Flask-Generic-Views Documentation

Release 0.1.1

Daniel Knell

Jun 13, 2017

Contents

Flask-Generic-Views is an extension to Flask that provides a set of generic class based views. It aims to simplify applications by providing a set of well tested base classes and pluggable views for common tasks.

CHAPTER 1

Getting Started

Installation

A minimal install without database support can be performed with the following:

```
pip install flask-generic-views
```

Optional packages

To avoid excessive dependencies some of the dependencies are broken out into setup tools "extra" feature.

You can safely mix multiple of the following in your `requirements.txt`.

SQLAlchemy

To install flask-generic-views with SQLAlchemy support use the following:

```
pip install flask-generic-views[sqlalchemy]
```

All

To install flask-generic-views with all optional dependencies use the following:

```
pip install flask-generic-views[all]
```

Quick Start

A Minimal Application

A minimal Flask-Generic-Views application looks something like this:

```
from flask import Flask
from flask_generic_views import TemplateView, RedirectView

app = Flask(__name__)

index = RedirectView('index', url='/home')

app.add_url_rule('/', view_func=index)

home = TemplateView('home', template_name='home.html')

app.add_url_rule('/home', view_func=home)

if __name__ == '__main__':
    app.run()
```

Save this as `app.py`, and create a template for your *home* view to render.

```
<h1>Hello World</h1>
```

Save this as `templates/home.html` and run the application with your Python interpreter.

```
$ python app.py
* Running on http://127.0.0.1:5000/
```

If you head to http://127.0.0.1:5000/ now you should see the rendered template.

An SQLAlchemy Application

```
from flask import Flask
from flask.ext.generic_views.sqlalchemy import (CreateView,
                                                DeleteView,
                                                DetailView,
                                                ListView,
                                                UpdateView)
from flask.ext.sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////tmp/test.db'
app.config['SECRET_KEY'] = '5up3r5ekr3t'

db = SQLAlchemy(app)

class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80))
    body = db.Column(db.Text(120))
    created_at = db.Column(db.DateTime, default=datetime.utcnow)
```

```
# index

index_view = ListView.as_view('index', model=Post,
                              ordering=[Post.created_at],
                              per_page=20)

app.add_url_rule('/', view_func=index_view)

# show

show_view = DetailView.as_view('show', model=Post)

app.add_url_rule('/<int:pk>', view_func=show_view)

# new

new_view = CreateView.as_view('new', model=Post,
                              fields=('name', 'body'),
                              success_url='/{id}')

app.add_url_rule('/new', view_func=new_view)

# edit

edit_view = UpdateView.as_view('edit', model=Post,
                               fields=('name', 'body'),
                               success_url='/{id}')

app.add_url_rule('/<int:pk>/edit', view_func=edit_view)

# delete

delete_view = DeleteView.as_view('delete', model=Post,
                                 success_url='/')

app.add_url_rule('/<int:pk>/delete', view_func=delete_view)

if __name__ == '__main__':
    app.run()
```

CHAPTER 2

Reference

API

Core

View logic is often repetitive, there are standard patterns we repeat over again both within and across projects, and reimplementing the same patterns can be a bore.

These views take some of those patterns and abstract them so you can create views for common tasks quickly without having to write too much code.

Tasks such as rendering a template or redirecting to a new url can be performed by passing parameters at instantiation without defining additional classes.

Views

class `flask_generic_views.core.`**`View`**(***kwargs*)

Bases: `flask.views.View`

The master class-based base view.

All other generic views inherit from this base class. This class itself inherits from `flask.views.View` and adds a generic constructor, that will convert any keyword arguments to instance attributes.

```
class GreetingView(View):
    greeting = 'Hello'

    def dispatch_request(self):
        return "{} World!".format(self.greeting)

bonjour_view = GreetingView.as_view('bonjour', greeting='Bonjour')

app.add_url_rule('/bonjour, view_func=bonjour_view)
```

The above example shows a generic view that allows us to change the greeting while setting up the URL rule.

class flask_generic_views.core.**MethodView**(***kwargs*)

Bases: flask.views.MethodView, *flask_generic_views.core.View*

View class that routes to methods based on HTTP verb.

