    }
]
```

Now:

```
OAUTH_PROVIDERS = [
    {'name':'google', 'icon':'fa-google', 'token_key':'access_token',
        'remote_app': {
            'client_id':'GOOGLE KEY',
            'client_secret':'GOOGLE SECRET',
            'api_base_url':'https://www.googleapis.com/oauth2/v2/',
            'client_kwargs':{
              'scope': 'email profile'
            },
            'request_token_url':None,
            'access_token_url':'https://accounts.google.com/o/oauth2/token',
            'authorize_url':'https://accounts.google.com/o/oauth2/auth'}
    }
]
```

Also make sure you change your dependency for flask-oauth to authlib