webargs

Release 1.3.4

June 11, 2016

Contents

A friendly library for parsing HTTP request arguments.

Release v1.3.4. (*Changelog*)

webargs is a Python library for parsing HTTP request arguments, with built-in support for popular web frameworks, including Flask, Django, Bottle, Tornado, Pyramid, webapp2, Falcon, and aiohttp.

```
from flask import Flask
from webargs import fields
from webargs.flaskparser import use_args

app = Flask(__name__)

hello_args = {
    'name': fields.Str(required=True)
}

@app.route('/')
@use_args(hello_args)
def index(args):
    return 'Hello ' + args['name']

if __name__ == '__main__':
    app.run()

# curl http://localhost:5000/\?name\='World'
# Hello World
```

Webargs will automatically parse:

Query Parameters

```
$ curl http://localhost:5000/\?name\='Freddie'
Hello Freddie
```

Form Data

```
$ curl -d 'name=Brian' http://localhost:5000/
Hello Brian
```

JSON Data

```
$ curl -X POST -H "Content-Type: application/json" -d '{"name":"Roger"}' http://localhost:5000/
Hello Roger
```

and, optionally:

- Headers
- Cookies
- Files
- Paths

CHAPTER 1

Why Use It

- **Simple, declarative syntax**. Define your arguments as a mapping rather than imperatively pulling values off of request objects.
- **Code reusability**. If you have multiple views that have the same request parameters, you only need to define your parameters once. You can also reuse validation and pre-processing routines.
- **Self-documentation**. Webargs makes it easy to understand the expected arguments and their types for your view functions.
- **Automatic documentation**. The metadata that webargs provides can serve as an aid for automatically generating API documentation.
- **Cross-framework compatibility**. Webargs provides a consistent request-parsing interface that will work across many Python web frameworks.
- **marshmallow integration**. Webargs uses marshmallow under the hood. When you need more flexibility than dictionaries, you can use marshmallow `Schemas` to define your request arguments.

CHAPTER 2

Get It Now

```
pip install -U webargs
```

Ready to get started? Go on to the *Quickstart tutorial* or check out some examples.

CHAPTER 3

User Guide

3.1 Install

webargs requires Python >= 2.6 or >= 3.3. It depends on marshmallow >= 2.0.

3.1.1 From the PyPI

To install the latest version from the PyPI:

```
pip install -U webargs
```

3.1.2 Get the Bleeding Edge Version

To get the latest development version of webargs, run

```
$ pip install -U git+https://github.com/sloria/webargs.git@dev
```

3.2 Quickstart

3.2.1 Basic Usage

Arguments are specified as a dictionary of name -> `Field` pairs.

```
from webargs import fields, validate

user_args = {

    # Required arguments
    'username': fields.Str(required=True),

    # Validation
    'password': fields.Str(validate=lambda p: len(p) >= 6),

    # OR use marshmallow's built-in validators
    'password': fields.Str(validate=validate.Length(min=6)),

    # Default value when argument is missing
```

```
    'display_per_page': fields.Int(missing=10),

    # Repeated parameter, e.g. "/?nickname=Fred&nickname=Freddie"
    'nickname': fields.List(fields.Str()),

    # Delimited list, e.g. "/?languages=python,javascript"
    'languages': fields.DelimitedList(fields.Str())

    # When you know where an argument should be parsed from
    'active': fields.Bool(location='query')

    # When value is keyed on a variable-unsafe name
    # or you want to rename a key
    'content_type': fields.Str(load_from='Content-Type',
                               location='headers')
}
```

Note: See the `marshmallow.fields` documentation for a full reference on available field types.

To parse request arguments, use the *`parse`* method of a *`Parser`* object.

```
from flask import request
from webargs.flaskparser import parser

@app.route('/register', methods=['POST'])
def register():
    args = parser.parse(user_args, request)
    return register_user(args['username'], args['password'],
        fullname=args['fullname'], per_page=args['display_per_page'])
```

3.2.2 Decorator API

As an alternative to `Parser.parse`, you can decorate your view with *`use_args`* or *`use_kwargs`*. The parsed arguments dictionary will be injected as a parameter of your view function or as keyword arguments, respectively.

```
from webargs.flaskparser import use_args, use_kwargs

@app.route('/register', methods=['POST'])
@use_args(user_args)  # Injects args dictionary
def register(args):
    return register_user(args['username'], args['password'],
        fullname=args['fullname'], per_page=args['display_per_page'])

@app.route('/settings', methods=['POST'])
@use_kwargs(user_args)  # Injects keyword arguments
def user_settings(username, password, fullname, display_per_page, nickname):
    return render_template('settings.html', username=username, nickname=nickname)
```

Note: When using `use_kwargs`, any missing values for non-required fields will take the special value `missing`.

```
from webargs import fields, missing

@use_kwargs({'name': fields.Str(), 'nickname': fields.Str(required=False)})
def myview(name, nickname):
```

```
    if nickname is missing:
        # ...
```

3.2.3 Request "Locations"

By default, webargs will search for arguments from the URL query string (e.g. `"/?name=foo"`), form data, and JSON data (in that order). You can explicitly specify which locations to search, like so:

```
@app.route('/register')
@use_args(user_args, locations=('json', 'form'))
def register(args):
    return 'registration page'
```

Available locations include:

- `'querystring'` (same as `'query'`)
- `'json'`
- `'form'`
- `'headers'`
- `'cookies'`
- `'files'`

3.2.4 Validation

Each `Field` object can be validated individually by passing the `validate` argument.

```
from webargs import fields

args = {
    'age': fields.Int(validate=lambda val: val > 0)
}
```

The validator may return either a `boolean` or raise a *`ValidationError`*.

```
from webargs import fields, ValidationError

def must_exist_in_db(val):
    if not User.query.get(val):
        # Optionally pass a status_code
        raise ValidationError('User does not exist')

argmap = {
    'id': fields.Int(validate=must_exist_in_db)
}
```

Note: You may also pass a list of validators to the `validate` parameter.

Note: You may pass an HTTP status code to *`ValidationError`*.

```
def must_exist_in_db(val):
    if not User.query.get(val):
        # Optionally pass a status_code
        raise ValidationError('User does not exist', status_code=404)

argmap = {
    'id': fields.Int(validate=must_exist_in_db)
}
```

The full arguments dictionary can also be validated by passing `validate` to *Parser.parse*, *Parser.use_args*, *Parser.use_kwargs*.

```
from webargs import fields
from webargs.flaskparser import parser

argmap = {
    'age': fields.Int(),
    'years_employed': fields.Int(),
}

# ...
result = parser.parse(argmap,
                      validate=lambda args: args['years_employed'] < args['age'])
```

3.2.5 Error Handling

Each parser has a default error handling method. To override the error handling callback, write a function that receives an error and handles it, then decorate that function with *Parser.error_handler*.

```
from webargs import core
parser = core.Parser()

class CustomError(Exception):
    pass

@parser.error_handler
def handle_error(error):
    raise CustomError(error.messages)
```

3.2.6 Nesting Fields

`Field` dictionaries can be nested within each other. This can be useful for validating nested data.

```
from webargs import fields

args = {
    'name': fields.Nested({
        'first': fields.Str(required=True),
        'last': fields.Str(required=True),
    })
}
```

Note: By default, webargs only parses nested fields using the `json` request location. You can, however, *implement*

your own parser to add nested field functionality to the other locations.

3.2.7 Next Steps

- Go on to *Advanced Usage* to learn how to add custom location handlers, use marshmallow Schemas, and more.
- See the *Framework Support* page for framework-specific guides.
- For example applications, check out the examples directory.

3.3 Advanced Usage

This section includes guides for advanced usage patterns.