This view allows us to break down logic based on the HTTP verb used, and avoid conditionals in our code.

```
class GreetingView(View):
    greeting = 'Hello'

    def get(self):
        return "{} World!".format(self.greeting)

    def post(self):
        name = request.form.get('name', 'World')

        return "{} {}!".format(self.greeting, name)

bonjour_view = GreetingView.as_view('bonjour', greeting='Bonjour')

app.add_url_rule('/bonjour, view_func=bonjour_view)
```

The above example will process the request differently depending on wether it was a HTTP POST or GET.

class flask_generic_views.core.**TemplateView**(***kwargs*)

Bases: *flask_generic_views.core.TemplateResponseMixin*, *flask_generic_views.core.ContextMixin*, *flask_generic_views.core.MethodView*

Renders a given template, with the context containing parameters captured by the URL rule.

```
class AboutView(View):
    template_name = 'about.html'

    def get_context_data(self, **kwargs):
        kwargs['staff'] = ('John Smith', 'Jane Doe')

        return super(AboutView, self).get_context_data(self, **kwargs)

app.add_url_rule('/about', view_func=AboutView.as_view('about'))
```

The `TemplateView` can be subclassed to create custom views that render a template.

```
about_view = TemplateView.as_view('about', template_name='about.html')

app.add_url_rule('/about', view_func=about_view,  defaults={
    'staff': ('John Smith', 'Jane Doe')
})
```

It can also be used directly in a URL rule to avoid having to create additional classes.

get(***kwargs*)

Handle request and return a template response.

Any keyword arguments will be passed to the views context.

Parameters **kwargs** (*dict*) – keyword arguments from url rule

Returns response

Return type werkzeug.wrappers.Response

class flask_generic_views.core.**RedirectView**(***kwargs*)

Bases: *flask_generic_views.core.View*

Redirects to a given URL.

The given URL may contain dictionary-style format fields which will be interpolated against the keyword arguments captured from the URL rule using the `format()` method.

An URL rule endpoint may be given instead, which will be passed to `url_for()` along with any keyword arguments captured by the URL rule.

When no URL can be found a `Gone` exception will be raised.

```
class ShortView(RedirectView):

    permanent = True
    query_string = True
    endpoint = 'post-detail'

    def get_redirect_url(self, **kwargs):
        post = Post.query.get_or_404(base62.decode(kwargs['code']))
        kwargs['slug'] = post.slug
        return super(ShortView, self).get_redirect_url(**kwargs)

   short_view = ShortView.as_view('short')

   app.add_url_rule('/s/<code>', view_func=short_view)
```

The above example will redirect "short links" where the pk is base62 encoded to the correct url.

```
google_view = RedirectView.as_view('google', url='http://google.com/')

app.add_url_rule('/google', view_func=google_view)
```

It can also be used directly in a URL rule to avoid having to create additional classes for simple redirects.

url = None

String containing the URL to redirect to or None to raise a `Gone` exception.

endpoint = None

The name of the endpoint to redirect to. URL generation will be done using the same keyword arguments as are passed in for this view.

permanent = False

Whether the redirect should be permanent. The only difference here is the HTTP status code returned. When True, then the redirect will use status code 301. When False, then the redirect will use status code 302.

query_string = False

Whether to pass along the query string to the new location. When True, then the query string is appended to the URL. When False, then the query string is discarded.

dispatch_request(***kwargs*)

Redirect the user to the result of.

get_redirect_url(), when by default it will issue a 302 temporary redirect, except when *permanent* is set to the `True`, then a 301 permanent redirect will be used.

When the redirect URL is None, a `Gone` exception will be raised.

Any keyword arguments will be used to build the URL.

Parameters **kwargs** (*dict*) – keyword arguments from url rule

Returns response

Return type werkzeug.wrappers.Response

get_redirect_url(***kwargs*)
Retrieve URL to redirect to.

When *url* is not None then it is returned after being interpolated with the keyword arguments using `format()`.

When *url* is None and *endpoint* is not None then it is passed to `url_for()` with the keyword arguments, and any query string is removed.

The query string from the current request can be added to the new URL by setting *query_string* to `True`.

