Flask-Pushrod Documentation

Release 0.1-dev

Nullable

October 19, 2012

CONTENTS

Contents:

CHAPTER

ONE

GETTING STARTED

Pushrod adds an extra layer between the Template and the View layers (from the Model-Template-View/MTV pattern, more commonly known as Model-View-Controller/MVC outside of the Python world), the renderer.

1.1 Hello, World!

As an example, we're going to adapt this simple Flask app (slightly adapted from the Flask hello world example) to Pushrod:

```
from flask import Flask, render_template

import random


app = Flask(__name__)


@app.route("/")
def hello():
      return render_template("hello.html", greeting=random.choice(("Hi", "Heya")))

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

Where `hello.html` might look like this:

```
<html>
     <head>
     </head>
     <body>
      Hello, {{ greeting }}.
     </body>
</html>
```

1.1.1 Adapting It To Pushrod

With Pushrod you don't do the rendering yourself in the View. Instead, you return a context which is passed to a renderer which is inferred from the request. This means that the same code-base can power both the UI and the API with minimal boilerplate. By default Pushrod is set up with a `Jinja2` renderer (for HTML) and a `JSON` renderer.

The first step is to add a Pushrod resolver and decorate the view with the `pushrod_view()` decorator, which will pass it through the Pushrod rendering pipeline. The code will still work because strings and `Response` objects are passed through unrendered. The code should now look like this:

```
from flask import Flask, render_template
from flask.ext.pushrod import Pushrod, pushrod_view

import random


app = Flask(__name__)
Pushrod(app)


@app.route("/")
@pushrod_view()
def hello():
      return render_template("hello.html", greeting=random.choice(("Hi", "Heya")))

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

Warning: Remember to add the `pushrod_view()` decorator closer to the function definition than the `route()` decorator.

While this works and all, we get absolutely no benefit from using Pushrod right now. So let's let Pushrod handle the rendering:

```
from flask import Flask, render_template
from flask.ext.pushrod import Pushrod, pushrod_view

import random


app = Flask(__name__)
Pushrod(app)


@app.route("/")
@pushrod_view(jinja_template="hello.html")
def hello():
      return {
              'greeting': random.choice(("Hi", "Heya"))
      }

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

That's it. While it might seem a bit longer than the regular non-Pushrod code, you now get JSON rendering (and any other renderers you decide to enable) for free!

1.1.2 Making Your Own Renderer

Sometimes the available Pushrod renderers might not meet your requirements. Fortunately, making your own renderer is very easy. Let's say you want a renderer that passes the response through `repr()`, it would look like this:

```
from flask.ext.pushrod.renderers import renderer

from repr import repr


@renderer(name='repr', mime_type='text/plain')
def repr_renderer(unrendered, **kwargs):
  return unrendered.rendered(
      repr(unrendered.response),
      'text/plain')
```

Warning: Always take a `**kwargs` in your renderer, since other renderers might take arguments that don't matter to your renderer.

Warning: Of course, you should never use `repr()` like this in production code, it is just an example to demostrate the syntax without having to go through the regular boilerplate code of creating the response ourselves.

And you would register it to your `Pushrod` instance using `register_renderer()`.

Note: Functions not decorated using `renderer()` may not be registered as renderers.

1.2 Formatting Actually Useful Data

Note: This guide assumes that you're familiar with Flask and Flask-SQLAlchemy.

While the shown examples might look neat for simple data, it can quickly get out of hand. For example, let's take a simple blog application (*let's call it pushrodr just to be original*):

```
from flask import Flask
from flask.ext.pushrod import Pushrod, pushrod_view
from flask.ext.sqlalchemy import SQLAlchemy

from sqlalchemy.sql.functions import now

app = Flask(__name__)
Pushrod(app)
db = SQLAlchemy(app)


class Author(db.Model):
    __tablename__ = "authors"

    id = db.Column(db.Integer, primary_key=True, nullable=False)
    name = db.Column(db.String(80), unique=True, nullable=False)
    description = db.Column(db.Text(), nullable=False)


class Post(db.Model):
    __tablename__ = "posts"

    id = db.Column(db.Integer, primary_key=True, nullable=False)
    timestamp = db.Column(db.DateTime, nullable=False, default=now())
```

