webargs

Release 8.5.0

unknown

Apr 25, 2024

CONTENTS

Release v8.5.0. (*Changelog*)

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

CHAPTER

ONE

UPGRADING FROM AN OLDER VERSION?

See the *Upgrading to Newer Releases* page for notes on getting your code up-to-date with the latest version.

CHAPTER

TWO

USAGE AND SIMPLE EXAMPLES

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

app = Flask(__name__)


@app.route("/")
@use_args({"name": fields.Str(required=True)}, location="query")
def index(args):
    return "Hello " + args["name"]


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

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

By default Webargs will automatically parse JSON request bodies. But it also has support for:

Query Parameters

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

# pass location="query" to use_args
```

Form Data

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

# pass location="form" to use_args
```

JSON Data

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

(continues on next page)

(continued from previous page)

```
# pass location="json" (or omit location) to use_args
```

and, optionally:

- Headers
- Cookies
- Files
- Paths

CHAPTER

THREE

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

FOUR

GET IT NOW

```
pip install -U webargs
```

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

CHAPTER

FIVE

USER GUIDE

5.1 Install

webargs requires Python >= 3.8. It depends on marshmallow >= 3.0.0.

5.1.1 From the PyPI

To install the latest version from the PyPI:

```
$ pip install -U webargs
```

5.1.2 Get the Bleeding Edge Version

To get the latest development version of webargs, run

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

5.2 Quickstart

5.2.1 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(load_default=10),
    # Repeated parameter, e.g. "/?nickname=Fred&nickname=Freddie"
    "nickname": fields.List(fields.Str()),
```

(continues on next page)

(continued from previous page)

```
    # Delimited list, e.g. "/?languages=python,javascript"
    "languages": fields.DelimitedList(fields.Str()),
    # When value is keyed on a variable-unsafe name
    # or you want to rename a key
    "user_type": fields.Str(data_key="user-type"),
}
```

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"],
    )
```

5.2.2 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 will be omitted from the arguments. Use `**kwargs` to handle optional arguments.

```
from webargs import fields, missing


@use_kwargs({"name": fields.Str(required=True), "nickname": fields.Str(required=False)})
def myview(name, **kwargs):
    if "nickname" not in kwargs:
        # ...
        pass
```

5.2.3 Request "Locations"

By default, webargs will search for arguments from the request body as JSON. You can specify a different location from which to load data like so:

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

Available locations include:

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

5.2.4 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")
```

(continues on next page)

(continued from previous page)

```
args = {"id": fields.Int(validate=must_exist_in_db)}
```

Note: If a validator returns `None`, validation will pass. A validator must return `False` or raise a *ValidationError* for validation to fail.

There are a number of built-in validators from `marshmallow.validate` (re-exported as `webargs.validate`).

```
from webargs import fields, validate

args = {
    "name": fields.Str(required=True, validate=[validate.Length(min=1, max=9999)]),
    "age": fields.Int(validate=[validate.Range(min=1, max=999)]),
}
```

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"]
)
```

5.2.5 Error Handling

Each parser has a default error handling method. To override the error handling callback, write a function that receives an error, the request, the `marshmallow.Schema` instance, status code, and headers. Then decorate that function with *Parser.error_handler*.

```
from webargs.flaskparser import parser


class CustomError(Exception):
    pass


@parser.error_handler
def handle_error(error, req, schema, *, error_status_code, error_headers):
    raise CustomError(error.messages)
```

5.2.6 Parsing Lists in Query Strings

Use *fields.DelimitedList* to parse comma-separated lists in query parameters, e.g. `/?permissions=read,write`

```
from webargs import fields

args = {"permissions": fields.DelimitedList(fields.Str())}
```

If you expect repeated query parameters, e.g. `/?repo=webargs&repo=marshmallow`, use `fields.List` instead.

```
from webargs import fields

args = {"repo": fields.List(fields.Str())}
```

5.2.7 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: Of the default supported locations in webargs, only the `json` request location supports nested datastructures. You can, however, *implement your own data loader* to add nested field functionality to the other locations.

5.2.8 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.

5.3 Advanced Usage

This section includes guides for advanced usage patterns.

5.3.1 Custom Location Handlers

To add your own custom location handler, write a function that receives a request, and a `Schema`, then decorate that function with *`Parser.location_loader`*.

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


@parser.location_loader("data")
def load_data(request, schema):
    return request.data


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

Note: The schema is passed so that it can be used to wrap multidict types and unpack List fields correctly. If you are writing a loader for a multidict type, consider looking at *`MultiDictProxy`* for an example of how to do this.

"meta" Locations

You can define your own locations which mix data from several existing locations.

The `json_or_form` location does this – first trying to load data as JSON and then falling back to a form body – and its implementation is quite simple:

```
def load_json_or_form(self, req, schema):
    """Load data from a request, accepting either JSON or form-encoded
    data.

    The data will first be loaded as JSON, and, if that fails, it will be
    loaded as a form post.
    """
    data = self.load_json(req, schema)
    if data is not missing:
        return data
    return self.load_form(req, schema)
```

You can imagine your own locations with custom behaviors like this. For example, to mix query parameters and form body data, you might write the following:

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


@parser.location_loader("query_and_form")
def load_data(request, schema):
    # relies on the Flask (werkzeug) MultiDict type's implementation of
```

(continues on next page)

(continued from previous page)

```
    # these methods, but when you're extending webargs, you may know things
    # about your framework of choice
    newdata = request.args.copy()
    newdata.update(request.form)
    return MultiDictProxy(newdata, schema)


# Now 'query_and_form' means you can send these values in either location,
# and they will be *mixed* together into a new dict to pass to your schema
@parser.use_args({"favorite_food": fields.String()}, location="query_and_form")
def set_favorite_food(args):
    ...  # do stuff
    return "your favorite food is now set to {}".format(args["favorite_food"])
```

5.3.2 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(load_default="")
    last_name = fields.Str(load_default="")
    date_registered = fields.DateTime(dump_only=True)


@use_args(UserSchema())
def profile_view(args):
    username = args["username"]
    # ...


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


# You can add additional parameters
@use_kwargs({"posts_per_page": fields.Int(load_default=10)}, location="query")
@use_args(UserSchema())
def profile_posts(args, posts_per_page):
    username = args["username"]
    # ...
```

5.3.3 Setting unknown

webargs supports several ways of setting and passing the unknown parameter for handling unknown fields.

You can pass unknown=... as a parameter to any of *Parser.parse*, *Parser.use_args*, and *Parser.use_kwargs*.

Note: The unknown value is passed to the schema's `load()` call. It therefore only applies to the top layer when nesting is used. To control unknown at multiple layers of a nested schema, you must use other mechanisms, like the unknown argument to `fields.Nested`.

Default unknown

By default, webargs will pass unknown=marshmallow.EXCLUDE except when the location is `json`, `form`, `json_or_form`, or `path`. In those cases, it uses unknown=marshmallow.RAISE instead.

You can change these defaults by overriding `DEFAULT_UNKNOWN_BY_LOCATION`. This is a mapping of locations to values to pass.

For example,

```
from flask import Flask
from marshmallow import EXCLUDE, fields
from webargs.flaskparser import FlaskParser

app = Flask(__name__)


class Parser(FlaskParser):
    DEFAULT_UNKNOWN_BY_LOCATION = {"query": EXCLUDE}


parser = Parser()


# location is "query", which is listed in DEFAULT_UNKNOWN_BY_LOCATION,
# so EXCLUDE will be used
@app.route("/", methods=["GET"])
@parser.use_args({"foo": fields.Int()}, location="query")
def get(args):
    return f"foo x 2 = {args['foo'] * 2}"


# location is "json", which is not in DEFAULT_UNKNOWN_BY_LOCATION,
# so no value will be passed for `unknown`
@app.route("/", methods=["POST"])
@parser.use_args({"foo": fields.Int(), "bar": fields.Int()}, location="json")
def post(args):
    return f"foo x bar = {args['foo'] * args['bar']}"
```

You can also define a default at parser instantiation, which will take precedence over these defaults, as in

```
from marshmallow import INCLUDE
```

(continues on next page)

(continued from previous page)

```
parser = Parser(unknown=INCLUDE)


# because `unknown` is set on the parser, `DEFAULT_UNKNOWN_BY_LOCATION` has
# effect and `INCLUDE` will always be used
@app.route("/", methods=["POST"])
@parser.use_args({"foo": fields.Int(), "bar": fields.Int()}, location="json")
def post(args):
    unexpected_args = [k for k in args.keys() if k not in ("foo", "bar")]
    return f"foo x bar = {args['foo'] * args['bar']}; unexpected args={unexpected_args}"
```

Using Schema-Specfied `unknown`

If you wish to use the value of `unknown` specified by a schema, simply pass `unknown=None`. This will disable webargs' automatic passing of values for `unknown`. For example,

```
from flask import Flask
from marshmallow import Schema, fields, EXCLUDE, missing
from webargs.flaskparser import use_args


class RectangleSchema(Schema):
    length = fields.Float()
    width = fields.Float()

    class Meta:
        unknown = EXCLUDE


app = Flask(__name__)


# because unknown=None was passed, no value is passed during schema loading
# as a result, the schema's behavior (EXCLUDE) is used
@app.route("/", methods=["POST"])
@use_args(RectangleSchema(), location="json", unknown=None)
def get(args):
    return f"area = {args['length'] * args['width']}"
```

You can also set `unknown=None` when instantiating a parser to make this behavior the default for a parser.

5.3.4 When to avoid `use_kwargs`

Any `Schema` passed to *`use_kwargs`* MUST deserialize to a dictionary of data. If your schema has a `post_load` method that returns a non-dictionary, you should use *`use_args`* instead.

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

(continues on next page)

(continued from previous page)

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width


class RectangleSchema(Schema):
    length = fields.Float()
    width = fields.Float()

    @post_load
    def make_object(self, data, **kwargs):
        return Rectangle(**data)


@use_args(RectangleSchema)
def post(rect: Rectangle):
    return f"Area: {rect.length * rect.width}"
```

Packages such as marshmallow-sqlalchemy and marshmallow-dataclass generate schemas that deserialize to non-dictionary objects. Therefore, *use_args* should be used with those schemas.

5.3.5 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 flask import Flask
from marshmallow import Schema, fields
from webargs.flaskparser import use_args

app = Flask(__name__)


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


def make_user_schema(request):
    # Filter based on 'fields' query parameter
    fields = request.args.get("fields", None)
    only = fields.split(",") if fields else None
```

(continues on next page)

(continued from previous page)

```
    # 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
@app.route("/profile/", methods=["GET", "POST", "PATCH"])
@use_args(make_user_schema)
def profile_view(args):
    username = args.get("username")
    # ...
```

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"
        return schema_cls(
            only=only, partial=partial, 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):
    # ...
    get_profile(**args)
```

5.3.6 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"]
    # ...
```

5.3.7 Custom Parsers

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

```
import re

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 load_querystring(self, req, schema):
        return _structure_dict(req.args)


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 = {}
```

(continues on next page)

(continued from previous page)

```
    for k, v in dict_.items():
        structure_dict_pair(r, k, v)
    return r
```

5.3.8 Parser pre_load

Similar to `@pre_load` decorated hooks on marshmallow Schemas, *Parser* classes define a method, *pre_load* which can be overridden to provide per-parser transformations of data. The only way to make use of *pre_load* is to subclass a *Parser* and provide an implementation.

pre_load is given the data fetched from a location, the schema which will be used, the request object, and the location name which was requested. For example, to define a `FlaskParser` which strips whitespace from `form` and `query` data, one could write the following:

```
from webargs.flaskparser import FlaskParser
import typing


def _strip_whitespace(value):
    if isinstance(value, str):
        value = value.strip()
    elif isinstance(value, typing.Mapping):
        return {k: _strip_whitespace(value[k]) for k in value}
    elif isinstance(value, (list, tuple)):
        return type(value)(map(_strip_whitespace, value))
    return value


class WhitspaceStrippingFlaskParser(FlaskParser):
    def pre_load(self, location_data, *, schema, req, location):
        if location in ("query", "form"):
            return _strip_whitespace(location_data)
        return location_data
```

Note that *Parser.pre_load* is run after location loading but before `Schema.load` is called. It can therefore be called on multiple types of mapping objects, including `MultiDictProxy`, depending on what the location loader returns.

