Falcon Documentation

Release 4.2.0

Kurt Griffiths et al.

Nov 10, 2025

CONTENTS

Release v4.2 (*Installation*)

Falcon is a minimalist ASGI/WSGI framework for building mission-critical REST APIs and microservices, with a focus on reliability, correctness, and performance at scale.

We like to think of Falcon as the *Dieter Rams* of web frameworks. Falcon encourages the REST architectural style, and tries to do as little as possible while remaining highly effective.

```
import falcon


class QuoteResource:
    def on_get(self, req: falcon.Request, resp: falcon.Response) -> None:
        """Handle GET requests."""
        resp.media = {
            'quote': "I've always been more interested in the future than in the␣
↪past.",
            'author': 'Grace Hopper',
        }


app = falcon.App()
app.add_route('/quote', QuoteResource())
```

For a fully working example, check out the *Quickstart*.

CHAPTER

ONE

QUICK LINKS

- Read the docs (FAQ - getting help - reference)
- Falcon add-ons and complementary packages
- Falcon articles, talks and podcasts
- falconry/user for Falcon users @ Gitter
- falconry/dev for Falcon contributors @ Gitter

CHAPTER

TWO

WHAT PEOPLE ARE SAYING

“Falcon is rock solid and it’s fast.”

“We have been using Falcon as a replacement for [another framework] and we simply love the performance (three times faster) and code base size (easily half of our [original] code).”

“I’m loving #falconframework! Super clean and simple, I finally have the speed and flexibility I need!”

“Falcon looks great so far. I hacked together a quick test for a tiny server of mine and was ~40% faster with only 20 minutes of work.”

“I feel like I’m just talking HTTP at last, with nothing in the middle. Falcon seems like the requests of backend.”

“The source code for Falcon is so good, I almost prefer it to documentation. It basically can’t be wrong.”

“What other framework has integrated support for 786 TRY IT NOW ?”

CHAPTER

THREE

FEATURES

Falcon tries to do as little as possible while remaining highly effective.

- *ASGI, WSGI*, and *WebSocket* support
- Native `asyncio` support
- No reliance on magic globals for routing and state management
- Stable interfaces with an emphasis on backwards-compatibility
- Simple API modeling through centralized RESTful *routing*
- Highly-optimized, extensible code base
- Easy access to headers and bodies through *request and response* objects
- DRY request processing via *middleware* components and hooks
- Strict adherence to RFCs
- Idiomatic *HTTP error* responses
- Straightforward exception handling
- Snappy *testing* with WSGI/ASGI helpers and mocks
- CPython 3.9+ and PyPy 3.9+ support

CHAPTER

FOUR

WHO'S USING FALCON?

Falcon is used around the world by a growing number of organizations, including:

- 7ideas
- Cronitor
- EMC
- Hurricane Electric
- Leadpages
- OpenStack
- Rackspace
- Shiftgig
- tempfil.es
- Opera Software

If you are using the Falcon framework for a community or commercial project, please consider adding your information to our wiki under Who's Using Falcon?

You might also like to view our Add-on Catalog, where you can find a list of add-ons maintained by the community.

CHAPTER

FIVE

DOCUMENTATION

5.1 User Guide

5.1.1 Introduction

> Perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away.
>
> - *Antoine de Saint-Exupéry*

Falcon is a reliable, high-performance Python web framework for building large-scale app backends and microservices. It encourages the REST architectural style, and tries to do as little as possible while remaining highly effective.

Falcon apps work with any WSGI server, and run like a champ under CPython 3.9+ and PyPy 3.9+.

Features

Falcon tries to do as little as possible while remaining highly effective.

- *ASGI, WSGI*, and *WebSocket* support
- Native `asyncio` support
- No reliance on magic globals for routing and state management
- Stable interfaces with an emphasis on backwards-compatibility
- Simple API modeling through centralized RESTful *routing*
- Highly-optimized, extensible code base
- Easy access to headers and bodies through *request and response* objects
- DRY request processing via *middleware* components and hooks
- Strict adherence to RFCs
- Idiomatic *HTTP error* responses
- Straightforward exception handling
- Snappy *testing* with WSGI/ASGI helpers and mocks
- CPython 3.9+ and PyPy 3.9+ support

How is Falcon different?

We designed Falcon to support the demanding needs of large-scale microservices and responsive app backends. Falcon complements more general Python web frameworks by providing bare-metal performance, reliability, and flexibility wherever you need it.

Fast. Same hardware, more requests. Falcon turns around requests several times faster than most other Python frameworks. For an extra speed boost, Falcon compiles itself with Cython when available, and also works well with PyPy. Considering a move to another programming language? Benchmark with Falcon + PyPy first.

Reliable. We go to great lengths to avoid introducing breaking changes, and when we do they are fully documented and only introduced (in the spirit of *SemVer*) with a major version increment. The code is rigorously tested with numerous inputs and we require 100% coverage at all times. Falcon does not depend on any external Python packages.

Debuggable. Falcon eschews magic. It's easy to tell which inputs lead to which outputs. To avoid incentivizing the use of hard-to-debug global state, Falcon does not use decorators to define routes. Unhandled exceptions are never encapsulated or masked. Potentially surprising behaviors, such as automatic request body parsing, are well-documented and disabled by default. Finally, we take care to keep logic paths within the framework simple, shallow and understandable. All of this makes it easier to reason about the code and to debug edge cases in large-scale deployments.

Flexible. Falcon leaves a lot of decisions and implementation details to you, the API developer. This gives you a lot of freedom to customize and tune your implementation. Due to Falcon's minimalist design, Python community members are free to independently innovate on Falcon add-ons and complementary packages.

About Apache 2.0

Falcon is released under the terms of the Apache 2.0 License. This means that you can use it in your commercial applications without having to also open-source your own code. It also means that if someone happens to contribute code that is associated with a patent, you are granted a free license to use said patent. That's a pretty sweet deal.

Now, if you do make changes to Falcon itself, please consider contributing your awesome work back to the community.

Falcon License

5.1.2 Installation

PyPy

PyPy is the fastest way to run your Falcon app. PyPy3.9+ is supported as of PyPy v7.3.10.

```
$ pip install falcon
```

Or, to install the latest beta or release candidate, if any:

```
$ pip install --pre falcon
```

CPython

Falcon fully supports CPython 3.9+.

The latest stable version of Falcon can be installed directly from PyPI:

```
$ pip install falcon
```

Or, to install the latest beta or release candidate, if any:

```
$ pip install --pre falcon
```

In order to provide an extra speed boost, Falcon automatically compiles itself with Cython. Wheels containing pre-compiled binaries are available from PyPI for the majority of common platforms (see *Binary Wheels* below for the complete list of the platforms that we target, or simply check Falcon files on PyPI).

However, even if a binary build for your platform of choice is not available, you can choose to stick with the generic pure-Python wheel (that `pip` should pick automatically), or cythonize Falcon for your environment (see *instructions below*). The pure-Python version is functionally identical to binary wheels; it is just slower on CPython.

Cythonizing Falcon

Falcon leverages PEP 517 to automatically compile (AKA *cythonize*) itself with Cython whenever it is installed from the source distribution. So if a suitable *binary wheel* is unavailable for your platform, or if you want to recompile locally, you simply need to instruct `pip` not to use prebuilt wheels:

```
$ pip install --no-binary :all: falcon
```

If you want to verify that Cython is being invoked, pass `-v` to `pip` in order to echo the compilation commands:

```
$ pip install -v --no-binary :all: falcon
```

Apart from the obvious requirement to have a functional compiler toolchain set up with CPython development headers, the only inconvenience of running cythonization on your side is the extra couple of minutes it takes (depending on your hardware; it can take much more on an underpowered CI runner, or if you are using emulation to prepare your software for another architecture).

Furthermore, you can also cythonize the latest developmental snapshot Falcon directly from the *source code* on GitHub:

```
$ pip install git+https://github.com/falconry/falcon/
```

Danger

Although we try to keep the main development branch in a good shape at all times, we strongly recommend to use only stable versions of Falcon in production.

Compiling on Mac OS

Tip

Pre-compiled Falcon wheels are available for macOS on Apple Silicon chips, so normally you should be fine with just `pip install falcon`.

Xcode Command Line Tools are required to compile Cython. Install them with this command:

```
$ xcode-select --install
```

The Clang compiler treats unrecognized command-line options as errors, for example:

```
clang: error: unknown argument: '-mno-fused-madd' [-Wunused-command-line-argument-
↪hard-error-in-future]
```

You might also see warnings about unused functions. You can work around these issues by setting additional Clang C compiler flags as follows:

```
$ export CFLAGS="-Qunused-arguments -Wno-unused-function"
```

Binary Wheels

Binary Falcon wheels are automatically built for many CPython platforms, courtesy of cibuildwheel.

The following table summarizes the wheel availability on different combinations of CPython versions vs CPython platforms:

Platform / CPython version	3.9	3.10	3.11	3.12	3.13	3.14
Linux Intel `manylinux` 64-bit		✓	✓	✓	✓	✓[1]
Linux Intel `musllinux` 64-bit		✓	✓	✓	✓	✓[1]
Linux Intel `manylinux` 32-bit						
Linux Intel `musllinux` 32-bit						
Linux ARM `manylinux` 64-bit		✓	✓	✓	✓	✓[1]
Linux ARM `musllinux` 64-bit		✓	✓	✓	✓	✓[1]
Linux PowerPC `manylinux` 64-bit						
Linux PowerPC `musllinux` 64-bit						
Linux IBM Z `manylinux`		✓	✓	✓	✓	✓
Linux IBM Z `musllinux`						
macOS Intel						
macOS Apple Silicon		✓	✓	✓	✓	✓
Windows 32-bit						
Windows 64-bit		✓	✓	✓	✓	✓
Windows ARM 64-bit						

[1] *A binary wheel is also available for free-threaded CPython.*

> **Note**
>
> As of Falcon *4.2.0*, free-threaded build was enabled for selected Linux x86 wheels. Other CPython platforms can still utilize free-threading using the pure Python wheel.
>
> See also: *Can I run Falcon on free-threaded CPython?*

While we believe that our build configuration covers the most common development and deployment scenarios, *let us know* if you are interested in any builds that are currently missing from our selection!

Dependencies

Falcon does not require the installation of any other packages.

WSGI Server

Falcon speaks WSGI, and so in order to serve a Falcon app, you will need a WSGI server. Gunicorn and uWSGI are some of the more popular ones out there, but anything that can load a WSGI app will do.

Windows users can try Waitress, a production-quality, pure-Python WSGI server. Other alternatives on Windows include running Gunicorn and uWSGI via WSL, as well as inside Linux Docker containers.

```
$ pip install [gunicorn|uwsgi|waitress]
```

ASGI Server

Conversely, in order to run an `async` *`Falcon ASGI app`*, you will need an ASGI application server (Falcon only supports ASGI 3.0+, aka the single-callable application style).

Uvicorn is a popular choice, owing to its fast and stable implementation. What is more, Uvicorn is supported on Windows, and on PyPy (however, both at a performance loss compared to CPython on Unix-like systems).

Falcon is also regularly tested against Daphne, the current ASGI reference server.

For a more in-depth overview of available servers, see also: ASGI Implementations.

```
$ pip install [uvicorn|daphne|hypercorn]
```

Note

By default, the `uvicorn` package comes only with a minimal set of pure-Python dependencies. For CPython-based production deployments, you can install Uvicorn along with more optimized alternatives such as `uvloop` (a faster event loop), `httptools` (a faster HTTP protocol implementation) etc:

```
$ pip install uvicorn[standard]
```

See also a longer explanation on Uvicorn's website: Quickstart.

Source Code

Falcon lives on GitHub, making the code easy to browse, download, fork, etc. *Pull requests* are always welcome! Also, please remember to star the project if it makes you happy. :)

Once you have cloned the repo or downloaded a tarball from GitHub, you can install Falcon like this:

```
$ # Clone over SSH:
$ #   git clone git@github.com:falconry/falcon.git
$ # Or, if you prefer, over HTTPS:
$ #   git clone https://github.com/falconry/falcon
$ cd falcon
$ pip install .
```

Tip

The above command will automatically install the *cythonized* version of Falcon. If you just want to experiment with the latest snapshot, you can skip the cythonization step by setting the `FALCON_DISABLE_CYTHON` environment variable to a non-empty value:

```
$ cd falcon
$ FALCON_DISABLE_CYTHON=Y pip install .
```

Or, if you want to edit the code, first fork the main repo, clone the fork to your desktop, and then run the following command to install it using symbolic linking, so that when you change your code, the changes will be automagically available to your app without having to reinstall the package:

```
$ cd falcon
$ FALCON_DISABLE_CYTHON=Y pip install -e .
```

You can manually test changes to the Falcon framework by switching to the directory of the cloned repo and then running pytest:

```
$ cd falcon
$ FALCON_DISABLE_CYTHON=Y pip install -e .
```

(continues on next page)

(continued from previous page)

```
$ pip install -r requirements/tests
$ pytest tests
```

Or, to run the default set of tests:

```
$ pip install tox && tox
```

> **Tip**
>
> See also the tox.ini file for a full list of available environments.

Finally, to build Falcon's docs from source, simply run:

```
$ pip install tox && tox -e docs
```

Once the docs have been built, you can view them by opening the following index page in your browser. On OS X it's as simple as:

```
$ open docs/_build/html/index.html
```

Or on Linux:

```
$ xdg-open docs/_build/html/index.html
```

5.1.3 Quickstart

If you haven't done so already, please take a moment to *install* the Falcon web framework before continuing.

Learning by Example

Here is a simple example from Falcon's README, showing how to get started writing an app.

WSGI

```
# examples/things.py

# Let's get this party started!
from wsgiref.simple_server import make_server

import falcon


# Falcon follows the REST architectural style, meaning (among
# other things) that you think in terms of resources and state
# transitions, which map to HTTP verbs.
class ThingsResource:
    def on_get(self, req, resp):
        """Handles GET requests"""
        resp.status = falcon.HTTP_200  # This is the default status
        resp.content_type = falcon.MEDIA_TEXT  # Default is JSON, so override
        resp.text = (
            '\nTwo things awe me most, the starry sky '
            'above me and the moral law within me.\n'
            '\n'
            '    ~ Immanuel Kant\n\n'
```

(continues on next page)

(continued from previous page)

```
        )


# falcon.App instances are callable WSGI apps
# in larger applications the app is created in a separate file
app = falcon.App()

# Resources are represented by long-lived class instances
things = ThingsResource()

# things will handle all requests to the '/things' URL path
app.add_route('/things', things)

if __name__ == '__main__':
    with make_server('', 8000, app) as httpd:
        print('Serving on port 8000...')

        # Serve until process is killed
        httpd.serve_forever()
```

You can run the above example directly using the included wsgiref server:

```
$ pip install falcon
$ python things.py
```

Then, in another terminal:

```
$ curl localhost:8000/things
```

As an alternative to Curl, you might want to give HTTPie a try:

```
$ pip install --upgrade httpie
$ http localhost:8000/things
```

ASGI

```
# examples/things_asgi.py

import falcon
import falcon.asgi


# Falcon follows the REST architectural style, meaning (among
# other things) that you think in terms of resources and state
# transitions, which map to HTTP verbs.
class ThingsResource:
    async def on_get(self, req, resp):
        """Handles GET requests"""
        resp.status = falcon.HTTP_200  # This is the default status
        resp.content_type = falcon.MEDIA_TEXT  # Default is JSON, so override
        resp.text = (
            '\nTwo things awe me most, the starry sky '
            'above me and the moral law within me.\n'
            '\n'
            '    ~ Immanuel Kant\n\n'
        )
```

(continues on next page)

(continued from previous page)

```
# falcon.asgi.App instances are callable ASGI apps...
# in larger applications the app is created in a separate file
app = falcon.asgi.App()

# Resources are represented by long-lived class instances
things = ThingsResource()

# things will handle all requests to the '/things' URL path
app.add_route('/things', things)
```

You can run the ASGI version with uvicorn or any other ASGI server:

```
$ pip install falcon uvicorn
$ uvicorn things_asgi:app
```

Then, in another terminal:

```
$ curl localhost:8000/things
```

As an alternative to Curl, you might want to give HTTPie a try:

```
$ pip install --upgrade httpie
$ http localhost:8000/things
```

A More Complex Example

Here is a more involved example that demonstrates reading headers and query parameters, handling errors, and working with request and response bodies.

WSGI

Note that this example assumes that the requests package has been installed.

```
# examples/things_advanced.py

import json
import logging
import uuid
from wsgiref import simple_server

import requests

import falcon


class StorageEngine:
    def get_things(self, marker, limit):
        return [{'id': str(uuid.uuid4()), 'color': 'green'}]

    def add_thing(self, thing):
        thing['id'] = str(uuid.uuid4())
        return thing


class StorageError(Exception):
```

(continues on next page)

(continued from previous page)

```
    @staticmethod
    def handle(req, resp, ex, params):
        # TODO: Log the error, clean up, etc. before raising
        raise falcon.HTTPInternalServerError()


class SinkAdapter:
    engines = {
        'ddg': 'https://duckduckgo.com',
        'y': 'https://search.yahoo.com/search',
    }

    def __call__(self, req, resp, engine):
        url = self.engines[engine]
        params = {'q': req.get_param('q', True)}
        result = requests.get(url, params=params)

        resp.status = falcon.code_to_http_status(result.status_code)
        resp.content_type = result.headers['content-type']
        resp.text = result.text


class AuthMiddleware:
    def process_request(self, req, resp):
        token = req.get_header('Authorization')
        account_id = req.get_header('Account-ID')

        challenges = ['Token type="Fernet"']

        if token is None:
            description = 'Please provide an auth token as part of the request.'

            raise falcon.HTTPUnauthorized(
                title='Auth token required',
                description=description,
                challenges=challenges,
                href='http://docs.example.com/auth',
            )

        if not self._token_is_valid(token, account_id):
            description = (
                'The provided auth token is not valid. '
                'Please request a new token and try again.'
            )

            raise falcon.HTTPUnauthorized(
                title='Authentication required',
                description=description,
                challenges=challenges,
                href='http://docs.example.com/auth',
            )

    def _token_is_valid(self, token, account_id):
        return True  # Suuuuuure it's valid...
```

(continues on next page)

(continued from previous page)

```
class RequireJSON:
    def process_request(self, req, resp):
        if not req.client_accepts_json:
            raise falcon.HTTPNotAcceptable(
                description='This API only supports responses encoded as JSON.',
                href='http://docs.examples.com/api/json',
            )

        if req.method in ('POST', 'PUT'):
            if 'application/json' not in req.content_type:
                raise falcon.HTTPUnsupportedMediaType(
                    title='This API only supports requests encoded as JSON.',
                    href='http://docs.examples.com/api/json',
                )


class JSONTranslator:
    # NOTE: Normally you would simply use req.media and resp.media for
    # this particular use case; this example serves only to illustrate
    # what is possible.

    def process_request(self, req, resp):
        # req.stream corresponds to the WSGI wsgi.input environ variable,
        # and allows you to read bytes from the request body.
        #
        # See also: PEP 3333
        if req.content_length in (None, 0):
            # Nothing to do
            return

        body = req.stream.read()
        if not body:
            raise falcon.HTTPBadRequest(
                title='Empty request body',
                description='A valid JSON document is required.',
            )

        try:
            req.context.doc = json.loads(body.decode('utf-8'))

        except (ValueError, UnicodeDecodeError):
            description = (
                'Could not decode the request body. The '
                'JSON was incorrect or not encoded as '
                'UTF-8.'
            )

            raise falcon.HTTPBadRequest(title='Malformed JSON',␣
↪description=description)

    def process_response(self, req, resp, resource, req_succeeded):
        if not hasattr(resp.context, 'result'):
            return

        resp.text = json.dumps(resp.context.result)
```

(continues on next page)

(continued from previous page)

```
def max_body(limit):
    def hook(req, resp, resource, params):
        length = req.content_length
        if length is not None and length > limit:
            msg = (
                'The size of the request is too large. The body must not '
                'exceed ' + str(limit) + ' bytes in length.'
            )

            raise falcon.HTTPContentTooLarge(
                title='Request body is too large', description=msg
            )

    return hook


class ThingsResource:
    def __init__(self, db):
        self.db = db
        self.logger = logging.getLogger('thingsapp.' + __name__)

    def on_get(self, req, resp, user_id):
        marker = req.get_param('marker') or ''
        limit = req.get_param_as_int('limit') or 50

        try:
            result = self.db.get_things(marker, limit)
        except Exception as ex:
            self.logger.error(ex)

            description = (
                'Aliens have attacked our base! We will '
                'be back as soon as we fight them off. '
                'We appreciate your patience.'
            )

            raise falcon.HTTPServiceUnavailable(
                title='Service Outage', description=description, retry_after=30
            )

        # NOTE: Normally you would use resp.media for this sort of thing;
        # this example serves only to demonstrate how the context can be
        # used to pass arbitrary values between middleware components,
        # hooks, and resources.
        resp.context.result = result

        resp.set_header('Powered-By', 'Falcon')
        resp.status = falcon.HTTP_200

    @falcon.before(max_body(64 * 1024))
    def on_post(self, req, resp, user_id):
        try:
            doc = req.context.doc
        except AttributeError:
            raise falcon.HTTPBadRequest(
```

(continues on next page)

(continued from previous page)

```
                title='Missing thing',
                description='A thing must be submitted in the request body.',
            )

        proper_thing = self.db.add_thing(doc)

        resp.status = falcon.HTTP_201
        resp.location = '/{}/things/{}'.format(user_id, proper_thing['id'])


# Configure your WSGI server to load "things.app" (app is a WSGI callable)
app = falcon.App(
    middleware=[
        AuthMiddleware(),
        RequireJSON(),
        JSONTranslator(),
    ]
)

db = StorageEngine()
things = ThingsResource(db)
app.add_route('/{user_id}/things', things)

# If a responder ever raises an instance of StorageError, pass control to
# the given handler.
app.add_error_handler(StorageError, StorageError.handle)

# Proxy some things to another service; this example shows how you might
# send parts of an API off to a legacy system that hasn't been upgraded
# yet, or perhaps is a single cluster that all data centers have to share.
sink = SinkAdapter()
app.add_sink(sink, r'/search/(?P<engine>ddg|y)\Z')

# Useful for debugging problems in your API; works with pdb.set_trace(). You
# can also use Gunicorn to host your app. Gunicorn can be configured to
# auto-restart workers when it detects a code change, and it also works
# with pdb.
if __name__ == '__main__':
    httpd = simple_server.make_server('127.0.0.1', 8000, app)
    httpd.serve_forever()
```

Again this code uses wsgiref, but you can also run the above example using any WSGI server, such as uWSGI or Gunicorn. For example:

```
$ pip install requests gunicorn
$ gunicorn things:app
```

On Windows you can run Gunicorn and uWSGI via WSL, or you might try Waitress:

```
$ pip install requests waitress
$ waitress-serve --port=8000 things:app
```

To test this example go to the another terminal and run:

```
$ http localhost:8000/1/things authorization:custom-token
```

To visualize the application configuration the *Inspect Module* can be used:

```
falcon-inspect-app things_advanced:app
```

This would print for this example application:

```
Falcon App (WSGI)
• Routes:
    ⇒ /{user_id}/things - ThingsResource:
       ├── GET - on_get
       └── POST - on_post
• Middleware (Middleware are independent):
    → AuthMiddleware.process_request
      → RequireJSON.process_request
        → JSONTranslator.process_request

            ├── Process route responder

        ↞ JSONTranslator.process_response
• Sinks:
    ⇥ /search/(?P<engine>ddg|y)\Z SinkAdapter
• Error handlers:
    ↞ StorageError handle
```

ASGI

Note that this example requires the httpx package in lieu of requests.

```
# examples/things_advanced_asgi.py

import json
import logging
import uuid

import httpx

import falcon
import falcon.asgi


class StorageEngine:
    async def get_things(self, marker, limit):
        return [{'id': str(uuid.uuid4()), 'color': 'green'}]

    async def add_thing(self, thing):
        thing['id'] = str(uuid.uuid4())
        return thing


class StorageError(Exception):
    @staticmethod
    async def handle(req, resp, ex, params):
        # TODO: Log the error, clean up, etc. before raising
        raise falcon.HTTPInternalServerError()


class SinkAdapter:
    engines = {
        'ddg': 'https://duckduckgo.com',
```

(continues on next page)

(continued from previous page)

```
        'y': 'https://search.yahoo.com/search',
    }

    async def __call__(self, req, resp, engine):
        url = self.engines[engine]
        params = {'q': req.get_param('q', True)}

        async with httpx.AsyncClient() as client:
            result = await client.get(url, params=params)

        resp.status = result.status_code
        resp.content_type = result.headers['content-type']
        resp.text = result.text


class AuthMiddleware:
    async def process_request(self, req, resp):
        token = req.get_header('Authorization')
        account_id = req.get_header('Account-ID')

        challenges = ['Token type="Fernet"']

        if token is None:
            description = 'Please provide an auth token as part of the request.'

            raise falcon.HTTPUnauthorized(
                title='Auth token required',
                description=description,
                challenges=challenges,
                href='http://docs.example.com/auth',
            )

        if not self._token_is_valid(token, account_id):
            description = (
                'The provided auth token is not valid. '
                'Please request a new token and try again.'
            )

            raise falcon.HTTPUnauthorized(
                title='Authentication required',
                description=description,
                challenges=challenges,
                href='http://docs.example.com/auth',
            )

    def _token_is_valid(self, token, account_id):
        return True  # Suuuuuure it's valid...


class RequireJSON:
    async def process_request(self, req, resp):
        if not req.client_accepts_json:
            raise falcon.HTTPNotAcceptable(
                description='This API only supports responses encoded as JSON.',
                href='http://docs.examples.com/api/json',
            )
```

(continues on next page)

(continued from previous page)

```
        if req.method in ('POST', 'PUT'):
            if 'application/json' not in req.content_type:
                raise falcon.HTTPUnsupportedMediaType(
                    title='This API only supports requests encoded as JSON.',
                    href='http://docs.examples.com/api/json',
                )


class JSONTranslator:
    # NOTE: Normally you would simply use req.get_media() and resp.media for
    # this particular use case; this example serves only to illustrate
    # what is possible.

    async def process_request(self, req, resp):
        # NOTE: Test explicitly for 0, since this property could be None in
        # the case that the Content-Length header is missing (in which case we
        # can't know if there is a body without actually attempting to read
        # it from the request stream.)
        if req.content_length == 0:
            # Nothing to do
            return

        body = await req.stream.read()
        if not body:
            raise falcon.HTTPBadRequest(
                title='Empty request body',
                description='A valid JSON document is required.',
            )

        try:
            req.context.doc = json.loads(body.decode('utf-8'))

        except (ValueError, UnicodeDecodeError):
            description = (
                'Could not decode the request body. The '
                'JSON was incorrect or not encoded as '
                'UTF-8.'
            )

            raise falcon.HTTPBadRequest(title='Malformed JSON',␣
↪description=description)

    async def process_response(self, req, resp, resource, req_succeeded):
        if not hasattr(resp.context, 'result'):
            return

        resp.text = json.dumps(resp.context.result)


def max_body(limit):
    async def hook(req, resp, resource, params):
        length = req.content_length
        if length is not None and length > limit:
            msg = (
                'The size of the request is too large. The body must not '
```

(continues on next page)

(continued from previous page)

```
                'exceed ' + str(limit) + ' bytes in length.'
            )

            raise falcon.HTTPContentTooLarge(
                title='Request body is too large', description=msg
            )

    return hook


class ThingsResource:
    def __init__(self, db):
        self.db = db
        self.logger = logging.getLogger('thingsapp.' + __name__)

    async def on_get(self, req, resp, user_id):
        marker = req.get_param('marker') or ''
        limit = req.get_param_as_int('limit') or 50

        try:
            result = await self.db.get_things(marker, limit)
        except Exception as ex:
            self.logger.error(ex)

            description = (
                'Aliens have attacked our base! We will '
                'be back as soon as we fight them off. '
                'We appreciate your patience.'
            )

            raise falcon.HTTPServiceUnavailable(
                title='Service Outage', description=description, retry_after=30
            )

        # NOTE: Normally you would use resp.media for this sort of thing;
        # this example serves only to demonstrate how the context can be
        # used to pass arbitrary values between middleware components,
        # hooks, and resources.
        resp.context.result = result

        resp.set_header('Powered-By', 'Falcon')
        resp.status = falcon.HTTP_200

    @falcon.before(max_body(64 * 1024))
    async def on_post(self, req, resp, user_id):
        try:
            doc = req.context.doc
        except AttributeError:
            raise falcon.HTTPBadRequest(
                title='Missing thing',
                description='A thing must be submitted in the request body.',
            )

        proper_thing = await self.db.add_thing(doc)

        resp.status = falcon.HTTP_201
```

(continues on next page)

(continued from previous page)

```
        resp.location = '/{}/things/{}'.format(user_id, proper_thing['id'])


# The app instance is an ASGI callable
app = falcon.asgi.App(
    middleware=[
        AuthMiddleware(),
        RequireJSON(),
        JSONTranslator(),
    ]
)

db = StorageEngine()
things = ThingsResource(db)
app.add_route('/{user_id}/things', things)

# If a responder ever raises an instance of StorageError, pass control to
# the given handler.
app.add_error_handler(StorageError, StorageError.handle)

# Proxy some things to another service; this example shows how you might
# send parts of an API off to a legacy system that hasn't been upgraded
# yet, or perhaps is a single cluster that all data centers have to share.
sink = SinkAdapter()
app.add_sink(sink, r'/search/(?P<engine>ddg|y)\Z')
```

You can run the ASGI version with any ASGI server, such as uvicorn:

```
$ pip install falcon httpx uvicorn
$ uvicorn things_advanced_asgi:app
```

5.1.4 Tutorial (WSGI)

In this tutorial we'll walk through building an API for a simple image sharing service. Along the way, we'll discuss Falcon's major features and introduce the terminology used by the framework.

> **Note**
>
> This tutorial covers the "traditional", synchronous flavor of Falcon using the WSGI protocol.
>
> Developing an `async` application? Check out our *ASGI tutorial* instead!

First Steps

The first thing we'll do is *install* Falcon inside a fresh virtualenv. To that end, let's create a new project folder called "look", and set up a virtual environment within it that we can use for the tutorial:

```
$ mkdir look
$ cd look
$ virtualenv .venv
$ source .venv/bin/activate
$ pip install falcon
```

It's customary for the project's top-level module to be called the same as the project, so let's create another "look" folder inside the first one and mark it as a python module by creating an empty `__init__.py` file in it:

```
$ mkdir look
$ touch look/__init__.py
```

Next, let's create a new file that will be the entry point into your app:

```
$ touch look/app.py
```

The file hierarchy should now look like this:

```
look
├── .venv
└── look
    ├── __init__.py
    └── app.py
```

Now, open `app.py` in your favorite text editor and add the following lines:

```
import falcon

app = application = falcon.App()
```

This code creates your WSGI application and aliases it as `app`. You can use any variable names you like, but we'll use `application` since that is what Gunicorn, by default, expects it to be called (we'll see how this works in the next section of the tutorial).

Note

A WSGI application is just a callable with a well-defined signature so that you can host the application with any web server that understands the WSGI protocol.

Next let's take a look at the *falcon.App* class. Install IPython and fire it up:

```
$ pip install ipython
$ ipython
```

Now, type the following to introspect the *falcon.App* callable:

```
In [1]: import falcon

In [2]: falcon.App.__call__?
```

Alternatively, you can use the standard Python `help()` function:

```
In [3]: help(falcon.App.__call__)
```

Note the method signature. `env` and `start_response` are standard WSGI params. Falcon adds a thin abstraction on top of these params so you don't have to interact with them directly.

The Falcon framework contains extensive inline documentation that you can query using the above technique.

Tip

In addition to IPython, the Python community maintains several other super-powered REPLs that you may wish to try, including bpython and ptpython.

Hosting Your App

Now that you have a simple Falcon app, you can take it for a spin with a WSGI server. Python includes a reference server for self-hosting, but let's use something more robust that you might use in production.

Open a new terminal and run the following:

```
$ source .venv/bin/activate
$ pip install gunicorn
$ gunicorn --reload look.app
```

(Note the use of the `--reload` option to tell Gunicorn to reload the app whenever its code changes.)

If you are a Windows user, Waitress can be used in lieu of Gunicorn, since the latter doesn't work under Windows:

```
$ pip install waitress
$ waitress-serve --port=8000 look.app:app
```

Now, in a different terminal, try querying the running app with curl:

```
$ curl -v localhost:8000
```

You should get a 404. That's actually OK, because we haven't specified any routes yet. Falcon includes a default 404 response handler that will fire for any requested path for which a route does not exist.

While curl certainly gets the job done, it can be a bit crufty to use. HTTPie is a modern, user-friendly alternative. Let's install HTTPie and use it from now on:

```
$ source .venv/bin/activate
$ pip install httpie
$ http localhost:8000
```

Creating Resources

Falcon's design borrows several key concepts from the REST architectural style.

Central to both REST and the Falcon framework is the concept of a "resource". Resources are simply all the things in your API or application that can be accessed by a URL. For example, an event booking application may have resources such as "ticket" and "venue", while a video game backend may have resources such as "achievements" and "player".

URLs provide a way for the client to uniquely identify resources. For example, `/players` might identify the "list of all players" resource, while `/players/45301f54` might identify the "individual player with ID 45301f54", and `/players/45301f54/achievements` the "list of all achievements for the player resource with ID 45301f54".

```
  POST        /players/45301f54/achievements
└──────┘    └──────────────────────────────────┘

 Action            Resource Identifier
```

In the REST architectural style, the URL only identifies the resource; it does not specify what action to take on that resource. Instead, users choose from a set of standard methods. For HTTP, these are the familiar GET, POST, HEAD, etc. Clients can query a resource to discover which methods it supports.

> **Note**
>
> This is one of the key differences between the REST and RPC architectural styles. REST applies a standard set of verbs across any number of resources, as opposed to having each application define its own unique set of methods.

Depending on the requested action, the server may or may not return a representation to the client. Representations may be encoded in any one of a number of Internet media types, such as JSON and HTML.

Falcon uses Python classes to represent resources. In practice, these classes act as controllers in your application. They convert an incoming request into one or more internal actions, and then compose a response back to the client based on the results of those actions.

```
            ┌──────────────┐
request   → │              │
            │ Resource     │ ↻ Orchestrate the requested action
            │ Controller   │ ↻ Compose the result
response  ← │              │
            └──────────────┘
```

A resource in Falcon is just a regular Python class that includes one or more methods representing the standard HTTP verbs supported by that resource. Each requested URL is mapped to a specific resource.

Since we are building an image-sharing API, let's start by creating an "images" resource. Create a new module, `images.py` next to `app.py`, and add the following code to it:

```
import json

import falcon


class Resource:

    def on_get(self, req, resp):
        doc = {
            'images': [
                {
                    'href': '/images/1eaf6ef1-7f2d-4ecc-a8d5-6e8adba7cc0e.png'
                }
            ]
        }

        # Create a JSON representation of the resource
        resp.text = json.dumps(doc, ensure_ascii=False)

        # The following line can be omitted because 200 is the default
        # status returned by the framework, but it is included here to
        # illustrate how this may be overridden as needed.
        resp.status = falcon.HTTP_200
```

As you can see, `Resource` is just a regular class. You can name the class anything you like. Falcon uses duck-typing, so you don't need to inherit from any sort of special base class.

The image resource above defines a single method, `on_get()`. For any HTTP method you want your resource to support, simply add an `on_*()` method to the class, where `*` is any one of the standard HTTP methods, lowercased (e.g., `on_get()`, `on_put()`, `on_head()`, etc.).

Note

Supported HTTP methods are those specified in RFC 7231 and RFC 5789. This includes GET, HEAD, POST, PUT, DELETE, CONNECT, OPTIONS, TRACE, and PATCH.

We call these well-known methods "responders". Each responder takes (at least) two params, one representing the HTTP request, and one representing the HTTP response to that request. By convention, these are called `req` and `resp`, respectively. Route templates and hooks can inject extra params, as we shall see later on.

Right now, the image resource responds to GET requests with a simple `200 OK` and a JSON body. Falcon's Internet media type defaults to `application/json` but you can set it to whatever you like. Noteworthy JSON alternatives

include YAML and MessagePack.

Next let's wire up this resource and see it in action. Go back to `app.py` and modify it so that it looks something like this:

```
import falcon

from .images import Resource


app = application = falcon.App()

images = Resource()
app.add_route('/images', images)
```

Now, when a request comes in for `/images`, Falcon will call the responder on the images resource that corresponds to the requested HTTP method.

Let's try it. Restart Gunicorn (unless you're using `--reload`), and send a GET request to the resource:

```
$ http localhost:8000/images
```

You should receive a `200 OK` response, including a JSON-encoded representation of the "images" resource.

Note

`add_route()` expects an instance of the resource class, not the class itself. The same instance is used for all requests. This strategy improves performance and reduces memory usage, but this also means that if you host your application with a threaded web server, resources and their dependencies must be thread-safe.

We can use the *Inspect Module* to visualize the application configuration:

```
falcon-inspect-app look.app:app
```

This prints the following, correctly indicating that we are handling `GET` requests in the `/images` route:

```
Falcon App (WSGI)
• Routes:
    ⇒ /images - Resource:
       └── GET - on_get
```

So far we have only implemented a responder for GET. Let's see what happens when a different method is requested:

```
$ http PUT localhost:8000/images
```

This time you should get back `405 Method Not Allowed`, since the resource does not support the `PUT` method. Note the value of the Allow header:

```
allow: GET, OPTIONS
```

This is generated automatically by Falcon based on the set of methods implemented by the target resource. If a resource does not include its own OPTIONS responder, the framework provides a default implementation. Therefore, OPTIONS is always included in the list of allowable methods.

Note

If you have a lot of experience with other Python web frameworks, you may be used to using decorators to set up your routes. Falcon's particular approach provides the following benefits:

- The URL structure of the application is centralized. This makes it easier to reason about and maintain the API over time.
- The use of resource classes maps somewhat naturally to the REST architectural style, in which a URL is used to identify a resource only, not the action to perform on that resource.
- Resource class methods provide a uniform interface that does not have to be reinvented (and maintained) from class to class and application to application.

Next, just for fun, let's modify our resource to use MessagePack instead of JSON. Start by installing the relevant package:

```
$ pip install msgpack-python
```

Then, update the responder to use the new media type:

```
import msgpack

import falcon

class Resource:

    def on_get(self, req, resp):
        doc = {
            'images': [
                {
                    'href': '/images/1eaf6ef1-7f2d-4ecc-a8d5-6e8adba7cc0e.png'
                }
            ]
        }

        resp.data = msgpack.packb(doc, use_bin_type=True)
        resp.content_type = falcon.MEDIA_MSGPACK
        resp.status = falcon.HTTP_200
```

Note the use of `resp.data` in lieu of `resp.text`. If you assign a bytestring to the latter, Falcon will figure it out, but you can realize a small performance gain by assigning directly to `resp.data`.

Also note the use of `falcon.MEDIA_MSGPACK`. The `falcon` module provides a number of constants for common media types, including `falcon.MEDIA_JSON`, `falcon.MEDIA_MSGPACK`, `falcon.MEDIA_YAML`, `falcon.MEDIA_XML`, `falcon.MEDIA_HTML`, `falcon.MEDIA_JS`, `falcon.MEDIA_TEXT`, `falcon.MEDIA_JPEG`, `falcon.MEDIA_PNG`, and `falcon.MEDIA_GIF`.

Restart Gunicorn (unless you're using `--reload`), and then try sending a GET request to the revised resource:

```
$ http localhost:8000/images
```

Testing your application

Fully exercising your code is critical to creating a robust application. Let's take a moment to write a test for what's been implemented so far.

First, create a `tests` directory with `__init__.py` and a test module (`test_app.py`) inside it. The project's structure should now look like this:

```
look
├── .venv
├── look
|   ├── __init__.py
```

(continues on next page)

(continued from previous page)

```
|   ├── app.py
|   └── images.py
└── tests
    ├── __init__.py
    └── test_app.py
```

Falcon supports *testing* its `App` object by simulating HTTP requests.

Tests can either be written using Python's standard `unittest` module, or with any of a number of third-party testing frameworks, such as pytest. For this tutorial we'll use pytest since it allows for more pythonic test code as compared to the JUnit-inspired `unittest` module.

Let's start by installing the pytest package:

```
$ pip install pytest
```

Next, edit `test_app.py` to look like this:

```
import msgpack
import pytest

import falcon
from falcon import testing

from look.app import app


@pytest.fixture
def client():
    return testing.TestClient(app)


# pytest will inject the object returned by the "client" function
# as an additional parameter.
def test_list_images(client):
    doc = {
        'images': [
            {
                'href': '/images/1eaf6ef1-7f2d-4ecc-a8d5-6e8adba7cc0e.png'
            }
        ]
    }

    response = client.simulate_get('/images')
    result_doc = msgpack.unpackb(response.content, raw=False)

    assert result_doc == doc
    assert response.status == falcon.HTTP_OK
```

From the main project directory, exercise your new test by running pytest against the `tests` directory:

```
$ pytest tests
```

If pytest reports any errors, take a moment to fix them up before proceeding to the next section of the tutorial.

Request and Response Objects

Each responder in a resource receives a `Request` object that can be used to read the headers, query parameters, and body of the request. You can use the standard `help()` function or IPython's magic `?` function to list the attributes and methods of Falcon's `Request` class:

```
In [1]: import falcon

In [2]: falcon.Request?
```

Each responder also receives a `Response` object that can be used for setting the status code, headers, and body of the response:

```
In [3]: falcon.Response?
```

This will be useful when creating a POST endpoint in the application that can add new image resources to our collection. We'll tackle this functionality next.

We'll use TDD this time around, to demonstrate how to apply this particular testing strategy when developing a Falcon application. Via tests, we'll first define precisely what we want the application to do, and then code until the tests tell us that we're done.

> **Note**
>
> To learn more about TDD, you may wish to check out one of the many books on the topic, such as Test Driven Development with Python. The examples in this particular book use the Django framework and even JavaScript, but the author covers a number of testing principles that are widely applicable.

Let's start by adding an additional import statement to `test_app.py`. We need to import two modules from `unittest.mock`:

```
from unittest.mock import mock_open, call
```

Now add the following test:

```
# "monkeypatch" is a special built-in pytest fixture that can be
# used to install mocks.
def test_posted_image_gets_saved(client, monkeypatch):
    mock_file_open = mock_open()
    monkeypatch.setattr('io.open', mock_file_open)

    fake_uuid = '123e4567-e89b-12d3-a456-426655440000'
    monkeypatch.setattr('uuid.uuid4', lambda: fake_uuid)

    # When the service receives an image through POST...
    fake_image_bytes = b'fake-image-bytes'
    response = client.simulate_post(
        '/images',
        body=fake_image_bytes,
        headers={'content-type': 'image/png'}
    )

    # ...it must return a 201 code, save the file, and return the
    # image's resource location.
    assert response.status == falcon.HTTP_CREATED
    assert call().write(fake_image_bytes) in mock_file_open.mock_calls
    assert response.headers['location'] == '/images/{}.png'.format(fake_uuid)
```

As you can see, this test relies heavily on mocking, making it somewhat fragile in the face of implementation changes. We'll revisit this later. For now, run the tests again and watch to make sure they fail. A key step in the TDD workflow is verifying that your tests **do not** pass before moving on to the implementation:

```
$ pytest tests
```

To make the new test pass, we need to add a new method for handling POSTs. Open `images.py` and add a POST responder to the `Resource` class as follows:

```
import io
import os
import uuid
import mimetypes

import msgpack

import falcon


class Resource:

    _CHUNK_SIZE_BYTES = 4096

    # The resource object must now be initialized with a path used during POST
    def __init__(self, storage_path):
        self._storage_path = storage_path

    # This is the method we implemented before
    def on_get(self, req, resp):
        doc = {
            'images': [
                {
                    'href': '/images/1eaf6ef1-7f2d-4ecc-a8d5-6e8adba7cc0e.png'
                }
            ]
        }

        resp.data = msgpack.packb(doc, use_bin_type=True)
        resp.content_type = falcon.MEDIA_MSGPACK
        resp.status = falcon.HTTP_200

    def on_post(self, req, resp):
        ext = mimetypes.guess_extension(req.content_type)
        name = '{uuid}{ext}'.format(uuid=uuid.uuid4(), ext=ext)
        image_path = os.path.join(self._storage_path, name)

        with io.open(image_path, 'wb') as image_file:
            while True:
                chunk = req.stream.read(self._CHUNK_SIZE_BYTES)
                if not chunk:
                    break

                image_file.write(chunk)

        resp.status = falcon.HTTP_201
        resp.location = '/images/' + name
```

As you can see, we generate a unique name for the image, and then write it out by reading from `req.stream`. It's called `stream` instead of `body` to emphasize the fact that you are really reading from an input stream; by default

Falcon does not spool or decode request data, instead giving you direct access to the incoming binary stream provided by the WSGI server.

Note the use of `falcon.HTTP_201` for setting the response status to "201 Created". We could have also used the `falcon.HTTP_CREATED` alias. For a full list of predefined status strings, simply call `help()` on `falcon.status_codes`:

```
In [4]: help(falcon.status_codes)
```

The last line in the `on_post()` responder sets the Location header for the newly created resource. (We will create a route for that path in just a minute.) The *Request* and *Response* classes contain convenient attributes for reading and setting common headers, but you can always access any header by name with the `req.get_header()` and `resp.set_header()` methods.

Take a moment to run pytest again to check your progress:

```
$ pytest tests
```

You should see a `TypeError` as a consequence of adding the `storage_path` parameter to `Resource.__init__()`.

To fix this, simply edit `app.py` and pass in a path to the initializer. For now, just use the working directory from which you started the service:

```
images = Resource(storage_path='.')
```

Try running the tests again. This time, they should pass with flying colors!

```
$ pytest tests
```

Finally, restart Gunicorn and then try sending a POST request to the resource from the command line (substituting `test.png` for a path to any PNG you like.)

```
$ http POST localhost:8000/images Content-Type:image/png < test.png
```

Now, if you check your storage directory, it should contain a copy of the image you just POSTed.

Upward and onward!

Refactoring for testability

Earlier we pointed out that our POST test relied heavily on mocking, relying on assumptions that may or may not hold true as the code evolves. To mitigate this problem, we'll not only have to refactor the tests, but also the application itself.

We'll start by factoring out the business logic from the resource's POST responder in `images.py` so that it can be tested independently. In this case, the resource's "business logic" is simply the image-saving operation:

```
import io
import mimetypes
import os
import uuid

import msgpack

import falcon


class Resource:

    def __init__(self, image_store):
```

(continues on next page)

(continued from previous page)

```
        self._image_store = image_store

    def on_get(self, req, resp):
        doc = {
            'images': [
                {
                    'href': '/images/1eaf6ef1-7f2d-4ecc-a8d5-6e8adba7cc0e.png'
                }
            ]
        }

        resp.data = msgpack.packb(doc, use_bin_type=True)
        resp.content_type = falcon.MEDIA_MSGPACK
        resp.status = falcon.HTTP_200

    def on_post(self, req, resp):
        name = self._image_store.save(req.stream, req.content_type)
        resp.status = falcon.HTTP_201
        resp.location = '/images/' + name


class ImageStore:

    _CHUNK_SIZE_BYTES = 4096

    # Note the use of dependency injection for standard library
    # methods. We'll use these later to avoid monkey-patching.
    def __init__(self, storage_path, uuidgen=uuid.uuid4, fopen=io.open):
        self._storage_path = storage_path
        self._uuidgen = uuidgen
        self._fopen = fopen

    def save(self, image_stream, image_content_type):
        ext = mimetypes.guess_extension(image_content_type)
        name = '{uuid}{ext}'.format(uuid=self._uuidgen(), ext=ext)
        image_path = os.path.join(self._storage_path, name)

        with self._fopen(image_path, 'wb') as image_file:
            while True:
                chunk = image_stream.read(self._CHUNK_SIZE_BYTES)
                if not chunk:
                    break

                image_file.write(chunk)

        return name
```

Let's check to see if we broke anything with the changes above:

```
$ pytest tests
```

Hmm, it looks like we forgot to update `app.py`. Let's do that now:

```
import falcon

from .images import ImageStore
from .images import Resource
```

(continues on next page)

(continued from previous page)

```
app = application = falcon.App()

image_store = ImageStore('.')
images = Resource(image_store)
app.add_route('/images', images)
```

Let's try again:

```
$ pytest tests
```

Now you should see a failed test assertion regarding `mock_file_open`. To fix this, we need to switch our strategy from monkey-patching to dependency injection. Return to `app.py` and modify it to look similar to the following:

```
import falcon

from .images import ImageStore
from .images import Resource


def create_app(image_store):
    image_resource = Resource(image_store)
    app = falcon.App()
    app.add_route('/images', image_resource)
    return app


def get_app():
    image_store = ImageStore('.')
    return create_app(image_store)
```

As you can see, the bulk of the setup logic has been moved to `create_app()`, which can be used to obtain an App object either for testing or for hosting in production. `get_app()` takes care of instantiating additional resources and configuring the application for hosting.

The command to run the application is now:

```
$ gunicorn --reload 'look.app:get_app()'
```

Finally, we need to update the test code. Modify `test_app.py` to look similar to this:

```
import io
from wsgiref.validate import InputWrapper

from unittest.mock import call, MagicMock, mock_open

import msgpack
import pytest

import falcon
from falcon import testing

import look.app
import look.images
```

(continues on next page)

(continued from previous page)

```
@pytest.fixture
def mock_store():
    return MagicMock()


@pytest.fixture
def client(mock_store):
    app = look.app.create_app(mock_store)
    return testing.TestClient(app)


def test_list_images(client):
    doc = {
        'images': [
            {
                'href': '/images/1eaf6ef1-7f2d-4ecc-a8d5-6e8adba7cc0e.png'
            }
        ]
    }

    response = client.simulate_get('/images')
    result_doc = msgpack.unpackb(response.content, raw=False)

    assert result_doc == doc
    assert response.status == falcon.HTTP_OK


# With clever composition of fixtures, we can observe what happens with
# the mock injected into the image resource.
def test_post_image(client, mock_store):
    file_name = 'fake-image-name.xyz'

    # We need to know what ImageStore method will be used
    mock_store.save.return_value = file_name
    image_content_type = 'image/xyz'

    response = client.simulate_post(
        '/images',
        body=b'some-fake-bytes',
        headers={'content-type': image_content_type}
    )

    assert response.status == falcon.HTTP_CREATED
    assert response.headers['location'] == '/images/{}'.format(file_name)
    saver_call = mock_store.save.call_args

    # saver_call is a unittest.mock.call tuple. It's first element is a
    # tuple of positional arguments supplied when calling the mock.
    assert isinstance(saver_call[0][0], InputWrapper)
    assert saver_call[0][1] == image_content_type
```

As you can see, we've redone the POST. While there are fewer mocks, the assertions have gotten more elaborate to properly check interactions at the interface boundaries.

Let's check our progress:

```
$ pytest tests
```

All green! But since we used a mock, we're no longer covering the actual saving of the image. Let's add a test for that:

```
def test_saving_image(monkeypatch):
    # This still has some mocks, but they are more localized and do not
    # have to be monkey-patched into standard library modules (always a
    # risky business).
    mock_file_open = mock_open()

    fake_uuid = '123e4567-e89b-12d3-a456-426655440000'
    def mock_uuidgen():
        return fake_uuid

    fake_image_bytes = b'fake-image-bytes'
    fake_request_stream = io.BytesIO(fake_image_bytes)
    storage_path = 'fake-storage-path'
    store = look.images.ImageStore(
        storage_path,
        uuidgen=mock_uuidgen,
        fopen=mock_file_open
    )

    assert store.save(fake_request_stream, 'image/png') == fake_uuid + '.png'
    assert call().write(fake_image_bytes) in mock_file_open.mock_calls
```

Now give it a try:

```
$ pytest tests -k test_saving_image
```

Like the former test, this one still uses mocks. But the structure of the code has been improved through the techniques of componentization and dependency inversion, making the application more flexible and testable.

Tip

Checking code coverage would have helped us detect the missing test above; it's always a good idea to include coverage testing in your workflow to ensure you don't have any bugs hiding off somewhere in an unexercised code path.

Functional tests

Functional tests define the application's behavior from the outside. When using TDD, this can be a more natural place to start as opposed to lower-level unit testing, since it is difficult to anticipate what internal interfaces and components are needed in advance of defining the application's user-facing functionality.

In the case of the refactoring work from the last section, we could have inadvertently introduced a functional bug into the application that our unit tests would not have caught. This can happen when a bug is a result of an unexpected interaction between multiple units, between the application and the web server, or between the application and any external services it depends on.

With test helpers such as `simulate_get()` and `simulate_post()`, we can create tests that span multiple units. But we can also go one step further and run the application as a normal, separate process (e.g. with Gunicorn). We can then write tests that interact with the running process through HTTP, behaving like a normal client.

Let's see this in action. Create a new test module, `tests/test_integration.py` with the following contents:

```
import os

import requests


def test_posted_image_gets_saved():
    file_save_prefix = '/tmp/'
    location_prefix = '/images/'
    fake_image_bytes = b'fake-image-bytes'

    response = requests.post(
        'http://localhost:8000/images',
        data=fake_image_bytes,
        headers={'content-type': 'image/png'}
    )

    assert response.status_code == 201
    location = response.headers['location']
    assert location.startswith(location_prefix)
    image_name = location.replace(location_prefix, '')

    file_path = file_save_prefix + image_name
    with open(file_path, 'rb') as image_file:
        assert image_file.read() == fake_image_bytes

    os.remove(file_path)
```

Next, install the `requests` package (as required by the new test) and make sure Gunicorn is up and running:

```
$ pip install requests
$ gunicorn 'look.app:get_app()'
```

Then, in another terminal, try running the new test:

```
$ pytest tests -k test_posted_image_gets_saved
```

The test will fail since it expects the image file to reside under `/tmp`. To fix this, modify `app.py` to add the ability to configure the image storage directory with an environment variable:

```
import os

import falcon

from .images import ImageStore
from .images import Resource


def create_app(image_store):
    image_resource = Resource(image_store)
    app = falcon.App()
    app.add_route('/images', image_resource)
    return app


def get_app():
    storage_path = os.environ.get('LOOK_STORAGE_PATH', '.')
    image_store = ImageStore(storage_path)
    return create_app(image_store)
```

Now you can re-run the app against the desired storage directory:

```
$ LOOK_STORAGE_PATH=/tmp gunicorn --reload 'look.app:get_app()'
```

You should now be able to re-run the test and see it succeed:

```
$ pytest tests -k test_posted_image_gets_saved
```

Note

The above process of starting, testing, stopping, and cleaning up after each test run can (and really should) be automated. Depending on your needs, you can develop your own automation fixtures, or use a library such as mountepy.

Many developers choose to write tests like the above to sanity-check their application's primary functionality, while leaving the bulk of testing to simulated requests and unit tests. These latter types of tests generally execute much faster and facilitate more fine-grained test assertions as compared to higher-level functional and system tests. That being said, testing strategies vary widely and you should choose the one that best suits your needs.

At this point, you should have a good grip on how to apply common testing strategies to your Falcon application. For the sake of brevity we'll omit further testing instructions from the following sections, focusing instead on showcasing more of Falcon's features.

Serving Images

Now that we have a way of getting images into the service, we of course need a way to get them back out. What we want to do is return an image when it is requested, using the path that came back in the Location header.

Try executing the following:

```
$ http localhost:8000/images/db79e518-c8d3-4a87-93fe-38b620f9d410.png
```

In response, you should get a `404 Not Found`. This is the default response given by Falcon when it can not find a resource that matches the requested URL path.

Let's address this by creating a separate class to represent a single image resource. We will then add an `on_get()` method to respond to the path above.

Go ahead and edit your `images.py` file to look something like this:

```
import io
import os
import re
import uuid
import mimetypes

import msgpack

import falcon


class Collection:

    def __init__(self, image_store):
        self._image_store = image_store

    def on_get(self, req, resp):
        # TODO: Modify this to return a list of href's based on
        # what images are actually available.
```

(continues on next page)

(continued from previous page)

```
        doc = {
            'images': [
                {
                    'href': '/images/1eaf6ef1-7f2d-4ecc-a8d5-6e8adba7cc0e.png'
                }
            ]
        }

        resp.data = msgpack.packb(doc, use_bin_type=True)
        resp.content_type = falcon.MEDIA_MSGPACK
        resp.status = falcon.HTTP_200

    def on_post(self, req, resp):
        name = self._image_store.save(req.stream, req.content_type)
        resp.status = falcon.HTTP_201
        resp.location = '/images/' + name


class Item:

    def __init__(self, image_store):
        self._image_store = image_store

    def on_get(self, req, resp, name):
        resp.content_type = mimetypes.guess_type(name)[0]
        resp.stream, resp.content_length = self._image_store.open(name)


class ImageStore:

    _CHUNK_SIZE_BYTES = 4096
    _IMAGE_NAME_PATTERN = re.compile(
        r'[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}\.[a-z]{2,4}$
↪'
    )

    def __init__(self, storage_path, uuidgen=uuid.uuid4, fopen=io.open):
        self._storage_path = storage_path
        self._uuidgen = uuidgen
        self._fopen = fopen

    def save(self, image_stream, image_content_type):
        ext = mimetypes.guess_extension(image_content_type)
        name = '{uuid}{ext}'.format(uuid=self._uuidgen(), ext=ext)
        image_path = os.path.join(self._storage_path, name)

        with self._fopen(image_path, 'wb') as image_file:
            while True:
                chunk = image_stream.read(self._CHUNK_SIZE_BYTES)
                if not chunk:
                    break

                image_file.write(chunk)

        return name
```

(continues on next page)

(continued from previous page)

```
    def open(self, name):
        # Always validate untrusted input!
        if not self._IMAGE_NAME_PATTERN.match(name):
            raise OSError('File not found')

        image_path = os.path.join(self._storage_path, name)
        stream = self._fopen(image_path, 'rb')
        content_length = os.path.getsize(image_path)

        return stream, content_length
```

As you can see, we renamed `Resource` to `Collection` and added a new `Item` class to represent a single image resource. Alternatively, these two classes could be consolidated into one by using suffixed responders. (See also: *add_route()*)

Also, note the `name` parameter for the `on_get()` responder. Any URI parameters that you specify in your routes will be turned into corresponding kwargs and passed into the target responder as such. We'll see how to specify URI parameters in a moment.

Inside the `on_get()` responder, we set the Content-Type header based on the filename extension, and then stream out the image directly from an open file handle. Note the use of `resp.content_length`. Whenever using `resp.stream` instead of `resp.text` or `resp.data`, you typically also specify the expected length of the stream using the Content-Length header, so that the web client knows how much data to read from the response.

Note

If you do not know the size of the stream in advance, you can work around that by using chunked encoding, but that's beyond the scope of this tutorial.

If `resp.status` is not set explicitly, it defaults to `200 OK`, which is exactly what we want `on_get()` to do.

Now let's wire everything up and give it a try. Edit `app.py` to look similar to the following:

```
import os

import falcon

from .images import Collection
from .images import ImageStore
from .images import Item


def create_app(image_store):
    app = falcon.App()
    app.add_route('/images', Collection(image_store))
    app.add_route('/images/{name}', Item(image_store))
    return app


def get_app():
    storage_path = os.environ.get('LOOK_STORAGE_PATH', '.')
    image_store = ImageStore(storage_path)
    return create_app(image_store)
```

As you can see, we specified a new route, `/images/{name}`. This causes Falcon to expect all associated responders to accept a `name` argument.