Parameters **kwargs** (*dict*) – keyword arguments

Returns URL

Return type str

class flask_generic_views.core.**FormView**(***kwargs*)
Bases: *flask_generic_views.core.TemplateResponseMixin*, *flask_generic_views.core.BaseFormView*

View class to display a `Form`. When invalid it shows the form with validation errors, when valid it redirects to a new URL.

```
class ContactForm(Form):
    email = StringField('Name', [required(), email()])
    message = TextAreaField('Message', [required()])


class ContactView(FormView):
    form_class = ContactForm
    success_url = '/thanks'
    template_name = 'contact.html'

    def form_valid(self, form):
        message = Message('Contact Form', body=form.message.data,
                          recipients=['contact@example.com'],
                          sender=form.email.data)

        mail.send(message)

        super(ContactView).form_valid(form)
```

The above example will render the template `contact.html` with an instance of `ContactForm` in the context variable `view`, when the user submits the form with valid data an email will be sent, and the user redirected to `/thanks`, when the form is submitted with invalid data `content.html` will be rendered again, and the form will contain any error messages.

Helpers

class flask_generic_views.core.**ContextMixin**
Bases: `object`

Default handling of view context data any mixins that modifies the views context data should inherit from this class.

```
class RandomMixin(ContextMixin):
    def get_context_data(self, **kwargs):
        kwargs.setdefault('number', random.randrange(1, 100))

        return super(RandomMixin, self).get_context_data(**kwargs)
```

get_context_data(***kwargs*)

Returns a dictionary representing the view context. Any keyword arguments provided will be included in the returned context.

The context of all class-based views will include a `view` variable that points to the *View* instance.

Parameters **kwargs** (*dict*) – context

Returns context

Return type dict

class flask_generic_views.core.**TemplateResponseMixin**

Bases: object

Creates Response instances with a rendered template based on the given context. The choice of template is configurable and can be customised by subclasses.

```
class RandomView(TemplateResponseMixin, MethodView):
    template_name = 'random.html'

    def get(self):
        context = {'number': random.randrange(1, 100)}
        return self.create_response(context)

random_view = RandomView.as_view('random')

app.add_url_rule('/random, view_func=random_view)
```

mimetype = None

The mime type type to use for the response. The mimetype is passed as a keyword argument to *response_class*.

response_class = flask.Response

The Response class to be returned by *create_response()*.

template_name = None

The string containing the full name of the template to use. Not defining *template_name* will cause the default implementation of `get_template_names()` to raise a `NotImplementedError` exception.

create_response(*context=None*, ***kwargs*)

Returns a *response_class* instance containing the rendered template.

If any keyword arguments are provided, they will be passed to the constructor of the response class.

Parameters

- **context** (*dict*) – context for template
- **kwargs** (*dict*) – response keyword arguments

Returns response

Return type werkzeug.wrappers.Response

get_template_list()
Returns a list of template names to use for when rendering the template.

The default implementation will return a list containing *template_name*, when not specified a NotImplementedError exception will be raised.

Returns template list

Return type list

Raises **NotImplementedError** – when *template_name* is not set

class flask_generic_views.core.**FormMixin**
Bases: *flask_generic_views.core.ContextMixin*

Provides facilities for creating and displaying forms.

data = {}
A dictionary containing initial data for the form.

form_class = None
The form class to instantiate.

success_url = None
The URL to redirect to when the form is successfully processed.

prefix = ''
The prefix for the generated form.

form_invalid(*form*)
Creates a response using the return value of.

get_context_data().

Parameters **form** (*flask_wtf.Form*) – form instance

Returns response

Return type werkzeug.wrappers.Response

form_valid(*form*)
Redirects to *get_success_url()*.

Parameters **form** (*flask_wtf.Form*) – form instance

Returns response

Return type werkzeug.wrappers.Response

get_context_data(***kwargs*)
Extends the view context with a form variable containing the return value of *get_form()*.

Parameters **kwargs** (*dict*) – context

Returns context

Return type dict

get_data()
Retrieve data to pass to the form.

By default returns a copy of *data*.

Returns data

Return type dict

get_form()
Create a Form instance using *get_form_class()* using *get_form_kwargs()*.

Returns form

Return type flask_wtf.Form

get_form_class()
Retrieve the form class to instantiate.

By default returns *form_class*.

Returns form class

Return type type

Raises **NotImplementedError** – when *form_class* is not set

get_form_kwargs()
Retrieve the keyword arguments required to instantiate the form.

The `data` argument is set using *get_data()* and the `prefix` argument is set using *get_prefix()*. When the request is a POST or PUT, then the `formdata` argument will be set using *get_formdata()*.

Returns keyword arguments

Return type dict

get_formdata()
Retrieve prefix to pass to the form.

