aiohttp admin 2

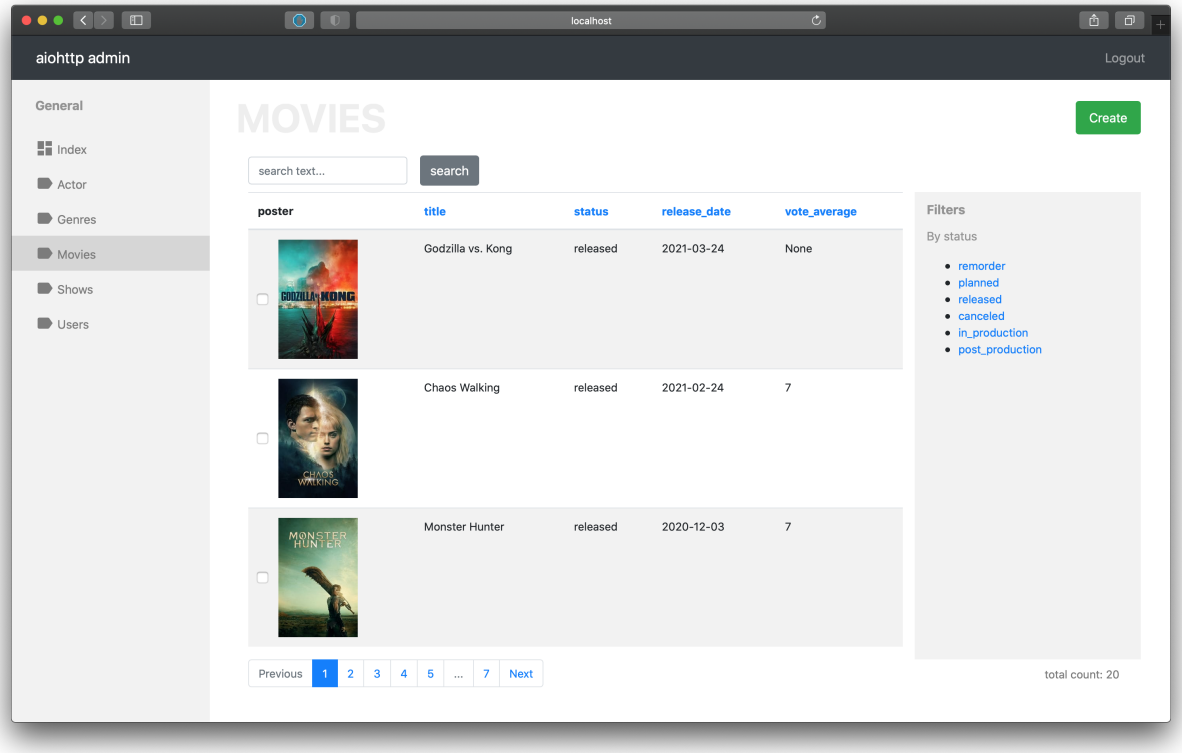
Mykhailo Havelia

Apr 05, 2024

CONTENTS:

Demo site | Demo source code.

The aiohttp admin is a library for build admin interface for applications based on the aiohttp. With this library you can ease to generate CRUD views for your data (for data storages which support by aiohttp admin) and flexibly customize representation and access to these.



CHAPTER

ONE

INSTALLATION

The first step which you need to do it's installing library

```
pip install aiohttp_admin2
```

If you need more detail information about installation look at *Installation* section.

CHAPTER

TWO

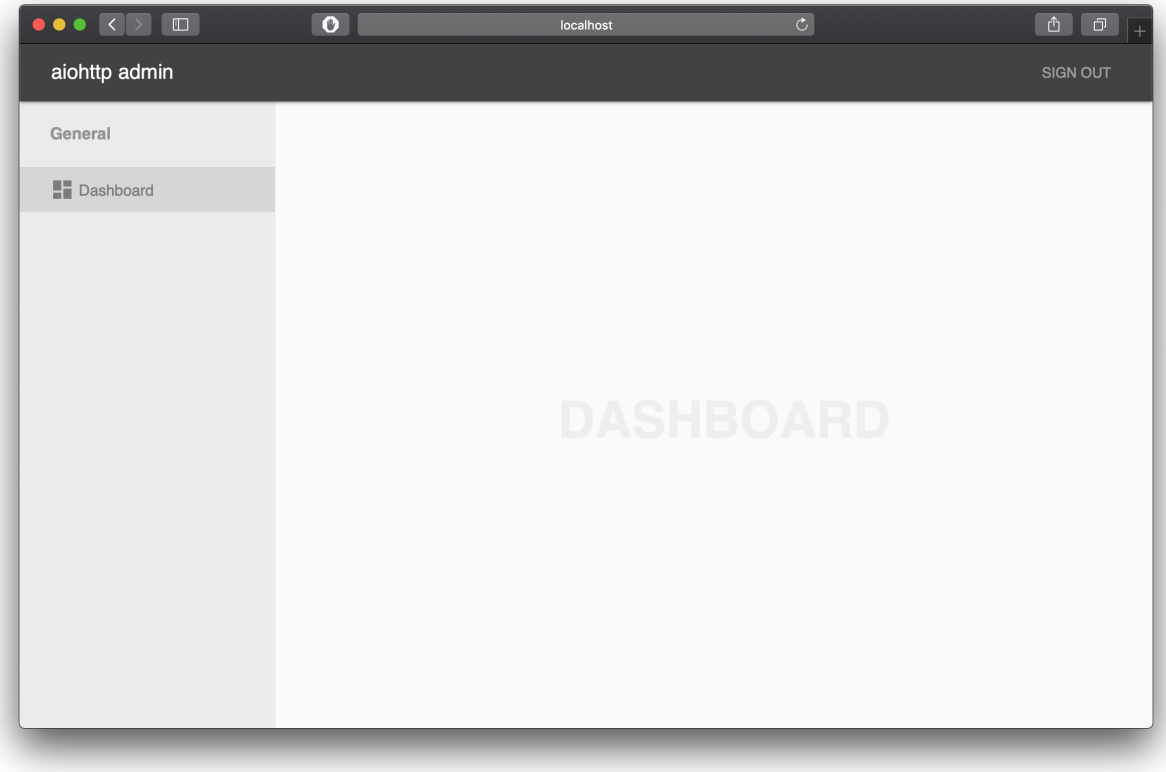
QUICK START

For simple start you need just import setup admin function and extend your existing aiohttp application. For example:

admin.py

```
from aiohttp import web
from aiohttp_admin2 import setup_admin

app = web.Application()

# setup admin interface
setup_admin(app)

web.run_app(app)
```

And run *python admin.py*. That is it. Now you can open in your browser *http://localhost:8080/admin/* and see home page of the our awesome admin interface.



Dashboard

The first page which you see when setup admin is dashboard. This is startup page and you can to customize it. For that u need to create your custom dashboard's class

```
from aiohttp import web
from aiohttp_admin2.views import DashboardView


class CustomDashboard(DashboardView):
    async def get_context(self, req):
        return {
            **await super().get_context(req=req),
            "content": "My custom content"
        }
```

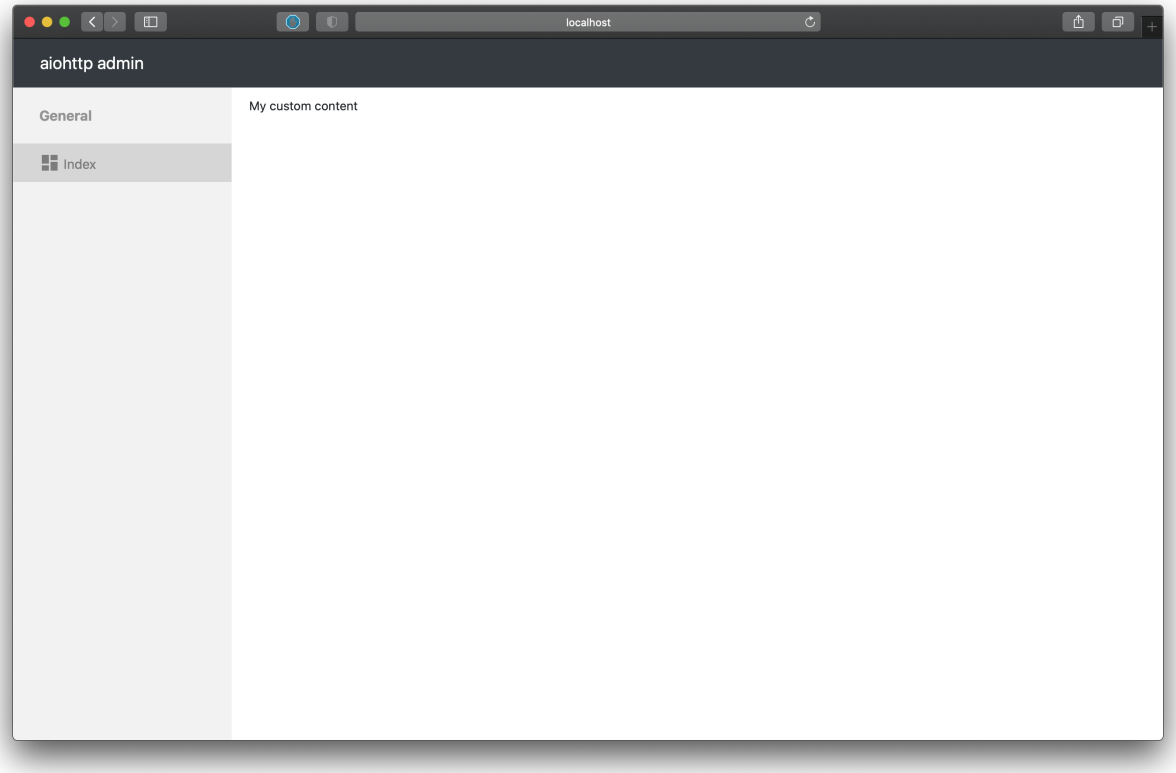
You can rewrite get_context method and put your new content to the jinja context. After that we need to create your custom admin class and put it into setup function:

```
from aiohttp import web
from aiohttp_admin2 import setup_admin
from aiohttp_admin2.views import Admin


class CustomAdmin(Admin):
    dashboard_class = CustomDashboard
```

(continues on next page)

(continued from previous page)

```
app = web.Application()

# setup admin interface
setup_admin(app, admin_class=CustomAdmin)

web.run_app(app)
```



As the alternative way we can to redefine the template for the our dashboard. The first thing which we need to do is create a new dashboard template.

templates/my_custom_dashboard.html

```
{% extends 'aiohttp_admin/layouts/base.html' %}

{% block main %}
  <h1>Dashboard</h1>
  <b>{{ content }}...</b>
{% endblock main %}
```

we nested from the base html template and put to the main block our new content. The second step is declare this template in the *CustomDashboard*.

```
class CustomDashboard(DashboardView):
    # redefine `template_name` attribute to your own
    template_name = 'my_custom_dashboard.html'
```

(continues on next page)

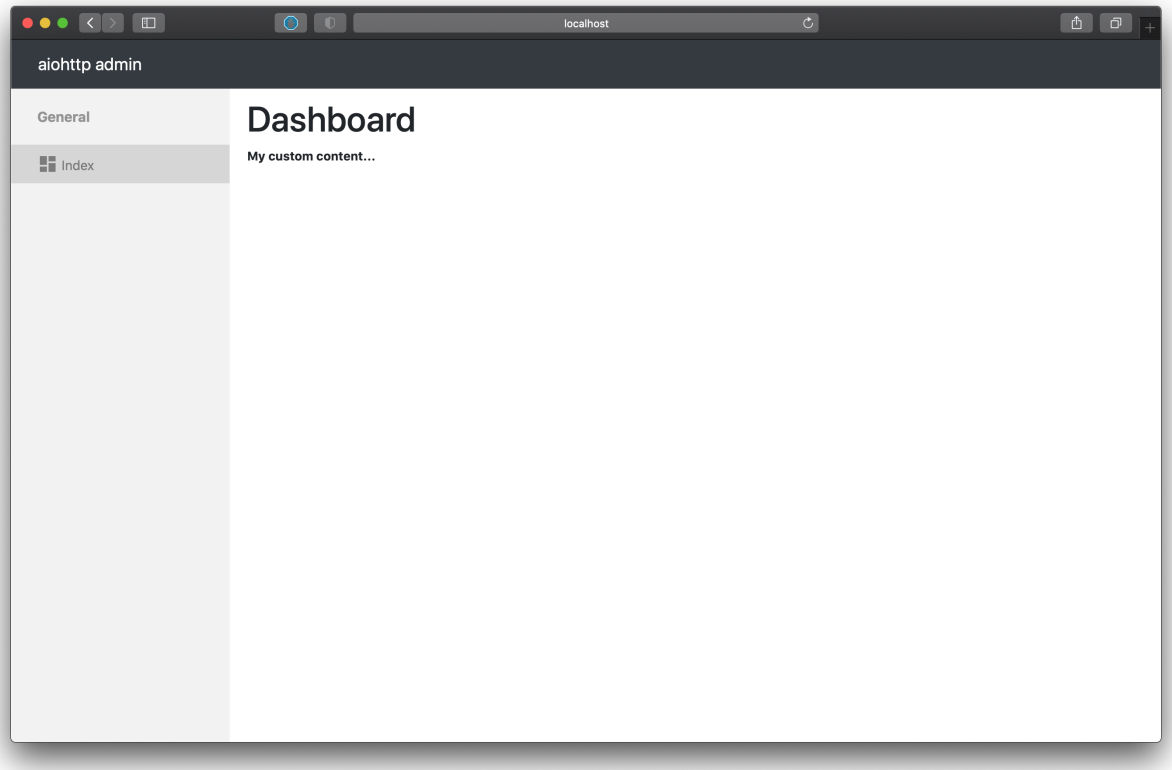
(continued from previous page)

```
    ...
```

The last step is setup jinja for your application and set path to the your templates directory

```
import aiohttp_jinja2
import jinja2
from pathlib import Path

# path to the your template directory
templates_directory = Path(__file__).parent / 'templates'

app = web.Application()

# setup jinja2
aiohttp_jinja2.setup(
    app=application,
    loader=jinja2.FileSystemLoader(str(templates_directory)),
)

# setup admin interface
setup_admin(app, admin_class=CustomAdmin)

web.run_app(app)
```

As result you can see that dashboard use your custom html.