> **Note**
>
> Falcon also supports more complex parameterized path segments that contain multiple values. For example, a version control API might use the following route template for diffing two code branches:
>
> ```
> /repos/{org}/{repo}/compare/{usr0}:{branch0}...{usr1}:{branch1}
> ```

Now re-run your app and try to POST another picture:

```
$ http POST localhost:8000/images Content-Type:image/png < test.png
```

Make a note of the path returned in the Location header, and use it to GET the image:

```
$ http localhost:8000/images/dddff30e-d2a6-4b57-be6a-b985ee67fa87.png
```

HTTPie won't display the image, but you can see that the response headers were set correctly. Just for fun, go ahead and paste the above URI into your browser. The image should display correctly.

Inspecting the application now returns:

```
falcon-inspect-app look.app:get_app
```

```
Falcon App (WSGI)
• Routes:
    ⇒ /images - Collection:
       ├── GET - on_get
       └── POST - on_post
    ⇒ /images/{name} - Item:
       └── GET - on_get
```

Query Strings

Now that we are able to get the images from the service, we need a way to get a list of available images. We have already set up this route. Before testing this route let's change its output format back to JSON to have a more terminal-friendly output. The top of file `images.py` should look like this:

```
import io
import os
import re
import uuid
import mimetypes

import falcon
import json


class Collection:

    def __init__(self, image_store):
        self._image_store = image_store

    def on_get(self, req, resp):
        # TODO: Modify this to return a list of href's based on
        # what images are actually available.
        doc = {
            'images': [
                {
```

(continues on next page)

(continued from previous page)

```
                    'href': '/images/1eaf6ef1-7f2d-4ecc-a8d5-6e8adba7cc0e.png'
                }
            ]
        }

        resp.text = json.dumps(doc, ensure_ascii=False)
        resp.status = falcon.HTTP_200

def on_post(self, req, resp):
    name = self._image_store.save(req.stream, req.content_type)
    resp.status = falcon.HTTP_201
    resp.location = '/images/' + name
```

Now try the following:

```
http localhost:8000/images
```

In response you should get the following data that we statically have put in the code.

```
{
    "images": [
        {
            "href": "/images/1eaf6ef1-7f2d-4ecc-a8d5-6e8adba7cc0e.png"
        }
    ]
}
```

Let's go back to the `on_get` method and create a dynamic response. We can use query strings to set maximum image size and get the list of all images smaller than the specified value. We will use method `get_param_as_int` to set a default value of `-1` in case no `maxsize` query string was provided and also to enable a minimum value validation.

```
import io
import os
import re
import uuid
import mimetypes

import falcon
import json


class Collection:

    def __init__(self, image_store):
        self._image_store = image_store

    def on_get(self, req, resp):
        max_size = req.get_param_as_int("maxsize", min_value=1, default=-1)
        images = self._image_store.list(max_size)
        doc = {
            'images': [
                {'href': '/images/' + image} for image in images
            ]
        }

        resp.text = json.dumps(doc, ensure_ascii=False)
        resp.status = falcon.HTTP_200
```

(continues on next page)

(continued from previous page)

```
    def on_post(self, req, resp):
        name = self._image_store.save(req.stream, req.content_type)
        resp.status = falcon.HTTP_201
        resp.location = '/images/' + name


class Item:

    def __init__(self, image_store):
        self._image_store = image_store

    def on_get(self, req, resp, name):
        resp.content_type = mimetypes.guess_type(name)[0]
        resp.stream, resp.content_length = self._image_store.open(name)


class ImageStore:

    _CHUNK_SIZE_BYTES = 4096
    _IMAGE_NAME_PATTERN = re.compile(
        r'[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}\.[a-z]{2,4}$
↪'
    )

    def __init__(self, storage_path, uuidgen=uuid.uuid4, fopen=io.open):
        self._storage_path = storage_path
        self._uuidgen = uuidgen
        self._fopen = fopen

    def save(self, image_stream, image_content_type):
        ext = mimetypes.guess_extension(image_content_type)
        name = '{uuid}{ext}'.format(uuid=self._uuidgen(), ext=ext)
        image_path = os.path.join(self._storage_path, name)

        with self._fopen(image_path, 'wb') as image_file:
            while True:
                chunk = image_stream.read(self._CHUNK_SIZE_BYTES)
                if not chunk:
                    break

                image_file.write(chunk)

        return name

    def open(self, name):
        # Always validate untrusted input!
        if not self._IMAGE_NAME_PATTERN.match(name):
            raise OSError('File not found')

        image_path = os.path.join(self._storage_path, name)
        stream = self._fopen(image_path, 'rb')
        content_length = os.path.getsize(image_path)

        return stream, content_length
```

(continues on next page)

(continued from previous page)

```
    def list(self, max_size):
        images = [
            image for image in os.listdir(self._storage_path)
            if self._IMAGE_NAME_PATTERN.match(image)
            and (
                max_size == -1
                or os.path.getsize(os.path.join(self._storage_path, image)) <= max_
↪size
            )
        ]
        return images
```

As you can see the method `list` has been added to `ImageStore` in order to return list of available images smaller than `max_size` unless it is not `-1`, in which case it will behave like there was no predicament of image size. Let's try to save some binary data as images in the service and then try to retrieve their list. Execute the following commands in order to simulate the creation of 3 files as images with different sizes. While these are not valid PNG files, they will work for this tutorial.

```
echo "First Case" > pseudo-image-1.png
echo "Second Case" > pseudo-image-2.png
echo "3rd Case" > pseudo-image-3.png
```

Now we need to store these files using `POST` request:

```
http POST localhost:8000/images Content-Type:image/png < pseudo-image-1.png
http POST localhost:8000/images Content-Type:image/png < pseudo-image-2.png
http POST localhost:8000/images Content-Type:image/png < pseudo-image-3.png
```

If we check the size of these files, we will see that they are 11, 12, 9 bytes respectively. Let's try to get the list of the images which are smaller or equal to 11 bytes.

```
http localhost:8000/images?maxsize=11
```

We expect to get a list of 2 files, which will be similar to the following:

```
{
    "images": [
        {
            "href": "/images/7ba2ebc9-726f-46b0-9615-a69824f5089b.png"
        },
        {
            "href": "/images/e4354a31-2161-4064-805c-3bc7c332e7e6.png"
        }
    ]
}
```

You could also now validate the response with getting the image files using the `href` value in the response and compare them with the original files.

Introducing Hooks

At this point you should have a pretty good understanding of the basic parts that make up a Falcon-based API. Before we finish up, let's just take a few minutes to clean up the code and add some error handling.

First, let's check the incoming media type when something is posted to make sure it is a common image type. We'll implement this with a `before` hook.

Start by defining a list of media types the service will accept. Place this constant near the top, just after the import statements in `images.py`:

```
ALLOWED_IMAGE_TYPES = (
    'image/gif',
    'image/jpeg',
    'image/png',
)
```

The idea here is to only accept GIF, JPEG, and PNG images. You can add others to the list if you like.

Next, let's create a hook that will run before each request to post a message. Add this method below the definition of `ALLOWED_IMAGE_TYPES`:

```
def validate_image_type(req, resp, resource, params):
    if req.content_type not in ALLOWED_IMAGE_TYPES:
        msg = 'Image type not allowed. Must be PNG, JPEG, or GIF'
        raise falcon.HTTPBadRequest(title='Bad request', description=msg)
```

And then attach the hook to the `on_post()` responder:

```
@falcon.before(validate_image_type)
def on_post(self, req, resp):
    pass
```

Now, before every call to that responder, Falcon will first invoke `validate_image_type()`. There isn't anything special about this function, other than it must accept four arguments. Every hook takes, as its first two arguments, a reference to the same `req` and `resp` objects that are passed into responders. The `resource` argument is a Resource instance associated with the request. The fourth argument, named `params` by convention, is a reference to the kwarg dictionary Falcon creates for each request. `params` will contain the route's URI template params and their values, if any.

As you can see in the example above, you can use `req` to get information about the incoming request. However, you can also use `resp` to play with the HTTP response as needed, and you can even use hooks to inject extra kwargs:

```
def extract_project_id(req, resp, resource, params):
    """Adds `project_id` to the list of params for all responders.

    Meant to be used as a `before` hook.
    """
    params['project_id'] = req.get_header('X-PROJECT-ID')
```

Now, you might imagine that such a hook should apply to all responders for a resource. In fact, hooks can be applied to an entire resource by simply decorating the class:

```
@falcon.before(extract_project_id)
class Message:
    pass
```

Similar logic can be applied globally with middleware. (See also: *falcon.middleware*)

Now that you've added a hook to validate the media type, you can see it in action by attempting to POST something nefarious:

```
$ http POST localhost:8000/images Content-Type:image/jpx
```

You should get back a `400 Bad Request` status and a nicely structured error body.

Tip

When something goes wrong, you usually want to give your users some info to help them resolve the issue. The exception to this rule is when an error occurs because the user is requested something they are not authorized to

access. In that case, you may wish to simply return `404 Not Found` with an empty body, in case a malicious user is fishing for information that will help them crack your app.

Check out the *hooks reference* to learn more.

Error Handling

Generally speaking, Falcon assumes that resource responders (`on_get()`, `on_post()`, etc.) will, for the most part, do the right thing. In other words, Falcon doesn't try very hard to protect responder code from itself.

This approach reduces the number of (often) extraneous checks that Falcon would otherwise have to perform, making the framework more efficient. With that in mind, writing a high-quality API based on Falcon requires that:

1. Resource responders set response variables to sane values.
2. Untrusted input (i.e., input from an external client or service) is validated.
3. Your code is well-tested, with high code coverage.
4. Errors are anticipated, detected, logged, and handled appropriately within each responder or by global error handling hooks.

When it comes to error handling, you can always directly set the error status, appropriate response headers, and error body using the `resp` object. However, Falcon tries to make things a little easier by providing a *set of error classes* you can raise when something goes wrong. Falcon will convert any instance or subclass of *`falcon.HTTPError`* raised by a responder, hook, or middleware component into an appropriate HTTP response.

You may raise an instance of *`falcon.HTTPError`* directly, or use any one of a number of *predefined errors* that are designed to set the response headers and body appropriately for each error type.

Tip

Error handlers may be registered for any type, including *`HTTPError`*. This feature provides a central location for logging and otherwise handling exceptions raised by responders, hooks, and middleware components.

See also: *`add_error_handler()`*.

Let's see a quick example of how this works. Try requesting an invalid image name from your application:

```
$ http localhost:8000/images/voltron.png
```

As you can see, the result isn't exactly graceful. To fix this, we'll need to add some exception handling. Modify your `Item` class as follows:

```
class Item:

    def __init__(self, image_store):
        self._image_store = image_store

    def on_get(self, req, resp, name):
        resp.content_type = mimetypes.guess_type(name)[0]

        try:
            resp.stream, resp.content_length = self._image_store.open(name)
        except OSError:
            # Normally you would also log the error.
            raise falcon.HTTPNotFound()
```

Now let's try that request again:

```
$ http localhost:8000/images/voltron.png
```

Additional information about error handling is available in the *error handling reference*.

What Now?

Our friendly community is available to answer your questions and help you work through sticky problems. See also: *Getting Help*.

As mentioned previously, Falcon's docstrings are quite extensive, and so you can learn a lot just by poking around Falcon's modules from a Python REPL, such as IPython or bpython.

Also, don't be shy about pulling up Falcon's source code on GitHub or in your favorite text editor. The team has tried to make the code as straightforward and readable as possible; where other documentation may fall short, the code basically can't be wrong.

A number of Falcon add-ons, templates, and complementary packages are available for use in your projects. We've listed several of these on the Falcon wiki as a starting point, but you may also wish to search PyPI for additional resources.

5.1.5 Tutorial (ASGI)

In this tutorial we'll walk through building an API for a simple image sharing service. Along the way, we'll discuss the basic anatomy of an asynchronous Falcon application: responders, routing, middleware, executing synchronous functions in an executor, and more!

> **Note**
>
> This tutorial covers the asynchronous flavor of Falcon using the ASGI protocol.
>
> Synchronous (WSGI) Falcon application development is covered in our *WSGI tutorial*.
>
> New Falcon users may also want to choose the WSGI flavor to familiarize themselves with Falcon's basic concepts.

First Steps

Let's start by creating a fresh environment and the corresponding project directory structure, along the lines of *First Steps* from the WSGI tutorial:

```
asgilook
├── .venv
└── asgilook
    ├── __init__.py
    └── app.py
```

We'll create a *virtualenv* using the `venv` module from the standard library (Falcon requires Python 3.9+):

```
$ mkdir asgilook
$ python3 -m venv asgilook/.venv
$ source asgilook/.venv/bin/activate
```

> **Note**
>
> If your Python distribution does not happen to include the `venv` module, you can always install and use virtualenv instead.

> **Tip**
>
> Some of us find it convenient to manage *virtualenv*s with virtualenvwrapper or pipenv, particularly when it comes to hopping between several environments.

Next, *install Falcon* into your *virtualenv*:

```
$ pip install falcon
```

You can then create a basic *Falcon ASGI application* by adding an `asgilook/app.py` module with the following contents:

```
import falcon.asgi

app = falcon.asgi.App()
```

As in the *WSGI tutorial's introduction*, let's not forget to mark `asgilook` as a Python package:

```
$ touch asgilook/__init__.py
```

Hosting Our App

For running our async application, we'll need an ASGI application server. Popular choices include:

- Uvicorn
- Daphne
- Hypercorn

For a simple tutorial application like ours, any of the above should do. Let's pick the popular `uvicorn` for now:

```
$ pip install uvicorn
```

See also: *ASGI Server Installation*.

While we're at it, let's install the handy HTTPie HTTP client to help us exercise our app:

```
$ pip install httpie
```

Now let's try loading our application:

```
$ uvicorn asgilook.app:app
INFO:     Started server process [2020]
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

We can verify it works by trying to access the URL provided above by `uvicorn`:

```
$ http http://127.0.0.1:8000
HTTP/1.1 404 Not Found
content-length: 0
content-type: application/json
date: Sun, 05 Jul 2020 13:37:01 GMT
server: uvicorn
```

Woohoo, it works!!!

Well, sort of. Onwards to adding some real functionality!

Debugging ASGI Applications

While developing and testing a Falcon ASGI application along the lines of this tutorial, you may encounter unexpected issues or behaviors, be it a copy-paste mistake, an idea that didn't work out, or unusual input where validation falls outside of the scope of this tutorial.

Unlike WSGI, the ASGI specification has no standard mechanism for logging errors back to the application server, so Falcon falls back to the stdlib's `logging` (using the `falcon` `logger`).

As a well-behaved library, Falcon does not configure any loggers since that might interfere with the user's logging setup. Here's how you can set up basic logging in your ASGI Falcon application via `logging.basicConfig()`:

```
import logging

import falcon

logging.basicConfig(level=logging.INFO)


class ErrorResource:
    def on_get(self, req, resp):
        raise Exception('Something went wrong!')


app = falcon.App()
app.add_route('/error', ErrorResource())
```

When the above route is accessed, Falcon will catch the unhandled exception and automatically log an error message. Below is an example of what the log output might look like:

```
ERROR:falcon.asgi.app:Unhandled exception in ASGI application
Traceback (most recent call last):
  File "/path/to/your/app.py", line 123, in __call__
    resp = resource.on_get(req, resp)
  File "/path/to/your/app.py", line 7, in on_get
    raise Exception("Something went wrong!")
Exception: Something went wrong!
```

Note

While logging is helpful for development and debugging, be mindful of logging sensitive information. Ensure that log files are stored securely and are not accessible to unauthorized users.

Note

Unhandled errors are only logged automatically by Falcon's default error handler for `Exception`. If you *`replace this handler`* with your own generic `Exception` handler, you are responsible for logging or reporting these errors yourself.

Configuration

Next, let's make our app configurable by allowing the user to modify the file system path where images are stored. We'll also allow the UUID generator to be customized.

As Falcon does not prescribe a specific configuration library or strategy, we are free to choose our own adventure (see also a related question in our FAQ: *What is the recommended approach for app configuration?*).

In this tutorial, we'll just pass around a `Config` instance to resource initializers for easier testing (coming later in this tutorial). Create a new module, `config.py` next to `app.py`, and add the following code to it:

```
import os
import pathlib
import uuid


class Config:
    DEFAULT_CONFIG_PATH = '/tmp/asgilook'
    DEFAULT_UUID_GENERATOR = uuid.uuid4

    def __init__(self):
        self.storage_path = pathlib.Path(
            os.environ.get('ASGI_LOOK_STORAGE_PATH', self.DEFAULT_CONFIG_PATH))
        self.storage_path.mkdir(parents=True, exist_ok=True)

        self.uuid_generator = Config.DEFAULT_UUID_GENERATOR
```

Image Store

Since we are going to read and write image files, care must be taken to avoid blocking the app during I/O. We'll give `aiofiles` a try:

```
pip install aiofiles
```

In addition, let's twist the original WSGI "Look" design a bit, and convert all uploaded images to JPEG with the popular Pillow library:

```
pip install Pillow
```

We can now implement a basic async image store. Save the following code as `store.py` next to `app.py` and `config.py`:

```
import asyncio
import datetime
import io

import aiofiles
import PIL.Image

import falcon


class Image:
    def __init__(self, config, image_id, size):
        self._config = config

        self.image_id = image_id
        self.size = size
        self.modified = datetime.datetime.now(datetime.timezone.utc)

    @property
    def path(self):
        return self._config.storage_path / self.image_id

    @property
    def uri(self):
```

(continues on next page)

(continued from previous page)

```
        return f'/images/{self.image_id}.jpeg'

    def serialize(self):
        return {
            'id': self.image_id,
            'image': self.uri,
            'modified': falcon.dt_to_http(self.modified),
            'size': self.size,
        }


class Store:
    def __init__(self, config):
        self._config = config
        self._images = {}

    def _load_from_bytes(self, data):
        return PIL.Image.open(io.BytesIO(data))

    def _convert(self, image):
        rgb_image = image.convert('RGB')

        converted = io.BytesIO()
        rgb_image.save(converted, 'JPEG')
        return converted.getvalue()

    def get(self, image_id):
        return self._images.get(image_id)

    def list_images(self):
        return sorted(self._images.values(), key=lambda item: item.modified)

    async def save(self, image_id, data):
        loop = asyncio.get_running_loop()
        image = await loop.run_in_executor(None, self._load_from_bytes, data)
        converted = await loop.run_in_executor(None, self._convert, image)

        path = self._config.storage_path / image_id
        async with aiofiles.open(path, 'wb') as output:
            await output.write(converted)

        stored = Image(self._config, image_id, image.size)
        self._images[image_id] = stored
        return stored
```

Here we store data using `aiofiles`, and run `Pillow` image transformation functions in the default `ThreadPoolExecutor`, hoping that at least some of these image operations release the GIL during processing.

Note

The `ProcessPoolExecutor` is another alternative for long running tasks that do not release the GIL, such as CPU-bound pure Python code. Note, however, that `ProcessPoolExecutor` builds upon the `multiprocessing` module, and thus inherits its caveats: higher synchronization overhead, and the requirement for the task and its arguments to be picklable (which also implies that the task must be reachable from the global namespace, i.e., an anonymous `lambda` simply won't work).

Images Resource(s)

In the ASGI flavor of Falcon, all responder methods, hooks and middleware methods must be awaitable coroutines. Let's see how this works by implementing a resource to represent both a single image and a collection of images. Place the code below in a file named `images.py`:

```
import aiofiles

import falcon


class Images:
    def __init__(self, config, store):
        self._config = config
        self._store = store

    async def on_get(self, req, resp):
        resp.media = [image.serialize() for image in self._store.list_images()]

    async def on_get_image(self, req, resp, image_id):
        # NOTE: image_id: UUID is converted back to a string identifier.
        image = self._store.get(str(image_id))
        resp.stream = await aiofiles.open(image.path, 'rb')
        resp.content_type = falcon.MEDIA_JPEG

    async def on_post(self, req, resp):
        data = await req.stream.read()
        image_id = str(self._config.uuid_generator())
        image = await self._store.save(image_id, data)

        resp.location = image.uri
        resp.media = image.serialize()
        resp.status = falcon.HTTP_201
```

This module is an example of a Falcon "resource" class, as described in *Routing*. Falcon uses resource-based routing to encourage a RESTful architectural style. For each HTTP method that a resource supports, the target resource class simply implements a corresponding Python method with a name that starts with `on_` and ends in the lowercased HTTP method name (e.g., `on_get()`, `on_patch()`, `on_delete()`, etc.)

> **Note**
>
> If a Python method is omitted for a given HTTP verb, the framework will automatically respond with `405 Method Not Allowed`. Falcon also provides a default responder for OPTIONS requests that takes into account which methods are implemented for the target resource.

Here we opted to implement support for both a single image (which supports `GET` for downloading the image) and a collection of images (which supports `GET` for listing the collection, and `POST` for uploading a new image) in the same Falcon resource class. In order to make this work, the Falcon router needs a way to determine which methods to call for the collection vs. the item. This is done by using a suffixed route, as described in `add_route()` (see also: *How do I implement both POSTing and GETing items for the same resource?*).

Alternatively, we could have split the implementation to strictly represent one RESTful resource per class. In that case, there would have been no need to use suffixed responders. Depending on the application, using two classes instead of one may lead to a cleaner design. (See also: *What is the recommended way to map related routes to resource classes?*)

> **Note**
>
> In this example, we serve the image by simply assigning an open `aiofiles` file to *`resp.stream`*. This works because Falcon includes special handling for streaming async file-like objects.

> **Warning**
>
> In production deployment, serving files directly from the web server, rather than through the Falcon ASGI app, will likely be more efficient, and therefore should be preferred. See also: *Can Falcon serve static files?*

Also worth noting is that the `on_get_image()` responder will be receiving an `image_id` of type `UUID`. So what's going on here? How will the `image_id` field, matched from a string path segment, become a `UUID`?

Falcon's default router supports simple validation and transformation using *field converters*. In this example, we'll use the *`UUIDConverter`* to validate the `image_id` input as `UUID`. Converters are specified for a route by including their *shorthand identifiers* in the URI template for the route; for instance, the route corresponding to `on_get_image()` will use the following template (see also the next chapter, as well as *Routing*):

```
/images/{image_id:uuid}.jpeg
```

Since our application is still internally centered on string identifiers, feel free to experiment with refactoring the image `Store` to use `UUID`s natively!

(Alternatively, one could implement a *custom field converter* to use `uuid` only for validation, but return an unmodified string.)

> **Note**
>
> In contrast to asynchronous building blocks (responders, middleware, hooks etc.) of a Falcon ASGI application, field converters are simple synchronous data transformation functions that are not expected to perform any I/O.

Running Our Application

Now we're ready to configure the routes for our app to map image paths in the request URL to an instance of our resource class.

Let's also refactor our `app.py` module to let us invoke `create_app()` wherever we need it. This will become useful later on when we start writing test cases.

Modify `app.py` to read as follows:

```
import falcon.asgi

from .config import Config
from .images import Images
from .store import Store


def create_app(config=None):
    config = config or Config()
    store = Store(config)
    images = Images(config, store)

    app = falcon.asgi.App()
    app.add_route('/images', images)
    app.add_route('/images/{image_id:uuid}.jpeg', images, suffix='image')
```

(continues on next page)

(continued from previous page)

```
    return app
```

As mentioned earlier, we need to use a route suffix for the `Images` class to distinguish between a GET for a single image vs. the entire collection of images.

Here, we map the `'/images/{image_id:uuid}.jpeg'` URI template to a single image resource. By specifying an `'image'` suffix, we cause the framework to look for responder methods that have names ending in `'_image'` (e.g, `on_get_image()`).

We also specify the `uuid` field converter as discussed in the previous section.

In order to bootstrap an ASGI app instance for `uvicorn` to reference, we'll create a simple `asgi.py` module with the following contents:

```
from .app import create_app

app = create_app()
```

Running the application is not too dissimilar from the previous command line:

```
$ uvicorn asgilook.asgi:app
```

Provided `uvicorn` is started as per the above command line, let's try uploading some images in a separate terminal (change the picture path below to point to an existing file):

```
$ http POST localhost:8000/images @/home/user/Pictures/test.png

HTTP/1.1 201 Created
content-length: 173
content-type: application/json
date: Tue, 24 Dec 2019 17:32:18 GMT
location: /images/5cfd9fb6-259a-4c72-b8b0-5f4c35edcd3c.jpeg
server: uvicorn

{
    "id": "5cfd9fb6-259a-4c72-b8b0-5f4c35edcd3c",
    "image": "/images/5cfd9fb6-259a-4c72-b8b0-5f4c35edcd3c.jpeg",
    "modified": "Tue, 24 Dec 2019 17:32:19 GMT",
    "size": [
        462,
        462
    ]
}
```

Next, try retrieving the uploaded image:

```
$ http localhost:8000/images/5cfd9fb6-259a-4c72-b8b0-5f4c35edcd3c.jpeg

HTTP/1.1 200 OK
content-type: image/jpeg
date: Tue, 24 Dec 2019 17:34:53 GMT
server: uvicorn
transfer-encoding: chunked

+-----------------------------------------+
| NOTE: binary data not shown in terminal |
+-----------------------------------------+
```

We could also open the link in a web browser or pipe it to an image viewer to verify that the image was successfully converted to a JPEG.

Let's check the image collection now:

```
$ http localhost:8000/images

HTTP/1.1 200 OK
content-length: 175
content-type: application/json
date: Tue, 24 Dec 2019 17:36:31 GMT
server: uvicorn

[
    {
        "id": "5cfd9fb6-259a-4c72-b8b0-5f4c35edcd3c",
        "image": "/images/5cfd9fb6-259a-4c72-b8b0-5f4c35edcd3c.jpeg",
        "modified": "Tue, 24 Dec 2019 17:32:19 GMT",
        "size": [
            462,
            462
        ]
    }
]
```

The application file layout should now look like this:

```
asgilook
├── .venv
└── asgilook
    ├── __init__.py
    ├── app.py
    ├── asgi.py
    ├── config.py
    ├── images.py
    └── store.py
```

Dynamic Thumbnails

Let's pretend our image service customers want to render images in multiple resolutions, for instance, as `srcset` for responsive HTML images or other purposes.

Let's add a new method `Store.make_thumbnail()` to perform scaling on the fly:

```
async def make_thumbnail(self, image, size):
    async with aiofiles.open(image.path, 'rb') as img_file:
        data = await img_file.read()

    loop = asyncio.get_running_loop()
    return await loop.run_in_executor(None, self._resize, data, size)
```

We'll also add an internal helper to run the `Pillow` thumbnail operation that is offloaded to a threadpool executor, again, in hoping that Pillow can release the GIL for some operations:

```
def _resize(self, data, size):
    image = PIL.Image.open(io.BytesIO(data))
    image.thumbnail(size)

    resized = io.BytesIO()
```

(continues on next page)

(continued from previous page)

```
    image.save(resized, 'JPEG')
    return resized.getvalue()
```

The `store.Image` class can be extended to also return URIs to thumbnails:

```
def thumbnails(self):
    def reductions(size, min_size):
        width, height = size
        factor = 2
        while width // factor >= min_size and height // factor >= min_size:
            yield (width // factor, height // factor)
            factor *= 2

    return [
        f'/thumbnails/{self.image_id}/{width}x{height}.jpeg'
        for width, height in reductions(
            self.size, self._config.min_thumb_size)]
```

Here, we only generate URIs for a series of downsized resolutions. The actual scaling will happen on the fly upon requesting these resources.

Each thumbnail in the series is approximately half the size (one quarter area-wise) of the previous one, similar to how mipmapping works in computer graphics. You may wish to experiment with this resolution distribution.

After updating `store.py`, the module should now look like this:

```
import asyncio
import datetime
import io

import aiofiles
import PIL.Image

import falcon


class Image:
    def __init__(self, config, image_id, size):
        self._config = config

        self.image_id = image_id
        self.size = size
        self.modified = datetime.datetime.now(datetime.timezone.utc)

    @property
    def path(self):
        return self._config.storage_path / self.image_id

    @property
    def uri(self):
        return f'/images/{self.image_id}.jpeg'

    def serialize(self):
        return {
            'id': self.image_id,
            'image': self.uri,
            'modified': falcon.dt_to_http(self.modified),
            'size': self.size,
```

(continues on next page)

(continued from previous page)

```
            'thumbnails': self.thumbnails(),
        }

    def thumbnails(self):
        def reductions(size, min_size):
            width, height = size
            factor = 2
            while width // factor >= min_size and height // factor >= min_size:
                yield (width // factor, height // factor)
                factor *= 2

        return [
            f'/thumbnails/{self.image_id}/{width}x{height}.jpeg'
            for width, height in reductions(self.size, self._config.min_thumb_size)
        ]


class Store:
    def __init__(self, config):
        self._config = config
        self._images = {}

    def _load_from_bytes(self, data):
        return PIL.Image.open(io.BytesIO(data))

    def _convert(self, image):
        rgb_image = image.convert('RGB')

        converted = io.BytesIO()
        rgb_image.save(converted, 'JPEG')
        return converted.getvalue()

    def _resize(self, data, size):
        image = PIL.Image.open(io.BytesIO(data))
        image.thumbnail(size)

        resized = io.BytesIO()
        image.save(resized, 'JPEG')
        return resized.getvalue()

    def get(self, image_id):
        return self._images.get(image_id)

    def list_images(self):
        return sorted(self._images.values(), key=lambda item: item.modified)

    async def make_thumbnail(self, image, size):
        async with aiofiles.open(image.path, 'rb') as img_file:
            data = await img_file.read()

        loop = asyncio.get_running_loop()
        return await loop.run_in_executor(None, self._resize, data, size)

    async def save(self, image_id, data):
        loop = asyncio.get_running_loop()
        image = await loop.run_in_executor(None, self._load_from_bytes, data)
```

(continues on next page)

(continued from previous page)

```
        converted = await loop.run_in_executor(None, self._convert, image)

        path = self._config.storage_path / image_id
        async with aiofiles.open(path, 'wb') as output:
            await output.write(converted)

        stored = Image(self._config, image_id, image.size)
        self._images[image_id] = stored
        return stored
```

Furthermore, it is practical to impose a minimum resolution, as any potential benefit from switching between very small thumbnails (a few kilobytes each) is likely to be overshadowed by the request overhead. As you may have noticed in the above snippet, we are referencing this lower size limit as `self._config.min_thumb_size`. The *app configuration* will need to be updated to add the `min_thumb_size` option (by default initialized to 64 pixels) as follows:

```
import os
import pathlib
import uuid


class Config:
    DEFAULT_CONFIG_PATH = '/tmp/asgilook'
    DEFAULT_MIN_THUMB_SIZE = 64
    DEFAULT_UUID_GENERATOR = uuid.uuid4

    def __init__(self):
        self.storage_path = pathlib.Path(
            os.environ.get('ASGI_LOOK_STORAGE_PATH', self.DEFAULT_CONFIG_PATH))
        self.storage_path.mkdir(parents=True, exist_ok=True)

        self.uuid_generator = Config.DEFAULT_UUID_GENERATOR
        self.min_thumb_size = self.DEFAULT_MIN_THUMB_SIZE
```

Let's also add a `Thumbnails` resource to expose the new functionality. The final version of `images.py` reads:

```
import aiofiles

import falcon


class Images:
    def __init__(self, config, store):
        self._config = config
        self._store = store

    async def on_get(self, req, resp):
        resp.media = [image.serialize() for image in self._store.list_images()]

    async def on_get_image(self, req, resp, image_id):
        # NOTE: image_id: UUID is converted back to a string identifier.
        image = self._store.get(str(image_id))
        if not image:
            raise falcon.HTTPNotFound

        resp.stream = await aiofiles.open(image.path, 'rb')
        resp.content_type = falcon.MEDIA_JPEG
```

(continues on next page)

(continued from previous page)

```
    async def on_post(self, req, resp):
        data = await req.stream.read()
        image_id = str(self._config.uuid_generator())
        image = await self._store.save(image_id, data)

        resp.location = image.uri
        resp.media = image.serialize()
        resp.status = falcon.HTTP_201


class Thumbnails:
    def __init__(self, store):
        self._store = store

    async def on_get(self, req, resp, image_id, width, height):
        image = self._store.get(str(image_id))
        if not image:
            raise falcon.HTTPNotFound
        if req.path not in image.thumbnails():
            raise falcon.HTTPNotFound

        resp.content_type = falcon.MEDIA_JPEG
        resp.data = await self._store.make_thumbnail(image, (width, height))
```

Note

Even though we are only building a sample application, it is a good idea to cultivate a habit of making your code secure by design and secure by default.

In this case, we see that generating thumbnails on the fly, based on arbitrary dimensions embedded in the URI, could easily be abused to create a denial-of-service attack.

This particular attack is mitigated by validating the input (in this case, the requested path) against a list of allowed values.

Finally, a new thumbnail *route* needs to be added in `app.py`. This step is left as an exercise for the reader.

Tip

Draw inspiration from the thumbnail URI formatting string:

```
f'/thumbnails/{self.image_id}/{width}x{height}.jpeg'
```

The actual URI template for the thumbnails route should look quite similar to the above.

Remember that we want to use the *uuid* converter for the `image_id` field, and image dimensions (`width` and `height`) should ideally be converted to *int*s.

(If you get stuck, see the final version of `app.py` later in this tutorial.)

Note

If you try to request a non-existent resource (e.g., due to a missing route , or simply a typo in the URI), the framework will automatically render an `HTTP 404 Not Found` response by raising an instance of *HTTPNotFound* (unless that exception is intercepted by a *custom error handler*, or if the path matches a sink prefix).

Conversely, if a route is matched to a resource, but there is no responder for the HTTP method in question, Falcon will render `HTTP 405 Method Not Allowed` via *HTTPMethodNotAllowed*.

The new `thumbnails` end-point should now render thumbnails on the fly:

```
$ http POST localhost:8000/images @/home/user/Pictures/test.png

HTTP/1.1 201 Created
content-length: 319
content-type: application/json
date: Tue, 24 Dec 2019 18:58:20 GMT
location: /images/f2375273-8049-4b10-b17e-8851db9ac7af.jpeg
server: uvicorn

{
    "id": "f2375273-8049-4b10-b17e-8851db9ac7af",
    "image": "/images/f2375273-8049-4b10-b17e-8851db9ac7af.jpeg",
    "modified": "Tue, 24 Dec 2019 18:58:21 GMT",
    "size": [
        462,
        462
    ],
    "thumbnails": [
        "/thumbnails/f2375273-8049-4b10-b17e-8851db9ac7af/231x231.jpeg",
        "/thumbnails/f2375273-8049-4b10-b17e-8851db9ac7af/115x115.jpeg"
    ]
}


$ http localhost:8000/thumbnails/f2375273-8049-4b10-b17e-8851db9ac7af/115x115.jpeg

HTTP/1.1 200 OK
content-length: 2985
content-type: image/jpeg
date: Tue, 24 Dec 2019 19:00:14 GMT
server: uvicorn

+-----------------------------------------+
| NOTE: binary data not shown in terminal |
+-----------------------------------------+
```

Again, we could also verify thumbnail URIs in the browser or image viewer that supports HTTP input.

Caching Responses

Although scaling thumbnails on-the-fly sounds cool, and we also avoid many pesky small files littering our storage, it consumes CPU resources, and we would soon find our application crumbling under load.

Let's mitigate this problem with response caching. We'll use Redis, taking advantage of redis for async support:

```
pip install redis
```

We will also need to serialize response data (the `Content-Type` header and the body in the first version); `msgpack` should do:

```
pip install msgpack
```

Our application will obviously need access to a Redis server. Apart from just installing Redis server on your machine, one could also:

- Spin up Redis in Docker, eg:

```
docker run -p 6379:6379 redis/redis-stack:latest
```

- Assuming Redis is installed on the machine, one could also try pifpaf for spinning up Redis just temporarily for `uvicorn`:

```
pifpaf run redis -- uvicorn asgilook.asgi:app
```

We will perform caching with a Falcon *Middleware* component. Again, note that all middleware callbacks must be asynchronous. Even calling `ping()` and `close()` on the Redis connection must be `await`ed. But how can we `await` coroutines from within our synchronous `create_app()` function?

ASGI application lifespan events come to the rescue. An ASGI application server emits these events upon application startup and shutdown.

Let's implement the `process_startup()` and `process_shutdown()` handlers in our middleware to execute code upon our application's startup and shutdown, respectively:

```
async def process_startup(self, scope, event):
    await self._redis.ping()

async def process_shutdown(self, scope, event):
    await self._redis.close()
```

Warning

The Lifespan Protocol is an optional extension; please check if your ASGI server of choice implements it.

`uvicorn` (that we picked for this tutorial) supports Lifespan.

At minimum, our middleware will need to know the Redis host(s) to use. Let's also make our Redis connection factory configurable to afford injecting different Redis client implementations for production and testing.

Note

Rather than requiring the caller to pass the host to the connection factory, a wrapper method could be used to implicitly reference `self.redis_host`. Such a design might prove helpful for apps that need to create client connections in more than one place.

Assuming we call our new *configuration* item `redis_host` the final version of `config.py` now reads:

```
import os
import pathlib
import uuid

import redis.asyncio


class Config:
    DEFAULT_CONFIG_PATH = '/tmp/asgilook'
    DEFAULT_MIN_THUMB_SIZE = 64
    DEFAULT_REDIS_FROM_URL = redis.asyncio.from_url
    DEFAULT_REDIS_HOST = 'redis://localhost'
    DEFAULT_UUID_GENERATOR = uuid.uuid4

    def __init__(self):
```

(continues on next page)

(continued from previous page)

```
        self.storage_path = pathlib.Path(
            os.environ.get('ASGI_LOOK_STORAGE_PATH', self.DEFAULT_CONFIG_PATH)
        )
        self.storage_path.mkdir(parents=True, exist_ok=True)

        self.min_thumb_size = self.DEFAULT_MIN_THUMB_SIZE
        self.redis_from_url = Config.DEFAULT_REDIS_FROM_URL
        self.redis_host = self.DEFAULT_REDIS_HOST
        self.uuid_generator = Config.DEFAULT_UUID_GENERATOR
```

Let's complete the Redis cache component by implementing two more middleware methods, in addition to `process_startup()` and `process_shutdown()`. Create a `cache.py` module containing the following code:

```
import msgpack


class RedisCache:
    PREFIX = 'asgilook:'
    INVALIDATE_ON = frozenset({'DELETE', 'POST', 'PUT'})
    CACHE_HEADER = 'X-ASGILook-Cache'
    TTL = 3600

    def __init__(self, config):
        self._config = config
        self._redis = self._config.redis_from_url(self._config.redis_host)

    async def _serialize_response(self, resp):
        data = await resp.render_body()
        return msgpack.packb([resp.content_type, data], use_bin_type=True)

    def _deserialize_response(self, resp, data):
        resp.content_type, resp.data = msgpack.unpackb(data, raw=False)
        resp.complete = True
        resp.context.cached = True

    async def process_startup(self, scope, event):
        await self._redis.ping()

    async def process_shutdown(self, scope, event):
        await self._redis.aclose()

    async def process_request(self, req, resp):
        resp.context.cached = False

        if req.method in self.INVALIDATE_ON:
            return

        key = f'{self.PREFIX}/{req.path}'
        data = await self._redis.get(key)
        if data is not None:
            self._deserialize_response(resp, data)
            resp.set_header(self.CACHE_HEADER, 'Hit')
        else:
            resp.set_header(self.CACHE_HEADER, 'Miss')

    async def process_response(self, req, resp, resource, req_succeeded):
```

(continues on next page)

(continued from previous page)

```
        if not req_succeeded:
            return

        key = f'{self.PREFIX}/{req.path}'

        if req.method in self.INVALIDATE_ON:
            await self._redis.delete(key)
        elif not resp.context.cached:
            data = await self._serialize_response(resp)
            await self._redis.set(key, data, ex=self.TTL)
```

For caching to take effect, we also need to modify `app.py` to add the `RedisCache` component to our application's middleware list. The final version of `app.py` should look something like this:

```
import falcon.asgi

from .cache import RedisCache
from .config import Config
from .images import Images
from .images import Thumbnails
from .store import Store


def create_app(config=None):
    config = config or Config()
    cache = RedisCache(config)
    store = Store(config)
    images = Images(config, store)
    thumbnails = Thumbnails(store)

    app = falcon.asgi.App(middleware=[cache])
    app.add_route('/images', images)
    app.add_route('/images/{image_id:uuid}.jpeg', images, suffix='image')
    app.add_route(
        '/thumbnails/{image_id:uuid}/{width:int}x{height:int}.jpeg', thumbnails
    )

    return app
```

Now, subsequent access to `/thumbnails` should be cached, as indicated by the `x-asgilook-cache` header:

```
$ http localhost:8000/thumbnails/167308e4-e444-4ad9-88b2-c8751a4e37d4/115x115.jpeg

HTTP/1.1 200 OK
content-length: 2985
content-type: image/jpeg
date: Tue, 24 Dec 2019 19:46:51 GMT
server: uvicorn
x-asgilook-cache: Hit

+-----------------------------------------+
| NOTE: binary data not shown in terminal |
+-----------------------------------------+
```

Note

Left as another exercise for the reader: individual images are streamed directly from `aiofiles` instances, and caching therefore does not work for them at the moment.

The project's structure should now look like this:

```
asgilook
├── .venv
└── asgilook
    ├── __init__.py
    ├── app.py
    ├── asgi.py
    ├── cache.py
    ├── config.py
    ├── images.py
    └── store.py
```

Testing Our Application

So far, so good? We have only tested our application by sending a handful of requests manually. Have we tested all code paths? Have we covered typical user inputs to the application?

Creating a comprehensive test suite is vital not only for verifying that the application is behaving correctly at the moment, but also for limiting the impact of any regressions introduced into the codebase over time.

In order to implement a test suite, we'll need to revise our dependencies and decide which abstraction level we are after:

- Will we run a real Redis server?
- Will we store "real" files on a filesystem or just provide a fixture for `aiofiles`?
- Will we inject real dependencies, or use mocks and monkey patching?

There is no right and wrong here, as different testing strategies (or a combination thereof) have their own advantages in terms of test running time, how easy it is to implement new tests, how similar the test environment is to production, etc.

Another thing to choose is a testing framework. Just as in the *WSGI tutorial*, let's use pytest. This is a matter of taste; if you prefer xUnit/JUnit-style layout, you'll feel at home with the stdlib's `unittest`.

In order to more quickly deliver a working solution, we'll allow our tests to access the real filesystem. For our convenience, `pytest` offers several temporary directory utilities out of the box. Let's wrap its `tmpdir_factory` to create a simple `storage_path` fixture that we'll share among all tests in the suite (in the `pytest` parlance, a "session"-scoped fixture).

Tip

The `pytest` website includes in-depth documentation on the use of fixtures. Please visit pytest fixtures: explicit, modular, scalable to learn more.

As mentioned in the *previous section*, there are many ways to spin up a temporary or permanent Redis server; or mock it altogether. For our tests, we'll try fakeredis, a pure Python implementation tailored specifically for writing unit tests.

`pytest` and `fakeredis` can be installed as:

```
$ pip install fakeredis pytest
```

We'll also create a directory for our tests and make it a Python package to avoid any problems with importing local utility modules or checking code coverage:

```
$ mkdir -p tests
$ touch tests/__init__.py
```

Next, let's implement fixtures to replace `uuid` and `redis`, and inject them into our tests via `conftest.py` (place your code in the newly created `tests` directory):

```
import io
import random
import uuid

import fakeredis.aioredis
import PIL.Image
import PIL.ImageDraw
import pytest

import falcon.asgi
import falcon.testing

from asgilook.app import create_app
from asgilook.config import Config


@pytest.fixture()
def predictable_uuid():
    fixtures = (
        uuid.UUID('36562622-48e5-4a61-be67-e426b11821ed'),
        uuid.UUID('3bc731ac-8cd8-4f39-b6fe-1a195d3b4e74'),
        uuid.UUID('ba1c4951-73bc-45a4-a1f6-aa2b958dafa4'),
    )

    def uuid_func():
        try:
            return next(fixtures_it)
        except StopIteration:
            return uuid.uuid4()

    fixtures_it = iter(fixtures)
    return uuid_func


@pytest.fixture(scope='session')
def storage_path(tmpdir_factory):
    return tmpdir_factory.mktemp('asgilook')


@pytest.fixture
def client(predictable_uuid, storage_path):
    # NOTE(vytas): Unlike the sync FakeRedis, fakeredis.aioredis.FakeRedis
    #   seems to share a global state in 2.17.0 (by design or oversight).
    #   Make sure we initialize a new fake server for every test case.
    def fake_redis_from_url(*args, **kwargs):
        server = fakeredis.FakeServer()
        return fakeredis.aioredis.FakeRedis(server=server)

    config = Config()
    config.redis_from_url = fake_redis_from_url
    config.redis_host = 'redis://localhost'
```

(continues on next page)

(continued from previous page)

```
    config.storage_path = storage_path
    config.uuid_generator = predictable_uuid

    app = create_app(config)
    return falcon.testing.TestClient(app)


@pytest.fixture(scope='session')
def png_image():
    image = PIL.Image.new('RGBA', (640, 360), color='black')

    draw = PIL.ImageDraw.Draw(image)
    for _ in range(32):
        x0 = random.randint(20, 620)
        y0 = random.randint(20, 340)
        x1 = random.randint(20, 620)
        y1 = random.randint(20, 340)
        if x0 > x1:
            x0, x1 = x1, x0
        if y0 > y1:
            y0, y1 = y1, y0
        draw.ellipse([(x0, y0), (x1, y1)], fill='yellow', outline='red')

    output = io.BytesIO()
    image.save(output, 'PNG')
    return output.getvalue()


@pytest.fixture(scope='session')
def image_size():
    def report_size(data):
        image = PIL.Image.open(io.BytesIO(data))
        return image.size

    return report_size
```

Note

In the `png_image` fixture above, we are drawing random images that will look different every time the tests are run.

If your testing flow affords that, it is often a great idea to introduce some unpredictability in your test inputs. This will provide more confidence that your application can handle a broader range of inputs than just 2-3 test cases crafted specifically for that sole purpose.

On the other hand, random inputs can make assertions less stringent and harder to formulate, so judge according to what is the most important for your application. You can also try to combine the best of both worlds by using a healthy mix of rigid fixtures and fuzz testing.

Note

More information on `conftest.py`'s anatomy and `pytest` configuration can be found in the latter's documentation: conftest.py: local per-directory plugins.

With the groundwork in place, we can start with a simple test that will attempt to GET the `/images` resource. Place

the following code in a new `tests/test_images.py` module:

```
def test_list_images(client):
    resp = client.simulate_get('/images')

    assert resp.status_code == 200
    assert resp.json == []
```

Let's give it a try:

```
$ pytest tests/test_images.py

========================== test session starts ==========================
platform linux -- Python 3.12.11, pytest-8.4.1, pluggy-1.6.0
rootdir: /falcon/tutorials/asgilook
collected 1 item

tests/test_images.py .                                            [100%]

=========================== 1 passed in 0.01s ===========================
```

Success! 🎉

At this point, our project structure, containing the `asgilook` and `test` packages, should look like this:

```
asgilook
├── .venv
├── asgilook
│   ├── __init__.py
│   ├── app.py
│   ├── asgi.py
│   ├── cache.py
│   ├── config.py
│   ├── images.py
│   └── store.py
└── tests
    ├── __init__.py
    ├── conftest.py
    └── test_images.py
```

Now, we need more tests! Try adding a few more test cases to `tests/test_images.py`, using the *WSGI Testing Tutorial* as your guide (the interface for Falcon's testing framework is mostly the same for ASGI vs. WSGI). Additional examples are available under `examples/asgilook/tests` in the Falcon repository.

> **Tip**
>
> For more advanced test cases, the *`falcon.testing.ASGIConductor`* class is worth a look.

Code Coverage

How much of our `asgilook` code is covered by these tests?

An easy way to get a coverage report is by using the `pytest-cov` plugin (available on PyPi).

After installing `pytest-cov` we can generate a coverage report as follows:

```
$ pytest --cov=asgilook --cov-report=term-missing tests/
```

Oh, wow! We do happen to have full line coverage, except for `asgilook/asgi.py`. If desired, we can instruct `coverage` to omit this module by listing it in the `omit` section of a `.coveragerc` file.

What is more, we could turn the current coverage into a requirement by adding `--cov-fail-under=100` (or any other percent threshold) to our `pytest` command.

> **Note**
>
> The `pytest-cov` plugin is quite simplistic; more advanced testing strategies such as blending different types of tests and/or running the same tests in multiple environments would most probably involve running `coverage` directly, and combining results.

What Now?

Congratulations, you have successfully completed the Falcon ASGI tutorial!

Needless to say, our sample ASGI application could still be improved in numerous ways:

- Make the image store persistent and reusable across worker processes. Maybe by using a database?
- Improve error handling for malformed images.
- Check how and when Pillow releases the GIL, and tune what is offloaded to a threadpool executor.
- Test Pillow-SIMD to boost performance.
- Publish image upload events via *`SSE`* or *WebSockets*.
- …And much more (patches welcome, as they say)!

> **Tip**
>
> If you want to add *WebSocket* support, please check out our *WebSocket tutorial* too!

Compared to the sync version, asynchronous code can at times be harder to design and reason about. Should you run into any issues, our friendly community is available to answer your questions and help you work through any sticky problems (see also: *Getting Help*).

5.1.6 Tutorial (WebSockets)

In this tutorial, we're going to build a WebSocket server using Falcon. We'll start with a simple server that echoes back any message it receives.

We'll then add more functionality to the server, such as sending JSON data and logging messages.

> **Note**
>
> This tutorial covers the asynchronous flavor of Falcon using the ASGI protocol.
>
> A Falcon WebSocket server builds upon the ASGI WebSocket specification. Therefore it's not supported in a Falcon WSGI application.

First Steps

We'll start with a clean working directory and create a new virtual environment using the `venv` module:

```
$ mkdir ws_tutorial
$ cd ws_tutorial
$ python3 -m venv .venv
$ source .venv/bin/activate
```

Create the following directory structure:

```
ws_tutorial
├── .venv
└── ws_tutorial
    ├── __init__.py
    └── app.py
```

And next we'll *install Falcon* and Uvicorn in our freshly created virtual environment:

```
$ pip install falcon uvicorn
```

Now, let's create a simple Falcon application to ensure our project is working as expected.

```
import falcon.asgi
import uvicorn

app = falcon.asgi.App()

class HelloWorldResource:
    async def on_get(self, req, resp):
        resp.media = {'hello': 'world'}

app.add_route('/hello', HelloWorldResource())

if __name__ == '__main__':
    uvicorn.run(app, host='localhost', port=8000)
```

Now we can test the application with `httpie` (installable with `pip install httpie`) by running the following command:

```
$ http localhost:8000/hello

 HTTP/1.1 200 OK
 content-length: 18
 content-type: application/json
 date: Sat, 13 Jul 2024 09:13:24 GMT
 server: uvicorn

 {
     "hello": "world"
 }
```

Awesome, it works! Now let's move on to building our WebSocket server.

WebSockets Server

We will update our server to include a websocket route that will echo back any message it receives. Later we'll update the server with more logic, but for now, let's keep it simple.

```
import falcon.asgi
from falcon import WebSocketDisconnected
from falcon.asgi import Request, WebSocket
import uvicorn

app = falcon.asgi.App()


class HelloWorldResource:
    async def on_get(self, req, resp):
```

(continues on next page)

(continued from previous page)

```
        resp.media = {'hello': 'world'}


class EchoWebSocketResource:
    async def on_websocket(self, req: Request, ws: WebSocket):
        try:
            await ws.accept()
        except WebSocketDisconnected:
            return

        while True:
            try:
                message = await ws.receive_text()
                await ws.send_text(f"Received the following text: {message}")
            except WebSocketDisconnected:
                return


app.add_route('/hello', HelloWorldResource())
app.add_route('/echo', EchoWebSocketResource())

if __name__ == '__main__':
    uvicorn.run(app, host='localhost', port=8000)
```

We'll also need to install a websockets library. There are multiple ways to do this:

```
$ pip install websockets
or
$ pip install uvicorn[standard]
or
$ wsproto
```

To test the new WebSocket route, we can use the websocat tool:

```
$ websocat ws://localhost:8000/echo
$ hello
Received the following text: hello
```

Cool! We have a working WebSocket server. Now let's add some more functionality to our server.

To make this easier, we'll create a simple client that will send messages to our server.

Simple Client

Create a new file called `client.py` in the same directory as `app.py`. The client will ask for your input and send it to the server.:

```
# This is a simple example of a WebSocket client that sends a message to the␣
↪server.
# Since it's an example using the `websockets` library, and it isn't using anything
# specific to Falcon, there are no tests. Coverage is skipped for this module.

import asyncio

import websockets
```

(continues on next page)

(continued from previous page)

```
async def send_message():
    uri = 'ws://localhost:8000/echo'

    async with websockets.connect(uri) as websocket:
        while True:
            message = input('Enter a message (q to exit): ')
            if message.casefold() == 'q':
                break
            await websocket.send(message)
            response = await websocket.recv()
            print(response)


if __name__ == '__main__':
    asyncio.run(send_message())
```

Run this client in a separate terminal:

```
$ python client.py
Enter a message: Hi
Received the following text: Hi
```

This will simplify testing our server.

Now let's add some more functionality to our server.

We've been working with text input/output - let's try sending sending some JSON data.

```
from datetime import datetime

import falcon.asgi
from falcon import WebSocketDisconnected
from falcon.asgi import Request, WebSocket
import uvicorn

app = falcon.asgi.App()


class HelloWorldResource:
    async def on_get(self, req, resp):
        resp.media = {'hello': 'world'}


class EchoWebSocketResource:
    async def on_websocket(self, req: Request, ws: WebSocket):
        try:
            await ws.accept()
        except WebSocketDisconnected:
            return

        while True:
            try:
                message = await ws.receive_text()
                await ws.send_media({'message': message, 'date': datetime.now().
↪isoformat()})
            except WebSocketDisconnected:
                return
```

(continues on next page)

(continued from previous page)

```
app.add_route('/hello', HelloWorldResource())
app.add_route('/echo', EchoWebSocketResource())

if __name__ == '__main__':
    uvicorn.run(app, host='localhost', port=8000)
```

```
$ python client.py
$ Enter a message: Hi
  {"message": "Hi", "date": "2024-07-13T12:11:51.758923"}
```

Note

By default, send_media() and receive_media() will serialize to (and deserialize from) JSON for a TEXT payload, and to/from MessagePack for a BINARY payload (see also: Built-in Media Handlers).

Lets try to query for data from the server. We'll create a new resource that will return a report based on the query.

Server side:

```
from datetime import datetime

import falcon.asgi
from falcon import WebSocketDisconnected
from falcon.asgi import Request, WebSocket
import uvicorn

REPORTS = {
    'report1': {
        'title': 'Report 1',
        'content': 'This is the content of report 1',
    },
    'report2': {
        'title': 'Report 2',
        'content': 'This is the content of report 2',
    },
    'report3': {
        'title': 'Report 3',
        'content': 'This is the content of report 3',
    },
    'report4': {
        'title': 'Report 4',
        'content': 'This is the content of report 4',
    },
}

app = falcon.asgi.App()


class HelloWorldResource:
    async def on_get(self, req, resp):
        resp.media = {'hello': 'world'}


class EchoWebSocketResource:
```

(continues on next page)

(continued from previous page)

```
    async def on_websocket(self, req: Request, ws: WebSocket):
        try:
            await ws.accept()
        except WebSocketDisconnected:
            return

        while True:
            try:
                message = await ws.receive_text()
                await ws.send_media({'message': message, 'date': datetime.now().
↪isoformat()})
            except WebSocketDisconnected:
                return

class ReportsResource:
    async def on_websocket(self, req: Request, ws: WebSocket):
        try:
            await ws.accept()
        except WebSocketDisconnected:
            return

        while True:
            try:
                query = await ws.receive_text()
                report = REPORTS.get(query, None)
                print(report)

                if report is None:
                    await ws.send_media({'error': 'report not found'})
                    continue

                await ws.send_media({'report': report["title"]})
            except WebSocketDisconnected:
                return


app.add_route('/hello', HelloWorldResource())
app.add_route('/echo', EchoWebSocketResource())
app.add_route('/reports', ReportsResource())


if __name__ == '__main__':
    uvicorn.run(app, host='localhost', port=8000)
```

We'll also create new client app (*reports_client.py*), that will connect to the reports endpoint. :

```
import asyncio
import websockets


async def send_message():
    uri = "ws://localhost:8000/reports"
    async with websockets.connect(uri) as websocket:
        while True:
            message = input("Name of the log: ")
            await websocket.send(message)
```

(continues on next page)

(continued from previous page)

```
            response = await websocket.recv()
            print(response)


if __name__ == "__main__":
    asyncio.run(send_message())
```

We've added a new resource that will return a report based on the query. The client will send a query to the server, and the server will respond with the report. If it can't find the report, it will respond with an error message.

This is a simple example, but you can easily extend it to include more complex logic like fetching data from a database.

Middleware

Falcon supports middleware, which can be used to add functionality to the application. For example, we can add a middleware that prints when a connection is established.