3.3.1 Custom Location Handlers

To add your own custom location handler, write a function that receives a request, an argument name, and a `Field`, then decorate that function with *`Parser.location_handler`*.

```
from webargs import fields
from webargs.flaskparser import parser

@parser.location_handler('data')
def parse_data(request, name, field):
    return request.data.get(name)

# Now 'data' can be specified as a location
@parser.use_args({'per_page': fields.Int()}, locations=('data', ))
def posts(args):
    return 'displaying {} posts'.format(args['per_page'])
```

3.3.2 Marshmallow Integration

When you need more flexibility in defining input schemas, you can pass a marshmallow `Schema` instead of a dictionary to *`Parser.parse`*, *`Parser.use_args`*, and *`Parser.use_kwargs`*.

```
from marshmallow import Schema, fields
from webargs.flaskparser import use_args

class UserSchema(Schema):
    id = fields.Int(dump_only=True)  # read-only (won't be parsed by webargs)
    username = fields.Str(required=True)
    password = fields.Str(load_only=True)  # write-only
    first_name = fields.Str(missing='')
    last_name = fields.Str(missing='')
    date_registered = fields.DateTime(dump_only=True)

    class Meta:
        strict = True


@use_args(UserSchema())
```

```
def profile_view(args):
    # ...

@use_kwargs(UserSchema())
def profile_update(username, password, first_name, last_name):
    # ...

# You can add additional paramters
@use_kwargs({'posts_per_page': fields.Int(missing=10, location='query')})
@use_args(UserSchema())
def profile_posts(args, posts_per_page):
    # ...
```

Note: You should always set `strict=True` (either as a `class Meta` option or in the Schema's constructor) when passing a schema to webargs. This will ensure that the parser's error handler is invoked when expected.

3.3.3 Schema Factories

If you need to parametrize a schema based on a given request, you can use a "Schema factory": a callable that receives the current `request` and returns a `marshmallow.Schema` instance.

Consider the following use cases:

- Filtering via a query parameter by passing `only` to the Schema.
- Handle partial updates for PATCH requests using marshmallow's partial loading API.

```
from marshmallow import Schema, fields
from webargs.flaskparser import use_args

class UserSchema(Schema):
    id = fields.Int(dump_only=True)
    username = fields.Str(required=True)
    password = fields.Str(load_only=True)
    first_name = fields.Str(missing='')
    last_name = fields.Str(missing='')
    date_registered = fields.DateTime(dump_only=True)

    class Meta:
        strict = True

def make_user_schema(request):
    # Filter based on 'fields' query parameter
    only = request.args.get('fields', None)
    # Respect partial updates for PATCH requests
    partial = request.method == 'PATCH'
    # Add current request to the schema's context
    return UserSchema(only=only, partial=partial, context={'request': request})

# Pass the factory to .parse, .use_args, or .use_kwargs
@use_args(make_user_schema):
def profile_view(args):
    # ...
```

Reducing Boilerplate

We can reduce boilerplate and improve [re]usability with a simple helper function:

```
from webargs.flaskparser import use_args

def use_args_with(schema_cls, schema_kwargs=None, **kwargs):
    schema_kwargs = schema_kwargs or {}
    def factory(request):
        # Filter based on 'fields' query parameter
        only = request.args.get('fields', None)
        # Respect partial updates for PATCH requests
        partial = request.method == 'PATCH'
        # Add current request to the schema's context
        # and ensure we're always using strict mode
        return schema_cls(
            only=only, partial=partial, strict=True,
            context={'request': request}, **schema_kwargs
        )
    return use_args(factory, **kwargs)
```

Now we can attach input schemas to our view functions like so:

```
@use_args_with(UserSchema)
def profile_view(args):
    # ...
```

3.3.4 Custom Fields

See the "Custom Fields" section of the marshmallow docs for a detailed guide on defining custom fields which you can pass to webargs parsers: https://marshmallow.readthedocs.io/en/latest/custom_fields.html.

Using `Method` and `Function` Fields with webargs

Using the `Method` and `Function` fields requires that you pass the `deserialize` parameter.

```
@use_args({
    'cube': fields.Function(deserialize=lambda x: int(x) ** 3)
})
def math_view(args):
    cube = args['cube']
    # ...
```

3.3.5 Custom Parsers

To add your own parser, extend *`Parser`* and implement the `parse_*` method(s) you need to override. For example, here is a custom Flask parser that handles nested query string arguments.

```
import re

from webargs import core
from webargs.flaskparser import FlaskParser

class NestedQueryFlaskParser(FlaskParser):
    """Parses nested query args
```

```
    This parser handles nested query args. It expects nested levels
    delimited by a period and then deserializes the query args into a
    nested dict.

    For example, the URL query params `?name.first=John&name.last=Boone`
    will yield the following dict:

        {
            'name': {
                'first': 'John',
                'last': 'Boone',
            }
        }
    """

    def parse_querystring(self, req, name, field):
        return core.get_value(_structure_dict(req.args), name, field)


def _structure_dict(dict_):
    def structure_dict_pair(r, key, value):
        m = re.match(r'(\w+)\.(.*)', key)
        if m:
            if r.get(m.group(1)) is None:
                r[m.group(1)] = {}
            structure_dict_pair(r[m.group(1)], m.group(2), value)
        else:
            r[key] = value
    r = {}
    for k, v in dict_.items():
        structure_dict_pair(r, k, v)
    return r
```

3.3.6 Bulk-type Arguments

In order to parse a JSON array of objects, pass `many=True` to your input `Schema` .

For example, you might implement JSON PATCH according to RFC 6902 like so:

```
from webargs import fields
from webargs.flaskparser import use_args
from marshmallow import Schema, validate

class PatchSchema(Schema):
    op = fields.Str(
        required=True,
        validate=validate.OneOf(['add', 'remove', 'replace', 'move', 'copy'])
    )
    path = fields.Str(required=True)
    value = fields.Str(required=True)

    class Meta:
        strict = True


@app.route('/profile/', methods=['patch'])
@use_args(PatchSchema(many=True), locations=('json', ))
```

```
def patch_blog(args):
    """Implements JSON Patch for the user profile

    Example JSON body:

    [
        {"op": "replace", "path": "/email", "value": "mynewemail@test.org"}
    ]
    """
    # ...
```

3.3.7 Mixing Locations

Arguments for different locations can be specified by passing `location` to each field individually:

```
@app.route('/stacked', methods=['POST'])
@use_args({
    'page': fields.Int(location='query')
    'q': fields.Str(location='query')
    'name': fields.Str(location='json'),
})
def viewfunc(args):
    # ...
```

Alternatively, you can pass multiple locations to *use_args*:

```
@app.route('/stacked', methods=['POST'])
@use_args({
    'page': fields.Int()
    'q': fields.Str()
    'name': fields.Str(),
} , locations=('query', 'json'))
def viewfunc(args):
    # ...
```

However, this allows `page` and `q` to be passed in the request body and `name` to be passed as a query parameter.

To restrict the arguments to single locations without having to pass `location` to every field, you can call the *use_args* multiple times:

```
query_args = {
    'page': fields.Int()
    'q': fields.Int()
}
json_args = {
    'name': fields.Str(),
}
@app.route('/stacked', methods=['POST'])
@use_args(query_args, locations=('query', ))
@use_args(json_args, locations=('json', ))
def viewfunc(query_parsed, json_parsed):
    # ...
```

To reduce boilerplate, you could create shortcuts, like so:

```
import functools

query = functools.partial(use_args, locations=('query', ))
```

```
body = functools.partial(use_args, locations=('json', ))

@query(query_args)
@body(json_args)
def viewfunc(query_parsed, json_parsed):
    # ...
```

3.3.8 Next Steps

- See the *Framework Support* page for framework-specific guides.
- For example applications, check out the examples directory.

3.4 Framework Support

This section includes notes for using webargs with specific web frameworks.

3.4.1 Flask

Flask support is available via the *webargs.flaskparser* module.

