invenio-admin Documentation

Release 1.1.1

CERN

Dec 26, 2018

Contents

Invenio-Admin allows for easy administration of any Invenio installation, simple data curation and execution of actions.

- Provides default administration panel using Flask-Admin.
- Easily extendible from other Invenio modules using entry points.

Further documentation at: https://invenio-admin.readthedocs.io/

CHAPTER 1

User's Guide

This part of the documentation will show you how to get started in using Invenio-Admin.

1.1 Installation

Invenio-Admin is on PyPI so all you need is:

```
$ pip install invenio-admin
```

1.2 Configuration

Configuration for Invenio-Admin.

invenio_admin.config.**ADMIN_APPNAME = 'Invenio'**
Name of the Flask-Admin app (also the page title of admin panel).

invenio_admin.config.**ADMIN_BASE_TEMPLATE = None**
Admin panel base template. By default (None) uses the Flask-Admin template.

invenio_admin.config.**ADMIN_LOGIN_ENDPOINT = 'security.login'**
Endpoint name of the login view. Anonymous users trying to access admin panel will be redirected to this endpoint.

invenio_admin.config.**ADMIN_LOGOUT_ENDPOINT = 'security.logout'**
Endpoint name of logout view.

invenio_admin.config.**ADMIN_PERMISSION_FACTORY = 'invenio_admin.permissions.admin_permissio**
Permission factory for the admin views.

invenio_admin.config.**ADMIN_TEMPLATE_MODE = 'bootstrap3'**
Flask-Admin template mode. Either bootstrap2 or bootstrap3.

1.3 Usage

Administration interface for Invenio applications.

Invenio-Admin is an optional component of Invenio, responsible for registering and customizing the administration panel for model views and user-defined admin pages. The module uses standard Flask-Admin features and assumes very little about other components installed within a given Invenio instance.

1.3.1 Quick start

This section presents a minimal working example of the Invenio-Admin.

First, let us create a new Flask application:

```
>>> from flask import Flask
>>> app = Flask('DinerApp')
```

and load the Invenio-DB and Invenio-Admin extensions:

```
>>> from invenio_db import InvenioDB
>>> from invenio_admin import InvenioAdmin
>>> ext_db = InvenioDB(app)
>>> ext_admin = InvenioAdmin(app, view_class_factory=lambda x: x)
```

Warning: We use the `view_class_factory` parameter above to disable the authentication to the admin panel, in order to simplify this tutorial. Do not use this for production systems, as you will grant access to the admin panel to anonymous users!

In full application with an authentication system in place, it is sufficient to instantiate the extension like:

```
ext_admin = InvenioAdmin(app)
```

Let's now define a simple model with a model view, and one base view:

```
>>> from invenio_db import db
>>> from flask_admin.contrib.sqla import ModelView
>>> from flask_admin.base import BaseView, expose
>>> class Lunch(db.Model):
...     __tablename__ = 'diner_lunch'
...     id = db.Column(db.Integer, primary_key=True)
...     meal_name = db.Column(db.String(255), nullable=False)
...     is_vegetarian = db.Column(db.Boolean(name='is_v'), default=False)
...
>>> class LunchModelView(ModelView):
...     can_create = True
...     can_edit = True
...
>>> class MenuCard(BaseView):
...     @expose('/')
...     def index(self):
...         return "HelloMenuCard!"
```

and register them in the admin extension:

```
>>> ext_admin.register_view(LunchModelView, Lunch, db.session)
>>> ext_admin.register_view(MenuCard)
```

Finally, initialize the database and run the development server:

```
>>> from sqlalchemy_utils.functions import create_database
>>> app.config.update(SQLALCHEMY_DATABASE_URI='sqlite:///test.db',
...     SECRET_KEY='SECRET')
...
>>> with app.app_context():
...     create_database(db.engine.url)
...     db.create_all()
>>> app.run()
```

You should now be able to access the admin panel http://localhost:5000/admin.

1.3.2 Adding admin views from Invenio module

In real-world scenarios you will most likley want to add an admin view for your custom models from within the Invenio module or an Invenio overlay application. Instead of registering it directly on the application as in the example above, you can use entry points to register those automatically.

Defining admin views

Let us start with defining the `admin.py` file inside your module or overlay, which will contain all admin-related classes and functions. For example, assuming a `Invenio-Diner` module, the file could reside in:

`invenio-diner/invenio_diner/admin.py`.

