Flask-LazyViews Documentation

Release 0.6

Igor Davydenko

July 03, 2015

Contents

Flask-LazyViews registers URL routes for Flask application or blueprint in lazy way. Ships with additional support of registering admin views, error handlers, custom static files and rendering Jinja2 templates.

- Based on original snippet from Flask documentation
- Works on Python 2.6, 2.7 and 3.3+
- BSD licensed
- Latest documentation on Read the Docs
- Source, issues and pull requests on GitHub

CHAPTER 1

Installation

Use pip to install Flask-LazyViews to your system or virtual environment:

```
$ pip install Flask-LazyViews
```

Otherwise you could download source dist from GitHub or PyPI and put `flask_lazyviews` directory somewhere to `$PYTHONPATH`, but this way is not recommended. Use pip for all good things.

CHAPTER 2

Necessity

The main purpose of registering lazy views instead of standard approach of adding routes by decorating view functions with `flask.Flask.route()` decorator is avoiding “Circular Import” errors while organizing code in big Flask project.

For example, when you have project with structure a like:

```
project/
+ app.py
+ models.py
+ settings.py
+ views.py
```

and want to use `@app.route` decorator you obviously need to import `app` instance in views module, but to be sure that these view functions registered as URL routes for your app you need make back import in app module. So you actually should make inner import if using application factories or just import views after global `app` instance already initialized and maybe configured. This looks not very Pythonic.

In case of using lazy views you don’t need import your views module when you instantiate your application, all you need to import *`LazyViews`* instance, init it and add URL routes where view function is a string with valid Python path. And yes adding view support all other route arguments as `endpoint`, `methods`, etc.

And yes, Flask-LazyViews supports `flask.Blueprint` instances and have extra features as registering error handlers, adding admin views, additional static routes and rendering Jinja2 templates without actual view function. Neat!

Usage

To get started all you need to instaniate *`LazyViews`* object after configuring application:

```
from flask import Flask
from flask_lazyviews import LazyViews

app = Flask(__name__)
views = LazyViews(app)
```

Or blueprint:

```
from flask import Blueprint

blueprint = Blueprint('name', __name__)
views = LazyViews(blueprint)
```

You can also pass the Flask application or blueprint object later, by calling *`init_app()`* or *`init_blueprint()`* respectfully:

```
views = LazyViews()

def create_app(name=None, **options):
    app = Flask(name or __name__)
    app.config.update(options)
    views.init_app(app, 'path.to.views')  # Full path to views module
    ...
    return app
```

Or:

```
def create_blueprint(name, **options):
    blueprint = Blueprint(name, __name__, **options)
    views.init_blueprint(blueprint, '.views')  # Rel path to views module
    ...
    return blueprint
```

Now you ready to go and start using Flask-LazyViews extension.

3.1 Adding URL routes

Main feature of Flask-LazyViews extension is adding lazy views (touche) to your Flask application or blueprint.

In most cases lazy view is just a string with full Python path to view function, like `app.views.view`, where:

- `app` is name of Python package with your Flask application/blueprint
- `views` is name of views module
- `view` is name of view function

In other adding lazy views are equal to registering routes with `flask.Flask.route()` method and it supports all its keyword arguments as `methods`, `endpoint`, etc.

To add lazy view to your application you need to call *`add()`* method:

```
views.add('/', 'app.views.index')
views.add('/comment/add', 'app.views.add_comment', methods=('GET', 'POST'))
views.add('/page/<int:page_id>', 'app.views.page')
```

To simplify things and avoid repeating base path to your views, like `app.views` above, you could use `import_prefix` while initializing *`LazyViews`*:

```
views = LazyViews(app, 'app.views')
```

After you don't need to repeat this prefix and could add views as:

```
views.add('/', 'index')
views.add('/comment/add', 'add_comment', methods=('GET', 'POST'))
views.add('/page/<int:page_id>', 'page')
```

3.2 Registering error handlers

Flask application and blueprint has ability to register error handlers to customize processing HTTP errors. In most cases this handlers are view functions which take `error` as first argument, so Flask-LazyViews allows you to add this error handlers with *`add_error()`* method:

```
views.add_error(404, 'views.not_found')
views.add_error(500, 'views.server_error')
views.add_error(AssertionError, 'views.server_error')
```

3.3 Registering app error handler for Blueprint

New in version 0.6.

In addition to registering error handlers for "own" URLs Flask blueprint has ability to register custom error handler for application. To do this in Flask-LazyViews, you need to pass `app=True` to *`add_error()`* method:

```
views.add_error(401, 'views.not_authorized', app=True)
views.add_error(ValueError, 'views.value_error', app=True)
```

3.4 Adding admin views

New in version 0.4.

Flask-Admin is one of the most popular choices for making CRUD admin panel for Flask application. Flask-LazyViews has support adding admin views in lazy way as done for plain routes. To do this you need to call *`add_admin()`* method:

```
views.add_admin('admin.AdminView',
                endpoint='app_admin',
                name='Custom Admin Page')
```

Note: Keyword arguments passed to *`add_admin()`* method would be transfer to instantiate admin view.

Important: Adding admin views only works for Flask application and when `flask.ext.admin.base.Admin` extension already initialized before calling *`add_admin()`* method.

3.5 Additional static routes

Changed in version 0.6.