5.3.9 Returning HTTP 400 Responses

If you'd prefer validation errors to return status code `400` instead of `422`, you can override `DEFAULT_VALIDATION_STATUS` on a *Parser*.

Subclass the parser for your framework to do so. For example, using Falcon:

```
from webargs.falconparser import FalconParser


class Parser(FalconParser):
    DEFAULT_VALIDATION_STATUS = 400
```

(continues on next page)

(continued from previous page)

```
parser = Parser()
use_args = parser.use_args
use_kwargs = parser.use_kwargs
```

5.3.10 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)


@app.route("/profile/", methods=["patch"])
@use_args(PatchSchema(many=True))
def patch_blog(args):
    """Implements JSON Patch for the user profile

    Example JSON body:

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

5.3.11 Multi-Field Detection

If a `List` field is used to parse data from a location like query parameters – where one or multiple values can be passed for a single parameter name – then webargs will automatically treat that field as a list and parse multiple values if present.

To implement this behavior, webargs will examine schemas for `marshmallow.fields.List` fields. `List` fields get unpacked to list values when data is loaded, and other fields do not. This also applies to fields which inherit from `List`.

Note: In webargs v8, `Tuple` will be treated this way as well, in addition to `List`.

What if you have a list which should be treated as a "multi-field" but which does not inherit from `List`? webargs offers two solutions. You can add the custom attribute `is_multiple=True` to your field or you can add your class to your

parser's list of `KNOWN_MULTI_FIELDS`.

First, let's define a "multiplexing field" which takes a string or list of strings to serve as an example:

```
# a custom field class which can accept values like List(String()) or String()
class CustomMultiplexingField(fields.String):
    def _deserialize(self, value, attr, data, **kwargs):
        if isinstance(value, str):
            return super()._deserialize(value, attr, data, **kwargs)
        return [
            self._deserialize(v, attr, data, **kwargs)
            for v in value
            if isinstance(v, str)
        ]

    def _serialize(self, value, attr, **kwargs):
        if isinstance(value, str):
            return super()._serialize(value, attr, **kwargs)
        return [self._serialize(v, attr, **kwargs) for v in value if isinstance(v, str)]
```

If you control the definition of `CustomMultiplexingField`, you can just add `is_multiple=True` to it:

```
# option 1: define the field with is_multiple = True
from webargs.flaskparser import parser


class CustomMultiplexingField(fields.Field):
    is_multiple = True  # <----- this marks this as a multi-field

    ...  # as above
```

If you don't control the definition of `CustomMultiplexingField`, for example because it comes from a library, you can add it to the list of known multifields:

```
# option 2: add the field to the parer's list of multi-fields
class MyParser(FlaskParser):
    KNOWN_MULTI_FIELDS = list(FlaskParser.KNOWN_MULTI_FIELDS) + [
        CustomMultiplexingField
    ]


parser = MyParser()
```

In either case, the end result is that you can use the multifield and it will be detected as a list when unpacking query string data:

```
# gracefully handles
#   ...?foo=a
#   ...?foo=a&foo=b
# and treats them as ["a"] and ["a", "b"] respectively
@parser.use_args({"foo": CustomMultiplexingField()}, location="query")
def show_foos(foo):
    ...
```

5.3.12 Mixing Locations

Arguments for different locations can be specified by passing `location` to each *use_args* call:

```
# "json" is the default, used explicitly below
@app.route("/stacked", methods=["POST"])
@use_args({"page": fields.Int(), "q": fields.Str()}, location="query")
@use_args({"name": fields.Str()}, location="json")
def viewfunc(query_parsed, json_parsed):
    page = query_parsed["page"]
    name = json_parsed["name"]
    # ...
```

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

```
import functools

query = functools.partial(use_args, location="query")
body = functools.partial(use_args, location="json")


@query({"page": fields.Int(), "q": fields.Int()})
@body({"name": fields.Str()})
def viewfunc(query_parsed, json_parsed):
    page = query_parsed["page"]
    name = json_parsed["name"]
    # ...
```

5.3.13 Argument Passing and `arg_name`

Note: This section describes behaviors which are planned to change in `webargs` version 9. In version 8, behavior will be as follows. In version 9, `USE_ARGS_POSITIONAL` will be removed and will always be `False`.

By default, `webargs` provides two ways of passing arguments via decorators, *Parser.use_args*, and *Parser.use_kwargs*. `use_args` passes parsed arguments as positionals, and `use_kwargs` expands dict-like parsed arguments into keyword arguments.

For `use_args`, the result is that sometimes it is non-obvious which order arguments will be passed in. Consider the following nearly identical example snippets:

```
# correct ordering, top-to-bottom
@use_args({"foo": fields.Int(), "bar": fields.Str()}, location="query")
@use_args({"baz": fields.Str()}, location="json")
def viewfunc(query_args, json_args):
    ...


# incorrect ordering, bottom-to-top
@use_args({"foo": fields.Int(), "bar": fields.Str()}, location="query")
@use_args({"baz": fields.Str()}, location="json")
def viewfunc(json_args, query_args):
    ...
```

To resolve this ambiguity, `webargs` version 9 will pass arguments from `use_args` as keyword arguments. You can opt-in to this behavior today by setting `USE_ARGS_POSITIONAL = False` on a parser class. This will cause webargs to pass arguments named `{location}_args` for each location used. For example,

```
from webargs.flaskparser import FlaskParser
from flask import Flask


class KeywordOnlyParser(FlaskParser):
    USE_ARGS_POSITIONAL = False


app = Flask(__name__)
parser = KeywordOnlyParser()


@app.route("/")
@parser.use_args({"foo": fields.Int(), "bar": fields.Str()}, location="query")
@parser.use_args({"baz": fields.Str()}, location="json")
def myview(*, query_args, json_args):
    ...
```

You can also customize the names of passed arguments using the `arg_name` parameter:

```
@app.route("/")
@parser.use_args(
    {"foo": fields.Int(), "bar": fields.Str()}, location="query", arg_name="query"
)
@parser.use_args({"baz": fields.Str()}, location="json", arg_name="payload")
def myview(*, query, payload):
    ...
```

Note that `arg_name` is available even on parsers where `USE_ARGS_POSITIONAL` is not set.

Using an Alternate Argument Name Convention

As described above, the default naming convention for `use_args` arguments is `{location}_args`. You can customize this by creating a parser class and overriding the `get_default_arg_name` method.

`get_default_arg_name` takes the `location` and the `schema` as arguments. The default implementation is:

```
def get_default_arg_name(self, location, schema):
    return f"{location}_args"
```

You can customize this to set different arg names. For example,

```
from webargs.flaskparser import FlaskParser


class MyParser(FlaskParser):
    USE_ARGS_POSITIONAL = False

    def get_default_arg_name(self, location, schema):
        if location in ("json", "form", "json_or_form"):
```

(continues on next page)

(continued from previous page)

```
            return "body"
        elif location in ("query", "querystring"):
            return "query"
        return location


@app.route("/")
@parser.use_args({"foo": fields.Int(), "bar": fields.Str()}, location="query")
@parser.use_args({"baz": fields.Str()}, location="json")
def myview(*, query, body):
    ...
```

Additionally, this makes it possible to make custom schema classes which provide an argument name. For example,

```
from marshmallow import Schema
from webargs.flaskparser import FlaskParser


class RectangleSchema(Schema):
    webargs_arg_name = "rectangle"

    length = fields.Float()
    width = fields.Float()


class MyParser(FlaskParser):
    USE_ARGS_POSITIONAL = False

    def get_default_arg_name(self, location, schema):
        if hasattr(schema, "webargs_arg_name"):
            if isinstance(schema.webargs_arg_name, str):
                return schema.webargs_arg_name
        return super().get_default_arg_name(location, schema)


@app.route("/")
@parser.use_args({"foo": fields.Int(), "bar": fields.Str()}, location="query")
@parser.use_args(RectangleSchema, location="json")
def myview(*, rectangle, query_args):
    ...
```

5.3.14 Next Steps

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

5.4 Framework Support

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

5.4.1 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()}, location="query")
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 `messages` attribute of the attached `ValidationError`.

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

```
from flask import jsonify


# Return validation errors as JSON
@app.errorhandler(422)
@app.errorhandler(400)
def handle_error(err):
    headers = err.data.get("headers", None)
    messages = err.data.get("messages", ["Invalid request."])
    if headers:
        return jsonify({"errors": messages}), err.code, headers
    else:
        return jsonify({"errors": messages}), err.code
```

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"])
```

5.4.2 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, location="form")
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, location="query")
    def get(self, request, args):
```

(continues on next page)

(continued from previous page)

```
        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, json

argmap = {"name": fields.Str(required=True)}


def index(request):
    try:
        args = parser.parse(argmap, request)
    except ValidationError as err:
        return JsonResponse(err.messages, status=422)
    except json.JSONDecodeError:
        return JsonResponse({"json": ["Invalid JSON body."]}, status=400)
    return JsonResponse({"message": "Hello {name}".format(name=name)})
```

5.4.3 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)
```

(continues on next page)

(continued from previous page)

```
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, exc, traceback = kwargs["exc_info"]
            if hasattr(exc, "messages"):
                self.write({"errors": exc.messages})
                if getattr(exc, "headers", None):
                    for name, val in exc.headers.items():
                        self.set_header(name, val)
                self.finish()
```

5.4.4 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({"uid": fields.Str(), "per_page": fields.Int()}, location="query")
def user_detail(request, args):
    uid = args["uid"]
    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


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

5.4.5 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
```

(continues on next page)

(continued from previous page)

```
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, resource, 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")
        # ...
```

5.4.6 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
```

(continues on next page)

(continued from previous page)

```
handler_args = {"name": fields.Str(load_default="World")}


async def handler(request):
    args = await 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


@use_args({"content": fields.Str(required=True)})
async 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 work with both `async def` coroutines and generator-based coroutines decorated with `asyncio.coroutine`.

```
import asyncio

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

hello_args = {"name": fields.Str(load_default="World")}

# The following are equivalent


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

(continues on next page)

(continued from previous page)

```
@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)
```

5.4.7 Bottle

Bottle support is available via the *webargs.bottleparser* module.

Decorator Usage

The preferred way to apply decorators to Bottle routes is using the `apply` argument.

