sprockets.mixins.mediatype

Release 3.0.4

unknown

Nov 02, 2020

CONTENTS

A mixin that performs Content-Type negotiation and request/response (de)serialization.

This mix-in adds two methods to a `tornado.web.RequestHandler` instance:

- `get_request_body() -> dict`: deserializes the request body according to the HTTP `Content-Type` header and returns the deserialized body.
- `send_response(object)`: serializes the response into the content type requested by the `Accept` header.

Before adding support for specific content types, you SHOULD install the content settings into your `tornado.web.Application` instance. If you don't install the content settings, then an instance will be created for you by the mix-in; however, the created instance will be empty. You should already have a function that creates the `Application` instance. If you don't, now is a good time to add one.

```
from sprockets.mixins.mediatype import content
from tornado import web

def make_application():
    application = web.Application([
        # insert your handlers here
    ])
    content.install(application, 'application/json', 'utf-8')
    return application
```

Support for a content types is enabled by calling `add_binary_content_type`, `add_text_content_type` or the `add_transcoder` functions with the `tornado.web.Application` instance, the content type, encoding and decoding functions as parameters:

```
import json

from sprockets.mixins.mediatype import content
from tornado import web

def make_application():
    application = web.Application([
        # insert your handlers here
    ])

    content.install(application, 'application/json', 'utf-8')
    content.add_text_content_type(application,
                                  'application/json', 'utf-8',
                                  json.dumps, json.loads)

    return application
```

The *add content type* functions will add a attribute to the `Application` instance that the mix-in uses to manipulate the request and response bodies. The *add_transcoder* function is similar except that it takes an object that implements transcoding methods instead of simple functions. The `transcoders` module includes ready-to-use transcoders for a few content types:

```
from sprockets.mixins.mediatype import content, transcoders
from tornado import web

def make_application():
    application = web.Application([
        # insert your handlers here
    ])
```

(continues on next page)

(continued from previous page)

```
    content.install(application, 'application/json', 'utf-8')
    content.add_transcoder(application, transcoders.JSONTranscoder())

    return application
```

In either case, the `ContentMixin` uses the registered content type information to provide transparent content type negotiation for your request handlers.

```
from sprockets.mixins.mediatype import content
from tornado import web

class SomeHandler(content.ContentMixin, web.RequestHandler):
    def get(self):
        self.send_response({'data': 'value'})

    def post(self):
        body = self.get_request_body()
        # do whatever
        self.send_response({'action': 'performed'})
```

Based on the settings stored in the `Application` instance and the HTTP headers, the request and response data will be handled correctly or the appropriate HTTP exceptions will be raised.

CHAPTER

ONE

EXAMPLES

```
import logging
import signal

from sprockets.mixins.mediatype import content, transcoders
from tornado import ioloop, web


class SimpleHandler(content.ContentMixin, web.RequestHandler):

    def post(self):
        body = self.get_request_body()
        self.set_status(200)
        self.send_response(body)


def make_application(**settings):
    application = web.Application([('/', SimpleHandler)], **settings)
    content.set_default_content_type(application, 'application/json',
                                     encoding='utf-8')
    content.add_transcoder(application, transcoders.MsgPackTranscoder())
    content.add_transcoder(application, transcoders.JSONTranscoder())
    return application


def _signal_handler(signo, _):
    logging.info('received signal %d, stopping application', signo)
    iol = ioloop.IOLoop.instance()
    iol.add_callback_from_signal(iol.stop)


if __name__ == '__main__':
    logging.basicConfig(level=logging.DEBUG,
                        format='%(levelname)1.1s - %(name)s: %(message)s')
    application = make_application(debug=True)
    application.listen(8000)
    signal.signal(signal.SIGINT, _signal_handler)
    signal.signal(signal.SIGTERM, _signal_handler)
    ioloop.IOLoop.instance().start()
```

CHAPTER

TWO

LICENSE

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of Sprockets nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

2.1 API Documentation

2.1.1 Content Type Handling

class sprockets.mixins.mediatype.content.**ContentMixin**

Mix this in to add some content handling methods.

```
class MyHandler(ContentMixin, web.RequestHandler):
   def post(self):
      body = self.get_request_body()
      # do stuff --> response_dict
      self.send_response(response_dict)
```

get_request_body() will deserialize the request data into a dictionary based on the Content-Type request header. Similarly, *send_response()* takes a dictionary, serializes it based on the Accept request header and the application *ContentSettings*, and writes it out, using `self.write()`.

get_request_body()
Fetch (and cache) the request body as a dictionary.