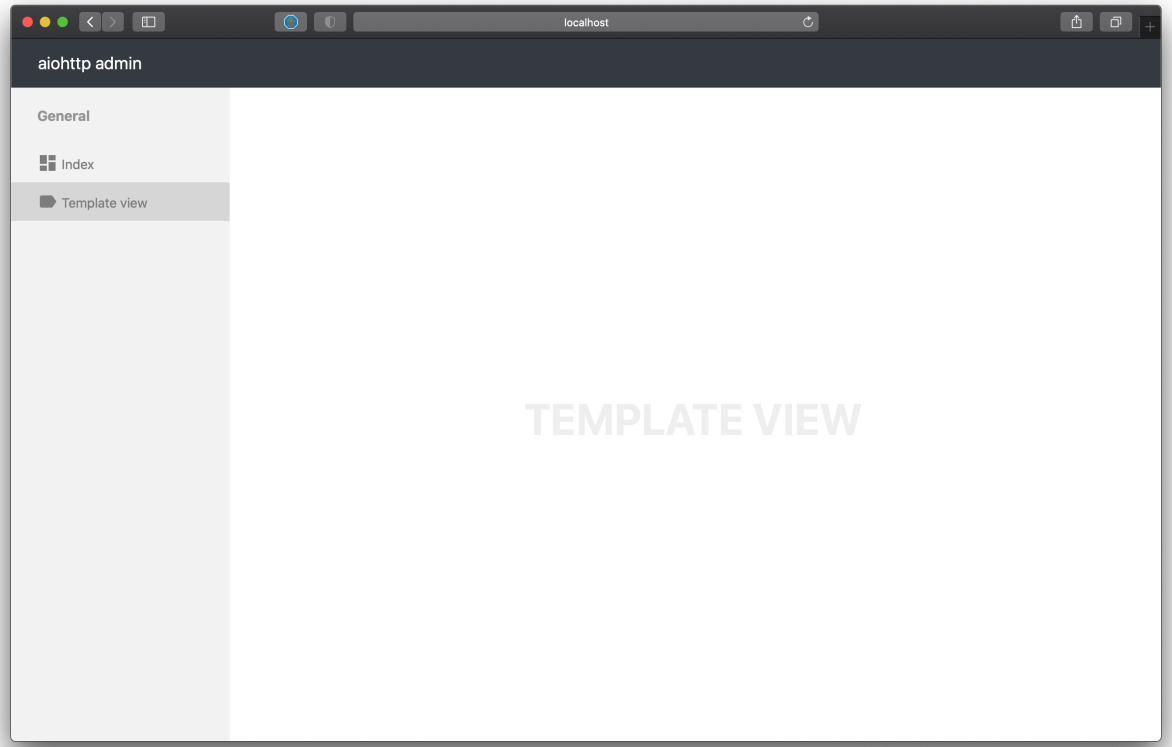


Custom views

The next thing which you can to do with your admin interface is create your own custom view. For that you need just create a new view and setup it together with admin interface. After that you can to see a new tab in the aside bar.

```
from aiohttp_admin2.views.aiohttp.views.template_view import TemplateView


class FirstCustomView(TemplateView):
    name = 'Template view'


# setup admin interface
setup_admin(
    application,
    admin_class=CustomAdmin,
    # put here your new template view to register it
    views=[FirstCustomView,]
)
```

The DashboardView class nested from the TemplateView class so you can do with it all things which we considered above for DashboardView class (redefine context and template).



If you don't want add current web page to the aside bar then you can specify *is_hide_view* attribute to *True*.

```
class FirstCustomView(TemplateView):
    name = 'Template view'
```

(continues on next page)

(continued from previous page)

```
    # remove link from aside bar
    is_hide_view = True
```

In this case you can to visit this web page got to the directly via url but admin interface will not to show any links to it.

CRUD views

The most helpful thing in the aiohttp_admin2 is possible to generate views based on models on your data from different databases.

Right now the library support models for:

- SQLAlchemy (postgres/mysql)
- umongo (mongodb)

So, if you have a above models then you can easy add create/delete/update views in the your admin interface. Let's consider a simple example how it might looks like for the SQLAlchemy.

Let's assume we have current SQLAlchemy's models:

```
from enum import Enum

import sqlalchemy as sa


metadata = sa.MetaData()


class PostStatusEnum(Enum):
    published = 'published'
    not_published = 'not published'


users = sa.Table('users', metadata,
    sa.Column('id', sa.Integer, primary_key=True),
    sa.Column('first_name', sa.String(255)),
    sa.Column('last_name', sa.String(255)),
    sa.Column('is_superuser', sa.Boolean),
    sa.Column('joined_at', sa.DateTime()),
)

books = sa.Table('post', metadata,
    sa.Column('id', sa.Integer, primary_key=True),
    sa.Column('title', sa.String(255)),
    sa.Column('body', sa.Text),
    sa.Column('status', sa.Enum(PostStatusEnum)),
    sa.Column('published_at', sa.DateTime()),
    sa.Column('author_id', sa.ForeignKey('users.id', ondelete='CASCADE')),
)
```

We have the simple users table and the posts table where allocated posts which have been created via our users. Users and posts tables related by foreignKey *author_id*.

The first thing which we need to do before start to generate CRUD views is set up the PostgreSQL connection in the our application

```
import aiopg.sa
from aiohttp_admin2.connection_injectors import ConnectionInjector


postgres_injector = ConnectionInjector()


async def init_db(app):
    engine = await aiopg.sa.create_engine(
        user='postgres',
        database='postgres',
        host='0.0.0.0',
        password='postgres',
    )
    app['db'] = engine

    # set our engine to the postgres_injector
    postgres_injector.init(engine)

    yield

    app['db'].close()
    await app['db'].wait_closed()

application = web.Application()
application.cleanup_ctx.extend([init_db])
```

It's a standard way to set up connection to the database into aiohttp but we have new lines related with *ConnectionInjector*. *ConnectionInjector* is just a class which used to share database connection with between aiohttp and admin library.

The second thing is create registered our model for the admin.

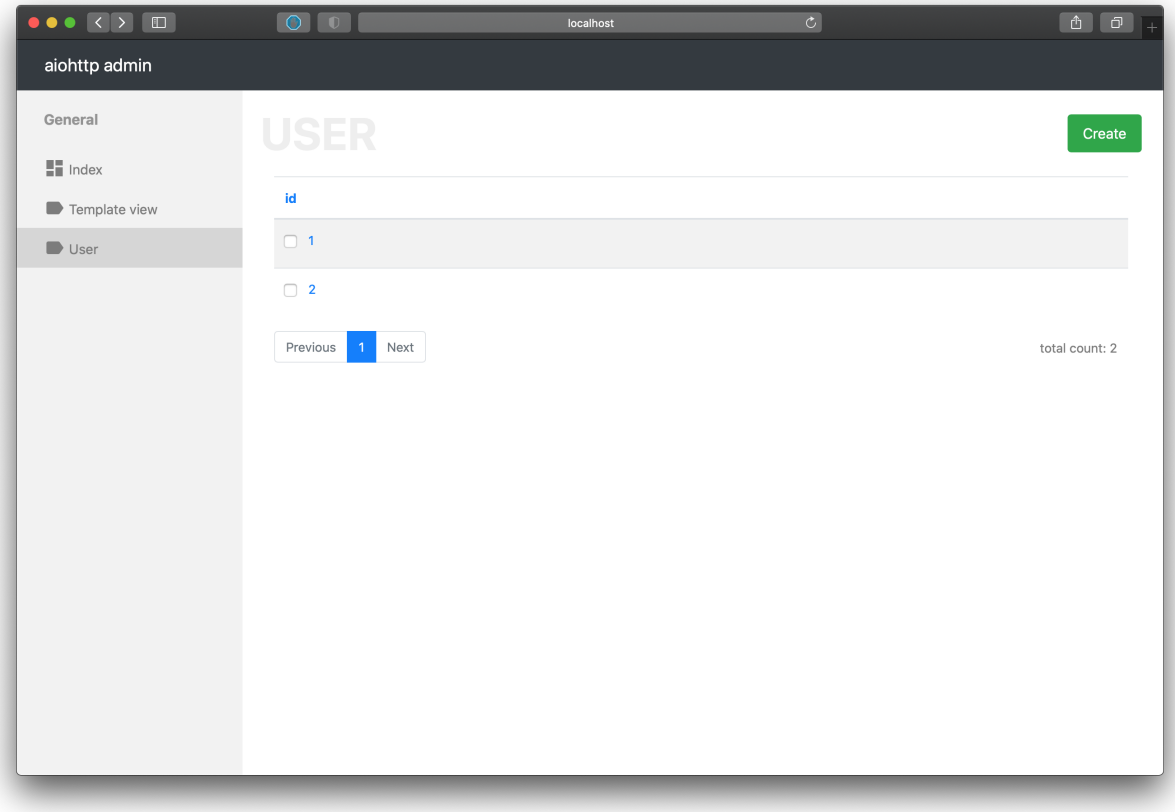
```
from aiohttp_admin2.views import ControllerView
from aiohttp_admin2.controllers.postgres_controller import PostgresController
from aiohttp_admin2.mappers.generics import PostgresMapperGeneric


# create a mapper for table
class UserMapper(PostgresMapperGeneric, table=users):
    pass


# create controller for table with UserMapper
@postgres_injector.inject
class UserController(PostgresController, table=users):
    mapper = UserMapper
    name = 'user'


# create view for table
class UserView(ControllerView):
    controller = UserController
```

And after that setup our admin interface with the *UserView*.

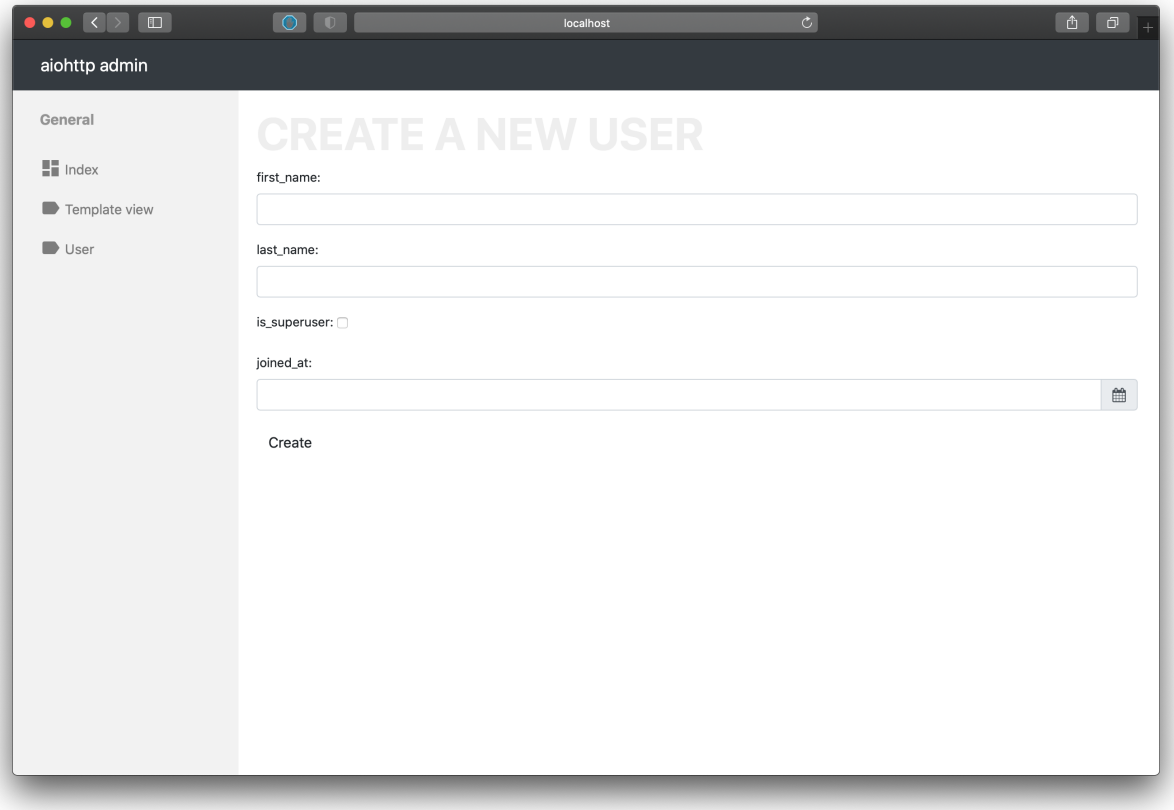
```
# setup admin interface
setup_admin(
    application,
    admin_class=CustomAdmin,
    views=[
        FirstCustomView,
        UserView, # added new view
    ]
)
```

Now, you can to see that admin interface has new tab in the aside bar and we have simple list, create, update and delete pages for the our user model.