```
from datetime import datetime

import falcon.asgi
from falcon import WebSocketDisconnected
from falcon.asgi import Request, WebSocket
import uvicorn

REPORTS = {
    'report1': {
        'title': 'Report 1',
        'content': 'This is the content of report 1',
    },
    'report2': {
        'title': 'Report 2',
        'content': 'This is the content of report 2',
    },
    'report3': {
        'title': 'Report 3',
        'content': 'This is the content of report 3',
    },
    'report4': {
        'title': 'Report 4',
        'content': 'This is the content of report 4',
    },
}

app = falcon.asgi.App()

class LoggerMiddleware:
    async def process_request_ws(self, req: Request, ws: WebSocket):
        # This will be called for the HTTP request that initiates the
        #   WebSocket handshake before routing.
        pass

    async def process_resource_ws(self, req: Request, ws: WebSocket, resource,␣
↪params):
        # This will be called for the HTTP request that initiates the
        #   WebSocket handshake after routing (if a route matches the
        #   request).
        print(f'WebSocket connection established on {req.path}')
```

(continues on next page)

(continued from previous page)

```
class HelloWorldResource:
    async def on_get(self, req, resp):
        resp.media = {'hello': 'world'}


class EchoWebSocketResource:
    async def on_websocket(self, req: Request, ws: WebSocket):
        try:
            await ws.accept()
        except WebSocketDisconnected:
            return

        while True:
            try:
                message = await ws.receive_text()
                await ws.send_media({'message': message, 'date': datetime.now().
↪isoformat()})
            except WebSocketDisconnected:
                return

class ReportsResource:
    async def on_websocket(self, req: Request, ws: WebSocket):
        try:
            await ws.accept()
        except WebSocketDisconnected:
            return

        while True:
            try:
                query = await ws.receive_text()
                report = REPORTS.get(query, None)
                print(report)

                if report is None:
                    await ws.send_media({'error': 'report not found'})
                    continue

                await ws.send_media({'report': report["title"]})
            except WebSocketDisconnected:
                return


app.add_route('/hello', HelloWorldResource())
app.add_route('/echo', EchoWebSocketResource())
app.add_route('/reports', ReportsResource())

app.add_middleware(LoggerMiddleware())


if __name__ == '__main__':
    uvicorn.run(app, host='localhost', port=8000)
```

Now, when you run the server, you should see a message in the console when a WebSocket connection is established.

Authentication

Adding authentication can be done with the help of middleware as well. Authentication can be done a few ways. In this example we'll use the **First message** method, as described on the websockets documentation.

There are some considerations to take into account when implementing authentication in a WebSocket server.

Updated server code:

```
from datetime import datetime
import logging
import pathlib

import uvicorn

from falcon import WebSocketDisconnected
import falcon.asgi
from falcon.asgi import Request
from falcon.asgi import WebSocket

logger = logging.getLogger('ws-logger')
logger.setLevel('INFO')
logger.addHandler(logging.StreamHandler())


REPORTS = {
    'report1': {
        'title': 'Report 1',
        'content': 'This is the content of report 1',
    },
    'report2': {
        'title': 'Report 2',
        'content': 'This is the content of report 2',
    },
    'report3': {
        'title': 'Report 3',
        'content': 'This is the content of report 3',
    },
    'report4': {
        'title': 'Report 4',
        'content': 'This is the content of report 4',
    },
}

app = falcon.asgi.App()


class LoggerMiddleware:
    async def process_request_ws(self, req: Request, ws: WebSocket):
        # This will be called for the HTTP request that initiates the
        #   WebSocket handshake before routing.
        pass

    async def process_resource_ws(self, req: Request, ws: WebSocket, resource,␣
↪params):
        # This will be called for the HTTP request that initiates the
        #   WebSocket handshake after routing (if a route matches the
        #   request).
        logger.info('WebSocket connection established on %r', req.path)
```

(continues on next page)

(continued from previous page)

```
class AuthMiddleware:
    def __init__(self, protected_routes: list[str] | None = None):
        if protected_routes is None:
            protected_routes = []

        self.protected_routes = protected_routes

    async def process_request_ws(self, req: Request, ws: WebSocket):
        # Opening a connection so we can receive the token
        await ws.accept()

        # Check if the route is protected
        if req.path not in self.protected_routes:
            return

        token = await ws.receive_text()

        if token != 'very secure token':
            await ws.close(1008)
            return

        # Never log tokens in production
        logger.info('Client with token %r Authenticated', token)


class HelloWorldResource:
    async def on_get(self, req, resp):
        resp.media = {'hello': 'world'}


class EchoWebSocketResource:
    async def on_websocket(self, req: Request, ws: WebSocket):
        while True:
            try:
                message = await ws.receive_text()
                await ws.send_media(
                    {'message': message, 'date': datetime.now().isoformat()}
                )
            except WebSocketDisconnected:
                return


class ReportsResource:
    async def on_websocket(self, req: Request, ws: WebSocket):
        while True:
            try:
                query = await ws.receive_text()
                report = REPORTS.get(query, None)
                logger.info('selected report: %s', report)

                if report is None:
                    await ws.send_media({'error': 'report not found'})
                    continue
```

(continues on next page)

(continued from previous page)

```
                await ws.send_media({'report': report['title']})
            except WebSocketDisconnected:
                return


app.add_route('/hello', HelloWorldResource())
app.add_route('/echo', EchoWebSocketResource())
app.add_route('/reports', ReportsResource())

app.add_middleware(LoggerMiddleware())
app.add_middleware(AuthMiddleware(['/reports']))

# usually a web server, like Nginx or Caddy, should serve static assets, but
# for the purpose of this example we use falcon.
static_path = pathlib.Path(__file__).parent / 'static'
app.add_static_route('/', static_path, fallback_filename='index.html')

if __name__ == '__main__':
    uvicorn.run(app, host='localhost', port=8000)  # pragma: no cover
```

Updated client code for the reports client:

```
# This is a simple example of a WebSocket client that sends a message to the␣
↪server.
# Since it's an example using the `websockets` library and it isn't using anything
# specific to Falcon, there are no tests. Coverage is skipped for this module.

import asyncio

import websockets


async def send_message():
    uri = 'ws://localhost:8000/reports'

    async with websockets.connect(uri) as websocket:
        # Send the authentication token
        await websocket.send('very secure token')

        while True:
            message = input('Name of the log (q to exit): ')
            if message.casefold() == 'q':
                break
            await websocket.send(message)
            response = await websocket.recv()
            print(response)


if __name__ == '__main__':
    asyncio.run(send_message())
```

Things we've changed:

- Added a new middleware class *AuthMiddleware* that will check the token on the first message.
- Opening a WebSocket connection is now handled by the middleware.
- The client now sends a token as the first message, if required for that route.

- Falcon was configured to serve a simple HTML page to use the echo WebSocket client for a browser.

If you try to query the reports endpoint now, everything works as expected on an authenticated route. But as soon as you remove/modify the token, the connection will be closed (after sending the first query - a downside of first-message authentication).

```
$ python reports_client.py
[...]
websockets.exceptions.ConnectionClosedError: received 1008 (policy violation);␣
↪then sent 1008 (policy violation)
```

Note

This is a simple example of how to add authentication to a WebSocket server. In a real-world application, you would want to use a more secure method of authentication, such as JWT tokens.

What Now

This tutorial is just the beginning. You can extend the server with more complex logic. For example, you could add a database to store/retrieve the reports, or add more routes to the server.

For more information on websockets in Falcon, check out the WebSocket API.

5.1.7 Recipes

Capitalizing Response Header Names

Falcon always renders WSGI response header names in lower case; see also: *Why is Falcon changing my header names to lowercase?*

While this should normally never be an issue for standards-conformant HTTP clients, it is possible to override HTTP headers using generic WSGI middleware:

```
class CustomHeadersMiddleware:
    def __init__(self, app, title_case=True, custom_capitalization=None):
        self._app = app
        self._title_case = title_case
        self._capitalization = custom_capitalization or {}

    def __call__(self, environ, start_response):
        def start_response_wrapper(status, response_headers, exc_info=None):
            if self._title_case:
                headers = [
                    (self._capitalization.get(name, name.title()), value)
                    for name, value in response_headers
                ]
            else:
                headers = [
                    (self._capitalization.get(name, name), value)
                    for name, value in response_headers
                ]
            start_response(status, headers, exc_info)

        return self._app(environ, start_response_wrapper)
```

We can now use this middleware to wrap a Falcon app:

```
import falcon
```

(continues on next page)

(continued from previous page)

```
# Import or copy CustomHeadersMiddleware from the above snippet
class CustomHeadersMiddleware: ...


class FunkyResource:
    def on_get(self, req, resp):
        resp.set_header('X-Funky-Header', 'test')
        resp.media = {'message': 'Hello'}


app = falcon.App()
app.add_route('/test', FunkyResource())

app = CustomHeadersMiddleware(
    app,
    custom_capitalization={'x-funky-header': 'X-FuNkY-HeADeR'},
)
```

As a bonus, this recipe applies to non-Falcon WSGI applications too.

Msgspec Integration

This recipe illustrates how the popular msgspec data serialization and validation library can be used with Falcon.

Media handlers

`msgspec` can be used for JSON serialization by simply instantiating *JSONHandler* with the `msgspec.json.decode()` and `msgspec.json.encode()` functions:

```
import msgspec

import falcon.media

json_handler = falcon.media.JSONHandler(
    dumps=msgspec.json.encode,
    loads=msgspec.json.decode,
)
```

Note

It would be more efficient to use preconstructed `msgspec.json.Decoder` and `msgspec.json.Encoder` instead of initializing them every time via `msgspec.json.decode()` and `msgspec.json.encode()`, respectively. This optimization is left as an exercise for the reader.

Using `msgspec` for handling MessagePack (or YAML) media is slightly more involved, as we need to implement the *BaseHandler* interface:

```
from typing import Optional

from msgspec import msgpack

from falcon import media
from falcon.typing import ReadableIO
```

(continues on next page)

(continued from previous page)

```
class MsgspecMessagePackHandler(media.BaseHandler):
    def deserialize(
        self,
        stream: ReadableIO,
        content_type: Optional[str],
        content_length: Optional[int],
    ) -> object:
        return msgpack.decode(stream.read())

    def serialize(self, media: object, content_type: str) -> bytes:
        return msgpack.encode(media)


msgpack_handler = MsgspecMessagePackHandler()
```

Attention

In contrast to *JSONHandler*, we would also need to implement error handling for invalid MessagePack payload. See more in the below chapter: *Error handling*.

We can now use these handlers for request and response media (see also: *Replacing the Default Handlers*).

Media validation

Falcon currently only provides optional *media validation* using JSON Schema, so we will implement validation separately using *process_resource middleware*. To that end, let us assume that resources can expose schema attributes based on the HTTP verb: `PATCH_SCHEMA`, `POST_SCHEMA`, etc, pointing to the `msgspec.Struct` in question. We will inject the validated object into *params*:

```
import msgspec

from falcon import Request
from falcon import Response


class MsgspecMiddleware:
    def process_resource(
        self, req: Request, resp: Response, resource: object, params: dict
    ) -> None:
        if schema := getattr(resource, f'{req.method}_SCHEMA', None):
            param = schema.__name__.lower()
            params[param] = msgspec.convert(req.get_media(), schema)
```

Here, the name of the injected parameter is simply a lowercase version of the schema's class name. We could instead store this name in a constant on the resource, or on the schema class.

Note that while the above middleware is only suitable for synchronous (WSGI) Falcon applications, you could easily transform this method to `async`, or even implement an async version in parallel on the same middleware as `process_request_async(...)`. Just do not forget to `await` *req.get_media()*!

The astute reader may also notice that the process can be optimized even further, as `msgspec` affords combined deserialization and validation in a single call to `decode()` without going via an intermediate Python object (in this case, effectively *req.media*). However, if one wants to support more than just a single request media handler, all the media functionality would need to be reimplemented almost from scratch.

We are looking into providing better affordances for this scenario in the framework itself (see also #2202 on GitHub),

as well as offering centralized media validation connected to the application's OpenAPI specification. Your input is extremely valuable for us, so do not hesitate to *get in touch*, and share your vision!

Error handling

Schema validation can fail, and the resulting exception would unexpectedly bubble up as a generic HTTP 500 error. We can do better! Skimming through `msgspec`'s docs, we find out that this case is represented by `msgspec.ValidationError`. We could either create an *error hander* that reraises this exception as *MediaValidationError*, or just use a `try.. except` clause, and reraise directly inside middleware.

`msgspec.json.decode()` has another peculiarity: unlike the stdlib's `json` or the majority of other JSON libraries, `msgspec.DecodeError` is not a subclass of `ValueError`:

```
>>> import msgspec
>>> issubclass(msgspec.DecodeError, ValueError)
False
```

Falcon's *JSONHandler* works around this discrepancy by detecting the actual deserialization exception type at the time of initialization, but you may still encounter the issue when using the library manually, or when using it for other media types such as MessagePack (see above). Furthermore, the problem has been reported upstream, and received positive feedback from the maintainer, so hopefully it could get resolved in the near future.

Complete recipe

Finally, we combine these snippets into a note taking application:

```
from datetime import datetime
from datetime import timezone
from functools import partial
from http import HTTPStatus
from typing import Annotated, Any
import uuid

import msgspec

import falcon
from falcon import Request
from falcon import Response
from falcon.media import JSONHandler


def _utc_now() -> datetime:
    return datetime.now(timezone.utc)


class Note(msgspec.Struct):
    text: Annotated[str, msgspec.Meta(max_length=256)]
    noteid: uuid.UUID = msgspec.field(default_factory=uuid.uuid4)
    created: datetime = msgspec.field(
        default_factory=partial(datetime.now, timezone.utc)
    )


class NoteResource:
    POST_SCHEMA = Note

    def __init__(self) -> None:
        # NOTE: In a real-world app, you would want to use persistent storage.
```

(continues on next page)

(continued from previous page)

```
        self._store: dict[str, Note] = {}

    def on_get_note(self, req: Request, resp: Response, noteid: uuid.UUID) -> None:
        resp.media = self._store.get(str(noteid))
        if not resp.media:
            raise falcon.HTTPNotFound(
                description=f'Note with {noteid=} is not in the store.'
            )

    def on_delete_note(self, req: Request, resp: Response, noteid: uuid.UUID) ->␣
↪None:
        self._store.pop(str(noteid), None)
        resp.status = HTTPStatus.NO_CONTENT

    def on_get(self, req: Request, resp: Response) -> None:
        resp.media = self._store

    def on_post(self, req: Request, resp: Response, note: Note) -> None:
        self._store[str(note.noteid)] = note
        resp.location = f'{req.path}/{note.noteid}'
        resp.media = note
        resp.status = HTTPStatus.CREATED


class MsgspecMiddleware:
    def process_resource(
        self, req: Request, resp: Response, resource: object, params: dict[str,␣
↪Any]
    ) -> None:
        if schema := getattr(resource, f'{req.method}_SCHEMA', None):
            param = schema.__name__.lower()
            params[param] = msgspec.convert(req.get_media(), schema)


def _handle_validation_error(
    req: Request, resp: Response, ex: msgspec.ValidationError, params: dict[str,␣
↪Any]
) -> None:
    raise falcon.HTTPUnprocessableEntity(description=str(ex))


def create_app() -> falcon.App:
    app = falcon.App(middleware=[MsgspecMiddleware()])
    app.add_error_handler(msgspec.ValidationError, _handle_validation_error)

    json_handler = JSONHandler(
        dumps=msgspec.json.encode,
        loads=msgspec.json.decode,
    )
    app.req_options.media_handlers[falcon.MEDIA_JSON] = json_handler
    app.resp_options.media_handlers[falcon.MEDIA_JSON] = json_handler

    notes = NoteResource()
    app.add_route('/notes', notes)
    app.add_route('/notes/{noteid:uuid}', notes, suffix='note')
```

(continues on next page)

(continued from previous page)

```
    return app


application = create_app()
```

We can now `POST` a new note, view the whole collection, or just `GET` an individual note. (MessagePack support was omitted for brevity.)

Parsing Nested Multipart Forms

Out of the box, Falcon does not offer official support for parsing nested multipart forms (i.e., where multiple files for a single field are transmitted using a nested `multipart/mixed` part).

> **Note**
>
> Nested multipart forms are considered deprecated according to the living HTML5 standard and RFC 7578, Section 4.3.

If your app needs to handle nested forms, this can be done in the same fashion as any other part embedded in the form – by installing an appropriate media handler.

Let us extend the multipart form parser *media handlers* to recursively parse embedded forms of the `multipart/mixed` content type:

```
import falcon
import falcon.media

parser = falcon.media.MultipartFormHandler()
parser.parse_options.media_handlers['multipart/mixed'] = (
    falcon.media.MultipartFormHandler()
)
```

> **Note**
>
> Here we create a new parser (with default options) for nested parts, effectively disallowing further recursion.
>
> If traversing into even deeper multipart form hierarchy is desired, we can just reuse the same parser.

Let us now use the nesting-aware parser in an app:

```
import falcon
import falcon.media


class Forms:
    def on_post(self, req, resp):
        example = {}
        for part in req.media:
            if part.content_type.startswith('multipart/mixed'):
                for nested in part.media:
                    example[nested.filename] = nested.text

        resp.media = example
```

(continues on next page)

(continued from previous page)

```
parser = falcon.media.MultipartFormHandler()
parser.parse_options.media_handlers['multipart/mixed'] = (
    falcon.media.MultipartFormHandler()
)

app = falcon.App()
app.req_options.media_handlers[falcon.MEDIA_MULTIPART] = parser
app.add_route('/forms', Forms())
```

We should now be able to consume a form containing a nested `multipart/mixed` part (the example is adapted from the now-obsolete RFC 1867):

```
--AaB03x
Content-Disposition: form-data; name="field1"

Joe Blow
--AaB03x
Content-Disposition: form-data; name="docs"
Content-Type: multipart/mixed; boundary=BbC04y

--BbC04y
Content-Disposition: attachment; filename="file1.txt"

This is file1.

--BbC04y
Content-Disposition: attachment; filename="file2.txt"

Hello, World!

--BbC04y--

--AaB03x--
```

Note that all line endings in the form above are assumed to be CRLF.

The form should be `POST`ed with the `Content-Type` header set to `multipart/form-data; boundary=AaB03x`.

Outputting CSV Files

Generating a CSV (or PDF, etc.) report and making it available as a downloadable file is a fairly common back-end service task.

The easiest approach is to simply write CSV rows to an `io.StringIO` stream, and then assign its value to *`resp.text`*:

WSGI

```
import csv
import io

import falcon


class Report:
    def on_get(self, req, resp):
        output = io.StringIO()
```

(continues on next page)

(continued from previous page)

```
        writer = csv.writer(output, quoting=csv.QUOTE_NONNUMERIC)
        writer.writerow(('fruit', 'quantity'))
        writer.writerow(('apples', 13))
        writer.writerow(('oranges', 37))

        resp.content_type = falcon.MEDIA_CSV
        resp.downloadable_as = 'report.csv'
        resp.text = output.getvalue()
```

ASGI

```
import csv
import io

import falcon


class Report:
    async def on_get(self, req, resp):
        output = io.StringIO()
        writer = csv.writer(output, quoting=csv.QUOTE_NONNUMERIC)
        writer.writerow(('fruit', 'quantity'))
        writer.writerow(('apples', 13))
        writer.writerow(('oranges', 37))

        resp.content_type = falcon.MEDIA_CSV
        resp.downloadable_as = 'report.csv'
        resp.text = output.getvalue()
```

Here we set the response `Content-Type` to `MEDIA_CSV` as recommended by RFC 4180, and assign the downloadable file name `report.csv` via the `Content-Disposition` header (see also: *How can I serve a downloadable file with Falcon?*).

Streaming Large CSV Files on the Fly

If generated CSV responses are expected to be very large, it might be worth streaming the CSV data as it is produced. This approach will both avoid excessive memory consumption, and reduce the viewer's time-to-first-byte (TTFB).

In order to stream CSV rows on the fly, we will initialize the CSV writer with our own pseudo stream object. Our stream's `write()` method will simply accumulate the CSV data in a list. We will then set `resp.stream` to a generator yielding data chunks from this list:

WSGI

```
import csv

import falcon


class Report:
    class PseudoTextStream:
        def __init__(self):
            self.clear()

        def clear(self):
            self.result = []
```

(continues on next page)

(continued from previous page)

```
        def write(self, data):
            self.result.append(data.encode())

    def fibonacci_generator(self, n=1000):
        stream = self.PseudoTextStream()
        writer = csv.writer(stream, quoting=csv.QUOTE_NONNUMERIC)
        writer.writerow(('n', 'Fibonacci Fn'))

        previous = 1
        current = 0
        for i in range(n + 1):
            writer.writerow((i, current))
            previous, current = current, current + previous

            yield from stream.result
            stream.clear()

    def on_get(self, req, resp):
        resp.content_type = falcon.MEDIA_CSV
        resp.downloadable_as = 'report.csv'
        resp.stream = self.fibonacci_generator()
```

ASGI

```
import csv

import falcon


class Report:
    class PseudoTextStream:
        def __init__(self):
            self.clear()

        def clear(self):
            self.result = []

        def write(self, data):
            self.result.append(data.encode())

    def fibonacci_generator(self, n=1000):
        stream = self.PseudoTextStream()
        writer = csv.writer(stream, quoting=csv.QUOTE_NONNUMERIC)
        writer.writerow(('n', 'Fibonacci Fn'))

        previous = 1
        current = 0
        for i in range(n + 1):
            writer.writerow((i, current))
            previous, current = current, current + previous

            yield from stream.result
            stream.clear()

    def on_get(self, req, resp):
```

(continues on next page)

(continued from previous page)

```
        resp.content_type = falcon.MEDIA_CSV
        resp.downloadable_as = 'report.csv'
        resp.stream = self.fibonacci_generator()
```

> **Note**
>
> At the time of writing, Python does not support `yield from` here in an asynchronous generator, so we substitute it with a loop expression.

Handling Plain Text as Media

Normally, it is easiest to render a plain text response by setting *`resp.text`* (see also: *When would I use media, data, text, and stream?*). However, if plain text is just one of the *Internet media types* that your application speaks, it may be useful to generalize handling plain text as *media* too.

This recipe demonstrates how to create a *custom media handler* to process the `text/plain` media type. The handler implements serialization and deserialization of textual content, respecting the charset specified in the `Content-Type` header (or defaulting to `utf-8` when no charset is provided).

```
import functools

import falcon


class TextHandler(falcon.media.BaseHandler):
    DEFAULT_CHARSET = 'utf-8'

    @classmethod
    @functools.lru_cache
    def _get_charset(cls, content_type):
        _, params = falcon.parse_header(content_type)
        return params.get('charset') or cls.DEFAULT_CHARSET

    def deserialize(self, stream, content_type, content_length):
        data = stream.read()
        return data.decode(self._get_charset(content_type))

    def serialize(self, media, content_type):
        return media.encode(self._get_charset(content_type))
```

To use this handler, *register* it in the Falcon application's media options for both request and response:

```
import falcon
from your_module import TextHandler

app = falcon.App()
app.req_options.media_handlers['text/plain'] = TextHandler()
app.resp_options.media_handlers['text/plain'] = TextHandler()
```

With this setup, the application can handle textual data directly as `text/plain`, ensuring support for various character encodings as needed.

> **Warning**
>
> Be sure to validate and limit the size of incoming data when working with textual content to prevent server overload or denial-of-service attacks.

Prettifying JSON Responses

To make JSON responses more human-readable, it may be desirable to prettify the output. By default, Falcon's *JSONHandler* is configured to minimize serialization overhead. However, you can easily customize the output by simply providing the desired `dumps` parameters:

```
import functools
import json

from falcon import media

json_handler = media.JSONHandler(
    dumps=functools.partial(json.dumps, indent=4, sort_keys=True),
)
```

You can now replace the default `application/json` *response media handlers* with this customized `json_handler` to make your application's JSON responses prettier (see also: *Replacing the Default Handlers*).

Note

Another alternative for debugging is prettifying JSON on the client side, for example, the popular HTTPie does it by default. Another option is to simply pipe the JSON response into jq.

If your debugging case allows it, the client side approach should be preferred since it neither incurs performance overhead on the server side nor requires any customization effort.

Supporting optional indentation

Internet media type (content-type) negotiation is the canonical way to express resource representation preferences. Although not a part of the `application/json` media type standard, some frameworks (such as the Django REST Framework) and services support requesting a specific JSON indentation level using the `indent` content-type parameter. This recipe leaves the interpretation to the reader as to whether such a parameter adds "new functionality" as per RFC 6836, Section 4.3.

Assuming we want to add JSON `indent` support to a Falcon app, this can be implemented with a *custom media handler*:

```
import json

import falcon


class CustomJSONHandler(falcon.media.BaseHandler):
    MAX_INDENT_LEVEL = 8

    def deserialize(self, stream, content_type, content_length):
        data = stream.read()
        return json.loads(data.decode())

    def serialize(self, media, content_type):
        _, params = falcon.parse_header(content_type)
        indent = params.get('indent')
        if indent is not None:
            try:
                indent = int(indent)
                # NOTE: Impose a reasonable indentation level limit.
                if indent < 0 or indent > self.MAX_INDENT_LEVEL:
                    indent = None
```

(continues on next page)

(continued from previous page)

```
            except ValueError:
                # TODO: Handle invalid params?
                indent = None

        result = json.dumps(media, indent=indent, sort_keys=bool(indent))
        return result.encode()
```

Furthermore, we'll need to implement content-type negotiation to accept the indented JSON content type for response serialization. The bare-minimum example uses a middleware component as described here: *Content-Type Negotiation*.

After installing this handler for `application/json` response media, as well as adding the negotiation middleware, we should be able to produce indented JSON output (building upon the frontpage `QuoteResource` example):

```
$ curl -H 'Accept: application/json; indent=4' http://localhost:8000/quote
{
    "author": "Grace Hopper",
    "quote": "I've always been more interested in the future than in the past."
}
```

Warning

Implementing this in a public API available to untrusted, unauthenticated clients could be viewed as an unnecessary attack vector.

In the case of a denial-of-service attack, you would be providing the attacker with a convenient way to increase CPU load by simply asking to indent the output, particularly if large JSON responses are available.

Furthermore, replaying exactly the same requests with and without indentation may reveal information that is useful for timing attacks, especially if the attacker is able to guess the exact flavor of the JSON module used.

Decoding Raw URL Path

This recipe demonstrates how to access the "raw" request path using non-standard (WSGI) or optional (ASGI) application server extensions. This is useful when, for instance, a URI field has been percent-encoded in order to distinguish between forward slashes inside the field's value, and slashes used to separate fields. See also: *Why is my URL with percent-encoded forward slashes (%2F) routed incorrectly?*

WSGI

In the WSGI flavor of the framework, *req.path* is based on the `PATH_INFO` CGI variable, which is already presented percent-decoded. Some application servers expose the raw URL under another, non-standard, CGI variable name. Let us implement a middleware component that understands two such extensions, `RAW_URI` (Gunicorn, Werkzeug's dev server) and `REQUEST_URI` (uWSGI, Waitress, Werkzeug's dev server), and replaces `req.path` with a value extracted from the raw URL:

```
import falcon
import falcon.uri


class RawPathComponent:
    def process_request(self, req, resp):
        raw_uri = req.env.get('RAW_URI') or req.env.get('REQUEST_URI')

        # NOTE: Reconstruct the percent-encoded path from the raw URI.
        if raw_uri:
            req.path, _, _ = raw_uri.partition('?')
```

(continues on next page)

(continued from previous page)

```
class URLResource:
    def on_get(self, req, resp, url):
        # NOTE: url here is potentially percent-encoded.
        url = falcon.uri.decode(url)

        resp.media = {'url': url}

    def on_get_status(self, req, resp, url):
        # NOTE: url here is potentially percent-encoded.
        url = falcon.uri.decode(url)

        resp.media = {'cached': True}


app = falcon.App(middleware=[RawPathComponent()])
app.add_route('/cache/{url}', URLResource())
app.add_route('/cache/{url}/status', URLResource(), suffix='status')
```

Running the above app with a supported server such as Gunicorn or uWSGI, the following response is rendered to a `GET /cache/http%3A%2F%2Ffalconframework.org` request:

```
{
    "url": "http://falconframework.org"
}
```

We can also check the status of this URI in our imaginary web caching system by accessing `/cache/http%3A%2F%2Ffalconframework.org/status`:

```
{
    "cached": true
}
```

If we removed `RawPathComponent()` from the app's middleware list, the request would be routed as `/cache/http://falconframework.org`, and no matching resource would be found:

```
{
    "title": "404 Not Found"
}
```

What is more, even if we could implement a flexible router that was capable of matching these complex URI patterns, the app would still not be able to distinguish between `/cache/http%3A%2F%2Ffalconframework.org%2Fstatus` and `/cache/http%3A%2F%2Ffalconframework.org/status` if both were presented only in the percent-decoded form.

ASGI

The ASGI version of *req.path* uses the `path` key from the ASGI scope, where percent-encoded sequences are already decoded into characters just like in WSGI's `PATH_INFO`. Similar to the WSGI snippet from the previous chapter, let us create a middleware component that replaces `req.path` with the value of `raw_path` (provided the latter is present in the ASGI HTTP scope):

```
import falcon.asgi
import falcon.uri
```

(continues on next page)

(continued from previous page)

```
class RawPathComponent:
    async def process_request(self, req, resp):
        raw_path = req.scope.get('raw_path')

        # NOTE: Decode the raw path from the raw_path bytestring, disallowing
        #   non-ASCII characters, assuming they are correctly percent-coded.
        if raw_path:
            req.path = raw_path.decode('ascii')


class URLResource:
    async def on_get(self, req, resp, url):
        # NOTE: url here is potentially percent-encoded.
        url = falcon.uri.decode(url)

        resp.media = {'url': url}

    async def on_get_status(self, req, resp, url):
        # NOTE: url here is potentially percent-encoded.
        url = falcon.uri.decode(url)

        resp.media = {'cached': True}


app = falcon.asgi.App(middleware=[RawPathComponent()])
app.add_route('/cache/{url}', URLResource())
app.add_route('/cache/{url}/status', URLResource(), suffix='status')
```

Running the above snippet with `uvicorn` (that supports `raw_path`), the percent-encoded `url` field is now correctly handled for a `GET /cache/http%3A%2F%2Ffalconframework.org%2Fstatus` request:

```
{
    "url": "http://falconframework.org/status"
}
```

Again, as in the WSGI version, removing `RawPathComponent()` no longer lets the app route the above request as intended:

```
{
    "title": "404 Not Found"
}
```

Request ID Logging

When things go wrong, it's important to be able to identify all relevant log messages for a particular request. This is commonly done by generating a unique ID for each request and then adding that ID to every log entry.

If you wish to trace each request throughout your application, including from within components that are deeply nested or otherwise live outside of the normal request context, you can use a `ContextVar` object to store the request ID:

Listing 1: *context.py*

```
import contextvars


class _Context:
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        self._request_id_var = contextvars.ContextVar('request_id', default=None)

    @property
    def request_id(self):
        return self._request_id_var.get()

    @request_id.setter
    def request_id(self, value):
        self._request_id_var.set(value)


ctx = _Context()
```

Then, you can create a *middleware* class to generate a unique ID for each request, persisting it in the *contextvars* object:

Listing 2: *middleware.py*

```
from uuid import uuid4

# Import the above context.py
from my_app.context import ctx


class RequestIDMiddleware:
    def process_request(self, req, resp):
        request_id = str(uuid4())
        ctx.request_id = request_id

    # It may also be helpful to include the ID in the response
    def process_response(self, req, resp, resource, req_succeeded):
        resp.set_header('X-Request-ID', ctx.request_id)
```

Alternatively, if all of your application logic has access to the *request*, you can simply use the `req.context` object to store the ID:

Listing 3: *middleware.py*

```
from uuid import uuid4

# Optional logging package (pip install structlog)
import structlog


class RequestIDMiddleware:
    def process_request(self, req, resp):
        request_id = str(uuid4())

        # Using Falcon 2.0+ context style
        req.context.request_id = request_id

        # Or if your logger has built-in support for contexts
        req.context.log = structlog.get_logger(request_id=request_id)

    # It may also be helpful to include the ID in the response
    def process_response(self, req, resp, resource, req_succeeded):
```

(continues on next page)

(continued from previous page)

```
        resp.set_header('X-Request-ID', req.context.request_id)
```

Note

If your app is deployed behind a reverse proxy that injects a request ID header, you can easily adapt this recipe to use the upstream ID rather than generating a new one. By doing so, you can provide traceability across the entire request path.

With this in mind, you may also wish to include this ID in any requests to downstream services.

Once you have access to a request ID, you can include it in your logs by subclassing `logging.Formatter` and overriding the `format()` method, or by using a third-party logging library such as structlog as demonstrated above.

In a pinch, you can also output the request ID directly:

Listing 4: *some_other_module.py*

```
import logging

# Import the above context.py
from my_app.context import ctx


def create_widget_object(name: str):
    request_id = f'request_id={ctx.request_id}'
    logging.debug('%s going to create widget: %s', request_id, name)

    try:
        # create the widget
        pass
    except Exception:
        logging.exception('%s something went wrong', request_id)

    logging.debug('%s created widget: %s', request_id, name)
```

5.1.8 FAQ

Design Philosophy

Why doesn't Falcon come with batteries included?

Falcon is designed for applications that require a high level of customization or performance tuning. The framework's minimalist design frees the developer to select the best strategies and 3rd-party packages for the task at hand.

The Python ecosystem offers a number of great packages that you can use from within your responders, hooks, and middleware components. As a starting point, the community maintains a list of Falcon add-ons and complementary packages.

Why doesn't Falcon create a new Resource instance for every request?

Falcon generally tries to minimize the number of objects that it instantiates. It does this for two reasons: first, to avoid the expense of creating the object, and second to reduce memory usage by reducing the total number of objects required under highly concurrent workloads. Therefore, when adding a route, Falcon requires an *instance* of your resource class, rather than the class type. That same instance will be used to serve all requests coming in on that route.

What happens if my responder raises an error?

Generally speaking, Falcon assumes that resource responders (such as `on_get()`, `on_post()`, etc.) will, for the most part, do the right thing. In other words, Falcon doesn't try very hard to protect responder code from itself.

> **Note**
>
> As of version 3.0, the framework will no longer propagate uncaught exceptions to the application server. Instead, the default `Exception` handler will return an HTTP 500 response and log details of the exception to `wsgi.errors`.

Although providing basic error handlers, Falcon optimizes for the most common case where resource responders do not raise any errors for valid requests. With that in mind, writing a high-quality API based on Falcon requires that:

1. Resource responders set response variables to sane values.
2. Your code is well-tested, with high code coverage.
3. Errors are anticipated, detected, and handled appropriately within each responder and with the aid of custom error handlers.

How do I generate API documentation for my Falcon API?

When it comes to API documentation, some developers prefer to use the API implementation as the user contract or source of truth (taking an implementation-first approach), while other developers prefer to use the API spec itself as the contract, implementing and testing the API against that spec (taking a design-first approach).

At the risk of erring on the side of flexibility, Falcon does not provide API spec support out of the box. However, there are several community projects available in this vein. Our Add on Catalog lists a couple of these projects, but you may also wish to search PyPI for additional packages.

If you are interested in the design-first approach mentioned above, you may also want to check out API design and gateway services such as Tyk, Apiary, Amazon API Gateway, or Google Cloud Endpoints.

Performance

Does Falcon work with HTTP/2?

Falcon is a WSGI framework and as such does not serve HTTP requests directly. However, you can get most of the benefits of HTTP/2 by simply deploying any HTTP/2-compliant web server or load balancer in front of your app to translate between HTTP/2 and HTTP/1.1. Eventually we expect that Python web servers (such as uWSGI) will support HTTP/2 natively, eliminating the need for a translation layer.

Is Falcon thread-safe?

The Falcon framework is, itself, thread-safe. For example, new *`Request`* and *`Response`* objects are created for each incoming HTTP request. However, a single instance of each resource class attached to a route is shared among all requests. Middleware objects and other types of hooks, such as custom error handlers, are likewise shared. Therefore, as long as you implement these classes and callables in a thread-safe manner, and ensure that any third-party libraries used by your app are also thread-safe, your WSGI app as a whole will be thread-safe.

Can I run Falcon on free-threaded CPython?

Starting with Falcon *4.2*, we began to ship binary wheels for the free-threaded CPython 3.14 build (in the first iteration, only on selected Linux x86 and ARM platforms).

We load-tested the WSGI flavor of the framework using Gunicorn 23.0.0 on free-threaded CPython 3.14.0, and observed no issues that would point toward Falcon's reliance on the GIL. Moreover, the throughput scaled almost linearly with the number of threads (up to the number of available cores on the system).

Thus, we would like to think that Falcon is still *thread-safe* even in free-threaded execution, but we are awaiting more community feedback. If you experimented with free-threading of Falcon applications, or even deployed it in production, please *share your experience*!

Does Falcon support asyncio?

Starting with version 3.0, the ASGI flavor of Falcon now proudly supports `asyncio`! Use the *`falcon.asgi.App`* class to create an async application, and serve it via an *ASGI application server* such as Uvicorn.

Alternatively, IO-bound WSGI applications can be scaled using the battle-tested gevent library via Gunicorn or uWSGI. meinheld has also been used successfully by the community to power high-throughput, low-latency WSGI services.

> **Tip**
>
> Note that if you use Gunicorn, you can combine gevent and PyPy to achieve an impressive level of performance. (Unfortunately, uWSGI does not yet support using gevent and PyPy together.)

Does Falcon support WebSocket?

The async flavor of Falcon supports the ASGI WebSocket protocol. See also: *WebSocket (ASGI Only)*.

WSGI applications might try leveraging uWSGI's native WebSocket support or gevent-websocket's `GeventWebSocketWorker` for Gunicorn.

As an option, it may make sense to design WebSocket support as a separate service due to very different performance characteristics and interaction patterns, compared to a regular RESTful API. In addition to (obviously!) Falcon's native ASGI support, a standalone WebSocket service could also be implemented via Aymeric Augustin's handy websockets library.

Routing

How do I implement CORS with Falcon?

In order for a website or SPA to access an API hosted under a different domain name, that API must implement Cross-Origin Resource Sharing (CORS). For a public API, implementing CORS in Falcon can be as simple as passing the `cors_enable` flag (set to `True`) when instantiating *your application*.

Further CORS customization is possible via *`CORSMiddleware`* (for more information on managing CORS in Falcon, see also *CORS*).

For even more sophisticated use cases, have a look at Falcon add-ons from the community, such as falcon-cors, or try one of the generic WSGI CORS libraries available on PyPI. If you use an API gateway, you might also look into what CORS functionality it provides at that level.

Why is my request with authorization blocked despite `cors_enable`?

When you are making a cross-origin request from the browser (or another HTTP client verifying CORS policy), and the request is authenticated using the Authorization header, the browser adds `authorization` to Access-Control-Request-Headers in the preflight (`OPTIONS`) request, however, the actual authorization credentials are omitted at this stage.

If your request authentication/authorization is performed in a *middleware* component which rejects requests lacking authorization credentials by raising an instance of *`HTTPUnauthorized`* (or rendering a 4XX response in another way), a common pitfall is that even an `OPTIONS` request (which is lacking authorization as per the above explanation) yields an error in this manner. As a result of the failed preflight, the browser chooses not proceed with the main request.

If you have implemented the authorization middleware yourself, you can simply let `OPTIONS` pass through:

```
class MyAuthMiddleware:
    def process_request(self, req, resp):
        # NOTE: Do not authenticate OPTIONS requests.
        if req.method == 'OPTIONS':
            return

        # -- snip --

        # My authorization logic...
```

Alternatively, if the middleware comes from a third-party library, it may be more practical to subclass it:

```
class CORSAwareMiddleware(SomeAuthMiddleware):
    def process_request(self, req, resp):
        # NOTE: Do not authenticate OPTIONS requests.
        if req.method != 'OPTIONS':
            super().process_request(req, resp)
```

In the case middleware in question instead hooks into `process_resource()`, you can use a similar treatment.

If you tried the above, and you still suspect the problem lies within Falcon's *CORS middleware*, it might be a bug! *Let us know* so we can help.

How do I implement redirects within Falcon?

Falcon provides a number of exception classes that can be raised to redirect the client to a different location (see also *Redirection*).

Note, however, that it is more efficient to handle permanent redirects directly with your web server, if possible, rather than placing additional load on your app for such requests.

How do I split requests between my original app and the part I migrated to Falcon?

It is common to carve out a portion of an app and reimplement it in Falcon to boost performance where it is most needed.

If you have access to your load balancer or reverse proxy configuration, we recommend setting up path or subdomain-based rules to split requests between your original implementation and the parts that have been migrated to Falcon (e.g., by adding an additional `location` directive to your NGINX config).

If the above approach isn't an option for your deployment, you can implement a simple WSGI wrapper that does the same thing:

```
def application(environ, start_response):
    try:
        # NOTE(kgriffs): Prefer the host header; the web server
        # isn't supposed to mess with it, so it should be what
        # the client actually sent.
        host = environ['HTTP_HOST']
    except KeyError:
        # NOTE(kgriffs): According to PEP-3333, this header
        # will always be present.
        host = environ['SERVER_NAME']

    if host.startswith('api.'):
        return falcon_app(environ, start_response)
    elif:
        return webapp2_app(environ, start_response)
```

See also PEP 3333 for a complete list of the variables that are provided via `environ`.

How do I implement both POSTing and GETing items for the same resource?

Suppose you have the following routes:

```
# Resource Collection
GET /resources{?marker, limit}
POST /resources

# Resource Item
GET /resources/{id}
PATCH /resources/{id}
DELETE /resources/{id}
```

You can implement this sort of API by simply using two Python classes, one to represent a single resource, and another to represent the collection of said resources. It is common to place both classes in the same module (see also *this section of the tutorial*.)

Alternatively, you can use suffixed responders to map both routes to the same resource class:

```
class MyResource:
    def on_get(self, req, resp, id):
        pass

    def on_patch(self, req, resp, id):
        pass

    def on_delete(self, req, resp, id):
        pass

    def on_get_collection(self, req, resp):
        pass

    def on_post_collection(self, req, resp):
        pass


# -- snip --


resource = MyResource()
app.add_route('/resources/{id}', resource)
app.add_route('/resources', resource, suffix='collection')
```

What is the recommended way to map related routes to resource classes?

Let's say we have the following URL schema:

```
GET  /game/ping
GET  /game/{game_id}
POST /game/{game_id}
GET  /game/{game_id}/state
POST /game/{game_id}/state
```

We can break this down into three resources:

```
Ping:

    GET  /game/ping
```

(continues on next page)

(continued from previous page)

```
Game:

    GET  /game/{game_id}
    POST /game/{game_id}

GameState:

    GET  /game/{game_id}/state
    POST /game/{game_id}/state
```

GameState may be thought of as a sub-resource of Game. It is a distinct logical entity encapsulated within a more general Game concept.

In Falcon, these resources would be implemented with standard classes:

```
class Ping:

    def on_get(self, req, resp):
        resp.text = '{"message": "pong"}'


class Game:

    def __init__(self, dao):
        self._dao = dao

    def on_get(self, req, resp, game_id):
        pass

    def on_post(self, req, resp, game_id):
        pass


class GameState:

    def __init__(self, dao):
        self._dao = dao

    def on_get(self, req, resp, game_id):
        pass

    def on_post(self, req, resp, game_id):
        pass


app = falcon.App()

# Game and GameState are closely related, and so it
# probably makes sense for them to share an object
# in the Data Access Layer. This could just as
# easily use a DB object or ORM layer.
#
# Note how the resources classes provide a layer
# of abstraction or indirection which makes your
# app more flexible since the data layer can
# evolve somewhat independently from the presentation
```

(continues on next page)

(continued from previous page)

```
# layer.
game_dao = myapp.DAL.Game(myconfig)

app.add_route('/game/ping', Ping())
app.add_route('/game/{game_id}', Game(game_dao))
app.add_route('/game/{game_id}/state', GameState(game_dao))
```

Alternatively, a single resource class could implement suffixed responders in order to handle all three routes:

```
class Game:

    def __init__(self, dao):
        self._dao = dao

    def on_get(self, req, resp, game_id):
        pass

    def on_post(self, req, resp, game_id):
        pass

    def on_get_state(self, req, resp, game_id):
        pass

    def on_post_state(self, req, resp, game_id):
        pass

    def on_get_ping(self, req, resp):
        resp.data = b'{"message": "pong"}'


# -- snip --


app = falcon.App()

game = Game(myapp.DAL.Game(myconfig))

app.add_route('/game/{game_id}', game)
app.add_route('/game/{game_id}/state', game, suffix='state')
app.add_route('/game/ping', game, suffix='ping')
```

Why is my URL with percent-encoded forward slashes (`%2F`) routed incorrectly?

This is an unfortunate artifact of the WSGI specification, which offers no standard means of accessing the "raw" request URL. According to PEP 3333, the recommended way to reconstruct a request's URL path is using the `PATH_INFO` CGI variable, which is already presented percent-decoded, effectively making originally percent-encoded forward slashes (`%2F`) indistinguishable from others passed verbatim (and intended to separate URI fields).

Although not standardized, some WSGI servers provide the raw URL as a non-standard extension; for instance, Gunicorn exposes it as `RAW_URI`, uWSGI calls it `REQUEST_URI`, etc. You can implement a WSGI (or ASGI, see the discussion below) middleware component to overwrite the request path with the path component of the raw URL, see more in the following recipe: *Decoding Raw URL Path*.

In contrast to WSGI, the ASGI specification does define a standard connection HTTP scope variable name (`raw_path`) for the unmodified HTTP path. However, it is not mandatory, and some applications servers may be unable to provide it. Nevertheless, we are exploring the possibility of adding an optional feature to use this raw path for routing in the ASGI flavor of the framework.

Extensibility

How do I use WSGI middleware with Falcon?

Instances of *falcon.App* are first-class WSGI apps, so you can use the standard pattern outlined in PEP-3333. In your main "app" file, you would simply wrap your api instance with a middleware app. For example:

```
import my_restful_service
import some_middleware

app = some_middleware.DoSomethingFancy(my_restful_service.app)
```

See also the WSGI middleware example given in PEP-3333.

How can I pass data from a hook to a responder, and between hooks?

You can inject extra responder kwargs from a hook by adding them to the *params* dict passed into the hook. You can also set custom attributes on the *req.context* object, as a way of passing contextual information around:

```
def authorize(req, resp, resource, params):
    # TODO: Check authentication/authorization

    # -- snip --

    req.context.role = 'root'
    req.context.scopes = ('storage', 'things')
    req.context.uid = 0

# -- snip --

@falcon.before(authorize)
def on_post(self, req, resp):
    pass
```

How can I write a custom handler for 404 and 500 pages in falcon?

When a route can not be found for an incoming request, Falcon uses a default responder that simply raises an instance of *HTTPRouteNotFound*, which the framework will in turn render as a 404 response. You can use *falcon.App.add_error_handler()* to override the default handler for this exception type (or for its parent type, *HTTPNotFound*). Alternatively, you may be able to configure your web server to transform the response for you (e.g., using nginx's `error_page` directive).

By default, non-system-exiting exceptions that do not inherit from *HTTPError* or *HTTPStatus* are handled by Falcon with a plain HTTP 500 error. To provide your own 500 logic, you can add a custom error handler for Python's base `Exception` type. This will not affect the default handlers for *HTTPError* and *HTTPStatus*.

See *Error Handling* and the *falcon.App.add_error_handler()* docs for more details.

Request Handling

How do I authenticate requests?

Hooks and middleware components can be used together to authenticate and authorize requests. For example, a middleware component could be used to parse incoming credentials and place the results in *req.context*. Downstream components or hooks could then use this information to authorize the request, taking into account the user's role and the requested resource.

Why does req.stream.read() hang for certain requests?

This behavior is an unfortunate artifact of the request body mechanics not being fully defined by the WSGI spec (PEP-3333). This is discussed in the reference documentation for *stream*, and a workaround is provided in the form of *bounded_stream*.

How does Falcon handle a trailing slash in the request path?

If your app sets *strip_url_path_trailing_slash* to `True`, Falcon will normalize incoming URI paths to simplify later processing and improve the predictability of application logic. This can be helpful when implementing a REST API schema that does not interpret a trailing slash character as referring to the name of an implicit sub-resource, as traditionally used by websites to reference index pages.

For example, with this option enabled, adding a route for `'/foo/bar'` implicitly adds a route for `'/foo/bar/'`. In other words, requests coming in for either path will be sent to the same resource.

Warning

If *strip_url_path_trailing_slash* is enabled, adding a route with a trailing slash will effectively make it unreachable from normal routing (theoretically, it may still be matched by rewriting the request path in middleware).

In this case, routes should be added without a trailing slash (obviously except the root path `'/'`), such as `'/foo/bar'` in the example above.

Note

Starting with version 2.0, the default for the *strip_url_path_trailing_slash* request option changed from `True` to `False`.

Why is my query parameter missing from the req object?

If a query param does not have a value and the *keep_blank_qs_values* request option is set to `False` (the default as of Falcon 2.0+ is `True`), Falcon will ignore that parameter. For example, passing `'foo'` or `'foo='` will result in the parameter being ignored.

If you would like to recognize such parameters, the *keep_blank_qs_values* request option should be set to `True` (or simply kept at its default value in Falcon 2.0+). Request options are set globally for each instance of *falcon.App* via the *req_options* property. For example:

```
app.req_options.keep_blank_qs_values = True
```

Why are '+' characters in my params being converted to spaces?

The + character is often used instead of `%20` to represent spaces in query string params, due to the historical conflation of form parameter encoding (`application/x-www-form-urlencoded`) and URI percent-encoding. Therefore, Falcon, converts + to a space when decoding strings.

To work around this, RFC 3986 specifies + as a reserved character, and recommends percent-encoding any such characters when their literal value is desired (`%2B` in the case of +).

How can I access POSTed form params?

By default, Falcon does not consume request bodies. However, a *media handler* for the `application/x-www-form-urlencoded` content type is installed by default, thus making the POSTed form available as *Request.media* with zero configuration:

```
import falcon


class MyResource:
    def on_post(self, req, resp):
        # TODO: Handle the submitted URL-encoded form
        form = req.get_media()

        # NOTE: Falcon chooses the right media handler automatically, but
        #   if we wanted to differentiate from, for instance, JSON, we
        #   could check whether req.content_type == falcon.MEDIA_URLENCODED
        #   or use mimeparse to implement more sophisticated logic.
```

Note

In prior versions of Falcon, a POSTed URL-encoded form could be automatically consumed and merged into *params* by setting the *auto_parse_form_urlencoded* option to `True`. This behavior is still supported in the Falcon 4.x series. However, it has been deprecated in favor of *URLEncodedFormHandler*, and the option to merge URL-encoded form data into *params* will be removed in the next major release (Falcon 5.0).

POSTed form parameters may also be read directly from *stream* and parsed via *falcon.uri.parse_query_string()* or `urllib.parse.parse_qs()`.

How can I access POSTed files?

If files are `POST`ed as part of a *multipart form*, the default *MultipartFormHandler* can be used to efficiently parse the submitted `multipart/form-data` *request media* by iterating over the multipart *body parts*:

```
for part in req.get_media():
    # TODO: Do something with the body part
    pass
```

How can I save POSTed files (from a multipart form) directly to AWS S3?

As highlighted in the previous answer dealing with *files posted as multipart form*, *falcon.media.MultipartFormHandler* may be used to iterate over the uploaded multipart body parts.

The *stream* of a body part is a file-like object implementing the `read()` method, making it compatible with `boto3`'s upload_fileobj:

WSGI

```
import boto3

# -- snip --

s3 = boto3.client('s3')

for part in req.get_media():
    if part.name == 'myfile':
        s3.upload_fileobj(part.stream, 'mybucket', 'mykey')
```

ASGI

```
import aioboto3

# -- snip --

session = aioboto3.Session()

form = await req.get_media()
async for part in form:
    if part.name == 'myfile':
        async with session.client('s3') as s3:
            await s3.upload_fileobj(part.stream, 'mybucket', 'mykey')
```

Note

The ASGI snippet requires the aioboto3 async wrapper in lieu of `boto3` (as the latter only offers a synchronous interface at the time of writing).

Note

Falcon is not endorsing any particular cloud service provider, and AWS S3 and `boto3` are referenced here just as a popular example. The same pattern can be applied to any storage API that supports streaming directly from a file-like object.

How do I parse a nested multipart form?

Falcon does not offer official support for parsing nested multipart forms (i.e., where multiple files for a single field are transmitted using a nested `multipart/mixed` part) at this time. The usage is considered deprecated according to the living HTML5 standard and RFC 7578, Section 4.3.

Tip

If your app absolutely must deal with such legacy forms, the parser may actually be capable of the task. See more in this recipe: *Parsing Nested Multipart Forms*.

How do I retrieve a JSON value from the query string?

To retrieve a JSON-encoded value from the query string, Falcon provides the *`get_param_as_json()`* method, an example of which is given below:

```
import falcon


class LocationResource:

    def on_get(self, req, resp):
        places = {
            'Chandigarh, India': {
                'lat': 30.692781,
                'long': 76.740875
            },
```

(continues on next page)

(continued from previous page)

```
            'Ontario, Canada': {
                'lat': 43.539814,
                'long': -80.246094
            }
        }

        coordinates = req.get_param_as_json('place')

        place = None
        for (key, value) in places.items():
            if coordinates == value:
                place = key
                break

        resp.media = {
            'place': place
        }


app = falcon.App()
app.add_route('/locations', LocationResource())
```

In the example above, `LocationResource` expects a query string containing a JSON-encoded value named `'place'`. This value can be fetched and decoded from JSON in a single step with the *`get_param_as_json()`* method. Given a request URL such as:

```
/locations?place={"lat":43.539814,"long":-80.246094}
```

The *coordinates* variable will be set to a `dict` as expected.

By default, the *`auto_parse_qs_csv`* option is set to `False`. The example above assumes this default.

On the other hand, when *`auto_parse_qs_csv`* is set to `True`, Falcon treats commas in a query string as literal characters delimiting a comma-separated list. For example, given the query string `?c=1,2,3`, Falcon will add this to your `request.params` dictionary as `{'c': ['1', '2', '3']}`. If you attempt to use JSON in the value of the query string, for example `?c={"a":1,"b":2}`, the value will be added to `request.params` in an unexpected way: `{'c': ['{"a":1', '"b":2}']}`.

Commas are a reserved character that can be escaped according to RFC 3986 - 2.2. Reserved Characters, so one possible solution is to percent encode any commas that appear in your JSON query string.

The other option is to leave *`auto_parse_qs_csv`* disabled and simply use JSON array syntax in lieu of CSV.

When *`auto_parse_qs_csv`* is not enabled, the value of the query string `?c={"a":1,"b":2}` will be added to the `req.params` dictionary as `{'c': '{"a":1,"b":2}'}`. This lets you consume JSON whether or not the client chooses to percent-encode commas in the request. In this case, you can retrieve the raw JSON string via *`get_param()`*, or use the *`get_param_as_json()`* convenience method as demonstrated above.

Can I use msgspec with Falcon?

msgspec is a fast serialization and validation library, with built-in support for JSON, MessagePack, YAML, and TOML.

You can use `msgspec` as *JSON handler* out of the box. Its awesome performance aside, you will also be able to assign instances of `msgspec.Struct` to *`resp.media`* without any additional configuration.

It is also fairly straightforward to set up request media validation and error handling; see more in the following recipe: *msgspec integration*.

How can I handle forward slashes within a route template field?

Falcon 4.0 shipped initial support for field converters that can match multiple segments. The `path` field converter is capable of consuming multiple path segments when placed at the end of the URL template.

In previous versions, you can work around the issue by implementing a Falcon middleware component to rewrite the path before it is routed. If you control the clients, you can percent-encode forward slashes inside the field in question, however, note that pre-processing is unavoidable in order to access the raw encoded URI too. See also: *Why is my URL with percent-encoded forward slashes (%2F) routed incorrectly?*

How do I adapt my code to default context type changes in Falcon 2.0?

The default request/response context type has been changed from dict to a bare class in Falcon 2.0. Instead of setting dictionary items, you can now simply set attributes on the object:

```
# Before Falcon 2.0
req.context['cache_backend'] = MyUltraFastCache.connect()

# Falcon 2.0
req.context.cache_backend = MyUltraFastCache.connect()
```

The new *default context type* emulates a dict-like mapping interface in a way that context attributes are linked to dict items, i.e. setting an object attribute also sets the corresponding dict item, and vice versa. As a result, existing code will largely work unmodified with Falcon 2.0+. Nevertheless, it is recommended to migrate to the new interface as setting attributes is more performant than inserting keys via the emulated mapping.

> **Warning**
>
> If you need to mix-and-match both approaches, beware that setting attributes such as *items* or *values* would obviously shadow the corresponding mapping interface functions.

If an existing project makes extensive use of dictionary contexts, the type can be explicitly overridden back to `dict` by employing custom request/response types:

```
class RequestWithDictContext(falcon.Request):
    context_type = dict

class ResponseWithDictContext(falcon.Response):
    context_type = dict

# -- snip --

app = falcon.App(request_type=RequestWithDictContext,
                 response_type=ResponseWithDictContext)
```

> **Attention**
>
> Note that third-party middleware might expect to be able to set attributes on `req.context` (or `resp.context`), following the new style.

Response Handling

When would I use media, data, text, and stream?

These four attributes are mutually exclusive, you should only set one when defining your response.

resp.media is used when you want to use the Falcon serialization mechanism. Just assign data to the attribute and Falcon will take care of the rest.

```
class MyResource:
    def on_get(self, req, resp):
        resp.media = {'hello': 'World'}
```

resp.text and *resp.data* are very similar, they both allow you to set the body of the response. The difference being, *text* takes a string, and *data* takes bytes.

```
class MyResource:
    def on_get(self, req, resp):
        resp.text = json.dumps({'hello': 'World'})

    def on_post(self, req, resp):
        resp.data = b'{"hello": "World"}'
```

resp.stream allows you to set a generator that yields bytes, or a file-like object with a `read()` method that returns bytes. In the case of a file-like object, the framework will call `read()` until the stream is exhausted.

```
class MyResource:
    def on_get(self, req, resp):
        resp.stream = open('myfile.json', mode='rb')
```

See also the *Outputting CSV Files* recipe for an example of using *resp.stream* with a generator.

How can I use resp.media with types like datetime?

The default JSON handler for *resp.media* only supports the objects and types listed in the table documented under `json.JSONEncoder`.

To handle additional types in JSON, you can either serialize them beforehand, or create a custom JSON media handler that sets the *default* param for `json.dumps()`. When deserializing an incoming request body, you may also wish to implement *object_hook* for `json.loads()`. Note, however, that setting the *default* or *object_hook* params can negatively impact the performance of (de)serialization.

If you use an alternative JSON library, you might also look whether it provides support for additional data types. For instance, the popular `orjson` and *msgspec* libraries opt to automatically serialize `dataclasses`, `enums`, `datetime` objects, etc.

Furthermore, different Internet media types such as YAML, *msgpack*, etc might support more data types than JSON, either as part of the respective (de)serialization format, or via custom type extensions.

See also

See *Custom JSON encoder* for an example on how to use a custom json encoder.

Note

When testing an application employing a custom JSON encoder, bear in mind that *TestClient* is decoupled from the app, and it simulates requests as if they were performed by a third-party client (just sans network). Therefore, passing the **json** parameter to *simulate_** methods will effectively use the stdlib's `json.dumps()`. If you want to serialize custom objects for testing, you will need to dump them into a string yourself, and pass it using the **body** parameter instead (accompanied by the `application/json` content type header).

Does Falcon set Content-Length or do I need to do that explicitly?

Falcon will try to do this for you, based on the value of *resp.text*, *resp.data* or *resp.media* (whichever is set in the response, checked in that order).

For dynamically-generated content, you can choose to not set *content_length*, in which case Falcon will then leave off the Content-Length header, and hopefully your WSGI server will do the Right Thing™ (assuming you've told the server to enable keep-alive, it may choose to use chunked encoding).

> **Note**
>
> PEP-3333 prohibits apps from setting hop-by-hop headers itself, such as Transfer-Encoding.

Similar to WSGI, the ASGI HTTP connection scope specification states that responses without Content-Length "may be chunked as the server sees fit".

Why is an empty response body returned when I raise an instance of HTTPError?

Falcon attempts to serialize the *HTTPError* instance using its *to_json()* or *to_xml()* methods, according to the Accept header in the request. If neither JSON nor XML is acceptable, no response body will be generated. You can override this behavior if needed via *set_error_serializer()*.

I'm setting a response body, but it isn't getting returned. What's going on?

Falcon skips processing the response body when, according to the HTTP spec, no body should be returned. If the client sends a HEAD request, the framework will always return an empty body. Falcon will also return an empty body whenever the response status is any of the following:

```
falcon.HTTP_100
falcon.HTTP_204
falcon.HTTP_416
falcon.HTTP_304
```

If you have another case where the body isn't being returned, it's probably a bug! *Let us know* so we can help.

I'm setting a cookie, but it isn't being returned in subsequent requests.

By default, Falcon enables the *secure* cookie attribute. Therefore, if you are testing your app over HTTP (instead of HTTPS), the client will not send the cookie in subsequent requests.

(See also the *cookie documentation*.)

How can I serve a downloadable file with Falcon?

In the `on_get()` responder method for the resource, you can tell the user agent to download the file by setting the Content-Disposition header. Falcon includes the *downloadable_as* property to make this easy:

```
resp.downloadable_as = 'report.pdf'
```

See also the *Outputting CSV Files* recipe for a more involved example of dynamically generated downloadable content.

Why is Falcon changing my header names to lowercase?

Falcon always lowercases header names before storing them in the internal *Response* structures in order to make the response header handling straightforward and performant, as header name lookup can be done using a simple `dict`. Since HTTP headers are case insensitive, this optimization should normally not affect your API consumers.

In the unlikely case you absolutely must deal with non-conformant HTTP clients expecting a specific header name capitalization, see this recipe how to override header names using generic WSGI middleware: *Capitalizing Response Header Names*.

Note that this question only applies to the WSGI flavor of Falcon. The ASGI HTTP scope specification requires HTTP header names to be lowercased.

Furthermore, the HTTP2 standard also mandates that header field names MUST be converted to lowercase (see RFC 7540, Section 8.1.2).

Can Falcon serve static files?

Falcon makes it easy to efficiently serve static files by simply assigning an open file to `resp.stream` *as demonstrated in the tutorial*. You can also serve an entire directory of files via *`falcon.App.add_static_route()`*. However, if possible, it is best to serve static files directly from a web server like Nginx, or from a CDN.

Misc.

How do I manage my database connections?

Assuming your database library manages its own connection pool, all you need to do is initialize the client and pass an instance of it into your resource classes. For example, using SQLAlchemy Core:

```
engine = create_engine('sqlite:///:memory:')
resource = SomeResource(engine)
```

Then, within `SomeResource`:

```
# Read from the DB
with self._engine.connect() as connection:
    result = connection.execute(some_table.select())
for row in result:
    # TODO: Do something with each row

result.close()

# -- snip --

# Write to the DB within a transaction
with self._engine.begin() as connection:
    r1 = connection.execute(some_table.select())

    # -- snip --

    connection.execute(
        some_table.insert(),
        col1=7,
        col2='this is some data'
    )
```

When using a data access layer, simply pass the engine into your data access objects instead. See also this sample Falcon project that demonstrates using an ORM with Falcon.

You can also create a middleware component to automatically check out database connections for each request, but this can make it harder to track down errors, or to tune for the needs of individual requests.

If you need to transparently handle reconnecting after an error, or for other use cases that may not be supported by your client library, simply encapsulate the client library within a management class that handles all the tricky bits, and pass that around instead.

If you are interested in the middleware approach, the falcon-sqla library can be used to automatically check out and close SQLAlchemy connections that way (although it also supports the explicit context manager pattern).

How do I manage my database connections with ASGI?

This example is similar to the above one, but it uses ASGI lifecycle hooks to set up a connection pool, and to dispose it at the end of the application. The example uses psycopg to connect to a PostgreSQL database, but a similar pattern may be adapted to other asynchronous database libraries.