Raises **web.HTTPError** –

- if the content type cannot be matched, then the status code is set to 415 Unsupported Media Type.
- if decoding the content body fails, then the status code is set to 400 Bad Syntax.

get_response_content_type()
Figure out what content type will be used in the response.

send_response(*body*, *set_content_type=True*)
Serialize and send `body` in the response.

Parameters

- **body** (*dict*) – the body to serialize
- **set_content_type** (*bool*) – should the Content-Type header be set? Defaults to `True`

2.1.2 Content Type Registration

sprockets.mixins.mediatype.content.**install**(*application*, *default_content_type*, *encoding=None*)
Install the media type management settings.

Parameters

- **application** (*tornado.web.Application*) – the application to install a *ContentSettings* object into.
- **default_content_type** (*str|NoneType*) –
- **encoding** (*str|NoneType*) –

Returns the content settings instance

Return type *sprockets.mixins.mediatype.content.ContentSettings*

sprockets.mixins.mediatype.content.**get_settings**(*application*, *force_instance=False*)
Retrieve the media type settings for a application.

Parameters

- **application** (*tornado.web.Application*) –
- **force_instance** (*bool*) – if `True` then create the instance if it does not exist

Returns the content settings instance

Return type *sprockets.mixins.mediatype.content.ContentSettings*

sprockets.mixins.mediatype.content.**set_default_content_type**(*application*, *content_type*, *encoding=None*)
Store the default content type for an application.

Parameters

- **application** (*tornado.web.Application*) – the application to modify
- **content_type** (*str*) – the content type to default to

- **encoding** (*str* | *None*) – encoding to use when one is unspecified

sprockets.mixins.mediatype.content.**add_binary_content_type**(*application*, *content_type*, *pack*, *unpack*)

Add handler for a binary content type.

Parameters

- **application** (*tornado.web.Application*) – the application to modify
- **content_type** (*str*) – the content type to add
- **pack** – function that packs a dictionary to a byte string. `pack(dict) -> bytes`
- **unpack** – function that takes a byte string and returns a dictionary. `unpack(bytes) -> dict`

sprockets.mixins.mediatype.content.**add_text_content_type**(*application*, *content_type*, *default_encoding*, *dumps*, *loads*)

Add handler for a text content type.

Parameters

- **application** (*tornado.web.Application*) – the application to modify
- **content_type** (*str*) – the content type to add
- **default_encoding** (*str*) – encoding to use when one is unspecified
- **dumps** – function that dumps a dictionary to a string. `dumps(dict, encoding:str) -> str`
- **loads** – function that loads a dictionary from a string. `loads(str, encoding:str) -> dict`

Note that the `charset` parameter is stripped from *content_type* if it is present.

sprockets.mixins.mediatype.content.**add_transcoder**(*application*, *transcoder*, *content_type=None*)

Register a transcoder for a specific content type.

Parameters

- **application** (*tornado.web.Application*) – the application to modify
- **transcoder** – object that translates between `bytes` and `object` instances
- **content_type** (*str*) – the content type to add. If this is unspecified or `None`, then the transcoder's `content_type` attribute is used.

The *transcoder* instance is required to implement the following simple protocol:

`transcoder.`**content_type**

`str` that identifies the MIME type that the transcoder implements.

`transcoder.`**to_bytes**(*inst_data*, *encoding=None*) → bytes

Parameters

- **inst_data** (*object*) – the object to encode
- **encoding** (*str*) – character encoding to apply or `None`

Returns the encoded `bytes` instance

transcoder.**from_bytes**(*data_bytes*, *encoding=None*) → object

Parameters

- **data_bytes** (*bytes*) – the bytes instance to decode
- **encoding** (*str*) – character encoding to use or None

Returns the decoded object instance

class sprockets.mixins.mediatype.content.**ContentSettings**

Content selection settings.

An instance of this class is stashed in application.settings under the SETTINGS_KEY key. Instead of creating an instance of this class yourself, use the *install()* function to install it into the application.

The settings instance contains the list of available content types and handlers associated with them. Each handler implements a simple interface:

- to_bytes(dict, encoding:str) -> bytes
- from_bytes(bytes, encoding:str) -> dict

Use the *add_binary_content_type()* and *add_text_content_type()* helper functions to modify the settings for the application.