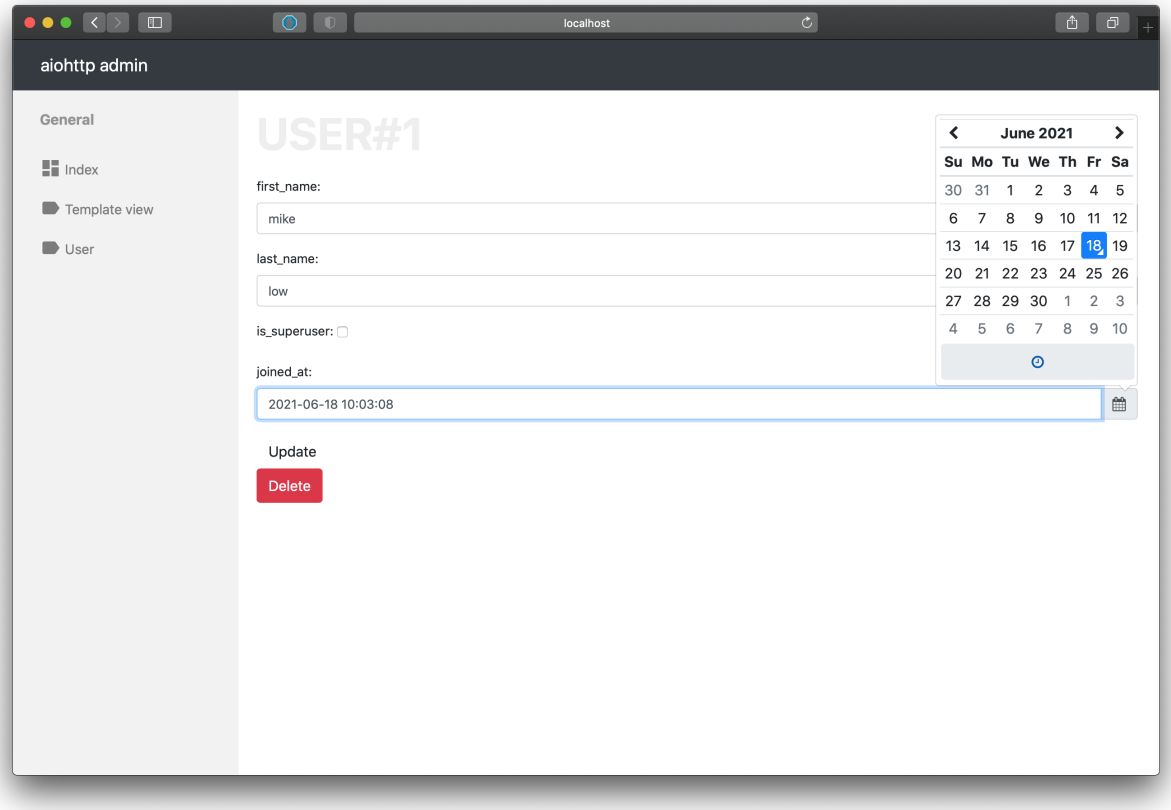
List page



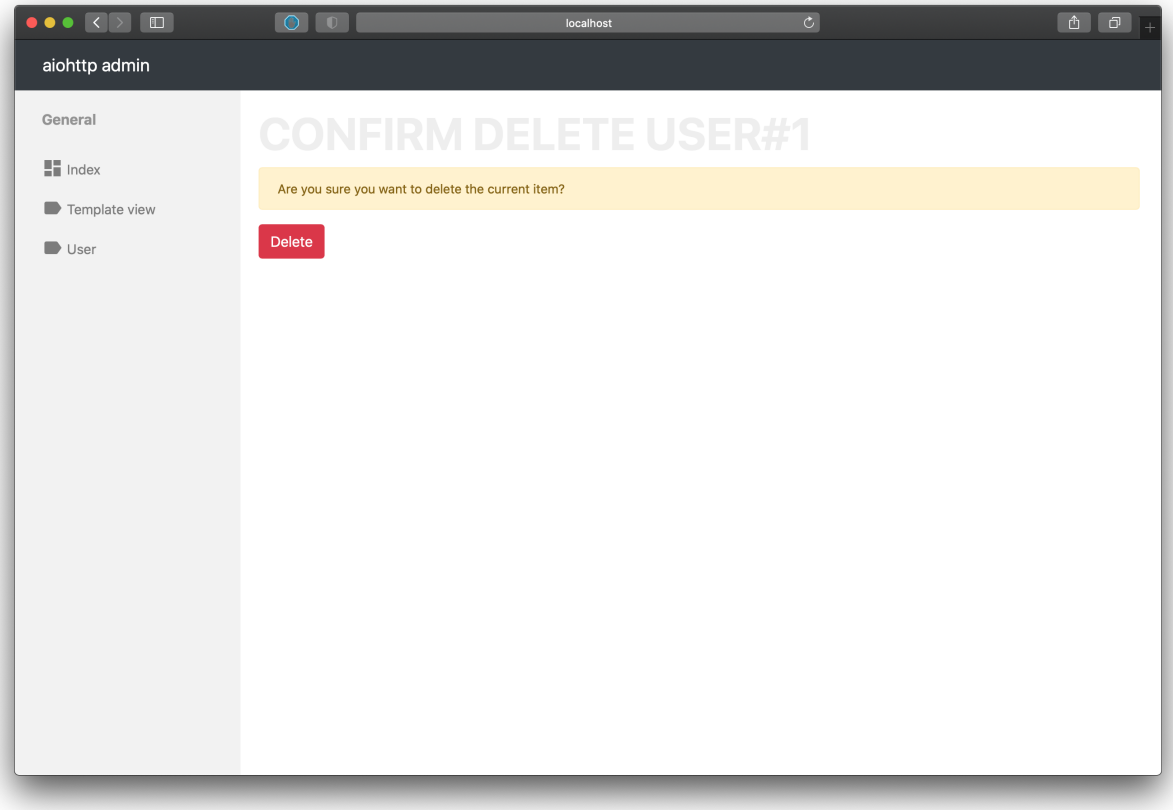
Create page


localhost
aiohttp admin
General
Index
Template view
User
CREATE A NEW USER
first_name:
last_name:
is_superuser:
joined_at:
Create

Update page


localhost
aiohttp admin
General
Index
Template view
User
USER#1
first_name:
mike
last_name:
low
is_superuser:
joined_at:
2021-06-18 10:03:08
Update
Delete
June 2021
Su Mo Tu We Th Fr Sa

Delete page



Let's make our list page view a little bit better. For that we can to show to user more information using *inline_fields* attribute in the *UserController* class.

```
@postgres_injector.inject
class UserController(PostgresController, table=users):
    mapper = UserMapper
    name = 'user'

    inline_fields = ['id', 'full_name', 'is_superuser', 'joined_at']
```

Our table has *id*, *is_superuser* and *joined_at* but don't has *full_name* but for end user we want to show full name. For do that we can to use custom filed and add to our class the *full_name_field* method (<field_name>_field).

```
@postgres_injector.inject
class UserController(PostgresController):

    ...

    async def full_name_field(self, obj):
        return f'{obj.data.first_name} {obj.data.last_name}'
```

Also we want to give a possible to user use search by *fist_name* and *last_name* fields. We can easy do that by add *search_fields* attribute and specify list of fields which we want to use to search.

```
@postgres_injector.inject
class UserController(PostgresController):
```

(continues on next page)

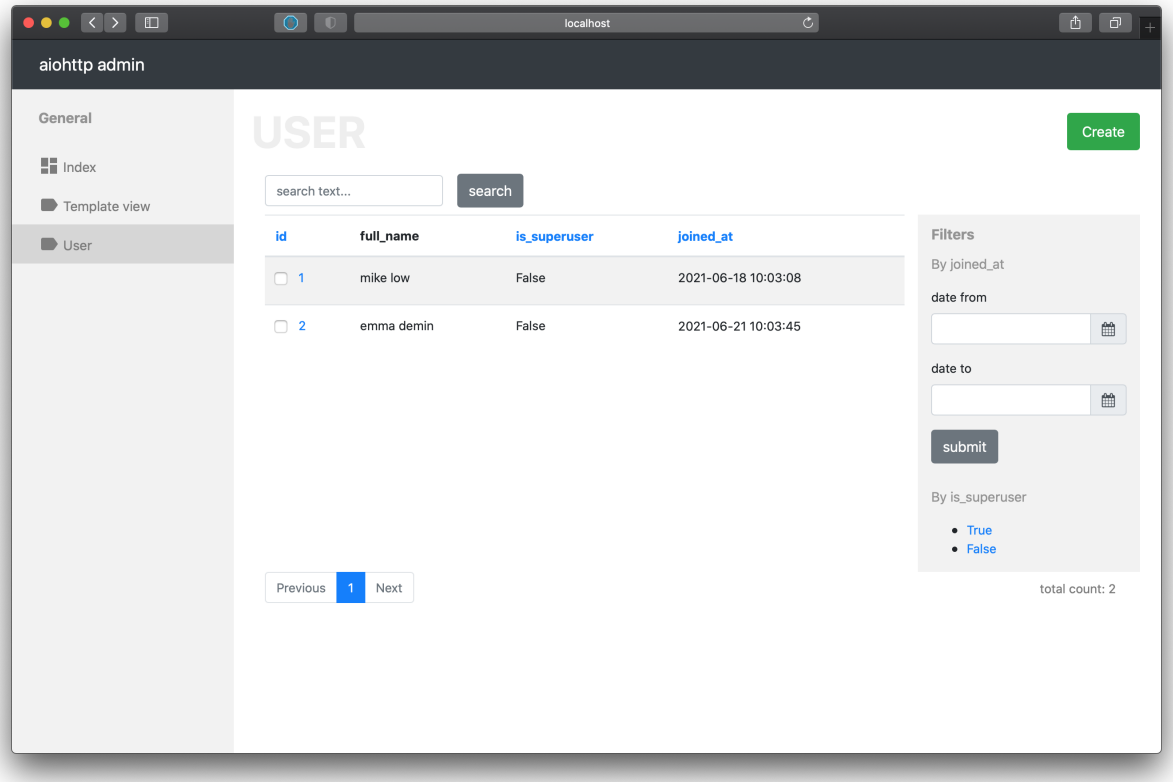
(continued from previous page)

```
    ...

    search_fields = ['first_name', 'last_name']
```

And as final step we want to give a possible to filter a list of our data. For achieve this goal we need only specify list of fields which will use for filtering to the *list_filter* attribute.

```
@postgres_injector.inject
class UserController(PostgresController):

    ...

    list_filter = ['joined_at', 'is_superuser', ]
```

After that view of our list page become much better



Let's make the same for the post model

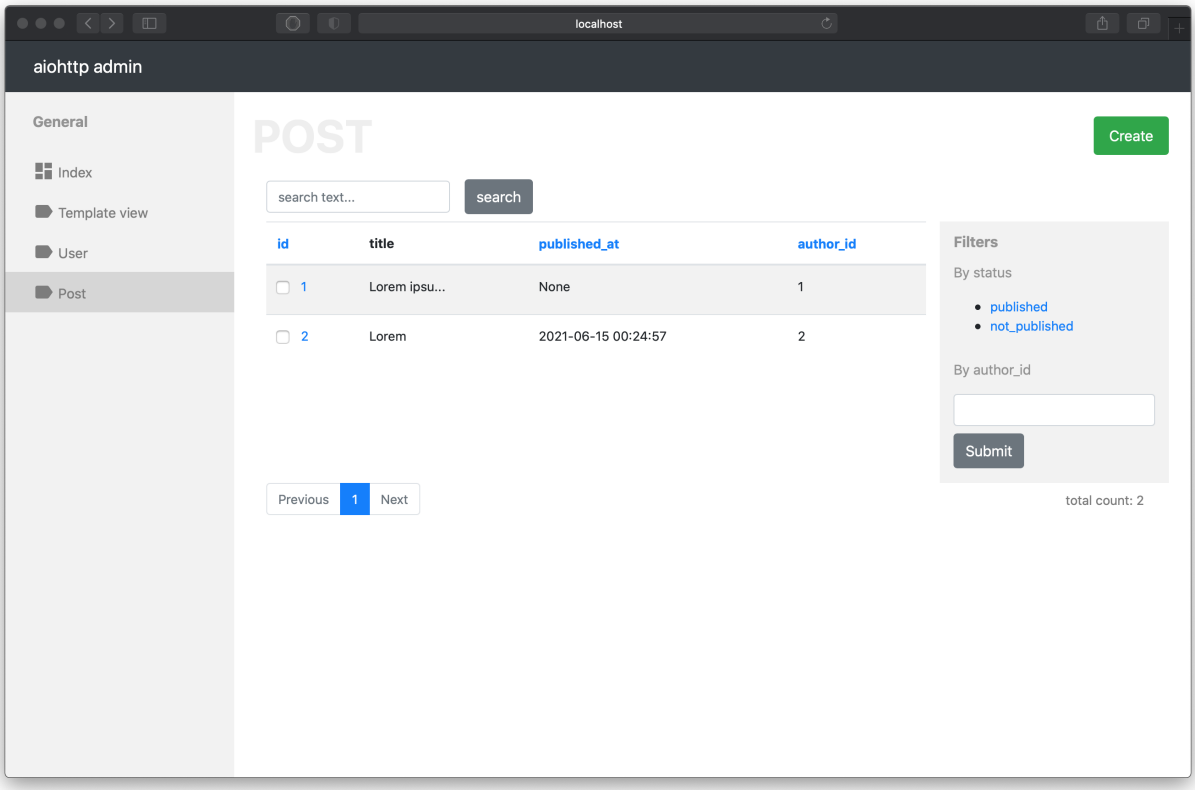
```
# create controller for table with UserMapper
@postgres_injector.inject
class PostController(PostgresController, table=post):
    mapper = PostMapper
    name = 'post'

    inline_fields = ['id', 'title', 'published_at', 'author_id', ]
```

(continues on next page)

(continued from previous page)

```
    search_fields = ['title', ]
    list_filter = ['status', 'author_id', ]

    async def title_field(self, obj):
        if len(obj.data.title) > 10:
            return f'{obj.data.title[:10]}...'
        return obj.data.title


# create view for table
class PostView(ControllerView):
    controller = PostController
```

The post table has *title* field but we want to change view of this field. In this cases we can also to use the custom field (*title_field*).



Okay. We has a representation of the post and the user models. This models have relations between each other and we need to show it in the admin interface.

First relation is one to one relation between post and author. Single post has only one author. To show this relation we can to specify *relations_to_one* attribute.

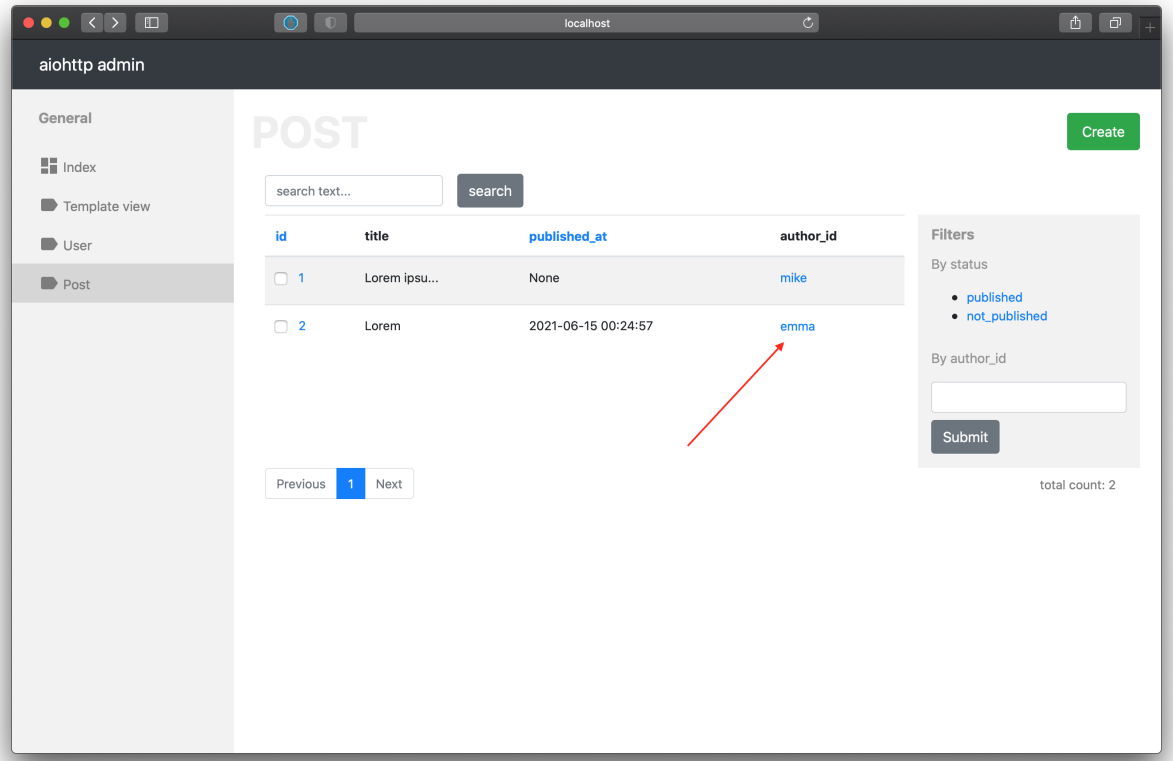
```
from aiohttp_admin2.controllers.relations import ToOneRelation


@postgres_injector.inject
```

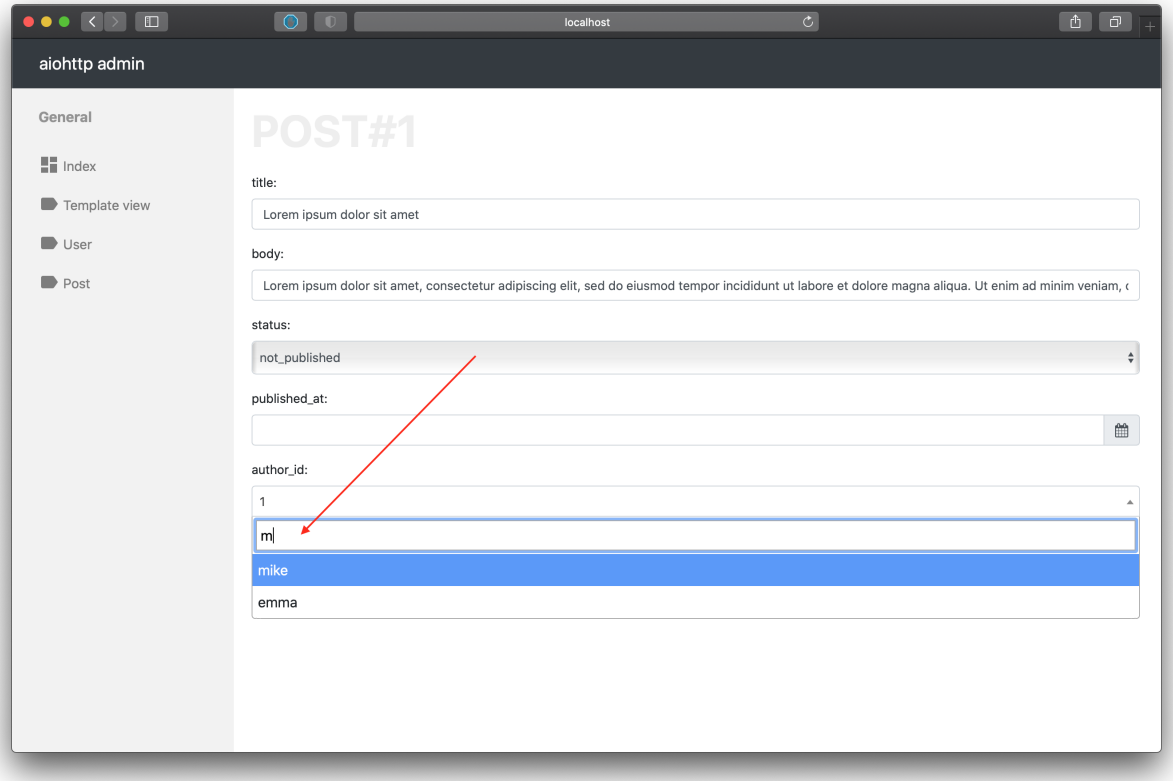
(continues on next page)

(continued from previous page)

```
class PostController(PostgresController):

    ...

    relations_to_one = [
        ToOneRelation(
            # the field name of current relation
            name='author_id',
            # the name of field which responsible for relation (foreignkey)
            field_name='author_id',
            # controller of the relation model
            controller=UserController
        ),
    ]


@postgres_injector.inject
class UserController(PostgresController):

    ...

    async def get_object_name(self, obj):
        # need just for better representation instances of current model
        return obj.data.first_name
```

In *ToOneRelation* we put *name* of field which will represented current relation (in our case we replace existing *author_id* field) and *field_name* field in table which responsible for current relation and controller of the relation model. After that we has link to the relational model on list page and autocomplete on create/update page.