```
import psycopg_pool

url = 'postgresql://scott:tiger@127.0.0.1:5432/test'

class AsyncPoolMiddleware:
    def __init__(self):
        self._pool = None

    async def process_startup(self, scope, event):
        self._pool = psycopg_pool.AsyncConnectionPool(url)
        await self._pool.wait()  # created the pooled connections

    async def process_shutdown(self, scope, event):
        if self._pool:
            await self._pool.close()

    async def process_request(self, req, resp):
        req.context.pool = self._pool

        try:
            req.context.conn = await self._pool.getconn()
        except Exception:
            req.context.conn = None
            raise

    async def process_response(self, req, resp, resource, req_succeeded):
        if req.context.conn:
            await self._pool.putconn(req.context.conn)
```

Then, an example resource may use the connection or the pool:

```
class Numbers:
    async def on_get(self, req, resp):
        # This endpoint uses the connection created for the request by the␣
↪Middleware
        async with req.context.conn.cursor() as cur:
            await cur.execute('SELECT value FROM numbers')
            rows = await cur.fetchall()

        resp.media = [row[0] for row in rows]

    async def on_get_with_pool(self, req, resp):
        # This endpoint uses the pool to acquire a connection
        async with req.context.pool.connection() as conn:
            cur = await conn.execute('SELECT value FROM numbers')
            rows = await cur.fetchall()
            await cur.close()

        resp.media = [row[0] for row in rows]
```

The application can then be used as

```
from falcon.asgi import App

app = App(middleware=[AsyncPoolMiddleware()])
num = Numbers()
app.add_route('/conn', num)
app.add_route('/pool', num, suffix='with_pool')
```

What is the recommended approach for app configuration?

When it comes to app configuration, Falcon is not opinionated. You are free to choose from any of the excellent general-purpose configuration libraries maintained by the Python community. It's pretty much up to you if you want to use the standard library or something like `aumbry` as demonstrated by this Falcon example app.

(See also the **Configuration** section of our Complementary Packages wiki page. You may also wish to search PyPI for other options).

After choosing a configuration library, the only remaining question is how to access configuration options throughout your app.

People usually fall into two camps when it comes to this question. The first camp likes to instantiate a config object and pass that around to the initializers of the resource classes so the data sharing is explicit. The second camp likes to create a config module and import that wherever it's needed.

With the latter approach, to control when the config is actually loaded, it's best not to instantiate it at the top level of the config module's namespace. This avoids any problematic side-effects that may be caused by loading the config whenever Python happens to process the first import of the config module. Instead, consider implementing a function in the module that returns a new or cached config object on demand.

How do I test my Falcon app? Can I use pytest?

Falcon's testing framework supports both `unittest` and `pytest`. In fact, the tutorial in the docs provides an excellent introduction to testing Falcon apps with pytest.

(See also: Testing)

Can I shut my server down cleanly from the app?

Normally, the lifetime of an app server is controlled by other means than from inside the running app, and there is no standardized way for a WSGI or ASGI framework to shut down the server programmatically.

However, if you need to spin up a real server for testing purposes (such as for collecting coverage while interacting with other services over the network), your app server of choice may offer a Python API or hooks that you can integrate into your app.

For instance, the stdlib's `wsgiref` server inherits from `TCPServer`, which can be stopped by calling its `shutdown()` method. Just make sure to perform the call from a different thread (otherwise it may deadlock):

```
import http
import threading
import wsgiref.simple_server

import falcon


class Shutdown:
    def __init__(self, httpd):
        self._httpd = httpd

    def on_post(self, req, resp):
        thread = threading.Thread(target=self._httpd.shutdown, daemon=True)
```

(continues on next page)

(continued from previous page)

```
        thread.start()

        resp.content_type = falcon.MEDIA_TEXT
        resp.text = 'Shutting down...\n'
        resp.status = http.HTTPStatus.ACCEPTED


with wsgiref.simple_server.make_server('', 8000, app := falcon.App()) as httpd:
    app.add_route('/shutdown', Shutdown(httpd))
    print('Serving on port 8000, POST to /shutdown to stop...')
    httpd.serve_forever()
```

Warning

While `wsgiref.simple_server` is handy for integration testing, it builds upon `http.server`, which is not recommended for production. (See *Installation* on how to install a production-ready WSGI or ASGI server.)

How can I set cookies when simulating requests?

The easiest way is to simply pass the `cookies` parameter into `simulate_request`. Here is an example:

```
import falcon
import falcon.testing
import pytest

class TastyCookies:

    def on_get(self, req, resp):
        resp.media = {'cookies': req.cookies}


@pytest.fixture
def client():
    app = falcon.App()
    app.add_route('/cookies', TastyCookies())

    return falcon.testing.TestClient(app)


def test_cookies(client):
    resp = client.simulate_get('/cookies', cookies={'cookie': 'cookie value'})

    assert resp.json == {'cookies': {'cookie': 'cookie value'}}
```

Alternatively, you can set the Cookie header directly as demonstrated in this version of `test_cookies()`

```
def test_cookies(client):
    resp = client.simulate_get('/cookies', headers={'Cookie': 'xxx=yyy'})

    assert resp.json == {'cookies': {'xxx': 'yyy'}}
```

To include multiple values, simply use `"; "` to separate each name-value pair. For example, if you were to pass `{'Cookie': 'xxx=yyy; hello=world'}`, you would get `{'cookies': {'xxx': 'yyy', 'hello': 'world'}}`.

Why do I see no error tracebacks in my ASGI application?

When using Falcon with an ASGI server like Uvicorn, you might notice that server errors do not include any traceback by default. This behavior differs from WSGI, where the PEP-3333 specification defines the wsgi.errors stream (which Falcon utilizes to log unhandled *internal server errors*).

Since there is no standardized way to log errors back to the ASGI server, the framework simply opts to log them using the `falcon` `logger`.

The easiest way to get started is configuring the root logger via `logging.basicConfig()`:

```
import logging

import falcon
import falcon.asgi

logging.basicConfig(
    format="%(asctime)s [%(levelname)s] %(message)s", level=logging.INFO)


class FaultyResource:
    async def on_get(self, req, resp):
        raise ValueError('foo')


app = falcon.asgi.App()
app.add_route('/things', FaultyResource())
```

By adding the above logging configuration, you should now see tracebacks logged to `stderr` when accessing `/things`.

For additional details on this topic, please refer to *Debugging ASGI Applications*.

5.2 Community Guide

5.2.1 Get Help

Welcome to the Falcon community! We are a pragmatic group of HTTP enthusiasts working on the next generation of web apps and cloud services. We would love to have you join us and share your ideas.

Please help us spread the word and grow the community!

FAQ

First, *take a quick look at our FAQ* to see if your question has already been addressed. If not, or if the answer is unclear, please don't hesitate to reach out via one of the channels below.

Chat

The Falconry community on Gitter is a great place to ask questions and share your ideas. You can find us in falconry/user. We also have a falconry/dev room for discussing the design and development of the framework itself.

In addition to Gitter, we are also evaluating the popular GitHub Discussions for less "realtime" communication such as usage questions or open-ended design ideas.

Per our *Code of Conduct*, we expect everyone who participates in community discussions to act professionally, and lead by example in encouraging constructive discussions. Each individual in the community is responsible for creating a positive, constructive, and productive culture.

Submit Issues

If you have an idea for a feature, run into something that is harder to use than it should be, or find a bug, please let the crew know in falconry/dev or by submitting an issue. We need your help to make Falcon awesome!

Pay it Forward

We'd like to invite you to help other community members with their questions in falconry/user, and to help peer-review pull requests. If you use the Chrome browser, we recommend installing the NotHub extension to stay up to date with PRs.

If you would like to contribute a new feature or fix a bug in the framework, please check out our Contributor's Guide for more information.

We'd love to have your help!

Code of Conduct

All contributors and maintainers of this project are subject to our *Code of Conduct*.

5.2.2 Contribute to Falcon

Thanks for your interest in the project! We welcome pull requests from developers of all skill levels. To get started, simply fork the master branch on GitHub to your personal account and then clone the fork into your development environment.

If you would like to contribute but don't already have something in mind, we invite you to take a look at the issues listed under our next milestone. If you see one you'd like to work on, please leave a quick comment so that we don't end up with duplicated effort. Thanks in advance!

The core Falcon project maintainers are:

- Kurt Griffiths, Project Lead (**kgriffs** on GH, Gitter, and Twitter)
- John Vrbanac (**jmvrbanac** on GH, Gitter, and Twitter)
- Vytautas Liuolia (**vytas7** on GH and Gitter, and **vliuolia** on Twitter)
- Nick Zaccardi (**nZac** on GH and Gitter)
- Federico Caselli (**CaselIT** on GH and Gitter)

Please don't hesitate to reach out if you have any questions, or just need a little help getting started. You can find us in falconry/dev on Gitter.

Please note that all contributors and maintainers of this project are subject to our Code of Conduct.

Pull Requests

Before submitting a pull request, please ensure you have added or updated tests as appropriate, and that all existing tests still pass with your changes. Please also ensure that your coding style follows PEP 8 and the `ruff` formatting style.

In order to reformat your code with `ruff`, simply issue:

```
$ pip install -U ruff
$ ruff format
```

You can also reformat your code, and apply safe `ruff` fixes, via the `reformat` `tox` environment:

```
$ pip install -U tox
$ tox -e reformat
```

You can check all this by running `tox` from within the Falcon project directory. Your environment must be based on CPython 3.10, 3.11, 3.12, 3.13, or 3.14:

```
$ pip install -U tox
$ tox --recreate
```

Reviews

Falcon is used in a number of mission-critical applications and is known for its stability and reliability. Therefore, we invest a lot of time in carefully reviewing PRs and working with contributors to ensure that every patch merged into the master branch is correct, complete, performant, well-documented, and appropriate.

Project maintainers review each PR for the following:

- **Design.** Does it do the right thing? Is the end goal well understood and correct?
- **Correctness.** Is the logic correct? Does it behave correctly according to the goal of the feature or bug fix?
- **Fit.** Is this feature or fix in keeping with the spirit of the project? Would this idea be better implemented as an add-on?
- **Standards.** Does this change align with approved or standards-track RFCs, de-facto standards, and currently accepted best practices?
- **Tests.** Does the PR implement sufficient test coverage in terms of value inputs, Python versions, and lines tested?
- **Compatibility.** Does it work across all of Falcon's supported Python versions and operating systems?
- **Performance.** Will this degrade performance for request or response handling? Are there opportunities to optimize the implementation?
- **Docs.** Does this impact any existing documentation or require new documentation? If so, does this PR include the aforementioned docs, and is the language friendly, clear, helpful, and grammatically correct with no misspellings? Do all docstrings conform to Google style ala sphinx.ext.napoleon?
- **Dependencies.** Does this PR bring in any unnecessary dependencies that would prevent us from keeping the framework lean and mean, jeopardize the reliability of the project, or significantly increase Falcon's attack service?
- **Changelog.** Does the PR have a changelog entry in newsfragments? Is the type correct? Try running `towncrier --draft` to ensure it renders correctly.

Test coverage

Pull requests must maintain 100% test coverage of all code branches. This helps ensure the quality of the Falcon framework. To check coverage before submitting a pull request:

```
$ tox
```

It is necessary to combine test coverage from multiple environments in order to account for branches in the code that are only taken for a given Python version.

Running the default sequence of `tox` environments generates an HTML coverage report that can be viewed by simply opening `.coverage_html/index.html` in a browser. This can be helpful in tracking down specific lines or branches that are missing coverage.

Debugging

We use pytest to run all of our tests. Pytest supports pdb and will break as expected on any `pdb.set_trace()` calls. If you would like to use pdb++ instead of the standard Python debugger, simply run the following tox environment. This environment also disables coverage checking to speed up the test run, making it ideal for quick sanity checks.

```
$ tox -e py3_debug
```

If you wish, you can customize Falcon's `tox.ini` to install alternative debuggers, such as ipdb or pudb.

Benchmarking

A few simple benchmarks are included with the source under `falcon/bench`. These can be taken as a rough measure of the performance impact (if any) that your changes have on the framework. You can run these tests by invoking one of the tox environments included for this purpose (see also the `tox.ini` file). For example:

```
$ tox -e py310_bench
```

Note that you may pass additional arguments via tox to the falcon-bench command:

```
$ tox -e py310_bench -- -h
$ tox -e py310_bench -- -b falcon -i 20000
```

Alternatively, you may run falcon-bench directly by creating a new virtual environment and installing falcon directly in development mode. In this example we use pyenv with pyenv-virtualenv from within a falcon source directory:

```
$ pyenv virtualenv 3.10.6 falcon-sandbox-310
$ pyenv shell falcon-sandbox-310
$ pip install -r requirements/bench
$ pip install -e .
$ falcon-bench
```

Note that benchmark results for the same code will vary between runs based on a number of factors, including overall system load and CPU scheduling. These factors may be somewhat mitigated by running the benchmarks on a Linux server dedicated to this purpose, and pinning the benchmark process to a specific CPU core.

Documentation

To check documentation changes (including docstrings), before submitting a PR, ensure the tox job builds the documentation correctly:

```
$ tox -e docs

# OS X
$ open docs/_build/html/index.html

# Gnome
$ gnome-open docs/_build/html/index.html

# Generic X Windows
$ xdg-open docs/_build/html/index.html
```

Recipes and code snippets

If you are adding new recipes (in `docs/user/recipes`), try to break out code snippets into separate files inside `examples/recipes`. This allows `ruff` to format these snippets to conform to our code style, as well as check for trivial errors. Then simply use `literalinclude` to embed these snippets into your `.rst` recipe.

If possible, try to implement tests for your recipe in `tests/test_recipes.py`. This helps to ensure that our recipes stay up-to-date as the framework's development progresses!

VS Code Dev Container development environment

When opening the project using the VS Code IDE, if you have Docker (or some drop-in replacement such as Podman or Colima or Rancher Desktop) installed, you can leverage the Dev Containers feature to start a container in the background with all the dependencies required to test and debug the Falcon code. VS Code integrates with the Dev Container seamlessly, which can be configured via devcontainer.json. Once you open the project in VS Code, you can execute the "Reopen in Container" command to start the Dev Container which will run the headless VS Code Server process that the local VS Code app will connect to via a published port.

Use of LLMs ("AI")

Large language models (LLM) for coding are getting increasingly sophisticated, and they *might* be useful for contributing to Falcon too. However, at the same time we are trying to avoid AI-generated "workslop", so if you do rely on an LLM to generate parts of your changeset, please carefully review and test the code manually before submitting the PR.

In addition, if a substantial part of your pull request has been generated by an LLM, we would appreciate if you noted this in your PR's description or comments.

If in doubt, it is often better to leave, e.g., documentation sections missing and ask for help instead of filling them with ostensibly legitimate LLM output.

Code style rules

- Docstrings are required for classes, attributes, methods, and functions. Follow the following guidelines for docstrings:
  - Docstrings should utilize the napoleon style in order to make them read well, regardless of whether they are viewed through `help()` or on Read the Docs.
  - Docstrings should begin with a short (~70 characters or less) summary line that ends in a period.
    - The summary line should begin immediately after the opening quotes (do not add a line break before the summary line)
    - The summary line should describe what it is if it is a class (e.g., "An asynchronous, file-like object for reading ASGI streams.")
    - The summary line should describe what it does when called, if it is a function, structured as an imperative (e.g., "Delete a header that was previously set for this response.")
  - Please try to be consistent with the way existing docstrings are formatted. In particular, note the use of single vs. double backticks as follows:
  - Double backticks
    - Inline code
    - Variables
    - Types
    - Decorators
  - Single backticks
    - Methods
    - Params
    - Attributes
- Format non-trivial comments using your GitHub nick and one of these prefixes:
  - TODO(riker): Damage report!
  - NOTE(riker): Well, that's certainly good to know.
  - PERF(riker): Travel time to the nearest starbase?
  - APPSEC(riker): In all trust, there is the possibility for betrayal.
- When catching exceptions, name the variable `ex`.
- Use whitespace to separate logical blocks of code and to improve readability.
- No single-character variable names except for trivial indexes when looping, or in mathematical expressions implementing well-known formulas.
- Heavily document code that is especially complex and/or clever.

- When in doubt, optimize for readability.

Changelog

We use towncrier to manage the changelog. Each PR that modifies the functionality of Falcon should include a short description in a news fragment file in the `docs/_newsfragments` directory.

The newsfragment file name should have the format `{issue_number}.{fragment_type}.rst`, where the fragment type is one of `breakingchange`, `newandimproved`, `bugfix`, or `misc`. If your PR closes another issue, then the original issue number should be used for the newsfragment; otherwise, use the PR number itself.

Commit Message Format

Falcon's commit message format uses AngularJS's style guide, reproduced here for convenience, with some minor edits for clarity.

Each commit message consists of a **header**, a **body** and a **footer**. The header has a special format that includes a **type**, a **scope** and a **subject**:

```
<type>(<scope>): <subject>
<BLANK LINE>
<body>
<BLANK LINE>
<footer>
```

No line may exceed 100 characters. This makes it easier to read the message on GitHub as well as in various git tools.

Type

Must be one of the following:

- **feat**: A new feature
- **fix**: A bug fix
- **docs**: Documentation only changes
- **style**: Changes that do not affect the meaning of the code (white-space, formatting, missing semi-colons, etc)
- **refactor**: A code change that neither fixes a bug or adds a feature
- **perf**: A code change that improves performance
- **test**: Adding missing tests
- **chore**: Changes to the build process or auxiliary tools and libraries such as documentation generation

Scope

The scope could be anything specifying place of the commit change. For example: `$location`, `$browser`, `$compile`, `$rootScope`, `ngHref`, `ngClick`, `ngView`, etc…

Subject

The subject contains succinct description of the change:

- use the imperative, present tense: "change" not "changed" nor "changes"
- don't capitalize first letter
- no dot (.) at the end

Body

Just as in the **subject**, use the imperative, present tense: "change" not "changed" nor "changes". The body should include the motivation for the change and contrast this with previous behavior.

Footer

The footer should contain any information about **Breaking Changes** and is also the place to reference GitHub issues that this commit **Closes**.

5.2.3 Releases and Versioning

Falcon is developed in the open (yes, You can *contribute* too!) primarily using the falconry/falcon GitHub repository. Newer releases are available from our GitHub release page. Unless noted otherwise in specific files, Falcon releases are covered by the *Apache 2.0 License*.

Semantic Versioning

Falcon strictly adheres to SemVer – incompatible API changes are only introduced in conjunction with a major version increment. Furthermore, when we make breaking changes in a new major version of the framework, the old behavior or feature removal is deprecated in advance at least one minor release leading to the change. In addition to documenting deprecations, we do our best to emit deprecation `warnings` (where feasible).

While there seems to be no clear consensus on whether removing platform support constitutes a SemVer breaking change, Falcon assumes that it does (unless we have communicated otherwise in advance, e.g., the *Falcon 4.x series* only guarantees Python 3.10+ support).

Supported Releases

Current Release

As Falcon is a relatively small project with limited resources and funding, we only actively support the current SemVer major/minor release.

Although older minor releases from the current Falcon series are formally considered *EOL*, the upgrade path to the current release ought to be effortless thanks to Falcon's strict *SemVer* guarantees.

Security Maintenance

We also provide limited security maintenance for the latest release of the old stable SemVer major series until *(1)* it falls back two major releases behind, or *(2)* 730 days (two years bar the possibility of a leap year) pass from the first release of the current stable series (whichever comes first).

"Security maintenance" of an old stable release here means providing new patch versions in response to security vulnerabilities. Other bugs (regardless of their impact) and compatibility with new Python versions are only addressed in the *Current Release*. Reported security vulnerabilities are evaluated on a case by case basis, with the Common Vulnerability Scoring System (CVSS) score reaching at least "Medium" (4.0+) as a starting point.

EOL

Other Falcon versions are considered End of Life (**EOL**).

If you are stuck on an old Falcon version, you are still welcome to *ask for help*! However, while we may be able to advise how to work around a specific issue, we will not ship new patch versions to EOL Falcon releases.

Stable Release Status

The below table summarizes the history of Falcon stable (1.0+) releases alongside their maintenance status:

Major/Minor Version	First Release	Latest Patch Release	Release Status
4.2	*4.2.0* (2025-11-10)	4.2.0	*Current Release*
4.1	*4.1.0* (2025-08-06)	4.1.0	*EOL*
4.0	*4.0.0* (2024-10-18)	*4.0.2* (2024-11-06)	*EOL*
3.1	*3.1.0* (2022-03-25)	*3.1.3* (2023-12-05)	*Security Maintenance* (until 2026-10-18)
3.0	*3.0.0* (2021-04-05)	*3.0.1* (2021-05-11)	*EOL*
2.0	*2.0.0* (2019-04-26)	2.0.0	*EOL*
1.4	*1.4.0* (2018-01-16)	*1.4.1* (2018-01-16)	*EOL*
1.3	*1.3.0* (2017-09-06)	1.3.0	*EOL*
1.2	*1.2.0* (2017-05-02)	1.2.0	*EOL*
1.1	*1.1.0* (2016-10-27)	1.1.0	*EOL*
1.0	*1.0.0* (2016-05-11)	1.0.0	*EOL*

(You can find even older (0.x) release history in our repository's commit tree, or PyPI.)

Pre-releases and Development Versions

The upcoming Falcon version release normally lives in our main development branch (`master`). If you absolutely need a new unreleased feature or fix, You can *install* it directly from GitHub, however, we strongly recommend to deploy only stable Falcon releases to production. That being said, we do try to keep the main development branch in good shape at all times; our Continuous Integration gates ensure that both `master` commits, and pull requests being merged, pass all tests on the supported platforms.

Prior to a new stable release, we normally cut several alphas, betas, and release candidates. Falcon has no fixed pre-release schedule (unlike, e.g., CPython), and the exact number of pre-releases depends on many factors, such as the scope of the impending stable release, the extent of issues found in beta testing, and ultimately, it is decided at the discretion of the release manager and the core maintainer team.

5.2.4 Packaging Guide

Normally, the recommended way to *install Falcon* into your project is using `pip` (or a compatible package/project manager, such as `poetry`, `uv`, and many others) that fetches package archives from PyPI.

However, the PyPI-based way of installation is not always applicable or optimal. For instance, when the system package manager of a Linux distribution is used to install Python-based software, it normally gets the dependencies from the same distribution channel, as the specific versions of the packages were carefully tested to work well together on the operating system in question.

This guide is primarily aimed at engineers who create and maintain Falcon packages for operating systems such as Linux and BSD distributions, as well as alternative Python distribution channels such as conda-forge.

> **Note**
>
> Unless noted otherwise in specific sections, this document is only applicable to Falcon 4.0.1 or newer.

If you run into any packaging issues, questions that this guide does not cover, or just find a bug in Falcon, please *let us know*!

Obtaining Release

In order to package a specific Falcon release, you first need to obtain its source archive.

> **Tip**
>
> Maintaining older versions of Falcon packages? See what we support on our side: *Supported Releases*.

It is up to you which authoritative source to use. The most common alternatives are:

- **Source distribution** (aka "*sdist*") on PyPI.

  If you are unsure, the recommended way is to use our source distribution from PyPI (also available on GitHub releases, see below).

  You can query PyPA Warehouse's JSON API in order to obtain the latest stable version of Falcon, fetch the *sdist* URL, and more.

  The API URL specifically for Falcon is https://pypi.org/pypi/falcon/json. Here is how you can query it using the popular `requests`:

  ```
  >>> import requests
  >>> resp = requests.get('https://pypi.org/pypi/falcon/json')
  >>> for url in resp.json()['urls']:
  ...     if url['packagetype'] == 'sdist':
  ...         print(f'Latest Falcon sdist: {url["url"]}')
  ...
  Latest Falcon sdist: https://files.pythonhosted.org/<...>/falcon-4.0.2.tar.gz
  ```

  (`4.0.2` was the latest version at the time of this writing.)

- **GitHub release archive**.

  Alternatively, you can download the archive from our Releases on GitHub. GitHub automatically archives the whole repository for every release, and attaches the tarball to the release page. In addition, our release automation also uploads the *sdist* (see above) to the release as well.

- **Clone GitHub repository**.

  If your packaging workflow is based on a Git repository that tracks both the framework's source code, and your patches or tooling scripts, you will probably want to clone our GitHGub repository instead.

  Every release has a corresponding annotated Git tag that shares the name with the package version on PyPI, e.g., `4.0.2`.

Semantic Versioning

Falcon strictly adheres to *SemVer* – incompatible API changes are only introduced in conjunction with a major version increment.

When updating your Falcon package, you should always carefully review *the changelog* for the new release that you targeting, especially if you are moving up to a new SemVer major version. (In that case, the release notes will include a **"Breaking Changes"** section.)

For a packager, another section worth checking is called **"Changes to Supported Platforms"**, where we announce support for new Python interpreter versions (or even new implementations), as well as deprecate or remove the old ones.

Attention

The SemVer guarantees primarily cover the publicly documented API from the framework user's perspective, so even a minor release may contain important changes to the build process, tests, and project tooling.

Metadata and Dependencies

It is recommend to synchronize the metadata such as the project's description with recent releases on PyPI.

Falcon has **no hard runtime dependencies** except the standard Python library. So depending on how Python is packaged in your distribution (i.e., whether parts of the stdlib are potentially broken out to separate packages), Falcon should only depend on the basic installation of the targeted Python interpreter.

Note

Falcon has no third-party dependencies since 2.0, however, we were vendoring the `python-mimeparse` library (which also had a different licence, MIT versus Falcon's Apache 2.0).

This is no longer a concern as the relevant functionality has been reimplemented from scratch in Falcon 4.0.0, also fixing some long standing behavioral quirks and bugs on the way. As a result, the Falcon 4.x series currently has no vendored dependencies.

Optional dependencies

Falcon has no official list of optional dependencies, but if you want to provide "suggested packages" or similar, various media (de-) serialization libraries can make good candidates, especially those that have official media handlers such as `msgpack` (*`MessagePackHandler`*). *`JSONHandler`* can be easily customized using faster JSON implementations such as `orjson`, `rapidjson`, etc, so you can suggest those that are already packaged for your distribution.

Otherwise, various ASGI and WSGI application servers could also fit the bill.

See also *Optional test dependencies* for the list of third party libraries that we test against in our Continuous Integration (CI) tests.

Building Binaries

The absolute minimum in terms of packaging is not building any binaries, but just distributing the Python modules found under `falcon/`. This is roughly equivalent to our pure-Python wheel on PyPI.

Tip

The easiest way to skip the binaries is to set the `FALCON_DISABLE_CYTHON` environment variable to a non-empty value in the build environment.

The framework would still function just fine, however, the overall performance would be somewhat (~30-40%) lower, and potentially much lower (an order of magnitude) for certain "hot" code paths that feature a dedicated implementation in Cython.

Note

The above notes on performance only apply to CPython.

In the unlikely case you are packaging Falcon for PyPy, we recommend simply sticking to pure-Python code.

In order to build a binary package, you will obviously need a compiler toolchain, and the CPython library headers. Hopefully your distribution already has Python tooling that speaks PEP 517 – this is how the framework's build process is implemented (using the popular setuptools).

We also use cibuildwheel to build our packages that are later uploaded to PyPI, but we realize that its isolated, Docker-centric approach probably does not lend itself very well to packaging for a distribution of an operating system.

If your build process does not support installation of build dependencies in a PEP 517 compatible way, you will also have to install Cython yourself (version 3.0.8 or newer is recommended to build Falcon).

Big-endian support

We regularly build and test *binary wheels* on the IBM Z platform (aka `s390x`) which is big-endian. We are not aware of any endianness-related issues.

32-bit support

Falcon is not very well tested on 32-bit systems, and we do not provide any 32-bit binary wheels either. We even explicitly fall back to pure-Python code in some cases such as the multipart form parser (as the smaller `Py_ssize_t` would interfere with uploading of files larger than 2 GiB) if we detect a 32-bit flavor of CPython.

If you do opt to provide 32-bit Falcon binaries, make sure that you run *extensive tests* against the built package.

Building Documentation

It is quite uncommon to also include offline documentation (or to provide a separate documentation package) as the user can simply browse our documentation at Read the Docs. Even if the package does not contain the latest version of Falcon, it is possible to switch to an older one using Read the Docs version picker.

If you do decide to ship the offline docs too, you can build it using `docs/Makefile` (you can also invoke `sphinx-build` directly).

> **Note**
>
> Building the HTML documentation requires the packages listed in `requirements/docs`.
>
> Building `man` pages requires only Sphinx itself and the plugins referenced directly in `docs/conf.py` (currently `myst-parser`, `sphinx-copybutton`, and `sphinx-design`).

- To build HTML docs, use `make html`.

  The resulting files will be built in `docs/_build/html/`.

- To build man pages, use `make man`.

  The resulting man page file will be called `docs/_build/man/falcon.1`.

  You will need to rename this file to match your package naming standards, and copy it an appropriate man page directory (typically under `/usr/share/man/` or similar).

Testing Package

When your Falcon package is ready, it is a common (highly recommended!) practice to install it into your distribution, and run tests verifying that the package functions as intended.

As of Falcon 4.0+, the only hard test dependency is `pytest`.

You can simply run it against Falcon's test suite found in the `tests/` subdirectory:

```
pytest tests/
```

These tests will provide decent (98-99%), although not complete, code coverage, and should ensure that the basic wiring of your package is correct (however, see also the next chapter: *Optional test dependencies*).

> **Tip**
>
> You can run `pytest` from any directory, i.e., the below should work just fine:
> `/usr/local/foo-bin/pytest /bar/baz/falcon-release-dir/tests/`
>
> This pattern is regularly exercised in our CI gates, as cibuildwheel (see above) does not run tests from the project's directory either.

Optional test dependencies

As mentioned above, Falcon has no hard test dependencies except `pytest`, however, our test suite includes optional integration tests against a selection of third-party libraries.

When building *wheels* with cibuildwheel, we install a small subset of the basic optional test dependencies, see the `requirements/cibwtest` file in the repository. Furthermore, when running our full test suite in the CI, we exercise integration with a larger number of optional libraries and applications servers (see the `requirements/tests` file, as well as various ASGI/WSGI server integration test definitions in `tox.ini`).

Ideally, if your distribution also provides packages for any of the above optional test dependencies, it may be a good idea to install them into your test environment as well. This will help verifying that your Falcon package is compatible with the specific versions of these packages in your distribution.

Thank You

If you are already maintaining Falcon packages, thank you!

Although we do not have the bandwidth to maintain Falcon packages for any distribution channel beyond PyPI ourselves, we are happy to help if you run into any problems. File an issue on GitHub, or just *send us a message*!

5.2.5 Code of Conduct

All contributors and maintainers of this project are subject to this Code of Conduct.

We pledge to respect everyone who contributes to the *Falcon* project or other associated activities by (including but not limited to) creating project issues, submitting pull requests, and providing feedback on the same. We also pledge to respect everyone who participates in discussions both online and at meetups and conferences.

Unacceptable behavior includes (but is not limited to) offensive verbal comments related to gender, gender identity and expression, age, sexual orientation, disability, physical appearance, body size, race, ethnicity, religion, technology choices, sexual images in public spaces, deliberate intimidation, stalking, following, harassing photography or recording, sustained disruption of talks or other events, inappropriate physical contact, and unwelcome sexual attention.

If anyone within the *Falcon* community fails to adhere to this Code of Conduct, they can expect to face action by way of: removing comments, removing issues, deleting pull requests, and generally being banned from participating in the *Falcon* community.

Do This

- Act professionally
- Treat others as friends and family
- Seek first to understand
- Be honest, transparent, and constructive
- Use clear, concise language
- Assume good intent

Don't Do This

- Use indecent, profane, or degrading language of any kind
- Hold a patch hostage for some ulterior motive
- Take part in gossiping or backbiting
- Engage in bullying behaviors, including but not limited to:
  - Belittling others' opinions
  - Persistent teasing or sarcasm
  - Insulting, threatening, or yelling at someone
  - Accusing someone of being incompetent
  - Setting someone up to fail
  - Humiliating someone
  - Isolating someone from others
  - Withholding information to gain an advantage
  - Falsely accusing someone of errors
  - Sabotaging someone's work

Moderation

Discussing things "in the open" is important. But it does not necessarily engender trust in a community, especially in the absence of cognizant moderation.

Unfortunately, open discussions sometimes lead to a culture of distrust. This happens when individuals in power publicly humiliate, intimidate, or even bully others in the community. Often this happens unconsciously or as a result of a poor choice of words. Regardless of whether the intent is actual or perceived, the damage to the community is the same, and must be quickly dealt with before it festers into a culture of enmity.

The project maintainer, with the support of the core reviewers, pledges to lead by example, as well as to hold contributors accountable for creating a positive, constructive, and productive culture. Inappropriate behavior will not be tolerated.

If a member of the *Falcon* community behaves unacceptably toward you or another individual, please contact Kurt Griffiths at inbox@kgriffs.com.

This Code of Conduct, created by Falcon Contributors and inspired by yourfirstpr, is licensed under a Creative Commons Attribution 4.0 International License.

5.3 Framework Reference

5.3.1 The App Class

Falcon supports both the WSGI (*falcon.App*) and ASGI (*falcon.asgi.App*) protocols. This is done by instantiating the respective `App` class to create a callable WSGI or ASGI "application".

Because Falcon's `App` classes are built on WSGI and ASGI, you can host them with any standard-compliant server.

```
import falcon
import falcon.asgi

wsgi_app = falcon.App()
asgi_app = falcon.asgi.App()
```

WSGI App

class falcon.**App**(*media_type: str = 'application/json'*, *request_type: type[_ReqT] | None = None*, *response_type: type[_RespT] | None = None*, *middleware: SyncMiddleware[_ReqT, _RespT] | Iterable[SyncMiddleware[_ReqT, _RespT]] | None = None*, *router: routing.CompiledRouter | None = None*, *independent_middleware: bool = True*, *cors_enable: bool = False*, *sink_before_static_route: bool = True*)

The main entry point into a Falcon-based WSGI app.

Each App instance provides a callable WSGI interface and a routing engine (for ASGI applications, see *falcon.asgi.App*).

> **Note**
>
> The `API` class was renamed to `App` in Falcon 3.0. The old class name remains available as an alias for backwards-compatibility, but will be removed in a future release.

Keyword Arguments

- **media_type** (*str*) – Default media type to use when initializing *RequestOptions* and *ResponseOptions*. The `falcon` module provides a number of constants for common media types, such as `falcon.MEDIA_MSGPACK`, `falcon.MEDIA_YAML`, `falcon.MEDIA_XML`, etc.
- **middleware** – Either a single middleware component object or an iterable of objects (instantiated classes) that implement the following middleware component interface. Note that it is only necessary to implement the methods for the events you would like to handle; Falcon simply skips over any missing middleware methods:

```
class ExampleMiddleware:
    def process_request(self, req: Request, resp: Response) ->␣
↪None:
        """Process the request before routing it.

        Note:
            Because Falcon routes each request based on
            req.path, a request can be effectively re-routed
            by setting that attribute to a new value from
            within process_request().

        Args:
            req: Request object that will eventually be
                routed to an on_* responder method.
            resp: Response object that will be routed to
                the on_* responder.
        """

        def process_resource(
            self,
            req: Request,
            resp: Response,
            resource: object,
            params: dict[str, Any],
        ) -> None:
        """Process the request and resource *after* routing.

        Note:
            This method is only called when the request matches
```

(continues on next page)

(continued from previous page)

```
            a route to a resource.

        Args:
            req: Request object that will be passed to the
                routed responder.
            resp: Response object that will be passed to the
                responder.
            resource: Resource object to which the request was
                routed. May be None if no route was found for
                the request.
            params: A dict-like object representing any
                additional params derived from the route's URI
                template fields, that will be passed to the
                resource's responder method as keyword
                arguments.
        """

    def process_response(
        self,
        req: Request,
        resp: Response,
        resource: object,
        req_succeeded: bool
    ) -> None:
        """Post-processing of the response (after routing).

        Args:
            req: Request object.
            resp: Response object.
            resource: Resource object to which the request was
                routed. May be None if no route was found
                for the request.
            req_succeeded: True if no exceptions were raised
                while the framework processed and routed the
                request; otherwise False.
        """
```

(See also: *Middleware*)

- **request_type** – `Request`-like class to use instead of Falcon's default class. Among other things, this feature affords inheriting from *falcon.Request* in order to override the `context_type` class variable (default: *falcon.Request*)
- **response_type** – `Response`-like class to use instead of Falcon's default class (default: *falcon.Response*)
- **router** (*object*) – An instance of a custom router to use in lieu of the default engine. (See also: *Custom Routers*)
- **independent_middleware** (*bool*) – Set to `False` if response middleware should not be executed independently of whether or not request middleware raises an exception (default `True`). When this option is set to `False`, a middleware component's `process_response()` method will NOT be called when that same component's `process_request()` (or that of a component higher up in the stack) raises an exception.
- **cors_enable** (*bool*) – Set this flag to `True` to enable a simple CORS policy for all responses, including support for preflighted requests. An instance of *CORSMiddleware* can instead be passed to the middleware argument to customize its behaviour. (default `False`). (See also: *CORS*)

- **sink_before_static_route** (*bool*) – Indicates if the sinks should be processed before (when `True`) or after (when `False`) the static routes. This has an effect only if no route was matched. (default `True`)

add_error_handler(*exception: type[Exception] | Iterable[type[Exception]]*, *handler: ErrorHandler[_ReqT, _RespT] | None = None*) → None

Register a handler for one or more exception types.

Error handlers may be registered for any exception type, including *HTTPError* or *HTTPStatus*. This feature provides a central location for logging and otherwise handling exceptions raised by responders, hooks, and middleware components.

A handler can raise an instance of *HTTPError* or *HTTPStatus* to communicate information about the issue to the client. Alternatively, a handler may modify *resp* directly.

An error handler "matches" a raised exception if the exception is an instance of the corresponding exception type. If more than one error handler matches the raised exception, the framework will choose the most specific one, as determined by the method resolution order of the raised exception type. If multiple error handlers are registered for the *same* exception class, then the most recently-registered handler is used.

For example, suppose we register error handlers as follows:

```
app = App()
app.add_error_handler(falcon.HTTPNotFound, custom_handle_not_found)
app.add_error_handler(falcon.HTTPError, custom_handle_http_error)
app.add_error_handler(Exception, custom_handle_uncaught_exception)
app.add_error_handler(falcon.HTTPNotFound, custom_handle_404)
```

If an instance of `falcon.HTTPForbidden` is raised, it will be handled by `custom_handle_http_error()`. `falcon.HTTPError` is a superclass of `falcon.HTTPForbidden` and a subclass of `Exception`, so it is the most specific exception type with a registered handler.

If an instance of `falcon.HTTPNotFound` is raised, it will be handled by `custom_handle_404()`, not by `custom_handle_not_found()`, because `custom_handle_404()` was registered more recently.

Note

By default, the framework installs three handlers, one for *HTTPError*, one for *HTTPStatus*, and one for the standard `Exception` type, which prevents passing uncaught exceptions to the WSGI server. These can be overridden by adding a custom error handler method for the exception type in question.

Parameters

- **exception** (*type or iterable of types*) – When handling a request, whenever an error occurs that is an instance of the specified type(s), the associated handler will be called. Either a single type or an iterable of types may be specified.
- **handler** (*callable*) – A function or callable object taking the form `func(req, resp, ex, params)`.

  If not specified explicitly, the handler will default to `exception.handle`, where `exception` is the error type specified above, and `handle` is a static method (i.e., decorated with `@staticmethod`) that accepts the same params just described. For example:

```
class CustomException(CustomBaseException):

    @staticmethod
    def handle(req, resp, ex, params):
        # TODO: Log the error
```

(continues on next page)

(continued from previous page)

```
        # Convert to an instance of falcon.HTTPError
        raise falcon.HTTPError(falcon.HTTP_792)
```

If an iterable of exception types is specified instead of a single type, the handler must be explicitly specified.

Changed in version 3.0: The error handler is now selected by the most-specific matching error class, rather than the most-recently registered matching error class.

add_middleware(*middleware: SyncMiddleware[_ReqT]*) → None

Add one or more additional middleware components.

Parameters

middleware – Either a single middleware component or an iterable of components to add. The component(s) will be invoked, in order, as if they had been appended to the original middleware list passed to the class initializer.

add_route(*uri_template: str*, *resource: object*, ***kwargs: Any*) → None

Associate a templatized URI path with a resource.

Falcon routes incoming requests to resources based on a set of URI templates. If the path requested by the client matches the template for a given route, the request is then passed on to the associated resource for processing.

Note

If no route matches the request, control then passes to a default responder that simply raises an instance of *HTTPRouteNotFound*. By default, this error will be rendered as a 404 response, but this behavior can be modified by adding a custom error handler (see also *this FAQ topic*).

On the other hand, if a route is matched but the resource does not implement a responder for the requested HTTP method, the framework invokes a default responder that raises an instance of *HTTPMethodNotAllowed*.

This method delegates to the configured router's `add_route()` method. To override the default behavior, pass a custom router object to the *App* initializer.

(See also: *Routing*)

Parameters

- **uri_template** (*str*) – A templatized URI. Care must be taken to ensure the template does not mask any sink patterns, if any are registered.

  (See also: *add_sink()*)

  Warning

  If *strip_url_path_trailing_slash* is enabled, *uri_template* should be provided without a trailing slash.

  (See also: *How does Falcon handle a trailing slash in the request path?*)

- **resource** (*instance*) – Object which represents a REST resource. Falcon will pass GET requests to `on_get()`, PUT requests to `on_put()`, etc. If any HTTP methods are not supported by your resource, simply don't define the corresponding request handlers, and Falcon will do the right thing.

> **Note**
>
> When using an async version of the `App`, all request handlers must be awaitable coroutine functions.

Keyword Arguments

- **`suffix`** (*str*) – Optional responder name suffix for this route. If a suffix is provided, Falcon will map GET requests to `on_get_{suffix}()`, POST requests to `on_post_{suffix}()`, etc. In this way, multiple closely-related routes can be mapped to the same resource. For example, a single resource class can use suffixed responders to distinguish requests for a single item vs. a collection of those same items. Another class might use a suffixed responder to handle a shortlink route in addition to the regular route for the resource. For example:

```
class Baz(object):

    def on_get_foo(self, req, resp):
        pass

    def on_get_bar(self, req, resp):
        pass

baz = Baz()
app = falcon.App()
app.add_route('/foo', baz, suffix='foo')
app.add_route('/bar', baz, suffix='bar')
```

- **`compile`** (*bool*) – Optional flag that can be provided when using the default *CompiledRouter* to compile the routing logic on this call, since it will otherwise delay compilation until the first request is routed. See *CompiledRouter.add_route()* for further details.

> **Note**
>
> Any additional keyword arguments not defined above are passed through to the underlying router's `add_route()` method. The default router ignores any additional keyword arguments, but custom routers may take advantage of this feature to receive additional options when setting up routes. Custom routers MUST accept such arguments using the variadic pattern (`**kwargs`), and ignore any keyword arguments that they don't support.

`add_sink`(*sink: SinkCallable[_ReqT, _RespT]*, *prefix: str | Pattern[str] = '/'*) → None

Register a sink method for the App.

If no route matches a request, but the path in the requested URI matches a sink prefix, Falcon will pass control to the associated sink, regardless of the HTTP method requested.

Using sinks, you can drain and dynamically handle a large number of routes, when creating static resources and responders would be impractical. For example, you might use a sink to create a smart proxy that forwards requests to one or more backend services.

> **Note**
>
> To support CORS preflight requests when using the default CORS middleware, either by setting `App.cors_enable=True` or by adding the *CORSMiddleware* to the `App.middleware`, the sink should set the `Allow` header in the request to the allowed method values when serving an `OPTIONS`

request. If the `Allow` header is missing from the response, the default CORS middleware will deny the preflight request.

Parameters

- **sink** (*callable*) – A callable taking the form `func(req, resp, **kwargs)`.

  Note

  When using an async version of the `App`, this must be a coroutine function taking the form `func(req, resp, ws=None, **kwargs)`.

  Similar to *error handlers*, in the case of a WebSocket connection, the *resp* argument will be `None`, whereas the *ws* keyword argument will receive the *WebSocket* connection object.

  For backwards-compatibility, when *ws* is absent from the sink's signature, or a regex match (see **prefix** below) contains a group named 'ws', the *WebSocket* object is passed in place of the incompatible *resp*.

  This behavior will change in Falcon 5.0: when draining a WebSocket connection, *resp* will always be set to `None` regardless of the sink's signature.

  Added in version 4.1: If an asynchronous sink callable explicitly defines a *ws* argument, it is used to pass the *WebSocket* connection object.

- **prefix** (*str*) – A regex string, typically starting with '/', which will trigger the sink if it matches the path portion of the request's URI. Both strings and precompiled regex objects may be specified. Characters are matched starting at the beginning of the URI path.

  Note

  Named groups are converted to kwargs and passed to the sink as such.

  Warning

  If the prefix overlaps a registered route template, the route will take precedence and mask the sink.

  (See also: *add_route()*)

add_static_route(*prefix: str*, *directory: str | Path*, *downloadable: bool = False*, *fallback_filename: str | None = None*) → None

Add a route to a directory of static files.

Static routes provide a way to serve files directly. This feature provides an alternative to serving files at the web server level when you don't have that option, when authorization is required, or for testing purposes.

Warning

Serving files directly from the web server, rather than through the Python app, will always be more efficient, and therefore should be preferred in production deployments. For security reasons, the

directory and the fallback_filename (if provided) should be read only for the account running the application.

Warning

If you need to serve large files and/or progressive downloads (such as in the case of video streaming) through the Falcon app, check that your application server's timeout settings can accommodate the expected request duration (for instance, the popular Gunicorn kills `sync` workers after 30 seconds unless configured otherwise).

Note

For ASGI apps, file reads are made non-blocking by scheduling them on the default executor.

Static routes are matched in LIFO order. Therefore, if the same prefix is used for two routes, the second one will override the first. This also means that more specific routes should be added *after* less specific ones. For example, the following sequence would result in `'/foo/bar/thing.js'` being mapped to the `'/foo/bar'` route, and `'/foo/xyz/thing.js'` being mapped to the `'/foo'` route:

```
app.add_static_route('/foo', foo_path)
app.add_static_route('/foo/bar', foobar_path)
```

Parameters

- **prefix** (*str*) – The path prefix to match for this route. If the path in the requested URI starts with this string, the remainder of the path will be appended to the source directory to determine the file to serve. This is done in a secure manner to prevent an attacker from requesting a file outside the specified directory.

  Note that static routes are matched in LIFO order, and are only attempted after checking dynamic routes and sinks.

- **directory** (*Union[str, pathlib.Path]*) – The source directory from which to serve files.

- **downloadable** (*bool*) – Set to `True` to include a Content-Disposition header in the response. The "filename" directive is simply set to the name of the requested file.

- **fallback_filename** (*str*) – Fallback filename used when the requested file is not found. Can be a relative path inside the prefix folder or any valid absolute path.

req_options: *RequestOptions*

A set of behavioral options related to incoming requests.

See also: *RequestOptions*

resp_options: *ResponseOptions*

A set of behavioral options related to outgoing responses.

See also: *ResponseOptions*

property router_options: *CompiledRouterOptions*

Configuration options for the router.

If a custom router is in use, and it does not expose any configurable options, referencing this attribute will raise an instance of `AttributeError`.

See also: *CompiledRouterOptions*.

set_error_serializer(*serializer: Callable[[_ReqT, _RespT, HTTPError], None]*) → None

Override the default serializer for instances of *HTTPError*.

When a responder raises an instance of *HTTPError*, Falcon converts it to an HTTP response automatically. The default serializer supports JSON and XML, but may be overridden by this method to use a custom serializer in order to support other media types.

> **Note**
>
> If a custom media type is used and the type includes a "+json" or "+xml" suffix, the default serializer will convert the error to JSON or XML, respectively.

> **Note**
>
> A custom serializer set with this method may not be called if the default error handler for *HTTPError* has been overridden. See *add_error_handler()* for more details.

The *HTTPError* class contains helper methods, such as *to_json()* and *to_dict()*, that can be used from within custom serializers. For example:

```
def my_serializer(
    req: Request, resp: Response, exception: HTTPError
) -> None:
    representation = None

    preferred = req.client_prefers((falcon.MEDIA_YAML, falcon.MEDIA_JSON))

    if preferred is not None:
        if preferred == falcon.MEDIA_JSON:
            resp.data = exception.to_json()
        else:
            resp.text = yaml.dump(exception.to_dict(), encoding=None)
        resp.content_type = preferred

    resp.append_header('Vary', 'Accept')
```

Parameters

serializer (*callable*) – A function taking the form `func(req, resp, exception)`, where *req* is the request object that was passed to the responder method, *resp* is the response object, and *exception* is an instance of `falcon.HTTPError`.

ASGI App

class falcon.asgi.**App**(*media_type: str = 'application/json', request_type: type[_ReqT] | None = None, response_type: type[_RespT] | None = None, middleware: AsyncMiddleware[_ReqT, _RespT] | Iterable[AsyncMiddleware[_ReqT, _RespT]] | None = None, router: routing.CompiledRouter | None = None, independent_middleware: bool = True, cors_enable: bool = False, sink_before_static_route: bool = True*)

The main entry point into a Falcon-based ASGI app.

Each App instance provides a callable ASGI interface and a routing engine (for WSGI applications, see *falcon.App*).

Keyword Arguments

- **media_type** (*str*) – Default media type to use when initializing *RequestOptions* and *ResponseOptions*. The `falcon` module provides a number of constants for com-

mon media types, such as `falcon.MEDIA_MSGPACK`, `falcon.MEDIA_YAML`, `falcon.MEDIA_XML`, etc.

- **`middleware`** – Either a single middleware component object or an iterable of objects (instantiated classes) that implement the middleware component interface shown below.

  The interface provides support for handling both ASGI worker lifespan events and per-request events. However, because lifespan events are an optional part of the ASGI specification, they may or may not fire depending on your ASGI server.

  A lifespan event handler can be used to perform startup or shutdown activities for the main event loop. An example of this would be creating a connection pool and subsequently closing the connection pool to release the connections.

  Note

  In a multi-process environment, lifespan events will be triggered independently for the individual event loop associated with each process.

  Note

  The framework requires that all middleware methods be implemented as coroutine functions via *async def*. However, it is possible to implement middleware classes that support both ASGI and WSGI apps by distinguishing the ASGI methods with an **_async* postfix (see also: *Middleware*).

  It is only necessary to implement the methods for the events you would like to handle; Falcon simply skips over any missing middleware methods:

```
class ExampleMiddleware:
    async def process_startup(
        self, scope: dict[str, Any], event: dict[str, Any]
    ) -> None:
        """Process the ASGI lifespan startup event.

        Invoked when the server is ready to start up and
        receive connections, but before it has started to
        do so.

        To halt startup processing and signal to the server␣
↪that it
        should terminate, simply raise an exception and the
        framework will convert it to a "lifespan.startup.failed"
        event for the server.

        Args:
            scope (dict): The ASGI scope dictionary for the
                lifespan protocol. The lifespan scope exists
                for the duration of the event loop.
            event (dict): The ASGI event dictionary for the
                startup event.
        """

    async def process_shutdown(
        self, scope: dict[str, Any], event: dict[str, Any]
    ) -> None:
        """Process the ASGI lifespan shutdown event.
```

(continues on next page)

(continued from previous page)

```
        Invoked when the server has stopped accepting
        connections and closed all active connections.

        To halt shutdown processing and signal to the server
        that it should immediately terminate, simply raise an
        exception and the framework will convert it to a
        "lifespan.shutdown.failed" event for the server.

        Args:
            scope (dict): The ASGI scope dictionary for the
                lifespan protocol. The lifespan scope exists
                for the duration of the event loop.
            event (dict): The ASGI event dictionary for the
                shutdown event.
        """

    async def process_request(
        self, req: Request, resp: Response
    ) -> None:
        """Process the request before routing it.

        Note:
            Because Falcon routes each request based on
            req.path, a request can be effectively re-routed
            by setting that attribute to a new value from
            within process_request().

        Args:
            req: Request object that will eventually be
                routed to an on_* responder method.
            resp: Response object that will be routed to
                the on_* responder.
        """

    async def process_resource(
        self,
        req: Request,
        resp: Response,
        resource: object,
        params: dict[str, Any],
    ) -> None:
        """Process the request and resource *after* routing.

        Note:
            This method is only called when the request matches
            a route to a resource.

        Args:
            req: Request object that will be passed to the
                routed responder.
            resp: Response object that will be passed to the
                responder.
            resource: Resource object to which the request was
                routed. May be ``None`` if no route was found␣
↪for
```

(continues on next page)

(continued from previous page)

```
                the request.
            params: A dict-like object representing any
                additional params derived from the route's URI
                template fields, that will be passed to the
                resource's responder method as keyword
                arguments.
        """

    async def process_response(
        self,
        req: Request,
        resp: Response,
        resource: object,
        req_succeeded: bool
    ) -> None:
        """Post-processing of the response (after routing).

        Args:
            req: Request object.
            resp: Response object.
            resource: Resource object to which the request was
                routed. May be ``None`` if no route was found
                for the request.
            req_succeeded: True if no exceptions were raised
                while the framework processed and routed the
                request; otherwise False.
        """

    # WebSocket methods
    async def process_request_ws(
        self, req: Request, ws: WebSocket
    ) -> None:
        """Process a WebSocket handshake request before routing
↪it.

        Note:
            Because Falcon routes each request based on req.
↪path, a
            request can be effectively re-routed by setting that
            attribute to a new value from within process_
↪request().

        Args:
            req: Request object that will eventually be
                passed into an on_websocket() responder method.
            ws: The WebSocket object that will be passed into
                on_websocket() after routing.
        """

    async def process_resource_ws(
        self,
        req: Request,
        ws: WebSocket,
        resource: object,
        params: dict[str, Any],
    ) -> None:
```

(continues on next page)

(continued from previous page)

```
        """Process a WebSocket handshake request after routing.

        Note:
            This method is only called when the request matches
            a route to a resource.

