bottle-streamline Documentation

Release 1.0.post3

Outernet Inc

September 01, 2016

Contents

This project is a collection of classes for writing complex, modullar, and/or reusable route handlers using bottle web framework.

Even though 'complex route handlers' may seem like an oximoron, there may be situations where support for multiple parameters are needed or where a simple branching in the route handler code may completely change the logic of your route handler. If you find yourself running into this type of situation and struggle to organize the code, you may find bottle-streamline is a good fit for your project.

CHAPTER 1

Source code

Source code is available on GitHub.

CHAPTER 2

License

bottle-streamline is released under BSD license. See the `LICENSE` file that comes with the source code.

CHAPTER 3

Reporting issues

You can report issues on GitHub using the bottle-streamline issue tracker.

CHAPTER 4

Documentation

4.1 Introduction to class-based route handlers

Class-based route handlers (CBRH) can help you make your code more modular, deal with complexity, and isolate pieces of code for better testability. In order to take full advantage of the CBRHs, however, you will need to to have good undestanding of Python OOP, and especially topics like inheritance and mixins. It also helps to know a bit more about how bottle deals with handlers in general, though we will disucss that briefly here.

4.1.1 Bottle route handlers behind the scenes

Conventional bottle route handlers are written as functions. There are also many decorators that you can use to modify the way these functions work (e.g., a `@view()` decorator that causes the output of your function to be rendered using a template).

Any parameters that are found in the path pattern that is associated with a route handler will be mapped to the function's parameters. When the request is made to the path, handler will receive parsed out data as arguments.

The return value of a route handler function can be:

- a string
- an iterable of strings
- `bottle.HTTPResponse` object
- `bottle.HTTPError` objecta string
- file-like object (`StringIO`, file object, et al)

Note: For all practical purposes, special exceptions in the form of `HTTPResponse` and `HTTPError` objects (e.g., raised by `bottle.abort()`) can be considered return values.

Bottle contains a function that looks at the return value, and then tries to determine the type. It does different things depending on what the value is, and converts it to a form that is compatible with the underlying WSGI protocol (it expects an iterable). This function is `bottle.Bottle._cast()`. Going through all the cases it handles would be too long, so we will only focus on the iterables which are relevant to bottle-streamline.

If the return value of a route handler is an iterable, then `_cast()` attempts to fetch the first object. Failing that, it returns an empty string. If the first object is found, it checks its type. It only allows two kinds of objects at this stage:

- `HTTPResponse`
- string (either `unicode` or `bytes`)

If the first object is an `HTTPResponse` object, `_cast()` is called on it.

4.1.2 How CBRH fit into bottle route handling

When a CBRH is invoked, it is basically instantiated (its `__init__` method is invoked). Unlike handler functions, CBRHs don't do any actual work when they are invoked/instantiated. Instead, they implement the container interface of the iterator types and presents itself as an iterator. The entire CBRH object is, therefore, acts as a http response.

Note: CBRH classes are *not* subclasses of `HTTPResponse`.

The main implication of CBRH being an iterator container is that the response body (discussed in later chapters) can only be a string or `HTTPResponse` object. In your handler code, you cannot expect to return any other kind of object. If you need to return objects other than strings you need to either raise or return `HTTPResponse` objects (these objects double as exception classes so they can be raised) and pass the object you want to return as body. Here is an example:

```
class MyRoute(RouteBase):
    def get(self):
        data = open('somefile.ext', 'r')
        return self.HTTPResponse(data)
```

The following chapters will go into the details of how to use CBRH classes in different scenarios.

4.2 The basics

The basic workflow when using bottle-streamline is to import a class that does exactly what you need, or almost what you need, and subclass it. The most basic of the classes is the *`streamline.base.RouteBase`*.

Although you rarely need to use the *`RouteBase`* class directly (unless you are trying to build your own reusable base class that does things a bit diferently), it is prefect for demonstrating the lower-level functionality of other classes. It is the base class after all.

Here is an example of a class that returns a simple string response:

```
from streamline import RouteBase


class MyRoute(RouteBase):
    def get(self, name):
        return 'Hello, {}!'.format(name)


MyRoute.route('/hello/:name')
```

You can map the handler to a path by invoking the *`route()`* method. This method takes a path as first argument, optional route name as second argument, and an optional application object as third. Under the hood, this method will invoke the `bottle.Bottle.route()` method to set up routing.

The `MyRoute` subclass has a single method, `get()` which (you've guessed correctly!) handles the HTTP GET method. The method takes a single argument, `name` which comes from the placeholder pattern in the path.