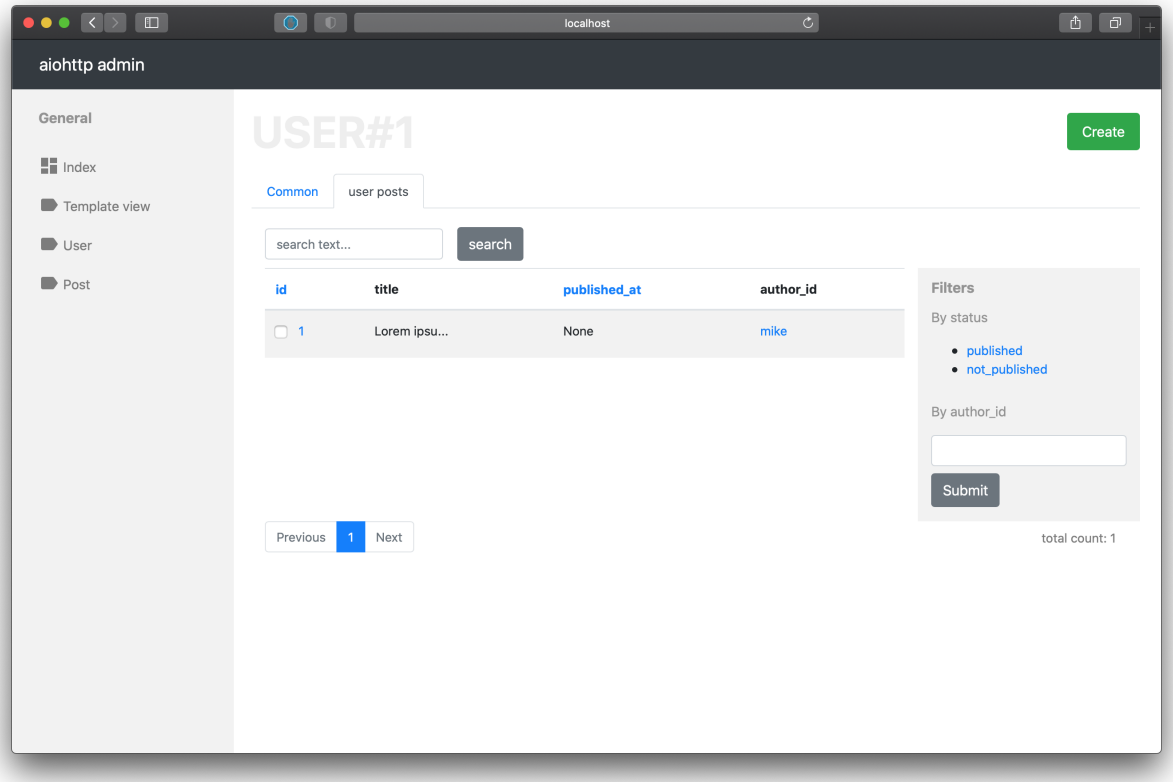

localhost
aiohttp admin
General
Index
Template view
User
Post
POST
Create
search text...
search
id
title
published_at
author_id
1
Lorem ipsu...
None
mike
2
Lorem
2021-06-15 00:24:57
emma
Filters
By status
published
not_published
By author_id
Submit
Previous
1
Next
total count: 2


localhost
aiohttp admin
General
Index
Template view
User
Post
POST#1
title:
Lorem ipsum dolor sit amet
body:
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
status:
not_published
published_at:
author_id:
1
m
mike
emma

The second relation is one to many relation between auth and posts. User can has many posts. To show this relation we can to specify *relations_to_many* attribute.

```
from aiohttp_admin2.controllers.relations import ToManyRelation

@postgres_injector.inject
class UserController(PostgresController):

    ...

    relations_to_many = [
        ToManyRelation(
            # the name of current relation
            name='user posts',
            # the name of field which responsible for relation in the
            # current table
            left_table_pk='id',
            # controller of the relation model (we can use controller
            # class or callable function which return it).
            relation_controller=lambda: PostController,
        )
    ]
```

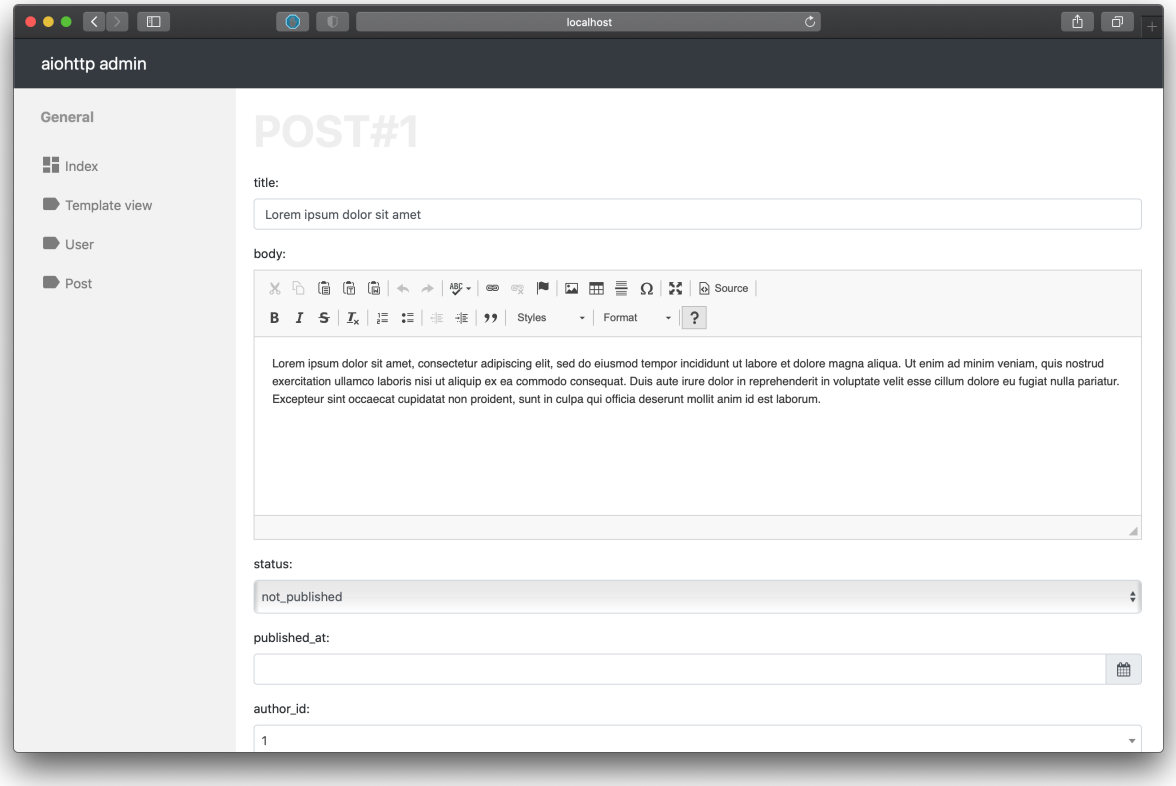
In *ToManyRelation* we put *name* which will use as title of the current relation, left_table_pk which describe field which responsible for relation between tables and *relation_controller* which receive the controller class of related model. After that we'll have a tab bar on detail page of author model. On this tab we see all post of the current user.



We are continuing improve admin interface and next step is customize detail page of the post model. The first thing which we can to improve its add html editor for the body field. Widgets responsible for view of input on detail page and we can change these for some particular type of field or for concretical field.

```
from aiohttp_admin2.views import widgets


# create view for table
class PostView(ControllerView):
    controller = PostController

    fields_widgets = {'body': widgets.CKEditorWidget}
```

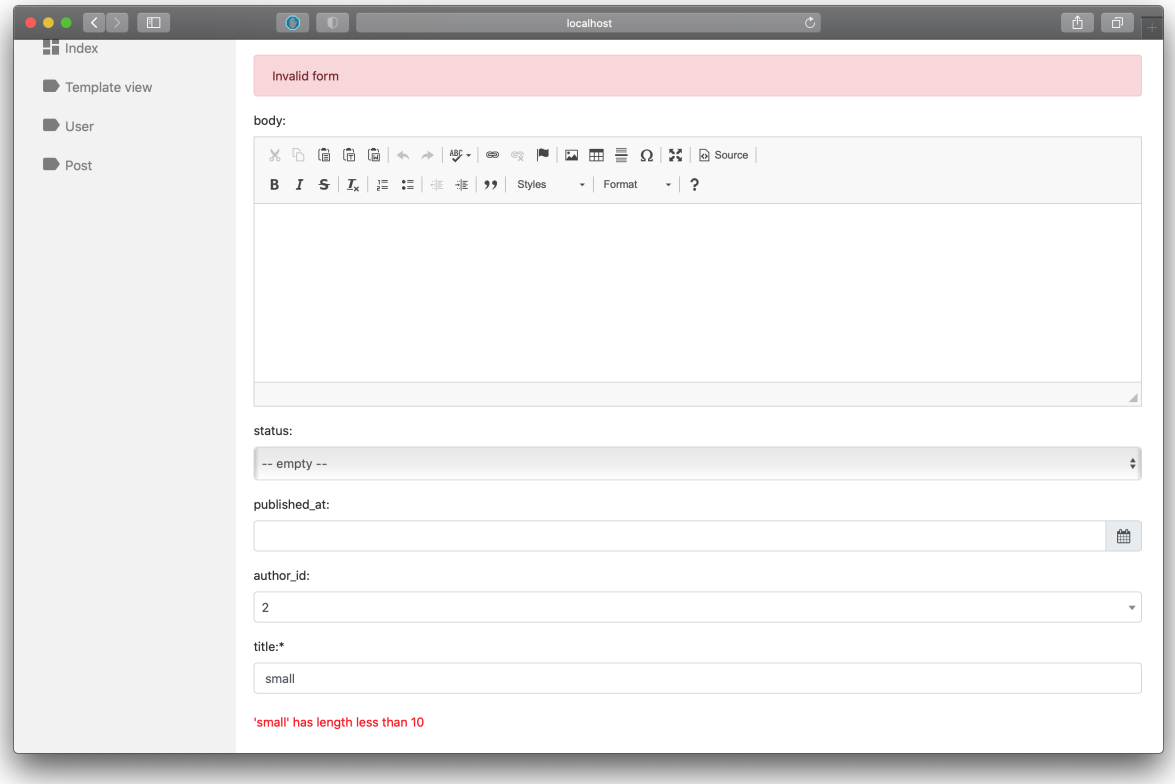
Two lines of code and we have a html editor for body field.



Also let's assume that we need to add some validation on create/update post in the admin interface. We can to do that via mappings. Mapping responsible for validation of data in aiohttp admin.

```
class PostMapper(PostgresMapperGeneric, table=post):
    title = fields.StringField(
        required=True,
        validators=[length(min_value=10)],
    )
```

The *PostgresMapperGeneric* mapper generate all fields of table but if we need specify for these fields some properties we need to redefine these. We just redefined title field to make it required and add validator to avoide cases when title will less then 10 symbol. Now, if we try to create a post with small title that we'll see an error message.



Authorization

The last thing which we need to add to complete our admin interface is authorization. For that we'll use *aiohttp_security* and *aiohttp_session* libs. As a first step let's setup *aiohttp_security*.

```
import base64
from aiohttp_security import AbstractAuthorizationPolicy
from aiohttp_security import SessionIdentityPolicy
from aiohttp_security import setup as setup_security
from aiohttp_session.cookie_storage import EncryptedCookieStorage
from aiohttp_session import setup as session_setup
from cryptography import fernet


class AuthorizationPolicy(AbstractAuthorizationPolicy):
    async def permits(self, identity, permission, context=None) -> bool:
        if identity == 'admin' and permission == 'admin':
            return True

        return False

    async def authorized_userid(self, identity) -> int:
        return identity
```

(continues on next page)

(continued from previous page)

```
async def security(application: web.Application) -> None:
    fernet_key = fernet.Fernet.generate_key()
    secret_key = base64.urlsafe_b64decode(fernet_key)

    session_setup(
        application,
        EncryptedCookieStorage(secret_key, cookie_name='API_SESSION'),
    )

    policy = SessionIdentityPolicy()
    setup_security(application, policy, AuthorizationPolicy())

    yield


application = web.Application()
application.cleanup_ctx.extend([init_db, security])
```

After that let's create a login/logut pages.

login.html

```
<form
    method="POST"
    action="{{ url('login_post') }}"
>
    <label for="username">Username</label>
    <input type="text" name="username" id="username" value="admin">
    <label for="password">Password</label>
    <input type="password" name="password" id="password" value="admin">
    <button type="submit">Submit</button>
</form>
```

```
import aiohttp_jinja2
from aiohttp import web
from aiohttp_security import is_anonymous
from aiohttp_security import permits
from aiohttp_security import remember
from aiohttp_security import forget


@aiohttp_jinja2.template('login.html')
async def login_page(request: web.Request) -> None:
    if not await is_anonymous(request):
        raise web.HTTPFound('/admin/')


@aiohttp_jinja2.template('login.html')
async def login_post(request: web.Request) -> None:
    data = await request.post()

    if data['username'] == 'admin' and 'admin' == data['password']:
```

(continues on next page)

(continued from previous page)

```
        admin_page = web.HTTPFound('/admin/')
        await remember(request, admin_page, 'admin')
        raise admin_page

    raise web.HTTPFound('/login')


async def logout_page(request: web.Request) -> None:
    redirect_response = web.HTTPFound('/login')
    await forget(request, redirect_response)
    raise redirect_response


# added these handlers to the web application

application.add_routes([
    web.get('/login', login_page, name='login'),
    web.post('/login', login_post, name='login_post'),
    web.get('/logout', logout_page, name='logout')
])
```

All these steps are not related with aiohttp admin and can be different in other project so we avoid to explain these (this is naive implementation of authorization, please not use it in production).