This class acts as a mapping from content-type string to the appropriate handler instance. Add new content types and find handlers using the common dict syntax:

```
class SomeHandler(web.RequestHandler):

   def get(self):
      settings = ContentSettings.get_settings(self.application)
      response_body = settings['application/msgpack'].to_bytes(
         response_dict, encoding='utf-8')
      self.write(response_body)

def make_application():
   app = web.Application([('/', SomeHandler)])
   add_binary_content_type(app, 'application/msgpack',
                           msgpack.packb, msgpack.unpackb)
   add_text_content_type(app, 'application/json', 'utf-8',
                         json.dumps, json.loads)
   return app
```

Of course, that is quite tedious, so use the *ContentMixin* instead.

property available_content_types

List of the content types that are registered.

This is a sequence of ietfparse.datastructures.ContentType instances.

2.1.3 Bundled Transcoders

class sprockets.mixins.mediatype.transcoders.**JSONTranscoder**(*content_type='application/json'*, *default_encoding='utf-8'*)

JSON transcoder instance.

Parameters

- **content_type** (*str*) – the content type that this encoder instance implements. If omitted, `application/json` is used. This is passed directly to the `TextContentHandler` initializer.
- **default_encoding** (*str*) – the encoding to use if none is specified. If omitted, this defaults to `utf-8`. This is passed directly to the `TextContentHandler` initializer.

This JSON encoder uses `json.loads()` and `json.dumps()` to implement JSON encoding/decoding. The *dump_object()* method is configured to handle types that the standard JSON module does not support.

dump_options
Keyword parameters that are passed to `json.dumps()` when *dumps()* is called. By default, the *dump_object()* method is enabled as the default object hook.

load_options
Keyword parameters that are passed to `json.loads()` when *loads()* is called.

dump_object(*obj*)
Called to encode unrecognized object.

Parameters **obj** (*object*) – the object to encode

Returns the encoded object

Raises **TypeError** – when *obj* cannot be encoded

This method is passed as the `default` keyword parameter to `json.dumps()`. It provides default representations for a number of Python language/standard library types.

Python Type	String Format
`bytes`, `bytearray`, `memoryview`	Base64 encoded string.
`datetime.datetime`	ISO8601 formatted timestamp in the extended format including separators, milliseconds, and the timezone designator.
`uuid.UUID`	Same as `str(value)`

dumps(*obj*)
Dump a `object` instance into a JSON `str`

Parameters **obj** (*object*) – the object to dump

Returns the JSON representation of `object`

loads(*str_repr*)
Transform `str` into an `object` instance.

Parameters **str_repr** (*str*) – the UNICODE representation of an object

Returns the decoded `object` representation

class sprockets.mixins.mediatype.transcoders.**MsgPackTranscoder**(*content_type='application/msgpack'*)
Msgpack Transcoder instance.

Parameters **content_type** (*str*) – the content type that this encoder instance implements. If omitted, `application/msgpack` is used. This is passed directly to the `BinaryContentHandler` initializer.

This transcoder uses the umsgpack library to encode and decode objects according to the msgpack format.

normalize_datum(*datum*)

Convert *datum* into something that umsgpack likes.

Parameters **datum** – something that we want to process with umsgpack

Returns a packable version of *datum*

Raises **TypeError** – if *datum* cannot be packed

This message is called by *packb()* to recursively normalize an input value before passing it to `umsgpack.packb()`. Values are normalized according to the following table.

Value	MsgPack Family
`None`	nil byte (0xC0)
`True`	true byte (0xC3)
`False`	false byte (0xC2)
`int`	integer family
`float`	float family
String	str family
`bytes`	bin family
`bytearray`	bin family
`memoryview`	bin family
`collections.abc.Sequence`	array family
`collections.abc.Set`	array family
`collections.abc.Mapping`	map family
`uuid.UUID`	Converted to String

packb(*data*)

Pack *data* into a `bytes` instance.

unpackb(*data*)

Unpack a `object` from a `bytes` instance.

2.2 How to Contribute

Do you want to contribute fixes or improvements?

> **AWesome!** *Thank you very much, and let's get started.*

2.2.1 Set up a development environment

The first thing that you need is a development environment so that you can run the test suite, update the documentation, and everything else that is involved in contributing. The easiest way to do that is to create a virtual environment for your endeavors:

```
$ python3 -m venv env
```

Don't worry about writing code against previous versions of Python unless you you don't have a choice. That is why we run our tests through tox. The next step is to install the development tools that this project uses. These are listed in *requires/development.txt*:

```
$ env/bin/pip install -qr requires/development.txt
```

At this point, you will have everything that you need to develop at your disposal. Use the unittest runner to run the test suite or the coverage utility to run the suite with coverage calculation enabled:

```
$ coverage run -m unittest
$ coverage report
```

You can also run the tox utility to verify the supported Python versions:

```
$ tox -p auto
✓ OK py37 in 2.636 seconds
✓ OK py38 in 2.661 seconds
✓ OK py39 in 2.705 seconds
_____________________________________________________________________________
  py37: commands succeeded
  py38: commands succeeded
  py39: commands succeeded
  congratulations :)
```

For other commands, *setup.py* is the swiss-army knife in your development tool chest. It provides the following commands:

./setup.py build_sphinx Generate the documentation using sphinx.