```
from bottle import route

user_args = {"name": fields.Str(load_default="Friend")}


@route("/users/<_id:int>", method="GET", apply=use_args(user_args))
def users(args, _id):
    """A welcome page."""
    return {"message": "Welcome, {}!".format(args["name"]), "_id": _id}
```

5.5 Ecosystem

A list of webargs-related libraries can be found at the GitHub wiki here:

https://github.com/marshmallow-code/webargs/wiki/Ecosystem

CHAPTER

SIX

API REFERENCE

6.1 API

6.1.1 webargs.core

class webargs.core.**Parser**(*location: str | None = None*, ***, *unknown: str | None = '_default'*, *error_handler: Callable[[...], NoReturn] | None = None*, *schema_class: type[Schema] | None = None*)

Base parser class that provides high-level implementation for parsing a request.

Descendant classes must provide lower-level implementations for reading data from different locations, e.g. `load_json`, `load_querystring`, etc.

Parameters

- **location** (*str*) – Default location to use for data
- **unknown** (*str*) – A default value to pass for `unknown` when calling the schema's `load` method. Defaults to `marshmallow.EXCLUDE` for non-body locations and `marshmallow.RAISE` for request bodies. Pass `None` to use the schema's setting instead.
- **error_handler** (*callable*) – Custom error handler function.

DEFAULT_LOCATION: str = 'json'

Default location to check for data

DEFAULT_SCHEMA_CLASS

The marshmallow Schema class to use when creating new schemas

alias of Schema

DEFAULT_VALIDATION_MESSAGE: str = 'Invalid value.'

Default error message for validation errors

DEFAULT_VALIDATION_STATUS: int = 422

Default status code to return for validation errors

KNOWN_MULTI_FIELDS: list[type] = [<class 'marshmallow.fields.List'>, <class 'marshmallow.fields.Tuple'>]

field types which should always be treated as if they set `is_multiple=True`

async async_parse(*argmap: Schema | Type[Schema] | Mapping[str, Field | Type[Field]] | Callable[[Request], Schema]*, *req: Request | None = None*, ***, *location: str | None = None*, *unknown: str | None = '_default'*, *validate: None | Callable | Iterable[Callable] = None*, *error_status_code: int | None = None*, *error_headers: Mapping[str, str] | None = None*) → Any

Coroutine variant of *webargs.core.Parser.parse*.

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

error_handler(*func: ErrorHandlerT*) → ErrorHandlerT

Decorator that registers a custom error handling function. The function should receive the raised error, request object, marshmallow.Schema instance used to parse the request, error status code, and headers to use for the error response. Overrides the parser's handle_error method.

Example:

```
from webargs import flaskparser

parser = flaskparser.FlaskParser()


class CustomError(Exception):
    pass


@parser.error_handler
def handle_error(error, req, schema, *, error_status_code, error_headers):
    raise CustomError(error.messages)
```

Parameters

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

get_default_arg_name(*location: str*, *schema: Schema | Type[Schema] | Mapping[str, Field | Type[Field]] | Callable[[Request], Schema]*) → str

This method provides the rule by which an argument name is derived for *use_args()* if no explicit arg_name is provided.

By default, the format used is {location}_args. Users may override this method to customize the default argument naming scheme.

schema will be the argument map or schema passed to *use_args()* unless a dict was used, in which case it will be the schema derived from that dict.

get_default_request() → Request | None

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

get_request_from_view_args(*view: Callable*, *args: tuple*, *kwargs: Mapping[str, Any]*) → Request | None

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: ValidationError*, *req: Request*, *schema: Schema*, *, *error_status_code: int | None*, *error_headers: Mapping[str, str] | None*) → NoReturn

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

load_cookies(*req: Request*, *schema: Schema*) → Any

Load the cookies from the request or return `missing` if no value can be found.

load_files(*req: Request*, *schema: Schema*) → Any

Load files from the request or return `missing` if no values can be found.

load_form(*req: Request*, *schema: Schema*) → Any

Load the form data of a request object or return `missing` if no value can be found.

load_headers(*req: Request*, *schema: Schema*) → Any

Load the headers or return `missing` if no value can be found.

load_json(*req: Request*, *schema: Schema*) → Any

Load JSON from a request object or return `missing` if no value can be found.

load_json_or_form(*req: Request*, *schema: Schema*) → Any

Load data from a request, accepting either JSON or form-encoded data.

The data will first be loaded as JSON, and, if that fails, it will be loaded as a form post.

load_querystring(*req: Request*, *schema: Schema*) → Any

Load the query string of a request object or return `missing` if no value can be found.

location_loader(*name: str*) → Callable[[C], C]

Decorator that registers a function for loading a request location. The wrapped function receives a schema and a request.

The schema will usually not be relevant, but it's important in some cases – most notably in order to correctly load multidict values into list fields. Without the schema, there would be no way to know whether to simply `get()` or `getall()` from a multidict for a given value.

Example:

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

@parser.location_loader("name")
def load_data(request, schema):
    return request.data
```

Parameters

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

parse(*argmap: Schema | Type[Schema] | Mapping[str, Field | Type[Field]] | Callable[[Request], Schema]*, *req: Request | None = None*, ***, *location: str | None = None*, *unknown: str | None = '_default'*, *validate: None | Callable | Iterable[Callable] = None*, *error_status_code: int | None = None*, *error_headers: Mapping[str, str] | None = None*) → Any

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.
- **location** (*str*) – Where on the request to load values. Can be any of the values in `__location_map__`. By default, that means one of `('json',`

'query', 'querystring', 'form', 'headers', 'cookies', 'files', 'json_or_form').

- **unknown** (*str*) – A value to pass for unknown when calling the schema's load method. Defaults to marshmallow.EXCLUDE for non-body locations and marshmallow.RAISE for request bodies. Pass None to use the schema's setting instead.
- **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*.
- **error_status_code** (*int*) – Status code passed to error handler functions when a *ValidationError* is raised.
- **error_headers** (*dict*) –

  Headers passed to error handler functions when a
  a *ValidationError* is raised.

  return
  A dictionary of parsed arguments

pre_load(*location_data: Mapping*, ***, *schema: Schema*, *req: Request*, *location: str*) → Mapping

A method of the parser which can transform data after location loading is done. By default it does nothing, but users can subclass parsers and override this method.

use_args(*argmap: Schema | Type[Schema] | Mapping[str, Field | Type[Field]] | Callable[[Request], Schema]*, *req: Request | None = None*, ***, *location: str | None = None*, *unknown: str | None = '_default'*, *as_kwargs: bool = False*, *arg_name: str | None = None*, *validate: None | Callable | Iterable[Callable] = None*, *error_status_code: int | None = None*, *error_headers: Mapping[str, str] | None = None*) → Callable[[...], Callable]

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()}, location="querystring")
def greet(querystring_args):
    return 'Hello ' + querystring_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.
- **location** (*str*) – Where on the request to load values.
- **unknown** (*str*) – A value to pass for unknown when calling the schema's load method.
- **as_kwargs** (*bool*) – Whether to insert arguments as keyword arguments.
- **arg_name** (*str*) – Keyword argument name to use for arguments. Mutually exclusive with as_kwargs.
- **validate** (*callable*) – Validation function that receives the dictionary of parsed arguments. If the function returns False, the parser will raise a *ValidationError*.

- **error_status_code** (*int*) – Status code passed to error handler functions when a *ValidationError* is raised.
- **error_headers** (*dict*) – Headers passed to error handler functions when a a *ValidationError* is raised.

use_kwargs(*argmap: Schema | Type[Schema] | Mapping[str, Field | Type[Field]] | Callable[[Request], Schema]*, *req: Request | None = None*, ***, *location: str | None = None*, *unknown: str | None = '_default'*, *validate: None | Callable | Iterable[Callable] = None*, *error_status_code: int | None = None*, *error_headers: Mapping[str, str] | None = None*) → Callable[[...], Callable]

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()*.

exception webargs.core.**ValidationError**(*message: str | list | dict*, *field_name: str = '_schema'*, *data: Mapping[str, Any] | Iterable[Mapping[str, Any]] | None = None*, *valid_data: list[dict[str, Any]] | dict[str, Any] | None = None*, ***kwargs*)

Raised when validation fails on a field or schema.

Validators and custom fields should raise this exception.

Parameters

- **message** – An error message, list of error messages, or dict of error messages. If a dict, the keys are subitems and the values are error messages.
- **field_name** – Field name to store the error on. If `None`, the error is stored as schema-level error.
- **data** – Raw input data.
- **valid_data** – Valid (de)serialized data.

add_note()

Exception.add_note(note) – add a note to the exception

with_traceback()

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

6.1.2 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(data_key="Content-Type", location="headers"),
}
```

class webargs.fields.**DelimitedList**(*cls_or_instance: Field | type*, *, *delimiter: str | None = None*, ***kwargs*)

A field which is similar to a List, but takes its input as a delimited string (e.g. "foo,bar,baz").

Like List, it can be given a nested field type which it will use to de/serialize each element of the list.

Parameters

- **cls_or_instance** (*Field*) – A field class or instance.
- **delimiter** (*str*) – Delimiter between values.

default_error_messages = {'invalid': 'Not a valid delimited list.'}

Default error messages.

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`.

Note: The schema class here will always be `marshmallow.Schema`, regardless of whether a custom schema class is set on the parser. Pass an explicit schema class if necessary.

6.1.3 webargs.multidictproxy

class webargs.multidictproxy.**MultiDictProxy**(*multidict*, *schema: ~marshmallow.schema.Schema*, *known_multi_fields: ~typing.Tuple[~typing.Type, ...] = (<class 'marshmallow.fields.List'>, <class 'marshmallow.fields.Tuple'>)*)

A proxy object which wraps multidict types along with a matching schema Whenever a value is looked up, it is checked against the schema to see if there is a matching field where `is_multiple` is True. If there is, then the data should be loaded as a list or tuple.

In all other cases, __getitem__ proxies directly to the input multidict.

6.1.4 webargs.asyncparser

Asynchronous request parser.

class webargs.asyncparser.**AsyncParser**(*location: str | None = None*, ***, *unknown: str | None = '_default'*, *error_handler: Callable[[...], NoReturn] | None = None*, *schema_class: type[Schema] | None = None*)

Asynchronous variant of *webargs.core.Parser*.

The `parse` method is redefined to be `async`.

DEFAULT_LOCATION: str = 'json'

Default location to check for data

DEFAULT_SCHEMA_CLASS

alias of Schema

DEFAULT_VALIDATION_MESSAGE: str = 'Invalid value.'

Default error message for validation errors

DEFAULT_VALIDATION_STATUS: int = 422

Default status code to return for validation errors

KNOWN_MULTI_FIELDS: list[type] = [<class 'marshmallow.fields.List'>, <class 'marshmallow.fields.Tuple'>]

field types which should always be treated as if they set `is_multiple=True`

async async_parse(*argmap: Schema | Type[Schema] | Mapping[str, Field | Type[Field]] | Callable[[Request], Schema]*, *req: Request | None = None*, ***, *location: str | None = None*, *unknown: str | None = '_default'*, *validate: None | Callable | Iterable[Callable] = None*, *error_status_code: int | None = None*, *error_headers: Mapping[str, str] | None = None*) → Any

Coroutine variant of *webargs.core.Parser.parse*.

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

error_handler(*func: ErrorHandlerT*) → ErrorHandlerT

Decorator that registers a custom error handling function. The function should receive the raised error, request object, marshmallow.Schema instance used to parse the request, error status code, and headers to use for the error response. Overrides the parser's `handle_error` method.

Example:

```
from webargs import flaskparser

parser = flaskparser.FlaskParser()


class CustomError(Exception):
    pass


@parser.error_handler
def handle_error(error, req, schema, *, error_status_code, error_headers):
    raise CustomError(error.messages)
```

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

get_default_arg_name(*location: str*, *schema: Schema | Type[Schema] | Mapping[str, Field | Type[Field]] | Callable[[Request], Schema]*) → str

This method provides the rule by which an argument name is derived for *use_args()* if no explicit `arg_name` is provided.

By default, the format used is `{location}_args`. Users may override this method to customize the default argument naming scheme.

`schema` will be the argument map or schema passed to *use_args()* unless a dict was used, in which case it will be the schema derived from that dict.

get_default_request() → Request | None

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

get_request_from_view_args(*view: Callable*, *args: tuple*, *kwargs: Mapping[str, Any]*) → Request | None

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: ValidationError*, *req: Request*, *schema: Schema*, *, *error_status_code: int | None*, *error_headers: Mapping[str, str] | None*) → NoReturn

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

load_cookies(*req: Request*, *schema: Schema*) → Any

Load the cookies from the request or return `missing` if no value can be found.

load_files(*req: Request*, *schema: Schema*) → Any

Load files from the request or return `missing` if no values can be found.

load_form(*req: Request*, *schema: Schema*) → Any

Load the form data of a request object or return `missing` if no value can be found.

load_headers(*req: Request*, *schema: Schema*) → Any

Load the headers or return `missing` if no value can be found.

load_json(*req: Request*, *schema: Schema*) → Any

Load JSON from a request object or return `missing` if no value can be found.

load_json_or_form(*req: Request*, *schema: Schema*) → Any

Load data from a request, accepting either JSON or form-encoded data.

The data will first be loaded as JSON, and, if that fails, it will be loaded as a form post.

load_querystring(*req: Request*, *schema: Schema*) → Any

Load the query string of a request object or return `missing` if no value can be found.

location_loader(*name: str*) → Callable[[C], C]

Decorator that registers a function for loading a request location. The wrapped function receives a schema and a request.

The schema will usually not be relevant, but it's important in some cases – most notably in order to correctly load multidict values into list fields. Without the schema, there would be no way to know whether to simply `get()` or `getall()` from a multidict for a given value.

Example:

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

@parser.location_loader("name")
def load_data(request, schema):
    return request.data
```

Parameters

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

async parse(*argmap: Schema | Type[Schema] | Mapping[str, Field | Type[Field]] | Callable[[Request], Schema], req: Request | None = None, *, location: str | None = None, unknown: str | None = '_default', validate: None | Callable | Iterable[Callable] = None, error_status_code: int | None = None, error_headers: Mapping[str, str] | None = None*) → Any

Coroutine variant of *webargs.core.Parser*.