```
    title = db.Column(db.String(255), unique=True, nullable=False)
    content = db.Column(db.Text, nullable=False)

    author_id = db.Column(db.Integer, db.ForeignKey(Author.id), nullable=False)
    author = db.relationship(Author, backref='posts')


class Comment(db.Model):
    __tablename__ = "comments"

    id = db.Column(db.Integer, primary_key=True, nullable=False)
    author = db.Column(db.String(80), nullable=False)
    timestamp = db.Column(db.DateTime, nullable=False, default=now())
    content = db.Column(db.Text, nullable=False)

    post_id = db.Column(db.Integer, db.ForeignKey(Post.id), nullable=False)
    post = db.relationship(Post, backref='comments')


@app.route("/")
@app.route("/<int:page>")
@pushrod_view()
def list_posts(page=1):
    posts = Post.query.paginate(page)

    return {
        'page': page,
        'pages': posts.pages,
        'total': posts.total,
        'items': [{
            'id': post.id,
            'title': post.title,
            'timestamp': unicode(post.timestamp),
            'content': post.content,
            'author': {
                'name': post.author.name,
                'description': post.author.description,
            },
        } for post in posts.items]
    }


@app.route("/posts/<int:id>")
@pushrod_view()
def blog_post(id):
    post = Post.query.get_or_404(id)

    return {
        'item': {
            'id': post.id,
            'title': post.title,
            'timestamp': unicode(post.timestamp),
            'content': post.content,

            'author': {
                'name': post.author.name,
                'description': post.author.description,
            },
```

```
            'comments': [{
                'author': comment.author,
                'timestamp': unicode(comment.timestamp),
                'content': comment.content,
            } for comment in post.comments]
        }
    }


if __name__ == '__main__':  # pragma: no cover
    app.run()
```

1.2.1 Enter Normalizers

As you can see that quickly starts looking redundant, and *stupid*. It's also going to cause problems if you're going to do any form validation using, say, Flask-WTF, or anything else that, while working perfectly for an API too, has special helpers for the GUI rendering. To help with these cases Flask-Pushrod has something called "normalizers". Normalizers are callables that take two arguments (the object and the `Pushrod` instance) and prepare the data for serialization (see `normalize()`). If a normalizer returns `NotImplemented` then the value is passed through to the next normalizer.

> **Warning:** Renderers can opt out of the normalization process, so that they are passed the un-normalized data (an example is `the Jinja2 renderer`). Because of this, the normalizer shouldn't add or rename data.

An example normalizer could look like this:

```
def my_normalizer(x, pushrod):
    if x:
        return NotImplemented
    else:
        return 0
```

This would return 0 for all "falsy" values, but let all other values normalize as usual. It could then be registered like this:

```
app = Flask(__name__)
pushrod = Pushrod(app)
pushrod.normalizers[object] = my_normalizer
```

Note: There is also `normalizer_overrides`. The difference is that `normalizers` is a `dict` of callables that should be used for the "standard" case, while `normalizer_overrides` is a `dict` of `lists` of callables should be used when you need to override the behaviour for a subset of cases.

Note: Both `normalizer_overrides` and `normalizers` are resolved in the regular "MRO" (method resolution order). Normalization is resolved in the same order as method calls.

1.2.2 Throwing Normalizers At The Problem

Now that you should have a basic grasp on normalizers, let's try to use some! Below is how the previous code would look using normalizers:

```
from flask import Flask, g
from flask.ext.pushrod import Pushrod, pushrod_view
from flask.ext.sqlalchemy import SQLAlchemy, Pagination

from sqlalchemy.sql.functions import now

app = Flask(__name__)
pushrod = Pushrod(app)
db = SQLAlchemy(app)


class Author(db.Model):
    __tablename__ = "authors"

    id = db.Column(db.Integer, primary_key=True, nullable=False)
    name = db.Column(db.String(80), unique=True, nullable=False)
    description = db.Column(db.Text(), nullable=False)


class Post(db.Model):
    __tablename__ = "posts"

    id = db.Column(db.Integer, primary_key=True, nullable=False)
    timestamp = db.Column(db.DateTime, nullable=False, default=now())
    title = db.Column(db.String(255), unique=True, nullable=False)
    content = db.Column(db.Text, nullable=False)

    author_id = db.Column(db.Integer, db.ForeignKey(Author.id), nullable=False)
    author = db.relationship(Author, backref='posts')


class Comment(db.Model):
    __tablename__ = "comments"

    id = db.Column(db.Integer, primary_key=True, nullable=False)
    author = db.Column(db.String(80), nullable=False)
    timestamp = db.Column(db.DateTime, nullable=False, default=now())
    content = db.Column(db.Text, nullable=False)

    post_id = db.Column(db.Integer, db.ForeignKey(Post.id), nullable=False)
    post = db.relationship(Post, backref='comments')


def normalize_author(x, pushrod):
    return pushrod.normalize({
        'name': x.name,
        'description': x.description
    })


def normalize_post(x, pushrod):
    data = {
        'id': x.id,
        'title': x.title,
        'timestamp': x.timestamp,
        'content': x.content,
        'author': x.author,
    }
```

```
    if not getattr(g, 'list_view', False):
        data['comments'] = x.comments

    return pushrod.normalize(data)


def normalize_comment(x, pushrod):
    return pushrod.normalize({
        'author': x.author,
        'timestamp': x.timestamp,
        'content': x.content,
    })


def normalize_pagination(x, pushrod):
    return pushrod.normalize({
        'page': x.page,
        'pages': x.pages,
        'total': x.total,
        'items': x.items,
    })

pushrod.normalizers.update({
    Author: normalize_author,
    Post: normalize_post,
    Comment: normalize_comment,
    Pagination: normalize_pagination,
})


@app.route("/")
@app.route("/<int:page>")
@pushrod_view()
def list_posts(page=1):
    g.list_view = True
    return Post.query.paginate(page)


@app.route("/posts/<int:id>")
@pushrod_view()
def blog_post(id):
    post = Post.query.get_or_404(id)
    return {'item': post}


if __name__ == '__main__':  # pragma: no cover
    app.run()
```

1.2.3 Moving The Normalizers Into The Classes

In the spirit of converters like `__bool__()`, the normalizers can also be defined inline in the classes, like this:

```
from flask import Flask, g
from flask.ext.pushrod import Pushrod, pushrod_view
from flask.ext.sqlalchemy import SQLAlchemy, Pagination
```

```
from sqlalchemy.sql.functions import now

app = Flask(__name__)
pushrod = Pushrod(app)
db = SQLAlchemy(app)


class Author(db.Model):
    __tablename__ = "authors"

    id = db.Column(db.Integer, primary_key=True, nullable=False)
    name = db.Column(db.String(80), unique=True, nullable=False)
    description = db.Column(db.Text(), nullable=False)

    def __pushrod_normalize__(self, pushrod):
        return pushrod.normalize({
            'name': self.name,
            'description': self.description
        })


class Post(db.Model):
    __tablename__ = "posts"

    id = db.Column(db.Integer, primary_key=True, nullable=False)
    timestamp = db.Column(db.DateTime, nullable=False, default=now())
    title = db.Column(db.String(255), unique=True, nullable=False)
    content = db.Column(db.Text, nullable=False)

    author_id = db.Column(db.Integer, db.ForeignKey(Author.id), nullable=False)
    author = db.relationship(Author, backref='posts')

    def __pushrod_normalize__(self, pushrod):
        data = {
            'id': self.id,
            'title': self.title,
            'timestamp': self.timestamp,
            'content': self.content,
            'author': self.author,
        }

        if not getattr(g, 'list_view', False):
            data['comments'] = self.comments

        return pushrod.normalize(data)


class Comment(db.Model):
    __tablename__ = "comments"

    id = db.Column(db.Integer, primary_key=True, nullable=False)
    author = db.Column(db.String(80), nullable=False)
    timestamp = db.Column(db.DateTime, nullable=False, default=now())
    content = db.Column(db.Text, nullable=False)

    post_id = db.Column(db.Integer, db.ForeignKey(Post.id), nullable=False)
    post = db.relationship(Post, backref='comments')
```