By default returns a `werkzeug.datastructures.CombinedMultiDict` containing `flask.request.form` and `flask.request.files`.

Returns form / file data

Return type werkzeug.datastructures.CombinedMultiDict

get_prefix()
Retrieve prefix to pass to the form.

By default returns *prefix*.

Returns prefix

Return type str

get_success_url()
Retrive the URL to redirect to when the form is successfully validated.

By default returns *success_url*.

Returns URL

Return type str

Raises **NotImplementedError** – when *success_url* is not set

class flask_generic_views.core.**ProcessFormView**(***kwargs*)
Bases: *flask_generic_views.core.MethodView*

Provides basic HTTP GET and POST processing for forms.

This class cannot be used directly and should be used with a suitable mixin.

get(***kwargs*)
Creates a response using the return value of.

get_context_data().

Parameters **kwargs** (*dict*) – keyword arguments from url rule

Returns response

Return type werkzeug.wrappers.Response

post(**kwargs)
Constructs and validates a form.

When the form is valid form_valid() is called, when the form is invalid form_invalid() is called.

Parameters **kwargs** (*dict*) – keyword arguments from url rule

Returns response

Return type werkzeug.wrappers.Response

put(**kwargs)
Passes all keyword arguments to *post()*.

Parameters **kwargs** (*dict*) – keyword arguments from url rule

Returns response

Return type werkzeug.wrappers.Response

class flask_generic_views.core.**BaseFormView**(**kwargs)
Bases: *flask_generic_views.core.FormMixin*, *flask_generic_views.core.ProcessFormView*

View class to process handle forms without response creation.

SQLAlchemy

Views logic often relates to retrieving and persisting data in a database, these views cover some of the most common patterns for working with models using the SQLAlchemy library.

Tasks such as displaying, listing, creating, updating, and deleting objects can be performed by passing parameters at instantiation without defining additional classes.

Views

class flask_generic_views.sqlalchemy.**DetailView**(**kwargs)
Bases: *flask_generic_views.sqlalchemy.SingleObjectTemplateResponseMixin*, *flask_generic_views.sqlalchemy.BaseDetailView*

Renders a given template,, with the context containing an object retrieved from the database.

```
class PostDetailView(DetailView):
    model = Post

    def context_data(self, **kwargs):
        kwargs.setdefault('now', datetime.now())

post_detail = PostDetailView.as_view('post_detail')

app.add_url_rule('/posts/<pk>', view_func=post_detail)
```

The above example will render the `post_detail.html` template from a folder named after the current blueprint, when no blueprint is used it will look in the root template folder. The view context will contain the object as `object` and the current date-time as `now`.

```
post_detail = DetailView.as_view('post_detail', model=Post)

app.add_url_rule('/post/<pk>', view_func=post_detail)
```

It can also be used directly in a URL rule to avoid having to create additional classes.

```
{# post_detail.html #}
<h1>{{ object.title }}</h1>
<p>{{ object.body }}</p>
<p>Published: {{ object.published_at }}</p>
<p>Date: {{ now }}</p>
```

class flask_generic_views.sqlalchemy.**ListView**(***kwargs*)

Bases: *flask_generic_views.sqlalchemy.MultipleObjectTemplateResponseMixin*, *flask_generic_views.sqlalchemy.BaseListView*

Renders a given template,, with the context containing a list of objects retrieved from the database.

```
class PostListView(DetailView):
    model = Post

    def context_data(self, **kwargs):
        kwargs.setdefault('now', datetime.now())

post_list = PostDetailView.as_view('post_list')

app.add_url_rule('/posts', view_func=post_list)
```

The above example will render the `post_list.html` template from a folder named after the current blueprint, when no blueprint is used it will look in the root template folder. The view context will contain the list of objects as `post_list` and the current date-time as `now`.

```
post_list = ListView.as_view('post_list', model=Post)

app.add_url_rule('/posts', view_func=post_list)
```

It can also be used directly in a URL rule to avoid having to create additional classes.

```
{# post_list.html #}
<ul>
{% for post in object_list  %}
  <li>{{ post.title }}</li>
{% else %}
  <li>No posts found.</li>
{% endfor %}
</ul>
```

class flask_generic_views.sqlalchemy.**CreateView**(***kwargs*)

Bases: *flask_generic_views.sqlalchemy.SingleObjectTemplateResponseMixin*, *flask_generic_views.sqlalchemy.BaseCreateView*