4.2.1 Named routes

Named routes can be useful if you want to change the paths later. Instead of using hard-coded paths, bottle allows you to construct the paths using a route name and parameters. This is facilitated by the (somewhat underdocumented) `get_url()` method.

This method takes a route name as its first argument, and route parameters as any number of additional keyword arguments.

All streamline CBRH are named by default, even if you don't explicitly specify the name when invoking the *`route()`* method. The default name is calculated by using the name of the module in which your subclass resides, and the name of the class itself, which is decamelized. For instance, if your subclass is `users.AccountList`, it will be named `users:account_list`. Keeping your route handlers neatly organized and named will help you take advantage of this naming convention.

If you are not happy with the default names, you can supply an alternative name:

```
MyRoute.route('/hello/:name', name='hello')
```

You can also overload the *`get_generic_name()`* method:

```
class MyRoute(RouteBase):
    ....
    name = 'hello'

    @classmethod
    def get_generic_name(cls):
        return cls.name
```

The above example allows you to specify the route name as a `name` property.

4.2.2 Handling different HTTP methods

In a single route, you can have any number of methods that match HTTP verbs, and the route handler will be valid for those verbs. These methods can be:

- `get()`
- `post()`
- `put()`
- `patch()`
- `delete()`

A handler will only be registered for the HTTP methods it supports, and result in a HTTP 405 response for missing verbs.

You can have multiple route handlers with different verbs on the same path (if, for example, you wish to have different handlers for different verbs).

4.2.3 Including and excluding plugins

If you are using the bottle plugins, you can include additional plugins or skip the default ones by using `include_plugins` and `exclude_plugins` attributes. Both are lists of plugin names.

Here is an example:

```
class MyRoute(RouteBase):
    include_plugins = ['auth', 'session']
    exclude_plugins = ['static']
    ....
```

4.2.4 Route configuration

Normally, when invoking the `bottle.route()` function (or using it as a decorator), you can pass additional keyword arguments which become part of what is known as *route configuration*. The same is possible with the `route()` method.

4.2.5 Convenience properties and methods

The `RouteBase` class has several properties and methods that are added to its namespace for your convenience. These are:

- `bottle`: the `bottle` module
- `request`: `bottle.request` object
- `response`: `bottle.response` object
- `abort()`: `bottle.abort()` function
- `redirect()`: `bottle.redirect()` function
- `HTTPResponse`: `bottle.HTTPResponse` class

After initialization, two more properties will be available in the instances:

- `app`: application object that is tied to the request
- `config`: application's configuration

4.3 Rendering templates

Rendering templates is one of the most common tasks in most web applications. bottle-streamline provides two classes that can be used for this purpose.

4.3.1 Simple template rendering

The simple variant is the `streamline.template.TemplateRoute`, which renders a single template for all methods.

Let's take a look at example code:

```
from streamline import TemplateRoute


class MyRoute(TemplateRoute):
    template_name = 'accounts/profile'

    def get(self, uid):
        return {'user': Users.get_by_id(uid)}
```

```
MyRoute.route('/profiles/:uid')
```

If you remember what we've discussed about the return value of route handlers, you will notice that a `dict` is not one of the allowed return values. This is because a `dict` is not returned directly by *`TemplateRoute`* handlers.

The return value of the business logic methods is used as a template context, and a template specified by the `template_name` is rendered and returned as response (a string).

If you return any object other than a `dict`, it wil also be made available to the template context as `body` variable. The only exception is a `HTTPResponse` object which completely bypasses template rendering.

4.3.2 XHR partial rendering

XHR partial rendering is a technique where we use full and partial HTML representations of a single resource and return one or the other depending on whether a request is made using XHR (AJAX) or not. This can be useful for partial page updates using XHR.

The *`streamline.template.XHRPartialRoute`* is a variant of the *`TemplateRoute`* which uses two template names and selects an alternative template when a request is made using XHR.

Here is an example:

```
from streamline import XHRPartialRoute


class MyRoute(XHRPartialRoute):
    template_name = 'accounts/profile'
    partial_template_name = 'accounts/_profile'

    def get(self, uid):
        return {'user': Users.get_by_id(uid)}


MyRoute.route('/profiles/:uid')
```

As we can see, most of the code is identical to the previous example, with the addition of `partial_template_name` property.

4.3.3 Using a different rendering function

The default implementation of the classes discussed in this chapter use `bottle.template()` function to render the templates. You can override this by assigning a different function to the `template_func` attribute on both classes.

4.3.4 Default context

The default context can be changed globally by modifying the `bottle.BaseTemplate.defaults` dict. This will change the default context for all route handlers.