Sometimes you need to create additional static route, for example to serve `favicon.ico` in root of your app. To do this you need to call *`add_static()`* method:

```
views.add_static('/favicon.ico', 'icons/favicon.ico', endpoint='favicon')
```

You also should pass `filename` to static handler in `defaults` dict (default approach prior to 0.6 version):

```
views.add_static('/favicon.ico',
                 defaults={'filename': 'icons/favicon.ico'},
                 endpoint='favicon')
```

3.6 Rendering Jinja2 templates without view functions

New in version 0.6.

Sometimes rendering template don't require any additional logic in view function, but you still need to define it. To avoid this you could render those templates directly with *`add_template()`* method:

```
views.add_template('/', 'index.html', endpoint='index')
```

You also can pass context to your templates, context could be a plain dict:

```
views.add_template('/',
                   'index.html',
                   context={'DUMMY_CONSTANT': True},
                   endpoint='index')
```

Or any callable which returns dict:

```
from flask import g

def settings_context():
    return {'settings': get_user_settings(g.user)}

views.add_template('/settings',
                   'settings.html',
                   context=settings_context,
                   endpoint='settings')
```

CHAPTER 4

Example

Flask-LazyViews ships with simple test application which shows basic principles for using extension.

In case you want to run this application in your local environment you need to bootstrap virtual environment for it and run server as:

```
$ cd /path/to/Flask-LazyViews
$ make -C testapp/ bootstrap
$ make -C testapp/ server
```

When done, point your browser to `http://127.0.0.1:8303/` to see results.

Note: To bootstrap project bootstrapper should be installed to your system as well as GNU Make.

CHAPTER 5

API

class `flask_lazyviews.`**`LazyViews`**(*instance=None*, *import_prefix=None*)

Main instance for adding *lazy* views to Flask application or blueprint.

`__init__`(*instance=None*, *import_prefix=None*)

Initialize *`LazyViews`* instance.

Basically it requires `app` or `blueprint` instance as first argument, but you could leave it empty and initialize it later with manually call *`init_app()`* method. It could be helpful, if you want to configure *`LazyViews`* instance somewhere outside your `app.py` or for multiple applications.

`add`(*url_rule*, *mixed*, ***options*)

Add URL rule to Flask application or blueprint.

`mixed` could be a real callable function, or a string Python path to callable view function. If `mixed` is a string, it would be wrapped into *`LazyView`* instance.

`add_admin`(*mixed*, **args*, ***kwargs*)

Add admin view if Flask-Admin extension added to application.

Important: This method only works for Flask applications, not blueprints.

`add_error`(*code_or_exception*, *mixed*, *app=False*)

Add error handler to Flask application or blueprint.

When passing `app=True` tries to register global app error handler for blueprint.

`add_static`(*url_rule*, *filename=None*, ***options*)

Add URL rule for serving static files to Flask app or blueprint.

`add_template`(*url_rule*, *template_name*, ***options*)

Render template name with context for given URL rule.

Context should be a plain dict or callable. If callable its result would be passed to `flask.render_template()` function.

`build_import_name`(*import_name*)

Prepend import prefix to import name if it earlier defined by user.

`get_view`(*mixed*)

If `mixed` value is callable it's our view, else wrap it with *`flask_lazyviews.utils.LazyView`* instance.

`init_app`(*app*, *import_prefix=None*)

Configure *`LazyViews`* instance, store `app` or `blueprint` instance and import prefix if any.

`init_blueprint`(*blueprint*, *import_prefix=None*)

Alias for init app function, cause basically there are no important differences between Flask app and blueprint if we only need to add URL rule.

class `flask_lazyviews.utils.`**`LazyView`**(*name*, **args*, ***kwargs*)

Import view function only when necessary.

`__call__`(**args*, ***kwargs*)

Make real call to the view.

`__eq__`(*other*)

Check that two lazy view instances have equal import names.

`__getattribute__`(*name*)

Proxify documentation attribute from original view if it could be imported.

`__init__`(*name*, **args*, ***kwargs*)

Initialize `LazyView` instance for view that would be imported from `name` path.

`__ne__`(*other*)

Check that two lazy view instances have not equal import names.

`__repr__`()

Show custom repr message if view function exists.

`view`

Import view from string and cache it to current class instance.

CHAPTER 6

Changelog

6.1 0.6 (2014-08-14)

- Render Jinja2 templates for given URL rule via `add_template()` method.
- Register global app error handler from Blueprint by passing `app=True` to `add_error()` method.
- Keep `import_prefix` in `LazyViews` instance.
- Easify registering additional static routes by auto-adding `filename` to `defaults` dict.
- Fixes #4. Fix registering multiple routes to same lazy view.
- Move documentation from README to Read the Docs

6.2 0.5.1 (2014-01-31)

- Fixes #3. Make `LazyView` proxy class lazy again. Fix circullar imports and working outside application context.

6.3 0.5 (2013-12-27)

- Python 3 support (only for Flask 0.10+).
- Flask 0.10+ support.
- Fixes #2. Access view function documentation and repr while loading views via strings.

6.4 0.4 (2012-10-28)

- Add support of adding admin views to Flask applications via `add_admin()` method.
- Configure Travis CI support.

6.5 0.3 (2012-10-04)

- Implement `add_error()` shortcut method for adding custom error handling for Flask application or blueprint.

6.6 0.2 (2012-09-17)

- Implement `init_app()` and `init_blueprint()` methods for configuring `LazyViews` instance outside main application module or for multiple applications.
- Add `add_static()` shortcut method for adding custom URL rules for serving static files.
- Add ability to register real view functions with `LazyViews` instance.

6.7 0.1 (2012-04-03)

- Initial release.

Python Module Index

Index