View class to display a form for creating an object. When invalid it shows the form with validation errors, when valid it saves a new object to the database and redirects to a new URL.

```
class PostCreateView(FormView):
    model = Post
    fields = ('title', 'body')
    success_url = '/posts/{id}'

post_create = PostCreateView.as_view('post_create')

app.add_url_rule('/posts/new', view_func=post_create)
```

The above example will render the template `post_form.html` with an instance of `flask_wtf.Form` in the context variable `form` with fields based on `fields` and `ModelFormView.model`, when the user submits the form with valid data an instance of Post will be saved to the database, and the user redirected to its page, when the form is submitted with invalid data `post_form.html` will be rendered again, and the form will contain any error messages.

```
post_create = CreateView.as_view('post_create', model=Post,
                                 fields=('title', 'body'),
                                 success_url = '/posts/{id}')

app.add_url_rule('/posts/new', view_func=post_create)
```

It can also be used directly in a URL rule to avoid having to create additional classes.

```
{# post_form.html #}
<form action="" method="post">
  <p>{{ form.title.label }} {{ form.title }}</p>
  <p>{{ form.title.label }} {{ form.title }}</p>
  <input type="submit" value="Save">
</form>
```

template_name_suffix = '_form'
The suffix to use when generating a template name from the model class

class flask_generic_views.sqlalchemy.**UpdateView**(***kwargs*)
Bases: *flask_generic_views.sqlalchemy.SingleObjectTemplateResponseMixin*, *flask_generic_views.sqlalchemy.BaseUpdateView*

View class to display a form for updating an object. When invalid it shows the form with validation errors, when valid it saves the updated object to the database and redirects to a new URL.

```
class PostUpdateView(FormView):
    model = Post
    fields = ('title', 'body')
    success_url = '/posts/{id}'

post_update = PostUpdateView.as_view('post_update')

app.add_url_rule('/posts/new', view_func=post_update)
```

The above example will render the template `post_form.html` with an instance of `flask_wtf.Form` in the context variable `form` with fields based on `fields` and `ModelFormView.model`, when the user submits the form with valid data an instance of Post will be updated in the database, and the user redirected to its page, when the form is submitted with invalid data `post_form.html` will be rendered again, and the form will contain any error messages.

```
post_update = UpdateView.as_view('post_update', model=Post,
                                 fields=('title', 'body'),
```

```
                                    success_url = '/posts/{id}')

app.add_url_rule('/posts/<pk>/edit', view_func=post_update)
```

It can also be used directly in a URL rule to avoid having to create additional classes.

```
{# post_form.html #}
<form action="" method="post">
  <p>{{ form.title.label }} {{ form.title }}</p>
  <p>{{ form.title.label }} {{ form.title }}</p>
  <input type="submit" value="Save">
</form>
```

template_name_suffix = '_form'
The suffix to use when generating a template name from the model class

class flask_generic_views.sqlalchemy.**DeleteView**(***kwargs*)
Bases: *flask_generic_views.sqlalchemy.SingleObjectTemplateResponseMixin*, *flask_generic_views.sqlalchemy.BaseDeleteView*

Displays a confirmation page and deletes an existing object. The object will ve deleted on POST or DELETE requests, for GET requests a conformation page will be shown which should contain a form to POST to the same URL.

```
class PostDeleteView(DeleteView):
    model = Post
    success_url = "/posts"

post_delete = PostDeleteView.as_view('post_delete')

app.add_url_rule('/posts/<id>', view_func=post_delete)
```

The above will render `post_delete.html` when accessed by GET request, for a POST or DELETE request an instance will be deleted from the database and the user redirected to `/posts`.

```
post_delete = DeleteView.as_view('post_delete', model=Post,
                                 success_url = '/posts')

app.add_url_rule('/posts/<id>/delete', view_func=post_delete)
```

It can also be used directly in a URL rule to avoid having to create additional classes.

```
{# post_form.html #}
<form action="" method="post">
  <p>Are you sure you want to delete "{{ object }}"?</p>
  <input type="submit" value="Delete">
</form>
```

template_name_suffix = '_delete'
The suffix to use when generating a template name from the model class

Helpers

flask_generic_views.sqlalchemy.**session**
A proxy to the current SQLAlchemy session provided by Flask-SQLAlchemy.

class flask_generic_views.sqlalchemy.**SingleObjectMixin**
Bases: *flask_generic_views.core.ContextMixin*

Provides the ability to retrieve an object based on the current HTTP request.

object
The the object used by the view.

model = None
The Model class used to retrieve the object used by this view.