        Args:
            req: Request object that will be passed to the
                routed responder.
            ws: WebSocket object that will be passed to the
                routed responder.
            resource: Resource object to which the request was
                routed.
            params: A dict-like object representing any additional
                params derived from the route's URI template fields,
                that will be passed to the resource's responder
                method as keyword arguments.
        """
```

(See also: *Middleware*)

- **request_type** – `Request`-like class to use instead of Falcon's default class. Among other things, this feature affords inheriting from *falcon.asgi.Request* in order to override the `context_type` class variable (default: *falcon.asgi.Request*)
- **response_type** – `Response`-like class to use instead of Falcon's default class (default: *falcon.asgi.Response*)
- **router** (*object*) – An instance of a custom router to use in lieu of the default engine. (See also: *Custom Routers*)
- **independent_middleware** (*bool*) – Set to `False` if response middleware should not be executed independently of whether or not request middleware raises an exception (default `True`). When this option is set to `False`, a middleware component's `process_response()` method will NOT be called when that same component's `process_request()` (or that of a component higher up in the stack) raises an exception.
- **cors_enable** (*bool*) – Set this flag to `True` to enable a simple CORS policy for all responses, including support for preflighted requests. An instance of `CORSMiddleware` can instead be passed to the middleware argument to customize its behaviour. (default `False`). (See also: *CORS*)
- **sink_before_static_route** (*bool*) – Indicates if the sinks should be processed before (when `True`) or after (when `False`) the static routes. This has an effect only if no route was matched. (default `True`)

add_error_handler(*exception: type[Exception] | Iterable[type[Exception]]*, *handler: AsgiErrorHandler[_ReqT, _RespT] | None = None*) → None

Register a handler for one or more exception types.

Error handlers may be registered for any exception type, including *HTTPError* or *HTTPStatus*. This feature provides a central location for logging and otherwise handling exceptions raised by responders, hooks, and middleware components.

A handler can raise an instance of *HTTPError* or *HTTPStatus* to communicate information about the issue to the client. Alternatively, a handler may modify *resp* directly.

An error handler "matches" a raised exception if the exception is an instance of the corresponding exception type. If more than one error handler matches the raised exception, the framework will choose the most specific one, as determined by the method resolution order of the raised exception type. If multiple

error handlers are registered for the *same* exception class, then the most recently-registered handler is used.

For example, suppose we register error handlers as follows:

```
app = App()
app.add_error_handler(falcon.HTTPNotFound, custom_handle_not_found)
app.add_error_handler(falcon.HTTPError, custom_handle_http_error)
app.add_error_handler(Exception, custom_handle_uncaught_exception)
app.add_error_handler(falcon.HTTPNotFound, custom_handle_404)
```

If an instance of `falcon.HTTPForbidden` is raised, it will be handled by `custom_handle_http_error()`. `falcon.HTTPError` is a superclass of `falcon.HTTPForbidden` and a subclass of `Exception`, so it is the most specific exception type with a registered handler.

If an instance of `falcon.HTTPNotFound` is raised, it will be handled by `custom_handle_404()`, not by `custom_handle_not_found()`, because `custom_handle_404()` was registered more recently.

Note

By default, the framework installs three handlers, one for *HTTPError*, one for *HTTPStatus*, and one for the standard `Exception` type, which prevents passing uncaught exceptions to the WSGI server. These can be overridden by adding a custom error handler method for the exception type in question.

When a generic unhandled exception is raised while handling a *WebSocket* connection, the default handler will close the connection with the standard close code `1011` (Internal Error). If your ASGI server does not support this code, the framework will use code `3011` instead; or you can customize it via the *error_close_code* property of *ws_options*.

On the other hand, if an `on_websocket()` responder raises an instance of *HTTPError*, the default error handler will close the *WebSocket* connection with a framework close code derived by adding `3000` to the HTTP status code (e.g., `3404`)

Parameters

- **exception** (*type or iterable of types*) – When handling a request, whenever an error occurs that is an instance of the specified type(s), the associated handler will be called. Either a single type or an iterable of types may be specified.
- **handler** (*callable*) – A coroutine function taking the form:

```
async def func(req, resp, ex, params, ws=None):
    pass
```

  In the case of a WebSocket connection, the *resp* argument will be `None`, while the *ws* keyword argument will receive the *WebSocket* object representing the connection.

  If the *handler* keyword argument is not provided to *add_error_handler()*, the handler will default to `exception.handle`, where `exception` is the error type specified above, and `handle` is a static method (i.e., decorated with `@staticmethod`) that accepts the params just described. For example:

```
class CustomException(CustomBaseException):

    @staticmethod
    async def handle(req, resp, ex, params):
        # TODO: Log the error
        # Convert to an instance of falcon.HTTPError
        raise falcon.HTTPError(falcon.HTTP_792)
```

Note, however, that if an iterable of exception types is specified instead of a single type, the handler must be explicitly specified using the *handler* keyword argument.

add_middleware(*middleware: AsyncMiddleware[_ReqT]*) → None

Add one or more additional middleware components.

Parameters

middleware – Either a single middleware component or an iterable of components to add. The component(s) will be invoked, in order, as if they had been appended to the original middleware list passed to the class initializer.

add_route(*uri_template: str*, *resource: object*, ***kwargs: Any*) → None

Associate a templatized URI path with a resource.

Falcon routes incoming requests to resources based on a set of URI templates. If the path requested by the client matches the template for a given route, the request is then passed on to the associated resource for processing.

Note

If no route matches the request, control then passes to a default responder that simply raises an instance of `HTTPRouteNotFound`. By default, this error will be rendered as a 404 response, but this behavior can be modified by adding a custom error handler (see also *this FAQ topic*).

On the other hand, if a route is matched but the resource does not implement a responder for the requested HTTP method, the framework invokes a default responder that raises an instance of `HTTPMethodNotAllowed`.

This method delegates to the configured router's `add_route()` method. To override the default behavior, pass a custom router object to the `App` initializer.

(See also: *Routing*)

Parameters

- **uri_template** (*str*) – A templatized URI. Care must be taken to ensure the template does not mask any sink patterns, if any are registered.

  (See also: `add_sink()`)

  Warning

  If `strip_url_path_trailing_slash` is enabled, *uri_template* should be provided without a trailing slash.

  (See also: *How does Falcon handle a trailing slash in the request path?*)

- **resource** (*instance*) – Object which represents a REST resource. Falcon will pass GET requests to `on_get()`, PUT requests to `on_put()`, etc. If any HTTP methods are not supported by your resource, simply don't define the corresponding request handlers, and Falcon will do the right thing.

  Note

  When using an async version of the `App`, all request handlers must be awaitable coroutine functions.

Keyword Arguments

- **suffix** (*str*) – Optional responder name suffix for this route. If a suffix is provided, Falcon will map GET requests to `on_get_{suffix}()`, POST requests to `on_post_{suffix}()`, etc. In this way, multiple closely-related routes can be mapped to the same resource. For example, a single resource class can use suffixed responders to distinguish requests for a single item vs. a collection of those same items. Another class might use a suffixed responder to handle a shortlink route in addition to the regular route for the resource. For example:

```
class Baz(object):

    def on_get_foo(self, req, resp):
        pass

    def on_get_bar(self, req, resp):
        pass

baz = Baz()
app = falcon.App()
app.add_route('/foo', baz, suffix='foo')
app.add_route('/bar', baz, suffix='bar')
```

- **compile** (*bool*) – Optional flag that can be provided when using the default *CompiledRouter* to compile the routing logic on this call, since it will otherwise delay compilation until the first request is routed. See *CompiledRouter.add_route()* for further details.

> **Note**
>
> Any additional keyword arguments not defined above are passed through to the underlying router's `add_route()` method. The default router ignores any additional keyword arguments, but custom routers may take advantage of this feature to receive additional options when setting up routes. Custom routers MUST accept such arguments using the variadic pattern (`**kwargs`), and ignore any keyword arguments that they don't support.

add_sink(*sink: AsgiSinkCallable[_ReqT, _RespT]*, *prefix: str | Pattern[str] = '/'*) → None

Register a sink method for the App.

If no route matches a request, but the path in the requested URI matches a sink prefix, Falcon will pass control to the associated sink, regardless of the HTTP method requested.

Using sinks, you can drain and dynamically handle a large number of routes, when creating static resources and responders would be impractical. For example, you might use a sink to create a smart proxy that forwards requests to one or more backend services.

> **Note**
>
> To support CORS preflight requests when using the default CORS middleware, either by setting `App.cors_enable=True` or by adding the *CORSMiddleware* to the `App.middleware`, the sink should set the `Allow` header in the request to the allowed method values when serving an `OPTIONS` request. If the `Allow` header is missing from the response, the default CORS middleware will deny the preflight request.

Parameters

- **sink** (*callable*) – A callable taking the form `func(req, resp, **kwargs)`.

> **Note**
>
> When using an async version of the `App`, this must be a coroutine function taking the form `func(req, resp, ws=None, **kwargs)`.
>
> Similar to *error handlers*, in the case of a WebSocket connection, the *resp* argument will be `None`, whereas the *ws* keyword argument will receive the *WebSocket* connection object.
>
> For backwards-compatibility, when *ws* is absent from the sink's signature, or a regex match (see **prefix** below) contains a group named 'ws', the *WebSocket* object is passed in place of the incompatible *resp*.
>
> This behavior will change in Falcon 5.0: when draining a WebSocket connection, *resp* will always be set to `None` regardless of the sink's signature.

Added in version 4.1: If an asynchronous sink callable explicitly defines a *ws* argument, it is used to pass the *WebSocket* connection object.

- **prefix** (*str*) – A regex string, typically starting with '/', which will trigger the sink if it matches the path portion of the request's URI. Both strings and precompiled regex objects may be specified. Characters are matched starting at the beginning of the URI path.

> **Note**
>
> Named groups are converted to kwargs and passed to the sink as such.

> **Warning**
>
> If the prefix overlaps a registered route template, the route will take precedence and mask the sink.
>
> (See also: *add_route()*)

ws_options: *WebSocketOptions*

A set of behavioral options related to WebSocket connections.

See also: *WebSocketOptions*.

Options

class falcon.**RequestOptions**

Defines a set of configurable request options.

An instance of this class is exposed via *falcon.App.req_options* and `falcon.asgi.App.req_options` for configuring certain *Request* and *falcon.asgi.Request* behaviors, respectively.

property auto_parse_form_urlencoded: bool

Set to `True` in order to automatically consume the request stream and merge the results into the request's query string params when the request's content type is `application/x-www-form-urlencoded` (default `False`).

Enabling this option for WSGI apps makes the form parameters accessible via *params*, *get_param()*, etc.

Deprecated since version 3.0: The *auto_parse_form_urlencoded* option is not supported for ASGI apps, and is considered deprecated for WSGI apps as of Falcon 3.0, in favor of accessing URL-encoded forms through *get_media()*.

The attribute and the auto-parsing functionality will be removed entirely in Falcon 5.0.

See also: *How can I access POSTed form params?*.

> **Warning**
>
> When this option is enabled, the request's body stream will be left at EOF. The original data is not retained by the framework.

> **Note**
>
> The character encoding for fields, before percent-encoding non-ASCII bytes, is assumed to be UTF-8. The special *_charset_* field is ignored if present.
>
> Falcon expects form-encoded request bodies to be encoded according to the standard W3C algorithm (see also https://html.spec.whatwg.org/multipage/form-control-infrastructure.html#application%2Fx-www-form-urlencoded-encoding-algorithm).

auto_parse_qs_csv: bool

Set to `True` to split query string values on any non-percent-encoded commas (default `False`).

When `False`, values containing commas are left as-is. In this mode, list items are taken only from multiples of the same parameter name within the query string (i.e. `t=1,2,3&t=4` becomes `['1,2,3', '4']`).

When *auto_parse_qs_csv* is set to `True`, the query string value is also split on non-percent-encoded commas and these items are added to the final list (i.e. `t=1,2,3&t=4,5` becomes `['1', '2', '3', '4', '5']`).

> **Warning**
>
> Enabling this option will cause the framework to misinterpret any JSON values that include literal (non-percent-encoded) commas. If the query string may include JSON, you can use JSON array syntax in lieu of CSV as a workaround.

default_media_type: str

The default media-type used to deserialize a request body, when the Content-Type header is missing or ambiguous.

This value is normally set to the media type provided to the *falcon.App* or *falcon.asgi.App* initializer; however, if created independently, this will default to `falcon.DEFAULT_MEDIA_TYPE`.

keep_blank_qs_values: bool

Set to `False` to ignore query string params that have missing or blank values (default `True`).

For comma-separated values, this option also determines whether or not empty elements in the parsed list are retained.

media_handlers: *Handlers*

A dict-like object for configuring the media-types to handle.

By default, handlers are provided for the `application/json`, `application/x-www-form-urlencoded` and `multipart/form-data` media types.

strip_url_path_trailing_slash: bool

Set to `True` in order to strip the trailing slash, if present, at the end of the URL path (default `False`).

When this option is enabled, the URL path is normalized by stripping the trailing slash character. This lets the application define a single route to a resource for a path that may or may not end in a forward slash.

However, this behavior can be problematic in certain cases, such as when working with authentication schemes that employ URL-based signatures.

class falcon.**ResponseOptions**

Defines a set of configurable response options.

An instance of this class is exposed via *falcon.App.resp_options* and falcon.asgi.App.resp_options for configuring certain *Response* behaviors.

default_media_type: str

The default Internet media type (RFC 2046) to use when rendering a response, when the Content-Type header is not set explicitly.

This value is normally set to the media type provided when a *falcon.App* is initialized; however, if created independently, this will default to falcon.DEFAULT_MEDIA_TYPE.

media_handlers: *Handlers*

A dict-like object for configuring the media-types to handle.

Default handlers are provided for the application/json, application/x-www-form-urlencoded and multipart/form-data media types.

secure_cookies_by_default: bool

Set to False in development environments to make the secure attribute for all cookies. (default True).

This can make testing easier by not requiring HTTPS. Note, however, that this setting can be overridden via *set_cookie()*'s secure kwarg.

static_media_types: dict[str, str]

A mapping of dot-prefixed file extensions to Internet media types (RFC 2046).

Defaults to mimetypes.types_map after calling mimetypes.init().

xml_error_serialization: bool

Set to False to disable automatic inclusion of the XML handler in the *default error serializer* (default True).

Enabling this option does not make Falcon automatically render all error responses in XML, but it is used only in the case the client prefers (via the Accept request header) XML to JSON and other configured media handlers.

> **Note**
>
> Falcon 5.0 will either change the default to False, or remove the automatic XML error serialization altogether.

> **Note**
>
> This option has no effect when a custom error serializer, set using *set_error_serializer()*, is in use.

Added in version 4.0.

class falcon.routing.**CompiledRouterOptions**

Defines a set of configurable router options.

An instance of this class is exposed via *falcon.App.router_options* and falcon.asgi.App.router_options for configuring certain *CompiledRouter* behaviors.

converters: ConverterDict

Represents the collection of named converters that may be referenced in URI template field expressions.

Adding additional converters is simply a matter of mapping an identifier to a converter class:

```
app.router_options.converters['mc'] = MyConverter
```

The identifier can then be used to employ the converter within a URI template:

```
app.add_route('/{some_field:mc}', some_resource)
```

Converter names may only contain ASCII letters, digits, and underscores, and must start with either a letter or an underscore.

> **Warning**
>
> Converter instances are shared between requests. Therefore, in threaded deployments, care must be taken to implement custom converters in a thread-safe manner.

(See also: *Field Converters*)

5.3.2 Request & Response

Similar to other frameworks, Falcon employs the inversion of control (IoC) pattern to coordinate with app methods in order to respond to HTTP requests. Resource responders, middleware methods, hooks, etc. receive a reference to the request and response objects that represent the current in-flight HTTP request. The app can use these objects to inspect the incoming HTTP request, and to manipulate the outgoing HTTP response.

Falcon uses different types to represent HTTP requests and responses for WSGI (`falcon.App`) vs. ASGI (`falcon.asgi.App`). However, the two interfaces are designed to be as similar as possible to minimize confusion and to facilitate porting.

(See also: *Routing*)

WSGI Request & Response

Instances of the `falcon.Request` and `falcon.Response` classes are passed into WSGI app responders as the second and third arguments, respectively:

```
import falcon


class Resource:

    def on_get(self, req, resp):
        resp.media = {'message': 'Hello world!'}
        resp.status = falcon.HTTP_200


# -- snip --


app = falcon.App()
app.add_route('/', Resource())
```

Request

class falcon.**Request**(*env: dict[str, Any]*, *options:* RequestOptions | *None* = *None*)

Represents a client's HTTP request.

> **Note**
>
> *Request* is not meant to be instantiated directly by responders.

Parameters

env (*dict*) – A WSGI environment dict passed in from the server. See also PEP-3333.

Keyword Arguments

options (RequestOptions) – Set of global options passed from the App handler.

property accept: str

Value of the Accept header, or `'*/*'` if the header is missing.

property access_route: list[str]

IP address of the original client, as well as any known addresses of proxies fronting the WSGI server.

The following request headers are checked, in order of preference, to determine the addresses:

- `Forwarded`
- `X-Forwarded-For`
- `X-Real-IP`

If none of these headers are available, the value of *remote_addr* is used instead.

> **Note**
>
> Per RFC 7239, the access route may contain "unknown" and obfuscated identifiers, in addition to IPv4 and IPv6 addresses

> **Warning**
>
> Headers can be forged by any client or proxy. Use this property with caution and validate all values before using them. Do not rely on the access route to authorize requests.

property app: str

Deprecated alias for *root_path*.

property auth: str | None

Value of the Authorization header, or `None` if the header is missing.

property bounded_stream: *BoundedStream*

File-like wrapper around *stream* to normalize certain differences between the native input objects employed by different WSGI servers. In particular, *bounded_stream* is aware of the expected Content-Length of the body, and will never block on out-of-bounds reads, assuming the client does not stall while transmitting the data to the server.

For example, the following will not block when Content-Length is 0 or the header is missing altogether:

```
data = req.bounded_stream.read()
```

This is also safe:

```
doc = json.load(req.bounded_stream)
```

client_accepts(*media_type: str*) → bool

Determine whether or not the client accepts a given media type.

Parameters
media_type (*str*) – An Internet media type to check.

Returns
`True` if the client has indicated in the Accept header that it accepts the specified media type. Otherwise, returns `False`.

Return type
bool

property client_accepts_json: bool

`True` if the Accept header indicates that the client is willing to receive JSON, otherwise `False`.

property client_accepts_msgpack: bool

`True` if the Accept header indicates that the client is willing to receive MessagePack, otherwise `False`.

property client_accepts_xml: bool

`True` if the Accept header indicates that the client is willing to receive XML, otherwise `False`.

client_prefers(*media_types: Iterable[str]*) → str | None

Return the client's preferred media type, given several choices.

Parameters
media_types (*iterable of str*) – One or more Internet media types from which to choose the client's preferred type. This value **must** be an iterable collection of strings.

Returns
The client's preferred media type, based on the Accept header. Returns `None` if the client does not accept any of the given types.

Return type
str

property content_length: int | None

Value of the Content-Length header converted to an `int`.

Returns `None` if the header is missing.

content_type: str | None

Value of the Content-Type header, or `None` if the header is missing.

context: *Context*

Empty object to hold any data (in its attributes) about the request which is specific to your app (e.g. session object). Falcon itself will not interact with this attribute after it has been initialized.

Note

The preferred way to pass request-specific data, when using the default context type, is to set attributes directly on the *context* object. For example:

```
req.context.role = 'trial'
req.context.user = 'guest'
```

context_type

alias of *Context*

property cookies: Mapping[str, str]

A dict of name/value cookie pairs.

The returned object should be treated as read-only to avoid unintended side-effects. If a cookie appears more than once in the request, only the first value encountered will be made available here.

See also: *get_cookie_values()* or *get_cookie_values()*.

property date: datetime | None

Value of the Date header, converted to a `datetime` instance.

The header value is assumed to conform to RFC 1123.

Changed in version 4.0: This property now returns timezone-aware `datetime` objects (or `None`).

env: dict[str, Any]

Reference to the WSGI environ `dict` passed in from the server. (See also PEP-3333.)

property expect: str | None

Value of the Expect header, or `None` if the header is missing.

property forwarded: list[*Forwarded*] | None

Value of the Forwarded header, as a parsed list of *falcon.Forwarded* objects, or `None` if the header is missing. If the header value is malformed, Falcon will make a best effort to parse what it can.

(See also: RFC 7239, Section 4)

property forwarded_host: str

Original host request header as received by the first proxy in front of the application server.

The following request headers are checked, in order of preference, to determine the forwarded scheme:

- `Forwarded`
- `X-Forwarded-Host`

If none of the above headers are available, or if the Forwarded header is available but the "host" parameter is not included in the first hop, the value of *host* is returned instead.

> **Note**
>
> Reverse proxies are often configured to set the Host header directly to the one that was originally requested by the user agent; in that case, using *host* is sufficient.

(See also: RFC 7239, Section 4)

property forwarded_prefix: str

The prefix of the original URI for proxied requests.

Uses *forwarded_scheme* and *forwarded_host* in order to reconstruct the original URI.

property forwarded_scheme: str

Original URL scheme requested by the user agent, if the request was proxied.

Typical values are 'http' or 'https'.

The following request headers are checked, in order of preference, to determine the forwarded scheme:

- `Forwarded`
- `X-Forwarded-For`

If none of these headers are available, or if the Forwarded header is available but does not contain a "proto" parameter in the first hop, the value of *scheme* is returned instead.

(See also: RFC 7239, Section 1)

property forwarded_uri: str

Original URI for proxied requests.

Uses *forwarded_scheme* and *forwarded_host* in order to reconstruct the original URI requested by the user agent.

get_cookie_values(*name: str*) → list[str] | None

Return all values provided in the Cookie header for the named cookie.

(See also: *Getting Cookies*)

Parameters
name (*str*) – Cookie name, case-sensitive.

Returns
Ordered list of all values specified in the Cookie header for the named cookie, or `None` if the cookie was not included in the request. If the cookie is specified more than once in the header, the returned list of values will preserve the ordering of the individual `cookie-pair`'s in the header.

Return type
list

get_header(*name: str*, *required: bool = False*, *default: str | None = None*) → str | None

Retrieve the raw string value for the given header.

Parameters
name (*str*) – Header name, case-insensitive (e.g., 'Content-Type')

Keyword Arguments

- **required** (*bool*) – Set to `True` to raise `HTTPBadRequest` instead of returning gracefully when the header is not found (default `False`).
- **default** (*any*) – Value to return if the header is not found (default `None`).

Returns
The value of the specified header if it exists, or the default value if the header is not found and is not required.

Return type
str

Raises
HTTPBadRequest – The header was not found in the request, but it was required.

get_header_as_datetime(*header: str*, *required: bool = False*, *obs_date: bool = False*) → datetime | None

Return an HTTP header with HTTP-Date values as a datetime.

Parameters
name (*str*) – Header name, case-insensitive (e.g., 'Date')

Keyword Arguments

- **required** (*bool*) – Set to `True` to raise `HTTPBadRequest` instead of returning gracefully when the header is not found (default `False`).
- **obs_date** (*bool*) – Support obs-date formats according to RFC 7231, e.g.: "Sunday, 06-Nov-94 08:49:37 GMT" (default `False`).

Returns
The value of the specified header if it exists, or `None` if the header is not found and is not required.

Return type
datetime

Raises

- ***HTTPBadRequest*** – The header was not found in the request, but it was required.
- **HttpInvalidHeader** – The header contained a malformed/invalid value.

Changed in version 4.0: This method now returns timezone-aware `datetime` objects.

get_header_as_int(*header: str*, *required: bool = False*) → int | None

Retrieve the int value for the given header.

Parameters

name (*str*) – Header name, case-insensitive (e.g., 'Content-Length')

Keyword Arguments

required (*bool*) – Set to `True` to raise `HTTPBadRequest` instead of returning gracefully when the header is not found (default `False`).

Returns

The value of the specified header if it exists, or `None` if the header is not found and is not required.

Return type

int

Raises

- ***HTTPBadRequest*** – The header was not found in the request, but it was required.
- **HttpInvalidHeader** – The header contained a malformed/invalid value.

Added in version 4.0.

get_media(*default_when_empty: Literal[_Unset.UNSET] | Any = _Unset.UNSET*) → Any

Return a deserialized form of the request stream.

The first time this method is called, the request stream will be deserialized using the Content-Type header as well as the media-type handlers configured via *falcon.RequestOptions*. The result will be cached and returned in subsequent calls:

```
deserialized_media = req.get_media()
```

If the matched media handler raises an error while attempting to deserialize the request body, the exception will propagate up to the caller.

See also *Media* for more information regarding media handling.

Note

When `get_media` is called on a request with an empty body, Falcon will let the media handler try to deserialize the body and will return the value returned by the handler or propagate the exception raised by it. To instead return a different value in case of an exception by the handler, specify the argument `default_when_empty`.

Warning

This operation will consume the request stream the first time it's called and cache the results. Follow-up calls will just retrieve a cached version of the object.

Parameters

default_when_empty – Fallback value to return when there is no body in the request and the media handler raises an error (like in the case of the default JSON media handler).

By default, Falcon uses the value returned by the media handler or propagates the raised exception, if any. This value is not cached, and will be used only for the current call.

Returns
The deserialized media representation.

Return type
media (object)

get_param(*name: str*, *required: bool = False*, *store: dict[str, Any] | None = None*, *default: str | None = None*) → str | None

Return the raw value of a query string parameter as a string.

Note

If an HTML form is POSTed to the API using the *application/x-www-form-urlencoded* media type, Falcon can automatically parse the parameters from the request body and merge them into the query string parameters. To enable this functionality, set `auto_parse_form_urlencoded` to `True` via `App.req_options`.

Note, however, that the `auto_parse_form_urlencoded` option is considered deprecated as of Falcon 3.0 in favor of accessing the URL-encoded form via `media`, and it may be removed in a future release.

See also: *How can I access POSTed form params?*

Note

Similar to the way multiple keys in form data are handled, if a query parameter is included in the query string multiple times, only one of those values will be returned, and it is undefined which one. This caveat also applies when `auto_parse_qs_csv` is enabled and the given parameter is assigned to a comma-separated list of values (e.g., `foo=a,b,c`).

When multiple values are expected for a parameter, `get_param_as_list()` can be used to retrieve all of them at once.

Parameters
name (*str*) – Parameter name, case-sensitive (e.g., 'sort').

Keyword Arguments

- **required** (*bool*) – Set to `True` to raise `HTTPBadRequest` instead of returning `None` when the parameter is not found (default `False`).
- **store** (*dict*) – A `dict`-like object in which to place the value of the param, but only if the param is present.
- **default** (*any*) – If the param is not found returns the given value instead of `None`

Returns
The value of the param as a string, or `None` if param is not found and is not required.

Return type
str

Raises
HTTPBadRequest – A required param is missing from the request.

get_param_as_bool(*name: str*, *required: bool = False*, *store: dict[str, Any] | None = None*, *blank_as_true: bool = True*, *default: bool | None = None*) → bool | None

Return the value of a query string parameter as a boolean.

This method treats valueless parameters as flags. By default, if no value is provided for the parameter in the query string, `True` is assumed and returned. If the parameter is missing altogether, `None` is returned as with other `get_param_*()` methods, which can be easily treated as falsy by the caller as needed.

The following boolean strings are supported:

```
TRUE_STRINGS = ('true', 'True', 't', 'yes', 'y', '1', 'on')
FALSE_STRINGS = ('false', 'False', 'f', 'no', 'n', '0', 'off')
```

Parameters
name (*str*) – Parameter name, case-sensitive (e.g., 'detailed').

Keyword Arguments

- **required** (*bool*) – Set to `True` to raise `HTTPBadRequest` instead of returning `None` when the parameter is not found or is not a recognized boolean string (default `False`).
- **store** (*dict*) – A `dict`-like object in which to place the value of the param, but only if the param is found (default `None`).
- **blank_as_true** (*bool*) – Valueless query string parameters are treated as flags, resulting in `True` being returned when such a parameter is present, and `False` otherwise. To require the client to explicitly opt-in to a truthy value, pass `blank_as_true=False` to return `False` when a value is not specified in the query string.
- **default** (*any*) – If the param is not found, return this value instead of `None`.

Returns
The value of the param if it is found and can be converted to a `bool`. If the param is not found, returns `None` unless *required* is `True`.

Return type
bool

Raises
HTTPBadRequest – A required param is missing from the request, or can not be converted to a `bool`.

get_param_as_date(*name: str*, *format_string: str = '%Y-%m-%d'*, *required: bool = False*, *store: dict[str, Any] | None = None*, *default: date | None = None*) → date | None

Return the value of a query string parameter as a date.

Parameters
name (*str*) – Parameter name, case-sensitive (e.g., 'ids').

Keyword Arguments

- **format_string** (*str*) – String used to parse the param value into a date. Any format recognized by strptime() is supported (default `"%Y-%m-%d"`).
- **required** (*bool*) – Set to `True` to raise `HTTPBadRequest` instead of returning `None` when the parameter is not found (default `False`).
- **store** (*dict*) – A `dict`-like object in which to place the value of the param, but only if the param is found (default `None`).
- **default** (*any*) – If the param is not found returns the given value instead of `None`

Returns
The value of the param if it is found and can be converted to a `date` according to the supplied format string. If the param is not found, returns `None` unless required is `True`.

Return type
datetime.date

Raises
HTTPBadRequest – A required param is missing from the request, or the value could not be converted to a `date`.

get_param_as_datetime(*name: str*, *format_string: str = '%Y-%m-%dT%H:%M:%S%z'*, *required: bool = False*, *store: dict[str, Any] | None = None*, *default: datetime | None = None*) → datetime | None

Return the value of a query string parameter as a datetime.

Parameters
name (*str*) – Parameter name, case-sensitive (e.g., 'ids').

Keyword Arguments

- **format_string** (*str*) – String used to parse the param value into a `datetime`. Any format recognized by strptime() is supported (default `'%Y-%m-%dT%H:%M:%S%z'`).
- **required** (*bool*) – Set to `True` to raise `HTTPBadRequest` instead of returning `None` when the parameter is not found (default `False`).
- **store** (*dict*) – A `dict`-like object in which to place the value of the param, but only if the param is found (default `None`).
- **default** (*any*) – If the param is not found returns the given value instead of `None`

Returns
The value of the param if it is found and can be converted to a `datetime` according to the supplied format string. If the param is not found, returns `None` unless required is `True`.

Return type
datetime.datetime

Raises
HTTPBadRequest – A required param is missing from the request, or the value could not be converted to a `datetime`.

Changed in version 4.0: The default value of *format_string* was changed from `'%Y-%m-%dT%H:%M:%SZ'` to `'%Y-%m-%dT%H:%M:%S%z'`.

The new format is a superset of the old one parsing-wise, however, the converted `datetime` object is now timezone-aware.

get_param_as_float(*name: str*, *required: bool = False*, *min_value: float | None = None*, *max_value: float | None = None*, *store: dict[str, Any] | None = None*, *default: float | None = None*) → float | None

Return the value of a query string parameter as an float.

Parameters
name (*str*) – Parameter name, case-sensitive (e.g., 'limit').

Keyword Arguments

- **required** (*bool*) – Set to `True` to raise `HTTPBadRequest` instead of returning `None` when the parameter is not found or is not an float (default `False`).
- **min_value** (*float*) – Set to the minimum value allowed for this param. If the param is found and it is less than min_value, an `HTTPError` is raised.
- **max_value** (*float*) – Set to the maximum value allowed for this param. If the param is found and its value is greater than max_value, an `HTTPError` is raised.
- **store** (*dict*) – A `dict`-like object in which to place the value of the param, but only if the param is found (default `None`).
- **default** (*any*) – If the param is not found returns the given value instead of `None`

Returns
The value of the param if it is found and can be converted to an `float`. If the param is not found, returns `None`, unless *required* is `True`.

Return type
float

Raises
HTTPBadRequest – The param was not found in the request, even though it was required to be there, or it was found but could not be converted to an `float`. Also raised if the param's value falls outside the given interval, i.e., the value must be in the interval: min_value <= value <= max_value to avoid triggering an error.

get_param_as_int(*name: str*, *required: bool = False*, *min_value: int | None = None*, *max_value: int | None = None*, *store: dict[str, Any] | None = None*, *default: int | None = None*) → int | None

Return the value of a query string parameter as an int.

Parameters
name (*str*) – Parameter name, case-sensitive (e.g., 'limit').

Keyword Arguments

- **required** (*bool*) – Set to `True` to raise `HTTPBadRequest` instead of returning `None` when the parameter is not found or is not an integer (default `False`).
- **min_value** (*int*) – Set to the minimum value allowed for this param. If the param is found and it is less than min_value, an `HTTPError` is raised.
- **max_value** (*int*) – Set to the maximum value allowed for this param. If the param is found and its value is greater than max_value, an `HTTPError` is raised.
- **store** (*dict*) – A `dict`-like object in which to place the value of the param, but only if the param is found (default `None`).
- **default** (*any*) – If the param is not found returns the given value instead of `None`

Returns
The value of the param if it is found and can be converted to an `int`. If the param is not found, returns `None`, unless *required* is `True`.

Return type
int

Raises
HTTPBadRequest – The param was not found in the request, even though it was required to be there, or it was found but could not be converted to an `int`. Also raised if the param's value falls outside the given interval, i.e., the value must be in the interval: min_value <= value <= max_value to avoid triggering an error.

get_param_as_json(*name: str*, *required: bool = False*, *store: dict[str, Any] | None = None*, *default: Any | None = None*) → Any

Return the decoded JSON value of a query string parameter.

Given a JSON value, decode it to an appropriate Python type, (e.g., `dict`, `list`, `str`, `int`, `bool`, etc.)

> **Warning**
>
> If the *auto_parse_qs_csv* option is set to `True` (default `False`), the framework will misinterpret any JSON values that include literal (non-percent-encoded) commas. If the query string may include JSON, you can use JSON array syntax in lieu of CSV as a workaround.

Parameters
name (*str*) – Parameter name, case-sensitive (e.g., 'payload').

Keyword Arguments

- **required** (*bool*) – Set to `True` to raise `HTTPBadRequest` instead of returning `None` when the parameter is not found (default `False`).
- **store** (*dict*) – A `dict`-like object in which to place the value of the param, but only if the param is found (default `None`).
- **default** (*any*) – If the param is not found returns the given value instead of `None`

Returns
The value of the param if it is found. Otherwise, returns `None` unless required is `True`.

Return type
dict

Raises
HTTPBadRequest – A required param is missing from the request, or the value could not be parsed as JSON.

get_param_as_list(*name: str*, *transform: Callable[[str], _T] | None = None*, *required: bool = False*, *store: dict[str, Any] | None = None*, *default: list[_T] | None = None*) → list[_T] | list[str] | None

Return the value of a query string parameter as a list.

List items must be comma-separated or must be provided as multiple instances of the same param in the query string ala *application/x-www-form-urlencoded*.

> **Note**
>
> To enable the interpretation of comma-separated parameter values, the *auto_parse_qs_csv* option must be set to `True` (default `False`).

Parameters
name (*str*) – Parameter name, case-sensitive (e.g., 'ids').

Keyword Arguments

- **transform** (*callable*) – An optional transform function that takes as input each element in the list as a `str` and outputs a transformed element for inclusion in the list that will be returned. For example, passing `int` will transform list items into numbers.
- **required** (*bool*) – Set to `True` to raise `HTTPBadRequest` instead of returning `None` when the parameter is not found (default `False`).
- **store** (*dict*) – A `dict`-like object in which to place the value of the param, but only if the param is found (default `None`).
- **default** (*any*) – If the param is not found returns the given value instead of `None`

Returns

The value of the param if it is found. Otherwise, returns `None` unless *required* is `True`.

Empty list elements will be included by default, but this behavior can be configured by setting the *keep_blank_qs_values* option. For example, by default the following query strings would both result in `['1', '', '3']`:

```
things=1&things=&things=3
things=1,,3
```

Note, however, that for the second example string above to be interpreted as a list, the *auto_parse_qs_csv* option must be set to `True`.

Return type
list

Raises
HTTPBadRequest – A required param is missing from the request, or a transform function raised an instance of `ValueError`.

get_param_as_uuid(*name: str*, *required: bool = False*, *store: dict[str, Any] | None = None*, *default: UUID | None = None*) → UUID | None

Return the value of a query string parameter as an UUID.

The value to convert must conform to the standard UUID string representation per RFC 4122. For example, the following strings are all valid:

```
# Lowercase
'64be949b-3433-4d36-a4a8-9f19d352fee8'

# Uppercase
'BE71ECAA-F719-4D42-87FD-32613C2EEB60'

# Mixed
'81c8155C-D6de-443B-9495-39Fa8FB239b5'
```

Parameters
name (*str*) – Parameter name, case-sensitive (e.g., 'id').

Keyword Arguments

- **required** (*bool*) – Set to `True` to raise `HTTPBadRequest` instead of returning `None` when the parameter is not found or is not a UUID (default `False`).
- **store** (*dict*) – A `dict`-like object in which to place the value of the param, but only if the param is found (default `None`).
- **default** (*any*) – If the param is not found returns the given value instead of `None`

Returns
The value of the param if it is found and can be converted to a `UUID`. If the param is not found, returns `default` (default `None`), unless *required* is `True`.

Return type
UUID

Raises
HTTPBadRequest – The param was not found in the request, even though it was required to be there, or it was found but could not be converted to a `UUID`.

has_param(*name: str*) → bool

Determine whether or not the query string parameter already exists.

Parameters
name (*str*) – Parameter name, case-sensitive (e.g., 'sort').

Returns
`True` if param is found, or `False` if param is not found.

Return type
bool

property headers: Mapping[str, str]

Raw HTTP headers from the request with dash-separated names normalized to uppercase.

> **Note**
>
> This property differs from the ASGI version of `Request.headers` in that the latter returns *lowercase* names. Middleware, such as tracing and logging components, that need to be compatible with both WSGI and ASGI apps should use *`headers_lower`* instead.

> **Warning**
>
> Parsing all the headers to create this dict is done the first time this attribute is accessed, and the returned object should be treated as read-only. Note that this parsing can be costly, so unless you need all the headers in this format, you should instead use the `get_header()` method or one of the convenience attributes to get a value for a specific header.

property headers_lower: Mapping[str, str]

Same as *`headers`* except header names are normalized to lowercase.

Added in version 4.0.

property host: str

Host request header field.

property if_match: list[*ETag* | Literal['*']] | None

Value of the If-Match header, as a parsed list of *`falcon.ETag`* objects or `None` if the header is missing or its value is blank.

This property provides a list of all `entity-tags` in the header, both strong and weak, in the same order as listed in the header.

(See also: RFC 7232, Section 3.1)

property if_modified_since: datetime | None

Value of the If-Modified-Since header.

Returns `None` if the header is missing.

Changed in version 4.0: This property now returns timezone-aware `datetime` objects (or `None`).

property if_none_match: list[*ETag* | Literal['*']] | None

Value of the If-None-Match header, as a parsed list of *`falcon.ETag`* objects or `None` if the header is missing or its value is blank.

This property provides a list of all `entity-tags` in the header, both strong and weak, in the same order as listed in the header.

(See also: RFC 7232, Section 3.2)

property if_range: str | None

Value of the If-Range header, or `None` if the header is missing.

property if_unmodified_since: datetime | None

Value of the If-Unmodified-Since header.

Returns `None` if the header is missing.

Changed in version 4.0: This property now returns timezone-aware `datetime` objects (or `None`).

is_websocket: bool

Always `False` in a sync `Request`.

log_error(*message: str*) → None

Write an error message to the server's log.

Prepends timestamp and request info to message, and writes the result out to the WSGI server's error stream (*wsgi.error*).

Parameters

message (*str*) – Description of the problem.

property media: Any

Property that acts as an alias for `get_media()`. This alias provides backwards-compatibility for apps that were built for versions of the framework prior to 3.0:

```
# Equivalent to: deserialized_media = req.get_media()
deserialized_media = req.media
```

New WSGI apps are encouraged to use `get_media()` directly instead of this property.

method: str

HTTP method requested, uppercase (e.g., `'GET'`, `'POST'`, etc.)

property netloc: str

port" portion of the request URL.

The port may be omitted if it is the default one for the URL's schema (80 for HTTP and 443 for HTTPS).

Type

Returns the "host

options: *RequestOptions*

Set of global options passed from the App handler.

property params: Mapping[str, str | list[str]]

The mapping of request query parameter names to their values.

Where the parameter appears multiple times in the query string, the value mapped to that parameter key will be a list of all the values in the order seen.

path: str

Path portion of the request URI (not including query string).

Warning

If this attribute is to be used by the app for any upstream requests, any non URL-safe characters in the path must be URL encoded back before making the request.

Note

`req.path` may be set to a new value by a `process_request()` middleware method in order to influence routing. If the original request path was URL encoded, it will be decoded before being returned by this attribute.

property port: int

Port used for the request.

If the Host header is present in the request, but does not specify a port, the default one for the given schema is returned (80 for HTTP and 443 for HTTPS). If the request does not include a Host header, the listening port for the server is returned instead.

property prefix: str

The prefix of the request URI, including scheme, host, and app *root_path* (if any).

query_string: str

Query string portion of the request URI, without the preceding '?' character.

property range: tuple[int, int] | None

A 2-member `tuple` parsed from the value of the Range header, or `None` if the header is missing.

The two members correspond to the first and last byte positions of the requested resource, inclusive. Negative indices indicate offset from the end of the resource, where -1 is the last byte, -2 is the second-to-last byte, and so forth.

Only continuous ranges are supported (e.g., "bytes=0-0,-1" would result in an HTTPBadRequest exception when the attribute is accessed).

property range_unit: str | None

Unit of the range parsed from the value of the Range header.

Returns `None` if the header is missing.

property referer: str | None

Value of the Referer header, or `None` if the header is missing.

property relative_uri: str

The path and query string portion of the request URI, omitting the scheme and host.

property remote_addr: str

IP address of the closest client or proxy to the WSGI server.

This property is determined by the value of `REMOTE_ADDR` in the WSGI environment dict. Since this address is not derived from an HTTP header, clients and proxies can not forge it.

> **Note**
>
> If your application is behind one or more reverse proxies, you can use *access_route* to retrieve the real IP address of the client.

property root_path: str

The initial portion of the request URI's path that corresponds to the application object, so that the application knows its virtual "location". This may be an empty string, if the application corresponds to the "root" of the server.

(In WSGI it corresponds to the "SCRIPT_NAME" environ variable defined by PEP-3333; in ASGI it Corresponds to the "root_path"ASGI HTTP scope field.)

property scheme: str

URL scheme used for the request. Either 'http' or 'https'.

> **Note**
>
> If the request was proxied, the scheme may not match what was originally requested by the client. *forwarded_scheme* can be used, instead, to handle such cases.

stream: *ReadableIO*

File-like input object for reading the body of the request, if any. This object provides direct access to the server's data stream and is non-seekable. In order to avoid unintended side effects, and to provide maximum flexibility to the application, Falcon itself does not buffer or spool the data in any way.

Since this object is provided by the WSGI server itself, rather than by Falcon, it may behave differently depending on how you host your app. For example, attempting to read more bytes than are expected (as

determined by the Content-Length header) may or may not block indefinitely. It's a good idea to test your WSGI server to find out how it behaves.

This can be particularly problematic when a request body is expected, but none is given. In this case, the following call blocks under certain WSGI servers:

```
# Blocks if Content-Length is 0
data = req.stream.read()
```

The workaround is fairly straightforward, if verbose:

```
# If Content-Length happens to be 0, or the header is
# missing altogether, this will not block.
data = req.stream.read(req.content_length or 0)
```

Alternatively, when passing the stream directly to a consumer, it may be necessary to branch off the value of the Content-Length header:

```
if req.content_length:
    doc = json.load(req.stream)
```

For a slight performance cost, you may instead wish to use *bounded_stream*, which wraps the native WSGI input object to normalize its behavior.

> **Note**
>
> If an HTML form is POSTed to the API using the *application/x-www-form-urlencoded* media type, and the *auto_parse_form_urlencoded* option is set, the framework will consume *stream* in order to parse the parameters and merge them into the query string parameters. In this case, the stream will be left at EOF.

property subdomain: str | None

Leftmost (i.e., most specific) subdomain from the hostname.

If only a single domain name is given, *subdomain* will be `None`.

> **Note**
>
> If the hostname in the request is an IP address, the value for *subdomain* is undefined.

property uri: str

The fully-qualified URI for the request.

uri_template: str | None

The template for the route that was matched for this request. May be `None` if the request has not yet been routed, as would be the case for `process_request()` middleware methods. May also be `None` if your app uses a custom routing engine and the engine does not provide the URI template when resolving a route.

property url: str

Alias for *Request.uri*.

property user_agent: str | None

Value of the User-Agent header, or `None` if the header is missing.

class falcon.Forwarded

Represents a parsed Forwarded header.

(See also: RFC 7239, Section 4)

dest: str | None

The value of the "by" parameter, or `None` if the parameter is absent.

Identifies the client-facing interface of the proxy.

host: str | None

The value of the "host" parameter, or `None` if the parameter is absent.

Provides the host request header field as received by the proxy.

scheme: str | None

The value of the "proto" parameter, or `None` if the parameter is absent.

Indicates the protocol that was used to make the request to the proxy.

src: str | None

The value of the "for" parameter, or `None` if the parameter is absent.

Identifies the node making the request to the proxy.

class falcon.stream.**BoundedStream**(*stream: BinaryIO*, *stream_len: int*)

Wrap *wsgi.input* streams to make them more robust.

`socket._fileobject` and `io.BufferedReader` are sometimes used to implement *wsgi.input*. However, app developers are often burned by the fact that the *read()* method for these objects block indefinitely if either no size is passed, or a size greater than the request's content length is passed to the method.

This class normalizes *wsgi.input* behavior between WSGI servers by implementing non-blocking behavior for the cases mentioned above. The caller is not allowed to read more than the number of bytes specified by the Content-Length header in the request.

Parameters

- **stream** – Instance of `socket._fileobject` from `environ['wsgi.input']`
- **stream_len** – Expected content length of the stream.

property eof: bool

`True` if there is no more data to read from the stream, otherwise `False`.

exhaust(*chunk_size: int = 65536*) → None

Exhaust the stream.

This consumes all the data left until the limit is reached.

Parameters

chunk_size (*int*) – The size for a chunk (default: 64 KB). It will read the chunk until the stream is exhausted.

property is_exhausted: bool

Deprecated alias for *eof*.

next() → bytes

Implement next(self).

read(*size: int | None = None*) → bytes

Read from the stream.

Parameters

size (*int*) – Maximum number of bytes/characters to read. Defaults to reading until EOF.

Returns

Data read from the stream.

Return type

bytes

readable() → bool

Return `True` always.

readline(*limit: int | None = None*) → bytes

Read a line from the stream.

Parameters
limit (*int*) – Maximum number of bytes/characters to read. Defaults to reading until EOF.

Returns
Data read from the stream.

Return type
bytes

readlines(*hint: int | None = None*) → list[bytes]

Read lines from the stream.

Parameters
hint (*int*) – Maximum number of bytes/characters to read. Defaults to reading until EOF.

Returns
Data read from the stream.

Return type
bytes

seekable() → bool

Return `False` always.

writable() → bool

Return `False` always.

write(*data: bytes*) → None

Raise OSError always; writing is not supported.

Response

class falcon.**Response**(*options: ResponseOptions | None = None*)

Represents an HTTP response to a client request.

> **Note**
> `Response` is not meant to be instantiated directly by responders.

Keyword Arguments
options (`ResponseOptions`) – Set of global options passed from the App handler.

property accept_ranges: str | None

Set the Accept-Ranges header.

The Accept-Ranges header field indicates to the client which range units are supported (e.g. “bytes”) for the target resource.

If range requests are not supported for the target resource, the header may be set to “none” to advise the client not to attempt any such requests.

> **Note**
>
> "none" is the literal string, not Python's built-in `None` type.

append_header(*name: str*, *value: str*) → None

Set or append a header for this response.

If the header already exists, the new value will normally be appended to it, delimited by a comma. The notable exception to this rule is Set-Cookie, in which case a separate header line for each value will be included in the response.

> **Note**
>
> While this method can be used to efficiently append raw Set-Cookie headers to the response, you may find `set_cookie()` to be more convenient.

Parameters

- **name** (*str*) – Header name (case-insensitive). The name may contain only US-ASCII characters.
- **value** (*str*) – Value for the header. As with the header's name, the value may contain only US-ASCII characters.

append_link(*target: str*, *rel: str*, *title: str | None = None*, *title_star: tuple[str, str] | None = None*, *anchor: str | None = None*, *hreflang: str | Iterable[str] | None = None*, *type_hint: str | None = None*, *crossorigin: str | None = None*, *link_extension: Iterable[tuple[str, str]] | None = None*) → None

Append a link header to the response.

(See also: RFC 5988, Section 1)

> **Note**
>
> Calling this method repeatedly will cause each link to be appended to the Link header value, separated by commas.

Parameters

- **target** (*str*) – Target IRI for the resource identified by the link. Will be converted to a URI, if necessary, per RFC 3987, Section 3.1.
- **rel** (*str*) – Relation type of the link, such as "next" or "bookmark".

  (See also: http://www.iana.org/assignments/link-relations/link-relations.xhtml)

Keyword Arguments

- **title** (*str*) – Human-readable label for the destination of the link (default `None`). If the title includes non-ASCII characters, you will need to use *title_star* instead, or provide both a US-ASCII version using *title* and a Unicode version using *title_star*.
- **title_star** (*tuple[str, str]*) – Localized title describing the destination of the link (default `None`). The value must be a two-member tuple in the form of (*language-tag*, *text*), where *language-tag* is a standard language identifier as defined in RFC 5646, Section 2.1, and *text* is a Unicode string.

> **Note**
>
> *language-tag* may be an empty string, in which case the client will assume the language from the general context of the current request.

> **Note**
>
> *text* will always be encoded as UTF-8.

- **anchor** (*str*) – Override the context IRI with a different URI (default None). By default, the context IRI for the link is simply the IRI of the requested resource. The value provided may be a relative URI.
- **hreflang** (*str or iterable*) – Either a single *language-tag*, or a `list` or `tuple` of such tags to provide a hint to the client as to the language of the result of following the link. A list of tags may be given in order to indicate to the client that the target resource is available in multiple languages.
- **type_hint** (*str*) – Provides a hint as to the media type of the result of dereferencing the link (default `None`). As noted in RFC 5988, this is only a hint and does not override the Content-Type header returned when the link is followed.
- **crossorigin** (*str*) – Determines how cross origin requests are handled. Can take values 'anonymous' or 'use-credentials' or None. (See: https://www.w3.org/TR/html50/infrastructure.html#cors-settings-attribute)
- **link_extension** – Provides additional custom attributes, as described in RFC 8288, Section 3.4.2; each member of the iterable must be a two-tuple in the form of (*param*, *value*).

property cache_control: str | None

Set the Cache-Control header.

Used to set a list of cache directives to use as the value of the Cache-Control header. The list will be joined with ", " to produce the value for the header.

complete: bool = False

Set to `True` from within a middleware method to signal to the framework that request processing should be short-circuited (see also *Middleware*).

property content_length: str | None

Set the Content-Length header.

This property can be used for responding to HEAD requests when you aren't actually providing the response body, or when streaming the response. If either the *text* property or the *data* property is set on the response, the framework will force Content-Length to be the length of the given text bytes. Therefore, it is only necessary to manually set the content length when those properties are not used.

> **Note**
>
> In cases where the response content is a stream (readable file-like object), Falcon will not supply a Content-Length header to the server unless *content_length* is explicitly set. Consequently, the server may choose to use chunked encoding in this case.

property content_location: str | None

Set the Content-Location header.

This value will be URI encoded per RFC 3986. If the value that is being set is already URI encoded it should be decoded first or the header should be set manually using the set_header method.

property content_range: str | None

A tuple to use in constructing a value for the Content-Range header.

The tuple has the form (*start*, *end*, *length*, [*unit*]), where *start* and *end* designate the range (inclusive), and *length* is the total length, or '*' if unknown. You may pass `int`'s for these numbers (no need to convert to `str` beforehand). The optional value *unit* describes the range unit and defaults to 'bytes'

> **Note**
>
> You only need to use the alternate form, 'bytes */1234', for responses that use the status '416 Range Not Satisfiable'. In this case, raising `falcon.HTTPRangeNotSatisfiable` will do the right thing.

(See also: RFC 7233, Section 4.2)

property content_type: str | None

Sets the Content-Type header.

The `falcon` module provides a number of constants for common media types, including `falcon.MEDIA_JSON`, `falcon.MEDIA_MSGPACK`, `falcon.MEDIA_YAML`, `falcon.MEDIA_XML`, `falcon.MEDIA_HTML`, `falcon.MEDIA_JS`, `falcon.MEDIA_TEXT`, `falcon.MEDIA_JPEG`, `falcon.MEDIA_PNG`, and `falcon.MEDIA_GIF`.

context: *structures.Context*

Empty object to hold any data (in its attributes) about the response which is specific to your app (e.g. session object). Falcon itself will not interact with this attribute after it has been initialized.

> **Note**
>
> The preferred way to pass response-specific data, when using the default context type, is to set attributes directly on the *context* object. For example:
>
> ```
> resp.context.cache_strategy = 'lru'
> ```

context_type

alias of *Context*

property data: bytes | None

Byte string representing response content.

Use this attribute in lieu of *text* when your content is already a byte string (of type `bytes`). See also the note below.

> **Warning**
>
> Always use the *text* attribute for text, or encode it first to `bytes` when using the *data* attribute, to ensure Unicode characters are properly encoded in the HTTP response.

delete_header(*name: str*) → None

Delete a header that was previously set for this response.

If the header was not previously set, nothing is done (no error is raised). Otherwise, all values set for the header will be removed from the response.

Note that calling this method is equivalent to setting the corresponding header property (when said property is available) to `None`. For example:

```
resp.etag = None
```

Warning

This method cannot be used with the Set-Cookie header. Instead, use *unset_cookie()* to remove a cookie and ensure that the user agent expires its own copy of the data as well.

Parameters

name (*str*) – Header name (case-insensitive). The name may contain only US-ASCII characters.

Raises

ValueError – *name* cannot be `'Set-Cookie'`.

property downloadable_as: str | None

Set the Content-Disposition header using the given filename.

The value will be used for the `filename` directive. For example, given `'report.pdf'`, the Content-Disposition header would be set to: `'attachment; filename="report.pdf"'`.

As per RFC 6266 recommendations, non-ASCII filenames will be encoded using the `filename*` directive, whereas `filename` will contain the US ASCII fallback.

property etag: str | None

Set the ETag header.

The ETag header will be wrapped with double quotes `"value"` in case the user didn't pass it.

property expires: str | None

Set the Expires header. Set to a `datetime` (UTC) instance.

Note

Falcon will format the `datetime` as an HTTP date string.

get_header(*name: str*, *default: str | None = None*) → str | None

Retrieve the raw string value for the given header.

Normally, when a header has multiple values, they will be returned as a single, comma-delimited string. However, the Set-Cookie header does not support this format, and so attempting to retrieve it will raise an error.

Parameters

name (*str*) – Header name, case-insensitive. Must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

Keyword Arguments

default – Value to return if the header is not found (default `None`).

Raises

ValueError – The value of the 'Set-Cookie' header(s) was requested.

Returns

The value of the specified header if set, or the default value if not set.

Return type

str

property headers: dict[str, str]

Copy of all headers set for the response, without cookies.

Note that a new copy is created and returned each time this property is referenced.

property last_modified: str | None

Set the Last-Modified header. Set to a `datetime` (UTC) instance.

> **Note**
>
> Falcon will format the `datetime` as an HTTP date string.

property location: str | None

Set the Location header.

This value will be URI encoded per RFC 3986. If the value that is being set is already URI encoded it should be decoded first or the header should be set manually using the set_header method.

property media: Any

A serializable object supported by the media handlers configured via *falcon.RequestOptions*.

> **Note**
>
> See also *Media* for more information regarding media handling.

options: *ResponseOptions*

Set of global options passed in from the App handler.

render_body() → bytes | None

Get the raw bytestring content for the response body.

This method returns the raw data for the HTTP response body, taking into account the *text*, *data*, and *media* attributes.

> **Note**
>
> This method ignores *stream*; the caller must check and handle that attribute directly.

Returns

The UTF-8 encoded value of the *text* attribute, if set. Otherwise, the value of the *data* attribute if set, or finally the serialized value of the *media* attribute. If none of these attributes are set, `None` is returned.

Return type

bytes

property retry_after: str | None

Set the Retry-After header.

The expected value is an integral number of seconds to use as the value for the header. The HTTP-date syntax is not supported.

set_cookie(*name: str*, *value: str*, *expires: datetime | None = None*, *max_age: int | None = None*, *domain: str | None = None*, *path: str | None = None*, *secure: bool | None = None*, *http_only: bool = True*, *same_site: str | None = None*, *partitioned: bool = False*) → None

Set a response cookie.

Note

This method can be called multiple times to add one or more cookies to the response.

See also

To learn more about setting cookies, see *Setting Cookies*. The parameters listed below correspond to those defined in RFC 6265.

Parameters

- **name** (*str*) – Cookie name
- **value** (*str*) – Cookie value

Keyword Arguments

- **expires** (*datetime*) – Specifies when the cookie should expire. By default, cookies expire when the user agent exits.

  (See also: RFC 6265, Section 4.1.2.1)

- **max_age** (*int*) – Defines the lifetime of the cookie in seconds. By default, cookies expire when the user agent exits. If both *max_age* and *expires* are set, the latter is ignored by the user agent.

  Note

  Coercion to `int` is attempted if provided with `float` or `str`.

  (See also: RFC 6265, Section 4.1.2.2)

- **domain** (*str*) – Restricts the cookie to a specific domain and any subdomains of that domain. By default, the user agent will return the cookie only to the origin server. When overriding this default behavior, the specified domain must include the origin server. Otherwise, the user agent will reject the cookie.

  Note

  Cookies do not provide isolation by port, so the domain should not provide one. (See also: RFC 6265, Section 8.5)

  (See also: RFC 6265, Section 4.1.2.3)

- **path** (*str*) – Scopes the cookie to the given path plus any subdirectories under that path (the "/" character is interpreted as a directory separator). If the cookie does not specify a path, the user agent defaults to the path component of the requested URI.

  Warning

  User agent interfaces do not always isolate cookies by path, and so this should not be considered an effective security measure.

  (See also: RFC 6265, Section 4.1.2.4)

- **secure** (*bool*) – Direct the client to only return the cookie in subsequent requests if they are made over HTTPS (default: `True`). This prevents attackers from reading sensitive cookie data.

  Note

  The default value for this argument is normally `True`, but can be modified by setting *secure_cookies_by_default* via *App.resp_options*.

  Warning

  For the *secure* cookie attribute to be effective, your application will need to enforce HTTPS.

  (See also: RFC 6265, Section 4.1.2.5)

- **http_only** (*bool*) – The HttpOnly attribute limits the scope of the cookie to HTTP requests. In particular, the attribute instructs the user agent to omit the cookie when providing access to cookies via "non-HTTP" APIs. This is intended to mitigate some forms of cross-site scripting. (default: `True`)

  Note

  HttpOnly cookies are not visible to javascript scripts in the browser. They are automatically sent to the server on javascript `XMLHttpRequest` or `Fetch` requests.

  (See also: RFC 6265, Section 4.1.2.6)

- **same_site** (*str*) – Helps protect against CSRF attacks by restricting when a cookie will be attached to the request by the user agent. When set to `'Strict'`, the cookie will only be sent along with "same-site" requests. If the value is `'Lax'`, the cookie will be sent with same-site requests, and with "cross-site" top-level navigations. If the value is `'None'`, the cookie will be sent with same-site and cross-site requests. Finally, when this attribute is not set on the cookie, the attribute will be treated as if it had been set to `'None'`.

  (See also: Same-Site RFC Draft)

- **partitioned** (*bool*) – Prevents cookies from being accessed from other subdomains. With partitioned enabled, a cookie set by https://3rd-party.example which is embedded inside https://site-a.example can no longer be accessed by https://site-b.example. While this attribute is not yet standardized, it is already used by Chrome.

  (See also: CHIPS)

  Added in version 4.0.

Raises

- **KeyError** – *name* is not a valid cookie name.
- **ValueError** – *value* is not a valid cookie value.

set_header(*name: str*, *value: str*) → None

Set a header for this response to a given value.

> **Warning**
>
> Calling this method overwrites any values already set for this header. To append an additional value for this header, use *append_header()* instead.

> **Warning**
>
> This method cannot be used to set cookies; instead, use *append_header()* or *set_cookie()*.

Parameters

- **name** (*str*) – Header name (case-insensitive). The name may contain only US-ASCII characters.
- **value** (*str*) – Value for the header. As with the header's name, the value may contain only US-ASCII characters.

Raises

ValueError – *name* cannot be `'Set-Cookie'`.

set_headers(*headers: Mapping[str, str] | Iterable[tuple[str, str]]*) → None

Set several headers at once.

This method can be used to set a collection of raw header names and values all at once.

> **Warning**
>
> Calling this method overwrites any existing values for the given header. If a list containing multiple instances of the same header is provided, only the last value will be used. To add multiple values to the response for a given header, see *append_header()*.

> **Warning**
>
> This method cannot be used to set cookies; instead, use *append_header()* or *set_cookie()*.

Parameters

headers (*Iterable[[str, str]]*) – An iterable of `[name, value]` two-member iterables, or a dict-like object that implements an `items()` method. Both *name* and *value* must be of type `str` and contain only US-ASCII characters.

> **Note**
>
> Falcon can process an iterable of tuples slightly faster than a dict.

Raises

ValueError – *headers* was not a `dict` or `list` of `tuple` or `Iterable[[str, str]]`.

set_stream(*stream:* ReadableIO | *Iterable[bytes]*, *content_length: int*) → None

Set both *stream* and *content_length*.

Although the *stream* and *content_length* properties may be set directly, using this method ensures *content_length* is not accidentally neglected when the length of the stream is known in advance. Using this method is also slightly more performant as compared to setting the properties individually.

> **Note**
>
> If the stream length is unknown, you can set *stream* directly, and ignore *content_length*. In this case, the ASGI server may choose to use chunked encoding or one of the other strategies suggested by PEP-3333.

Parameters

- **stream** – A readable file-like object.
- **content_length** (*int*) – Length of the stream, used for the Content-Length header in the response.

status: str | int | http.HTTPStatus

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string (e.g., `'200 OK'`), or an `int`.

> **Note**
>
> The Falcon framework itself provides a number of constants for common status codes. They all start with the `HTTP_` prefix, as in: `falcon.HTTP_204`. (See also: *Status Codes*.)

property status_code: int

HTTP status code normalized from *status*.

When a code is assigned to this property, *status* is updated, and vice-versa. The status code can be useful when needing to check in middleware for codes that fall into a certain class, e.g.:

```
if resp.status_code >= 400:
    log.warning(f'returning error response: {resp.status_code}')
```

stream: *ReadableIO* | Iterable[bytes] | None

Either a file-like object with a *read()* method that takes an optional size argument and returns a block of bytes, or an iterable object, representing response content, and yielding blocks as byte strings. Falcon will use *wsgi.file_wrapper*, if provided by the WSGI server, in order to efficiently serve file-like objects.

> **Note**
>
> If the stream is set to an iterable object that requires resource cleanup, it can implement a close() method to do so. The close() method will be called upon completion of the request.

text: str | None

String representing response content.

> **Note**
>
> Falcon will encode the given text as UTF-8 in the response. If the content is already a byte string, use the *data* attribute instead (it's faster).

unset_cookie(*name: str*, *domain: str | None = None*, *path: str | None = None*, *same_site: str = 'Lax'*, *samesite: str | None = None*) → None

Unset a cookie in the response.

Clears the contents of the cookie, and instructs the user agent to immediately expire its own copy of the cookie.

> **Note**
>
> Modern browsers place restriction on cookies without the "same-site" cookie attribute set. To that end this attribute is set to `'Lax'` by this method.
>
> (See also: Same-Site warnings)

> **Warning**
>
> In order to successfully remove a cookie, both the path and the domain must match the values that were used when the cookie was created.

Parameters

name (*str*) – Cookie name

Keyword Arguments

- **domain** (*str*) – Restricts the cookie to a specific domain and any subdomains of that domain. By default, the user agent will return the cookie only to the origin server. When overriding this default behavior, the specified domain must include the origin server. Otherwise, the user agent will reject the cookie.

  > **Note**
  >
  > Cookies do not provide isolation by port, so the domain should not provide one. (See also: RFC 6265, Section 8.5)

  (See also: RFC 6265, Section 4.1.2.3)

- **path** (*str*) – Scopes the cookie to the given path plus any subdirectories under that path (the "/" character is interpreted as a directory separator). If the cookie does not specify a path, the user agent defaults to the path component of the requested URI.

  > **Warning**
  >
  > User agent interfaces do not always isolate cookies by path, and so this should not be considered an effective security measure.

  (See also: RFC 6265, Section 4.1.2.4)

- **same_site** (*str*) – Allows to override the default 'Lax' same_site setting for the unset cookie.

  Added in version 4.1.

- **samesite** (*str*) – Deprecated. Use *same_site* instead.

  Deprecated since version 4.1: Please use the `same_site` parameter instead.

property vary: str | None

Value to use for the Vary header.

Set this property to an iterable of header names. For a single asterisk or field value, simply pass a single-element `list` or `tuple`.

The "Vary" header field in a response describes what parts of a request message, aside from the method, Host header field, and request target, might influence the origin server's process for selecting and representing this response. The value consists of either a single asterisk ("*") or a list of header field names (case-insensitive).

(See also: RFC 7231, Section 7.1.4)

property viewable_as: str | None

Set an inline Content-Disposition header using the given filename.

The value will be used for the `filename` directive. For example, given `'report.pdf'`, the Content-Disposition header would be set to: `'inline; filename="report.pdf"'`.

As per RFC 6266 recommendations, non-ASCII filenames will be encoded using the `filename*` directive, whereas `filename` will contain the US ASCII fallback.

Added in version 3.1.

ASGI Request & Response

Instances of the *falcon.asgi.Request* and *falcon.asgi.Response* classes are passed into responders as the second and third arguments, respectively:

```
import falcon.asgi


class Resource:

    async def on_get(self, req, resp):
        resp.media = {'message': 'Hello world!'}
        resp.status = 200


# -- snip --


app = falcon.asgi.App()
app.add_route('/', Resource())
```

Request

class falcon.asgi.**Request**(*scope: dict[str, Any]*, *receive: Callable[[], Awaitable[Mapping[str, Any]]]*, *first_event: Mapping[str, Any] | None = None*, *options:* RequestOptions *| None = None*)

Represents a client's HTTP request.

> **Note**
>
> *Request* is not meant to be instantiated directly by responders.

Parameters

- **scope** (*dict*) – ASGI HTTP connection scope passed in from the server (see also: Connection Scope).
- **receive** (*awaitable*) – ASGI awaitable callable that will yield a new event dictionary when one is available.

Keyword Arguments

- **first_event** (*dict*) – First ASGI event received from the client, if one was preloaded (default `None`).
- **options** (`falcon.request.RequestOptions`) – Set of global request options passed from the App handler.

property accept: str

Value of the Accept header, or `'*/*'` if the header is missing.

property access_route: list[str]

IP address of the original client (if known), as well as any known addresses of proxies fronting the ASGI server.

The following request headers are checked, in order of preference, to determine the addresses:

- `Forwarded`
- `X-Forwarded-For`
- `X-Real-IP`

In addition, the value of the "client" field from the ASGI connection scope will be appended to the end of the list if not already included in one of the above headers. If the "client" field is not available, it will default to `'127.0.0.1'`.

Note

Per RFC 7239, the access route may contain "unknown" and obfuscated identifiers, in addition to IPv4 and IPv6 addresses

Warning

Headers can be forged by any client or proxy. Use this property with caution and validate all values before using them. Do not rely on the access route to authorize requests!

property app: str

Deprecated alias for *root_path*.

property auth: str | None

property bounded_stream: *BoundedStream*

Alias to *stream*.

client_accepts(*media_type: str*) → bool

Determine whether or not the client accepts a given media type.

Parameters

media_type (*str*) – An Internet media type to check.

Returns

`True` if the client has indicated in the Accept header that it accepts the specified media type. Otherwise, returns `False`.

Return type

bool

property client_accepts_json: bool

`True` if the Accept header indicates that the client is willing to receive JSON, otherwise `False`.

property client_accepts_msgpack: bool

`True` if the Accept header indicates that the client is willing to receive MessagePack, otherwise `False`.

property client_accepts_xml: bool

`True` if the Accept header indicates that the client is willing to receive XML, otherwise `False`.

client_prefers(*media_types: Iterable[str]*) → str | None

Return the client's preferred media type, given several choices.

Parameters

media_types (*iterable of str*) – One or more Internet media types from which to choose the client's preferred type. This value **must** be an iterable collection of strings.

Returns

The client's preferred media type, based on the Accept header. Returns `None` if the client does not accept any of the given types.

Return type

str

property content_length: int | None

Value of the Content-Length header converted to an `int`.

Returns `None` if the header is missing.

content_type: str | None

Value of the Content-Type header, or `None` if the header is missing.

context: *structures.Context*

Empty object to hold any data (in its attributes) about the request which is specific to your app (e.g. session object). Falcon itself will not interact with this attribute after it has been initialized.

Note

The preferred way to pass request-specific data, when using the default context type, is to set attributes directly on the *context* object. For example:

```
req.context.role = 'trial'
req.context.user = 'guest'
```

context_type

alias of *Context*

property cookies: Mapping[str, str]

A dict of name/value cookie pairs.

The returned object should be treated as read-only to avoid unintended side-effects. If a cookie appears more than once in the request, only the first value encountered will be made available here.

See also: *get_cookie_values()* or *get_cookie_values()*.

property date: datetime | None

Value of the Date header, converted to a `datetime` instance.

The header value is assumed to conform to RFC 1123.

Changed in version 4.0: This property now returns timezone-aware `datetime` objects (or `None`).

property env: NoReturn

The env property is not available in ASGI. Use `store` instead.

property expect: str | None

property forwarded: list[*Forwarded*] | None

Value of the Forwarded header, as a parsed list of *falcon.Forwarded* objects, or None if the header is missing. If the header value is malformed, Falcon will make a best effort to parse what it can.

(See also: RFC 7239, Section 4)

property forwarded_host: str

Original host request header as received by the first proxy in front of the application server.

The following request headers are checked, in order of preference, to determine the forwarded scheme:

- Forwarded
- X-Forwarded-Host

If none of the above headers are available, or if the Forwarded header is available but the "host" parameter is not included in the first hop, the value of *host* is returned instead.

> **Note**
>
> Reverse proxies are often configured to set the Host header directly to the one that was originally requested by the user agent; in that case, using *host* is sufficient.

(See also: RFC 7239, Section 4)

property forwarded_prefix: str

The prefix of the original URI for proxied requests.

Uses *forwarded_scheme* and *forwarded_host* in order to reconstruct the original URI.

property forwarded_scheme: str

Original URL scheme requested by the user agent, if the request was proxied.

Typical values are 'http' or 'https'.

The following request headers are checked, in order of preference, to determine the forwarded scheme:

- Forwarded
- X-Forwarded-For

If none of these headers are available, or if the Forwarded header is available but does not contain a "proto" parameter in the first hop, the value of *scheme* is returned instead.

(See also: RFC 7239, Section 1)

property forwarded_uri: str

Original URI for proxied requests.

Uses *forwarded_scheme* and *forwarded_host* in order to reconstruct the original URI requested by the user agent.

get_cookie_values(*name: str*) → list[str] | None

Return all values provided in the Cookie header for the named cookie.

(See also: *Getting Cookies*)

Parameters

name (*str*) – Cookie name, case-sensitive.

Returns

Ordered list of all values specified in the Cookie header for the named cookie, or None if the cookie was not included in the request. If the cookie is specified more than once in the header, the returned list of values will preserve the ordering of the individual cookie-pair's in the header.

Return type
list

get_header(*name: str*, *required: bool = False*, *default: str | None = None*, *bytes] = {}*) → str | None

Retrieve the raw string value for the given header.

Parameters
name (*str*) – Header name, case-insensitive (e.g., 'Content-Type')

Keyword Arguments

- **required** (*bool*) – Set to `True` to raise `HTTPBadRequest` instead of returning gracefully when the header is not found (default `False`).
- **default** (*any*) – Value to return if the header is not found (default `None`).

Returns
The value of the specified header if it exists, or the default value if the header is not found and is not required.

Return type
str

Raises
HTTPBadRequest – The header was not found in the request, but it was required.

get_header_as_datetime(*header: str*, *required: bool = False*, *obs_date: bool = False*) → datetime | None

Return an HTTP header with HTTP-Date values as a datetime.

Parameters
name (*str*) – Header name, case-insensitive (e.g., 'Date')

Keyword Arguments

- **required** (*bool*) – Set to `True` to raise `HTTPBadRequest` instead of returning gracefully when the header is not found (default `False`).
- **obs_date** (*bool*) – Support obs-date formats according to RFC 7231, e.g.: "Sunday, 06-Nov-94 08:49:37 GMT" (default `False`).

Returns
The value of the specified header if it exists, or `None` if the header is not found and is not required.

Return type
datetime

Raises

- ***HTTPBadRequest*** – The header was not found in the request, but it was required.
- **HttpInvalidHeader** – The header contained a malformed/invalid value.

Changed in version 4.0: This method now returns timezone-aware `datetime` objects.

get_header_as_int(*header: str*, *required: bool = False*) → int | None

Retrieve the int value for the given header.

Parameters
name (*str*) – Header name, case-insensitive (e.g., 'Content-Length')

Keyword Arguments
required (*bool*) – Set to `True` to raise `HTTPBadRequest` instead of returning gracefully when the header is not found (default `False`).

Returns
The value of the specified header if it exists, or `None` if the header is not found and is not required.

Return type
int

Raises

- **HTTPBadRequest** – The header was not found in the request, but it was required.
- **HttpInvalidHeader** – The header contained a malformed/invalid value.

Added in version 4.0.

async get_media(*default_when_empty: Literal[_Unset.UNSET] | Any = _Unset.UNSET*) → Any

Return a deserialized form of the request stream.

The first time this method is called, the request stream will be deserialized using the Content-Type header as well as the media-type handlers configured via `falcon.RequestOptions`. The result will be cached and returned in subsequent calls:

```
deserialized_media = await req.get_media()
```

If the matched media handler raises an error while attempting to deserialize the request body, the exception will propagate up to the caller.

See also *Media* for more information regarding media handling.

Note

When `get_media` is called on a request with an empty body, Falcon will let the media handler try to deserialize the body and will return the value returned by the handler or propagate the exception raised by it. To instead return a different value in case of an exception by the handler, specify the argument `default_when_empty`.

Warning

This operation will consume the request stream the first time it's called and cache the results. Follow-up calls will just retrieve a cached version of the object.

Parameters
default_when_empty – Fallback value to return when there is no body in the request and the media handler raises an error (like in the case of the default JSON media handler). By default, Falcon uses the value returned by the media handler or propagates the raised exception, if any. This value is not cached, and will be used only for the current call.

Returns
The deserialized media representation.

Return type
media (object)

get_param(*name: str*, *required: bool = False*, *store: dict[str, Any] | None = None*, *default: str | None = None*) → str | None

Return the raw value of a query string parameter as a string.

Note

If an HTML form is POSTed to the API using the *application/x-www-form-urlencoded* media type, Falcon can automatically parse the parameters from the request body via `get_media()`.

See also: *How can I access POSTed form params?*

> **Note**
>
> Similar to the way multiple keys in form data are handled, if a query parameter is included in the query string multiple times, only one of those values will be returned, and it is undefined which one. This caveat also applies when *auto_parse_qs_csv* is enabled and the given parameter is assigned to a comma-separated list of values (e.g., `foo=a,b,c`).
>
> When multiple values are expected for a parameter, *get_param_as_list()* can be used to retrieve all of them at once.

Parameters
name (*str*) – Parameter name, case-sensitive (e.g., 'sort').

Keyword Arguments

- **required** (*bool*) – Set to `True` to raise `HTTPBadRequest` instead of returning `None` when the parameter is not found (default `False`).
- **store** (*dict*) – A `dict`-like object in which to place the value of the param, but only if the param is present.
- **default** (*any*) – If the param is not found returns the given value instead of `None`

Returns
The value of the param as a string, or `None` if param is not found and is not required.

Return type
str

Raises
HTTPBadRequest – A required param is missing from the request.

get_param_as_bool(*name: str*, *required: bool = False*, *store: dict[str, Any] | None = None*, *blank_as_true: bool = True*, *default: bool | None = None*) → bool | None

Return the value of a query string parameter as a boolean.

This method treats valueless parameters as flags. By default, if no value is provided for the parameter in the query string, `True` is assumed and returned. If the parameter is missing altogether, `None` is returned as with other `get_param_*()` methods, which can be easily treated as falsy by the caller as needed.

The following boolean strings are supported:

```
TRUE_STRINGS = ('true', 'True', 't', 'yes', 'y', '1', 'on')
FALSE_STRINGS = ('false', 'False', 'f', 'no', 'n', '0', 'off')
```

Parameters
name (*str*) – Parameter name, case-sensitive (e.g., 'detailed').

Keyword Arguments

- **required** (*bool*) – Set to `True` to raise `HTTPBadRequest` instead of returning `None` when the parameter is not found or is not a recognized boolean string (default `False`).
- **store** (*dict*) – A `dict`-like object in which to place the value of the param, but only if the param is found (default `None`).
- **blank_as_true** (*bool*) – Valueless query string parameters are treated as flags, resulting in `True` being returned when such a parameter is present, and `False` otherwise. To require the client to explicitly opt-in to a truthy value, pass `blank_as_true=False` to return `False` when a value is not specified in the query string.
- **default** (*any*) – If the param is not found, return this value instead of `None`.

Returns
The value of the param if it is found and can be converted to a `bool`. If the param is not found, returns `None` unless *required* is `True`.

Return type
bool

Raises
HTTPBadRequest – A required param is missing from the request, or can not be converted to a `bool`.

get_param_as_date(*name: str*, *format_string: str = '%Y-%m-%d'*, *required: bool = False*, *store: dict[str, Any] | None = None*, *default: date | None = None*) → date | None

Return the value of a query string parameter as a date.

Parameters
name (*str*) – Parameter name, case-sensitive (e.g., 'ids').

Keyword Arguments

- **format_string** (*str*) – String used to parse the param value into a date. Any format recognized by strptime() is supported (default `"%Y-%m-%d"`).
- **required** (*bool*) – Set to `True` to raise `HTTPBadRequest` instead of returning `None` when the parameter is not found (default `False`).
- **store** (*dict*) – A `dict`-like object in which to place the value of the param, but only if the param is found (default `None`).
- **default** (*any*) – If the param is not found returns the given value instead of `None`

Returns
The value of the param if it is found and can be converted to a `date` according to the supplied format string. If the param is not found, returns `None` unless required is `True`.

Return type
datetime.date

Raises
HTTPBadRequest – A required param is missing from the request, or the value could not be converted to a `date`.

get_param_as_datetime(*name: str*, *format_string: str = '%Y-%m-%dT%H:%M:%S%z'*, *required: bool = False*, *store: dict[str, Any] | None = None*, *default: datetime | None = None*) → datetime | None

Return the value of a query string parameter as a datetime.

Parameters
name (*str*) – Parameter name, case-sensitive (e.g., 'ids').

Keyword Arguments

- **format_string** (*str*) – String used to parse the param value into a `datetime`. Any format recognized by strptime() is supported (default `'%Y-%m-%dT%H:%M:%S%z'`).
- **required** (*bool*) – Set to `True` to raise `HTTPBadRequest` instead of returning `None` when the parameter is not found (default `False`).
- **store** (*dict*) – A `dict`-like object in which to place the value of the param, but only if the param is found (default `None`).
- **default** (*any*) – If the param is not found returns the given value instead of `None`

Returns
The value of the param if it is found and can be converted to a `datetime` according to the supplied format string. If the param is not found, returns `None` unless required is `True`.

Return type
datetime.datetime

Raises
HTTPBadRequest – A required param is missing from the request, or the value could not be converted to a `datetime`.

Changed in version 4.0: The default value of *format_string* was changed from `'%Y-%m-%dT%H:%M:%SZ'` to `'%Y-%m-%dT%H:%M:%S%z'`.

The new format is a superset of the old one parsing-wise, however, the converted `datetime` object is now timezone-aware.

get_param_as_float(*name: str*, *required: bool = False*, *min_value: float | None = None*, *max_value: float | None = None*, *store: dict[str, Any] | None = None*, *default: float | None = None*) → float | None

Return the value of a query string parameter as an float.

Parameters
name (*str*) – Parameter name, case-sensitive (e.g., 'limit').

Keyword Arguments

- **required** (*bool*) – Set to `True` to raise `HTTPBadRequest` instead of returning `None` when the parameter is not found or is not an float (default `False`).
- **min_value** (*float*) – Set to the minimum value allowed for this param. If the param is found and it is less than min_value, an `HTTPError` is raised.
- **max_value** (*float*) – Set to the maximum value allowed for this param. If the param is found and its value is greater than max_value, an `HTTPError` is raised.
- **store** (*dict*) – A `dict`-like object in which to place the value of the param, but only if the param is found (default `None`).
- **default** (*any*) – If the param is not found returns the given value instead of `None`

Returns
The value of the param if it is found and can be converted to an `float`. If the param is not found, returns `None`, unless *required* is `True`.

Return type
float

Raises
HTTPBadRequest – The param was not found in the request, even though it was required to be there, or it was found but could not be converted to an `float`. Also raised if the param's value falls outside the given interval, i.e., the value must be in the interval: min_value <= value <= max_value to avoid triggering an error.

get_param_as_int(*name: str*, *required: bool = False*, *min_value: int | None = None*, *max_value: int | None = None*, *store: dict[str, Any] | None = None*, *default: int | None = None*) → int | None

Return the value of a query string parameter as an int.

Parameters
name (*str*) – Parameter name, case-sensitive (e.g., 'limit').

Keyword Arguments

- **required** (*bool*) – Set to `True` to raise `HTTPBadRequest` instead of returning `None` when the parameter is not found or is not an integer (default `False`).
- **min_value** (*int*) – Set to the minimum value allowed for this param. If the param is found and it is less than min_value, an `HTTPError` is raised.
- **max_value** (*int*) – Set to the maximum value allowed for this param. If the param is found and its value is greater than max_value, an `HTTPError` is raised.

- **store** (*dict*) – A `dict`-like object in which to place the value of the param, but only if the param is found (default `None`).
- **default** (*any*) – If the param is not found returns the given value instead of `None`

Returns
The value of the param if it is found and can be converted to an `int`. If the param is not found, returns `None`, unless *required* is `True`.

Return type
int

Raises
HTTPBadRequest – The param was not found in the request, even though it was required to be there, or it was found but could not be converted to an `int`. Also raised if the param's value falls outside the given interval, i.e., the value must be in the interval: min_value <= value <= max_value to avoid triggering an error.

get_param_as_json(*name: str*, *required: bool = False*, *store: dict[str, Any] | None = None*, *default: Any | None = None*) → Any

Return the decoded JSON value of a query string parameter.

Given a JSON value, decode it to an appropriate Python type, (e.g., `dict`, `list`, `str`, `int`, `bool`, etc.)

Warning

If the *auto_parse_qs_csv* option is set to `True` (default `False`), the framework will misinterpret any JSON values that include literal (non-percent-encoded) commas. If the query string may include JSON, you can use JSON array syntax in lieu of CSV as a workaround.

Parameters
name (*str*) – Parameter name, case-sensitive (e.g., 'payload').

Keyword Arguments

- **required** (*bool*) – Set to `True` to raise `HTTPBadRequest` instead of returning `None` when the parameter is not found (default `False`).
- **store** (*dict*) – A `dict`-like object in which to place the value of the param, but only if the param is found (default `None`).
- **default** (*any*) – If the param is not found returns the given value instead of `None`

Returns
The value of the param if it is found. Otherwise, returns `None` unless required is `True`.

Return type
dict

Raises
HTTPBadRequest – A required param is missing from the request, or the value could not be parsed as JSON.

get_param_as_list(*name: str*, *transform: Callable[[str], _T] | None = None*, *required: bool = False*, *store: dict[str, Any] | None = None*, *default: list[_T] | None = None*) → list[_T] | list[str] | None

Return the value of a query string parameter as a list.

List items must be comma-separated or must be provided as multiple instances of the same param in the query string ala *application/x-www-form-urlencoded*.

> **Note**
>
> To enable the interpretation of comma-separated parameter values, the *auto_parse_qs_csv* option must be set to `True` (default `False`).

Parameters

name (*str*) – Parameter name, case-sensitive (e.g., 'ids').

Keyword Arguments

- **transform** (*callable*) – An optional transform function that takes as input each element in the list as a `str` and outputs a transformed element for inclusion in the list that will be returned. For example, passing `int` will transform list items into numbers.
- **required** (*bool*) – Set to `True` to raise `HTTPBadRequest` instead of returning `None` when the parameter is not found (default `False`).
- **store** (*dict*) – A `dict`-like object in which to place the value of the param, but only if the param is found (default `None`).
- **default** (*any*) – If the param is not found returns the given value instead of `None`

Returns

The value of the param if it is found. Otherwise, returns `None` unless *required* is `True`.

Empty list elements will be included by default, but this behavior can be configured by setting the *keep_blank_qs_values* option. For example, by default the following query strings would both result in `['1', '', '3']`:

```
things=1&things=&things=3
things=1,,3
```

Note, however, that for the second example string above to be interpreted as a list, the *auto_parse_qs_csv* option must be set to `True`.

Return type

list

Raises

HTTPBadRequest – A required param is missing from the request, or a transform function raised an instance of `ValueError`.

get_param_as_uuid(*name: str*, *required: bool = False*, *store: dict[str, Any] | None = None*, *default: UUID | None = None*) → UUID | None

Return the value of a query string parameter as an UUID.

The value to convert must conform to the standard UUID string representation per RFC 4122. For example, the following strings are all valid:

```
# Lowercase
'64be949b-3433-4d36-a4a8-9f19d352fee8'

# Uppercase
'BE71ECAA-F719-4D42-87FD-32613C2EEB60'

# Mixed
'81c8155C-D6de-443B-9495-39Fa8FB239b5'
```

Parameters

name (*str*) – Parameter name, case-sensitive (e.g., 'id').

Keyword Arguments

- **required** (*bool*) – Set to `True` to raise `HTTPBadRequest` instead of returning `None` when the parameter is not found or is not a UUID (default `False`).
- **store** (*dict*) – A `dict`-like object in which to place the value of the param, but only if the param is found (default `None`).
- **default** (*any*) – If the param is not found returns the given value instead of `None`

Returns
The value of the param if it is found and can be converted to a `UUID`. If the param is not found, returns `default` (default `None`), unless *required* is `True`.

Return type
UUID

Raises
HTTPBadRequest – The param was not found in the request, even though it was required to be there, or it was found but could not be converted to a `UUID`.

has_param(*name: str*) → bool

Determine whether or not the query string parameter already exists.

Parameters
name (*str*) – Parameter name, case-sensitive (e.g., 'sort').

Returns
`True` if param is found, or `False` if param is not found.

Return type
bool

property headers: Mapping[str, str]

Raw HTTP headers from the request with dash-separated names normalized to lowercase.

> **Note**
>
> This property differs from the WSGI version of `Request.headers` in that the latter returns *upper-case* names for historical reasons. Middleware, such as tracing and logging components, that need to be compatible with both WSGI and ASGI apps should use *headers_lower* instead.

> **Warning**
>
> Parsing all the headers to create this dict is done the first time this attribute is accessed, and the returned object should be treated as read-only. Note that this parsing can be costly, so unless you need all the headers in this format, you should instead use the `get_header()` method or one of the convenience attributes to get a value for a specific header.

property headers_lower: Mapping[str, str]

Alias for *headers* provided to expose a uniform way to get lowercased headers for both WSGI and ASGI apps.

property host: str

Host request header field, if present.

If the Host header is missing, this attribute resolves to the ASGI server's listening host name or IP address.

property if_match: list[*ETag* | Literal['*']] | None

Value of the If-Match header, as a parsed list of *falcon.ETag* objects or `None` if the header is missing or its value is blank.

This property provides a list of all `entity-tags` in the header, both strong and weak, in the same order as listed in the header.

(See also: RFC 7232, Section 3.1)

property if_modified_since: datetime | None

Value of the If-Modified-Since header.

Returns `None` if the header is missing.

Changed in version 4.0: This property now returns timezone-aware `datetime` objects (or `None`).

property if_none_match: list[*ETag* | Literal['*']] | None

Value of the If-None-Match header, as a parsed list of *falcon.ETag* objects or `None` if the header is missing or its value is blank.

This property provides a list of all `entity-tags` in the header, both strong and weak, in the same order as listed in the header.

(See also: RFC 7232, Section 3.2)

property if_range: str | None

property if_unmodified_since: datetime | None

Value of the If-Unmodified-Since header.

Returns `None` if the header is missing.

Changed in version 4.0: This property now returns timezone-aware `datetime` objects (or `None`).

is_websocket: bool

Set to `True` IFF this request was made as part of a WebSocket handshake.

log_error(*message: str*) → NoReturn

Write a message to the server's log.

> **Warning**
>
> Although this method is inherited from the WSGI Request class, it is not supported for ASGI apps. Please use the standard library logging framework instead.

property media: Any

An awaitable property that acts as an alias for *get_media()*. This can be used to ease the porting of a WSGI app to ASGI, although the `await` keyword must still be added when referencing the property:

```
deserialized_media = await req.media
```

method: str

HTTP method requested, uppercase (e.g., `'GET'`, `'POST'`, etc.)

property netloc: str

port" portion of the request URL.

The port may be omitted if it is the default one for the URL's schema (80 for HTTP and 443 for HTTPS).

Type
Returns the "host

options: *RequestOptions*

Set of global options passed from the App handler.

property params: Mapping[str, str | list[str]]

The mapping of request query parameter names to their values.

Where the parameter appears multiple times in the query string, the value mapped to that parameter key will be a list of all the values in the order seen.

path: str

Path portion of the request URI (not including query string).

> **Warning**
>
> If this attribute is to be used by the app for any upstream requests, any non URL-safe characters in the path must be URL encoded back before making the request.

> **Note**
>
> `req.path` may be set to a new value by a `process_request()` middleware method in order to influence routing. If the original request path was URL encoded, it will be decoded before being returned by this attribute.

property port: int

Port used for the request.

If the Host header is present in the request, but does not specify a port, the default one for the given schema is returned (80 for HTTP and 443 for HTTPS). If the request does not include a Host header, the listening port for the server is returned instead.

property prefix: str

The prefix of the request URI, including scheme, host, and app *root_path* (if any).

query_string: str

Query string portion of the request URI, without the preceding '?' character.

property range: tuple[int, int] | None

A 2-member `tuple` parsed from the value of the Range header, or `None` if the header is missing.

The two members correspond to the first and last byte positions of the requested resource, inclusive. Negative indices indicate offset from the end of the resource, where -1 is the last byte, -2 is the second-to-last byte, and so forth.

Only continuous ranges are supported (e.g., "bytes=0-0,-1" would result in an HTTPBadRequest exception when the attribute is accessed).

property range_unit: str | None

Unit of the range parsed from the value of the Range header.

Returns `None` if the header is missing.

property referer: str | None

property relative_uri: str

The path and query string portion of the request URI, omitting the scheme and host.

property remote_addr: str

IP address of the closest known client or proxy to the ASGI server, or `'127.0.0.1'` if unknown.

This property's value is equivalent to the last element of the *access_route* property.

property root_path: str

The initial portion of the request URI's path that corresponds to the application object, so that the application knows its virtual "location". This may be an empty string, if the application corresponds to the "root" of the server.

(In WSGI it corresponds to the "SCRIPT_NAME" environ variable defined by PEP-3333; in ASGI it Corresponds to the "root_path"ASGI HTTP scope field.)

property scheme: str

URL scheme used for the request.

One of `'http'`, `'https'`, `'ws'`, or `'wss'`. Defaults to `'http'` for the `http` scope, or `'ws'` for the `websocket` scope, when the ASGI server does not include the scheme in the connection scope.

> **Note**
>
> If the request was proxied, the scheme may not match what was originally requested by the client. *`forwarded_scheme`* can be used, instead, to handle such cases.

scope: dict[str, Any]

Reference to the ASGI HTTP connection scope passed in from the server (see also: Connection Scope).

property stream: *BoundedStream*

File-like input object for reading the body of the request, if any.

property subdomain: str | None

Leftmost (i.e., most specific) subdomain from the hostname.

If only a single domain name is given, *subdomain* will be `None`.

> **Note**
>
> If the hostname in the request is an IP address, the value for *subdomain* is undefined.

property uri: str

The fully-qualified URI for the request.

uri_template: str | None

The template for the route that was matched for this request. May be `None` if the request has not yet been routed, as would be the case for `process_request()` middleware methods. May also be `None` if your app uses a custom routing engine and the engine does not provide the URI template when resolving a route.

property url: str

The fully-qualified URI for the request.

property user_agent: str | None

class falcon.asgi.**BoundedStream**(*receive: Callable[[], Awaitable[Mapping[str, Any]]]*, *first_event: Mapping[str, Any] | None = None*, *content_length: int | None = None*)

File-like input object for reading the body of the request, if any.

This class implements coroutine functions for asynchronous reading or iteration, but otherwise provides an interface similar to that defined by `io.IOBase`.

If the request includes a Content-Length header, the number of bytes in the stream will be truncated to the length specified by the header. Otherwise, the stream will yield data until the ASGI server indicates that no more bytes are available.

For large request bodies, the preferred method of using the stream object is as an asynchronous iterator. In this mode, each body chunk is simply yielded in its entirety, as it is received from the ASGI server. Because no data is buffered by the framework, this is the most memory-efficient way of reading the request body:

```
# If the request body is empty or has already be consumed, the iteration
#   will immediately stop without yielding any data chunks. Otherwise, a
#   series of byte #   strings will be yielded until the entire request
#   body has been yielded or the client disconnects.
async for data_chunk in req.stream
    pass
```

The stream object also supports asynchronous `read()` and `readall()` methods:

```
# Read all of the data at once; use only when you are confident
#   that the request body is small enough to not eat up all of
#   your memory. For small bodies, this is the most performant
#   option.
data = await req.stream.readall()

# ...or call read() without arguments
data = await req.stream.read()

# ...or read the data in chunks. You may choose to read more
#   or less than 32 KiB as shown in this example. But note that
#   this approach will generally be less efficient as compared
#   to async iteration, resulting in more usage and
#   copying of memory.
while True:
    data_chunk = await req.stream.read(32 * 1024)
    if not data_chunk:
        break
```

Warning

Apps may not use both `read()` and the asynchronous iterator interface to consume the same request body; the only time that it is safe to do so is when one or the other method is used to completely read the entire body *before* the other method is even attempted. Therefore, it is important to always call *exhaust()* or *close()* if a body has only been partially read and the remaining data is to be ignored.

Note

The stream object provides a convenient abstraction over the series of body chunks contained in any ASGI "http.request" events received by the app. As such, some request body data may be temporarily buffered in memory during and between calls to read from the stream. The framework has been designed to minimize the amount of data that must be buffered in this manner.

Parameters

receive (*awaitable*) – ASGI awaitable callable that will yield a new request event dictionary when one is available.

Keyword Arguments

- **first_event** (*dict*) – First ASGI event received from the client, if one was preloaded (default `None`).
- **content_length** (*int*) – Expected content length of the stream, derived from the Content-Length header in the request (if available).

close() → None

Clear any buffered data and close this stream.

Once the stream is closed, any operation on it will raise an instance of `ValueError`.

As a convenience, it is allowed to call this method more than once; only the first call, however, will have an effect.

async exhaust() → None

Consume and immediately discard any remaining data in the stream.

fileno() → NoReturn

Raise an instance of OSError since a file descriptor is not used.

isatty() → bool

Return `False` always.

async read(*size: int | None = None*) → bytes

Read some or all of the remaining bytes in the request body.

> **Warning**
>
> A size should always be specified, unless you can be certain that you have enough free memory for the entire request body, and that you have configured your web server to limit request bodies to a reasonable size (to guard against malicious requests).

> **Warning**
>
> Apps may not use both `read()` and the asynchronous iterator interface to consume the same request body; the only time that it is safe to do so is when one or the other method is used to completely read the entire body *before* the other method is even attempted. Therefore, it is important to always call *exhaust()* or *close()* if a body has only been partially read and the remaining data is to be ignored.

Keyword Arguments

size (*int*) – The maximum number of bytes to read. The actual amount of data that can be read will depend on how much is available, and may be smaller than the amount requested. If the size is -1 or not specified, all remaining data is read and returned.

Returns

The request body data, or `b''` if the body is empty or has already been consumed.

Return type

bytes

readable() → bool

Return `True` always.

async readall() → bytes

Read and return all remaining data in the request body.

> **Warning**
>
> Only use this method when you can be certain that you have enough free memory for the entire request body, and that you have configured your web server to limit request bodies to a reasonable size (to guard against malicious requests).

Returns
The request body data, or `b''` if the body is empty or has already been consumed.

Return type
bytes

seekable() → bool
Return `False` always.

tell() → int
Return the number of bytes read from the stream so far.

writable() → bool
Return `False` always.

Response

class falcon.asgi.**Response**(*options:* ResponseOptions | *None* = *None*)
Represents an HTTP response to a client request.

> **Note**
> `Response` is not meant to be instantiated directly by responders.

Keyword Arguments
options (*dict*) – Set of global options passed from the App handler.

property accept_ranges: str | None
Set the Accept-Ranges header.

The Accept-Ranges header field indicates to the client which range units are supported (e.g. "bytes") for the target resource.

If range requests are not supported for the target resource, the header may be set to "none" to advise the client not to attempt any such requests.

> **Note**
> "none" is the literal string, not Python's built-in `None` type.

append_header(*name: str*, *value: str*) → None
Set or append a header for this response.

If the header already exists, the new value will normally be appended to it, delimited by a comma. The notable exception to this rule is Set-Cookie, in which case a separate header line for each value will be included in the response.

> **Note**
> While this method can be used to efficiently append raw Set-Cookie headers to the response, you may find *set_cookie()* to be more convenient.

Parameters

- **name** (*str*) – Header name (case-insensitive). The name may contain only US-ASCII characters.

- **value** (*str*) – Value for the header. As with the header's name, the value may contain only US-ASCII characters.

append_link(*target: str*, *rel: str*, *title: str | None = None*, *title_star: tuple[str, str] | None = None*, *anchor: str | None = None*, *hreflang: str | Iterable[str] | None = None*, *type_hint: str | None = None*, *crossorigin: str | None = None*, *link_extension: Iterable[tuple[str, str]] | None = None*) → None

Append a link header to the response.

(See also: RFC 5988, Section 1)

> **Note**
>
> Calling this method repeatedly will cause each link to be appended to the Link header value, separated by commas.

Parameters

- **target** (*str*) – Target IRI for the resource identified by the link. Will be converted to a URI, if necessary, per RFC 3987, Section 3.1.
- **rel** (*str*) – Relation type of the link, such as "next" or "bookmark".

  (See also: http://www.iana.org/assignments/link-relations/link-relations.xhtml)

Keyword Arguments

- **title** (*str*) – Human-readable label for the destination of the link (default `None`). If the title includes non-ASCII characters, you will need to use *title_star* instead, or provide both a US-ASCII version using *title* and a Unicode version using *title_star*.
- **title_star** (*tuple[str, str]*) – Localized title describing the destination of the link (default `None`). The value must be a two-member tuple in the form of (*language-tag*, *text*), where *language-tag* is a standard language identifier as defined in RFC 5646, Section 2.1, and *text* is a Unicode string.

  > **Note**
  >
  > *language-tag* may be an empty string, in which case the client will assume the language from the general context of the current request.

  > **Note**
  >
  > *text* will always be encoded as UTF-8.

- **anchor** (*str*) – Override the context IRI with a different URI (default None). By default, the context IRI for the link is simply the IRI of the requested resource. The value provided may be a relative URI.
- **hreflang** (*str or iterable*) – Either a single *language-tag*, or a `list` or `tuple` of such tags to provide a hint to the client as to the language of the result of following the link. A list of tags may be given in order to indicate to the client that the target resource is available in multiple languages.
- **type_hint** (*str*) – Provides a hint as to the media type of the result of dereferencing the link (default `None`). As noted in RFC 5988, this is only a hint and does not override the Content-Type header returned when the link is followed.

- **crossorigin** (*str*) – Determines how cross origin requests are handled. Can take values 'anonymous' or 'use-credentials' or None. (See: https://www.w3.org/TR/html50/infrastructure.html#cors-settings-attribute)
- **link_extension** – Provides additional custom attributes, as described in RFC 8288, Section 3.4.2; each member of the iterable must be a two-tuple in the form of (*param*, *value*).

property cache_control: str | None

Set the Cache-Control header.

Used to set a list of cache directives to use as the value of the Cache-Control header. The list will be joined with ", " to produce the value for the header.

complete: bool = False

Set to `True` from within a middleware method to signal to the framework that request processing should be short-circuited (see also *Middleware*).

property content_length: str | None

Set the Content-Length header.

This property can be used for responding to HEAD requests when you aren't actually providing the response body, or when streaming the response. If either the *text* property or the *data* property is set on the response, the framework will force Content-Length to be the length of the given text bytes. Therefore, it is only necessary to manually set the content length when those properties are not used.

> **Note**
>
> In cases where the response content is a stream (readable file-like object), Falcon will not supply a Content-Length header to the server unless *content_length* is explicitly set. Consequently, the server may choose to use chunked encoding in this case.

property content_location: str | None

Set the Content-Location header.

This value will be URI encoded per RFC 3986. If the value that is being set is already URI encoded it should be decoded first or the header should be set manually using the set_header method.

property content_range: str | None

A tuple to use in constructing a value for the Content-Range header.

The tuple has the form (*start*, *end*, *length*, [*unit*]), where *start* and *end* designate the range (inclusive), and *length* is the total length, or '*' if unknown. You may pass `int`'s for these numbers (no need to convert to `str` beforehand). The optional value *unit* describes the range unit and defaults to 'bytes'

> **Note**
>
> You only need to use the alternate form, 'bytes */1234', for responses that use the status '416 Range Not Satisfiable'. In this case, raising `falcon.HTTPRangeNotSatisfiable` will do the right thing.

(See also: RFC 7233, Section 4.2)

property content_type: str | None

Sets the Content-Type header.

The `falcon` module provides a number of constants for common media types, including `falcon.MEDIA_JSON`, `falcon.MEDIA_MSGPACK`, `falcon.MEDIA_YAML`, `falcon.MEDIA_XML`, `falcon.MEDIA_HTML`, `falcon.MEDIA_JS`, `falcon.MEDIA_TEXT`, `falcon.MEDIA_JPEG`, `falcon.MEDIA_PNG`, and `falcon.MEDIA_GIF`.

context: *structures.Context*

Empty object to hold any data (in its attributes) about the response which is specific to your app (e.g. session object). Falcon itself will not interact with this attribute after it has been initialized.

> **Note**
>
> The preferred way to pass response-specific data, when using the default context type, is to set attributes directly on the *context* object. For example:
>
> ```
> resp.context.cache_strategy = 'lru'
> ```

context_type

alias of *Context*

property data: bytes | None

Byte string representing response content.

Use this attribute in lieu of *text* when your content is already a byte string (of type `bytes`). See also the note below.

> **Warning**
>
> Always use the *text* attribute for text, or encode it first to `bytes` when using the *data* attribute, to ensure Unicode characters are properly encoded in the HTTP response.

delete_header(*name: str*) → None

Delete a header that was previously set for this response.

If the header was not previously set, nothing is done (no error is raised). Otherwise, all values set for the header will be removed from the response.

Note that calling this method is equivalent to setting the corresponding header property (when said property is available) to `None`. For example:

```
resp.etag = None
```

> **Warning**
>
> This method cannot be used with the Set-Cookie header. Instead, use *unset_cookie()* to remove a cookie and ensure that the user agent expires its own copy of the data as well.

Parameters

name (*str*) – Header name (case-insensitive). The name may contain only US-ASCII characters.

Raises

ValueError – *name* cannot be `'Set-Cookie'`.

property downloadable_as: str | None

Set the Content-Disposition header using the given filename.

The value will be used for the `filename` directive. For example, given `'report.pdf'`, the Content-Disposition header would be set to: `'attachment; filename="report.pdf"'`.

As per RFC 6266 recommendations, non-ASCII filenames will be encoded using the `filename*` directive, whereas `filename` will contain the US ASCII fallback.

property etag: str | None

Set the ETag header.

The ETag header will be wrapped with double quotes `"value"` in case the user didn't pass it.

property expires: str | None

Set the Expires header. Set to a `datetime` (UTC) instance.

> **Note**
>
> Falcon will format the `datetime` as an HTTP date string.

get_header(*name: str*, *default: str | None = None*) → str | None

Retrieve the raw string value for the given header.

Normally, when a header has multiple values, they will be returned as a single, comma-delimited string. However, the Set-Cookie header does not support this format, and so attempting to retrieve it will raise an error.

Parameters

name (*str*) – Header name, case-insensitive. Must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

Keyword Arguments

default – Value to return if the header is not found (default `None`).

Raises

ValueError – The value of the 'Set-Cookie' header(s) was requested.

Returns

The value of the specified header if set, or the default value if not set.

Return type

str

property headers: dict[str, str]

Copy of all headers set for the response, without cookies.

Note that a new copy is created and returned each time this property is referenced.

property last_modified: str | None

Set the Last-Modified header. Set to a `datetime` (UTC) instance.

> **Note**
>
> Falcon will format the `datetime` as an HTTP date string.

property location: str | None

Set the Location header.

This value will be URI encoded per RFC 3986. If the value that is being set is already URI encoded it should be decoded first or the header should be set manually using the set_header method.

property media: Any

A serializable object supported by the media handlers configured via *falcon.RequestOptions*.

> **Note**
>
> See also *Media* for more information regarding media handling.

options: *ResponseOptions*

Set of global options passed in from the App handler.

async render_body() → bytes | None

Get the raw bytestring content for the response body.

This coroutine can be awaited to get the raw data for the HTTP response body, taking into account the *text*, *data*, and *media* attributes.

> **Note**
>
> This method ignores *stream*; the caller must check and handle that attribute directly.

Returns

The UTF-8 encoded value of the *text* attribute, if set. Otherwise, the value of the *data* attribute if set, or finally the serialized value of the *media* attribute. If none of these attributes are set, `None` is returned.

Return type

bytes

property retry_after: str | None

Set the Retry-After header.

The expected value is an integral number of seconds to use as the value for the header. The HTTP-date syntax is not supported.

schedule(*callback: Callable[[], Awaitable[None]]*) → None

Schedule an async callback to run soon after sending the HTTP response.

This method can be used to execute a background job after the response has been returned to the client.

The callback is assumed to be an async coroutine function. It will be scheduled to run on the event loop as soon as possible.

The callback will be invoked without arguments. Use `functools.partial` to pass arguments to the callback as needed.

> **Note**
>
> If an unhandled exception is raised while processing the request, the callback will not be scheduled to run.

> **Note**
>
> When an SSE emitter has been set on the response, the callback will be scheduled before the first call to the emitter.

> **Warning**
>
> Because coroutines run on the main request thread, care should be taken to ensure they are non-blocking. Long-running operations must use async libraries or delegate to an `Executor` pool to avoid blocking the processing of subsequent requests.

Parameters

callback (*object*) – An async coroutine function. The callback will be invoked without arguments.

schedule_sync(*callback: Callable[[], None]*) → None

Schedule a synchronous callback to run soon after sending the HTTP response.

This method can be used to execute a background job after the response has been returned to the client.

The callback is assumed to be a synchronous (non-coroutine) function. It will be scheduled on the event loop's default `Executor` (which can be overridden via `asyncio.AbstractEventLoop.set_default_executor()`).

The callback will be invoked without arguments. Use `functools.partial` to pass arguments to the callback as needed.

Note

If an unhandled exception is raised while processing the request, the callback will not be scheduled to run.

Note

When an SSE emitter has been set on the response, the callback will be scheduled before the first call to the emitter.

Warning

Synchronous callables run on the event loop's default `Executor`, which uses an instance of `ThreadPoolExecutor` unless `asyncio.AbstractEventLoop.set_default_executor()` is used to change it to something else. Due to the GIL, CPU-bound jobs will block request processing for the current process unless the default `Executor` is changed to one that is process-based instead of thread-based (e.g., an instance of `concurrent.futures.ProcessPoolExecutor`).

Parameters

callback (*object*) – An async coroutine function or a synchronous callable. The callback will be called without arguments.

set_cookie(*name: str*, *value: str*, *expires: datetime | None = None*, *max_age: int | None = None*, *domain: str | None = None*, *path: str | None = None*, *secure: bool | None = None*, *http_only: bool = True*, *same_site: str | None = None*, *partitioned: bool = False*) → None

Set a response cookie.

Note

This method can be called multiple times to add one or more cookies to the response.

See also

To learn more about setting cookies, see *Setting Cookies*. The parameters listed below correspond to those defined in RFC 6265.

Parameters

- **name** (*str*) – Cookie name
- **value** (*str*) – Cookie value

Keyword Arguments

- **expires** (*datetime*) – Specifies when the cookie should expire. By default, cookies expire when the user agent exits.

  (See also: RFC 6265, Section 4.1.2.1)

- **max_age** (*int*) – Defines the lifetime of the cookie in seconds. By default, cookies expire when the user agent exits. If both *max_age* and *expires* are set, the latter is ignored by the user agent.

  Note

  Coercion to `int` is attempted if provided with `float` or `str`.

  (See also: RFC 6265, Section 4.1.2.2)

- **domain** (*str*) – Restricts the cookie to a specific domain and any subdomains of that domain. By default, the user agent will return the cookie only to the origin server. When overriding this default behavior, the specified domain must include the origin server. Otherwise, the user agent will reject the cookie.

  Note

  Cookies do not provide isolation by port, so the domain should not provide one. (See also: RFC 6265, Section 8.5)

  (See also: RFC 6265, Section 4.1.2.3)

- **path** (*str*) – Scopes the cookie to the given path plus any subdirectories under that path (the "/" character is interpreted as a directory separator). If the cookie does not specify a path, the user agent defaults to the path component of the requested URI.

  Warning

  User agent interfaces do not always isolate cookies by path, and so this should not be considered an effective security measure.

  (See also: RFC 6265, Section 4.1.2.4)

- **secure** (*bool*) – Direct the client to only return the cookie in subsequent requests if they are made over HTTPS (default: `True`). This prevents attackers from reading sensitive cookie data.

  Note

  The default value for this argument is normally `True`, but can be modified by setting *secure_cookies_by_default* via *App.resp_options*.

> **Warning**
>
> For the *secure* cookie attribute to be effective, your application will need to enforce HTTPS.

(See also: RFC 6265, Section 4.1.2.5)

- **http_only** (*bool*) – The HttpOnly attribute limits the scope of the cookie to HTTP requests. In particular, the attribute instructs the user agent to omit the cookie when providing access to cookies via "non-HTTP" APIs. This is intended to mitigate some forms of cross-site scripting. (default: `True`)

  > **Note**
  >
  > HttpOnly cookies are not visible to javascript scripts in the browser. They are automatically sent to the server on javascript `XMLHttpRequest` or `Fetch` requests.

  (See also: RFC 6265, Section 4.1.2.6)

- **same_site** (*str*) – Helps protect against CSRF attacks by restricting when a cookie will be attached to the request by the user agent. When set to `'Strict'`, the cookie will only be sent along with "same-site" requests. If the value is `'Lax'`, the cookie will be sent with same-site requests, and with "cross-site" top-level navigations. If the value is `'None'`, the cookie will be sent with same-site and cross-site requests. Finally, when this attribute is not set on the cookie, the attribute will be treated as if it had been set to `'None'`.

  (See also: Same-Site RFC Draft)

- **partitioned** (*bool*) – Prevents cookies from being accessed from other subdomains. With partitioned enabled, a cookie set by https://3rd-party.example which is embedded inside https://site-a.example can no longer be accessed by https://site-b.example. While this attribute is not yet standardized, it is already used by Chrome.

  (See also: CHIPS)

  Added in version 4.0.

Raises

- **KeyError** – *name* is not a valid cookie name.
- **ValueError** – *value* is not a valid cookie value.

set_header(*name: str*, *value: str*) → None

Set a header for this response to a given value.

> **Warning**
>
> Calling this method overwrites any values already set for this header. To append an additional value for this header, use *append_header()* instead.

> **Warning**
>
> This method cannot be used to set cookies; instead, use *append_header()* or *set_cookie()*.

Parameters

- **name** (*str*) – Header name (case-insensitive). The name may contain only US-ASCII characters.
- **value** (*str*) – Value for the header. As with the header's name, the value may contain only US-ASCII characters.

Raises

ValueError – *name* cannot be `'Set-Cookie'`.

set_headers(*headers: Mapping[str, str] | Iterable[tuple[str, str]]*) → None

Set several headers at once.

This method can be used to set a collection of raw header names and values all at once.

Warning

Calling this method overwrites any existing values for the given header. If a list containing multiple instances of the same header is provided, only the last value will be used. To add multiple values to the response for a given header, see *append_header()*.

Warning

This method cannot be used to set cookies; instead, use *append_header()* or *set_cookie()*.

Parameters

headers (*Iterable[[str, str]]*) – An iterable of `[name, value]` two-member iterables, or a dict-like object that implements an `items()` method. Both *name* and *value* must be of type `str` and contain only US-ASCII characters.

Note

Falcon can process an iterable of tuples slightly faster than a dict.

Raises

ValueError – *headers* was not a `dict` or `list` of `tuple` or `Iterable[[str, str]]`.

set_stream(*stream:* AsyncReadableIO | *AsyncIterator[bytes]*, *content_length: int*) → None

Set both *stream* and *content_length*.

Although the *stream* and *content_length* properties may be set directly, using this method ensures *content_length* is not accidentally neglected when the length of the stream is known in advance. Using this method is also slightly more performant as compared to setting the properties individually.

Note

If the stream length is unknown, you can set *stream* directly, and ignore *content_length*. In this case, the ASGI server may choose to use chunked encoding for HTTP/1.1

Parameters

- **stream** – A readable, awaitable file-like object or async iterable that returns byte strings. If the object implements a close() method, it will be called after reading all of the data.
- **content_length** (*int*) – Length of the stream, used for the Content-Length header in the response.

property sse: SSEEmitter | None

A Server-Sent Event (SSE) emitter, implemented as an async iterator or generator that yields a series of of *falcon.asgi.SSEvent* instances. Each event will be serialized and sent to the client as HTML5 Server-Sent Events:

```
async def emitter():
    while True:
        some_event = await get_next_event()

        if not some_event:
            # Send an event consisting of a single "ping"
            #   comment to keep the connection alive.
            yield SSEvent()

            # Alternatively, one can simply yield None and
            #   a "ping" will also be sent as above.

            # yield

            continue

        yield SSEvent(json=some_event, retry=5000)

        # ...or

        yield SSEvent(data=b'something', event_id=some_id)

        # Alternatively, you may yield anything that implements
        #   a serialize() method that returns a byte string
        #   conforming to the SSE event stream format.

        # yield some_event

resp.sse = emitter()
```

Note

When the *sse* property is set, it supersedes both the *text* and *data* properties.

Note

When hosting an app that emits Server-Sent Events, the web server should be set with a relatively long keep-alive TTL to minimize the overhead of connection renegotiations.

status: str | int | http.HTTPStatus

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string (e.g., `'200 OK'`), or an `int`.

Note

The Falcon framework itself provides a number of constants for common status codes. They all start with the `HTTP_` prefix, as in: `falcon.HTTP_204`. (See also: *Status Codes*.)

property status_code: int

HTTP status code normalized from *status*.

When a code is assigned to this property, *status* is updated, and vice-versa. The status code can be useful when needing to check in middleware for codes that fall into a certain class, e.g.:

```
if resp.status_code >= 400:
    log.warning(f'returning error response: {resp.status_code}')
```

stream: *AsyncReadableIO* | AsyncIterator[bytes] | None

An async iterator or generator that yields a series of byte strings that will be streamed to the ASGI server as a series of "http.response.body" events. Falcon will assume the body is complete when the iterable is exhausted or as soon as it yields `None` rather than an instance of `bytes`:

```
async def producer():
    while True:
        data_chunk = await read_data()
        if not data_chunk:
            break

        yield data_chunk

resp.stream = producer
```

Alternatively, a file-like object may be used as long as it implements an awaitable `read()` method:

```
resp.stream = await aiofiles.open('resp_data.bin', 'rb')
```

If the object assigned to *stream* holds any resources (such as a file handle) that must be explicitly released, the object must implement a `close()` method. The `close()` method will be called after exhausting the iterable or file-like object.

> **Note**
>
> In order to be compatible with Python 3.7+ and PEP 479, async iterators must return `None` instead of raising `StopIteration`. This requirement does not apply to async generators (PEP 525).

> **Note**
>
> If the stream length is known in advance, you may wish to also set the Content-Length header on the response.

text: str | None

String representing response content.

> **Note**
>
> Falcon will encode the given text as UTF-8 in the response. If the content is already a byte string, use the *data* attribute instead (it's faster).

unset_cookie(*name: str*, *domain: str | None = None*, *path: str | None = None*, *same_site: str = 'Lax'*, *samesite: str | None = None*) → None

Unset a cookie in the response.

Clears the contents of the cookie, and instructs the user agent to immediately expire its own copy of the cookie.

> **Note**
>
> Modern browsers place restriction on cookies without the "same-site" cookie attribute set. To that end this attribute is set to `'Lax'` by this method.
>
> (See also: Same-Site warnings)

> **Warning**
>
> In order to successfully remove a cookie, both the path and the domain must match the values that were used when the cookie was created.

Parameters

name (*str*) – Cookie name

Keyword Arguments

- **domain** (*str*) – Restricts the cookie to a specific domain and any subdomains of that domain. By default, the user agent will return the cookie only to the origin server. When overriding this default behavior, the specified domain must include the origin server. Otherwise, the user agent will reject the cookie.

  > **Note**
  >
  > Cookies do not provide isolation by port, so the domain should not provide one. (See also: RFC 6265, Section 8.5)

  (See also: RFC 6265, Section 4.1.2.3)

- **path** (*str*) – Scopes the cookie to the given path plus any subdirectories under that path (the "/" character is interpreted as a directory separator). If the cookie does not specify a path, the user agent defaults to the path component of the requested URI.

  > **Warning**
  >
  > User agent interfaces do not always isolate cookies by path, and so this should not be considered an effective security measure.

  (See also: RFC 6265, Section 4.1.2.4)

- **same_site** (*str*) – Allows to override the default 'Lax' same_site setting for the unset cookie.

  Added in version 4.1.

- **samesite** (*str*) – Deprecated. Use *same_site* instead.

  Deprecated since version 4.1: Please use the `same_site` parameter instead.

property vary: str | None

Value to use for the Vary header.

Set this property to an iterable of header names. For a single asterisk or field value, simply pass a single-element `list` or `tuple`.

The "Vary" header field in a response describes what parts of a request message, aside from the method, Host header field, and request target, might influence the origin server's process for selecting and repre-

senting this response. The value consists of either a single asterisk ("*") or a list of header field names (case-insensitive).

(See also: RFC 7231, Section 7.1.4)

property viewable_as: str | None

Set an inline Content-Disposition header using the given filename.

The value will be used for the `filename` directive. For example, given `'report.pdf'`, the Content-Disposition header would be set to: `'inline; filename="report.pdf"'`.

As per RFC 6266 recommendations, non-ASCII filenames will be encoded using the `filename*` directive, whereas `filename` will contain the US ASCII fallback.

Added in version 3.1.

class falcon.asgi.**SSEvent**(*data: bytes | None = None*, *text: str | None = None*, *json: object | None = None*, *event: str | None = None*, *event_id: str | None = None*, *retry: int | None = None*, *comment: str | None = None*)

Represents a Server-Sent Event (SSE).

Instances of this class can be yielded by an async generator in order to send a series of Server-Sent Events to the user agent.

(See also: *falcon.asgi.Response.sse*)

Keyword Arguments

- **data** (*bytes*) – Raw byte string to use as the `data` field for the event message. Takes precedence over both *text* and *json*.
- **text** (*str*) – String to use for the `data` field in the message. Will be encoded as UTF-8 in the event. Takes precedence over *json*.
- **json** (*object*) – JSON-serializable object to be converted to JSON and used as the `data` field in the event message.
- **event** (*str*) – A string identifying the event type (AKA event name).
- **event_id** (*str*) – The event ID that the User Agent should use for the *EventSource* object's last event ID value.
- **retry** (*int*) – The reconnection time to use when attempting to send the event. This must be an integer, specifying the reconnection time in milliseconds.
- **comment** (*str*) – Comment to include in the event message; this is normally ignored by the user agent, but is useful when composing a periodic "ping" message to keep the connection alive. Since this is a common use case, a default "ping" comment will be included in any event that would otherwise be blank (i.e., one that does not specify any fields when initializing the *SSEvent* instance.)

comment: str | None

Comment to include in the event message.

This is normally ignored by the user agent, but is useful when composing a periodic "ping" message to keep the connection alive. Since this is a common use case, a default "ping" comment will be included in any event that would otherwise be blank (i.e., one that does not specify any of the fields when initializing the *SSEvent* instance.)

data: bytes | None

Raw byte string to use as the `data` field for the event message. Takes precedence over both *text* and *json*.

event: str | None

A string identifying the event type (AKA event name).

event_id: str | None

The event ID that the User Agent should use for the *EventSource* object's last event ID value.

json: object

JSON-serializable object to be converted to JSON and used as the `data` field in the event message.

retry: int | None

The reconnection time to use when attempting to send the event.

This must be an integer, specifying the reconnection time in milliseconds.

serialize(*handler:* BaseHandler | *None* = *None*) → bytes

Serialize this event to string.

Parameters

handler – Handler object that will be used to serialize the `json` attribute to string. When not provided, a default handler using the builtin JSON library will be used (default `None`).

Returns

string representation of this event.

Return type

bytes

text: str | None

String to use for the `data` field in the message. Will be encoded as UTF-8 in the event. Takes precedence over *json*.

5.3.3 WebSocket (ASGI Only)

Falcon builds upon the ASGI WebSocket Specification to provide a simple, no-nonsense WebSocket server implementation.

With support for both WebSocket and Server-Sent Events (SSE), Falcon facilitates real-time, event-oriented communication between an ASGI application and a web browser, mobile app, or other client application.

Note

See also *falcon.asgi.Response.sse* to learn more about Falcon's Server-Sent Event (SSE) support.

Usage

With Falcon you can easily add WebSocket support to any route in your ASGI app, simply by implementing an `on_websocket()` responder in the resource class for that route. As with regular HTTP requests, WebSocket flows can be augmented with middleware components and media handlers.

When a WebSocket handshake arrives (via a standard HTTP request), Falcon will first route it as usual to a specific resource class instance. Along the way, the following middleware methods will be invoked, if implemented on any middleware objects configured for the app:

```
from typing import Any
from falcon.asgi import Request, WebSocket


class SomeMiddleware:
    async def process_request_ws(self, req: Request, ws: WebSocket) -> None:
        """Process a WebSocket handshake request before routing it.