```
    def __pushrod_normalize__(self, pushrod):
        return pushrod.normalize({
            'author': self.author,
            'timestamp': self.timestamp,
            'content': self.content,
        })


def normalize_pagination(x, pushrod):
    return pushrod.normalize({
        'page': x.page,
        'pages': x.pages,
        'total': x.total,
        'items': x.items,
    })

pushrod.normalizers[Pagination] = normalize_pagination


@app.route("/")
@app.route("/<int:page>")
@pushrod_view()
def list_posts(page=1):
    g.list_view = True
    return Post.query.paginate(page)


@app.route("/posts/<int:id>")
@pushrod_view()
def blog_post(id):
    post = Post.query.get_or_404(id)
    return {'item': post}


if __name__ == '__main__':  # pragma: no cover
    app.run()
```

1.2.4 Simplifying The Normalizers Even Further

While it's much better than the original, as you can see the normalizers are all of the kind `{'y': x.y}`. However, the inline normalizer syntax, also has a shortcut for defining fields to include, like this:

```
from flask import Flask, g
from flask.ext.pushrod import Pushrod, pushrod_view
from flask.ext.sqlalchemy import SQLAlchemy, Pagination

from sqlalchemy.sql.functions import now

app = Flask(__name__)
pushrod = Pushrod(app)
db = SQLAlchemy(app)


class Author(db.Model):
    __tablename__ = "authors"
    __pushrod_fields__ = ("name", "description")
```

```
    id = db.Column(db.Integer, primary_key=True, nullable=False)
    name = db.Column(db.String(80), unique=True, nullable=False)
    description = db.Column(db.Text(), nullable=False)


class Post(db.Model):
    __tablename__ = "posts"

    id = db.Column(db.Integer, primary_key=True, nullable=False)
    timestamp = db.Column(db.DateTime, nullable=False, default=now())
    title = db.Column(db.String(255), unique=True, nullable=False)
    content = db.Column(db.Text, nullable=False)

    author_id = db.Column(db.Integer, db.ForeignKey(Author.id), nullable=False)
    author = db.relationship(Author, backref='posts')

    def __pushrod_fields__(self):
        fields = ["id", "timestamp", "title", "content", "author"]

        if not getattr(g, 'list_view', False):
            fields.append("comments")

        return fields


class Comment(db.Model):
    __tablename__ = "comments"
    __pushrod_fields__ = ("author", "timestamp", "content")

    id = db.Column(db.Integer, primary_key=True, nullable=False)
    author = db.Column(db.String(80), nullable=False)
    timestamp = db.Column(db.DateTime, nullable=False, default=now())
    content = db.Column(db.Text, nullable=False)

    post_id = db.Column(db.Integer, db.ForeignKey(Post.id), nullable=False)
    post = db.relationship(Post, backref='comments')


def normalize_pagination(x, pushrod):
    return pushrod.normalize({
        'page': x.page,
        'pages': x.pages,
        'total': x.total,
        'items': x.items,
    })

pushrod.normalizers[Pagination] = normalize_pagination


@app.route("/")
@app.route("/<int:page>")
@pushrod_view()
def list_posts(page=1):
    g.list_view = True
    return Post.query.paginate(page)


@app.route("/posts/<int:id>")
```

```
@pushrod_view()
def blog_post(id):
    post = Post.query.get_or_404(id)
    return {'item': post}


if __name__ == '__main__':  # pragma: no cover
    app.run()
```

Note: There is also another shortcut for inline definition, for delegation. It's used like this, and it simply uses that field instead of itself (in this case, `x.that_field`:

```
__pushrod_field__ = "that_field"
```

CHAPTER

TWO

API

2.1 Resolvers

class flask.ext.pushrod.**Pushrod**(*app=None*, *renderers=('json', 'jinja2')*, *default_renderer='html'*)
The main resolver class for Pushrod.

Parameters **renderers** – A tuple of renderers that are registered immediately (can also be strings, which are currently expanded to flask.ext.pushrod.renderers.%s_renderer)

app = None
The current app, only set from the constructor, not if using `init_app()`.

format_arg_name = 'format'
The query string argument checked for an explicit renderer (to override header-based content type negotiation).

Note: This is set on the class level, not the instance level.

---

get_renderers_for_request(*request=None*)
Inspects a Flask `Request` for hints regarding what renderer to use.

Parameters **request** – The request to be inspected (defaults to `flask.request`)

Returns List of matching renderers, in order of user preference

init_app(*app*)
Registers the Pushrod resolver with the Flask app (can also be done by passing the app to the constructor).

logger
Gets the logger to use, mainly for internal use.

The current resolution order looks as follows:

- `self.app`
- `flask.current_app`
- `logging`

mime_type_renderers = None
The renderers keyed by MIME type.

named_renderers = None
The renderers keyed by output format name (such as html).

normalize(*obj*)
Runs an object through the normalizer mechanism, with the goal of producing a value consisting only of "native types" (`unicode`, `int`, `long`, `float`, `dict`, `list`, etc).

The resolution order looks like this:

- Loop through `self.normalizer_overrides[type(obj)]` (taking parent classes into account), should be a callable taking (obj, pushrod), falls through on `NotImplemented`
- `self.normalizers[type(obj)]` (taking parent classes into account), should be a callable taking (obj, pushrod), falls through on `NotImplemented`

See *Bundled Normalizers* for all default normalizers.

Parameters **obj** – The object to normalize.

normalizer_overrides = None
Hooks for overriding a class' normalizer, even if they explicitly define one.

All items should be lists of callables. All values default to an empty list.

normalizers = None
Hooks for providing a class with a fallback normalizer, which is called only if it doesn't define one. All items should be callables.

register_renderer(*renderer*, *default=False*)
Registers a renderer with the Pushrod resolver (can also be done by passing the renderer to the constructor).

render_response(*response*, *renderer=None*, *renderer_kwargs=None*)
Renders an unrendered response (a bare value, a (response, status, headers)-`tuple`, or an `UnrenderedResponse` object).

Throws RendererNotFound If a usable renderer could not be found (explicit renderer argument points to an invalid render, or no acceptable mime types can be used as targets and there is no default renderer)

Parameters

- **response** – The response to render
- **renderer** – The renderer(s) to use (defaults to using `get_renderer_for_request()`)
- **renderer_kwargs** – Any extra arguments to pass to the renderer

---

Note: For convenience, a bare string (`unicode`, `str`, or any other `basestring` derivative), or a derivative of `werkzeug.wrappers.BaseResponse` (such as `flask.Response`) is passed through unchanged.

Note: A renderer may mark itself as unable to render a specific response by returning `None`, in which case the next possible renderer is attempted.

2.2 Views

`flask.ext.pushrod`.**pushrod_view**(***renderer_kwargs*)
Decorator that wraps view functions and renders their responses through `flask.ext.pushrod.Pushrod.render_response()`.

Note: Views should only return `dicts` or a type that `normalizes` down to `dicts`.

Parameters **renderer_kwargs** – Any extra arguments to pass to the renderer

2.3 Renderers

`flask.ext.pushrod.renderers.`**`renderer`**(*name=None*, *mime_type=None*, *normalize=True*)