This property is a shorthand, `model = Post` and `query = Post.query` are functionally identical.

query = None
The Query instance used to retrieve the object used by this view.

slug_field = 'slug'
The name of model field that contains the slug

context_object_name = None
The name of the context variable name to store the *object* in.

slug_view_arg = 'slug'
The name of the view keyword argument that contains the slug.

pk_view_arg = 'pk'
The name of the view keyword argument that contains the primary-key.

query_pk_and_slug = False
When True *get_object()* will filter the query by both primary-key and slug when available.

get_context_data(***kwargs*)
Extends the view context with *object*.

When *object* is set, an `object` variable containing *object* is added to the context.

A variable named with the result of *get_context_object_name()* containing *object* will be added to the context.

Parameters **kwargs** (*dict*) – context

Returns context

Return type dict

get_context_object_name()
Retrieve the context variable name that *object* will be stored under.

By default it will return *context_object_name*, falling back to a name based on the model from *query* or *model*, the model `BlogPost` would have the context object name `blog_post`.

Returns context object name

Return type str

get_model()
Retrieve the model used to retrieve the object used by this view.

By default returns the model associated with *query* when it's set, otherwise it will return *model*.

Returns model

Return type flask_sqlalchemy.Model

get_object()
Retrieve the object used by the view.

The `Query` object from *get_query()* will be used as a base query for the object.

When *pk_view_arg* exists in the current requests `view_args` it will be used to filter the query by primary-key.

When *slug_view_arg* exists in the current requests `view_args` and either no primary-key was found or *query_pk_and_slug* is `True` then it will be used to filter the query by *slug_field*.

Returns object

Return type flask_sqlalchemy.Model

Raises **werkzeug.exceptions.NotFound** – when no result found

get_query()
Retrieve the query used to retrieve the object used by this view.

By default returns *query* when it's set, otherwise it will return a query for *model*.

Returns query

Return type sqlalchemy.orm.query.Query

get_slug_field()
Retrive the name of model field that contains the slug.

By default it will return *slug_field*.

Returns slug field

Return type str

class flask_generic_views.sqlalchemy.**BaseDetailView**(***kwargs*)
Bases: *flask_generic_views.sqlalchemy.SingleObjectMixin*, *flask_generic_views.core.MethodView*

View class to retrieve an object.

get(***kwargs*)
Set `object` to the result of `get_object()` and create a response using the return value of `get_context_data()`.

Parameters **kwargs** (*dict*) – keyword arguments from url rule

Returns response

Return type werkzeug.wrappers.Response

class flask_generic_views.sqlalchemy.**SingleObjectTemplateResponseMixin**
Bases: *flask_generic_views.core.TemplateResponseMixin*

Creates `Response` instances with a rendered template based on the given context.

When no template names are provided, the class will try and generate one based on the model name.

get_template_list()
Retrives a list of template names to use for when rendering the template.

When no *template_name* is set then the following will be provided instead:

- the value of the `template_name_field` field on the model when availible.

- A template based on `template_name_suffix`, the model, and the current blueprint. The model `BlogArticle` in blueprint `blogging` would generate the template name `blogging/blog_article_detail.html`, no blueprint would generate `blog_article_detail.html`

Returns list of template names

Return type list

class `flask_generic_views.sqlalchemy.`**`MultipleObjectMixin`**
Bases: *`flask_generic_views.core.ContextMixin`*

Provides the ability to retrieve a list of objects based on the current HTTP request.

If `paginate_by` is specified, the object list will be paginated. You can specify the page number in the URL in one of two ways:

- Pass the page as a view argument in the url rule.

```
post_list = PostListView.as_view('post_list')

app.add_url_rule('/posts/page/<int:page>', view_func=post_list)
```

- Pass the page as a query-string argument in the request url.

```
/posts?page=5
```

When no page is provided it defaults to 1.

When *`error_out`* is set, a non numeric page number or empty page (other than the first page) will result in a `NotFound` exception.

`object_list`
The final `Query` instance used by the view.

`error_out` = False
Wether to raise a `NotFound` exception when no results are found.

`query` = None
The base `Query` instance used by this view.

`model` = None
The `Model` class used to retrieve the object used by this view.