As the last step we need to implement method for admin interface which will detect user with access to admin interface. For these purposes we'll use middleware

```
import aiohttp_jinja2
from aiohttp import web
from aiohttp_security import is_anonymous
from aiohttp_security import permits


@web.middleware
async def admin_access_middleware(request, handler):
    if await is_anonymous(request):
        raise web.HTTPFound('/')

    if not await permits(request, 'admin'):
        raise web.HTTPFound('/')

    return await handler(request)

# add current middleware to the admin setup

setup_admin(
    application,
    admin_class=CustomAdmin,
    views=[FirstCustomView, UserView, PostView],
     # add middleware for admin
    middleware_list=[admin_access_middleware, ],
    # set logout path
    logout_path='/logout'
```

(continues on next page)

(continued from previous page)

```
)
```

After that unauthorized users will not access to admin interface. If you need to give access only for particular model, you can use *access_hook* method in view class (read detail in the docs).

The source code of current examples you might to find here.

2.1 Installation

2.1.1 Stable release

To install aiohttp admin 2, run this command in your terminal:

```
$ pip install aiohttp_admin2
```

This is the preferred method to install aiohttp admin 2, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.1.2 From sources

The sources for aiohttp admin 2 can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/arfey/aiohttp_admin2
```

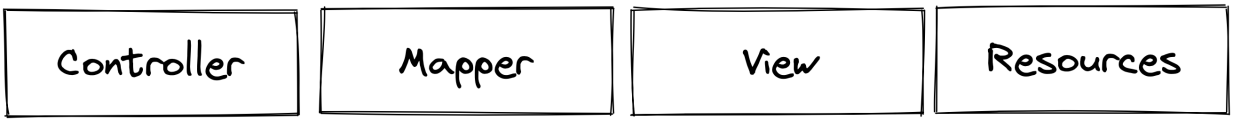
Or download the tarball:

```
$ curl -OJL https://github.com/arfey/aiohttp_admin2/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

2.2 Usage aiohttp admin

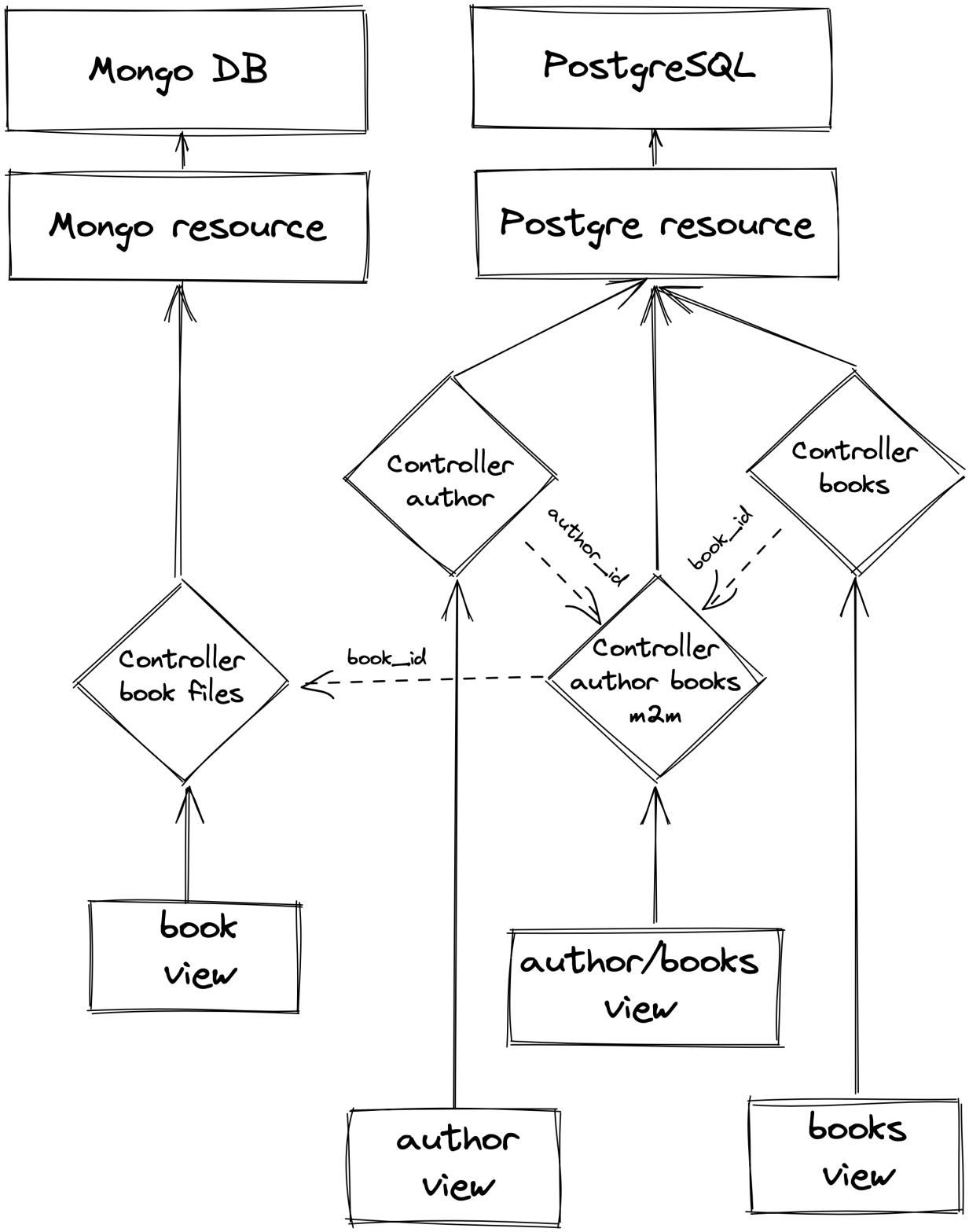


For the beginning let's make a simple overview of architecture and main components. The *aiohttp_admin* has small list of main components which responsible for different function of admin interface:

- **resource** - this component implement all method (get/delete/update etc) that need to communicate with databases. So, if you want to add support of database which is not exist right now on *aiohttp_admin* then can just create your resource object for that and all other components of admin interface will work with it togather without any problems

- **controller** - in this component allocate business logic related with some model. What are fields we need to show on list view? How many items do we need to show on each list page? What is order we need to use by default? What do we need to do before/after create/update instance? How do models related? All these question about controller. Controller use *resource* to get access to database and *mapper* for validate input data from user.
- **mapper** - this component responsible for validation and convert input data from user which will use for update or create instance.
- **view** - this component responsible for represent result for user via some async web framework (now we use aiohttp but you can to implement views for other web framework and use other all components of aiohttp admin without any problems).

Let's consider a simple example of books library. We have table of authors and books. Each book has one or more authors. Each author has one or more books. Relation between author and books is many to many and implement via a separate table. All these tables stores in the PostgreSQL but large book's files allocated in the MongoDB.



We can to see that relation between models implement on controller level and we can to bind models from different storages together.

2.2.1 Authorization & Permissions

Authorization

If you need an authorization to the your admin interface then you can add custom middleware to achieve this goal. It might look like this:

```
from aiohttp import web
from aiohttp_security import is_anonymous
from aiohttp_security import permits
from aiohttp_admin2 import setup_admin


@web.middleware
async def admin_access_middleware(request, handler):
    if await is_anonymous(request):
        raise web.HTTPFound('/')

    if not await permits(request, 'admin'):
        raise web.HTTPFound('/')

    return await handler(request)


setup_admin(
    application,
    # You can specify here a list of middlewares which you want to apply
    # to each request related with admin interface
    middleware_list=[admin_access_middleware, ],
    ...
)
```

In the snippet above we just check that user is not anonymous and has admin permissions. The logic of *is_anonymous* and *permits* you have to implement by yourself because admin just guarantee the for each admin route will apply list of middlewares which you specify in the **middleware_list** param.

The *aiohhtp_admin2* don't provide any views for login/logout logic so all of this logic you need implement by yourself.

Permissions

For organization permissions in your admin interface you have to use the *access_hook* method in the view class. Since the *aiohhtp_admin2* instantiate new view for each request and after that run *access_hook* method therefore inside this method you can easy change any propery of the current view instance to restrict access.

As an example you have a ActorView class and you want to show information related with this view only for users who have correct rights for that.

```
from aiohttp_admin2.view import ControllerView


class ActorView(ControllerView):
    controller = ActorController

    async def access_hook(self) -> None:
```

(continues on next page)

(continued from previous page)

```
        if not user_can_view(self.request, 'ators'):
            self.has_access = False
```

In *access_hook* method we can to get current request so we just pass it to the predicate function (*user_can_view*) and change property if need. If user without right access visit any route related with current view then he gets the *PermissionDenied* exception. If you want only hide view from aside menu than you have to use *is_hide_view* property instead.

Let's consider case when you need to give only read right or give right to create but without edit rights.

```
from aiohttp_admin2.view import ControllerView


class ActorView(ControllerView):
    controller = ActorController

    async def access_hook(self) -> None:
        # here we get controller instance of the current view
        controller = self.get_controller()

        controller.can_view = user_can_view(self.request, 'ators')
        controller.can_edit = user_can_edit(self.request, 'ators')
        controller.can_delete = user_can_delete(self.request, 'ators')
        controller.can_create = user_can_create(self.request, 'ators')

        if is_guest(self.request):
          controller.inline_fields = ['id', ]
          self.template_detail_name = 'aiohttp_admin/detail_view_for_guest.html'
          controller.per_page = 20
```

We can change any property of controller even *inline_fields* or *per_page* if we need to do that.

Warning: The *access_hook* method is async function so you actually can to do request to databases inside it to check permission but it's not a good idea because for each request the admin call this method for each view (to check that we can show link to views in aside menu) and that can produce n + 1 requests. The better approach is get all rights inside *middelware* and set this info to request and inside *access_hook* method just check that request contain right access.

2.2.2 Mappers

Mapper is schema for validation and converting data which income from user and use for create or update instances. You can create mapper in two ways.

Custom mappers

You can create your own mapper with custom fields:

```
from aiohttp_admin2.mappers import Mapper
from aiohttp_admin2.mappers import fields


class UserMapper(Mapper):
    """Mapper for user instance."""
    name = fields.StringField(required=True)
    age =  fields.IntField(default=18)
```

Mappers generator

If you create admin page for SQLalchemy or Umongo instances then you can generate mapping automatically by specifying models.

```
from aiohttp_admin2.mappers.generics import PostgresMapperGeneric
from aiohttp_admin2.mappers import fields


user = sa.Table('user', metadata,
    sa.Column('name', sa.String(255)),
    sa.Column('age', sa.Integer),
)


class UserMapper(PostgresMapperGeneric, table=user):
    """Mapper for user instance."""
    pass
```

but if you want to rewrite some field you can do it some like that

```
from aiohttp_admin2.mappers.generics import PostgresMapperGeneric
from aiohttp_admin2.mappers import fields


class UserMapper(PostgresMapperGeneric, table=user):
    """Mapper for user instance."""
    age = fields.StringField(required=True)
```

In this case generic will generate all fields for you but will use age field which you specify.

Fields

StringField, LongStringField, UrlImageField, UrlFileField, UrlField - field for represented string data.

- *required* - add validation for empty value if set to *True*
- *default* - replace empty value if specify
- *validators* - list of validators
- *primary_key* - *True* if current field is a primary key

IntField, SmallIntField - field for represented integer data.

- *required* - add validation for empty value if set to *True*
- *default* - replace empty value if specify
- *validators* - list of validators
- *primary_key* - *True* if current field is a primary key

FloatField - field for represented float data.

- *required* - add validation for empty value if set to *True*
- *default* - replace empty value if specify
- *validators* - list of validators
- *primary_key* - *True* if current field is a primary key

DateTimeField, DateField - field for represented datetime data.

- *required* - add validation for empty value if set to *True*
- *default* - replace empty value if specify (you can specify str or datetime/date object)
- *validators* - list of validators
- *primary_key* - *True* if current field is a primary key

BooleanField - field for represented boolean data. If value contains '0', 'false' or 'f' than value will be parse as *False* in other case as *True*.

- *required* - add validation for empty value if set to *True*
- *default* - replace empty value if specify
- *validators* - list of validators
- *primary_key* - *True* if current field is a primary key

ChoicesField - add predefined values. If you have some finite list of values and want that this list will represented like select tag you need to use current field type.

- *required* - add validation for empty value if set to *True*
- *default* - replace empty value if specify
- *validators* - list of validators
- *field_cls* - field type which will represent selected value
- *choices* - tuple of tuple with values. It might look like this *[('admin title of value1', 'value1'), ('admin title of value1', 'value2')])*
- *primary_key* - *True* if current field is a primary key
- *empty_value* - need to to specify string which will show if a value is not set. By default it's – *empty* –.

ArrayField - field for represented array data. Instances inside array must to have the same type. To specify this type you have to provide *field_cls*

- *required* - add validation for empty value if set to *True*
- *default* - replace empty value if specify
- *validators* - list of validators
- *field_cls* - field type which will represent data type of items inside array
- *primary_key* - *True* if current field is a primary key
- *min_length, max_length* - add validation related with min/max length of array

JsonField - field for represented data in json type format.

- *required* - add validation for empty value if set to *True*
- *default* - replace empty value if specify
- *validators* - list of validators
- *primary_key* - *True* if current field is a primary key

```
from aiohttp_admin2.mappers.generics import PostgresMapperGeneric
from aiohttp_admin2.mappers import fields


class UserMapper(PostgresMapperGeneric, table=user):
    """Mapper for user instance."""
    GENDER_CHOICES = (
        ('male', "male"),
        ('female', "female"),
    )

    gender = fields.ChoicesField(
        field_cls=fields.StringField,
        choices=GENDER_CHOICES,
        default='male'
    )
```

In common you do not use mappers you need to create these only for internal usage for aiohttp admin but for a better understanding of why they are needed, let's take a look at how they are used.

```
from aiohttp_admin2.mappers import Mapper
from aiohttp_admin2.mappers import fields


class UserMapper(Mapper):
    """Mapper for user instance."""
    name = fields.StringField(required=True)
    age =  fields.IntField(default=18)
```

Let's try to validate wrong data

```
user_data = UserMapper({"age": '38'})

# return False because name is required
user_data.is_valid()
```

Now, try to check corrected data

```
user_data = UserMapper({"age": '38', "name": "mike"})

# return True because all is fine
user_data.is_valid()

print(user_data.data)
# {'name': 'mike', 'age': 38}
```

user_data.data return converting data in right type. We can see that string '38' have been successful converting to int value 38.