Flags a function as a Pushrod renderer.

Note: Before it is recognized by `flask.ext.pushrod.Pushrod.get_renderers_for_request()` (and, by extension, `render_response()`) it must be registered to the app's `Pushrod` instance (using `register_renderer()`, or passed as part of the `renderers` argument to the `Pushrod` constructor).

Parameters

- **name** – A `basestring` or a tuple of basestrings to match against when explicitly requested in the query string
- **mime_type** – A `basestring` or a tuple of basestrings to match against against when using HTTP content negotiation
- **normalize** – If True then the unrendered response will be passed through `flask.ext.pushrod.Pushrod.normalize()`

class `flask.ext.pushrod.renderers.`**`UnrenderedResponse`**(*response=None*, *status=None*, *headers=None*)

Holds basic response data from the view function until it is processed by the renderer.

`rendered`(*rendered_response*, *mime_type*)

Constructs a `rendered_class` (`flask.Response` by default) based on the response parameters.

`rendered_class`

The class to construct with the rendered response, defaults to `flask.Response`.

alias of `Response`

2.3.1 Bundled Renderers

`flask.ext.pushrod.renderers.`**`json_renderer`**(*unrendered*, ***kwargs*)

Renderers a response using `json.dumps()`.

Renderer MIME type triggers

- application/json

Renderer name triggers

- json

`flask.ext.pushrod.renderers.`**`jinja2_renderer`**(*unrendered*, ***kwargs*)

Renderers a response using `flask.render_template()`.

Renderer MIME type triggers

- text/html

Renderer name triggers

- html

2.4 Bundled Normalizers

Flask-Pushrod ships with a few normalizers by default. For more info, see `normalize()`.

Flask-Pushrod's built-in normalizers are generally named with the scheme `normalize_type`.

Note: Normalizers also apply to subclasses, unless the subclass defines another normalizer.

`flask.ext.pushrod.normalizers.`**`normalize_basestring`**(*x*, *pushrod*)

Takes

- `basestring`
- `datetime`
- `time`
- `date`

Returns `unicode`

`flask.ext.pushrod.normalizers.`**`normalize_iterable`**(*x*, *pushrod*)

Takes

- `list`
- `tuple`
- generator (see **PEP 255**, **PEP 342**, and **PEP 289**)

Returns `list` with all values `normalized`

`flask.ext.pushrod.normalizers.`**`normalize_dict`**(*x*, *pushrod*)

Takes `dict`

Returns `dict` with all keys converted to `unicode` and then `normalized` and all values `normalized`

`flask.ext.pushrod.normalizers.`**`normalize_int`**(*x*, *pushrod*)

Takes

- `int`
- `long`

Returns `int` or `long`

`flask.ext.pushrod.normalizers.`**`normalize_float`**(*x*, *pushrod*)

Takes `float`

Returns `float`

`flask.ext.pushrod.normalizers.`**`normalize_bool`**(*x*, *pushrod*)

Takes `bool`

Returns `bool`

flask.ext.pushrod.normalizers.**normalize_none**(*x*, *pushrod*)

Takes None

Returns None

flask.ext.pushrod.normalizers.**normalize_object**(*x*, *pushrod*)

Delegates normalization to the object itself, looking for the following attributes/methods (in this order):

- __pushrod_normalize__ - Essentially treated as if a normalizer was explicitly registered
- __pushrod_fields__ - A list of names fields, which is essentially treated like `{k: getattr(x, k) for k in x.__pushrod_fields__}`
- __pushrod_field__ - A name of a single field, x is then substituted for what is (simplified) `getattr(x, x.__pushrod_field)`

Note: __pushrod_fields__ and __pushrod_field__ can be either a callable or an attribute, while __pushrod_normalize__ must be a callable.

Takes object

2.5 Exceptions

exception flask.ext.pushrod.renderers.**RendererNotFound**

Thrown when no acceptable renderer can be found, see flask.ext.pushrod.Pushrod.get_renderers_for_reques

---

Note: This class inherits from werkzeug.exceptions.NotAcceptable, so it's automatically converted to `406 Not Acceptable` by Werkzeug if not explicitly handled (which is usually the intended behaviour).

CHAPTER

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

f