Decorator Usage

When using the `use_args` decorator, the arguments dictionary will be *before* any URL variable parameters.

```
from webargs import fields
from webargs.flaskparser import use_args

@app.route('/user/<int:uid>')
@use_args({'per_page': fields.Int()})
def user_detail(args, uid):
    return ('The user page for user {uid}, '
            'showing {per_page} posts.').format(uid=uid,
                                                per_page=args['per_page'])
```

Error Handling

Webargs uses Flask's `abort` function to raise an `HTTPException` when a validation error occurs. If you use the `Flask.errorhandler` method to handle errors, you can access validation messages from the `data` attribute of an error.

Here is an example error handler that returns validation messages to the client as JSON.

```
from flask import jsonify

@app.errorhandler(422)
def handle_unprocessable_entity(err):
    # webargs attaches additional metadata to the `data` attribute
    data = getattr(err, 'data')
    if data:
        # Get validations from the ValidationError object
        messages = data['exc'].messages
```

```
    else:
        messages = ['Invalid request']
    return jsonify({
        'messages': messages,
    }), 422
```

URL Matches

The `FlaskParser` supports parsing values from a request's `view_args`.

```
from webargs.flaskparser import use_args

@app.route('/greeting/<name>/')
@use_args({'name': fields.Str(location='view_args')})
def greeting(args, **kwargs):
    return 'Hello {}'.format(args['name'])
```

3.4.2 Django

Django support is available via the *`webargs.djangoparser`* module.

Webargs can parse Django request arguments in both function-based and class-based views.

Decorator Usage

When using the `use_args` decorator, the arguments dictionary will positioned after the `request` argument.

Function-based Views

```
from django.http import HttpResponse
from webargs import Arg
from webargs.djangoparser import use_args

account_args = {
  'username': fields.Str(required=True),
  'password': fields.Str(required=True),
}

@use_args(account_args)
def login_user(request, args):
    if request.method == 'POST':
        login(args['username'], args['password'])
    return HttpResponse('Login page')
```

Class-based Views

```
from django.views.generic import View
from django.shortcuts import render_to_response
from webargs import fields
from webargs.djangoparser import use_args

blog_args = {
    'title': fields.Str(),
    'author': fields.Str(),
}
```

```
class BlogPostView(View):

    @use_args(blog_args)
    def get(self, request, args):
      blog_post = Post.objects.get(title__iexact=args['title'],
                                   author=args['author'])
      return render_to_response('post_template.html',
                                {'post': blog_post})
```

Error Handling

The `DjangoParser` does not override *`handle_error`*, so your Django views are responsible for catching any `ValidationErrors` raised by the parser and returning the appropriate `HTTPResponse`.

```
from django.http import JsonResponse

from webargs import fields, ValidationError

argmap = {
    'name': fields.Str(required=True)
}
def index(request):
    try:
        args = parser.parse(argmap, request)
    except ValidationError as err:
        return JsonResponse(err.messages, status=err.status_code)
    return JsonResponse({'message': 'Hello {name}'.format(name=name)})
```

3.4.3 Tornado

Tornado argument parsing is available via the *`webargs.tornadoparser`* module.

The *`webargs.tornadoparser.TornadoParser`* parses arguments from a `tornado.httpserver.HTTPRequest` object. The *`TornadoParser`* can be used directly, or you can decorate handler methods with `use_args` or `use_kwargs`.

```
import tornado.ioloop
import tornado.web

from webargs import fields
from webargs.tornadoparser import parser


class HelloHandler(tornado.web.RequestHandler):

    hello_args = {
        'name': fields.Str()
    }

    def post(self, id):
        reqargs = parser.parse(self.hello_args, self.request)
        response = {
            'message': 'Hello {}'.format(reqargs['name'])
        }
        self.write(response)
```

```
application = tornado.web.Application([
    (r"/hello/([0-9]+)", HelloHandler),
], debug=True)

if __name__ == "__main__":
    application.listen(8888)
    tornado.ioloop.IOLoop.instance().start()
```

Decorator Usage

When using the `use_args` decorator, the decorated method will have the dictionary of parsed arguments passed as a positional argument after `self` and any regex match groups from the URL spec.

```
from webargs import fields
from webargs.tornadoparser import use_args

class HelloHandler(tornado.web.RequestHandler):

    @use_args({'name': fields.Str()})
    def post(self, id, reqargs):
        response = {
            'message': 'Hello {}'.format(reqargs['name'])
        }
        self.write(response)

application = tornado.web.Application([
    (r"/hello/([0-9]+)", HelloHandler),
], debug=True)
```

As with the other parser modules, `use_kwargs` will add keyword arguments to the view callable.

Error Handling

A *HTTPError* will be raised in the event of a validation error. Your `RequestHandlers` are responsible for handling these errors.

Here is how you could write the error messages to a JSON response.

```
from tornado.web import RequestHandler

class MyRequestHandler(RequestHandler):

    def write_error(self, status_code, **kwargs):
        """Write errors as JSON."""
        self.set_header('Content-Type', 'application/json')
        if 'exc_info' in kwargs:
            etype, value, traceback = kwargs['exc_info']
            if hasattr(value, 'messages'):
                self.write({'errors': value.messages})
                self.finish()
```

3.4.4 Pyramid

Pyramid support is available via the *webargs.pyramidparser* module.

Decorator Usage

When using the *use_args* decorator on a view callable, the arguments dictionary will be positioned after the `request` argument.

```
from pyramid.response import Response
from webargs import fields
from webargs.pyramidparser import use_args

@use_args({'per_page': fields.Int()})
def user_detail(request, args):
    return Response('The user page for user {uid}, '
            'showing {per_page} posts.').format(uid=uid,
                                                per_page=args['per_page']))
```

As with the other parser modules, `use_kwargs` will add keyword arguments to the view callable.

URL Matches

The `PyramidParser` supports parsing values from a request's matchdict.

```
from pyramid.response import Response
from webargs.pyramidparser import use_args

@parser.use_args({'mymatch': fields.Int()}, locations=('matchdict',))
def matched(request, args):
    return Response('The value for mymatch is {}'.format(args['mymatch'])))
```

3.4.5 Falcon

Falcon support is available via the *webargs.falconparser* module.

Decorator Usage

When using the `use_args` decorator on a resource method, the arguments dictionary will be positioned directly after the request and response arguments.

```
import falcon
from webargs import fields
from webargs.falconparser import use_args

class BlogResource:
    request_args = {
        'title': fields.Str(required=True)
    }

    @use_args(request_args)
    def on_post(self, req, resp, args, post_id):
        content = args['title']
        # ...

api = application = falcon.API()
api.add_route('/blogs/{post_id}')
```

As with the other parser modules, `use_kwargs` will add keyword arguments to your resource methods.

Hook Usage

You can easily implement hooks by using `parser.parse` directly.

```
import falcon
from webargs import fields
from webargs.falconparser import parser

def add_args(argmap, **kwargs):
    def hook(req, resp, params):
        parsed_args = parser.parse(argmap, req=req, **kwargs)
        req.context['args'] = parsed_args
    return hook

@falcon.before(add_args({'page': fields.Int(location='query')}))
class AuthorResource:

    def on_get(self, req, resp):
        args = req.context['args']
        page = args.get('page')
        # ...
```

3.4.6 aiohttp

aiohttp support is available via the *webargs.aiohttpparser* module.

The `parse` method of *AIOHTTPParser* is a `coroutine`.

```
import asyncio

from aiohttp import web
from webargs import fields
from webargs.aiohttpparser import parser

handler_args = {
    'name': fields.Str(missing='World')
}
@asyncio.coroutine
def handler(request):
    args = yield from parser.parse(handler_args, request)
    return web.Response(
        body='Hello, {}'.format(args['name']).encode('utf-8')
    )
```

Decorator Usage

When using the `use_args` decorator on a handler, the parsed arguments dictionary will be the last positional argument.

```
import asyncio

from aiohttp import web
from webargs import fields
from webargs.aiohttpparser import use_args

@asyncio.coroutine
```

```
@use_args({'content': fields.Str(required=True)})
def create_comment(request, args):
    content = args['content']
    # ...

app = web.Application()
app.router.add_route('POST', '/comments/', create_comment)
```

As with the other parser modules, `use_kwargs` will add keyword arguments to your resource methods.