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

pre_load(*location_data: Mapping, *, schema: Schema, req: Request, location: str*) → Mapping

A method of the parser which can transform data after location loading is done. By default it does nothing, but users can subclass parsers and override this method.

use_args(*argmap: Schema | Type[Schema] | Mapping[str, Field | Type[Field]] | Callable[[Request], Schema], req: Request | None = None, *, location: str | None = None, unknown: str | None = '_default', as_kwargs: bool = False, arg_name: str | None = None, validate: None | Callable | Iterable[Callable] = None, error_status_code: int | None = None, error_headers: Mapping[str, str] | None = None*) → Callable[[...], Callable]

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()}, location="querystring")
def greet(querystring_args):
    return 'Hello ' + querystring_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`.
- **location** (*str*) – Where on the request to load values.
- **unknown** (*str*) – A value to pass for `unknown` when calling the schema's `load` method.

- **as_kwargs** (*bool*) – Whether to insert arguments as keyword arguments.
- **arg_name** (*str*) – Keyword argument name to use for arguments. Mutually exclusive with as_kwargs.
- **validate** (*callable*) – Validation function that receives the dictionary of parsed arguments. If the function returns `False`, the parser will raise a `ValidationError`.
- **error_status_code** (*int*) – Status code passed to error handler functions when a `ValidationError` is raised.
- **error_headers** (*dict*) – Headers passed to error handler functions when a a `ValidationError` is raised.

use_kwargs(*argmap: Schema | Type[Schema] | Mapping[str, Field | Type[Field]] | Callable[[Request], Schema]*, *req: Request | None = None*, ***, *location: str | None = None*, *unknown: str | None = '_default'*, *validate: None | Callable | Iterable[Callable] = None*, *error_status_code: int | None = None*, *error_headers: Mapping[str, str] | None = None*) → Callable[[...], Callable]

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()*.

6.1.5 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__)

user_detail_args = {
    'per_page': fields.Int()
}

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

class webargs.flaskparser.**FlaskParser**(*location: str | None = None*, ***, *unknown: str | None = '_default'*, *error_handler: Callable[[...], NoReturn] | None = None*, *schema_class: type[Schema] | None = None*)

Flask request argument parser.

get_default_request() → Request

Override to use Flask's thread-local request object by default

handle_error(*error: ValidationError*, *req: Request*, *schema: Schema*, ***, *error_status_code: int | None*, *error_headers: Mapping[str, str] | None*) → NoReturn

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

load_cookies(*req: Request*, *schema: Schema*) → Any

Return cookies from the request.

load_files(*req: Request*, *schema: Schema*) → Any

Return files from the request as a MultiDictProxy.

load_form(*req: Request*, *schema: Schema*) → Any

Return form values from the request as a MultiDictProxy.

load_headers(*req: Request*, *schema: Schema*) → Any

Return headers from the request as a MultiDictProxy.

load_querystring(*req: Request*, *schema: Schema*) → Any

Return query params from the request as a MultiDictProxy.

load_view_args(*req: Request*, *schema: Schema*) → Any

Return the request's `view_args` or `missing` if there are none.

webargs.flaskparser.**abort**(*http_status_code: int*, *exc: Exception | None = None*, ***kwargs: Any*) → NoReturn

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.

6.1.6 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(load_default='World')
}

class MyView(View):

    @use_args(hello_args)
```

(continues on next page)

(continued from previous page)

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

class webargs.djangoparser.**DjangoParser**(*location: str | None = None*, ***, *unknown: str | None = '_default'*, *error_handler: Callable[[...], NoReturn] | None = None*, *schema_class: type[Schema] | None = 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`.

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`.

load_cookies(*req: HttpRequest*, *schema*)

Return cookies from the request.

load_files(*req: HttpRequest*, *schema*)

Return files from the request as a MultiDictProxy.

load_form(*req: HttpRequest*, *schema*)

Return form values from the request as a MultiDictProxy.

load_headers(*req: HttpRequest*, *schema*)

Return headers from the request.

load_querystring(*req: HttpRequest*, *schema*)

Return query params from the request as a MultiDictProxy.

6.1.7 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(load_default='World')
}
@route('/', method='GET', apply=use_args(hello_args))
```

(continues on next page)

(continued from previous page)

```
def index(args):
    return 'Hello ' + args['name']

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

class webargs.bottleparser.**BottleParser**(*location: str | None = None*, ***, *unknown: str | None = '_default'*, *error_handler: Callable[[...], NoReturn] | None = None*, *schema_class: type[Schema] | None = None*)

Bottle.py request argument parser.

get_default_request()

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

handle_error(*error*, *req*, *schema*, ***, *error_status_code*, *error_headers*)

Handles errors during parsing. Aborts the current request with a 400 error.

load_cookies(*req*, *schema*)

Return cookies from the request.

load_files(*req*, *schema*)

Return files from the request as a MultiDictProxy.

load_form(*req*, *schema*)

Return form values from the request as a MultiDictProxy.

load_headers(*req*, *schema*)

Return headers from the request as a MultiDictProxy.

load_querystring(*req*, *schema*)

Return query params from the request as a MultiDictProxy.

6.1.8 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(load_default='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**(*location: str | None = None*, ***, *unknown: str | None = '_default'*, *error_handler: Callable[[...], NoReturn] | None = None*, *schema_class: type[Schema] | None = None*)

Tornado request argument parser.

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*, *req: HTTPServerRequest*, *schema*, ***, *error_status_code*, *error_headers*)

Handles errors during parsing. Raises a `tornado.web.HTTPError` with a 400 error.

load_cookies(*req: HTTPServerRequest*, *schema*)

Return cookies from the request as a MultiDictProxy.

load_files(*req: HTTPServerRequest*, *schema*)

Return files from the request as a MultiDictProxy.

load_form(*req: HTTPServerRequest*, *schema*)

Return form values from the request as a MultiDictProxy.

load_headers(*req: HTTPServerRequest*, *schema*)

Return headers from the request as a MultiDictProxy.

load_querystring(*req: HTTPServerRequest*, *schema*)

Return query params from the request as a MultiDictProxy.

class webargs.tornadoparser.**WebArgsTornadoCookiesMultiDictProxy**(*multidict*, *schema: ~marshmallow.schema.Schema*, *known_multi_fields: ~typing.Tuple[~typing.Type, ...] = (<class 'marshmallow.fields.List'>, <class 'marshmallow.fields.Tuple'>)*)

And a special override for cookies because they come back as objects with a `value` attribute we need to extract. Also, does not use the `_unicode` decoding step

class webargs.tornadoparser.**WebArgsTornadoMultiDictProxy**(*multidict*, *schema: ~marshmallow.schema.Schema*, *known_multi_fields: ~typing.Tuple[~typing.Type, ...] = (<class 'marshmallow.fields.List'>, <class 'marshmallow.fields.Tuple'>)*)

Override class for Tornado multidicts, handles argument decoding requirements.

6.1.9 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(load_default='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**(*location: str | None = None*, ***, *unknown: str | None = '_default'*, *error_handler: Callable[[...], NoReturn] | None = None*, *schema_class: type[Schema] | None = None*)

Pyramid request argument parser.

handle_error(*error*, *req: Request*, *schema*, ***, *error_status_code*, *error_headers*)

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

load_cookies(*req: Request*, *schema*)

Return cookies from the request as a MultiDictProxy.

load_files(*req: Request*, *schema*)

Return files from the request as a MultiDictProxy.

load_form(*req: Request*, *schema*)

Return form values from the request as a MultiDictProxy.

load_headers(*req: Request*, *schema*)

Return headers from the request as a MultiDictProxy.

load_matchdict(*req: Request*, *schema*)

Return the request's `matchdict` as a MultiDictProxy.

load_querystring(*req: Request*, *schema*)

Return query params from the request as a MultiDictProxy.

use_args(*argmap*, *req: Request | None = None*, ***, *location='json'*, *unknown=None*, *as_kwargs=False*, *arg_name=None*, *validate=None*, *error_status_code=None*, *error_headers=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.
- **location** (*str*) – Where on the request to load values.
- **unknown** (*str*) – A value to pass for `unknown` when calling the schema's `load` method.
- **as_kwargs** (*bool*) – Whether to insert arguments as keyword arguments.
- **arg_name** (*str*) – Keyword argument name to use for arguments. Mutually exclusive with as_kwargs.
- **validate** (*callable*) – Validation function that receives the dictionary of parsed arguments. If the function returns `False`, the parser will raise a `ValidationError`.
- **error_status_code** (*int*) – Status code passed to error handler functions when a `ValidationError` is raised.
- **error_headers** (*dict*) – Headers passed to error handler functions when a a `ValidationError` is raised.

webargs.pyramidparser.**use_args**(*argmap*, *req: Request | None = None*, ***, *location='json'*, *unknown=None*, *as_kwargs=False*, *arg_name=None*, *validate=None*, *error_status_code=None*, *error_headers=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.
- **location** (*str*) – Where on the request to load values.
- **unknown** (*str*) – A value to pass for `unknown` when calling the schema's `load` method.
- **as_kwargs** (*bool*) – Whether to insert arguments as keyword arguments.
- **arg_name** (*str*) – Keyword argument name to use for arguments. Mutually exclusive with as_kwargs.
- **validate** (*callable*) – Validation function that receives the dictionary of parsed arguments. If the function returns `False`, the parser will raise a `ValidationError`.
- **error_status_code** (*int*) – Status code passed to error handler functions when a `ValidationError` is raised.
- **error_headers** (*dict*) – Headers passed to error handler functions when a a `ValidationError` is raised.

6.1.10 webargs.falconparser

Falcon request argument parsing module.

class webargs.falconparser.**FalconParser**(*location: str | None = None*, ***, *unknown: str | None = '_default'*, *error_handler: Callable[[...], NoReturn] | None = None*, *schema_class: type[Schema] | None = None*)

Falcon request argument parser.

Defaults to using the `media` location. See *load_media()* for details on the media location.

DEFAULT_LOCATION: str = 'media'

Default location to check for data

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*, *req: Request*, *schema*, ***, *error_status_code*, *error_headers*)

Handles errors during parsing.

load_cookies(*req: Request*, *schema*)

Return cookies from the request.

load_files(*req: Request*, *schema*)

Load files from the request or return `missing` if no values can be found.

load_form(*req: Request*, *schema*)

Return form values from the request as a MultiDictProxy

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

load_headers(*req: Request*, *schema*)

Return headers from the request.

load_media(*req: Request*, *schema*)

Return data unpacked and parsed by one of Falcon's media handlers. By default, Falcon only handles JSON payloads.

To configure additional media handlers, see the Falcon documentation on media types.

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

load_querystring(*req: Request*, *schema*)

Return query params from the request as a MultiDictProxy.

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.

6.1.11 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**(*location: str | None = None*, ***, *unknown: str | None = '_default'*, *error_handler: Callable[[...], NoReturn] | None = None*, *schema_class: type[Schema] | None = None*)

aiohttp request argument parser.

get_request_from_view_args(*view: Callable*, *args: Iterable*, *kwargs: Mapping*)

Get request object from a handler function or method. Used internally by `use_args` and `use_kwargs`.

handle_error(*error:* ValidationError, *req*, *schema: Schema*, ***, *error_status_code: int | None*, *error_headers: Mapping[str, str] | None*) → NoReturn

Handle ValidationErrors and return a JSON response of error messages to the client.

load_cookies(*req*, *schema: Schema*) → *MultiDictProxy*

Return cookies from the request as a MultiDictProxy.

load_files(*req*, *schema: Schema*) → NoReturn

Load files from the request or return `missing` if no values can be found.

async load_form(*req*, *schema: Schema*) → *MultiDictProxy*

Return form values from the request as a MultiDictProxy.

load_headers(*req*, *schema: Schema*) → *MultiDictProxy*

Return headers from the request as a MultiDictProxy.

async load_json(*req*, *schema: Schema*)

Return a parsed json payload from the request.

async load_json_or_form(*req*, *schema: Schema*) → dict | *MultiDictProxy*

Load data from a request, accepting either JSON or form-encoded data.

The data will first be loaded as JSON, and, if that fails, it will be loaded as a form post.

load_match_info(*req*, *schema: Schema*) → Mapping

Load the request's `match_info`.

load_querystring(*req*, *schema: Schema*) → *MultiDictProxy*

Return query params from the request as a MultiDictProxy.

exception webargs.aiohttpparser.**HTTPUnprocessableEntity**(*, *headers: Mapping[str | istr, str] | CIMultiDict | CIMultiDictProxy | None = None*, *reason: str | None = None*, *body: Any = None*, *text: str | None = None*, *content_type: str | None = None*)

CHAPTER

SEVEN

PROJECT INFO

7.1 License