With CBRH, you can additionally change the default context of the route handler classes by modifying the `default_context` property. By default, the default context is `{'request': bottle.request}`.

4.3.5 Dynamically changing parameters at runtime

All of the values we've covered thus far (`template_name`, `partial_template_name`, `template_func` and `default_context`) can be also modified dynamically by overloading the following methods:

- `get_template_name()`
- `get_template_func()`
- `get_default_context()`

In addition, the way final template context is calculated can also be changed. By default, the context is calculated by merging the default context and the return value of the business logic methods. By overloading the `get_context()` function, you can change this behavior.

4.3.6 Customizing the rendering function invocation

The rendering function is invoked by passing the template name as first positional argument, and template context as second. The function returns the rendered template as a string or an iterable of strings. This is done in the `render_template()` method. You can customize the behavior by overloading this method.

4.4 Working with forms

While form handling is conceptually simple - user submits a form, which is processed on the server - there are potentially many nuances that make things more or less complicated. Instead of accounting for every possible workflow involving forms, bottle-streamline provides simple classes that provide the base for workflow-specific subclasses.

4.4.1 Form CBRH basics

The `streamline.forms.SimpleFormRoute` is, as its name suggest, a simple variant of form-handling CBRHs but it has all the fundamental pieces that you need to know in order to work with other variants. Let's take a look at an example of how to works:

```
import bottle
from streamline import FormRoute

class Simple(FormRoute):
    def show_form(self):
        return 'Imagine this is a form'

    def form_valid(self):
        return 'OK'

    def form_invalid(self):
        self.response.status = 400
        return 'WRONG'

Simple.route('/simple')
```

The `show_form()` method is normally used to render the form or otherwise make the form controls available to the user. In this case, we're simply returning a dummy string for simplicity. It's more or less an alias for `get()` and there is nothing wrong with overloading the `get()` method instead.

The `form_valid()` and `form_invalid()` methods are called depending on whether the form is valid or not. `Simple` class has no validation code, so it will always validate.

Let's add some validation code:

```
try:
    unicode = unicode
except NameError:
    unicode = str

required = lambda v: v and v.strip()
numeric = lambda v: unicode(v).strip().isnumeric() if v else True
```

The first lambda verfies that any data is entered and that the data is not just a series of whitespace characters. Second lambda makes the value optional, but, if specified, required it to be an integer. To make the two lambdas do anything, we first need to add them to the validator dictionary:

```
Simple.form_factory.validators['string'] = required
Simple.form_factory.validators['number'] = numeric
```

Now the `Simple` class is fully equipped to perform validation. Submitting data to an app running

---

Note: The form CBRHs have a `form_factory` property, which is a function or a class that returns objects that implement the *`streamline.forms.FormAdaptor`* API. The class itself can be used as a stand-alone rudimentary support code for data validation.

Let's test the form using curl:

```
$ curl --data "string=test" localhost:8080/simple
OK
$ curl --data "number=12" localhost:8080/simple
WRONG
```

It appears to be working.

Within the `form_valid()` and `form_invalid()` methods, the form object (as returned by the factory function) is available as the `form` property. Let's use this to show a bit more information on sucessful submission:

```
def form_valid(self):
    return 'OK: {} {}'.format(self.form.data.get('string', ''),
                              self.form.data.get('number', ''))
```

Let's test this:

```
$ curl --data "string=test" localhost:8080/simple
OK: test
$ curl --data "string=test&number=2" localhost:8080/simple
OK: test 2
```

4.4.2 Using form handling with template rendering

Form handling with template rendering is a cross between the *`FormRoute`* and the two template classes, *`TemplateRoute`* and *`XHRPartialRoute`*. The two classes are *`TemplateFormRoute`* and *`XHRPartialFormRoute`*.

We won't go into the details of how they work because they are simply a mix of fetures provided by the template CBRHs and the features outlined in the previous section.

4.4.3 Customizing form validation

Form validation can be customized in a few different ways. Most straightforward way is to use a different (and proper) form or validation library and write an adaptor for it. Another possibility is to override the `validate_form()` method.

Here is an example of a custom form adaptor:

```
from streamline.forms import FormAdaptor


class MyAdaptor(FormAdaptor):
    def __init__(self, data):
        self.messages = {}
        super(MyAdaptor, self).__init__(data)

    def is_valid(self):
        try:
            my_cool_validation_library(self.data)
        except ValidationError as e:
            for error in e.errors:
                self.messages[error.field_name] = error.message
        return not self.messages:


class Simple(FormRoute):
    form_factory = MyAdaptor
    ...
```

The custom adaptor saves error messages in the `messages` property on the adaptor object so that it can be accessed in the `form_invalid()` method later.