This property is a shorthand, `model = Post` and `query = Post.query` are functionally identical.

`per_page` = None
The number of results to return per page, when None pagination will be disabled.

`context_object_name` = None
The name of the context variable name to store the *`object_list`* in.

`pagination_class` = flask_sqlalchemy.Pagination
The `Pagination` class to be returned by *`get_pagination()`*.

`page_arg` = 'page'
The name of the view / query-string keyword argument that contains the page number.

`order_by` = None
A `tuple` of criteria to pass to pass to the query `order_by()` method.

apply_pagination(*object_list*, *per_page*, *error_out*)
Retrieves a 3-item tuple containing (pagination, object_list, is_paginated).

The `pagination` from *`get_pagination()`*, The `object_list` paginated with page from *`get_page()`* and `per_page`, and wether there is more than one page will be returned.

When `error_out` is set then a `NotFound` exception will be raised when the page number is invalid, or refers to an empty page greater than 1.

Parameters

- **query** (*`flask_sqlalchemy.BaseQuery`*) – sqlalchemy query
- **per_page** (*`int`*) – items per page
- **error_out** (*`bool`*) – error out

Returns pagination instance, object list, is paginated

Return type tuple

Raises **werkzeug.exceptions.NotFound** – when page number is invalid

get_context_data(***kwargs*)
Extends the view context with `object`.

When the return value of *`get_per_page()`* is not `None`, then *`object_list`* will be paginated with *`apply_pagination()`* and the resulting `pagination`, `object_list` and `is_paginated` will be stored in the view context. Otherwise the result of executing *`object_list`* will be stored in `object_list`, `pagination` will be `None`, and `is_paginated` will be `False`.

A variable named with the result of *`get_context_object_name()`* containing `object_list` will be added to the context.

Parameters **kwargs** (*`dict`*) – context

Returns context

Return type dict

get_context_object_name()
Retrieve the context variable name that `object` will be stored under.

By default it will return *`context_object_name`*, falling back to a name based on the model from *`query`* or *`model`*, the model `BlogPost` would have the context object name `blog_post_list`.

Returns context object name

Return type str

get_error_out()
Retrive how invalid page numbers or empty pages are handled.

When `True` a `werkzeug.exceptions.NotFound` will be raised for invalid page numbers or empty pages greater than one.

When `False` invalid page numbers will default to 1 and empty pages will be rendered with an empty `object_list`.

By default returns *`error_out`*.

Returns error out

Return type bool

get_model()
Retrieve the model used to retrieve the object used by this view.

By default returns the model associated with *query* when it's set, otherwise it will return *model*.

Returns model

Return type flask_sqlalchemy.Model

get_order_by()
Retrieve a `tuple` of criteria to pass to pass to the query `order_by()` method.

Returns list of order by criteria

Return type list

get_page(*error_out*)
Retrieve the current page number.

The page is first checked for in the view keyword arguments, and then the query-string arguments using the key from *page_arg*.

When the value is not a an unsigned integer greater than zero and `error_out` is `True` then a `NotFound` exception will be raised. When `False` the page will default to 1.

Parameters **error_out** (*bool*) – raise error on invalid page number

Returns number of items per page

Return type int

get_pagination(*query*, *page*, *per_page*, *total*, *items*)

Parameters

- **query** (*flask_sqlalchemy.BaseQuery*) – sqlalchemy query
- **page** (*int*) – page number
- **per_page** (*int*) – items per page
- **total** (*int*) – total items
- **items** (*list*) – list of objects

Returns pagination instance

Return type flask_sqlalchemy.Pagination

get_per_page()
Retrieve the number of items to show per page.

By default returns *per_page*.

Returns number of items per page

Return type int

get_query()
Retrieve the query used to retrieve the object used by this view.

By default returns *query* when it's set, otherwise it will return a query for *model*.

Returns query

Return type flask_sqlalchemy.BaseQuery

class flask_generic_views.sqlalchemy.**BaseListView**(***kwargs*)
Bases: *flask_generic_views.sqlalchemy.MultipleObjectMixin*, *flask_generic_views.core.MethodView*

View class to retrieve a list of objects.

get(***kwargs*)
Set `object_list` to the result of `get_query()` and create a view from the context.