Usage with coroutines

The `use_args` and `use_kwargs` decorators will not work with `async def` coroutines. You must either use a generator-based coroutine decorated with `asyncio.coroutine` or use `parser.parse`.

```
from aiohttp import web

from webargs import fields

hello_args = {
    'name': fields.Str(missing='World')
}

# YES
from webargs.aiohttpparser import parser

async def hello(request):
    args = await parser.parse(hello_args, request)
    return web.Response(
        body='Hello, {}'.format(name).encode('utf-8')
    )

# YES
import asyncio
from webargs.aiohttpparser import use_kwargs

@asyncio.coroutine
@use_kwargs(hello_args)
def hello(request, name):
    return web.Response(
        body='Hello, {}'.format(name).encode('utf-8')
    )

# NO: use_args and use_kwargs are incompatible with async def
@use_kwargs(hello_args)
async def hello(request, name):
    return web.Response(
        body='Hello, {}'.format(name).encode('utf-8')
    )
```

URL Matches

The *AIOHTTPParser* supports parsing values from a request's `match_info`.

```
from aiohttp import web
from webargs.aiohttpparser import use_args
```

```
@parser.use_args({'slug': fields.Str(location='match_info')})
def article_detail(request, args):
    return web.Response(
        body='Slug: {}'.format(args['slug']).encode('utf-8')
    )

app = web.Application()
app.router.add_route('GET', '/articles/{slug}', article_detail)
```

CHAPTER 4

API Reference

4.1 API

4.1.1 webargs.core

exception webargs.core.**WebargsError**
Base class for all webargs-related errors.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception webargs.core.**ValidationError**(*message*, *status_code=422*, *headers=None*, ***kwargs*)
Raised when validation fails on user input. Same as marshmallow.ValidationError, with the addition of the status_code and headers arguments.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

webargs.core.**argmap2schema**(*argmap*, *instance=False*, ***kwargs*)
Generate a marshmallow.Schema class given a dictionary of argument names to Fields.

webargs.core.**is_multiple**(*field*)
Return whether or not field handles repeated/multi-value arguments.

class webargs.core.**Parser**(*locations=None*, *error_handler=None*)
Base parser class that provides high-level implementation for parsing a request.

Descendant classes must provide lower-level implementations for parsing different locations, e.g. parse_json, parse_querystring, etc.

Parameters

- **locations** (*tuple*) – Default locations to parse.
- **error_handler** (*callable*) – Custom error handler function.

clear_cache()
Invalidate the parser's cache.

error_handler(*func*)
Decorator that registers a custom error handling function. The function should received the raised error. Overrides the parser's handle_error method.

Example:

```
from webargs import core
parser = core.Parser()

class CustomError(Exception):
    pass

@parser.error_handler
def handle_error(error):
    raise CustomError(error)
```

Parameters **func** (*callable*) – The error callback to register.

get_default_request()
Optional override. Provides a hook for frameworks that use thread-local request objects.

get_request_from_view_args(*view*, *args*, *kwargs*)
Optional override. Returns the request object to be parsed, given a view function's args and kwargs.

Used by the *use_args* and *use_kwargs* to get a request object from a view's arguments.

Parameters

- **view** (*callable*) – The view function or method being decorated by *use_args* or *use_kwargs*
- **args** (*tuple*) – Positional arguments passed to `view`.
- **kwargs** (*dict*) – Keyword arguments passed to `view`.

handle_error(*error*)
Called if an error occurs while parsing args. By default, just logs and raises `error`.

location_handler(*name*)
Decorator that registers a function for parsing a request location. The wrapped function receives a request, the name of the argument, and the corresponding `Field` object.

Example:

```
from webargs import core
parser = core.Parser()

@parser.location_handler('name')
def parse_data(request, name, field):
    return request.data.get(name)
```

Parameters **name** (*str*) – The name of the location to register.

parse(*argmap*, *req=None*, *locations=None*, *validate=None*, *force_all=False*)
Main request parsing method.

Parameters

- **argmap** – Either a `marshmallow.Schema`, a `dict` of argname -> `marshmallow.fields.Field` pairs, or a callable which accepts a request and returns a `marshmallow.Schema`.
- **req** – The request object to parse.
- **locations** (*tuple*) – Where on the request to search for values. Can include one or more of `('json', 'querystring', 'form', 'headers', 'cookies', 'files')`.

- **validate** (*callable*) –

  Validation function or list of validation functions that receives the dictionary of parsed arguments. Validator either returns a boolean or raises a *ValidationError*.

  return A dictionary of parsed arguments

parse_arg(*name*, *field*, *req*, *locations=None*)
Parse a single argument from a request.

Note: This method does not perform validation on the argument.

Parameters

- **name** (*str*) – The name of the value.
- **field** (*marshmallow.fields.Field*) – The marshmallow `Field` for the request parameter.
- **req** – The request object to parse.
- **locations** (*tuple*) – The locations ('json', 'querystring', etc.) where to search for the value.

Returns The unvalidated argument value or `missing` if the value cannot be found on the request.

parse_cookies(*req*, *name*, *arg*)
Pull a cookie value from the request or return `missing` if the value cannot be found.

parse_files(*req*, *name*, *arg*)
Pull a file from the request or return `missing` if the value file cannot be found.

parse_form(*req*, *name*, *arg*)
Pull a value from the form data of a request object or return `missing` if the value cannot be found.

parse_headers(*req*, *name*, *arg*)
Pull a value from the headers or return `missing` if the value cannot be found.

parse_json(*req*, *name*, *arg*)
Pull a JSON value from a request object or return `missing` if the value cannot be found.

parse_querystring(*req*, *name*, *arg*)
Pull a value from the query string of a request object or return `missing` if the value cannot be found.

use_args(*argmap*, *req=None*, *locations=None*, *as_kwargs=False*, *validate=None*)
Decorator that injects parsed arguments into a view function or method.

Example usage with Flask:

```
@app.route('/echo', methods=['get', 'post'])
@parser.use_args({'name': fields.Str()})
def greet(args):
    return 'Hello ' + args['name']
```

Parameters

- **argmap** – Either a `marshmallow.Schema`, a `dict` of argname -> `marshmallow.fields.Field` pairs, or a callable which accepts a request and returns a `marshmallow.Schema`.

- **locations** (*tuple*) – Where on the request to search for values.
- **as_kwargs** (*bool*) – Whether to insert arguments as keyword arguments.
- **validate** (*callable*) – Validation function that receives the dictionary of parsed arguments. If the function returns `False`, the parser will raise a *ValidationError*.

use_kwargs(**args*, ***kwargs*)

Decorator that injects parsed arguments into a view function or method as keyword arguments.

This is a shortcut to *use_args()* with `as_kwargs=True`.

Example usage with Flask:

```
@app.route('/echo', methods=['get', 'post'])
@parser.use_kwargs({'name': fields.Str()})
def greet(name):
    return 'Hello ' + name
```

Receives the same `args` and `kwargs` as *use_args()*.

`webargs.core.`**get_value**(*data*, *name*, *field*, *allow_many_nested=False*)

Get a value from a dictionary. Handles `MultiDict` types when `multiple=True`. If the value is not found, return `missing`.

Parameters

- **data** (*object*) – Mapping (e.g. **:type:'dict'**) or list-like instance to pull the value from.
- **name** (*str*) – Name of the key.
- **multiple** (*bool*) – Whether to handle multiple values.
- **allow_many_nested** (*bool*) – Whether to allow a list of nested objects (it is valid only for JSON format, so it is set to True in `parse_json` methods).

4.1.2 webargs.fields

Field classes.

Includes all fields from `marshmallow.fields` in addition to a custom *Nested* field and *DelimitedList*.