```
Copyright 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.
```

7.2 Changelog

7.2.1 8.5.0 (2024-04-25)

Other changes:

- Test against Python 3.12.
- Async location loading now supports loader functions which are not themselves async, but which return an awaitable result. This means that users who are already handling awaitable objects can return them from non-async loaders and expect `webargs` to `await` them. This allows some async interactions with supported frameworks, where the `webargs`-provided parser returns a framework object and the framework can be set to return awaitables, e.g., the Falcon parser. Thanks @j0k2r for the PR! (#935)

7.2.2 8.4.0 (2024-01-07)

Features:

- Add a new class attribute, `empty_value` to `DelimitedList` and `DelimitedTuple`, with a default of `""`. This controls the value deserialized when an empty string is seen.

`empty_value` can be used to handle types other than strings more gracefully, e.g.

```
from webargs import fields


class IntList(fields.DelimitedList):
    empty_value = 0


myfield = IntList(fields.Int())
```

Note: `empty_value` will be changing in webargs v9.0 to be `missing` by default. This will allow use of fields with `load_default` to specify handling of the empty value.

- The rule for default argument names has been made configurable by overriding the `get_default_arg_name` method. This is described in the argument passing documentation.

Other changes:

- Drop support for Python 3.7, which is EOL.
- Type annotations for `FlaskParser` have been improved.

7.2.3 8.3.0 (2023-07-10)

Features:

- `webargs.Parser` now inherits from `typing.Generic` and is parametrizable over the type of the request object. Various framework-specific parsers are parametrized over their relevant request object classes.
- `webargs.Parser` and its subclasses now support passing arguments as a single keyword argument without expanding the parsed data into its components. For more details, see advanced docs on `Argument Passing and arg_name`.

Other changes:

- Type annotations have been improved to allow `Mapping` for dict-like schemas where previously `dict` was used. This makes the type covariant rather than invariant (#836).
- Test against Python 3.11 (#787).

7.2.4 8.2.0 (2022-07-11)

Features:

- A new method, `webargs.Parser.async_parse`, can be used for async-aware parsing from the base parser class. This can handle async location loader functions and async error handlers.
- `webargs.Parser.use_args` and `use_kwargs` can now be used to decorate async functions, and will use `async_parse` if the decorated function is also async. They will call the non-async `parse` method when used to decorate non-async functions.
- As a result of the changes to `webargs.Parser`, `FlaskParser`, `DjangoParser`, and `FalconParser` now all support async views. Thanks @Isira-Seneviratne for the initial PR.

Changes:

- The implementation of `AsyncParser` has changed. Now that `webargs.Parser` has built-in support for async usage, the primary purpose of `AsyncParser` is to redefine `parse` as an alias for `async_parse`
- Set `python_requires>=3.7.2` in package metadata (#692). Thanks @kasium for the PR.

7.2.5 8.1.0 (2022-01-12)

Bug fixes:

- Fix publishing type hints per PEP-561. (#650).
- Add DelimitedTuple to fields.__all__ (#678).
- Narrow type of `argmap` from `Mapping` to `Dict` (#682).

Other changes:

- Test against Python 3.10 (#647).
- Drop support for Python 3.6 (#673).
- Address distutils deprecation warning in Python 3.10 (#652). Thanks @kkirsche for the PR.
- Use postponed evaluation of annotations (#663). Thanks @Isira-Seneviratne for the PR.
- Pin mypy version in tox (#674).
- Improve type annotations for `__version_info__` (#680).

7.2.6 8.0.1 (2021-08-12)

Bug fixes:

- Fix "`DelimitedList` deserializes empty string as `['']`" (#623). Thanks @TTWSchell for reporting and for the PR.

Other changes:

- New documentation theme with `furo`. Thanks to @pradyunsg for writing furo!
- Webargs has a new logo. Thanks to @michaelizergit! (#312)
- Don't build universal wheels. We don't support Python 2 anymore. (#632)
- Make the build reproducible (#631).

7.2.7 8.0.0 (2021-04-08)

Features:

- Add `Parser.pre_load` as a method for allowing users to modify data before schema loading, but without redefining location loaders. See advanced docs on `Parser pre_load` for usage information. (#583)
- *Backwards-incompatible*: `unknown` defaults to `None` for body locations (`json`, `form` and `json_or_form`) (#580).
- Detection of fields as "multi-value" for unpacking lists from multi-dict types is now extensible with the `is_multiple` attribute. If a field sets `is_multiple = True` it will be detected as a multi-value field. If `is_multiple` is not set or is set to `None`, webargs will check if the field is an instance of `List` or `Tuple`. (#563)
- A new attribute on `Parser` objects, `Parser.KNOWN_MULTI_FIELDS` can be used to set fields which should be detected as `is_multiple=True` even when the attribute is not set (#592).

See docs on "Multi-Field Detection" for more details.

Bug fixes:

- `Tuple` field now behaves as a "multiple" field (#585).

7.2.8 7.0.1 (2020-12-14)

Bug fixes:

- Fix `DelimitedList` and `DelimitedTuple` to pass additional keyword arguments through their `_serialize` methods to the child fields and fix type checking on these classes. (#569) Thanks to @decaz for reporting.

7.2.9 7.0.0 (2020-12-10)

Changes:

- *Backwards-incompatible*: Drop support for webapp2 (#565).
- Add type annotations to `Parser` class, `DelimitedList`, and `DelimitedTuple`. (#566)

7.2.10 7.0.0b2 (2020-12-01)

Features:

- `DjangoParser` now supports the `headers` location. (#540)
- `FalconParser` now supports a new `media` location, which uses Falcon's `media` decoding. (#253)

`media` behaves very similarly to the `json` location but also supports any registered media handler. See the Falcon documentation on media types for more details.

Changes:

- `FalconParser` defaults to the `media` location instead of `json`. (#253)
- Test against Python 3.9 (#552).
- *Backwards-incompatible*: Drop support for Python 3.5 (#553).

7.2.11 7.0.0b1 (2020-09-11)

Refactoring:

- *Backwards-incompatible*: Remove support for marshmallow2 (#539)
- *Backwards-incompatible*: Remove `dict2schema`

  Users desiring the `dict2schema` functionality may now rely upon `marshmallow.Schema.from_dict`. Rewrite any code using `dict2schema` like so:

```
import marshmallow as ma

# webargs 6.x and older
from webargs import dict2schema

myschema = dict2schema({"q1", ma.fields.Int()})

# webargs 7.x
myschema = ma.Schema.from_dict({"q1", ma.fields.Int()})
```

Features:

- Add `unknown` as a parameter to `Parser.parse`, `Parser.use_args`, `Parser.use_kwargs`, and parser instantiation. When set, it will be passed to `Schema.load`. When not set, the value passed will depend on the parser's settings. If set to `None`, the schema's default behavior will be used (i.e. no value is passed to `Schema.load`) and parser settings will be ignored.

This allows usages like

```
import marshmallow as ma


@parser.use_kwargs(
    {"q1": ma.fields.Int(), "q2": ma.fields.Int()}, location="query", unknown=ma.EXCLUDE
)
def foo(q1, q2):
    ...
```

- Defaults for `unknown` may be customized on parser classes via `Parser.DEFAULT_UNKNOWN_BY_LOCATION`, which maps location names to values to use.

Usages are varied, but include

```
import marshmallow as ma
from webargs.flaskparser import FlaskParser


# as well as...
class MyParser(FlaskParser):
    DEFAULT_UNKNOWN_BY_LOCATION = {"query": ma.INCLUDE}


parser = MyParser()
```

Setting the `unknown` value for a Parser instance has higher precedence. So

```
parser = MyParser(unknown=ma.RAISE)
```

will always pass `RAISE`, even when the location is `query`.

- By default, webargs will pass `unknown=EXCLUDE` for all locations except for request bodies (`json`, `form`, and `json_or_form`) and path parameters. Request bodies and path parameters will pass `unknown=RAISE`. This behavior is defined by the default value for `DEFAULT_UNKNOWN_BY_LOCATION`.

Changes:

- Registered `error_handler` callbacks are required to raise an exception. If a handler is invoked and no exception is raised, *webargs* will raise a `ValueError` (#527)

7.2.12 6.1.1 (2020-09-08)

Bug fixes:

- Failure to validate flask headers would produce error data which contained tuples as keys, and was therefore not JSON-serializable. (#500) These errors will now extract the headername as the key correctly. Thanks to @shughes-uk for reporting.

7.2.13 6.1.0 (2020-04-05)

Features:

- Add `fields.DelimitedTuple` when using marshmallow 3. This behaves as a combination of `fields.DelimitedList` and `marshmallow.fields.Tuple`. It takes an iterable of fields, plus a delimiter (defaults to `,`), and parses delimiter-separated strings into tuples. (#509)
- Add `__str__` and `__repr__` to MultiDictProxy to make it easier to work with (#488)

Support:

- Various docs updates (#482, #486, #489, #498, #508). Thanks @lefterisjp, @timgates42, and @ugultopu for the PRs.

7.2.14 6.0.0 (2020-02-27)

Features:

- `FalconParser`: Pass request content length to `req.stream.read` to provide compatibility with `falcon.testing` (#477). Thanks @suola for the PR.
- *Backwards-incompatible*: Factorize the `use_args` / `use_kwargs` branch in all parsers. When `as_kwargs` is `False`, arguments are now consistently appended to the arguments list by the `use_args` decorator. Before this change, the `PyramidParser` would prepend the argument list on each call to `use_args`. Pyramid view functions must reverse the order of their arguments. (#478)

7.2.15 6.0.0b8 (2020-02-16)

Refactoring:

- *Backwards-incompatible*: Use keyword-only arguments (#472).

7.2.16 6.0.0b7 (2020-02-14)

Features:

- *Backwards-incompatible*: webargs will rewrite the error messages in ValidationErrors to be namespaced under the location which raised the error. The `messages` field on errors will therefore be one layer deeper with a single top-level key.

7.2.17 6.0.0b6 (2020-01-31)

Refactoring:

- Remove the cache attached to webargs parsers. Due to changes between webargs v5 and v6, the cache is no longer considered useful.

Other changes:

- Import `Mapping` from `collections.abc` in pyramidparser.py (#471). Thanks @tirkarthi for the PR.

7.2.18 6.0.0b5 (2020-01-30)

Refactoring:

- *Backwards-incompatible*: `DelimitedList` now requires that its input be a string and always serializes as a string. It can still serialize and deserialize using another field, e.g. `DelimitedList(Int())` is still valid and requires that the values in the list parse as ints.

7.2.19 6.0.0b4 (2020-01-28)

Bug fixes:

- CVE-2020-7965: Don't attempt to parse JSON if request's content type is mismatched (bugfix from 5.5.3).

7.2.20 6.0.0b3 (2020-01-21)

Features:

- *Backwards-incompatible*: Support Falcon 2.0. Drop support for Falcon 1.x (#459). Thanks @dodumosu and @Nateyo for the PR.

7.2.21 6.0.0b2 (2020-01-07)

Other changes:

- *Backwards-incompatible*: Drop support for Python 2 (#440). Thanks @hugovk for the PR.

7.2.22 6.0.0b1 (2020-01-06)

Features:

- *Backwards-incompatible*: Schemas will now load all data from a location, not only data specified by fields. As a result, schemas with validators which examine the full input data may change in behavior. The `unknown` parameter on schemas may be used to alter this. For example, `unknown=marshmallow.EXCLUDE` will produce a behavior similar to webargs v5.

Bug fixes:

- *Backwards-incompatible*: All parsers now require the Content-Type to be set correctly when processing JSON request bodies. This impacts `DjangoParser`, `FalconParser`, `FlaskParser`, and `PyramidParser`

Refactoring:

- *Backwards-incompatible*: Schema fields may not specify a location any longer, and `Parser.use_args` and `Parser.use_kwargs` now accept `location` (singular) instead of `locations` (plural). Instead of using a single field or schema with multiple `locations`, users are recommended to make multiple calls to `use_args` or `use_kwargs` with a distinct schema per location. For example, code should be rewritten like this:

```
# webargs 5.x and older
@parser.use_args(
    {
        "q1": ma.fields.Int(location="query"),
        "q2": ma.fields.Int(location="query"),
        "h1": ma.fields.Int(location="headers"),
    },
    locations=("query", "headers"),
)
def foo(q1, q2, h1):
    ...


# webargs 6.x
@parser.use_args({"q1": ma.fields.Int(), "q2": ma.fields.Int()}, location="query")
@parser.use_args({"h1": ma.fields.Int()}, location="headers")
def foo(q1, q2, h1):
    ...
```

- The `location_handler` decorator has been removed and replaced with `location_loader`. `location_loader` serves the same purpose (letting you write custom hooks for loading data) but its expected method signature is different. See the docs on `location_loader` for proper usage.

Thanks @sirosen for the PR!

7.2.23 5.5.3 (2020-01-28)

Bug fixes:

- CVE-2020-7965: Don't attempt to parse JSON if request's content type is mismatched.

7.2.24 5.5.2 (2019-10-06)

Bug fixes:

- Handle `UnicodeDecodeError` when parsing JSON payloads (#427). Thanks @lindycoder for the catch and patch.

7.2.25 5.5.1 (2019-09-15)

Bug fixes:

- Remove usage of deprecated `Field.fail` when using marshmallow 3.

7.2.26 5.5.0 (2019-09-07)

Support:

- Various docs updates (#414, #421).

Refactoring:

- Don't mutate `globals()` in `webargs.fields` (#411).
- Use marshmallow 3's `Schema.from_dict` if available (#415).

7.2.27 5.4.0 (2019-07-23)

Changes:

- Use explicit type check for `fields.DelimitedList` when deciding to parse value with `getlist()` (#406 (comment)).

Support:

- Add "Parsing Lists in Query Strings" section to docs (#406).

7.2.28 5.3.2 (2019-06-19)

Bug fixes:

- marshmallow 3.0.0rc7 compatibility (#395).

7.2.29 5.3.1 (2019-05-05)

Bug fixes:

- marshmallow 3.0.0rc6 compatibility (#384).

7.2.30 5.3.0 (2019-04-08)

Features:

- Add `"path"` location to `AIOHTTPParser`, `FlaskParser`, and `PyramidParser` (#379). Thanks @zhenhua32 for the PR.
- Add `webargs.__version_info__`.

7.2.31 5.2.0 (2019-03-16)

Features:

- Make the schema class used when generating a schema from a dict overridable (#375). Thanks @ThiefMaster.

7.2.32 5.1.3 (2019-03-11)

Bug fixes:

- CVE-2019-9710: Fix race condition between parallel requests when the cache is used (#371). Thanks @ThiefMaster for reporting and fixing.

7.2.33 5.1.2 (2019-02-03)

Bug fixes:

- Remove lingering usages of `ValidationError.status_code` (#365). Thanks @decaz for reporting.
- Avoid `AttributeError` on Python<3.5.4 (#366).
- Fix incorrect type annotations for `error_headers`.
- Fix outdated docs (#367). Thanks @alexandersoto for reporting.

7.2.34 5.1.1.post0 (2019-01-30)

- Include LICENSE in sdist (#364).

7.2.35 5.1.1 (2019-01-28)

Bug fixes:

- Fix installing `simplejson` on Python 2 by distributing a Python 2-only wheel (#363).

7.2.36 5.1.0 (2019-01-11)

Features:

- Error handlers for `AsyncParser` classes may be coroutine functions.
- Add type annotations to `AsyncParser` and `AIOHTTPParser`.

Bug fixes:

- Fix compatibility with Flask<1.0 (#355). Thanks @hoatle for reporting.
- Address warning on Python 3.7 about importing from `collections.abc`.

7.2.37 5.0.0 (2019-01-03)

Features:

- *Backwards-incompatible*: A 400 HTTPError is raised when an invalid JSON payload is passed. (#329). Thanks @zedrdave for reporting.

Other changes:

- *Backwards-incompatible*: `webargs.argmap2schema` is removed. Use `webargs.dict2schema` instead.
- *Backwards-incompatible*: `webargs.ValidationError` is removed. Use `marshmallow.ValidationError` instead.

```
# <5.0.0
from webargs import ValidationError


def auth_validator(value):
    # ...
    raise ValidationError("Authentication failed", status_code=401)


@use_args({"auth": fields.Field(validate=auth_validator)})
def auth_view(args):
    return jsonify(args)


# >=5.0.0
from marshmallow import ValidationError


def auth_validator(value):
    # ...
    raise ValidationError("Authentication failed")
```

(continues on next page)

(continued from previous page)

```
@use_args({"auth": fields.Field(validate=auth_validator)}, error_status_code=401)
def auth_view(args):
    return jsonify(args)
```

- *Backwards-incompatible*: Missing arguments will no longer be filled in when using `@use_kwargs` (#342#307#252). Use `**kwargs` to account for non-required fields.

```
# <5.0.0
@use_kwargs(
    {"first_name": fields.Str(required=True), "last_name": fields.Str(required=False)}
)
def myview(first_name, last_name):
    # last_name is webargs.missing if it's missing from the request
    return {"first_name": first_name}


# >=5.0.0
@use_kwargs(
    {"first_name": fields.Str(required=True), "last_name": fields.Str(required=False)}
)
def myview(first_name, **kwargs):
    # last_name will not be in kwargs if it's missing from the request
    return {"first_name": first_name}
```

- simplejson is now a required dependency on Python 2 (#334). This ensures consistency of behavior across Python 2 and 3.

7.2.38 4.4.1 (2018-01-03)

Bug fixes:

- Remove usages of `argmap2schema` from `fields.Nested`, `AsyncParser`, and `PyramidParser`.

7.2.39 4.4.0 (2019-01-03)

- *Deprecation*: `argmap2schema` is deprecated in favor of `dict2schema` (#352).

7.2.40 4.3.1 (2018-12-31)

- Add `force_all` param to `PyramidParser.use_args`.
- Add warning about missing arguments to `AsyncParser`.

7.2.41 4.3.0 (2018-12-30)

- *Deprecation*: Add warning about missing arguments getting added to parsed arguments dictionary (#342). This behavior will be removed in version 5.0.0.

7.2.42 4.2.0 (2018-12-27)

Features:

- Add `force_all` argument to `use_args` and `use_kwargs` (#252, #307). Thanks @piroux for reporting.
- *Deprecation*: The `status_code` and `headers` arguments to `ValidationError` are deprecated. Pass `error_status_code` and `error_headers` to `Parser.parse`, `Parser.use_args`, and `Parser.use_kwargs` instead. (#327, #336).
- Custom error handlers receive `error_status_code` and `error_headers` arguments. (#327).

```
# <4.2.0
@parser.error_handler
def handle_error(error, req, schema):
    raise CustomError(error.messages)


class MyParser(FlaskParser):
    def handle_error(self, error, req, schema):
        # ...
        raise CustomError(error.messages)


# >=4.2.0
@parser.error_handler
def handle_error(error, req, schema, status_code, headers):
    raise CustomError(error.messages)


# OR


@parser.error_handler
def handle_error(error, **kwargs):
    raise CustomError(error.messages)


class MyParser(FlaskParser):
    def handle_error(self, error, req, schema, status_code, headers):
        # ...
        raise CustomError(error.messages)

    # OR

    def handle_error(self, error, req, **kwargs):
        # ...
        raise CustomError(error.messages)
```

Legacy error handlers will be supported until version 5.0.0.

7.2.43 4.1.3 (2018-12-02)

Bug fixes:

- Fix bug in `AIOHTTParser` that prevented calling `use_args` on the same view function multiple times (#273). Thanks to @dnp1 for reporting and @jangelo for the fix.
- Fix compatibility with marshmallow 3.0.0rc1 (#330).

7.2.44 4.1.2 (2018-11-03)

Bug fixes:

- Fix serialization behavior of `DelimitedList` (#319). Thanks @lee3164 for the PR.

Other changes:

- Test against Python 3.7.

7.2.45 4.1.1 (2018-10-25)

Bug fixes:

- Fix bug in `AIOHTTPParser` that caused a `JSONDecode` error when parsing empty payloads (#229). Thanks @explosic4 for reporting and thanks user @kochab for the PR.

7.2.46 4.1.0 (2018-09-17)

Features:

- Add `webargs.testing` module, which exposes `CommonTestCase` to third-party parser libraries (see comments in #287).

7.2.47 4.0.0 (2018-07-15)

Features:

- *Backwards-incompatible*: Custom error handlers receive the `marshmallow.Schema` instance as the third argument. Update any functions decorated with `Parser.error_handler` to take a `schema` argument, like so:

```
# 3.x
@parser.error_handler
def handle_error(error, req):
    raise CustomError(error.messages)


# 4.x
@parser.error_handler
def handle_error(error, req, schema):
    raise CustomError(error.messages)
```

See marshmallow-code/marshmallow#840 (comment) for more information about this change.

Bug fixes:

- *Backwards-incompatible*: Rename `webargs.async` to `webargs.asyncparser` to fix compatibility with Python 3.7 (#240). Thanks @Reskov for the catch and patch.

Other changes:

- *Backwards-incompatible*: Drop support for Python 3.4 (#243). Python 2.7 and >=3.5 are supported.
- *Backwards-incompatible*: Drop support for marshmallow<2.15.0. marshmallow>=2.15.0 and >=3.0.0b12 are officially supported.
- Use black with pre-commit for code formatting (#244).

7.2.48 3.0.2 (2018-07-05)

Bug fixes:

- Fix compatibility with marshmallow 3.0.0b12 (#242). Thanks @lafrech.

7.2.49 3.0.1 (2018-06-06)

Bug fixes:

- Respect `Parser.DEFAULT_VALIDATION_STATUS` when a `status_code` is not explicitly passed to `ValidationError` (#180). Thanks @foresmac for finding this.

Support:

- Add "Returning HTTP 400 Responses" section to docs (#180).

7.2.50 3.0.0 (2018-05-06)

Changes:

- *Backwards-incompatible*: Custom error handlers receive the request object as the second argument. Update any functions decorated with `Parser.error_handler` to take a `req` argument, like so:

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


# 3.x
@parser.error_handler
def handle_error(error, req):
    raise CustomError(error.messages)
```

- *Backwards-incompatible*: Remove unused `instance` and `kwargs` arguments of `argmap2schema`.
- *Backwards-incompatible*: Remove `Parser.load` method (`Parser` now calls `Schema.load` directly).

