Flask-Generic-Views Documentation

Release 0.1.0

Daniel Knell

April 28, 2016

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

1.1 Installation

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

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

1.1.1 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]
```

1.2 Quick Start

1.2.1 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.

1.3 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

2.1 API

2.1.1 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.

2.1.2 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

Helpers

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

CHAPTER 3

Additional Notes

3.1 Change Log

Here you can see the full list of changes.

3.1.1 Version 0.1.0

Released on 2016-01-05

- First public preview release.

3.2 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.

3.2.1 MIT License

f

B

C

D

E

F

G

M

P

Q

R

S

T

U

V