All fields can optionally take a special `location` keyword argument, which tells webargs where to parse the request argument from.

```
args = {
    'active': fields.Bool(location='query')
    'content_type': fields.Str(load_from='Content-Type',
                               location='headers')
}
```

class `webargs.fields.`**Nested**(*nested*, **args*, ***kwargs*)

Same as `marshmallow.fields.Nested`, except can be passed a dictionary as the first argument, which will be converted to a `marshmallow.Schema`.

class `webargs.fields.`**DelimitedList**(*cls_or_instance*, *delimiter=None*, *as_string=False*, ***kwargs*)

Same as `marshmallow.fields.List`, except can load from either a list or a delimited string (e.g. "foo,bar,baz").

Parameters

- **cls_or_instance** (*Field*) – A field class or instance.
- **delimiter** (*str*) – Delimiter between values.
- **as_string** (*bool*) – Dump values to string.

4.1.3 webargs.async

Asynchronous request parser. Compatible with Python>=3.4.

class webargs.async.**AsyncParser**(*locations=None*, *error_handler=None*)

Asynchronous variant of *webargs.core.Parser*, where parsing methods may be either coroutines or regular methods.

clear_cache()

Invalidate the parser's cache.

error_handler(*func*)

Decorator that registers a custom error handling function. The function should received the raised error. Overrides the parser's `handle_error` method.

Example:

```
from webargs import core
parser = core.Parser()

class CustomError(Exception):
    pass

@parser.error_handler
def handle_error(error):
    raise CustomError(error)
```

Parameters **func** (*callable*) – The error callback to register.

get_default_request()

Optional override. Provides a hook for frameworks that use thread-local request objects.

get_request_from_view_args(*view*, *args*, *kwargs*)

Optional override. Returns the request object to be parsed, given a view function's args and kwargs.

Used by the *use_args* and *use_kwargs* to get a request object from a view's arguments.

Parameters

- **view** (*callable*) – The view function or method being decorated by *use_args* or *use_kwargs*
- **args** (*tuple*) – Positional arguments passed to `view`.
- **kwargs** (*dict*) – Keyword arguments passed to `view`.

handle_error(*error*)

Called if an error occurs while parsing args. By default, just logs and raises `error`.

location_handler(*name*)

Decorator that registers a function for parsing a request location. The wrapped function receives a request, the name of the argument, and the corresponding `Field` object.

Example:

```
        from webargs import core
        parser = core.Parser()

        @parser.location_handler('name')
        def parse_data(request, name, field):
            return request.data.get(name)
```

Parameters **name** (*str*) – The name of the location to register.

parse(*argmap*, *req=None*, *locations=None*, *validate=None*, *force_all=False*)
Coroutine variant of *webargs.core.Parser*.

Receives the same arguments as *webargs.core.Parser.parse*.

parse_cookies(*req*, *name*, *arg*)
Pull a cookie value from the request or return `missing` if the value cannot be found.

parse_files(*req*, *name*, *arg*)
Pull a file from the request or return `missing` if the value file cannot be found.

parse_form(*req*, *name*, *arg*)
Pull a value from the form data of a request object or return `missing` if the value cannot be found.

parse_headers(*req*, *name*, *arg*)
Pull a value from the headers or return `missing` if the value cannot be found.

parse_json(*req*, *name*, *arg*)
Pull a JSON value from a request object or return `missing` if the value cannot be found.

parse_querystring(*req*, *name*, *arg*)
Pull a value from the query string of a request object or return `missing` if the value cannot be found.

use_args(*argmap*, *req=None*, *locations=None*, *as_kwargs=False*, *validate=None*)
Decorator that injects parsed arguments into a view function or method.

> **Warning:** This will not work with `async def` coroutines. Either use a generator-based coroutine decorated with `asyncio.coroutine` or use the *parse* method.

Receives the same arguments as *webargs.core.Parser.use_args*.

use_kwargs(**args*, ***kwargs*)
Decorator that injects parsed arguments into a view function or method.

> **Warning:** This will not work with `async def` coroutines. Either use a generator-based coroutine decorated with `asyncio.coroutine` or use the *parse* method.

Receives the same arguments as *webargs.core.Parser.use_kwargs*.

4.1.4 webargs.flaskparser

Flask request argument parsing module.

Example:

```
from flask import Flask

from webargs import fields
```

```
from webargs.flaskparser import use_args

app = Flask(__name__)

hello_args = {
    'name': fields.Str(required=True)
}

@app.route('/')
@use_args(hello_args)
def index(args):
    return 'Hello ' + args['name']
```

class webargs.flaskparser.**FlaskParser**(*locations=None*, *error_handler=None*)
Flask request argument parser.

get_default_request()
Override to use Flask's thread-local request objec by default

handle_error(*error*)
Handles errors during parsing. Aborts the current HTTP request and responds with a 422 error.

parse_cookies(*req*, *name*, *field*)
Pull a value from the cookiejar.

parse_files(*req*, *name*, *field*)
Pull a file from the request.

parse_form(*req*, *name*, *field*)
Pull a form value from the request.

parse_headers(*req*, *name*, *field*)
Pull a value from the header data.

parse_json(*req*, *name*, *field*)
Pull a json value from the request.

parse_querystring(*req*, *name*, *field*)
Pull a querystring value from the request.

parse_view_args(*req*, *name*, *field*)
Pull a value from the request's `view_args`.

webargs.flaskparser.**abort**(*http_status_code*, ***kwargs*)
Raise a HTTPException for the given http_status_code. Attach any keyword arguments to the exception for later processing.

From Flask-Restful. See NOTICE file for license information.

4.1.5 webargs.djangoparser

Django request argument parsing.

Example usage:

```
from django.views.generic import View
from django.http import HttpResponse
from marshmallow import fields
from webargs.djangoparser import use_args
```

```
hello_args = {
    'name': fields.Str(missing='World')
}

class MyView(View):

    @use_args(hello_args)
    def get(self, args, request):
        return HttpResponse('Hello ' + args['name'])
```

class webargs.djangoparser.**DjangoParser**(*locations=None*, *error_handler=None*)
Django request argument parser.

> **Warning:** *DjangoParser* does not override *handle_error*, so your Django views are responsible for catching any ValidationErrors raised by the parser and returning the appropriate HTTPResponse.

parse_cookies(*req*, *name*, *field*)
Pull the value from the cookiejar.

parse_files(*req*, *name*, *field*)
Pull a file from the request.

parse_form(*req*, *name*, *field*)
Pull the form value from the request.

parse_json(*req*, *name*, *field*)
Pull a json value from the request body.

parse_querystring(*req*, *name*, *field*)
Pull the querystring value from the request.

4.1.6 webargs.bottleparser

Bottle request argument parsing module.

Example:

```
from bottle import route, run
from marshmallow import fields
from webargs.bottleparser import use_args

hello_args = {
    'name': fields.Str(missing='World')
}
@route('/', method='GET')
@use_args(hello_args)
def index(args):
    return 'Hello ' + args['name']

if __name__ == '__main__':
    run(debug=True)
```

class webargs.bottleparser.**BottleParser**(*locations=None*, *error_handler=None*)
Bottle.py request argument parser.

get_default_request()
Override to use bottle's thread-local request object by default.

handle_error(*error*)
Handles errors during parsing. Aborts the current request with a 400 error.

parse_cookies(*req*, *name*, *field*)
Pull a value from the cookiejar.

parse_files(*req*, *name*, *field*)
Pull a file from the request.

parse_form(*req*, *name*, *field*)
Pull a form value from the request.

parse_headers(*req*, *name*, *field*)
Pull a value from the header data.

parse_json(*req*, *name*, *field*)
Pull a json value from the request.

parse_querystring(*req*, *name*, *field*)
Pull a querystring value from the request.

4.1.7 webargs.tornadoparser

Tornado request argument parsing module.