These changes shouldn't affect most users. However, they might break custom parsers calling these methods. (#222)

- Drop support for aiohttp<3.0.0.

7.2.51 2.1.0 (2018-04-01)

Features:

- Respect `data_key` field argument (in marshmallow 3). Thanks @lafrech.

7.2.52 2.0.0 (2018-02-08)

Changes:

- Drop support for aiohttp<2.0.0.
- Remove use of deprecated `Request.has_body` attribute in aiohttpparser (#186). Thanks @ariddell for reporting.

7.2.53 1.10.0 (2018-02-08)

Features:

- Add support for marshmallow>=3.0.0b7 (#188). Thanks @lafrech.

Deprecations:

- Support for aiohttp<2.0.0 is deprecated and will be removed in webargs 2.0.0.

7.2.54 1.9.0 (2018-02-03)

Changes:

- `HTTPExceptions` raised with *`webargs.flaskparser.abort`* will always have the `data` attribute, even if no additional keywords arguments are passed (#184). Thanks @lafrech.

Support:

- Fix examples in examples/ directory.

7.2.55 1.8.1 (2017-07-17)

Bug fixes:

- Fix behavior of `AIOHTTPParser.use_args` when `as_kwargs=True` is passed with a `Schema` (#179). Thanks @Itayazolay.

7.2.56 1.8.0 (2017-07-16)

Features:

- `AIOHTTPParser` supports class-based views, i.e. `aiohttp.web.View` (#177). Thanks @daniel98321.

7.2.57 1.7.0 (2017-06-03)

Features:

- `AIOHTTPParser.use_args` and `AIOHTTPParser.use_kwargs` work with `async def` coroutines (#170). Thanks @zaro.

7.2.58 1.6.3 (2017-05-18)

Support:

- Fix Flask error handling docs in "Framework support" section (#168). Thanks @nebularazer.

7.2.59 1.6.2 (2017-05-16)

Bug fixes:

- Fix parsing multiple arguments in `AIOHTTParser` (#165). Thanks @ariddell for reporting and thanks @zaro for reporting.

7.2.60 1.6.1 (2017-04-30)

Bug fixes:

- Fix form parsing in aiohttp>=2.0.0. Thanks @DmitriyS for the PR.

7.2.61 1.6.0 (2017-03-14)

Bug fixes:

- Fix compatibility with marshmallow 3.x.

Other changes:

- Drop support for Python 2.6 and 3.3.
- Support marshmallow>=2.7.0.

7.2.62 1.5.3 (2017-02-04)

Bug fixes:

- Port fix from release 1.5.2 to `AsyncParser`. This fixes #146 for `AIOHTTPParser`.
- Handle invalid types passed to `DelimitedList` (#149). Thanks @psconnect-dev for reporting.

7.2.63 1.5.2 (2017-01-08)

Bug fixes:

- Don't add `marshmallow.missing` to `original_data` when using `marshmallow.validates_schema(pass_original=True)` (#146). Thanks @lafrech for reporting and for the fix.

Other changes:

- Test against Python 3.6.

7.2.64 1.5.1 (2016-11-27)

Bug fixes:

- Fix handling missing nested args when `many=True` (#120, #145). Thanks @chavz and @Bangertm for reporting.
- Fix behavior of `load_from` in `AIOHTTPParser`.

7.2.65 1.5.0 (2016-11-22)

Features:

- The `use_args` and `use_kwargs` decorators add a reference to the undecorated function via the `__wrapped__` attribute. This is useful for unit-testing purposes (#144). Thanks @EFF for the PR.

Bug fixes:

- If `load_from` is specified on a field, first check the field name before checking `load_from` (#118). Thanks @jasonab for reporting.

7.2.66 1.4.0 (2016-09-29)

Bug fixes:

- Prevent error when rendering validation errors to JSON in Flask (e.g. when using Flask-RESTful) (#122). Thanks @frol for the catch and patch. NOTE: Though this is a bugfix, this is a potentially breaking change for code that needs to access the original `ValidationError` object.

```
# Before
@app.errorhandler(422)
def handle_validation_error(err):
    return jsonify({"errors": err.messages}), 422


# After
@app.errorhandler(422)
def handle_validation_error(err):
    # The marshmallow.ValidationError is available on err.exc
    return jsonify({"errors": err.exc.messages}), 422
```

7.2.67 1.3.4 (2016-06-11)

Bug fixes:

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

7.2.68 1.3.3 (2016-05-29)

Bug fixes:

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

7.2.69 1.3.2 (2016-04-14)

Bug fixes:

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

7.2.70 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.

7.2.71 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.

7.2.72 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.

7.2.73 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`.

7.2.74 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.

7.2.75 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.

7.2.76 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!

7.2.77 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`.

7.2.78 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
```

(continues on next page)

(continued from previous page)

```
args = {
    "name": fields.Str(required=True),
    "password": fields.Str(validate=lambda p: len(p) >= 6),
    "display_per_page": fields.Int(load_default=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.

7.2.79 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.

7.2.80 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).

7.2.81 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`.

7.2.82 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.

7.2.83 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`.

7.2.84 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.

7.2.85 0.9.1 (2014-12-11)

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

7.2.86 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.

7.2.87 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.

7.2.88 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.

7.2.89 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.

7.2.90 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.

7.2.91 0.6.1 (2014-09-28)

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

7.2.92 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`.

7.2.93 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.

7.2.94 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.

7.2.95 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.

7.2.96 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`.

7.2.97 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.

7.2.98 0.3.2 (2014-03-04)

- Fix bug with parsing JSON in Flask and Bottle.

7.2.99 0.3.1 (2014-03-03)

- Remove print statements in core.py. Oops.

7.2.100 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`.

7.2.101 0.2.0 (2014-02-26)

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

7.2.102 0.1.0 (2014-02-16)

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

7.3 Upgrading to Newer Releases

This section documents migration paths to new releases.

7.3.1 Upgrading to 8.0

In 8.0, the default values for `unknown` were changed. When the location is set to `json`, `form`, or `json_or_form`, the default for `unknown` is now `None`. Previously, the default was `RAISE`.

Because `RAISE` is the default value for `unknown` on marshmallow schemas, this change only affects usage in which the following conditions are met:

- A schema with `unknown` set to `INCLUDE` or `EXCLUDE` is passed to webargs `use_args`, `use_kwargs`, or `parse`
- `unknown` is not passed explicitly to the webargs function
- `location` is not set (default of `json`) or is set explicitly to `json`, `form`, or `json_or__form`

For example

```
import marshmallow as ma


class BodySchema(ma.Schema):
    foo = ma.fields.String()

    class Meta:
        unknown = ma.EXCLUDE


@parser.use_args(BodySchema)
def foo(data):
    ...
```

In this case, under webargs 7.0 the schema `unknown` setting of `EXCLUDE` would be ignored. Instead, `unknown=RAISE` would be used.

In webargs 8.0, the schema `unknown` is used.

To get the webargs 7.0 behavior (overriding the Schema `unknown`), simply pass `unknown` to `use_args`, as in

```
@parser.use_args(BodySchema, unknown=ma.RAISE)
def foo(data):
    ...
```

7.3.2 Upgrading to 7.0

`unknown` is Now Settable by the Parser

As of 7.0, `Parsers` have multiple settings for controlling the value for `unknown` which is passed to `schema.load` when parsing.

To set unknown behavior on a parser, see the advanced doc on this topic: *Setting unknown*.

Importantly, by default, any schema setting for `unknown` will be overridden by the `unknown` settings for the parser.

In order to use a schema's `unknown` value, set `unknown=None` on the parser. In 6.x versions of webargs, schema values for `unknown` are used, so the `unknown=None` setting is the best way to emulate this.

To get identical behavior:

```
# assuming you have a schema named MySchema


# webargs 6.x
@parser.use_args(MySchema)
def foo(args):
    ...


# webargs 7.x
# as a parameter to use_args or parse
@parser.use_args(MySchema, unknown=None)
def foo(args):
    ...


# webargs 7.x
# as a parser setting
# example with flaskparser, but any parser class works
parser = FlaskParser(unknown=None)


@parser.use_args(MySchema)
def foo(args):
    ...
```

7.3.3 Upgrading to 6.0

Multiple Locations Are No Longer Supported In A Single Call

The default location is JSON/body.

Under webargs 5.x, code often did not have to specify a location.

Because webargs would parse data from multiple locations automatically, users did not need to specify where a parameter, call it `q`, was passed. `q` could be in a query parameter or in a JSON or form-post body.

Now, webargs requires that users specify only one location for data loading per `use_args` call, and `"json"` is the default. If `q` is intended to be a query parameter, the developer must be explicit and rewrite like so:

```
# webargs 5.x
@parser.use_args({"q": ma.fields.String()})
def foo(args):
    return some_function(user_query=args.get("q"))


# webargs 6.x
@parser.use_args({"q": ma.fields.String()}, location="query")
```

(continues on next page)

(continued from previous page)

```
def foo(args):
    return some_function(user_query=args.get("q"))
```

This also means that another usage from 5.x is not supported. Code with multiple locations in a single `use_args`, `use_kwargs`, or `parse` call must be rewritten in multiple separate `use_args` or `use_kwargs` invocations, like so:

```
# webargs 5.x
@parser.use_kwargs(
    {
        "q1": ma.fields.Int(location="query"),
        "q2": ma.fields.Int(location="query"),
        "h1": ma.fields.Int(location="headers"),
    },
    locations=("query", "headers"),
)
def foo(q1, q2, h1):
    ...


# webargs 6.x
@parser.use_kwargs({"q1": ma.fields.Int(), "q2": ma.fields.Int()}, location="query")
@parser.use_kwargs({"h1": ma.fields.Int()}, location="headers")
def foo(q1, q2, h1):
    ...
```

Fields No Longer Support location=...

Because a single `parser.use_args`, `parser.use_kwargs`, or `parser.parse` call cannot specify multiple locations, it is not necessary for a field to be able to specify its location. Rewrite code like so:

```
# webargs 5.x
@parser.use_args({"q": ma.fields.String(location="query")})
def foo(args):
    return some_function(user_query=args.get("q"))


# webargs 6.x
@parser.use_args({"q": ma.fields.String()}, location="query")
def foo(args):
    return some_function(user_query=args.get("q"))
```

location_handler Has Been Replaced With location_loader

This is not just a name change. The expected signature of a `location_loader` is slightly different from the signature for a `location_handler`.

Where previously a `location_handler` code took the incoming request data and details of a single field being loaded, a `location_loader` takes the request and the schema as a pair. It does not return a specific field's data, but data for the whole location.

Rewrite code like this:

```
# webargs 5.x
@parser.location_handler("data")
def load_data(request, name, field):
    return request.data.get(name)


# webargs 6.x
@parser.location_loader("data")
def load_data(request, schema):
    return request.data
```

Data Is Not Filtered Before Being Passed To Schemas, And It May Be Proxified

In webargs 5.x, the deserialization schema was used to pull data out of the request object. That data was compiled into a dictionary which was then passed to the schema.

One of the major changes in webargs 6.x allows the use of `unknown` parameter on schemas. This lets a schema decide what to do with fields not specified in the schema. In order to achieve this, webargs now passes the full data from the specified location to the schema.