Now let's take a look at the second option of overloading the validation function:

```
class Simple(FormRoute):
    def __init__(self, *args, **kwargs):
        super(Simple, self).__init__(*args, **kwargs)
        self.errors = {}

    def validate_form(self, _):
        # We will completely ignore the form that is passed in, and instead
        # use ``request.forms``
        data = self.request.forms
        try:
            my_cool_validation_library(data)
        except ValidationError as e:
            for error in e.errors:
                self.errors[error.field_name] = error.message
        return not self.errors
```

It works similar to the first example, except we have chosen to ignore the form object that is passed in as an argument to `validate_form()`. The form object passed to this method is a form object returned by the factory function so we could have actually used the `form.data` instead of `request.forms` (they are identical).

4.4.4 Performing a redirect on sucessful submission

The most common way of performing a redirect is to use the bottle's own `redirect()` function. This function is made available as a method on the CBRH instances. Here is an example:

```
class Simple(FormRoute):
    ...
    def form_valid(self):
        return self.redirect('/see-other')
```

4.5 API documentation

This section contains API documentation for the bottle-streamline classes.

4.5.1 streamline.base

This module contains the base class which is used by all other class-based route handler classes.

class `streamline.base.`**`NonIterableResponseMixin`**

This mixin prevents the response data from being iterated upon byte-by-byte by bottle, which in case of large responses has a big performance impact, by wrapping the response data in a list.

class `streamline.base.`**`NonIterableRouteBase`**`(`**args*, ***kwargs*`)`

Provides the exact same functionality as *`RouteBase`*, only with *`NonIterableResponseMixin`* included, so that large response bodies are returned more efficiently.

`streamline.base.`**`Route`**

alias of *`RouteBase`*

class `streamline.base.`**`RouteBase`**`(`**args*, ***kwargs*`)`

Base class for class-based route handlers. This class produces iterable objects which are initialized with request parameters, and perform the parameter processing and constructing the response body.

When the response is handled by bottle, the object is iterated using the `__iter__()` method. Therefore, this object may be turned into a lazy object simply by postponing any evaluation until the the method is called.

static **`abort`**`(`*code=500*, *text='Unknown Error.'*`)`

Aborts execution and causes a HTTP error.

static **`after`**`(`*fn*`)`

Register a function as an after-create hook. After-create hooks are invoked after the `create_response` call, before iteration over the response body begins. They take the route handler object as the only argument, and their return value is ignored.

static **`before`**`(`*fn*`)`

Register a function as a before-create hook. Before-create hooks are invoked before the `create_response` call. They take the route handler object as the only argument, and their return value is ignored.

`bottle` **= <module 'bottle' from '/home/docs/checkouts/readthedocs.org/user_builds/bottle-streamline/envs/latest/lib/pyt**

classmethod **`get_generic_name`**`()`

Returns a generic name that can be used for naming a route. This name is in the `<module_name>:<decamelized_class_name>` format. For example if we have a class that is named `MyRoute` in a module called `beans`, the resulting generic name will be `beans:my_route`.

classmethod **`get_name`**`()`

Return the value of `name` attribute and fall back on a generic name returned by *`get_generic_name()`* class method.

classmethod **`get_path`**`()`

Return the value of `path` attribute.

static **`redirect`**(*url*, *code=None*)
Aborts execution and causes a 303 or 302 redirect, depending on the HTTP protocol version.

classmethod **`route`**(*path=None*, *name=None*, *app=None*, ***kwargs*)
Register a route by using class' configuration. This method will take an optional path, optional route name, and optional app object, and register a route for the specified path using class properties.

If `path` is not specified, a path will be obtained by invoking the *`get_path()`* class method. Similarly, if `name` is not specified, it will be obtained by invoking the *`get_name()`* class method. The default app that is used when `app` argument is missing is the Bottle's defalt app.

The handler is registered for http verbs (e.g., GET, POST) for which a lower-case method name exists that matches the verb.

List of plugins that should be applied or skipped can be specified by `include_plugins` and `exclude_plugins` properties respectively. These properties should be iterables containing the plugin names as per bottle API.

`streamline.base.`**`after`**(*fn*)
Decorator that register a function as an after hook.

Example:

```
@after
def myhook(route):
    pass
```

`streamline.base.`**`before`**(*fn*)
Decorator that registers a function as a before hook.

Example:

```
@before
def myhook(route):
    pass
```

4.5.2 streamline.template

This module contains mixins and classes for working with templates.