Example:

```
import tornado.web
from marshmallow import fields
from webargs.tornadoparser import use_args

class HelloHandler(tornado.web.RequestHandler):

    @use_args({'name': fields.Str(missing='World')})
    def get(self, args):
        response = {'message': 'Hello {}'.format(args['name'])}
        self.write(response)
```

exception webargs.tornadoparser.**HTTPError**(**args*, ***kwargs*)
`tornado.web.HTTPError` that stores validation errors.

class webargs.tornadoparser.**TornadoParser**(**args*, ***kwargs*)
Tornado request argument parser.

handle_error(*error*)
Handles errors during parsing. Raises a `tornado.web.HTTPError` with a 400 error.

parse_cookies(*req*, *name*, *field*)
Pull a value from the header data.

parse_files(*req*, *name*, *field*)
Pull a file from the request.

parse_form(*req*, *name*, *field*)
Pull a form value from the request.

parse_headers(*req*, *name*, *field*)
Pull a value from the header data.

parse_json(*req*, *name*, *field*)
Pull a json value from the request.

parse_querystring(*req*, *name*, *field*)
Pull a querystring value from the request.

webargs.tornadoparser.**decode_argument**(*value*, *name=None*)
Decodes an argument from the request.

webargs.tornadoparser.**get_value**(*d*, *name*, *field*)
Handle gets from 'multidicts' made of lists

It handles cases: `{"key": [value]}` and `{"key": value}`

webargs.tornadoparser.**parse_json_body**(*req*)
Return the decoded JSON body from the request.

4.1.8 webargs.pyramidparser

Pyramid request argument parsing.

Example usage:

```
from wsgiref.simple_server import make_server
from pyramid.config import Configurator
from pyramid.response import Response
from marshmallow import fields
from webargs.pyramidparser import use_args

hello_args = {
    'name': fields.Str(missing='World')
}

@use_args(hello_args)
def hello_world(request, args):
    return Response('Hello ' + args['name'])

if __name__ == '__main__':
    config = Configurator()
    config.add_route('hello', '/')
    config.add_view(hello_world, route_name='hello')
    app = config.make_wsgi_app()
    server = make_server('0.0.0.0', 6543, app)
    server.serve_forever()
```

class webargs.pyramidparser.**PyramidParser**(*locations=None*, *error_handler=None*)
Pyramid request argument parser.

handle_error(*error*)
Handles errors during parsing. Aborts the current HTTP request and responds with a 400 error.

parse_cookies(*req*, *name*, *field*)
Pull the value from the cookiejar.

parse_files(*req*, *name*, *field*)
Pull a file from the request.

parse_form(*req*, *name*, *field*)
Pull a form value from the request.

parse_headers(*req*, *name*, *field*)
Pull a value from the header data.

parse_json(*req*, *name*, *field*)
Pull a json value from the request.

parse_matchdict(*req*, *name*, *field*)
Pull a value from the request's `matchdict`.

parse_querystring(*req*, *name*, *field*)
Pull a querystring value from the request.

use_args(*argmap*, *req=None*, *locations=('querystring', 'form', 'json')*, *as_kwargs=False*, *validate=None*)
Decorator that injects parsed arguments into a view callable. Supports the *Class-based View* pattern where `request` is saved as an instance attribute on a view class.

Parameters

- **argmap** (*dict*) – Either a `marshmallow.Schema`, a `dict` of argname -> `marshmallow.fields.Field` pairs, or a callable which accepts a request and returns a `marshmallow.Schema`.
- **req** – The request object to parse. Pulled off of the view by default.
- **locations** (*tuple*) – Where on the request to search for values.
- **as_kwargs** (*bool*) – Whether to insert arguments as keyword arguments.
- **validate** (*callable*) – Validation function that receives the dictionary of parsed arguments. If the function returns `False`, the parser will raise a `ValidationError`.

4.1.9 webargs.webapp2parser

4.1.10 webargs.falconparser

Falcon request argument parsing module.

class `webargs.falconparser.`**FalconParser**(*locations=None*, *error_handler=None*)
Falcon request argument parser.

get_request_from_view_args(*view*, *args*, *kwargs*)
Get request from a resource method's arguments. Assumes that request is the second argument.

handle_error(*error*)
Handles errors during parsing.

parse_cookies(*req*, *name*, *field*)
Pull a cookie value from the request.

parse_form(*req*, *name*, *field*)
Pull a form value from the request.

> **Note:** The request stream will be read and left at EOF.

parse_headers(*req*, *name*, *field*)
Pull a header value from the request.

parse_json(*req*, *name*, *field*)
Pull a JSON body value from the request.

> **Note:** The request stream will be read and left at EOF.

parse_querystring(*req*, *name*, *field*)
Pull a querystring value from the request.

exception webargs.falconparser.**HTTPError**(*status*, *errors*, **args*, ***kwargs*)
HTTPError that stores a dictionary of validation error messages.

to_dict(**args*, ***kwargs*)
Override falcon.HTTPError to include error messages in responses.

4.1.11 webargs.aiohttpparser

aiohttp request argument parsing module.

Example:

```
import asyncio
from aiohttp import web

from webargs import fields
from webargs.aiohttpparser import use_args


hello_args = {
    'name': fields.Str(required=True)
}
@asyncio.coroutine
@use_args(hello_args)
def index(request, args):
    return web.Response(
        body='Hello {}'.format(args['name']).encode('utf-8')
    )

app = web.Application()
app.router.add_route('GET', '/', index)
```

class webargs.aiohttpparser.**AIOHTTPParser**(*locations=None*, *error_handler=None*)
aiohttp request argument parser.

get_request_from_view_args(*view*, *args*, *kwargs*)
Get request object from a handler function or method. Used internally by use_args and use_kwargs.

handle_error(*error*)
Handle ValidationErrors and return a JSON response of error messages to the client.

parse_cookies(*req*, *name*, *field*)
Pull a value from the cookiejar.

parse_form(*req*, *name*, *field*)
Pull a form value from the request.

parse_headers(*req*, *name*, *field*)
Pull a value from the header data.

parse_json(*req*, *name*, *field*)
Pull a json value from the request.

parse_match_info(*req*, *name*, *field*)
Pull a value from the request's `match_info`.

parse_querystring(*req*, *name*, *field*)
Pull a querystring value from the request.

CHAPTER 5

Project Info

5.1 License

```
Copyright 2014-2016 Steven Loria and contributors

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
```

5.2 Changelog

5.2.1 1.3.4 (2016-06-11)

Bug fixes:

- Fix bug in parsing form in Falcon>=1.0.

5.2.2 1.3.3 (2016-05-29)

Bug fixes:

- Fix behavior for nullable List fields (#107). Thanks @shaicantor for reporting.

5.2.3 1.3.2 (2016-04-14)

Bug fixes:

- Fix passing a schema factory to `use_kwargs` (#103). Thanks @ksesong for reporting.

5.2.4 1.3.1 (2016-04-13)

Bug fixes:

- Fix memory leak when calling `parser.parse` with a `dict` in a view (#101). Thanks @frankslaughter for reporting.
- aiohttpparser: Fix bug in handling bulk-type arguments.

Support:

- Massive refactor of tests (#98).
- Docs: Fix incorrect use_args example in Tornado section (#100). Thanks @frankslaughter for reporting.
- Docs: Add "Mixing Locations" section (#90). Thanks @tuukkamustonen.

5.2.5 1.3.0 (2016-04-05)

Features:

- Add bulk-type arguments support for JSON parsing by passing `many=True` to a `Schema` (#81). Thanks @frol.

Bug fixes:

- Fix JSON parsing in Flask<=0.9.0. Thanks @brettdh for the PR.
- Fix behavior of `status_code` argument to `ValidationError` (#85). This requires **marshmallow>=2.7.0**. Thanks @ParthGandhi for reporting.

Support:

- Docs: Add "Custom Fields" section with example of using a `Function` field (#94). Thanks @brettdh for the suggestion.