Note: The primary key is required fields for any models when we wanna update instance but when we need to create instance we don't know it (when a storage autoincrement it). For these purposes fields have *primary_key* property. If this property set to True and we try to create instance then mapper will ignore *required* errors related with current field. For that we need just specify *skip_primary* to *True* into *is_valid* method.

```
from aiohttp_admin2.mappers import Mapper
from aiohttp_admin2.mappers import fields


class UserMapper(Mapper):
    """Mapper for user instance."""
    id =  fields.IntField(primary_key=True, required=True)
    name = fields.StringField(required=True)

# False
UserMapper({"name": "Mike", "id": None}).is_valid()

# True
UserMapper({"name": "Mike", "id": None}).is_valid(skip_primary=True)
```

So when you don't use generators for your models or rewrite primary key fields then don't forget to specify *primary key* property.

Validators

We also can add custom validators for some particular field. Let's consider case when we need to validate string value and check that this value has valid format for phone number. To do this we need to create validation function which raise exception if value is not corrected.

```
import re

from aiohttp_admin2.mappers import Mapper
from aiohttp_admin2.mappers import fields
from aiohttp_admin2.mappers.exceptions import ValidationError


PHONE_REG = re.compile(r'^[0-9]{10,14}$')
```

(continues on next page)

(continued from previous page)

```
def phone_validator(value):
    if not PHONE_REG.match(value):
        raise ValidationError("wrong phone format")


class UserMapper(Mapper):
    """Mapper for user instance."""
    name = fields.StringField(required=True)
    phone =  fields.StringField(validators=[phone_validator])


# return False because '1234' is not valid format for a phone number
UserMapper({'name': 'Mike', 'phone': '1234'}).is_valid()
```

You also can to use standard validators from the *aiohttp_admin2.mappers.validators* module.

```
from aiohttp_admin2.mappers import Mapper
from aiohttp_admin2.mappers import fields
from aiohttp_admin2.mappers.validators import length


class UserMapper(Mapper):
    """Mapper for user instance."""
    name = fields.StringField(validators=[length(max_value=10, min_value=3)])
```

2.2.3 Controllers

The controller is class that generate access to the your data based on some engine (Resource). Out of the box you have engines for different storages

- PostgreSQL
- MySQL
- MongoDB (in progress)

but you actually can easy to add your own engine.

The controller is framework and database agnostic part of the admin. It's mean that controller have not to know any about request/response, generation of urls, templates and so on. Also it have not to know about how to get/update/delete data from some database (this logic need to allocate into the resource class).

For the PostgreSQL, an easier way to create a controller is to use the *PostgresController*.

```
from aiohttp_admin2.controllers.postgres_controller import PostgresController


@postgres_injector.inject
class UserController(PostgresController, table=user):
    mapper = UserMapper
    name = 'user'
    per_page = 10
```

For the *MongoDB* and the *MySQL* you can use *MongoController* and *MySQLController* apropriate.

The Controller need to have connection for engine. For this goal we need to inject connection by *ConnectionInjector*.

```
from aiohttp_admin2.connection_injectors import ConnectionInjector


postgres_injector = ConnectionInjector()


async def init_db(app):
    # Context function for initialize connection to db
    engine = await aiopg.sa.create_engine(
        user='postgres',
        database='postgres',
        host='0.0.0.0',
        password='postgres',
    )
    app['db'] = engine

    # here we add connection for our injector
    postgres_injector.init(engine)
```

After that you can user *postgres_injector* to decorate your controllers. For *MongoController* you don't need to use *ConnectionInjector* because connection to db exist in table instance.

Note: If you don't need to customize some field or add new field in mapper that based on you model then you may don't put mapper in the controller class. In this case controller will generate this mapper instead of you. Examples which represented below are equals:

```
from aiohttp_admin2.controllers.postgres_controller import PostgresController

from aiohttp_admin2.mappers.generics import PostgresMapperGeneric
from aiohttp_admin2.mappers import fields


class UserMapper(PostgresMapperGeneric, table=user):
    """Mapper for user instance."""

@postgres_injector.inject
class UserController(PostgresController, table=user):
    # implicit specify a mapper
    mapper = UserMapper
    name = 'user'
    per_page = 10
```

```
@postgres_injector.inject
class UserController(PostgresController, table=user):
    name = 'user'
    per_page = 10
```

Common settings

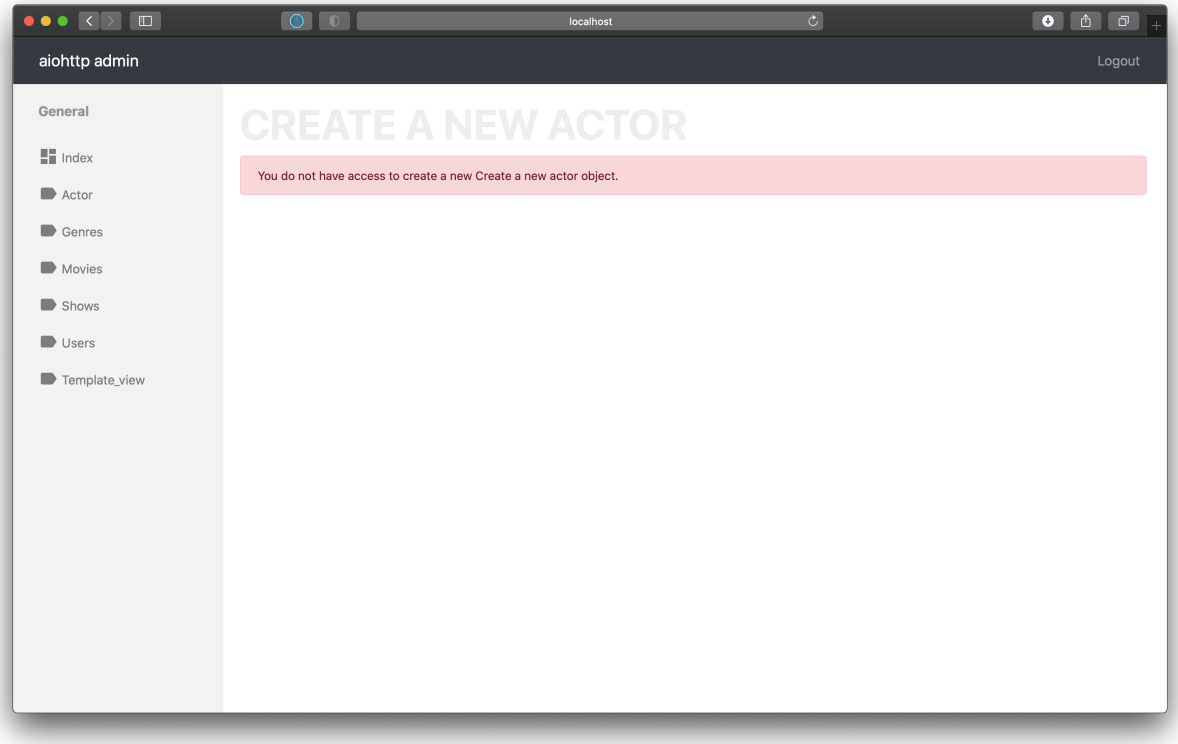
access settings

- *can_create (default True)* - *True* if can to edit an instance
- *can_update (default True)* - *True* if can to update an instance
- *can_delete (default True)* - *True* if can to delete an instance
- *can_view (default True)* - *True* if can to show an instance

If we remove access for some user to some controller then *aiohttp admin* will automatically hide all url to do this action from interface but if user visit current page directly then admin show error message.

snippet from the demo

```
class ActorController(PostgresController, table=actors):
    mapper = ActorMapper

    can_create = False
```

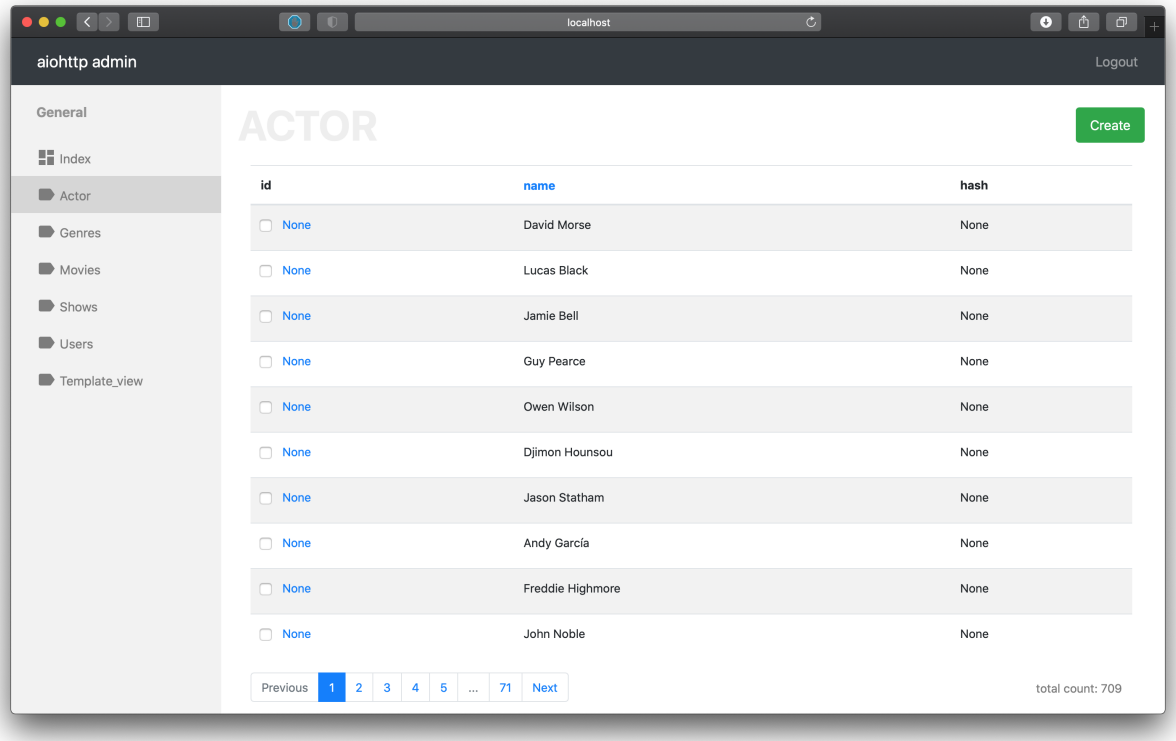


list settings

- *inline_fields (default ['id'])* - list of fields which will show on the list page

snippet from the demo

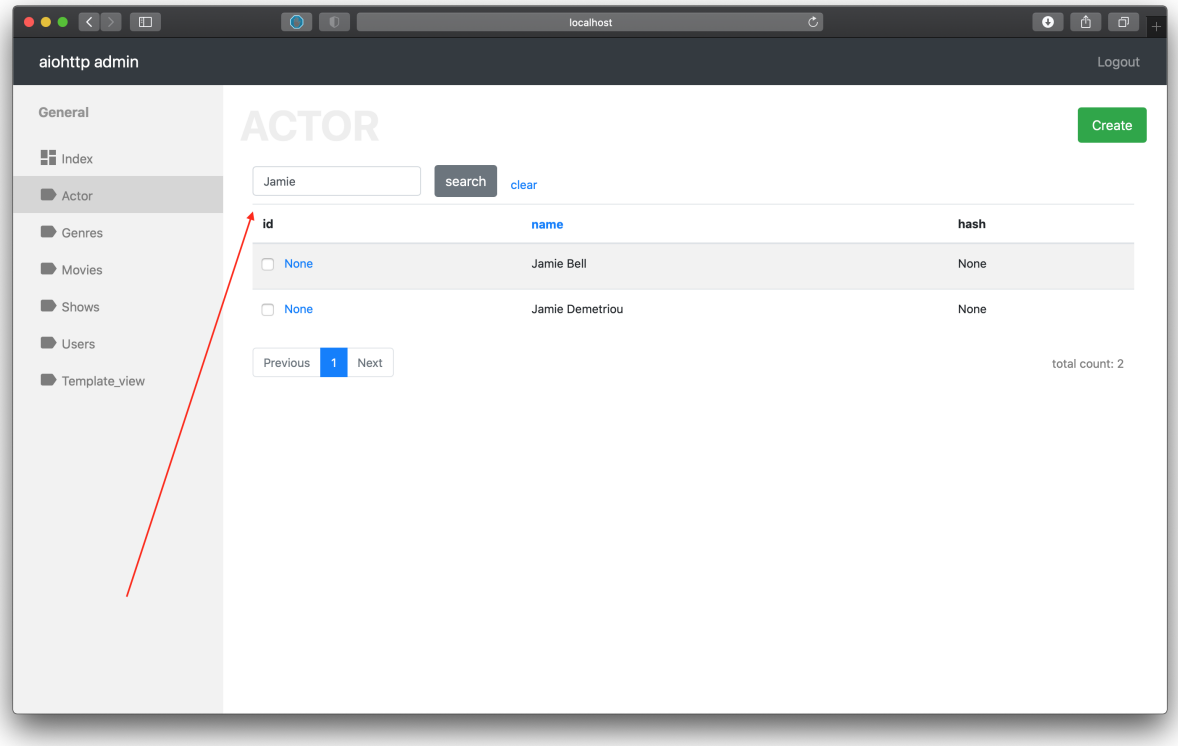
```
class ActorController(PostgresController, table=actors):
    mapper = ActorMapper

    inline_fields = ['id', 'name', 'hash', ]
```



For user on the list page we show only three fields.

- *search_fields (default [])* - list of fields which will use for do search (fields must be searchable)

```
class ActorController(PostgresController, table=actors):
    mapper = ActorMapper

    search_fields = ['name', ]
```

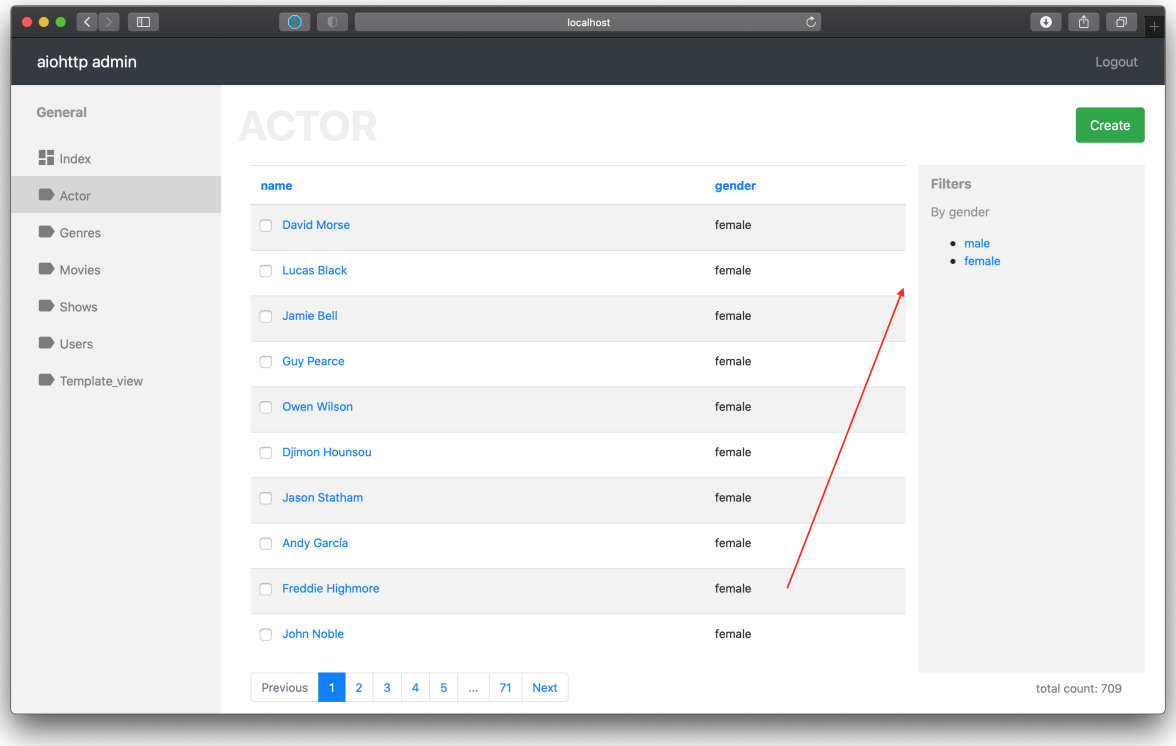


After specify current settings into admin interface you can see search input.