        Note:
            Because Falcon routes each request based on req.path, a
            request can be effectively re-routed by setting that
            attribute to a new value from within process_request().
```

(continues on next page)

(continued from previous page)

```
        Args:
            req: Request object that will eventually be
                passed into an on_websocket() responder method.
            ws: The WebSocket object that will be passed into
                on_websocket() after routing.
        """

    async def process_resource_ws(
        self,
        req: Request,
        ws: WebSocket,
        resource: object,
        params: dict[str, Any],
    ) -> None:
        """Process a WebSocket handshake request after routing.

        Note:
            This method is only called when the request matches
            a route to a resource.

        Args:
            req: Request object that will be passed to the
                routed responder.
            ws: WebSocket object that will be passed to the
                routed responder.
            resource: Resource object to which the request was
                routed.
            params: A dict-like object representing any additional
                params derived from the route's URI template fields,
                that will be passed to the resource's responder
                method as keyword arguments.
        """
```

If a route is found for the requested path, the framework will then check for a responder coroutine named `on_websocket()` on the target resource. If the responder is found, it is invoked in a similar manner to a regular `on_get()` responder, except that a *falcon.asgi.WebSocket* object is passed in, instead of an object of type *falcon.asgi.Response*.

For example, given a route that includes an `account_id` path parameter, the framework would expect an `on_websocket()` responder similar to this:

```
async def on_websocket(self, req: Request, ws: WebSocket, account_id: str) -> None:
    pass
```

Just like other HTTP requests, WebSocket connections can also be handled dynamically by *sinks*. In order to receive a *WebSocket* connection object, the sink method ought to define a *ws* keyword argument:

```
async def sink(
    req: Request,
    resp: Response | None,
    ws: WebSocket | None = None,
    **kwargs: Any,
) -> None:
    if ws is not None:
        # WebSocket connection (resp is None)
        ...
    else:
```

(continues on next page)

(continued from previous page)

```
        # Ordinary HTTP request (ws is None)
        ...
```

If no route or sink matches the path requested in the WebSocket handshake, control then passes to a default responder that simply raises an instance of *HTTPRouteNotFound*. By default, this error will be rendered as a 403 response with a 3404 close code. This behavior can be modified by adding a custom error handler (see also: *add_error_handler()*).

Similarly, if a route exists but the target resource does not implement an `on_websocket()` responder, the framework invokes a default responder that raises an instance of *HTTPMethodNotAllowed*. This class will be rendered by default as a 403 response with a 3405 close code.

Lost Connections

When the app attempts to receive a message from the client, the ASGI server emits a `disconnect` event if the connection has been lost for any reason. Falcon surfaces this event by raising an instance of *WebSocketDisconnected* to the caller.

On the other hand, the ASGI spec previously required the ASGI server to silently consume messages sent by the app after the connection has been lost (i.e., it should not be considered an error). Therefore, an endpoint that primarily streams outbound events to the client could continue consuming resources unnecessarily for some time after the connection is lost. This aspect has been rectified in the ASGI HTTP spec version `2.4`, and calling `send()` on a closed connection should now raise an error. Unfortunately, not all ASGI servers have adopted this new behavior uniformly yet.

As a workaround, Falcon implements a small incoming message queue that is used to detect a lost connection and then raise an instance of *WebSocketDisconnected* to the caller the next time it attempts to send a message. If your ASGI server of choice adheres to the spec version `2.4`, this receive queue can be safely disabled for a slight performance boost by setting *max_receive_queue* to `0` via *ws_options*. (We may revise this setting, and disable the queue by default in the future if our testing indicates that all major ASGI servers have caught up with the spec.)

Furthermore, even on non-compliant or older ASGI servers, this workaround is only necessary when the app itself does not consume messages from the client often enough to quickly detect when the connection is lost. Otherwise, Falcon's receive queue can also be disabled as described above.

Error Handling

Falcon handles errors raised by an `on_websocket()` responder in a similar way to errors raised by other responders, with the following caveats.

First, when calling a custom error handler, the framework will pass `None` for the *resp* argument, while the *WebSocket* object representing the current connection will be passed as a keyword argument named *ws*:

```
async def my_error_handler(req, resp, ex, params, ws=None):
    # When invoked as a result of an error being raised by an
    #   on_websocket() responder, resp will be None and
    #   ws will be the same falcon.asgi.WebSocket object that
    #   was passed into the responder.
    pass
```

Second, it's important to note that if no route matches the path in the WebSocket handshake request, or the matched resource does not implement an `on_websocket()` responder, the default HTTP error responders will be invoked, resulting in the request being denied with an `HTTP 403` response and a WebSocket close code of either `3404` (Not Found) or `3405` (Method Not Allowed). Generally speaking, if either a default responder or `on_websocket()` raises an instance of *HTTPError*, the default error handler will close the *WebSocket* connection with a framework close code derived by adding `3000` to the HTTP status code (e.g., `3404`).

Finally, in the case of a generic unhandled exception, a default error handler is invoked that will do its best to clean up the connection, closing it with the standard WebSocket close code `1011` (Internal Error). If your ASGI server does not support this code, the framework will use code `3011` instead; or you can customize it via the *error_close_code* property of *ws_options*.

As with any responder, the default error handlers for the app may be overridden via *add_error_handler()*.

Media Handlers

By default, *send_media()* and *receive_media()* will serialize to (and deserialize from) JSON for a TEXT payload, and to/from MessagePack for a BINARY payload (see also: *Built-in Media Handlers*).

Note

In order to use the default MessagePack handler, the extra `msgpack` package (version 0.5.2 or higher) must be installed in addition to `falcon` from PyPI:

```
$ pip install msgpack
```

WebSocket media handling can be customized by using *falcon.asgi.App.ws_options* to specify an alternative handler for one or both payload types, as in the following example.

```
# Let's say we want to use a faster JSON library. You could also use this
#   pattern to add serialization support for custom types that aren't
#   normally JSON-serializable out of the box.
class RapidJSONHandler(falcon.media.TextBaseHandlerWS):
    def serialize(self, media: object) -> str:
        return rapidjson.dumps(media, ensure_ascii=False)

    # The raw TEXT payload will be passed as a Unicode string
    def deserialize(self, payload: str) -> object:
        return rapidjson.loads(payload)


# And/or for binary mode we want to use CBOR:
class CBORHandler(media.BinaryBaseHandlerWS):
    def serialize(self, media: object) -> bytes:
        return cbor2.dumps(media)

    # The raw BINARY payload will be passed as a byte string
    def deserialize(self, payload: bytes) -> object:
        return cbor2.loads(payload)

app = falcon.asgi.App()

# Expected to (de)serialize from/to str
json_handler = RapidJSONHandler()
app.ws_options.media_handlers[falcon.WebSocketPayloadType.TEXT] = json_handler

# Expected to (de)serialize from/to bytes, bytearray, or memoryview
cbor_handler = ProtocolBuffersHandler()
app.ws_options.media_handlers[falcon.WebSocketPayloadType.BINARY] = cbor_handler
```

The `falcon` module defines the following `Enum` values for specifying the WebSocket payload type:

```
falcon.WebSocketPayloadType.TEXT
falcon.WebSocketPayloadType.BINARY
```

Extended Example

Here is a more comprehensive (albeit rather contrived) example that illustrates some of the different ways an application can interact with a WebSocket connection. This example also introduces some common WebSocket errors raised by the framework.

```
import falcon.asgi
import falcon.media


class SomeResource:

    # Get a paginated list of events via a regular HTTP request.
    #
    #   For small-scale, all-in-one apps, it may make sense to support
    #   both a regular HTTP interface and one based on WebSocket
    #   side-by-side in the same deployment. However, these two
    #   interaction models have very different performance characteristics,
    #   and so larger scale-out deployments may wish to specifically
    #   designate instance groups for one type of traffic vs. the
    #   other (although the actual applications may still be capable
    #   of handling both modes).
    #
    async def on_get(self, req: Request, account_id: str):
        pass

    # Push event stream to client. Note that the framework will pass
    #   parameters defined in the URI template as with HTTP method
    #   responders.
    async def on_websocket(self, req: Request, ws: WebSocket, account_id: str):

        # The HTTP request used to initiate the WebSocket handshake can be
        #   examined as needed.
        some_header_value = req.get_header('Some-Header')

        # Reject it?
        if some_condition:
            # If close() is called before accept() the code kwarg is
            #   ignored, if present, and the server returns a 403
            #   HTTP response without upgrading the connection.
            await ws.close()
            return

        # Examine subprotocols advertised by the client. Here let's just
        #   assume we only support wamp, so if the client doesn't advertise
        #   it we reject the connection.
        if 'wamp' not in ws.subprotocols:
            # If close() is not called explicitly, the framework will
            #   take care of it automatically with the default code (1000).
            return

        # If, after examining the connection info, you would like to accept
        #   it, simply call accept() as follows:
        try:
            await ws.accept(subprotocol='wamp')
        except WebSocketDisconnected:
            return
```

(continues on next page)

(continued from previous page)

```
        # Simply start sending messages to the client if this is an event
        #   feed endpoint.
        while True:
            try:
                event = await my_next_event()

                # Send an instance of str as a WebSocket TEXT (0x01) payload
                await ws.send_text(event)

                # Send an instance of bytes, bytearray, or memoryview as a
                #   WebSocket BINARY (0x02) payload.
                await ws.send_data(event)

                # Or if you want it to be serialized to JSON (by default; can
                #   be customized via app.ws_options.media_handlers):
                await ws.send_media(event)  # Defaults to WebSocketPayloadType.TEXT
            except WebSocketDisconnected:
                # Do any necessary cleanup, then bail out
                return

        # ...or loop like this to implement a simple request-response protocol
        while True:
            try:
                # Use this if you expect a WebSocket TEXT (0x01) payload,
                #   decoded from UTF-8 to a Unicode string.
                payload_str = await ws.receive_text()

                # Or if you are expecting a WebSocket BINARY (0x02) payload,
                #   in which case you will end up with a byte string result:
                payload_bytes = await ws.receive_data()

                # Or if you want to get a serialized media object (defaults to
                #   JSON deserialization of text payloads, and MessagePack
                #   deserialization for BINARY payloads, but this can be
                #   customized via app.ws_options.media_handlers).
                media_object = await ws.receive_media()

            except WebSocketDisconnected:
                # Do any necessary cleanup, then bail out
                return
            except TypeError:
                # The received message payload was not of the expected
                #   type (e.g., got BINARY when TEXT was expected).
                pass
            except json.JSONDecodeError:
                # The default media deserializer uses the json standard
                #   library, so you might see this error raised as well.
                pass

            # At any time, you may decide to close the websocket. If the
            #   socket is already closed, this call does nothing (it will
            #   not raise an error.)
            if we_are_so_done_with_this_conversation():
                # https://developer.mozilla.org/en-US/docs/Web/API/CloseEvent
                await ws.close(code=1000)
                return
```

(continues on next page)

(continued from previous page)

```
        try:
            # Here we are sending as a binary (0x02) payload type, which
            #   will go find the handler configured for that (defaults to
            #   MessagePack which assumes you've also installed that
            #   package, but this can be customized as mentioned above.')
            await ws.send_media(
                {'event': 'message'},
                payload_type=WebSocketPayloadType.BINARY,
            )

        except WebSocketDisconnected:
            # Do any necessary cleanup, then bail out. If ws.close() was
            #   not already called by the app, the framework will take
            #   care of it.

            # NOTE: If you do not handle this exception, it will be
            #   bubbled up to a default error handler that simply
            #   logs the message as a warning and then closes the
            #   server side of the connection. This handler can be
            #   overridden as with any other error handler for the app.

            return

    # ...or run a couple of different loops in parallel to support
    #  independent bidirectional message streams.

    messages = collections.deque()

    async def sink():
        while True:
            try:
                message = await ws.receive_text()
            except falcon.WebSocketDisconnected:
                break

            messages.append(message)

    sink_task = falcon.create_task(sink())

    while not sink_task.done():
        while ws.ready and not messages and not sink_task.done():
            await asyncio.sleep(0)

        try:
            await ws.send_text(messages.popleft())
        except falcon.WebSocketDisconnected:
            break

    sink_task.cancel()
    try:
        await sink_task
    except asyncio.CancelledError:
        pass
```

(continues on next page)

(continued from previous page)

```
class SomeMiddleware:
    async def process_request_ws(self, req: Request, ws: WebSocket):
        # This will be called for the HTTP request that initiates the
        #   WebSocket handshake before routing.
        pass

    async def process_resource_ws(self, req: Request, ws: WebSocket, resource,␣
↪params):
        # This will be called for the HTTP request that initiates the
        #   WebSocket handshake after routing (if a route matches the
        #   request).
        pass


app = falcon.asgi.App(middleware=SomeMiddleware())
app.add_route('/{account_id}/messages', SomeResource())
```

> **Tip**
>
> If you prefer to learn by doing, feel free to continue experimenting along the lines of our *WebSocket tutorial*!

Testing

Falcon's testing framework includes support for simulating WebSocket connections with the `falcon.testing.ASGIConductor` class, as demonstrated in the following example.

```
# This context manages the ASGI app lifecycle, including lifespan events
async with testing.ASGIConductor(some_app) as c:
    async def post_events():
        for i in range(100):
            await c.simulate_post('/events', json={'id': i}):
            await asyncio.sleep(0.01)

    async def get_events_ws():
        # Simulate a WebSocket connection
        async with c.simulate_ws('/events') as ws:
            while some_condition:
                message = await ws.receive_text()

    asyncio.gather(post_events(), get_events_ws())
```

See also: `simulate_ws()`.

Reference

WebSocket Class

The framework passes an instance of the following class into the `on_websocket()` responder. Conceptually, this class takes the place of the `falcon.asgi.Response` class for WebSocket connections.

class falcon.asgi.**WebSocket**(*ver: str*, *scope: dict[str, Any]*, *receive: Callable[[], Awaitable[Mapping[str, Any]]]*, *send: Callable[[dict[str, Any]], Awaitable[None]]*, *media_handlers: Mapping[WebSocketPayloadType,* BinaryBaseHandlerWS | TextBaseHandlerWS*]*, *max_receive_queue: int*, *default_close_reasons: dict[int, str]*)

Represents a single WebSocket connection with a client.

async accept(*subprotocol: str | None = None, headers: Mapping[str, str] | Iterable[tuple[str, str]] | None = None*) → None

Accept the incoming WebSocket connection.

If, after examining the connection's attributes (headers, advertised subprotocols, etc.) the request should be accepted, the responder must first await this coroutine method to finalize the WebSocket handshake. Alternatively, the responder may deny the connection request by awaiting the `close()` method.

Keyword Arguments

- **subprotocol** (*str*) – The subprotocol the app wishes to accept, out of the list of protocols that the client suggested. If more than one of the suggested protocols is acceptable, the first one in the list from the client should be selected (see also: `subprotocols`).

  When left unspecified, a Sec-WebSocket-Protocol header will not be included in the response to the client. The client may choose to abandon the connection in this case, if it does not receive an explicit protocol selection.

- **headers** (*HeaderArg*) – An iterable of `(name: str, value: str)` two-item iterables, representing a collection of HTTP headers to include in the handshake response. Both *name* and *value* must be of type `str` and contain only US-ASCII characters.

  Alternatively, a dict-like object may be passed that implements an `items()` method.

  Note

  This argument is only supported for ASGI servers that implement spec version 2.1 or better. If an app needs to be compatible with multiple ASGI servers, it can reference the `supports_accept_headers` property to determine if the hosting server supports this feature.

async close(*code: int | None = None, reason: str | None = None*) → None

Close the WebSocket connection.

This coroutine method sends a WebSocket `CloseEvent` to the client and then proceeds to actually close the connection.

The responder can also use this method to deny a connection request simply by awaiting it instead of `accept()`. In this case, the client will receive an HTTP 403 response to the handshake.

Keyword Arguments

- **code** (*int*) – The close code to use for the `CloseEvent` (default 1000). See also: https://developer.mozilla.org/en-US/docs/Web/API/CloseEvent/code.
- **reason** (*str*) – The string reason indicating why the server closed the connection. See also: https://developer.mozilla.org/en-US/docs/Web/API/CloseEvent/reason.

  If there is no reason provided, Falcon will try to automatically look it up from the above *code* and `default_close_reasons`.

Note

The close *reason* will only be propagated if the ASGI app server supports this.

Version `2.3+` of the HTTP & WebSocket ASGI protocol is required for *reason*.

property closed: bool

`True` if the WebSocket connection has been closed by the server or the client has disconnected.

property ready: bool

`True` if the WebSocket connection has been accepted and the client is still connected, `False` otherwise.

async receive_data() → bytes

Receive a message from the client with a binary data payload.

Awaiting this coroutine will block until a message is available or the WebSocket is disconnected.

async receive_media() → object

Receive a deserialized object from the client.

The incoming payload type determines the media handler that will be used to deserialize the object (see also: *Media Handlers*).

async receive_text() → str

Receive a message from the client with a Unicode string payload.

Awaiting this coroutine will block until a message is available or the WebSocket is disconnected.

async send_data(*payload: bytes | bytearray | memoryview*) → None

Send a message to the client with a binary data payload.

Parameters

payload (*Union[bytes, bytearray, memoryview]*) – The binary data to send.

async send_media(*media: object*, *payload_type: WebSocketPayloadType = WebSocketPayloadType.TEXT*) → None

Send a serializable object to the client.

The payload type determines the media handler that will be used to serialize the given object (see also: *Media Handlers*).

Parameters

media (*object*) – The object to send.

Keyword Arguments

payload_type (*falcon.WebSocketPayloadType*) – The payload type to use for the message (default `falcon.WebSocketPayloadType.TEXT`).

Must be one of:

```
falcon.WebSocketPayloadType.TEXT
falcon.WebSocketPayloadType.BINARY
```

async send_text(*payload: str*) → None

Send a message to the client with a Unicode string payload.

Parameters

payload (*str*) – The string to send.

subprotocols: tuple[str, ...]

The list of subprotocol strings advertised by the client, or an empty tuple if no subprotocols were specified.

property supports_accept_headers: bool

`True` if the ASGI server hosting the app supports sending headers when accepting the WebSocket connection, `False` otherwise.

property unaccepted: bool

`True` if the WebSocket connection has not yet been accepted, `False` otherwise.

Built-in Media Handlers

class falcon.media.**TextBaseHandlerWS**

Abstract Base Class for a WebSocket TEXT media handler.

deserialize(*payload: str*) → object

Deserialize TEXT payloads from a Unicode string.

By default, this method raises an instance of `NotImplementedError`. Therefore, it must be overridden if the child class wishes to support deserialization from TEXT (0x01) message payloads.

Parameters
payload (*str*) – Message payload to deserialize.

Returns
A deserialized object.

Return type
object

serialize(*media: object*) → str

Serialize the media object to a Unicode string.

By default, this method raises an instance of `NotImplementedError`. Therefore, it must be overridden if the child class wishes to support serialization to TEXT (0x01) message payloads.

Parameters
media (*object*) – A serializable object.

Returns
The resulting serialized string from the input object.

Return type
str

class falcon.media.**BinaryBaseHandlerWS**

Abstract Base Class for a WebSocket BINARY media handler.

deserialize(*payload: bytes*) → object

Deserialize BINARY payloads from a byte string.

By default, this method raises an instance of `NotImplementedError`. Therefore, it must be overridden if the child class wishes to support deserialization from BINARY (0x02) message payloads.

Parameters
payload (*bytes*) – Message payload to deserialize.

Returns
A deserialized object.

Return type
object

serialize(*media: object*) → bytes | bytearray | memoryview

Serialize the media object to a byte string.

By default, this method raises an instance of `NotImplementedError`. Therefore, it must be overridden if the child class wishes to support serialization to BINARY (0x02) message payloads.

Parameters
media (*object*) – A serializable object.

Returns
The resulting serialized byte string from the input object. May be an instance of `bytes`, `bytearray`, or `memoryview`.

Return type
bytes

class falcon.media.**JSONHandlerWS**(*dumps: Callable[[Any], str] | None = None*, *loads: Callable[[str], Any] | None = None*)

WebSocket media handler for de(serializing) JSON to/from TEXT payloads.

This handler uses Python's standard `json` library by default, but can be easily configured to use any of a number of third-party JSON libraries, depending on your needs. For example, you can often realize a significant performance boost under CPython by using an alternative library. Good options in this respect include *orjson*, *python-rapidjson*, and *mujson*.

> **Note**
>
> If you are deploying to PyPy, we recommend sticking with the standard library's JSON implementation, since it will be faster in most cases as compared to a third-party library.

Overriding the default JSON implementation is simply a matter of specifying the desired `dumps` and `loads` functions:

```
import falcon
from falcon import media

import rapidjson

json_handler = media.JSONHandlerWS(
    dumps=rapidjson.dumps,
    loads=rapidjson.loads,
)

app = falcon.asgi.App()
app.ws_options.media_handlers[falcon.WebSocketPayloadType.TEXT] = json_handler
```

By default, `ensure_ascii` is passed to the `json.dumps` function. If you override the `dumps` function, you will need to explicitly set `ensure_ascii` to `False` in order to enable the serialization of Unicode characters to UTF-8. This is easily done by using `functools.partial` to apply the desired keyword argument. In fact, you can use this same technique to customize any option supported by the `dumps` and `loads` functions:

```
from functools import partial

from falcon import media
import rapidjson

json_handler = media.JSONHandlerWS(
    dumps=partial(
        rapidjson.dumps,
        ensure_ascii=False, sort_keys=True
    ),
)
```

Keyword Arguments

- **dumps** (*func*) – Function to use when serializing JSON.
- **loads** (*func*) – Function to use when deserializing JSON.

class falcon.media.**MessagePackHandlerWS**

WebSocket media handler for de(serializing) MessagePack to/from BINARY payloads.

This handler uses `msgpack.unpackb()` and `msgpack.packb()`. The MessagePack `bin` type is used to distinguish between Unicode strings (of type `str`) and byte strings (of type `bytes`).

> **Note**
>
> This handler requires the extra `msgpack` package (version 0.5.2 or higher), which must be installed in addition to `falcon` from PyPI:
>
> ```
> $ pip install msgpack
> ```

Error Types

class falcon.**WebSocketDisconnected**(*code: int | None = None*)

The websocket connection is lost.

This error is raised when attempting to perform an operation on the WebSocket and it is determined that either the client has closed the connection, the server closed the connection, or the socket has otherwise been lost.

Keyword Arguments
code (*int*) – The WebSocket close code, as per the WebSocket spec (default `1000`).

code: int

The WebSocket close code, as per the WebSocket spec.

class falcon.**WebSocketPathNotFound**(*code: int | None = None*)

No route could be found for the requested path.

A simulated WebSocket connection was attempted but the path specified in the handshake request did not match any of the app's routes.

class falcon.**WebSocketHandlerNotFound**(*code: int | None = None*)

The routed resource does not contain an `on_websocket()` handler.

class falcon.**WebSocketServerError**(*code: int | None = None*)

The server encountered an unexpected error.

class falcon.**PayloadTypeError**

The WebSocket message payload was not of the expected type.

Options

class falcon.asgi.**WebSocketOptions**

Defines a set of configurable WebSocket options.

An instance of this class is exposed via *falcon.asgi.App.ws_options* for configuring certain *WebSocket* behaviors.

default_close_reasons: dict[int, str]

A default mapping between the Websocket close code, and the reason why the connection is closed. Close codes corresponding to HTTP errors are also included in this mapping.

error_close_code: int

The WebSocket close code to use when an unhandled error is raised while handling a WebSocket connection (default `1011`).

For a list of valid close codes and ranges, see also: https://tools.ietf.org/html/rfc6455#section-7.4.

max_receive_queue: int

The maximum number of incoming messages to enqueue if the reception rate exceeds the consumption rate of the application (default `4`).

When this limit is reached, the framework will wait to accept new messages from the ASGI server until the application is able to catch up.

This limit applies to Falcon's incoming message queue, and should generally be kept small since the ASGI server maintains its own receive queue. Falcon's queue can be disabled altogether by setting *max_receive_queue* to `0` (see also: *Lost Connections*).

media_handlers: dict[WebSocketPayloadType, *TextBaseHandlerWS* | *BinaryBaseHandlerWS*]

A dict-like object for configuring media handlers according to the WebSocket payload type (TEXT vs. BINARY) of a given message.

See also: *Media Handlers*.

5.3.4 Cookies

This page describes the API provided by Falcon to manipulate cookies.

Getting Cookies

Cookies can be read from a request either via the *get_cookie_values()* method or the *cookies* attribute on the *Request* object. Generally speaking, the *get_cookie_values()* method should be used unless you need a collection of all the cookies in the request.

> **Note**
>
> *falcon.asgi.Request* implements the same cookie methods and properties as *falcon.Request*.

Here's an example showing how to get cookies from a request:

WSGI

```
class Resource:
    def on_get(self, req, resp):

        # Get a dict of name/value cookie pairs.
        cookies = req.cookies

        my_cookie_values = req.get_cookie_values('my_cookie')

        if my_cookie_values:
            # NOTE: If there are multiple values set for the cookie, you
            #   will need to choose how to handle the additional values.
            v = my_cookie_values[0]
```

ASGI

```
class Resource:
    async def on_get(self, req, resp):

        # Get a dict of name/value cookie pairs.
        cookies = req.cookies

        # NOTE: Since get_cookie_values() is synchronous, it does
        #   not need to be await'd.
        my_cookie_values = req.get_cookie_values('my_cookie')

        if my_cookie_values:
            # NOTE: If there are multiple values set for the cookie, you
```

(continues on next page)

(continued from previous page)

```
            #   will need to choose how to handle the additional values.
            v = my_cookie_values[0]
```

Setting Cookies

Setting cookies on a response may be done either via *set_cookie()* or *append_header()*.

One of these methods should be used instead of *set_header()*. With *set_header()* you cannot set multiple headers with the same name (which is how multiple cookies are sent to the client).

Note

falcon.asgi.Request implements the same cookie methods and properties as *falcon.Request*. The ASGI versions of *set_cookie()* and *append_header()* are synchronous, so they do not need to be `await`'d.

Simple example:

```
# Set the cookie 'my_cookie' to the value 'my cookie value'
resp.set_cookie('my_cookie', 'my cookie value')
```

You can of course also set the domain, path and lifetime of the cookie.

```
# Set the maximum age of the cookie to 10 minutes (600 seconds)
#   and the cookie's domain to 'example.com'
resp.set_cookie('my_cookie', 'my cookie value',
                max_age=600, domain='example.com')
```

You can also instruct the client to remove a cookie with the *unset_cookie()* method:

```
# Set a cookie in middleware or in a previous request.
resp.set_cookie('my_cookie', 'my cookie value')

# -- snip --

# Clear the cookie for the current request and instruct the user agent
#   to expire its own copy of the cookie (if any).
resp.unset_cookie('my_cookie')
```

The Secure Attribute

By default, Falcon sets the *secure* attribute for cookies. This instructs the client to never transmit the cookie in the clear over HTTP, in order to protect any sensitive data that cookie might contain. If a cookie is set, and a subsequent request is made over HTTP (rather than HTTPS), the client will not include that cookie in the request.

Warning

For this attribute to be effective, your web server or load balancer will need to enforce HTTPS when setting the cookie, as well as in all subsequent requests that require the cookie to be sent back from the client.

When running your application in a development environment, you can disable this default behavior by setting *secure_cookies_by_default* to `False` via *falcon.App.resp_options* or `falcon.asgi.App.resp_options`. This lets you test your app locally without having to set up TLS. You can make this option configurable to easily switch between development and production environments.

See also: RFC 6265, Section 4.1.2.5

The SameSite Attribute

The *SameSite* attribute may be set on a cookie using the *set_cookie()* method. It is generally a good idea to at least set this attribute to `'Lax'` in order to mitigate CSRF attacks.

Currently, *set_cookie()* does not set *SameSite* by default, although this may change in a future release.

When unsetting a cookie, *unset_cookie()*, the default *SameSite* setting of the unset cookie is `'Lax'`, but can be changed by setting the 'same_site' kwarg.

The Partitioned Attribute

Starting from Q1 2024, Google Chrome started to phase out support for third-party cookies. If your site is relying on cross-site cookies, it might be necessary to set the `Partitioned` attribute. `Partitioned` usually requires the *Secure* attribute to be set. While this is not enforced by Falcon, the framework does set `Secure` by default, unless specified otherwise (see also *secure_cookies_by_default*).

Currently, *set_cookie()* does not set `Partitioned` automatically depending on other attributes (like `SameSite`), although this may change in a future release.

> **Note**
>
> The standard `http.cookies` module does not support the *Partitioned* attribute in versions prior to Python 3.14. Therefore, Falcon performs a simple monkey-patch on the standard library module to backport this feature for apps running on older Python versions.

5.3.5 Status Codes

Falcon provides a list of constants for common HTTP response status codes.

For example:

```
# Override the default "200 OK" response status
resp.status = falcon.HTTP_409
```

Or, using the more verbose name:

```
resp.status = falcon.HTTP_CONFLICT
```

Using these constants helps avoid typos and cuts down on the number of string objects that must be created when preparing responses. However, starting with Falcon version 3.0, an LRU is used to enable efficient use of `http.HTTPStatus` and bare `int` codes as well. Essentially, the WSGI flavor of *resp.status* can now be set to anything that *code_to_http_status()* accepts:

```
resp.status = 201
```

ASGI *resp.status* also supports the same broad selection of status types via *http_status_to_code()*:

```
resp.status = HTTPStatus.CREATED
```

> **Note**
>
> One notable difference between WSGI and ASGI is that the latter's HTTP send event defines `status` as an `int` code. Effectively, this precludes rendering a custom status line string or a non-standard status code that the ASGI app server is unaware of.

Falcon also provides a generic *HTTPStatus* exception class. Simply raise an instance of this class from any hook, middleware, or a responder to stop handling the request and skip to the response handling. It takes status, additional headers and body as input arguments.

HTTPStatus

class falcon.**HTTPStatus**(*status: ResponseStatus*, *headers: HeaderArg | None = None*, *text: str | None = None*)

Represents a generic HTTP status.

Raise an instance of this class from a hook, middleware, or responder to short-circuit request processing in a manner similar to `falcon.HTTPError`, but for non-error status codes.

Parameters

- **status** (*Union[str, int]*) – HTTP status code or line (e.g., `'400 Bad Request'`). This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.
- **headers** (*dict*) – Extra headers to add to the response.
- **text** (*str*) – String representing response content. Falcon will encode this value as UTF-8 in the response.

headers: HeaderArg | None

Extra headers to add to the response.

status: ResponseStatus

The HTTP status line or integer code for the status that this exception represents.

property status_code: int

HTTP status code normalized from *status*.

text: str | None

String representing response content. Falcon will encode this value as UTF-8 in the response.

1xx Informational

```
HTTP_CONTINUE = HTTP_100
HTTP_SWITCHING_PROTOCOLS = HTTP_101
HTTP_PROCESSING = HTTP_102
HTTP_EARLY_HINTS = HTTP_103

HTTP_100 = '100 Continue'
HTTP_101 = '101 Switching Protocols'
HTTP_102 = '102 Processing'
HTTP_103 = '103 Early Hints'
```

2xx Success

```
HTTP_OK = HTTP_200
HTTP_CREATED = HTTP_201
HTTP_ACCEPTED = HTTP_202
HTTP_NON_AUTHORITATIVE_INFORMATION = HTTP_203
HTTP_NO_CONTENT = HTTP_204
HTTP_RESET_CONTENT = HTTP_205
HTTP_PARTIAL_CONTENT = HTTP_206
HTTP_MULTI_STATUS = HTTP_207
HTTP_ALREADY_REPORTED = HTTP_208
HTTP_IM_USED = HTTP_226

HTTP_200 = '200 OK'
HTTP_201 = '201 Created'
HTTP_202 = '202 Accepted'
HTTP_203 = '203 Non-Authoritative Information'
```

(continues on next page)

(continued from previous page)

```
HTTP_204 = '204 No Content'
HTTP_205 = '205 Reset Content'
HTTP_206 = '206 Partial Content'
HTTP_207 = '207 Multi-Status'
HTTP_208 = '208 Already Reported'
HTTP_226 = '226 IM Used'
```

3xx Redirection

```
HTTP_MULTIPLE_CHOICES = HTTP_300
HTTP_MOVED_PERMANENTLY = HTTP_301
HTTP_FOUND = HTTP_302
HTTP_SEE_OTHER = HTTP_303
HTTP_NOT_MODIFIED = HTTP_304
HTTP_USE_PROXY = HTTP_305
HTTP_TEMPORARY_REDIRECT = HTTP_307
HTTP_PERMANENT_REDIRECT = HTTP_308

HTTP_300 = '300 Multiple Choices'
HTTP_301 = '301 Moved Permanently'
HTTP_302 = '302 Found'
HTTP_303 = '303 See Other'
HTTP_304 = '304 Not Modified'
HTTP_305 = '305 Use Proxy'
HTTP_307 = '307 Temporary Redirect'
HTTP_308 = '308 Permanent Redirect'
```

4xx Client Error

```
HTTP_BAD_REQUEST = HTTP_400
HTTP_UNAUTHORIZED = HTTP_401  # <-- Really means "unauthenticated"
HTTP_PAYMENT_REQUIRED = HTTP_402
HTTP_FORBIDDEN = HTTP_403  # <-- Really means "unauthorized"
HTTP_NOT_FOUND = HTTP_404
HTTP_METHOD_NOT_ALLOWED = HTTP_405
HTTP_NOT_ACCEPTABLE = HTTP_406
HTTP_PROXY_AUTHENTICATION_REQUIRED = HTTP_407
HTTP_REQUEST_TIMEOUT = HTTP_408
HTTP_CONFLICT = HTTP_409
HTTP_GONE = HTTP_410
HTTP_LENGTH_REQUIRED = HTTP_411
HTTP_PRECONDITION_FAILED = HTTP_412
HTTP_CONTENT_TOO_LARGE = HTTP_413
HTTP_REQUEST_URI_TOO_LONG = HTTP_414
HTTP_UNSUPPORTED_MEDIA_TYPE = HTTP_415
HTTP_REQUESTED_RANGE_NOT_SATISFIABLE = HTTP_416
HTTP_EXPECTATION_FAILED = HTTP_417
HTTP_IM_A_TEAPOT = HTTP_418
HTTP_MISDIRECTED_REQUEST = HTTP_421
HTTP_UNPROCESSABLE_ENTITY = HTTP_422
HTTP_LOCKED = HTTP_423
HTTP_FAILED_DEPENDENCY = HTTP_424
HTTP_UPGRADE_REQUIRED = HTTP_426
HTTP_PRECONDITION_REQUIRED = HTTP_428
HTTP_TOO_MANY_REQUESTS = HTTP_429
```

(continues on next page)

(continued from previous page)

```
HTTP_REQUEST_HEADER_FIELDS_TOO_LARGE = HTTP_431
HTTP_UNAVAILABLE_FOR_LEGAL_REASONS = HTTP_451

HTTP_400 = '400 Bad Request'
HTTP_401 = '401 Unauthorized'  # <-- Really means "unauthenticated"
HTTP_402 = '402 Payment Required'
HTTP_403 = '403 Forbidden'  # <-- Really means "unauthorized"
HTTP_404 = '404 Not Found'
HTTP_405 = '405 Method Not Allowed'
HTTP_406 = '406 Not Acceptable'
HTTP_407 = '407 Proxy Authentication Required'
HTTP_408 = '408 Request Timeout'
HTTP_409 = '409 Conflict'
HTTP_410 = '410 Gone'
HTTP_411 = '411 Length Required'
HTTP_412 = '412 Precondition Failed'
HTTP_413 = '413 Content Too Large'
HTTP_414 = '414 URI Too Long'
HTTP_415 = '415 Unsupported Media Type'
HTTP_416 = '416 Range Not Satisfiable'
HTTP_417 = '417 Expectation Failed'
HTTP_418 = "418 I'm a teapot"
HTTP_421 = '421 Misdirected Request'
HTTP_422 = "422 Unprocessable Entity"
HTTP_423 = '423 Locked'
HTTP_424 = '424 Failed Dependency'
HTTP_426 = '426 Upgrade Required'
HTTP_428 = '428 Precondition Required'
HTTP_429 = '429 Too Many Requests'
HTTP_431 = '431 Request Header Fields Too Large'
HTTP_451 = '451 Unavailable For Legal Reasons'
```

5xx Server Error

```
HTTP_INTERNAL_SERVER_ERROR = HTTP_500
HTTP_NOT_IMPLEMENTED = HTTP_501
HTTP_BAD_GATEWAY = HTTP_502
HTTP_SERVICE_UNAVAILABLE = HTTP_503
HTTP_GATEWAY_TIMEOUT = HTTP_504
HTTP_HTTP_VERSION_NOT_SUPPORTED = HTTP_505
HTTP_VARIANT_ALSO_NEGOTIATES = HTTP_506
HTTP_INSUFFICIENT_STORAGE = HTTP_507
HTTP_LOOP_DETECTED = HTTP_508
HTTP_NOT_EXTENDED = HTTP_510
HTTP_NETWORK_AUTHENTICATION_REQUIRED = HTTP_511

HTTP_500 = '500 Internal Server Error'
HTTP_501 = '501 Not Implemented'
HTTP_502 = '502 Bad Gateway'
HTTP_503 = '503 Service Unavailable'
HTTP_504 = '504 Gateway Timeout'
HTTP_505 = '505 HTTP Version Not Supported'
HTTP_506 = '506 Variant Also Negotiates'
HTTP_507 = '507 Insufficient Storage'
HTTP_508 = '508 Loop Detected'
HTTP_510 = '510 Not Extended'
```

(continues on next page)

(continued from previous page)

```
HTTP_511 = '511 Network Authentication Required'
```

5.3.6 Error Handling

When it comes to error handling, you can always directly set the error status, appropriate response headers, and error body using the `resp` object. However, Falcon tries to make things a little easier by providing a set of error classes you can raise when something goes wrong. All of these classes inherit from *HTTPError*.

Falcon will convert any instance or subclass of *HTTPError* raised by a responder, hook, or middleware component into an appropriate HTTP response. The default error serializer supports both JSON and XML. If the client indicates acceptance of both JSON and XML with equal weight, JSON will be chosen. Other media types may be supported by overriding the default serializer via *set_error_serializer()*.

> **Note**
>
> If a custom media type is used and the type includes a "+json" or "+xml" suffix, the default serializer will convert the error to JSON or XML, respectively.

To customize what data is passed to the serializer, subclass *HTTPError* or any of its child classes, and override the *to_dict()* method. To also support XML, override the *to_xml()* method. For example:

```
class HTTPNotAcceptable(falcon.HTTPNotAcceptable):

    def __init__(self, acceptable):
        description = (
            'Please see "acceptable" for a list of media types '
            'and profiles that are currently supported.'
        )

        super().__init__(description=description)
        self._acceptable = acceptable

    def to_dict(self, obj_type=dict):
        result = super().to_dict(obj_type)
        result['acceptable'] = self._acceptable
        return result
```

All classes are available directly in the `falcon` package namespace:

WSGI

```
import falcon

class MessageResource:
    def on_get(self, req, resp):

        # -- snip --

        raise falcon.HTTPBadRequest(
            title="TTL Out of Range",
            description="The message's TTL must be between 60 and 300 seconds,␣
↪inclusive."
        )

        # -- snip --
```

ASGI

```
import falcon

class MessageResource:
    async def on_get(self, req, resp):

        # -- snip --

        raise falcon.HTTPBadRequest(
            title="TTL Out of Range",
            description="The message's TTL must be between 60 and 300 seconds,␣
↪inclusive."
        )