5.2.6 1.2.0 (2016-01-04)

Features:

- Add `view_args` request location to `FlaskParser` (#82). Thanks @oreza for the suggestion.

Bug fixes:

- Use the value of `load_from` as the key for error messages when it is provided (#83). Thanks @immerrr for the catch and patch.

5.2.7 1.1.1 (2015-11-14)

Bug fixes:

- aiohttpparser: Fix bug that raised a `JSONDecodeError` raised when parsing non-JSON requests using default `locations` (#80). Thanks @leonidumanskiy for reporting.

- Fix parsing JSON requests that have a vendor media type, e.g. `application/vnd.api+json`.

5.2.8 1.1.0 (2015-11-08)

Features:

- `Parser.parse`, `Parser.use_args` and `Parser.use_kwargs` can take a Schema factory as the first argument (#73). Thanks @DamianHeard for the suggestion and the PR.

Support:

- Docs: Add "Custom Parsers" section with example of parsing nested querystring arguments (#74). Thanks @dwieeb.
- Docs: Add "Advanced Usage" page.

5.2.9 1.0.0 (2015-10-19)

Features:

- Add `AIOHTTPParser` (#71).
- Add `webargs.async` module with `AsyncParser`.

Bug fixes:

- If an empty list is passed to a List argument, it will be parsed as an empty list rather than being excluded from the parsed arguments dict (#70). Thanks @mTatcher for catching this.

Other changes:

- *Backwards-incompatible*: When decorating resource methods with `FalconParser.use_args`, the parsed arguments dictionary will be positioned **after** the request and response arguments.
- *Backwards-incompatible*: When decorating views with `DjangoParser.use_args`, the parsed arguments dictionary will be positioned **after** the request argument.
- *Backwards-incompatible*: `Parser.get_request_from_view_args` gets passed a view function as its first argument.
- *Backwards-incompatible*: Remove logging from default error handlers.

5.2.10 0.18.0 (2015-10-04)

Features:

- Add `FalconParser` (#63).
- Add `fields.DelimitedList` (#66). Thanks @jmcarp.
- `TornadoParser` will parse json with `simplejson` if it is installed.
- `BottleParser` caches parsed json per-request for improved performance.

No breaking changes. Yay!

5.2.11 0.17.0 (2015-09-29)

Features:

- `TornadoParser` returns unicode strings rather than bytestrings (#41). Thanks @thomasboyt for the suggestion.
- Add `Parser.get_default_request` and `Parser.get_request_from_view_args` hooks to simplify `Parser` implementations.
- *Backwards-compatible*: `webargs.core.get_value` takes a `Field` as its last argument. Note: this is technically a breaking change, but this won't affect most users since `get_value` is only used internally by `Parser` classes.

Support:

- Add `examples/annotations_example.py` (demonstrates using Python 3 function annotations to define request arguments).
- Fix examples. Thanks @hyunchel for catching an error in the Flask error handling docs.

Bug fixes:

- Correctly pass `validate` and `force_all` params to `PyramidParser.use_args`.

5.2.12 0.16.0 (2015-09-27)

The major change in this release is that webargs now depends on marshmallow for defining arguments and validation.

Your code will need to be updated to use `Fields` rather than `Args`.

```
# Old API
from webargs import Arg

args = {
    'name': Arg(str, required=True)
    'password': Arg(str, validate=lambda p: len(p) >= 6),
    'display_per_page': Arg(int, default=10),
    'nickname': Arg(multiple=True),
    'Content-Type': Arg(dest='content_type', location='headers'),
    'location': Arg({
        'city': Arg(str),
        'state': Arg(str)
    })
    'meta': Arg(dict),
}

# New API
from webargs import fields

args = {
    'name': fields.Str(required=True)
    'password': fields.Str(validate=lambda p: len(p) >= 6),
    'display_per_page': fields.Int(missing=10),
    'nickname': fields.List(fields.Str()),
    'content_type': fields.Str(load_from='Content-Type'),
    'location': fields.Nested({
        'city': fields.Str(),
        'state': fields.Str()
    }),
```

```
    'meta': fields.Dict(),
}
```

Features:

- Error messages for all arguments are "bundled" (#58).

Changes:

- *Backwards-incompatible*: Replace `Args` with marshmallow fields (#61).
- *Backwards-incompatible*: When using `use_kwargs`, missing arguments will have the special value `missing` rather than `None`.
- `TornadoParser` raises a custom `HTTPError` with a `messages` attribute when validation fails.

Bug fixes:

- Fix required validation of nested arguments (#39, #51). These are fixed by virtue of using marshmallow's `Nested` field. Thanks @ewang and @chavz for reporting.

Support:

- Updated docs.
- Add `examples/schema_example.py`.
- Tested against Python 3.5.

5.2.13 0.15.0 (2015-08-22)

Changes:

- If a parsed argument is `None`, the type conversion function is not called #54. Thanks @marcellarius.

Bug fixes:

- Fix parsing nested `Args` when the argument is missing from the input (#52). Thanks @stas.

5.2.14 0.14.0 (2015-06-28)

Features:

- Add parsing of `matchdict` to `PyramidParser`. Thanks @hartror.

Bug fixes:

- Fix `PyramidParser's use_kwargs` method (#42). Thanks @hartror for the catch and patch.
- Correctly use locations passed to Parser's constructor when using `use_args` (#44). Thanks @jacebrowning for the catch and patch.
- Fix behavior of `default` and `dest` argument on nested `Args` (#40 and #46). Thanks @stas.

Changes:

- A 422 response is returned to the client when a `ValidationError` is raised by a parser (#38).

5.2.15 0.13.0 (2015-04-05)

Features:

- Support for webapp2 via the `webargs.webapp2parser` module. Thanks @Trii.
- Store argument name on `RequiredArgMissingError`. Thanks @stas.
- Allow error messages for required validation to be overriden. Thanks again @stas.

Removals:

- Remove `source` parameter from `Arg`.

5.2.16 0.12.0 (2015-03-22)

Features:

- Store argument name on `ValidationError` (#32). Thanks @alexmic for the suggestion. Thanks @stas for the patch.
- Allow nesting of dict subtypes.

5.2.17 0.11.0 (2015-03-01)

Changes:

- Add `dest` parameter to `Arg` constructor which determines the key to be added to the parsed arguments dictionary (#32).
- *Backwards-incompatible*: Rename `targets` parameter to `locations` in `Parser` constructor, `Parser#parse_arg`, `Parser#parse`, `Parser#use_args`, and `Parser#use_kwargs`.
- *Backwards-incompatible*: Rename `Parser#target_handler` to `Parser#location_handler`.

Deprecation:

- The `source` parameter is deprecated in favor of the `dest` parameter.

Bug fixes:

- Fix `validate` parameter of `DjangoParser#use_args`.

5.2.18 0.10.0 (2014-12-23)

- When parsing a nested `Arg`, filter out extra arguments that are not part of the `Arg's` nested `dict` (#28). Thanks Derrick Gilland for the suggestion.
- Fix bug in parsing `Args` with both type coercion and `multiple=True` (#30). Thanks Steven Manuatu for reporting.
- Raise `RequiredArgMissingError` when a required argument is missing on a request.

5.2.19 0.9.1 (2014-12-11)

- Fix behavior of `multiple=True` when nesting Args (#29). Thanks Derrick Gilland for reporting.

5.2.20 0.9.0 (2014-12-08)

- Pyramid support thanks to @philtay.
- User-friendly error messages when `Arg` type conversion/validation fails. Thanks Andriy Yurchuk.
- Allow `use` argument to be a list of functions.
- Allow `Args` to be nested within each other, e.g. for nested dict validation. Thanks @saritasa for the suggestion.
- *Backwards-incompatible*: Parser will only pass `ValidationErrors` to its error handler function, rather than catching all generic Exceptions.
- *Backwards-incompatible*: Rename `Parser.TARGET_MAP` to `Parser.__target_map__`.
- Add a short-lived cache to the `Parser` class that can be used to store processed request data for reuse.
- Docs: Add example usage with Flask-RESTful.