- *order_by (defaault `id`)* - name of field for the default sorting
- *per_page (defaault `50`)* - default count of items per page
- *list_filter (default [])* - list of fields which can to use filters

snippet from the demo

```
class ActorController(PostgresController, table=actors):
    mapper = ActorMapper

    inline_fields = ['name', 'gender', ]
    list_filter = ['gender', ]
```



After specify current settings into admin interface you can see filter sidebar with filter for corresponding field.

detail settings

- *read_only_fields (default [])* - list of fields which can't modify (on the detail page u can see current fields but can't edit)
- *exclude_update_fields (default `id`)* - list of fields which can't update (fields will be hide on update page)
- *exclude_create_fields (default `id`)* - list of fields which can't specify during create a new instance
- *fields (default `__all__`)* - list of available fields
- *autocomplete_search_fields (default [])* - list of feilds which will use to the autocomplete (when you update/create relation fields you just set primary key to input. For improve user experience you can set list of fields which will use to search suggestion items in current input.)

common settings

- *mapper* - a mapper for the current controller
- *relations_to_one (default [])* - list of *ToOneRelation* which describe one-to-one relation with other controllers
- *relations_to_many (default [])* - list of *ToManyRelation* which describe many-to-many relation with other controllers

Operations hooks

If you need to do some before/after create/update or delete some data you can use hooks:

- *pre_create* - run before create instance
- *pre_delete* - run before delete instance
- *pre_update* - run before update instance
- *post_create* - run after create instance
- *post_delete* - run after delete instance
- *post_update* - run after update instance

Let's say that you need to delete key in Redis after delete user instance in PostgeSQL. It might look like this

```
from aiohttp_admin2.controllers.postgres_controller import PostgresController
from .redis import redis_client


@postgres_injector.inject
class UserController(PostgresController, table=user):
    mapper = UserMapper
    name = 'user'

    async post_delete(self, pk):
        await redis_client.delete(f'user:{pk}')
```

Relations

One-to-one relation

To declare one-to-one relation in *aiohttp admin* you need to create the *ToOneRelation* from the *aiohttp_admin2.controllers.relations* module. Created object you need to add to *relations_to_one* list in apropriate controller.

snippet from the demo

```
class ActorMovieController(PostgresController, table=movies_actors):
    mapper = ActorMoviesMapper

    relations_to_one = [
        ToOneRelation(
            name='movie_id',
            field_name='movie_id',
            controller=MoviesController,
        ),
    ]
```

ToOneRelation

- *name* - name of relation
- *field_name* - name of the field which responsible for the current relation
- *controller* - controller of related models (can be callable object)

Many-to-many relation

To declare many-to-many relation in aiohttp admin you need to create the *ToManyRelation* from the *aiohttp_admin2.controllers.relations* module. Created object you need to add to *relations_to_many* list in apropriate controller.

snippet from the demo

```
class MoviesController(PostgresController, table=movies):
    mapper = MoviesMapper
    name = 'movies'

    relations_to_many = [
        ToManyRelation(
            name='Actors',
            left_table_pk='movie_id',
            relation_controller=lambda: ActorMovieController
        ),
    ]
```

ToManyRelation

- *name* - name of relation
- *left_table_pk* - name of the field which responsible for the current relation
- *relation_controller* - controller of related models (can be callable object)

Custom fields

On list page you can add custom fields or rewrite view of existing. Let's consider case from the demo related with image representation. Each movie has a picture url but on list page view want to show image block.

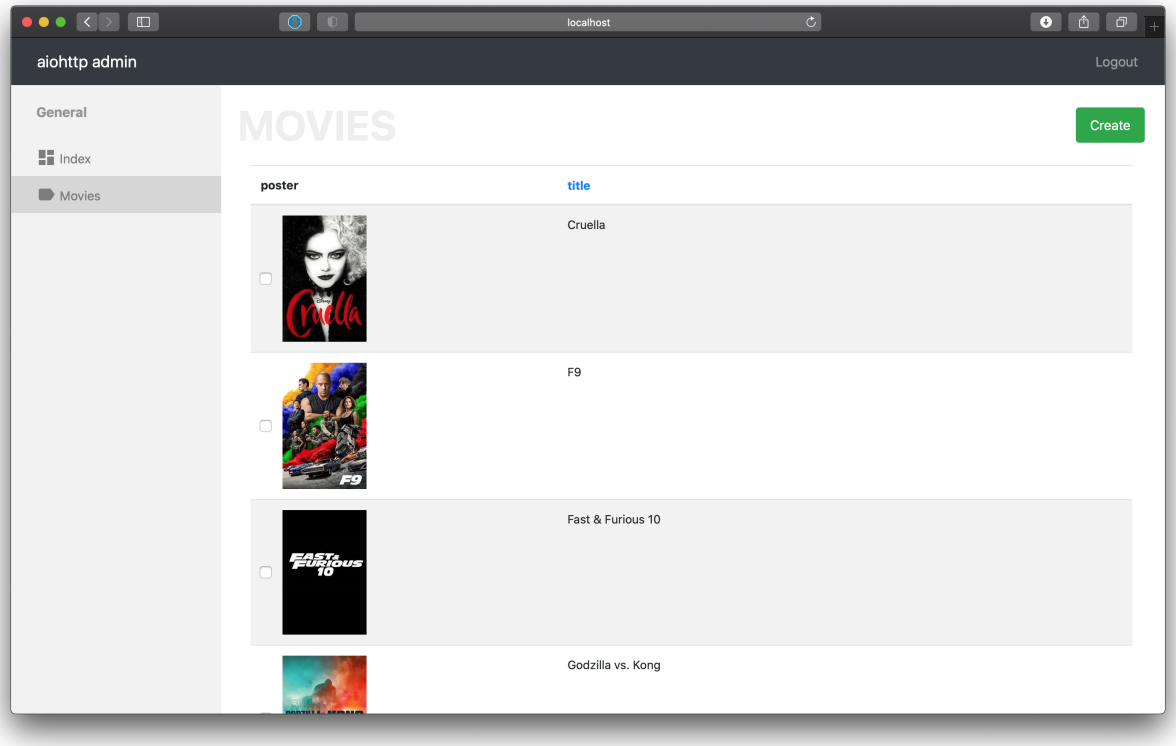
snippet from the demo

```
from markupsafe import Markup

class MoviesController(PostgresController, table=movies):
    mapper = MoviesMapper
    name = 'movies'
    inline_fields = ['poster', 'title', ]

    async def poster_field(self, obj):
        return Markup('<img src="{path}" width="100">')\
            .format(path=obj.data.poster_path)
```

For that into *inline_fields* we add new field *poster* and create a function *poster_field* (<field_name>_field) which receive as second argument the current *Instance* object. Also for give access use html in field without escaping we need to wrap our html in a *Markup* object.

To get the field value from the *Instance* object, we need to get the data property and try to get the field which we need.

```
async def poster_field(self, obj):
    return obj.data.poster_path
```



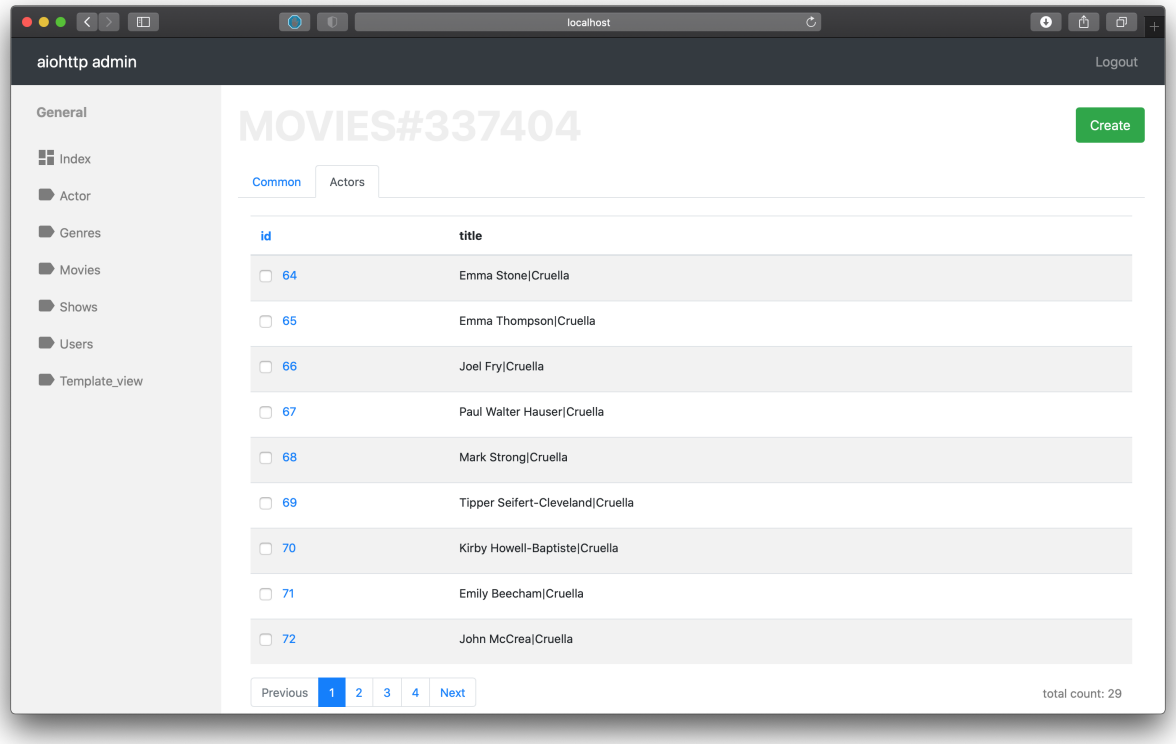
Also you can to get relation instances inside custom fields, for that just use *get_relation* method of *Instance* class to get related *Instance* object from other controller.

```
from aiohttp_admin2.controllers.relations import ToOneRelation


class ActorMovieController(PostgresController, table=movies_actors):
    mapper = ActorMoviesMapper
    inline_fields = ['id', 'title', ]

    relations_to_one = [
        ToOneRelation(
            # relation name
            name='movie_id',
            field_name='movie_id',
            controller=MoviesController,
        ),
        ToOneRelation(
            # relation name
            name='actor_id',
            field_name='actor_id',
            controller=ActorController,
        ),
    ]

    async def title_field(self, obj):
        # get via relation name
        actor = await obj.get_relation('actor_id')
```

(continues on next page)

(continued from previous page)

```
        # get via relation name
        movie = await obj.get_relation('movie_id')

        return actor.data.name + "|" + movie.data.title
```



Custom sort

To specify custom sorting we need to provide sort method into contorller class for the current field (<field_name>_sort). This function receive *is_reverse* that mean need we return reverse sorting or not.

In example below we add custom field which from json field *data* get key and implement sorting for this field in the *data_field_sort* method.

```
@postgres_injector.inject
class UsersController(PostgresController, table=users):
    mapper = UsersMapper

    inline_fields = ['id', 'data', ]

    async def data_field(self, obj) -> str:
        if obj.data.payload and isinstance(obj.data.payload, dict):
            return obj.data.data

        return ''

    def data_field_sort(self, is_reverse):
```

(continues on next page)

(continued from previous page)

```
        if is_reverse:
            return sa.text("payload ->> 'data' desc")
        return sa.text("payload ->> 'data'")
```

2.2.4 Views

This class use for represent data on the admin interface. The simples view which you can to create is *TemplateView*.

TemplateView

```
from aiohttp_admin2.view import TemplateView


class NewPage(TemplateView):
    title = 'new page'
    template_name = 'aiohttp_admin/my_template.html'
```

You can change specify template for you custom view as in example above or specify *content* variable in jinja's context.

```
from aiohttp_admin2.view import TemplateView


class NewPage(TemplateView):
    title = 'new page'

    async def get_context(self, req):
        return {
            **await super().get_context(req=req),
            "content": "My custom content"
        }
```

- template_name - path to template for current page

Dashboards view is just subclass of *TemplateView* which you can to customize in the same way.

Common settings

All view has properties which describe below:

- *is_hide_view* - if we don't want to show link on current views in the aside bar then we need to set True
- *group_name* - If views have the same group name then they will organize together into separate block in the aside bar
- *name* - This string will use as the pretty name of the current views in the admin interface.

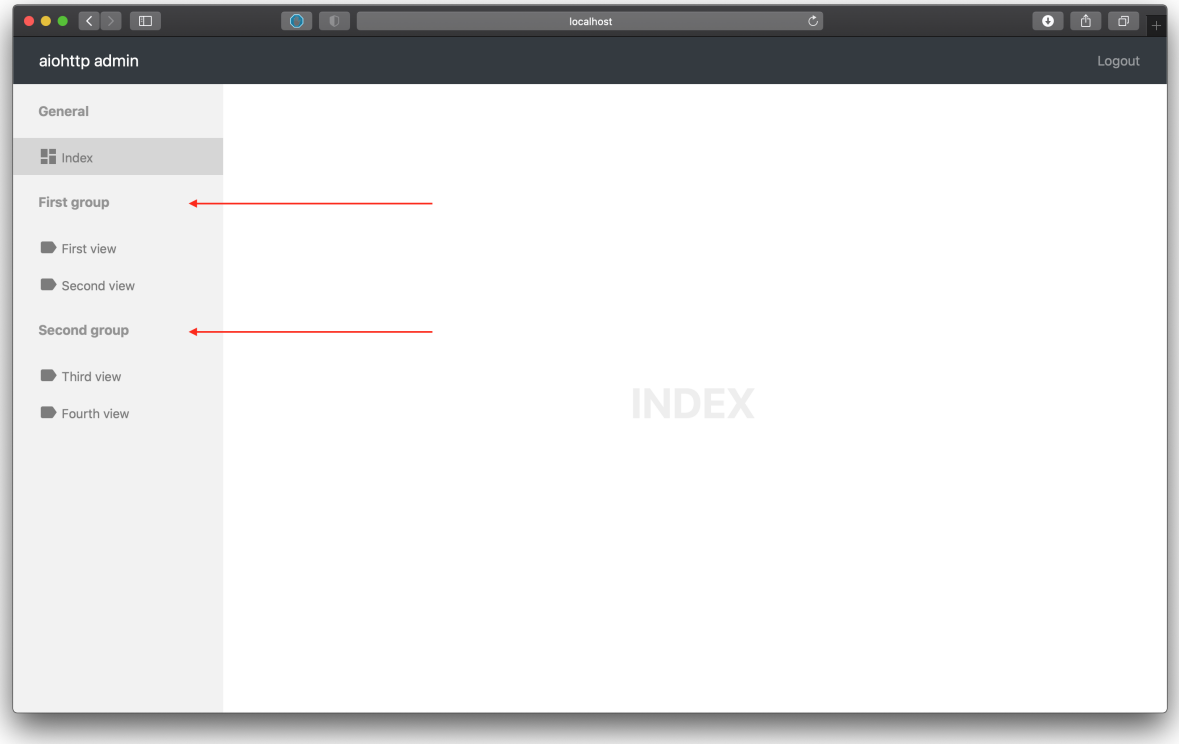
We can to see how below settings work together

```
from aiohttp_admin2.views import TemplateView


class FirstView(TemplateView):
```

(continues on next page)

(continued from previous page)

```
    group_name = 'first group'
    name = 'first view'


class SecondView(TemplateView):
    group_name = 'first group'
    name = 'second view'


class ThirdView(TemplateView):
    group_name = 'second group'
    name = 'third view'


class FourthView(TemplateView):
    group_name = 'second group'
    name = 'fourth view'


class FifthView(TemplateView):
    group_name = 'second group'
    name = 'fifth view'

    # hide current view
    is_hide_view = True
```



We can see that first and second views concat in single group in a side menu because common *group_name* and the

same story with third and fourth views but fifth doesn't exist in menu because the view has *is_hide_view* setting set to *True*.