In this example we will define two model views for two database models and one separate base view for statistics page. The content of the file is as follows:

```
# invenio-diner/invenio_diner/admin.py
from flask_admin.base import BaseView, expose
from flask_admin.contrib.sqla import ModelView
from invenio_db import db
from .models import Snack, Breakfast

class SnackModelView(ModelView):
    can_create = True
    can_edit = True
    can_view_details = True
    column_list = ('id', 'name', 'price', )

class BreakfastModelView(ModelView):
    can_create = False
    can_edit = False
    can_view_details = True
    column_searchable_list = ('id', 'toast', 'eggs', 'bacon' )

class DinerStats(BaseView):
    @expose('/')
    def index(self):
        return "Welcome to the Invenio-Diner statistics page!"
```

(continues on next page)

(continued from previous page)

```
    @expose('/sales/')
    def sales(self):
        return "You have served 0 meals!"

snack_adminview = {
    'view_class': Snack,
    'args': [SnackModelView, db.session],
    'kwargs': {'category': 'Diner'},
}

breakfast_adminview = {
    'view_class': Breakfast,
    'args': [BreakfastModelView, db.session],
    'kwargs': {'category': 'Diner'},
}

stats_adminview = {
    'view_class': DinerStats,
    'kwargs': {'name': 'Invenio Diner Stats'},
}

__all__ = (
    'snack_adminview',
    'breakfast_adminview',
    'stats_adminview',
)
```

Note: You have to define a dictionary for each BaseView and Model-ModelView pairs (see `stats_adminview`, `snack_adminview` and `breakfast_adminview` above) in order to have the admin views automatically registered via entry points (see next section).

The `args` and `kwargs` keys in the dictionaries are passed to the constructor of the view class once it is intialized.

Registering the entry point

The default way of adding admin views to the admin panel is though setuptools' entry point discovery. To do that, a newly created module has to register an entry point under the group `invenio_admin.views` inside its `setup.py` as follows:

```
# invenio-diner/setup.py
setup(
  entry_points={
    'invenio_admin.views': [
      'invenio_diner_snack = invenio_diner.admin.snack_adminview',
      'invenio_diner_breakfast = invenio_diner.admin.breakfast_adminview',
      'invenio_diner_stats = invenio_diner.admin.stats_adminview',
    ],
  },
)
```

1.3.3 Authentication and authorization

By default Invenio-Admin protects the admin views from unauthenticated users with Flask-Login and restricts the access on a per-permission basis using Flask-Security. In order to login to a Invenio-Admin panel the user needs to be authenticated using Flask-Login and have a Flask-Security identity which provides the `ActionNeed('admin-access')`.

Note: If you want to use a custom permission rule, you can easily specify your own permission factory in the configuration variable *invenio_admin.config.ADMIN_PERMISSION_FACTORY*.

For more information, see the default factory: *invenio_admin.permissions.admin_permission_factory()* and how the the view is using it: *invenio_admin.views.protected_adminview_factory()*

1.3.4 Styling

At core, Invenio-Admin uses Flask-Admin for rendering the admin panel and all of its views. All of the features for defining the ModelViews and BaseViews can be found in the official Flask-Admin documentation. Nonetheless, we will mention some of the ones that were already made easy to use directly in Invenio-Admin.

Custom database type filters

Non-basic data types can be made easier to search for and filter using type filters. This way, fields of certain type that is not searchable by default can be extended with that functionality. For example see a built-in UUID filter *invenio_admin.filters.UUIDEqualFilter*. You can enable the custom fields filters, by setting a variable `filter_converter` on the ModelView class. See an example of a custom filter converter in *invenio_admin.filters.FilterConverter*.

Assuming that the `id` field in `Snack` model from the example above is a UUID-type field, you could enable the UUID filtering on this model as follows:

```
from invenio_admin.filters import FilterConverter

class SnackModelView(ModelView):
    filter_converter = FilterConverter()  # Add filter converter
    can_create = True
    can_edit = True
    can_view_details = True
    column_list = ('id', 'name', 'price', )
```

Base template

Styling of the administration interface can be changed via the configuration variable `ADMIN_BASE_TEMPLATE`.

If Invenio-Theme is installed, `ADMIN_BASE_TEMPLATE` is automatically set to use the AdminLTE theme which provides an extra configuration variable `ADMIN_UI_SKIN` which controls the AdminLTE skin (e.g. `skin-blue` or `skin-black`). See AdminLTE documentation for details on supported skins.

If Invenio-Theme is not installed the default Flask-Admin templates will be used (based on Bootstrap).

View template mode

Flask-Admin view templates (forms etc.) can either use Bootstap 2 or 3. By default the template mode is set to Bootstrap 3 but can be controlled through `ADMIN_TEMPLATE_MODE` configuration variable.

1.4 Example application

First, install Invenio-Admin, setup the application and load fixture data by running:

```
$ pip install -e .[all]
$ cd examples
$ ./app-setup.sh
$ ./app-fixtures.sh
```

Next, start the development server:

```
$ export FLASK_APP=app.py FLASK_DEBUG=1
$ flask run
```

and open the example application in your browser:

```
$ open http://127.0.0.1:5000/
```

To reset the example application run:

```
$ ./app-teardown.sh
```

CHAPTER 2

API Reference

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

2.1 API Docs

Invenio-Admin Flask extension.

class invenio_admin.ext.**InvenioAdmin**(*app=None*, ***kwargs*)

Invenio-Admin extension.

Invenio-Admin extension initialization.

Parameters

- **app** – The Flask application. (Default: `None`)
- **kwargs** – Passed to *init_app()*.

init_app(*app*, *entry_point_group='invenio_admin.views'*, *permission_factory=None*, *view_class_factory=<function protected_adminview_factory>*, *index_view_class=<class 'flask_admin.base.AdminIndexView'>*)

Flask application initialization.