./setup.py flake8 Run flake8 over the code and report style violations.

If any of the preceding commands give you problems, then you will have to fix them **before** your pull request will be accepted.

2.2.2 Running Tests

The easiest (and quickest) way to run the test suite is to use the unittest runner:

```
$ python -m unittest tests
```

That's the quick way to run tests. The slightly longer way is to run the tox utility. It will run the test suite against all of the supported python versions in parallel. This is essentially what Travis-CI will do when you issue a pull request anyway:

```
$ tox -p auto
✓ OK py37 in 2.636 seconds
✓ OK py38 in 2.661 seconds
✓ OK py39 in 2.705 seconds
_____________________________________________________________________________
  py37: commands succeeded
  py38: commands succeeded
  py39: commands succeeded
  congratulations :)
```

This is what you want to see. Now you can make your modifications and keep the tests passing.

2.2.3 Submitting a Pull Request

Once you have made your modifications, gotten all of the tests to pass, and added any necessary documentation, it is time to contribute back for posterity. You've probably already cloned this repository and created a new branch. If you haven't, then checkout what you have as a branch and roll back *master* to where you found it. Then push your repository up to github and issue a pull request. Describe your changes in the request, if Travis isn't too annoyed someone will review it, and eventually merge it back.

2.3 Version History

2.3.1 3.0.4 (2 Nov 2020)

- Return a "400 Bad Request" when an invalid Content-Type header is received instead of failing with an internal server error

2.3.2 3.0.3 (14 Sep 2020)

- Import from collections.abc instead of collections (thanks @nullsvm)

2.3.3 3.0.2 (4 May 2020)

- Do not log tracebacks when decoding the request body fails

2.3.4 3.0.1 (5 Mar 2019)

- Set Tornado PIN to >=5, <7
- Remove setuptools_scm

2.3.5 3.0.0 (4 Dec 2018)

- Add MessagePack dependencies to package extras (eg. *pip install sprockets.mixins.mediatype[msgpack]*)
- Update to minimum of ietfparse 1.5.1
- Drop support for Python < 3.7
- Drop support for Tornado < 5
- Remove deprecated `sprockets.mixins.mediatype.content.ContentSettings.from_application()`.

2.3.6 2.2.2 (7 Apr 2018)

- Add support for Python 3.5 through 3.7
- Add support for Tornado < 6

2.3.7 2.2.1 (12 Apr 2018)

- Pin `ietfparse` to avoid breakages introduced in 1.5.0.

2.3.8 2.2.0 (7 Jun 2017)

- Add *sprockets.mixins.mediatype.content.install()*.
- Add *sprockets.mixins.mediatype.content.get_settings()*.
- Deprecate `sprockets.mixins.mediatype.content.ContentSettings.from_application()`.
- Update to ietfparse 1.4.

2.3.9 2.1.0 (16 Mar 2016)

- Set the Vary header if we are setting the content type.

2.3.10 2.0.1 (29 Feb 2016)

- Removed deprecation wrapper since it seems to cause really interesting problems including the much feared meta-class error.

2.3.11 2.0.0 (24 Feb 2016)

- Repackage from a module into a package. Distributing raw modules inside of a namespace package is unreliable and questionably correct.
- Add *sprockets.mixins.mediatype.content.add_transcoder()*.
- Add *sprockets.mixins.mediatype.transcoders.JSONTranscoder*.
- Add *sprockets.mixins.mediatype.transcoders.MsgPackTranscoder*.
- Add `sprockets.mixins.mediatype.transcoders.BinaryWrapper`.
- Normalize registered MIME types.
- Raise a 400 status when content body decoding fails.

2.3.12 1.0.4 (14 Sep 2015)

- Support using the default_content_type in the settings if request does not contain the Accept header

2.3.13 1.0.3 (10 Sep 2015)

- Update installation files

2.3.14 1.0.2 (9 Sep 2015)

- Rename package to mediatype

2.3.15 1.0.1 (9 Sep 2015)

- Repackaged for Travis-CI configuration.

2.3.16 1.0.0 (9 Sep 2015)

- Initial Release

INDEX

U