Therefore, users should specify `unknown=marshmallow.EXCLUDE` on their schemas in order to filter out unknown fields. Like so:

```
# webargs 5.x
# this can assume that "q" is the only parameter passed, and all other
# parameters will be ignored
@parser.use_kwargs({"q": ma.fields.String()}, locations=("query",))
def foo(q):
    ...


# webargs 6.x, Solution 1: declare a schema with Meta.unknown set
class QuerySchema(ma.Schema):
    q = ma.fields.String()

    class Meta:
        unknown = ma.EXCLUDE


@parser.use_kwargs(QuerySchema, location="query")
def foo(q):
    ...


# webargs 6.x, Solution 2: instantiate a schema with unknown set
class QuerySchema(ma.Schema):
    q = ma.fields.String()


@parser.use_kwargs(QuerySchema(unknown=ma.EXCLUDE), location="query")
def foo(q):
    ...
```

This also allows usage which passes the unknown parameters through, like so:

```
# webargs 6.x only! cannot be done in 5.x
class QuerySchema(ma.Schema):
    q = ma.fields.String()


# will pass *all* query params through as "kwargs"
@parser.use_kwargs(QuerySchema(unknown=ma.INCLUDE), location="query")
def foo(q, **kwargs):
    ...
```

However, many types of request data are so-called "multidicts" – dictionary-like types which can return one or multiple values. To handle `marshmallow.fields.List` and *`webargs.fields.DelimitedList`* fields correctly, passing list data, webargs must combine schema information with the raw request data. This is done in the *`MultiDictProxy`* type, which will often be passed to schemas.

This means that if a schema has a `pre_load` hook which interacts with the data, it may need modifications. For example, a `flask` query string will be parsed into an `ImmutableMultiDict` type, which will break pre-load hooks which modify the data in-place. Such usages need rewrites like so:

```
# webargs 5.x
# flask query params is just an example -- applies to several types
from webargs.flaskparser import use_kwargs


class QuerySchema(ma.Schema):
    q = ma.fields.String()

    @ma.pre_load
    def convert_nil_to_none(self, obj, **kwargs):
        if obj.get("q") == "nil":
            obj["q"] = None
        return obj


@use_kwargs(QuerySchema, locations=("query",))
def foo(q):
    ...


# webargs 6.x
class QuerySchema(ma.Schema):
    q = ma.fields.String()

    # unlike under 5.x, we cannot modify 'obj' in-place because writing
    # to the MultiDictProxy will try to write to the underlying
    # ImmutableMultiDict, which is not allowed
    @ma.pre_load
    def convert_nil_to_none(self, obj, **kwargs):
        # creating a dict from a MultiDictProxy works well because it
        # "unwraps" lists and delimited lists correctly
        data = dict(obj)
        if data.get("q") == "nil":
```

(continues on next page)

(continued from previous page)

```
            data["q"] = None
        return data


@parser.use_kwargs(QuerySchema, location="query")
def foo(q):
    ...
```

DelimitedList Now Only Takes A String Input

Combining `List` and string parsing functionality in a single type had some messy corner cases. For the most part, this should not require rewrites. But for APIs which need to allow both usages, rewrites are possible like so:

```
# webargs 5.x
# this allows ...?x=1&x=2&x=3
# as well as ...?x=1,2,3
@use_kwargs({"x": webargs.fields.DelimitedList(ma.fields.Int)}, locations=("query",))
def foo(x):
    ...


# webargs 6.x
# this accepts x=1,2,3 but NOT x=1&x=2&x=3
@use_kwargs({"x": webargs.fields.DelimitedList(ma.fields.Int)}, location="query")
def foo(x):
    ...


# webargs 6.x
# this accepts x=1,2,3 ; x=1&x=2&x=3 ; x=1,2&x=3
# to do this, it needs a post_load hook which will flatten out the list data
class UnpackingDelimitedListSchema(ma.Schema):
    x = ma.fields.List(webargs.fields.DelimitedList(ma.fields.Int))

    @ma.post_load
    def flatten_lists(self, data, **kwargs):
        new_x = []
        for x in data["x"]:
            new_x.extend(x)
        data["x"] = new_x
        return data


@parser.use_kwargs(UnpackingDelimitedListSchema, location="query")
def foo(x):
    ...
```

ValidationError Messages Are Namespaced Under The Location

Code parsing ValidationError messages will notice a change in the messages produced by webargs. What would previously have come back with messages like `{"foo":["Not a valid integer."]}` will now have messages nested one layer deeper, like `{"json":{"foo":["Not a valid integer."]}}`.

To rewrite code which was handling these errors, the handler will need to be prepared to traverse messages by one additional level. For example:

```
import logging

log = logging.getLogger(__name__)


# webargs 5.x
# logs debug messages like
#   bad value for 'foo': ["Not a valid integer."]
#   bad value for 'bar': ["Not a valid boolean."]
def log_invalid_parameters(validation_error):
    for field, messages in validation_error.messages.items():
        log.debug("bad value for '{}': {}".format(field, messages))


# webargs 6.x
# logs debug messages like
#   bad value for 'foo' [query]: ["Not a valid integer."]
#   bad value for 'bar' [json]: ["Not a valid boolean."]
def log_invalid_parameters(validation_error):
    for location, fielddata in validation_error.messages.items():
        for field, messages in fielddata.items():
            log.debug("bad value for '{}' [{}]: {}".format(field, location, messages))
```

Custom Error Handler Argument Names Changed

If you define a custom error handler via `@parser.error_handler` the function arguments are now keyword-only and `status_code` and `headers` have been renamed `error_status_code` and `error_headers`.

```
# webargs 5.x
@parser.error_handler
def custom_handle_error(error, req, schema, status_code, headers):
    ...


# webargs 6.x
@parser.error_handler
def custom_handle_error(error, req, schema, *, error_status_code, error_headers):
    ...
```

Some Functions Take Keyword-Only Arguments Now

The signature of several methods has changed to have keyword-only arguments. For the most part, this should not require any changes, but here's a list of the changes.

`parser.error_handler` methods:

```
# webargs 5.x
def handle_error(error, req, schema, status_code, headers):
    ...


# webargs 6.x
def handle_error(error, req, schema, *, error_status_code, error_headers):
    ...
```

`parser.__init__` methods:

```
# webargs 5.x
def __init__(self, location=None, error_handler=None, schema_class=None):
    ...


# webargs 6.x
def __init__(self, location=None, *, error_handler=None, schema_class=None):
    ...
```

`parser.parse`, `parser.use_args`, and `parser.use_kwargs` methods:

```
# webargs 5.x
def parse(
    self,
    argmap,
    req=None,
    location=None,
    validate=None,
    error_status_code=None,
    error_headers=None,
):
    ...


# webargs 6.x
def parse(
    self,
    argmap,
    req=None,
    *,
    location=None,
    validate=None,
    error_status_code=None,
    error_headers=None,
):
    ...
```

(continues on next page)

(continued from previous page)

```
# webargs 5.x
def use_args(
    self,
    argmap,
    req=None,
    location=None,
    as_kwargs=False,
    validate=None,
    error_status_code=None,
    error_headers=None,
):
    ...


# webargs 6.x
def use_args(
    self,
    argmap,
    req=None,
    *,
    location=None,
    as_kwargs=False,
    validate=None,
    error_status_code=None,
    error_headers=None,
):
    ...


# use_kwargs is just an alias for use_args with as_kwargs=True
```

and finally, the `dict2schema` function:

```
# webargs 5.x
def dict2schema(dct, schema_class=ma.Schema):
    ...


# webargs 6.x
def dict2schema(dct, *, schema_class=ma.Schema):
    ...
```

PyramidParser Now Appends Arguments (Used To Prepend)

`PyramidParser.use_args` was not conformant with the other parsers in webargs. While all other parsers added new arguments to the end of the argument list of a decorated view function, the Pyramid implementation added them to the front of the argument list.

This has been corrected, but as a result pyramid views with `use_args` may need to be rewritten. The `request` object is always passed first in both versions, so the issue is only apparent with view functions taking other positional arguments.

For example, imagine code with a decorator for passing user information, `pass_userinfo`, like so:

```
# a decorator which gets information about the authenticated user
def pass_userinfo(f):
    def decorator(request, *args, **kwargs):
        return f(request, get_userinfo(), *args, **kwargs)

    return decorator
```

You will see a behavioral change if `pass_userinfo` is called on a function decorated with `use_args`. The difference between the two versions will be like so:

```
from webargs.pyramidparser import use_args


# webargs 5.x
# pass_userinfo is called first, webargs sees positional arguments of
#   (userinfo,)
# and changes it to
#   (request, args, userinfo)
@pass_userinfo
@use_args({"q": ma.fields.String()}, locations=("query",))
def viewfunc(request, args, userinfo):
    q = args.get("q")
    ...


# webargs 6.x
# pass_userinfo is called first, webargs sees positional arguments of
#   (userinfo,)
# and changes it to
#   (request, userinfo, args)
@pass_userinfo
@use_args({"q": ma.fields.String()}, location="query")
def viewfunc(request, userinfo, args):
    q = args.get("q")
    ...
```

7.4 Authors

7.4.1 Lead

- Steven Loria @sloria
- Jérôme Lafréchoux @lafrech
- Stephen Rosen @sirosen

7.4.2 Contributors (chronological)

- Steven Manuatu @venuatu
- Javier Santacruz @jvrsantacruz
- Josh Carp @jmcarp
- @philtay
- Andriy Yurchuk @Ch00k
- Stas Suşcov @stas
- Josh Johnston @Trii
- Rory Hart @hartror
- Jace Browning @jacebrowning
- marcellarius @marcellarius
- Damian Heard @DamianHeard
- Daniel Imhoff @dwieeb
- @immerrr
- Brett Higgins @brettdh
- Vlad Frolov @frol
- Tuukka Mustonen @tuukkamustonen
- Francois-Xavier Darveau @EFF
- Jérôme Lafréchoux @lafrech
- @DmitriyS
- Svetlozar Argirov @zaro
- Florian S. @nebularazer
- @daniel98321
- @Itayazolay
- @Reskov
- @cedzz
- F. Moukayed () @kochab
- Xiaoyu Lee @lee3164
- Jonathan Angelo @jangelo

- @zhenhua32
- Martin Roy @lindycoder
- Kubilay Kocak @koobs
- @dodumosu
- Nate Dellinger @Nateyo
- Karthikeyan Singaravelan @tirkarthi
- Sami Salonen @suola
- Tim Gates @timgates42
- Lefteris Karapetsas @lefterisjp
- Utku Gultopu @ugultopu
- Jason Williams @jaswilli
- Grey Li @greyli
- @michaelizergit
- Legolas Bloom @TTWShell
- Kevin Kirsche @kkirsche
- Isira Seneviratne @Isira-Seneviratne
- Anton Ostapenko @AVOstap

7.5 Contributing Guidelines

7.5.1 Security Contact Information

To report a security vulnerability, please use the Tidelift security contact. Tidelift will coordinate the fix and disclosure.

7.5.2 Questions, Feature Requests, Bug Reports, and Feedback…

…should all be reported on the GitHub Issue Tracker .

7.5.3 Contributing Code

Integration with a Another Web Framework…

…should be released as a separate package.

Pull requests adding support for another framework will not be accepted. In order to keep webargs small and easy to maintain, we are not currently adding support for more frameworks. Instead, release your framework integration as a separate package and add it to the Ecosystem page in the GitHub wiki .

Setting Up for Local Development

1. Fork webargs on GitHub.

```
$ git clone https://github.com/marshmallow-code/webargs.git
$ cd webargs
```

2. Install development requirements. **It is highly recommended that you use a virtualenv.** Use the following command to install an editable version of webargs along with its development requirements.

```
# After activating your virtualenv
$ pip install -e '.[dev]'
```

3. (Optional, but recommended) Install the pre-commit hooks, which will format and lint your git staged files.

```
# The pre-commit CLI was installed above
$ pre-commit install
```

Git Branch Structure

Webargs abides by the following branching model:

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

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 `marshmallow-code:dev` or the appropriate maintenance branch. The CI build must be passing before your pull request is merged.

Running Tests

To run all tests:

```
$ pytest
```

To run syntax checks:

```
$ tox -e lint
```

(Optional) To run tests in all supported Python versions in their own virtual environments (must have each interpreter installed):

```
$ tox
```

Documentation

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

To build the docs in "watch" mode:

```
$ tox -e watch-docs
```

Changes in the `docs/` directory will automatically trigger a rebuild.

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.

PYTHON MODULE INDEX

W

INDEX

K

L