Parameters

- **app** – The Flask application.
- **entry_point_group** – Name of entry point group to load views/models from. (Default: `'invenio_admin.views'`)
- **permission_factory** – Default permission factory to use when protecting an admin view. (Default: *admin_permission_factory()*)
- **view_class_factory** – Factory for creating admin view classes on the fly. Used to protect admin views with authentication and authorization. (Default: *protected_adminview_factory()*)

- **index_view_class** – Specify administrative interface index page. (Default: flask_admin.base.AdminIndexView)
- **kwargs** – Passed to flask_admin.base.Admin.

Returns Extension state.

static init_config(*app*)
Initialize configuration.

Parameters **app** – The Flask application.

2.1.1 Views

Admin view class factory for creating protected admin views on-the-fly.

invenio_admin.views.**init_menu**()
Initialize menu before first request.

invenio_admin.views.**protected_adminview_factory**(*base_class*)
Factory for creating protected admin view classes.

The factory will ensure that the admin view will check if a user is authenticated and has the necessary permissions (as defined by the permission factory). The factory creates a new class using the provided class as base class and overwrites `is_accessible()` and `inaccessible_callback()` methods. Super is called for both methods, so the base class can implement further restrictions if needed.

Parameters **base_class** (flask_admin.base.BaseView) – Class to use as base class.

Returns Admin view class which provides authentication and authorization.

2.1.2 Forms

Flask-Admin form utilities.

class invenio_admin.forms.**LazyChoices**(*func*)
Lazy form choices.

Initialize lazy choices.

Parameters **func** – Function returning an iterable of choices.

2.1.3 Filters

Flask-Admin filter utilities.

class invenio_admin.filters.**FilterConverter**
Filter converter for dealing with UUIDs and variants.

conv_uuid(*column*, *name*, ***kwargs*)
Convert UUID filter.

conv_variant(*column*, *name*, ***kwargs*)
Convert variants.

class invenio_admin.filters.**UUIDEqualFilter**(*column*, *name*, *options=None*, *data_type=None*)
UUID aware filter.

Constructor.

Parameters

- **column** – Model field
- **name** – Display name
- **options** – Fixed set of options
- **data_type** – Client data type

apply(*query*, *value*, *alias*)
Convert UUID.

Parameters

- **query** – SQLAlchemy query object.
- **value** – UUID value.
- **alias** – Alias of the column.

Returns Filtered query matching the UUID value.

2.1.4 Permissions

Permissions for Invenio-Admin.

invenio_admin.permissions.**action_admin_access = Need(method='action', value='admin-access')**
Define the action needed by the default permission factory.

invenio_admin.permissions.**admin_permission_factory**(*admin_view*)
Default factory for creating a permission for an admin.

It tries to load a `invenio_access.permissions.Permission` instance if *invenio_access* is installed. Otherwise, it loads a `flask_principal.Permission` instance.

Parameters **admin_view** – Instance of administration view which is currently being protected.

Returns Permission instance.

CHAPTER 3

Additional Notes

Notes on how to contribute, legal information and changes are here for the interested.

3.1 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

3.1.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/inveniosoftware/invenio-admin/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

Invenio-Admin could always use more documentation, whether as part of the official Invenio-Admin docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/inveniosoftware/invenio-admin/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

3.1.2 Get Started!

Ready to contribute? Here's how to set up *invenio-admin* for local development.

1. Fork the *inveniosoftware/invenio-admin* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/invenio-admin.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv invenio-admin
$ cd invenio-admin/
$ pip install -e .[all]
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass tests:

```
$ ./run-tests.sh
```

   The tests will provide you with test coverage and also check PEP8 (code style), PEP257 (documentation), flake8 as well as build the Sphinx documentation and run doctests.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -s
    -m "component: title without verbs"
    -m "* NEW Adds your new feature."
    -m "* FIX Fixes an existing issue."
    -m "* BETTER Improves and existing feature."
    -m "* Changes something that should not be visible in release notes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

3.1.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests and must not decrease test coverage.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring.
3. The pull request should work for Python 2.7, 3.3, 3.4 and 3.5. Check https://travis-ci.org/inveniosoftware/invenio-admin/pull_requests and make sure that the tests pass for all supported Python versions.

3.2 Changes

Version 1.1.1 (released 2018-12-26)

- Minimum version of Flask-Admin bumped to v1.5.3 due to Cross-Site Scripting vulnerability in previous versions.

Version 1.1.0 (released 2018-12-14)

- Changed to using Webpack for static assets management instead of using AMD/RequireJS.

Version 1.0.0 (released 2018-03-23)

- Initial public release.

3.3 License

3.4 Contributors

- Alexander Ioannidis
- Alizee Pace
- Esteban J. G. Gabancho
- Jiri Kuncar
- Krzysztof Nowak
- Lars Holm Nielsen
- Leonardo Rossi
- Nicola Tarocco
- Nikos Filippakis
- Sami Hiltunen
- Tibor Simko

Python Module Index

i

Index