Parameters **kwargs** (*dict*) – keyword arguments from url rule

Returns response

Return type werkzeug.wrappers.Response

class flask_generic_views.sqlalchemy.**MultipleObjectTemplateResponseMixin**
Bases: *flask_generic_views.core.TemplateResponseMixin*

template_name_suffix = '_list'
The suffix to use when generating a template name from the model class

get_template_list()
Retrives a list of template names to use for when rendering the template.

When no *template_name* is set then the following will be provided instead:

- A template based on *template_name_suffix*, the model, and the current blueprint. The model `BlogArticle` in blueprint `blogging` would generate the template name `blogging/blog_article_list.html`, no blueprint would generate `blog_article_list.html`

Returns list of template names

Return type list

class flask_generic_views.sqlalchemy.**ModelFormMixin**
Bases: *flask_generic_views.core.FormMixin*, *flask_generic_views.sqlalchemy.SingleObjectMixin*

fields = None
A `tuple` of `str` mapping to the names of column attribute on the `model`, these will be added as form fields on the automatically generated form.

form_valid(*form*)
Creates or updates `object` from `model`, persists it to database, and redirects to *get_success_url()*.

Parameters **form** (*flask_wtf.Form*) – form instance

Returns response

Return type werkzeug.wrappers.Response

get_form_class()
Retrieve the form class to instantiate.

When `form_class` is not set, a form class will be automatically generated using `model` and *fields*.

Returns form class

Return type type

get_form_kwargs()
Extends the form keyword arguments with *obj* containing `object` when set.

Returns keyword arguments

Return type dict

get_success_url()
Retrive the URL to redirect to when the form is successfully validated.

By default returns `success_url` after being interpolated with the object attributes using `format()`. So `"/posts/{id}"` will be populated with `self.object.id`

Returns URL

Return type str

class flask_generic_views.sqlalchemy.**BaseCreateView**(***kwargs*)
Bases: *flask_generic_views.sqlalchemy.ModelFormMixin*, *flask_generic_views.core.ProcessFormView*

View class for creating an object.

class flask_generic_views.sqlalchemy.**BaseUpdateView**(***kwargs*)
Bases: *flask_generic_views.sqlalchemy.ModelFormMixin*, *flask_generic_views.core.ProcessFormView*

View class for updating an object.

get(***kwargs*)
Set `object` to the result of `get_object()` and create a response using the return value of `get_context_data()`.

Parameters **kwargs** (*dict*) – keyword arguments from url rule

Returns response

Return type werkzeug.wrappers.Response

post(***kwargs*)
Set `object` to the result of `get_object()` and construct and validates a form.

When the form is valid `form_valid()` is called, when the form is invalid `form_invalid()` is called.

Parameters **kwargs** (*dict*) – keyword arguments from url rule

Returns response

Return type werkzeug.wrappers.Response

class flask_generic_views.sqlalchemy.**DeletionMixin**
Bases: `object`

Handle the DELETE http method.

success_url = None
The URL to redirect to after deletion.

delete(***kwargs*)
Set `object` to the result of `get_object()`, delete the object from the database, and create a response using the return value of `get_context_data()`.

Parameters **kwargs** (*dict*) – keyword arguments from url rule

Returns response

Return type werkzeug.wrappers.Response

get_success_url()
Retrive the URL to redirect to when the form is successfully validated.

By default returns *success_url* after being interpolated with the object attributes using format(). So "/posts/{id}" will be populated with self.object.id

Returns URL

Return type str

post(***kwargs*)

Passes all keyword arguments to *delete()*.

Parameters **kwargs** (*dict*) – keyword arguments from url rule

Returns response

Return type werkzeug.wrappers.Response

class flask_generic_views.sqlalchemy.**BaseDeleteView**(***kwargs*)

Bases: *flask_generic_views.sqlalchemy.DeletionMixin*, *flask_generic_views.sqlalchemy.BaseDetailView*

View class for deleting an object.

CHAPTER 3

Additional Notes

Change Log

Here you can see the full list of changes.

Version 0.1.1

Released on 2016-11-19

- Fixing typos in tests and updating hypothesis
- Fix bug in MultipleObjectMixin.get_page
- Fix docs for sqlalchemy related classes on readthedocs

Version 0.1.0

Released on 2016-01-05

- First public preview release.

License

Flask-Generic-Views is licensed under the MIT license. Basically, you can do whatever you want as long as you include the original copyright and license notice in any copy of the software/source.

MIT License

Python Module Index

f

Index

L

M

O

P

Q

R

S

T

U

V