`streamline.template.`**`ROCARoute`**
alias of *`XHRPartialRoute`*

class `streamline.template.`**`TemplateMixin`**
Mixin that contains methods required to render templates. This class can be used by adding template-related features to any class-based route handler.

`get_context`()
Returns the complete template context. This context is is a dict, and is built from the default context by augmenting it with the contents of the `body` attribute on the route handler class.

If `body` attribute is a dict, its keys are copied to inthe context. Otherwise, a new key, `'body'` is added and the value of the attribute is assigned to it.

`get_default_context`()
Returns default context. Default behavior is to return the value of the `default_context` property. The property must be a dict, and it is copied before being returned so that the original remains intact.

`get_template_func`()
Return a template rendering function. Default behavior is to return the *`template_func`*, which must be a callable.

get_template_name(*template_name=None*)
Returns template name. Default behavior is to return the value of the `template_name` property.

If `template_name` argument is specified, it will be used instead of the property.

render_template()
Renders the template using the template name, context, and function obtained by calling the respective methods.

static **template_func**(**args*, ***kwargs*)
Get a rendered template as a string iterator. You can use a name, a filename or a template string as first parameter. Template rendering arguments can be passed as dictionaries or directly (as keyword arguments).

class `streamline.template.`**TemplateRoute**(**args*, ***kwargs*)
Class that renders the response into a template.

Subclasses `RouteBase`

Includes `TemplateMixin`

class `streamline.template.`**XHRPartialRoute**(**args*, ***kwargs*)
Class that renders different templates depending on whether request is XHR or not.

Subclasses `TemplateRoute`

4.5.3 streamline.forms

This module contains mixins and classes for working with forms.

class `streamline.forms.`**FormAdaptor**(*data={}*)
This class servers as an adaptor for 3rd party form APIs that works with the default form-handling API in this module. It can also be used on its own as it provides basic means of performing validation of form data.

The adaptor is instantiated with a dict-like object (usually `FormsDict`).

is_valid()
Perform validation and return a boolean result.

class `streamline.forms.`**FormBase**
Base mixin for form-related CBRH.

get(**args*, ***kwargs*)
Delegate to `show_form()`.

post(**args*, ***kwargs*)
Delegate to `validate()`.

class `streamline.forms.`**FormMixin**
Mixin that provides form-related functionality. The methods and properties in this class are implemented in a way that allows easy customization for any form API by focusing on the workflow rather than concreate APIs.

form_factory
alias of `FormAdaptor`

form_invalid(**args*, ***kwargs*)
Handle negative form validation outcome.

form_valid(**args*, ***kwargs*)
Handle positive form validation outcome.

get_bound_form()
Return bound form object.

get_form()
Return form object. Depending on the HTTP verb, this method returns either an unbound (GET) or a bound (all other verbs) form. Once a form object is created, this method will keep returning the previously created instance.

get_form_factory()
Return form factory function/class. Default behvarior is to return the *form_factory* property.

get_unbound_form()
Return unbound form object.

show_form(**args*, ***kwargs*)
Prepare for rendering a blank, unbound form.

validate(**args*, ***kwargs*)
Branch to one of the outcome methods depending on form validation result. The business logic should be writen in the outcome methods *form_valid()* and *form_invalid()*.

> **Warning:** An object that includes this mixin **must** set its `form` property to a form object before this method is invoked. Failure to do this will result in `AttributeError`. This is automatically handled in classes that include the *FormBase* mixin.

validate_form(*form*)
Perform validation and return the results of validation.

class streamline.forms.**FormRoute**(**args*, ***kwargs*)
Class for form handling without templates.

Subclasses *RouteBase*

Includes *FormMixin*, *FormBase*

streamline.forms.**ROCAFormRoute**
alias of *XHRPartialFormRoute*

class streamline.forms.**TemplateFormRoute**(**args*, ***kwargs*)
Class for form handling with template rendering.

Subclasses *TemplateRoute*

Includes *FormMixin*, *FormBase*

class streamline.forms.**XHRPartialFormRoute**(**args*, ***kwargs*)
Class for form handling with XHR partial rendering support.

Subclasses *XHRPartialRoute*

Includes *FormMixin FormBase*

4.5.4 streamline.utils

This module includes utility functions that are used in other modules.

streamline.utils.**decamelize**(*s*)
Convert CamelCase string to lowercase. Boundary between words is converted to underscore.

Example:

```
>>> decamelize('FreshFruit')
'fresh_fruit'
```

Note: This function is meant to be used with Python class names, which typically start with an upper-case letter. Names that start with a lower-case letter (e.g., 'freshFruit') will lose the first word during decamelization.

Python Module Index

Index