5.2.21 0.8.1 (2014-10-28)

- Fix bug in `TornadoParser` that raised an error when request body is not a string (e.g when it is a `Future`). Thanks Josh Carp.

5.2.22 0.8.0 (2014-10-26)

- Fix `Parser.use_kwargs` behavior when an `Arg` is allowed missing. The `allow_missing` attribute is ignored when `use_kwargs` is called.
- `default` may be a callable.
- Allow `ValidationError` to specify a HTTP status code for the error response.
- Improved error logging.
- Add `'query'` as a valid target name.
- Allow a list of validators to be passed to an `Arg` or `Parser.parse`.
- A more useful `__repr__` for `Arg`.
- Add examples and updated docs.

5.2.23 0.7.0 (2014-10-18)

- Add `source` parameter to `Arg` constructor. Allows renaming of keys in the parsed arguments dictionary. Thanks Josh Carp.
- `FlaskParser's handle_error` method attaches the string representation of validation errors on `err.data['message']`. The raised exception is stored on `err.data['exc']`.
- Additional keyword arguments passed to `Arg` are stored as metadata.

5.2.24 0.6.2 (2014-10-05)

- Fix bug in `TornadoParser's handle_error` method. Thanks Josh Carp.
- Add `error` parameter to `Parser` constructor that allows a custom error message to be used if schema-level validation fails.

- Fix bug that raised a `UnicodeEncodeError` on Python 2 when an Arg's validator function received non-ASCII input.

5.2.25 0.6.1 (2014-09-28)

- Fix regression with parsing an `Arg` with both `default` and `target` set (see issue #11).

5.2.26 0.6.0 (2014-09-23)

- Add `validate` parameter to `Parser.parse` and `Parser.use_args`. Allows validation of the full parsed output.
- If `allow_missing` is `True` on an `Arg` for which `None` is explicitly passed, the value will still be present in the parsed arguments dictionary.
- *Backwards-incompatible*: `Parser's parse_*` methods return `webargs.core.Missing` if the value cannot be found on the request. NOTE: `webargs.core.Missing` will *not* show up in the final output of `Parser.parse`.
- Fix bug with parsing empty request bodies with `TornadoParser`.

5.2.27 0.5.1 (2014-08-30)

- Fix behavior of `Arg's allow_missing` parameter when `multiple=True`.
- Fix bug in tornadoparser that caused parsing JSON arguments to fail.

5.2.28 0.5.0 (2014-07-27)

- Fix JSON parsing in Flask parser when Content-Type header contains more than just `application/json`. Thanks Samir Uppaluru for reporting.
- *Backwards-incompatible*: The `use` parameter to `Arg` is called before type conversion occurs. Thanks Eric Wang for the suggestion.
- Tested on Tornado>=4.0.

5.2.29 0.4.0 (2014-05-04)

- Custom target handlers can be defined using the `Parser.target_handler` decorator.
- Error handler can be specified using the `Parser.error_handler` decorator.
- `Args` can define their request target by passing in a `target` argument.
- *Backwards-incompatible*: `DEFAULT_TARGETS` is now a class member of `Parser`. This allows subclasses to override it.

5.2.30 0.3.4 (2014-04-27)

- Fix bug that caused `use_args` to fail on class-based views in Flask.
- Add `allow_missing` parameter to `Arg`.

5.2.31 0.3.3 (2014-03-20)

- Awesome contributions from the open-source community!
- Add `use_kwargs` decorator. Thanks @venuatu.
- Tornado support thanks to @jvrsantacruz.
- Tested on Python 3.4.

5.2.32 0.3.2 (2014-03-04)

- Fix bug with parsing JSON in Flask and Bottle.

5.2.33 0.3.1 (2014-03-03)

- Remove print statements in core.py. Oops.

5.2.34 0.3.0 (2014-03-02)

- Add support for repeated parameters (#1).
- *Backwards-incompatible*: All `parse_*` methods take `arg` as their fourth argument.
- Add `error_handler` param to `Parser`.

5.2.35 0.2.0 (2014-02-26)

- Bottle support.
- Add `targets` param to `Parser`. Allows setting default targets.
- Add `files` target.

5.2.36 0.1.0 (2014-02-16)

- First release.
- Parses JSON, querystring, forms, headers, and cookies.
- Support for Flask and Django.

5.3 Authors

5.3.1 Lead

- Steven Loria <sloria1@gmail.com>

5.3.2 Contributors (chronological)

- @venuatu <https://github.com/venuatu>
- Javier Santacruz @jvrsantacruz <javier.santacruz.lc@gmail.com>
- Josh Carp <https://github.com/jmcarp>
- @philtay <https://github.com/philtay>
- Andriy Yurchuk <https://github.com/Ch00k>
- Stas Sucov <https://github.com/stas>
- Josh Johnston <https://github.com/Trii>
- Rory Hart <https://github.com/hartror>
- Jace Browning <https://github.com/jacebrowning>
- @marcellarius <https://github.com/marcellarius>
- Damian Heard <https://github.com/DamianHeard>
- Daniel Imhoff <https://github.com/dwieeb>
- immerrr <https://github.com/immerrr>
- Brett Higgins <https://github.com/brettdh>
- Vlad Frolov <https://github.com/frol>
- Tuukka Mustonen <https://github.com/tuukkamustonen>

5.4 Contributing Guidelines

5.4.1 In General

- PEP 8, when sensible.
- Test ruthlessly. Write docs for new features.
- Even more important than Test-Driven Development–*Human-Driven Development*.

5.4.2 In Particular

Questions, Feature Requests, Bug Reports, and Feedback. . .

. . .should all be reported on the Github Issue Tracker .

Setting Up for Local Development

1. Fork webargs on Github.

```
$ git clone https://github.com/sloria/webargs.git
$ cd webargs
```

2. Install development requirements. It is highly recommended that you use a virtualenv.

```
    # After activating your virtualenv
    $ pip install -r dev-requirements.txt
```

3. Install webargs in develop mode.

```
    $ pip install -e .
```

Git Branch Structure

Webargs abides by the following branching model:

`dev` Current development branch. **New features should branch off here**.

`pypi` Current production release on PyPI.

`X.Y-line` Maintenance branch for release `X.Y`. **Bug fixes should be sent to the most recent release branch.**. The maintainer will forward-port the fix to `dev`. Note: exceptions may be made for bug fixes that introduce large code changes.

Always make a new branch for your work, no matter how small. Also, **do not put unrelated changes in the same branch or pull request**. This makes it more difficult to merge your changes.

Pull Requests

1. Create a new local branch.

```
# For a new feature
$ git checkout -b name-of-feature dev

# For a bugfix
$ git checkout -b fix-something 1.2-line
```

2. Commit your changes. Write good commit messages.

```
$ git commit -m "Detailed commit message"
$ git push origin name-of-feature
```

3. Before submitting a pull request, check the following:
   - If the pull request adds functionality, it is tested and the docs are updated.
   - You've added yourself to `AUTHORS.rst`.
4. Submit a pull request to `sloria:dev` or the appropriate maintenance branch. The Travis CI build must be passing before your pull request is merged.

Running Tests

To run all tests:

```
$ invoke test
```

To run tests on Python 2.6, 2.7, 3.3, and 3.4 virtual environments (must have each interpreter installed):

```
$ tox
```

Documentation

Contributions to the documentation are welcome. Documentation is written in reStructured Text (rST). A quick rST reference can be found here. Builds are powered by Sphinx.

To install the packages for building the docs:

```
$ pip install -r docs/requirements.txt
```

To build the docs:

```
$ invoke docs -b
```

The `-b` (for "browse") automatically opens up the docs in your browser after building.

Contributing Examples

Have a usage example you'd like to share? Feel free to add it to the examples directory and send a pull request.

W

T

U

V

W