- *index_url* - The url prefix path for all routes related with the current views
- *icon* - This string set a type of icon which will use in aside bar for the current views (full list of available icons you can to find here)

ControllerView

Controller view is view for representation information related with your models.

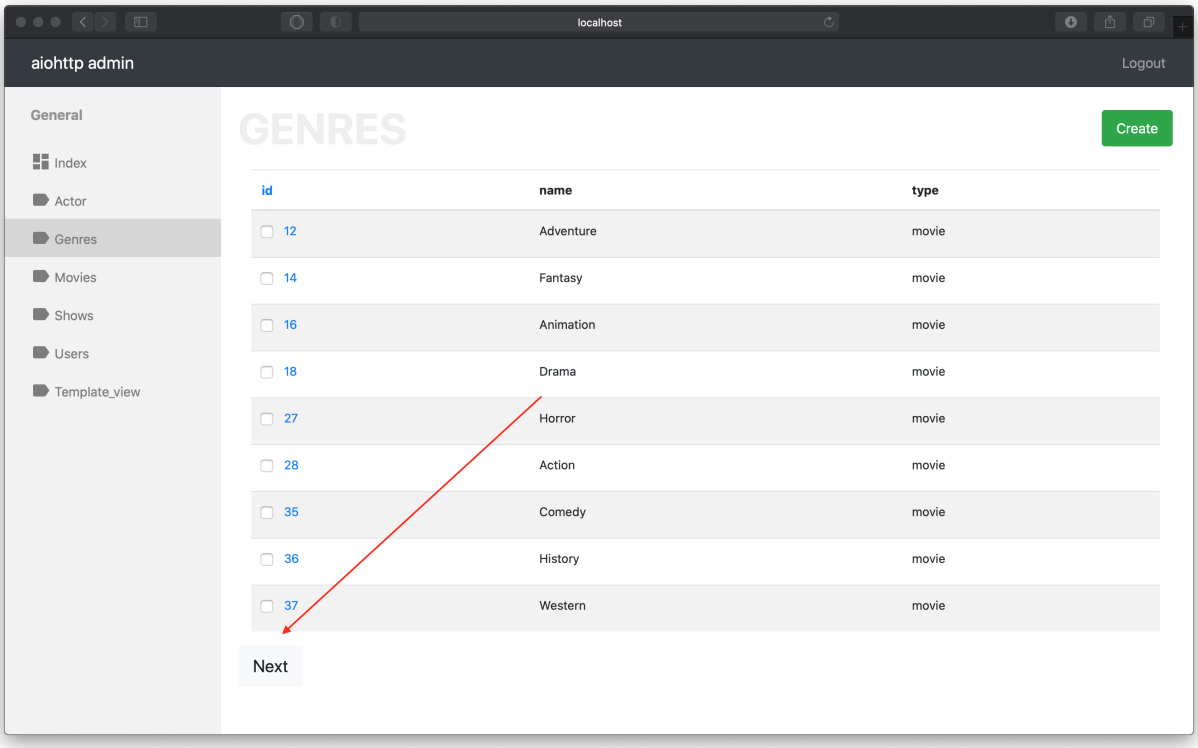
```
from aiohttp_admin2.view import ControllerView


class UserView(ControllerView):
    controller = UserController
```

You can specify templates which you wanna use for instead of default:

- *template_list_name* - the template for list page (with a simple pagination)
- *template_list_cursor_name* - the template for list page (with an infinite scroll)
- *template_detail_name* - the template for detail page in read only mode
- *template_detail_edit_name* - the template for detail page in edit mode
- *template_detail_create_name* - the template for create page
- *template_delete_name* - the template for delete page

also you can to specify:

- *infinite_scroll* (True/False default False) - if set to *True* then will use infinite scroll instead of standard pagination. It can be very helpful when table is so large and count query (which need to generate standard pagination bar) is so cost.



After specify current setting to *True* we can to see that standard pagination bar has been replaced by *Next* button.

- *search_filter* (default *SearchFilter*) - filter which will use for search (for search input at the top of list page)
- *fields_widgets* (default empty dict) - a map of field names and coresponding widngets. It's helpful if you want to specify a some special widget for the particular field.
- *type_widgets* (default empty dict) - a map of field type and coresponding widngets. It's helpful if you want to specify a some special widget for the particular type of field.
- *foreignkey_widget* (default *AutocompleteStringWidget*) - a widget which will use for the autocomplete

View's Widgets and Filters

The widgets and filters class need only to allocated path to the template and extra *.css* and *.js* files which need to corrected render of it. Custom widget have to inherit from *BaseWidget*. Custom filter have to inherit from *FilerBase*.

Custom Routes

You can use *@route* decorator to add custom endpoint to your view

```
from aiohttp_admin2.view import ControllerView
from aiohttp_admin2.views.aiohttp.views.utils import route

class UserView(ControllerView):
    controller = UserController

    @route(r'/{pk:\d+}/ban/', method='POST')
    def ban_user(self, req):
```

(continues on next page)

(continued from previous page)

```
        # ban_user(req.match_info['pk'])
        return await self.get_detail(req)
```

@*route* takes 2 parameters: url and method. Valid methods are: *POST*, *GET*, *PUT*, *DELETE*, *HEAD*. The URL must always start and end with /.

2.2.5 Templates

For generate pages *aiohttp_admin* use *jinja2*.

If you setup *aiohttp_jinja2* with not default *jinja_app_key* argument then you should initialize admin interface with your *jinja_app_key* argument.

```
aiohttp_admin.setup_admin(app, jinja_app_key='my_jinja_value')
```

Overriding jinja templates

You can rewrite native templates for *aiohttp_admin*. For that you should create *aiohttp_admin* directory into templates's directory for the *jinja2* and create your template with name of template witch you want to rewrite.

The full list of templates you can see below:

- aiohttp_admin/blocks/header.html - the header for base layout
- aiohttp_admin/layouts/base.html - the base layout
- aiohttp_admin/layouts/create_page.html - the content for create page
- aiohttp_admin/layouts/delete_page.html - the content for confirm delete page
- aiohttp_admin/layouts/detail_view_page.html - the content for detail page in read only mode
- aiohttp_admin/layouts/detail_edit_page.html - the content for edit page
- aiohttp_admin/layouts/custom_page.html - the content for custom page
- aiohttp_admin/layouts/custom_tab_page.html - the content for custom tab
- aiohttp_admin/layouts/list_page.html - the content for list page (with a simple pagination)
- aiohttp_admin/layouts/list_cursor_page.html - the content for list page (with an infinite scroll)
- aiohttp_admin/blocks/from/form.html - the main form for create and update
- aiohttp_admin/blocks/from/field_errors.html - the macro for form's errors
- aiohttp_admin/blocks/from/field_title.html - the macro for form's title
- aiohttp_admin/blocks/from/fields/* - the macros for different types of fields
- aiohttp_admin/blocks/filters/* - the macros for different types of filters (in the left aside bar)
- aiohttp_admin/blocks/pagination.html - the pagination block
- aiohttp_admin/blocks/cursor_pagination.html - the infinity scroll pagination block
- aiohttp_admin/blocks/list_action_buttons.html - the list actions for list page
- aiohttp_admin/blocks/list_cell.html - the macro for table cell
- aiohttp_admin/blocks/list_objects_block.html - the table for list page

- aiohttp_admin/blocks/list_objects_header_block.html - the header of table for list page
- aiohttp_admin/blocks/messages.html - the macro for message's notification bar
- aiohttp_admin/blocks/nav_aside.html - the aside with pages links
- aiohttp_admin/blocks/tabs_bar.html - the template for tabs

Overriding view templates

You also can specify template for some special *ControllerView*.

```
class UserPage(ControllerView):
    controller = UserController

    template_list_name = 'aiohttp_admin/layouts/list_page.html'
    template_list_cursor_name = 'aiohttp_admin/layouts/list_cursor_page.html'
    template_detail_name = 'aiohttp_admin/layouts/detail_view_page.html'
    template_detail_edit_name = 'aiohttp_admin/layouts/detail_edit_page.html'
    template_detail_create_name = 'aiohttp_admin/layouts/create_page.html'
    template_delete_name = 'aiohttp_admin/layouts/delete_page.html'
```

2.2.6 Resources

So, we already told that *Resources* is a class which implement method to work with some particular database. If you want to implement your own *Resources* you need just inherit from *AbstractResource* and implement methods which described below:

- **get_one** - Get one an instance from a storage. This method receive primary key of an database's object and return the *Instance* if object exist else raise the *InstanceDoesNotExist* exception.
- **get_many** - Get many instances by ids from a storage. This method will use as a dataloader. This method mainly will use on list page in cases when need to show field with data from related model for prevent N + 1. This method receive list of primary keys of an database's objects and name of primary key after that return dict where keys are primary keys and as a values corresponding Instance objects (InstanceMapper).
- **delete** - Delete instance. This method receive primary key of instance and delete it or raise the *InstanceDoesNotExist* exception if object doesn't exist.
- **create** - Create instance. This method receive *Instance* object and return it from databases after create.
- **update** - Update instance. This method receive primary key and *Instance* object after that update an object in databases and return corresponding *Instance* object.
- **get_list** - Get list of instances. This method will use for show list of instances. The current method have to implement possible to pagination, filtering and sorting.

PostgresResource

- **get_list_select** - In this method you can redefine query. It might helpful when you need to use need to do join or add to response a field based on some aggregation

Filters

For filtering data resources use Filters objects. Filter object can apply condition expressions to query. Each filter inherit from *ABCFilter* class and provide *apply* method which will apply to query conditions.

2.3 aiohttp_admin2

2.3.1 aiohttp_admin2 package

Subpackages

aiohttp_admin2.controllers package

Submodules

aiohttp_admin2.controllers.controller module

aiohttp_admin2.controllers.exceptions module

Module contents

aiohttp_admin2.mappers package

Subpackages

aiohttp_admin2.mappers.fields package

Submodules

aiohttp_admin2.mappers.fields.abc module

aiohttp_admin2.mappers.fields.common_fields module

aiohttp_admin2.mappers.fields.mongo_fields module

Module contents

Submodules

aiohttp_admin2.mappers.base module

aiohttp_admin2.mappers.exceptions module

aiohttp_admin2.mappers.generics module

Module contents

aiohttp_admin2.resources package

Subpackages

aiohttp_admin2.resources.dict_resource package

Submodules

aiohttp_admin2.resources.dict_resource.dict_resource module

aiohttp_admin2.resources.dict_resource.filters module

Module contents

aiohttp_admin2.resources.mongo_resource package

Submodules

aiohttp_admin2.resources.mongo_resource.filters module

aiohttp_admin2.resources.mongo_resource.mongo_resource module

Module contents

aiohttp_admin2.resources.mysql_resource package

Submodules

aiohttp_admin2.resources.mysql_resource.mysql_resource module

Module contents

aiohttp_admin2.resources.postgres_resource package

Submodules

aiohttp_admin2.resources.postgres_resource.filters module

aiohttp_admin2.resources.postgres_resource.postgres_resource module

aiohttp_admin2.resources.postgres_resource.utils module

Module contents

Submodules

aiohttp_admin2.resources.abc module

aiohttp_admin2.resources.exceptions module

aiohttp_admin2.resources.types module

Module contents

aiohttp_admin2.views package

Subpackages

aiohttp_admin2.views.aiohttp package

Subpackages

aiohttp_admin2.views.aiohttp.views package

Submodules

aiohttp_admin2.views.aiohttp.views.base module

aiohttp_admin2.views.aiohttp.views.controller_view module

aiohttp_admin2.views.aiohttp.views.dashboard module

aiohttp_admin2.views.aiohttp.views.many_to_many_tab_view module

aiohttp_admin2.views.aiohttp.views.tab_base_view module

aiohttp_admin2.views.aiohttp.views.tab_template_view module

aiohttp_admin2.views.aiohttp.views.template_view module

aiohttp_admin2.views.aiohttp.views.utils module

Module contents

Submodules

aiohttp_admin2.views.aiohttp.admin module

aiohttp_admin2.views.aiohttp.exceptions module

aiohttp_admin2.views.aiohttp.setup module

aiohttp_admin2.views.aiohttp.utils module

Module contents

Submodules

aiohttp_admin2.views.filters module

aiohttp_admin2.views.widgets module

Module contents

Submodules

aiohttp_admin2.connection_injectors module

aiohttp_admin2.exceptions module

Module contents

2.4 Contributing

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

You can contribute in many ways:

2.4.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/arfey/aiohttp_admin2/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" and "help wanted" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

Write Documentation

aiohttp admin 2 could always use more documentation, whether as part of the official aiohttp admin 2 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/arfey/aiohttp_admin2/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 :)

2.4.2 Get Started!

Ready to contribute? Here's how to set up *aiohttp_admin2* for local development.

1. Fork the *aiohttp_admin2* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/aiohttp_admin2.git
```

3. You need to have preinstalled poetry and docker.

   $ poetry config virtualenvs.create true –local # create virtualenv in project directory $ cd aiohttp_admin2/ $ poetry install –with "dev, test" –all-extras

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 linters and tests:

```
$ make lint
$ make test
```

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

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

2.4.3 Tips

To run a subset of tests:

```
$ pytest --slow -v -s -p no:warnings  tests/resources/common_resource
```

Run some particular test:

```
$ pytest --slow -v -s -p no:warnings  tests/resources/common_resource/test_create.
↪py::test_create_with_error
```

All features you can to test in demo application.

2.5 Credits

2.5.1 Development Lead

- Mykhailo Havelia <misha.gavela@gmail.com>

2.5.2 Contributors

None yet. Why not be the first?

2.6 History

Changelog

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX

A

M