        # -- snip --
```

Note also that any exception (not just instances of *HTTPError*) can be caught, logged, and otherwise handled at the global level by registering one or more custom error handlers. See also *add_error_handler()* to learn more about this feature.

Note

By default, any uncaught exceptions will return an HTTP 500 response and log details of the exception to `wsgi.errors`.

Base Class

class falcon.**HTTPError**(*status: ResponseStatus*, ** (Keyword-only parameters separator (PEP 3102))*, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, *href: str | None = None*, *href_text: str | None = None*, *code: int | None = None*)

Represents a generic HTTP error.

Raise an instance or subclass of `HTTPError` to have Falcon return a formatted error response and an appropriate HTTP status code to the client when something goes wrong. JSON and XML media types are supported by default.

To customize the error presentation, implement a custom error serializer and set it on the *App* instance via *set_error_serializer()*.

To customize what data is passed to the serializer, subclass `HTTPError` and override the `to_dict()` method (`to_json()` is implemented via `to_dict()`).

status is the only positional argument allowed, the other arguments are defined as keyword-only.

Parameters

status (*Union[str, int]*) – HTTP status code or line (e.g., `'400 Bad Request'`). This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

Keyword Arguments

- **title** (*str*) – Human-friendly error title. If not provided, defaults to the HTTP status line as determined by the `status` argument.
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two (default `None`).
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

> **Note**
>
> The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

> **Note**
>
> Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'App documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: **int** | **None**

An internal application code that a user can reference when requesting support for the error.

description: **str** | **None**

Description of the error to send to the client.

headers: **HeaderArg** | **None**

Extra headers to add to the response.

link: **Link** | **None**

An href that the client can provide to the user for getting help.

status: **ResponseStatus**

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

property status_code: **int**

HTTP status code normalized from the `status` argument passed to the initializer.

title: **str**

Error title to send to the client.

Derived from the `status` if not provided.

to_dict(*obj_type: type[MutableMapping[str, str | int | None | Link]] = <class 'dict'>*) → MutableMapping[str, str | int | None | Link]

Return a basic dictionary representing the error.

This method can be useful when serializing the error to hash-like media types, such as YAML, JSON, and MessagePack.

Parameters

obj_type – A dict-like type that will be used to store the error information (default `dict`).

Returns

A dictionary populated with the error's title, description, etc.

Return type

dict

to_json(*handler:* BaseHandler | *None* = *None*) → bytes

Return a JSON representation of the error.

Parameters

handler – Handler object that will be used to serialize the representation of this error to JSON. When not provided, a default handler using the builtin JSON library will be used (default `None`).

Returns

A JSON document for the error.

Return type

bytes

to_xml() → bytes

Return an XML-encoded representation of the error.

Returns

An XML document for the error.

Return type

bytes

Deprecated since version 4.0: Automatic error serialization to XML is deprecated. Please serialize the output of `to_dict()` to XML instead.

Predefined Errors

class falcon.**HTTPBadRequest**(*, *title: str* | *None* = *None*, *description: str* | *None* = *None*, *headers: HeaderArg* | *None* = *None*, ***kwargs: HTTPErrorKeywordArguments*)

400 Bad Request.

The server cannot or will not process the request due to something that is perceived to be a client error (e.g., malformed request syntax, invalid request message framing, or deceptive request routing).

(See also: RFC 7231, Section 6.5.1)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Error title (default '400 Bad Request').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note

  The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  Note

  Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.

- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: int | None

An internal application code that a user can reference when requesting support for the error.

description: str | None

Description of the error to send to the client.

headers: HeaderArg | None

Extra headers to add to the response.

link: Link | None

An href that the client can provide to the user for getting help.

status: ResponseStatus

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: str

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPInvalidHeader**(*msg: str*, *header_name: str*, *, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

400 Bad Request.

One of the headers in the request is invalid.

msg and *header_name* are the only positional arguments allowed, the other arguments are defined as keyword-only.

Parameters

- **msg** (*str*) – A description of why the value is invalid.
- **header_name** (*str*) – The name of the invalid header.

Keyword Arguments

- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

> **Note**
>
> The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

> **Note**
>
> Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.

- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: int | None

An internal application code that a user can reference when requesting support for the error.

description: str | None

Description of the error to send to the client.

headers: HeaderArg | None

Extra headers to add to the response.

link: Link | None

An href that the client can provide to the user for getting help.

status: ResponseStatus

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: str

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPMissingHeader**(*header_name: str*, *, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

400 Bad Request.

A header is missing from the request.

header_name is the only positional argument allowed, the other arguments are defined as keyword-only.

Parameters

header_name (*str*) – The name of the missing header.

Keyword Arguments

- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note

  The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  Note

  Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').

- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: int | None

An internal application code that a user can reference when requesting support for the error.

description: str | None

Description of the error to send to the client.

headers: HeaderArg | None

Extra headers to add to the response.

link: Link | None

An href that the client can provide to the user for getting help.

status: ResponseStatus

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: str

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPInvalidParam**(*msg: str*, *param_name: str*, ***, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

400 Bad Request.

A parameter in the request is invalid. This error may refer to a parameter in a query string, form, or document that was submitted with the request.

msg and *param_name* are the only positional arguments allowed, the other arguments are defined as keyword-only.

Parameters

- **msg** (*str*) – A description of the invalid parameter.
- **param_name** (*str*) – The name of the parameter.

Keyword Arguments

- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note

  The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  Note

  Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').

- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: int | None

An internal application code that a user can reference when requesting support for the error.

description: str | None

Description of the error to send to the client.

headers: HeaderArg | None

Extra headers to add to the response.

link: Link | None

An href that the client can provide to the user for getting help.

status: ResponseStatus

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: str

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPMissingParam**(*param_name: str*, ***, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

400 Bad Request.

A parameter is missing from the request. This error may refer to a parameter in a query string, form, or document that was submitted with the request.

param_name is the only positional argument allowed, the other arguments are defined as keyword-only.

Parameters

param_name (*str*) – The name of the missing parameter.

Keyword Arguments

- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note

  The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  Note

  Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: int | None
An internal application code that a user can reference when requesting support for the error.

description: str | None
Description of the error to send to the client.

headers: HeaderArg | None
Extra headers to add to the response.

link: Link | None
An href that the client can provide to the user for getting help.

status: ResponseStatus
HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: str
Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPUnauthorized**(*, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, *challenges: Iterable[str] | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

401 Unauthorized.

The request has not been applied because it lacks valid authentication credentials for the target resource.

The server generating a 401 response MUST send a WWW-Authenticate header field containing at least one challenge applicable to the target resource.

If the request included authentication credentials, then the 401 response indicates that authorization has been refused for those credentials. The user agent MAY repeat the request with a new or replaced Authorization header field. If the 401 response contains the same challenge as the prior response, and the user agent has already attempted authentication at least once, then the user agent SHOULD present the enclosed representation to the user, since it usually contains relevant diagnostic information.

(See also: RFC 7235, Section 3.1)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Error title (default '401 Unauthorized').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

> **Note**
>
> The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

> **Note**
>
> Falcon can process a list of `tuple` slightly faster than a `dict`.

- **challenges** (*iterable of str*) – One or more authentication challenges to use as the value of the WWW-Authenticate header in the response.

  Note

  The existing value of the WWW-Authenticate in headers will be overridden by this value

  (See also: RFC 7235, Section 2.1)

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: **int** | **None**

An internal application code that a user can reference when requesting support for the error.

description: **str** | **None**

Description of the error to send to the client.

headers: **HeaderArg** | **None**

Extra headers to add to the response.

link: **Link** | **None**

An href that the client can provide to the user for getting help.

status: **ResponseStatus**

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: **str**

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPForbidden**(*, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

403 Forbidden.

The server understood the request but refuses to authorize it.

A server that wishes to make public why the request has been forbidden can describe that reason in the response payload (if any).

If authentication credentials were provided in the request, the server considers them insufficient to grant access. The client SHOULD NOT automatically repeat the request with the same credentials. The client MAY repeat the request with new or different credentials. However, a request might be forbidden for reasons unrelated to the credentials.

An origin server that wishes to "hide" the current existence of a forbidden target resource MAY instead respond with a status code of 404 Not Found.

(See also: RFC 7231, Section 6.5.4)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Error title (default '403 Forbidden').

- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

> **Note**
>
> The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

> **Note**
>
> Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: **int** | **None**

An internal application code that a user can reference when requesting support for the error.

description: **str** | **None**

Description of the error to send to the client.

headers: **HeaderArg** | **None**

Extra headers to add to the response.

link: **Link** | **None**

An href that the client can provide to the user for getting help.

status: **ResponseStatus**

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: **str**

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPNotFound**(**, title: str | None = None, description: str | None = None, headers: HeaderArg | None = None, **kwargs: HTTPErrorKeywordArguments*)

404 Not Found.

The origin server did not find a current representation for the target resource or is not willing to disclose that one exists.

A 404 status code does not indicate whether this lack of representation is temporary or permanent; the 410 Gone status code is preferred over 404 if the origin server knows, presumably through some configurable means, that the condition is likely to be permanent.

A 404 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls.

(See also: RFC 7231, Section 6.5.3)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Human-friendly error title. If not provided, and *description* is also not provided, no body will be included in the response.
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two (default `None`).
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note

  The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  Note

  Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: **int** | **None**

An internal application code that a user can reference when requesting support for the error.

description: **str** | **None**

Description of the error to send to the client.

headers: **HeaderArg** | **None**

Extra headers to add to the response.

link: **Link** | **None**

An href that the client can provide to the user for getting help.

status: **ResponseStatus**

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: **str**

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPRouteNotFound**(*, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

404 Not Found.

The request did not match any routes configured for the application.

This subclass of *HTTPNotFound* is raised by the framework to provide a default 404 response when no route matches the request. This behavior can be customized by registering a custom error handler for *HTTPRouteNotFound*.

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Human-friendly error title. If not provided, and *description* is also not provided, no body will be included in the response.
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two (default `None`).
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note

  The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  Note

  Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: **int** | **None**

An internal application code that a user can reference when requesting support for the error.

description: **str** | **None**

Description of the error to send to the client.

headers: **HeaderArg** | **None**

Extra headers to add to the response.

link: **Link** | **None**

An href that the client can provide to the user for getting help.

status: **ResponseStatus**

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: **str**

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPMethodNotAllowed**(*allowed_methods: Iterable[str]*, *, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

405 Method Not Allowed.

The method received in the request-line is known by the origin server but not supported by the target resource.

The origin server MUST generate an Allow header field in a 405 response containing a list of the target resource's currently supported methods.

A 405 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls.

(See also: RFC 7231, Section 6.5.5)

allowed_methods is the only positional argument allowed, the other arguments are defined as keyword-only.

Parameters

allowed_methods (*list of str*) – Allowed HTTP methods for this resource (e.g., `['GET', 'POST', 'HEAD']`).

> **Note**
>
> If previously set, the Allow response header will be overridden by this value.

Keyword Arguments

- **title** (*str*) – Human-friendly error title. If not provided, and *description* is also not provided, no body will be included in the response.
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two (default `None`).
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  > **Note**
  >
  > The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  > **Note**
  >
  > Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: **int | None**

An internal application code that a user can reference when requesting support for the error.

description: str | None

Description of the error to send to the client.

headers: HeaderArg | None

Extra headers to add to the response.

link: Link | None

An href that the client can provide to the user for getting help.

status: ResponseStatus

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: str

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPNotAcceptable**(*, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

406 Not Acceptable.

The target resource does not have a current representation that would be acceptable to the user agent, according to the proactive negotiation header fields received in the request, and the server is unwilling to supply a default representation.

The server SHOULD generate a payload containing a list of available representation characteristics and corresponding resource identifiers from which the user or user agent can choose the one most appropriate. A user agent MAY automatically select the most appropriate choice from that list. However, this specification does not define any standard for such automatic selection, as described in RFC 7231, Section 6.4.1

(See also: RFC 7231, Section 6.5.6)

All the arguments are defined as keyword-only.

Keyword Arguments

- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note

  The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  Note

  Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').

- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: **int** | **None**

An internal application code that a user can reference when requesting support for the error.

description: **str** | **None**

Description of the error to send to the client.

headers: **HeaderArg** | **None**

Extra headers to add to the response.

link: **Link** | **None**

An href that the client can provide to the user for getting help.

status: **ResponseStatus**

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: **str**

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPConflict**(*, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

409 Conflict.

The request could not be completed due to a conflict with the current state of the target resource. This code is used in situations where the user might be able to resolve the conflict and resubmit the request.

The server SHOULD generate a payload that includes enough information for a user to recognize the source of the conflict.

Conflicts are most likely to occur in response to a PUT request. For example, if versioning were being used and the representation being PUT included changes to a resource that conflict with those made by an earlier (third-party) request, the origin server might use a 409 response to indicate that it can't complete the request. In this case, the response representation would likely contain information useful for merging the differences based on the revision history.

(See also: RFC 7231, Section 6.5.8)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Error title (default '409 Conflict').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

> **Note**
>
> The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

> **Note**
>
> Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: **int** | **None**

An internal application code that a user can reference when requesting support for the error.

description: **str** | **None**

Description of the error to send to the client.

headers: **HeaderArg** | **None**

Extra headers to add to the response.

link: **Link** | **None**

An href that the client can provide to the user for getting help.

status: **ResponseStatus**

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: **str**

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPGone**(*, *title: str* | *None* = *None*, *description: str* | *None* = *None*, *headers: HeaderArg* | *None* = *None*, ***kwargs: HTTPErrorKeywordArguments*)

410 Gone.

The target resource is no longer available at the origin server and this condition is likely to be permanent.

If the origin server does not know, or has no facility to determine, whether or not the condition is permanent, the status code 404 Not Found ought to be used instead.

The 410 response is primarily intended to assist the task of web maintenance by notifying the recipient that the resource is intentionally unavailable and that the server owners desire that remote links to that resource be removed. Such an event is common for limited-time, promotional services and for resources belonging to individuals no longer associated with the origin server's site. It is not necessary to mark all permanently unavailable resources as "gone" or to keep the mark for any length of time – that is left to the discretion of the server owner.

A 410 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls.

(See also: RFC 7231, Section 6.5.9)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Human-friendly error title. If not provided, and *description* is also not provided, no body will be included in the response.
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two (default `None`).

- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  > **Note**
  >
  > The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  > **Note**
  >
  > Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: int | None
: An internal application code that a user can reference when requesting support for the error.

description: str | None
: Description of the error to send to the client.

headers: HeaderArg | None
: Extra headers to add to the response.

link: Link | None
: An href that the client can provide to the user for getting help.

status: ResponseStatus
: HTTP status code or line (e.g., `'200 OK'`).

  This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: str
: Error title to send to the client.

  Derived from the `status` if not provided.

class falcon.**HTTPLengthRequired**(*, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

411 Length Required.

The server refuses to accept the request without a defined Content- Length.

The client MAY repeat the request if it adds a valid Content-Length header field containing the length of the message body in the request message.

(See also: RFC 7231, Section 6.5.10)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Error title (default '411 Length Required').

- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

> **Note**
>
> The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

> **Note**
>
> Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: **int** | **None**

An internal application code that a user can reference when requesting support for the error.

description: **str** | **None**

Description of the error to send to the client.

headers: **HeaderArg** | **None**

Extra headers to add to the response.

link: **Link** | **None**

An href that the client can provide to the user for getting help.

status: **ResponseStatus**

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: **str**

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPPreconditionFailed**(*, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

412 Precondition Failed.

One or more conditions given in the request header fields evaluated to false when tested on the server.

This response code allows the client to place preconditions on the current resource state (its current representations and metadata) and, thus, prevent the request method from being applied if the target resource is in an unexpected state.

(See also: RFC 7232, Section 4.2)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Error title (default '412 Precondition Failed').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note

  The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  Note

  Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: **int** | **None**

An internal application code that a user can reference when requesting support for the error.

description: **str** | **None**

Description of the error to send to the client.

headers: **HeaderArg** | **None**

Extra headers to add to the response.

link: **Link** | **None**

An href that the client can provide to the user for getting help.

status: **ResponseStatus**

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: **str**

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPContentTooLarge**(**, title: str | None = None, description: str | None = None, retry_after: RetryAfter = None, headers: HeaderArg | None = None, **kwargs: HTTPErrorKeywordArguments*)

413 Content Too Large.

The server is refusing to process a request because the request payload is larger than the server is willing or able to process.

The server MAY close the connection to prevent the client from continuing the request.

If the condition is temporary, the server SHOULD generate a Retry- After header field to indicate that it is temporary and after what time the client MAY try again.

(See also: RFC 7231, Section 6.5.11)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Error title (default '413 Payload Too Large').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note

  The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  Note

  Falcon can process a list of `tuple` slightly faster than a `dict`.

- **retry_after** (*datetime or int*) – Value for the Retry-After header. If a `datetime` object, will serialize as an HTTP date. Otherwise, a non-negative `int` is expected, representing the number of seconds to wait.

  Note

  The existing value of the Retry-After in headers will be overridden by this value

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

Added in version 4.0.

code: int | None

An internal application code that a user can reference when requesting support for the error.

description: str | None

Description of the error to send to the client.

headers: HeaderArg | None

Extra headers to add to the response.

link: Link | None

An href that the client can provide to the user for getting help.

status: ResponseStatus

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: str

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPUriTooLong**(*, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

414 URI Too Long.

The server is refusing to service the request because the request- target is longer than the server is willing to interpret.

This rare condition is only likely to occur when a client has improperly converted a POST request to a GET request with long query information, when the client has descended into a "black hole" of redirection (e.g., a redirected URI prefix that points to a suffix of itself) or when the server is under attack by a client attempting to exploit potential security holes.

A 414 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls.

(See also: RFC 7231, Section 6.5.12)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Error title (default '414 URI Too Long').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two (default `None`).
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note

  The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  Note

  Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: int | None

An internal application code that a user can reference when requesting support for the error.

description: str | None

Description of the error to send to the client.

headers: HeaderArg | None

Extra headers to add to the response.

link: Link | None

An href that the client can provide to the user for getting help.

status: ResponseStatus

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: str

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPUnsupportedMediaType**(*, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

415 Unsupported Media Type.

The origin server is refusing to service the request because the payload is in a format not supported by this method on the target resource.

The format problem might be due to the request's indicated Content- Type or Content-Encoding, or as a result of inspecting the data directly.

(See also: RFC 7231, Section 6.5.13)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Error title (default '415 Unsupported Media Type').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

> **Note**
>
> The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

> **Note**
>
> Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').

- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: **int** | **None**

An internal application code that a user can reference when requesting support for the error.

description: **str** | **None**

Description of the error to send to the client.

headers: **HeaderArg** | **None**

Extra headers to add to the response.

link: **Link** | **None**

An href that the client can provide to the user for getting help.

status: **ResponseStatus**

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: **str**

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPRangeNotSatisfiable**(*resource_length: int*, *, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

416 Range Not Satisfiable.

None of the ranges in the request's Range header field overlap the current extent of the selected resource or that the set of ranges requested has been rejected due to invalid ranges or an excessive request of small or overlapping ranges.

For byte ranges, failing to overlap the current extent means that the first-byte-pos of all of the byte-range-spec values were greater than the current length of the selected representation. When this status code is generated in response to a byte-range request, the sender SHOULD generate a Content-Range header field specifying the current length of the selected representation.

(See also: RFC 7233, Section 4.4)

resource_length is the only positional argument allowed, the other arguments are defined as keyword-only.

Parameters

resource_length – The maximum value for the last-byte-pos of a range request. Used to set the Content-Range header.

> **Note**
>
> The existing value of the Content-Range in headers will be overridden by this value

Keyword Arguments

- **title** (*str*) – Error title (default '416 Range Not Satisfiable').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

> **Note**
>
> The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

> **Note**
>
> Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: **int** | **None**

An internal application code that a user can reference when requesting support for the error.

description: **str** | **None**

Description of the error to send to the client.

headers: **HeaderArg** | **None**

Extra headers to add to the response.

link: **Link** | **None**

An href that the client can provide to the user for getting help.

status: **ResponseStatus**

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: **str**

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPUnprocessableEntity**(**, title: str | None = None, description: str | None = None, headers: HeaderArg | None = None, **kwargs: HTTPErrorKeywordArguments*)

422 Unprocessable Entity.

The server understands the content type of the request entity (hence a 415 Unsupported Media Type status code is inappropriate), and the syntax of the request entity is correct (thus a 400 Bad Request status code is inappropriate) but was unable to process the contained instructions.

For example, this error condition may occur if an XML request body contains well-formed (i.e., syntactically correct), but semantically erroneous, XML instructions.

(See also: RFC 4918, Section 11.2)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Error title (default '422 Unprocessable Entity').

- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

> **Note**
>
> The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

> **Note**
>
> Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: **int** | **None**

An internal application code that a user can reference when requesting support for the error.

description: **str** | **None**

Description of the error to send to the client.

headers: **HeaderArg** | **None**

Extra headers to add to the response.

link: **Link** | **None**

An href that the client can provide to the user for getting help.

status: **ResponseStatus**

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: **str**

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPLocked**(**, title: str | None = None, description: str | None = None, headers: HeaderArg | None = None, **kwargs: HTTPErrorKeywordArguments*)

423 Locked.

The 423 (Locked) status code means the source or destination resource of a method is locked. This response SHOULD contain an appropriate precondition or postcondition code, such as 'lock-token-submitted' or 'no-conflicting-lock'.

(See also: RFC 4918, Section 11.3)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Error title (default '423 Locked').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note

  The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  Note

  Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: int | None

An internal application code that a user can reference when requesting support for the error.

description: str | None

Description of the error to send to the client.

headers: HeaderArg | None

Extra headers to add to the response.

link: Link | None

An href that the client can provide to the user for getting help.

status: ResponseStatus

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: str

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPFailedDependency**(*, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

424 Failed Dependency.

The 424 (Failed Dependency) status code means that the method could not be performed on the resource because the requested action depended on another action and that action failed.

(See also: RFC 4918, Section 11.4)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Error title (default '424 Failed Dependency').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note

  The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  Note

  Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: int | None

An internal application code that a user can reference when requesting support for the error.

description: str | None

Description of the error to send to the client.

headers: HeaderArg | None

Extra headers to add to the response.

link: Link | None

An href that the client can provide to the user for getting help.

status: ResponseStatus

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: str

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPPreconditionRequired**(*, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

428 Precondition Required.

The 428 status code indicates that the origin server requires the request to be conditional.

Its typical use is to avoid the "lost update" problem, where a client GETs a resource's state, modifies it, and PUTs it back to the server, when meanwhile a third party has modified the state on the server, leading to a

conflict. By requiring requests to be conditional, the server can assure that clients are working with the correct copies.

Responses using this status code SHOULD explain how to resubmit the request successfully.

(See also: RFC 6585, Section 3)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Error title (default '428 Precondition Required').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note

  The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  Note

  Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: **int** | **None**

An internal application code that a user can reference when requesting support for the error.

description: **str** | **None**

Description of the error to send to the client.

headers: **HeaderArg** | **None**

Extra headers to add to the response.

link: **Link** | **None**

An href that the client can provide to the user for getting help.

status: **ResponseStatus**

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: **str**

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPTooManyRequests**(*, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, *retry_after: RetryAfter = None*, ***kwargs: HTTPErrorKeywordArguments*)

429 Too Many Requests.

The user has sent too many requests in a given amount of time ("rate limiting").

The response representations SHOULD include details explaining the condition, and MAY include a Retry-After header indicating how long to wait before making a new request.

Responses with the 429 status code MUST NOT be stored by a cache.

(See also: RFC 6585, Section 4)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Error title (default '429 Too Many Requests').
- **description** (*str*) – Human-friendly description of the rate limit that was exceeded.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note

  The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  Note

  Falcon can process a list of `tuple` slightly faster than a `dict`.

- **retry_after** (*datetime or int*) – Value for the Retry-After header. If a `datetime` object, will serialize as an HTTP date. Otherwise, a non-negative `int` is expected, representing the number of seconds to wait.

  Note

  The existing value of the Retry-After in headers will be overridden by this value

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: **int | None**

An internal application code that a user can reference when requesting support for the error.

description: **str | None**

Description of the error to send to the client.

headers: HeaderArg | None

Extra headers to add to the response.

link: Link | None

An href that the client can provide to the user for getting help.

status: ResponseStatus

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: str

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPRequestHeaderFieldsTooLarge**(*, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

431 Request Header Fields Too Large.

The 431 status code indicates that the server is unwilling to process the request because its header fields are too large. The request MAY be resubmitted after reducing the size of the request header fields.

It can be used both when the set of request header fields in total is too large, and when a single header field is at fault. In the latter case, the response representation SHOULD specify which header field was too large.

Responses with the 431 status code MUST NOT be stored by a cache.

(See also: RFC 6585, Section 5)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Error title (default '431 Request Header Fields Too Large').
- **description** (*str*) – Human-friendly description of the rate limit that was exceeded.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note

  The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  Note

  Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: int | None

An internal application code that a user can reference when requesting support for the error.

description: str | None

Description of the error to send to the client.

headers: HeaderArg | None

Extra headers to add to the response.

link: Link | None

An href that the client can provide to the user for getting help.

status: ResponseStatus

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: str

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPUnavailableForLegalReasons**(*, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

451 Unavailable For Legal Reasons.

The server is denying access to the resource as a consequence of a legal demand.

The server in question might not be an origin server. This type of legal demand typically most directly affects the operations of ISPs and search engines.

Responses using this status code SHOULD include an explanation, in the response body, of the details of the legal demand: the party making it, the applicable legislation or regulation, and what classes of person and resource it applies to.

Note that in many cases clients can still access the denied resource by using technical countermeasures such as a VPN or the Tor network.

A 451 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls.

(See also: RFC 7725, Section 3)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Error title (default '451 Unavailable For Legal Reasons').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two (default `None`).
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

> **Note**
>
> The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

> **Note**
>
> Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: **int** | **None**

An internal application code that a user can reference when requesting support for the error.

description: **str** | **None**

Description of the error to send to the client.

headers: **HeaderArg** | **None**

Extra headers to add to the response.

link: **Link** | **None**

An href that the client can provide to the user for getting help.

status: **ResponseStatus**

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: **str**

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPInternalServerError**(*, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

500 Internal Server Error.

The server encountered an unexpected condition that prevented it from fulfilling the request.

(See also: RFC 7231, Section 6.6.1)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Error title (default '500 Internal Server Error').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

> **Note**
>
> The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

> **Note**
>
> Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: **int** | **None**

An internal application code that a user can reference when requesting support for the error.

description: **str** | **None**

Description of the error to send to the client.

headers: **HeaderArg** | **None**

Extra headers to add to the response.

link: **Link** | **None**

An href that the client can provide to the user for getting help.

status: **ResponseStatus**

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: **str**

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPNotImplemented**(*, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

501 Not Implemented.

The 501 (Not Implemented) status code indicates that the server does not support the functionality required to fulfill the request. This is the appropriate response when the server does not recognize the request method and is not capable of supporting it for any resource.

A 501 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls as described in RFC 7234, Section 4.2.2.

(See also: RFC 7231, Section 6.6.2)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Error title (default '500 Internal Server Error').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

> **Note**
>
> The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

> **Note**
>
> Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: **int** | **None**

An internal application code that a user can reference when requesting support for the error.

description: **str** | **None**

Description of the error to send to the client.

headers: **HeaderArg** | **None**

Extra headers to add to the response.

link: **Link** | **None**

An href that the client can provide to the user for getting help.

status: **ResponseStatus**

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: **str**

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPBadGateway**(*, *title: str* | *None* = *None*, *description: str* | *None* = *None*, *headers: HeaderArg* | *None* = *None*, ***kwargs: HTTPErrorKeywordArguments*)

502 Bad Gateway.

The server, while acting as a gateway or proxy, received an invalid response from an inbound server it accessed while attempting to fulfill the request.

(See also: RFC 7231, Section 6.6.3)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Error title (default '502 Bad Gateway').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

> **Note**
>
> The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

> **Note**
>
> Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: **int** | **None**

An internal application code that a user can reference when requesting support for the error.

description: **str** | **None**

Description of the error to send to the client.

headers: **HeaderArg** | **None**

Extra headers to add to the response.

link: **Link** | **None**

An href that the client can provide to the user for getting help.

status: **ResponseStatus**

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: **str**

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPServiceUnavailable**(*, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, *retry_after: RetryAfter = None*, ***kwargs: HTTPErrorKeywordArguments*)

503 Service Unavailable.

The server is currently unable to handle the request due to a temporary overload or scheduled maintenance, which will likely be alleviated after some delay.

The server MAY send a Retry-After header field to suggest an appropriate amount of time for the client to wait before retrying the request.

Note: The existence of the 503 status code does not imply that a server has to use it when becoming overloaded. Some servers might simply refuse the connection.

(See also: RFC 7231, Section 6.6.4)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Error title (default '503 Service Unavailable').

- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note

  The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  Note

  Falcon can process a list of `tuple` slightly faster than a `dict`.

- **retry_after** (*datetime or int*) – Value for the Retry-After header. If a `datetime` object, will serialize as an HTTP date. Otherwise, a non-negative `int` is expected, representing the number of seconds to wait.

  Note

  The existing value of the Retry-After in headers will be overridden by this value

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: **int** | **None**

An internal application code that a user can reference when requesting support for the error.

description: **str** | **None**

Description of the error to send to the client.

headers: **HeaderArg** | **None**

Extra headers to add to the response.

link: **Link** | **None**

An href that the client can provide to the user for getting help.

status: **ResponseStatus**

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: **str**

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPGatewayTimeout**(*, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

504 Gateway Timeout.

The 504 (Gateway Timeout) status code indicates that the server, while acting as a gateway or proxy, did not receive a timely response from an upstream server it needed to access in order to complete the request.

(See also: RFC 7231, Section 6.6.5)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Error title (default '503 Service Unavailable').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note

  The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  Note

  Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: **int** | **None**

An internal application code that a user can reference when requesting support for the error.

description: **str** | **None**

Description of the error to send to the client.

headers: **HeaderArg** | **None**

Extra headers to add to the response.

link: **Link** | **None**

An href that the client can provide to the user for getting help.

status: **ResponseStatus**

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: **str**

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPVersionNotSupported**(*, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

505 HTTP Version Not Supported.

The 505 (HTTP Version Not Supported) status code indicates that the server does not support, or refuses to support, the major version of HTTP that was used in the request message. The server is indicating that it is unable or unwilling to complete the request using the same major version as the client (as described in RFC 7230, Section 2.6), other than with this error message. The server SHOULD generate a representation for the 505 response that describes why that version is not supported and what other protocols are supported by that server.

(See also: RFC 7231, Section 6.6.6)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Error title (default '503 Service Unavailable').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note

  The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  Note

  Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: **int | None**

An internal application code that a user can reference when requesting support for the error.

description: **str | None**

Description of the error to send to the client.

headers: **HeaderArg | None**

Extra headers to add to the response.

link: **Link | None**

An href that the client can provide to the user for getting help.

status: **ResponseStatus**

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: str

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPInsufficientStorage**(*, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

507 Insufficient Storage.

The 507 (Insufficient Storage) status code means the method could not be performed on the resource because the server is unable to store the representation needed to successfully complete the request. This condition is considered to be temporary. If the request that received this status code was the result of a user action, the request MUST NOT be repeated until it is requested by a separate user action.

(See also: RFC 4918, Section 11.5)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Error title (default '507 Insufficient Storage').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note

  The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  Note

  Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: int | None

An internal application code that a user can reference when requesting support for the error.

description: str | None

Description of the error to send to the client.

headers: HeaderArg | None

Extra headers to add to the response.

link: Link | None

An href that the client can provide to the user for getting help.

status: ResponseStatus

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: str

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPLoopDetected**(*, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

508 Loop Detected.

The 508 (Loop Detected) status code indicates that the server terminated an operation because it encountered an infinite loop while processing a request with "Depth: infinity". This status indicates that the entire operation failed.

(See also: RFC 5842, Section 7.2)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Error title (default '508 Loop Detected').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note

  The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  Note

  Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: int | None

An internal application code that a user can reference when requesting support for the error.

description: str | None

Description of the error to send to the client.

headers: HeaderArg | None

Extra headers to add to the response.

link: Link | None

An href that the client can provide to the user for getting help.

status: ResponseStatus

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: str

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**HTTPNetworkAuthenticationRequired**(*, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

511 Network Authentication Required.

The 511 status code indicates that the client needs to authenticate to gain network access.

The response representation SHOULD contain a link to a resource that allows the user to submit credentials.

Note that the 511 response SHOULD NOT contain a challenge or the authentication interface itself, because clients would show the interface as being associated with the originally requested URL, which may cause confusion.

The 511 status SHOULD NOT be generated by origin servers; it is intended for use by intercepting proxies that are interposed as a means of controlling access to the network.

Responses with the 511 status code MUST NOT be stored by a cache.

(See also: RFC 6585, Section 6)

All the arguments are defined as keyword-only.

Keyword Arguments

- **title** (*str*) – Error title (default '511 Network Authentication Required').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note

  The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  Note

  Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.

- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: int | None

An internal application code that a user can reference when requesting support for the error.

description: str | None

Description of the error to send to the client.

headers: HeaderArg | None

Extra headers to add to the response.

link: Link | None

An href that the client can provide to the user for getting help.

status: ResponseStatus

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: str

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**MediaNotFoundError**(*media_type: str*, ***kwargs: str | int | None*)

400 Bad Request.

Exception raised by a media handler when trying to parse an empty body.

> **Note**
>
> Some media handlers, like the one for URL-encoded forms, allow an empty body. In these cases this exception will not be raised.

Parameters

media_type (*str*) – The media type that was expected.

Keyword Arguments

- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  > **Note**
  >
  > The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  > **Note**
  >
  > Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.

- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: int | None

An internal application code that a user can reference when requesting support for the error.

description: str | None

Description of the error to send to the client.

headers: HeaderArg | None

Extra headers to add to the response.

link: Link | None

An href that the client can provide to the user for getting help.

status: ResponseStatus

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: str

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**MediaMalformedError**(*media_type: str*, ***kwargs: HeaderArg | HTTPErrorKeywordArguments*)

400 Bad Request.

Exception raised by a media handler when trying to parse a malformed body. The cause of this exception, if any, is stored in the `__cause__` attribute using the "raise … from" form when raising.

Parameters

media_type (*str*) – The media type that was expected.

Keyword Arguments

- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note

  The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  Note

  Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').

- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: **int** | **None**

An internal application code that a user can reference when requesting support for the error.

description: **str** | **None**

Description of the error to send to the client.

headers: **HeaderArg** | **None**

Extra headers to add to the response.

link: **Link** | **None**

An href that the client can provide to the user for getting help.

status: **ResponseStatus**

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: **str**

Error title to send to the client.

Derived from the `status` if not provided.

class falcon.**MediaValidationError**(*, *title: str | None = None*, *description: str | None = None*, *headers: HeaderArg | None = None*, ***kwargs: HTTPErrorKeywordArguments*)

400 Bad Request.

Request media is invalid. This exception is raised by a media validator (such as *jsonschema.validate*) when `req.media` is successfully deserialized, but fails to validate against the configured schema.

The cause of this exception, if any, is stored in the `__cause__` attribute using the "raise … from" form when raising.

Note

All the arguments must be passed as keyword only.

Keyword Arguments

- **title** (*str*) – Error title (default '400 Bad Request').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

Note

The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

> **Note**
>
> Falcon can process a list of `tuple` slightly faster than a `dict`.

- **href** (*str*) – A URL someone can visit to find out more information (default `None`). Unicode characters are percent-encoded.
- **href_text** (*str*) – If href is given, use this as the friendly title/description for the link (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

code: int | None

An internal application code that a user can reference when requesting support for the error.

description: str | None

Description of the error to send to the client.

headers: HeaderArg | None

Extra headers to add to the response.

link: Link | None

An href that the client can provide to the user for getting help.

status: ResponseStatus

HTTP status code or line (e.g., `'200 OK'`).

This may be set to a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int`.

title: str

Error title to send to the client.

Derived from the `status` if not provided.

5.3.7 Media

Falcon allows for easy and customizable internet media type handling. By default Falcon only enables handlers for JSON and HTML (URL-encoded and multipart) forms. However, additional handlers can be configured through the *falcon.RequestOptions* and *falcon.ResponseOptions* objects specified on your *falcon.App*.

> **Note**
>
> WebSocket media is handled differently from regular HTTP requests. For information regarding WebSocket media handlers, please see: *Media Handlers* in the WebSocket section.

Usage

Zero configuration is needed if you're creating a JSON API. Simply use *get_media()* and *media* (WSGI) , or *get_media()* and *media* (ASGI) to let Falcon do the heavy lifting for you.

WSGI

```
import falcon


class EchoResource:
```

(continues on next page)

(continued from previous page)

```
    def on_post(self, req, resp):
        # Deserialize the request body based on the Content-Type
        #   header in the request, or the default media type
        #   when the Content-Type header is generic ('*/*') or
        #   missing.
        obj = req.get_media()

        message = obj.get('message')

        # The framework will look for a media handler that matches
        #   the response's Content-Type header, or fall back to the
        #   default media type (typically JSON) when the app does
        #   not explicitly set the Content-Type header.
        resp.media = {'message': message}
        resp.status = falcon.HTTP_200
```

ASGI

```
import falcon


class EchoResource:
    async def on_post(self, req, resp):
        # Deserialize the request body. Note that the ASGI version
        #   of this method must be awaited.
        obj = await req.get_media()

        message = obj.get('message')

        # The framework will look for a media handler that matches
        #   the response's Content-Type header, or fall back to the
        #   default media type (typically JSON) when the app does
        #   not explicitly set the Content-Type header.
        resp.media = {'message': message}
        resp.status = falcon.HTTP_200
```

Warning

Once *falcon.Request.get_media()* or *falcon.asgi.Request.get_media()* is called on a request, it will consume the request's body stream. To avoid unnecessary overhead, Falcon will only process request media the first time it is referenced. Subsequent interactions will use a cached object.

Validating Media

Falcon currently only provides a JSON Schema media validator; however, JSON Schema is very versatile and can be used to validate any deserialized media type that JSON also supports (i.e. dicts, lists, etc).

falcon.media.validators.jsonschema.**validate**(*req_schema: dict[str, Any] | None = None*, *resp_schema: dict[str, Any] | None = None*) → Callable[[Callable[[...], Any]], Callable[[...], Any]]

Validate `req.media` using JSON Schema.

This decorator provides standard JSON Schema validation via the `jsonschema` package available from PyPI. Semantic validation via the *format* keyword is enabled for the default checkers implemented by `jsonschema.FormatChecker`.

In the case of failed request media validation, an instance of *MediaValidationError* is raised by the decorator. By default, this error is rendered as a 400 (*HTTPBadRequest*) response with the `title` and `description` attributes explaining the validation failure, but this behavior can be modified by adding a custom error *handler* for *MediaValidationError*.

Note

The `jsonschema` package must be installed separately in order to use this decorator, as Falcon does not install it by default.

See json-schema.org for more information on defining a compatible dictionary.

Keyword Arguments

- **req_schema** (*dict*) – A dictionary that follows the JSON Schema specification. The request will be validated against this schema.
- **resp_schema** (*dict*) – A dictionary that follows the JSON Schema specification. The response will be validated against this schema.

Example

WSGI

```
from falcon.media.validators import jsonschema

# -- snip --

@jsonschema.validate(my_post_schema)
def on_post(self, req, resp):

# -- snip --
```

ASGI

```
from falcon.media.validators import jsonschema

# -- snip --

@jsonschema.validate(my_post_schema)
async def on_post(self, req, resp):

# -- snip --
```

If JSON Schema does not meet your needs, a custom validator may be implemented in a similar manner to the one above.

Content-Type Negotiation

Falcon currently only supports partial negotiation out of the box. By default, when the `get_media()` method or the `media` attribute is used, the framework attempts to (de)serialize based on the `Content-Type` header value. The missing link that Falcon doesn't provide is the connection between the `Accept` header provided by a user and the `Content-Type` header set on the response.

If you do need full negotiation, it is very easy to bridge the gap using middleware. Here is an example of how this can be done:

WSGI

```
from falcon import Request, Response

class NegotiationMiddleware:
    def process_request(self, req: Request, resp: Response) -> None:
        resp.content_type = req.accept
```

ASGI

```
from falcon.asgi import Request, Response

class NegotiationMiddleware:
    async def process_request(self, req: Request, resp: Response) -> None:
        resp.content_type = req.accept
```

Exception Handling

Version 3 of Falcon updated how the handling of exceptions raised by handlers behaves:

- Falcon lets the media handler try to deserialized an empty body. For the media types that don't allow empty bodies as a valid value, such as `JSON`, an instance of *falcon.MediaNotFoundError* should be raised. By default, this error will be rendered as a `400 Bad Request` response to the client. This exception may be suppressed by passing a value to the `default_when_empty` argument when calling `Request.get_media()`. In this case, this value will be returned by the call.
- If a handler encounters an error while parsing a non-empty body, an instance of *falcon.MediaMalformedError* should be raised. The original exception, if any, is stored in the `__cause__` attribute of the raised instance. By default, this error will be rendered as a `400 Bad Request` response to the client.

If any exception was raised by the handler while parsing the body, all subsequent invocations of `Request.get_media()` or `Request.media` will result in a re-raise of the same exception, unless the exception was a *falcon.MediaNotFoundError* and a default value is passed to the `default_when_empty` attribute of the current invocation.

External handlers should update their logic to align to the internal Falcon handlers.

Replacing the Default Handlers

By default, the framework installs *falcon.media.JSONHandler*, *falcon.media.URLEncodedFormHandler*, and *falcon.media.MultipartFormHandler* for the `application/json`, `application/x-www-form-urlencoded`, and `multipart/form-data` media types, respectively.

When creating your App object you can either add or completely replace all of the handlers. For example, let's say you want to write an API that sends and receives MessagePack. We can easily do this by telling our Falcon API that we want a default media type of `application/msgpack`, and then creating a new *Handlers* object to map that media type to an appropriate handler.

The following example demonstrates how to replace the default handlers. Because Falcon provides a *MessagePackHandler* that is not enabled by default, we use it in our examples below. However, you can always substitute a *custom media handler* as needed.

```
import falcon
from falcon import media


handlers = media.Handlers({
    falcon.MEDIA_MSGPACK: media.MessagePackHandler(),
})
```

(continues on next page)

(continued from previous page)

```
app = falcon.App(media_type=falcon.MEDIA_MSGPACK)

app.req_options.media_handlers = handlers
app.resp_options.media_handlers = handlers
```

Alternatively, you can simply update the existing *Handlers* object to retain the default handlers:

```
import falcon
from falcon import media


extra_handlers = {
    falcon.MEDIA_MSGPACK: media.MessagePackHandler(),
}

app = falcon.App()

app.req_options.media_handlers.update(extra_handlers)
app.resp_options.media_handlers.update(extra_handlers)
```

The `falcon` module provides a number of constants for common media types. See also: *Media Type Constants*.

Note

The configured *falcon.Response* JSON handler is also used to serialize *falcon.HTTPError* and the `json` attribute of *falcon.asgi.SSEvent*. The JSON handler configured in *falcon.Request* is used by *falcon.Request.get_param_as_json()* to deserialize query params.

Therefore, when implementing a custom handler for the JSON media type, it is required that the sync interface methods, meaning *falcon.media.BaseHandler.serialize()* and *falcon.media.BaseHandler.deserialize()*, are implemented even in `ASGI` applications. The default JSON handler, *falcon.media.JSONHandler*, already implements the methods required to work with both types of applications.

Supported Handler Types

class falcon.media.**JSONHandler**(*dumps: Callable[[Any], str | bytes] | None = None*, *loads: Callable[[str], Any] | None = None*)

JSON media handler.

This handler uses Python's standard `json` library by default, but can be easily configured to use any of a number of third-party JSON libraries, depending on your needs. For example, you can often realize a significant performance boost under CPython by using an alternative library. Good options in this respect include *msgspec*, *orjson*, *python-rapidjson*, and *mujson*.

This handler will raise a *falcon.MediaNotFoundError* when attempting to parse an empty body, or a *falcon.MediaMalformedError* if an error happens while parsing the body.

Note

If you are deploying to PyPy, we recommend sticking with the standard library's JSON implementation, since it will be faster in most cases as compared to a third-party library.

Custom JSON library

You can replace the default JSON handler by using a custom JSON library (see also: *Replacing the Default Handlers*). Overriding the default JSON implementation is simply a matter of specifying the desired `dumps` and `loads` functions:

```
import falcon
from falcon import media

import rapidjson

json_handler = media.JSONHandler(
    dumps=rapidjson.dumps,
    loads=rapidjson.loads,
)
extra_handlers = {
    'application/json': json_handler,
}

app = falcon.App()
app.req_options.media_handlers.update(extra_handlers)
app.resp_options.media_handlers.update(extra_handlers)
```

Custom serialization parameters

Even if you decide to stick with the stdlib's `json.dumps` and `json.loads`, you can wrap them using `functools.partial` to provide custom serialization or deserialization parameters supported by the `dumps` and `loads` functions, respectively (see also: *Prettifying JSON Responses*):

```
import falcon
from falcon import media

from functools import partial

json_handler = media.JSONHandler(
    dumps=partial(
        json.dumps,
        default=str,
        sort_keys=True,
    ),
)
extra_handlers = {
    'application/json': json_handler,
}

app = falcon.App()
app.req_options.media_handlers.update(extra_handlers)
app.resp_options.media_handlers.update(extra_handlers)
```

By default, `ensure_ascii` is passed to the `json.dumps` function. If you override the `dumps` function, you will need to explicitly set `ensure_ascii` to `False` in order to enable the serialization of Unicode characters to UTF-8. This is easily done by using `functools.partial` to apply the desired keyword argument. As also demonstrated in the previous paragraph, you can use this same technique to customize any option supported by the `dumps` and `loads` functions:

```
from functools import partial

from falcon import media
```

(continues on next page)

(continued from previous page)

```
import rapidjson

json_handler = media.JSONHandler(
    dumps=partial(
        rapidjson.dumps,
        ensure_ascii=False, sort_keys=True
    ),
)
```

Custom JSON encoder

You can also override the default `JSONEncoder` by using a custom Encoder and updating the media handlers for `application/json` type to use that:

```
import json
from datetime import datetime
from functools import partial

import falcon
from falcon import media

class DatetimeEncoder(json.JSONEncoder):
    """Json Encoder that supports datetime objects."""

    def default(self, obj):
        if isinstance(obj, datetime):
            return obj.isoformat()
        return super().default(obj)

app = falcon.App()

json_handler = media.JSONHandler(
    dumps=partial(json.dumps, cls=DatetimeEncoder),
)
extra_handlers = {
    'application/json': json_handler,
}

app.req_options.media_handlers.update(extra_handlers)
app.resp_options.media_handlers.update(extra_handlers)
```

Note

When testing an application employing a custom JSON encoder, bear in mind that *TestClient* is decoupled from the app, and it simulates requests as if they were performed by a third-party client (just sans network). Therefore, passing the **json** parameter to *simulate_** methods will effectively use the stdlib's `json.dumps()`. If you want to serialize custom objects for testing, you will need to dump them into a string yourself, and pass it using the **body** parameter instead (accompanied by the `application/json` content type header).

Keyword Arguments

- **dumps** (*func*) – Function to use when serializing JSON responses.
- **loads** (*func*) – Function to use when deserializing JSON requests.

class falcon.media.**MessagePackHandler**

Handler built using the `msgpack` module.

This handler uses `msgpack.unpackb()` and `msgpack.Packer().pack()`. The MessagePack `bin` type is used to distinguish between Unicode strings (of type `str`) and byte strings (of type `bytes`).

This handler will raise a *falcon.MediaNotFoundError* when attempting to parse an empty body; it will raise a *falcon.MediaMalformedError* if an error happens while parsing the body.

> **Note**
>
> This handler requires the extra `msgpack` package (version 0.5.2 or higher), which must be installed in addition to `falcon` from PyPI:
>
> ```
> $ pip install msgpack
> ```

class falcon.media.**MultipartFormHandler**(*parse_options:* MultipartParseOptions | *None* = *None*)

Multipart form (content type `multipart/form-data`) media handler.

The `multipart/form-data` media type for HTML5 forms is defined in RFC 7578.

The multipart media type itself is defined in RFC 2046 section 5.1.

> **Note**
>
> Unlike many form parsing implementations in other frameworks, this handler does not consume the stream immediately. Rather, the stream is consumed on-demand and parsed into individual body parts while iterating over the media object.

For examples on parsing the request form, see also: *Multipart Forms*.

class falcon.media.**URLEncodedFormHandler**(*keep_blank: bool* = *True*, *csv: bool* = *False*)

URL-encoded form data handler.

This handler parses `application/x-www-form-urlencoded` HTML forms to a `dict`, similar to how URL query parameters are parsed. An empty body will be parsed as an empty dict.

When deserializing, this handler will raise *falcon.MediaMalformedError* if the request payload cannot be parsed as ASCII or if any of the URL-encoded strings in the payload are not valid UTF-8.

As documented for `urllib.parse.urlencode`, when serializing, the media object must either be a `dict` or a sequence of two-element `tuple`'s. If any values in the media object are sequences, each sequence element is converted to a separate parameter.

Keyword Arguments

- **keep_blank** (*bool*) – Whether to keep empty-string values from the form when deserializing.
- **csv** (*bool*) – Whether to split comma-separated form values into list when deserializing.

Custom Handler Type

If Falcon doesn't have an Internet media type handler that supports your use case, you can easily implement your own using the abstract base class provided by Falcon and documented below.

In general `WSGI` applications only use the sync methods, while `ASGI` applications only use the async one. The JSON handled is an exception to this, since it's used also by other parts of the framework, not only in the media handling. See the *note above* for more details.

class falcon.media.**BaseHandler**

Abstract Base Class for an internet media type handler.

serialize(*media: object*, *content_type: str*) → bytes

Serialize the media object on a *falcon.Response*.

By default, this method raises an instance of NotImplementedError. Therefore, it must be overridden in order to work with WSGI apps. Child classes can ignore this method if they are only to be used with ASGI apps, as long as they override *serialize_async()*.

> **Note**
>
> The JSON media handler is an exception in requiring the implementation of the sync version also for ASGI apps. See the *this section* for more details.

Parameters

- **media** (*object*) – A serializable object.
- **content_type** (*str*) – Type of response content.

Returns

The resulting serialized bytes from the input object.

Return type

bytes

async serialize_async(*media: object*, *content_type: str*) → bytes

Serialize the media object on a *falcon.Response*.

This method is similar to *serialize()* except that it is asynchronous. The default implementation simply calls *serialize()*. If the media object may be awaitable, or is otherwise something that should be read asynchronously, subclasses must override the default implementation in order to handle that case.

> **Note**
>
> By default, the *serialize()* method raises an instance of NotImplementedError. Therefore, child classes must either override *serialize()* or *serialize_async()* in order to be compatible with ASGI apps.

Parameters

- **media** (*object*) – A serializable object.
- **content_type** (*str*) – Type of response content.

Returns

The resulting serialized bytes from the input object.

Return type

bytes

deserialize(*stream: ReadableIO*, *content_type: str | None*, *content_length: int | None*) → object

Deserialize the *falcon.Request* body.

By default, this method raises an instance of NotImplementedError. Therefore, it must be overridden in order to work with WSGI apps. Child classes can ignore this method if they are only to be used with ASGI apps, as long as they override *deserialize_async()*.

> **Note**
>
> The JSON media handler is an exception in requiring the implementation of the sync version also for ASGI apps. See the *this section* for more details.

Parameters

- **stream** (*object*) – Readable file-like object to deserialize.
- **content_type** (*str*) – Type of request content.
- **content_length** (*int*) – Length of request content.

Returns
A deserialized object.

Return type
object

async deserialize_async(*stream: AsyncReadableIO*, *content_type: str | None*, *content_length: int | None*) → object

Deserialize the *falcon.Request* body.

This method is similar to *deserialize()* except that it is asynchronous. The default implementation adapts the synchronous *deserialize()* method via `io.BytesIO`. For improved performance, media handlers should override this method.

> **Note**
>
> By default, the *deserialize()* method raises an instance of `NotImplementedError`. Therefore, child classes must either override *deserialize()* or *deserialize_async()* in order to be compatible with ASGI apps.

Parameters

- **stream** (*object*) – Asynchronous file-like object to deserialize.
- **content_type** (*str*) – Type of request content.
- **content_length** (*int*) – Length of request content, or `None` if the Content-Length header is missing.

Returns
A deserialized object.

Return type
object

exhaust_stream = False

Whether to exhaust the input stream upon finishing deserialization.

Exhausting the stream may be useful for handlers that do not necessarily consume the whole stream, but the deserialized media object is complete and does not involve further streaming.

> **Tip**
>
> In order to use your custom media handler in a *Falcon app*, you'll have to add an instance of your class to the app's media handlers (specified in *RequestOptions* and *ResponseOptions*, respectively).
>
> See also: *Replacing the Default Handlers*.

Handlers Mapping

class falcon.media.**Handlers**(*initial: Mapping[str, BaseHandler] | None = None*)

A dict-like object that manages Internet media type handlers.

copy() → *Handlers*

Create a shallow copy of this instance of handlers.

The resulting copy contains the same keys and values, but it can be customized separately without affecting the original object.

Returns
A shallow copy of handlers.

Added in version 4.0.

Media Type Constants

The falcon module provides a number of constants for common media type strings, including the following:

```
falcon.MEDIA_JSON
falcon.MEDIA_MSGPACK
falcon.MEDIA_MULTIPART
falcon.MEDIA_URLENCODED
falcon.MEDIA_CSV
falcon.MEDIA_PARQUET
falcon.MEDIA_YAML
falcon.MEDIA_XML
falcon.MEDIA_HTML
falcon.MEDIA_JS
falcon.MEDIA_TEXT
falcon.MEDIA_JPEG
falcon.MEDIA_PNG
falcon.MEDIA_GIF
```

5.3.8 Multipart Forms

Falcon features easy and efficient access to submitted multipart forms by using *MultipartFormHandler* to handle the multipart/form-data *media* type. This handler is enabled by default, allowing you to use req.get_media() to iterate over the *body parts* in a form:

WSGI

```
form = req.get_media()
for part in form:
    if part.content_type == 'application/json':
        # Body part is a JSON document, do something useful with it
        resp.media = part.get_media()
    elif part.name == 'datafile':
        while True:
            # Do something with the uploaded data (file)
            chunk = part.stream.read(8192)
            if not chunk:
                break
            feed_data(chunk)
    elif part.name == 'imagedata':
        # Store this body part in a file.
        filename = os.path.join(UPLOAD_PATH, part.secure_filename)
        with open(filename, 'wb') as dest:
```

(continues on next page)

(continued from previous page)

```
            part.stream.pipe(dest)
    else:
        # Do something else
        form_data[part.name] = part.text
```

ASGI

```
form = await req.get_media()
async for part in form:
    if part.content_type == 'application/json':
        # Body part is a JSON document, do something useful with it
        resp.media = await part.get_media()
    elif part.name == 'datafile':
        # Do something with the uploaded data (file)
        async for chunk in part.stream:
            await feed_data(chunk)
    elif part.name == 'imagedata':
        # Store this body part in a file.
        filename = os.path.join(UPLOAD_PATH, part.secure_filename)
        async with aiofiles.open(filename, 'wb') as dest:
            await part.stream.pipe(dest)
    else:
        # Do something else
        form_data[part.name] = await part.text
```

Note

Rather than being read in and buffered all at once, the request stream is only consumed on-demand, while iterating over the body parts in the form.

For each part, you can choose whether to read the whole part into memory, write it out to a file, or *upload it to the cloud*. Falcon offers straightforward support for all of these scenarios.

Multipart Form and Body Part Types

WSGI

class falcon.media.multipart.**MultipartForm**(*stream:* ReadableIO, *boundary:* *bytes*, *content_length:* *int* | *None*, *parse_options:* MultipartParseOptions)

Iterable object that returns each form part as `BodyPart` instances.

Typical usage illustrated below:

```
def on_post(self, req: Request, resp: Response) -> None:
    form: MultipartForm = req.get_media()

    for part in form:
        if part.name == 'foo':
            ...
        else:
            ...
```

Note

`MultipartForm` is meant to be instantiated directly only by the `MultipartFormHandler` parser.

class falcon.media.multipart.**BodyPart**(*stream: PyBufferedReader*, *headers: dict[bytes, bytes]*, *parse_options:* MultipartParseOptions)

Represents a body part in a multipart form in a WSGI application.

> **Note**
>
> *BodyPart* is meant to be instantiated directly only by the *MultipartFormHandler* parser.

property content_type: str

Value of the Content-Type header.

When the header is missing returns the multipart form default `text/plain`.

property data: bytes

Property that acts as a convenience alias for *get_data()*.

```
# Equivalent to: content = part.get_data()
content = part.data
```

property filename: str | None

File name if the body part is an attached file, and `None` otherwise.

get_data() → bytes

Return the body part content bytes.

The maximum number of bytes that may be read is configurable via *MultipartParseOptions*, and a *MultipartParseError* is raised if the body part is larger that this size.

The size limit guards against reading unexpectedly large amount of data into memory by referencing *data* and *text* properties that build upon this method. For large bodies, such as attached files, use the input *stream* directly.

> **Note**
>
> Calling this method the first time will consume the part's input stream. The result is cached for subsequent access, and follow-up calls will just retrieve the cached content.

Returns
The body part content.

Return type
bytes

get_media() → Any

Return a deserialized form of the multipart body part.

When called, this method will attempt to deserialize the body part stream using the Content-Type header as well as the media-type handlers configured via *MultipartParseOptions*.

The result will be cached and returned in subsequent calls:

```
deserialized_media = part.get_media()
```

Returns
The deserialized media representation.

Return type
object

get_text() → str | None

Return the body part content decoded as a text string.

Text is decoded from the part content (as returned by *get_data()*) using the charset specified in the *Content-Type* header, or, if omitted, the *default charset*. The charset must be supported by Python's `bytes.decode()` function. The list of standard encodings (charsets) supported by the Python 3 standard library can be found here.

If decoding fails due to invalid *data* bytes (for the specified encoding), or the specified encoding itself is unsupported, a *MultipartParseError* will be raised when referencing this property.

> **Note**
>
> As this method builds upon *get_data()*, it will consume the part's input stream in the same way.

Returns
The part decoded as a text string provided the part is encoded as `text/plain`, `None` otherwise.

Return type
str

property media: Any

Property that acts as a convenience alias for *get_media()*.

```
# Equivalent to: deserialized_media = part.get_media()
deserialized_media = req.media
```

property name: str | None

The name parameter of the Content-Disposition header.

The value of the "name" parameter is the original field name from the submitted HTML form.

> **Note**
>
> According to RFC 7578, section 4.2, each part MUST include a Content-Disposition header field of type "form-data", where the name parameter is mandatory.
>
> However, Falcon will not raise any error if this parameter is missing; the property value will be `None` in that case.

property secure_filename: str

The sanitized version of *filename* using only the most common ASCII characters for maximum portability and safety wrt using this name as a filename on a regular file system.

If *filename* is empty or unset when referencing this property, an instance of *MultipartParseError* will be raised.

See also: *secure_filename()*

stream: PyBufferedReader

File-like input object for reading the body part of the multipart form request, if any. This object provides direct access to the server's data stream and is non-seekable. The stream is automatically delimited according to the multipart stream boundary.

With the exception of being buffered to keep track of the boundary, the wrapped body part stream interface and behavior mimic *Request.bounded_stream*.

Reading the whole part content:

```
data = part.stream.read()
```

This is also safe:

```
doc = yaml.safe_load(part.stream)
```

property text: str | None

Property that acts as a convenience alias for *get_text()*.

```
# Equivalent to: decoded_text = part.get_text()
decoded_text = part.text
```

ASGI

class falcon.asgi.multipart.**MultipartForm**(*stream:* AsyncReadableIO, *boundary: bytes*, *content_length: int | None*, *parse_options:* MultipartParseOptions)

Iterable object that returns each form part as *BodyPart* instances.

Typical usage illustrated below:

```
async def on_post(self, req: Request, resp: Response) -> None:
    form: MultipartForm = await req.get_media()

    async for part in form:
        if part.name == 'foo':
            ...
        else:
            ...
```

Note

MultipartForm is meant to be instantiated directly only by the `MultipartFormHandler` parser.

class falcon.asgi.multipart.**BodyPart**(*stream: PyBufferedReader*, *headers: dict[bytes, bytes]*, *parse_options:* MultipartParseOptions)

Represents a body part in a multipart form in an ASGI application.

Note

BodyPart is meant to be instantiated directly only by the `MultipartFormHandler` parser.

property data: bytes

Property that acts as a convenience alias for *get_data()*.

The `await` keyword must still be added when referencing the property:

```
# Equivalent to: content = await part.get_data()
content = await part.data
```

async get_data() → bytes

Return the body part content bytes.

The maximum number of bytes that may be read is configurable via *MultipartParseOptions*, and a *MultipartParseError* is raised if the body part is larger that this size.

The size limit guards against reading unexpectedly large amount of data into memory by referencing *data* and *text* properties that build upon this method. For large bodies, such as attached files, use the input *stream* directly.

> **Note**
>
> Calling this method the first time will consume the part's input stream. The result is cached for subsequent access, and follow-up calls will just retrieve the cached content.

Returns
The body part content.

Return type
bytes

async get_media() → Any

Return a deserialized form of the multipart body part.

When called, this method will attempt to deserialize the body part stream using the Content-Type header as well as the media-type handlers configured via *MultipartParseOptions*.

The result will be cached and returned in subsequent calls:

```
deserialized_media = await part.get_media()
```

Returns
The deserialized media representation.

Return type
object

async get_text() → str | None

Return the body part content decoded as a text string.

Text is decoded from the part content (as returned by *get_data()*) using the charset specified in the *Content-Type* header, or, if omitted, the `default charset`. The charset must be supported by Python's `bytes.decode()` function. The list of standard encodings (charsets) supported by the Python 3 standard library can be found here.

If decoding fails due to invalid *data* bytes (for the specified encoding), or the specified encoding itself is unsupported, a *MultipartParseError* will be raised when referencing this property.

> **Note**
>
> As this method builds upon *get_data()*, it will consume the part's input stream in the same way.

Returns
The part decoded as a text string provided the part is encoded as `text/plain`, `None` otherwise.

Return type
str

property media: Any

Property that acts as a convenience alias for *get_media()*.

The `await` keyword must still be added when referencing the property:

```
# Equivalent to: deserialized_media = await part.get_media()
deserialized_media = await part.media
```

stream: BufferedReader

File-like input object for reading the body part of the multipart form request, if any. This object provides direct access to the server's data stream and is non-seekable. The stream is automatically delimited according to the multipart stream boundary.

With the exception of being buffered to keep track of the boundary, the wrapped body part stream interface and behavior mimic *Request.stream*.

Similarly to *BoundedStream*, the most efficient way to read the body part content is asynchronous iteration over part data chunks:

```
async for data_chunk in part.stream:
    pass
```

property text: str | None

Property that acts as a convenience alias for *get_text()*.

The `await` keyword must still be added when referencing the property:

```
# Equivalent to: decoded_text = await part.get_text()
decoded_text = await part.text
```

Parser Configuration

Similar to *falcon.App*'s *req_options* and *resp_options*, instantiating a *MultipartFormHandler* also fills its `parse_options` attribute with a set of sane default values suitable for many use cases out of the box. If you need to customize certain form parsing aspects of your application, the preferred way is to directly modify the properties of this attribute on the media handler (parser) in question:

```
import falcon
import falcon.media

handler = falcon.media.MultipartFormHandler()

# Assume text fields to be encoded in latin-1 instead of utf-8
handler.parse_options.default_charset = 'latin-1'

# Allow an unlimited number of body parts in the form
handler.parse_options.max_body_part_count = 0

# Afford parsing msgpack-encoded body parts directly via part.get_media()
extra_handlers = {
    falcon.MEDIA_MSGPACK: falcon.media.MessagePackHandler(),
}
handler.parse_options.media_handlers.update(extra_handlers)
```

In order to use your customized handler in an app, simply replace the default handler for `multipart/form-data` with the new one:

WSGI

```
app = falcon.App()

# handler is instantiated and configured as per the above snippet
app.req_options.media_handlers[falcon.MEDIA_MULTIPART] = handler
```

ASGI

```
app = falcon.asgi.App()

# handler is instantiated and configured as per the above snippet
app.req_options.media_handlers[falcon.MEDIA_MULTIPART] = handler
```

> **Tip**
>
> For more information on customizing media handlers, see also: *Replacing the Default Handlers*.

Parsing Options

class falcon.media.multipart.**MultipartParseOptions**

Defines a set of configurable multipart form parser options.

An instance of this class is exposed via the `MultipartFormHandler.parse_options` attribute. The handler's options are also passed down to every *BodyPart* it instantiates.

See also: *Parser Configuration*.

default_charset: **str**

The default character encoding for *text fields* (default `utf-8`).

max_body_part_buffer_size: **int**

The maximum number of bytes to buffer and return when the *BodyPart.get_data()* method is called (default `1 MiB`).

If the body part size exceeds this value, an instance of *MultipartParseError* will be raised.

max_body_part_count: **int**

The maximum number of body parts in the form (default `64`).

If the form contains more parts than this number, an instance of *MultipartParseError* will be raised. If this option is set to 0, no limit will be imposed by the parser.

max_body_part_headers_size: **int**

The maximum size (in bytes) of the body part headers structure (default `8192`).

If the body part headers size exceeds this value, an instance of *MultipartParseError* will be raised.

max_secure_filename_length: **int** | **None**

The maximum number characters for a secure filename (default `None`).

The value of this option is passed as the *max_length* keyword argument to *secure_filename()* when evaluating the *BodyPart.secure_filename* property.

> **Note**
>
> In Falcon 5.0, the default value of this option will change to a reasonable finite number (e.g., 64 or 96) of characters.

Added in version 4.1.

media_handlers: ***Handlers***

A dict-like object for configuring the media-types to handle.

By default, handlers are provided for the `application/json` and `application/x-www-form-urlencoded` media types.

Parsing Errors

class falcon.**MultipartParseError**(*, *description: str | None = None*, ***kwargs: HeaderArg | HTTPErrorKeywordArguments*)

Represents a multipart form parsing error.

This error may refer to a malformed or truncated form, usage of deprecated or unsupported features, or form parameters exceeding limits configured in *MultipartParseOptions*.

MultipartParseError instances raised in this module always include a short human-readable description of the error.

The cause of this exception, if any, is stored in the __cause__ attribute using the "raise … from" form when raising.

Parameters

source_error (*Exception*) – The source exception that was the cause of this one.

code: int | None

An internal application code that a user can reference when requesting support for the error.

description: str | None

Description of the error to send to the client.

headers: HeaderArg | None

Extra headers to add to the response.

link: Link | None

An href that the client can provide to the user for getting help.

status: ResponseStatus

HTTP status code or line (e.g., '200 OK').

This may be set to a member of http.HTTPStatus, an HTTP status line string or byte string (e.g., '200 OK'), or an int.

title: str

Error title to send to the client.

Derived from the status if not provided.

5.3.9 Redirection

Falcon defines a set of exceptions that can be raised within a middleware method, hook, or responder in order to trigger a 3xx (Redirection) response to the client. Raising one of these classes short-circuits request processing in a manner similar to raising an instance or subclass of *HTTPError*.

exception falcon.**HTTPMovedPermanently**(*location: str*, *headers: Headers | None = None*)

301 Moved Permanently.

The 301 (Moved Permanently) status code indicates that the target resource has been assigned a new permanent URI.

> **Note**
>
> For historical reasons, a user agent MAY change the request method from POST to GET for the subsequent request. If this behavior is undesired, the 308 (Permanent Redirect) status code can be used instead.

(See also: RFC 7231, Section 6.4.2)

Parameters

location (*str*) – URI to provide as the Location header in the response.

exception falcon.**HTTPFound**(*location: str*, *headers: Headers | None = None*)

302 Found.

The 302 (Found) status code indicates that the target resource resides temporarily under a different URI. Since the redirection might be altered on occasion, the client ought to continue to use the effective request URI for future requests.

> **Note**
>
> For historical reasons, a user agent MAY change the request method from POST to GET for the subsequent request. If this behavior is undesired, the 307 (Temporary Redirect) status code can be used instead.

(See also: RFC 7231, Section 6.4.3)

Parameters

location (*str*) – URI to provide as the Location header in the response.

exception falcon.**HTTPSeeOther**(*location: str*, *headers: Headers | None = None*)

303 See Other.

The 303 (See Other) status code indicates that the server is redirecting the user agent to a *different* resource, as indicated by a URI in the Location header field, which is intended to provide an indirect response to the original request.

A 303 response to a GET request indicates that the origin server does not have a representation of the target resource that can be transferred over HTTP. However, the Location header in the response may be dereferenced to obtain a representation for an alternative resource. The recipient may find this alternative useful, even though it does not represent the original target resource.

> **Note**
>
> The new URI in the Location header field is not considered equivalent to the effective request URI.

(See also: RFC 7231, Section 6.4.4)

Parameters

location (*str*) – URI to provide as the Location header in the response.

exception falcon.**HTTPTemporaryRedirect**(*location: str*, *headers: Headers | None = None*)

307 Temporary Redirect.

The 307 (Temporary Redirect) status code indicates that the target resource resides temporarily under a different URI and the user agent MUST NOT change the request method if it performs an automatic redirection to that URI. Since the redirection can change over time, the client ought to continue using the original effective request URI for future requests.

> **Note**
>
> This status code is similar to 302 (Found), except that it does not allow changing the request method from POST to GET.

(See also: RFC 7231, Section 6.4.7)

Parameters

location (*str*) – URI to provide as the Location header in the response.

exception falcon.**HTTPPermanentRedirect**(*location: str*, *headers: Headers | None = None*)

308 Permanent Redirect.

The 308 (Permanent Redirect) status code indicates that the target resource has been assigned a new permanent URI.

> **Note**
>
> This status code is similar to 301 (Moved Permanently), except that it does not allow changing the request method from POST to GET.

(See also: RFC 7238, Section 3)

Parameters

location (*str*) – URI to provide as the Location header in the response.

5.3.10 Middleware

Middleware components provide a way to execute logic before the framework routes each request, after each request is routed but before the target responder is called, or just before the response is returned for each request.

> **Note**
>
> Unlike hooks, middleware methods apply globally to the entire App.

Components are registered with the *middleware* kwarg when instantiating Falcon's *App class*. A middleware component is simply a class that implements one or more of the event handler methods defined below.

WSGI

Falcon's middleware interface is defined as follows:

```
from typing import Any
from falcon import Request, Response

class ExampleMiddleware:
    def process_request(self, req: Request, resp: Response) -> None:
        """Process the request before routing it.

        Note:
            Because Falcon routes each request based on req.path, a
            request can be effectively re-routed by setting that
            attribute to a new value from within process_request().

        Args:
            req: Request object that will eventually be
                routed to an on_* responder method.
            resp: Response object that will be routed to
                the on_* responder.
        """

    def process_resource(
        self,
        req: Request,
        resp: Response,
        resource: object,
        params: dict[str, Any],
    ) -> None:
        """Process the request after routing.
```

(continues on next page)

(continued from previous page)

```
        Note:
            This method is only called when the request matches
            a route to a resource.

        Args:
            req: Request object that will be passed to the
                routed responder.
            resp: Response object that will be passed to the
                responder.
            resource: Resource object to which the request was
                routed.
            params: A dict-like object representing any additional
                params derived from the route's URI template fields,
                that will be passed to the resource's responder
                method as keyword arguments.
        """

    def process_response(
        self,
        req: Request,
        resp: Response,
        resource: object,
        req_succeeded: bool
    ) -> None:
        """Post-processing of the response (after routing).

        Args:
            req: Request object.
            resp: Response object.
            resource: Resource object to which the request was
                routed. May be None if no route was found
                for the request.
            req_succeeded: True if no exceptions were raised while
                the framework processed and routed the request;
                otherwise False.
        """

# A middleware instance can then be added to the Falcon app init
from falcon import App

app = App(middleware=[ExampleMiddleware()])
```

ASGI

The ASGI middleware interface is similar to WSGI, but also supports the standard ASGI lifespan events. However, because lifespan events are an optional part of the ASGI specification, they may or may not fire depending on your ASGI server.

```
from typing import Any
from falcon.asgi import Request, Response, WebSocket

class ExampleMiddleware:
    async def process_startup(
        self, scope: dict[str, Any], event: dict[str, Any]
    ) -> None:
```

(continues on next page)

(continued from previous page)

```
        """Process the ASGI lifespan startup event.

        Invoked when the server is ready to start up and
        receive connections, but before it has started to
        do so.

        To halt startup processing and signal to the server that it
        should terminate, simply raise an exception and the
        framework will convert it to a "lifespan.startup.failed"
        event for the server.

        Args:
            scope (dict): The ASGI scope dictionary for the
                lifespan protocol. The lifespan scope exists
                for the duration of the event loop.
            event (dict): The ASGI event dictionary for the
                startup event.
        """

    async def process_shutdown(
        self, scope: dict[str, Any], event: dict[str, Any]
    ) -> None:
        """Process the ASGI lifespan shutdown event.

        Invoked when the server has stopped accepting
        connections and closed all active connections.

        To halt shutdown processing and signal to the server
        that it should immediately terminate, simply raise an
        exception and the framework will convert it to a
        "lifespan.shutdown.failed" event for the server.

        Args:
            scope (dict): The ASGI scope dictionary for the
                lifespan protocol. The lifespan scope exists
                for the duration of the event loop.
            event (dict): The ASGI event dictionary for the
                shutdown event.
        """

    async def process_request(self, req: Request, resp: Response) -> None:
        """Process the request before routing it.

        Note:
            Because Falcon routes each request based on req.path, a
            request can be effectively re-routed by setting that
            attribute to a new value from within process_request().

        Args:
            req: Request object that will eventually be
                routed to an on_* responder method.
            resp: Response object that will be routed to
                the on_* responder.
        """

    async def process_resource(
```

(continues on next page)

(continued from previous page)

```
        self,
        req: Request,
        resp: Response,
        resource: object,
        params: dict[str, Any],
    ) -> None:
        """Process the request after routing.

        Note:
            This method is only called when the request matches
            a route to a resource.

        Args:
            req: Request object that will be passed to the
                routed responder.
            resp: Response object that will be passed to the
                responder.
            resource: Resource object to which the request was
                routed.
            params: A dict-like object representing any additional
                params derived from the route's URI template fields,
                that will be passed to the resource's responder
                method as keyword arguments.
        """

    async def process_response(
        self,
        req: Request,
        resp: Response,
        resource: object,
        req_succeeded: bool
    ) -> None:
        """Post-processing of the response (after routing).

        Args:
            req: Request object.
            resp: Response object.
            resource: Resource object to which the request was
                routed. May be None if no route was found
                for the request.
            req_succeeded: True if no exceptions were raised while
                the framework processed and routed the request;
                otherwise False.
        """

    async def process_request_ws(self, req: Request, ws: WebSocket) -> None:
        """Process a WebSocket handshake request before routing it.

        Note:
            Because Falcon routes each request based on req.path, a
            request can be effectively re-routed by setting that
            attribute to a new value from within process_request().

        Args:
            req: Request object that will eventually be
                passed into an on_websocket() responder method.
```

(continues on next page)

(continued from previous page)

```
            ws: The WebSocket object that will be passed into
                on_websocket() after routing.
        """

    async def process_resource_ws(
        self,
        req: Request,
        ws: WebSocket,
        resource: object,
        params: dict[str, Any],
    ) -> None:
        """Process a WebSocket handshake request after routing.

        Note:
            This method is only called when the request matches
            a route to a resource.

        Args:
            req: Request object that will be passed to the
                routed responder.
            ws: WebSocket object that will be passed to the
                routed responder.
            resource: Resource object to which the request was
                routed.
            params: A dict-like object representing any additional
                params derived from the route's URI template fields,
                that will be passed to the resource's responder
                method as keyword arguments.
        """

# A middleware instance can then be added to the Falcon app init
from falcon.asgi import App

app = App(middleware=[ExampleMiddleware()])
```

It is also possible to implement a middleware component that is compatible with both ASGI and WSGI apps. This is done by applying an **_async* postfix to distinguish the two different versions of each middleware method, as in the following example:

```
import falcon as wsgi
from falcon import asgi

class ExampleMiddleware:
    def process_request(self, req: wsgi.Request, resp: wsgi.Response) -> None:
        """Process WSGI request using synchronous logic.

        Note that req and resp are instances of falcon.Request and
        falcon.Response, respectively.
        """

    async def process_request_async(self, req: asgi.Request, resp: asgi.Response) -
↪> None:
        """Process ASGI request using asynchronous logic.

        Note that req and resp are instances of falcon.asgi.Request and
        falcon.asgi.Response, respectively.
        """
```

Tip

Because *process_request* executes before routing has occurred, if a component modifies `req.path` in its *process_request* method, the framework will use the modified value to route the request.

For example:

```
# Route requests based on the host header.
req.path = '/' + req.host + req.path
```

Tip

The *process_resource* method is only called when the request matches a route to a resource. To take action when a route is not found, a *sink* may be used instead.

Tip

In order to pass data from a middleware function to a resource function use the `req.context` and `resp.context` objects. These context objects are intended to hold request and response data specific to your app as it passes through the framework.

Each component's *process_request*, *process_resource*, and *process_response* methods are executed hierarchically, as a stack, following the ordering of the list passed via the *middleware* kwarg of *falcon.App* or *falcon.asgi.App*. For example, if a list of middleware objects are passed as `[mob1, mob2, mob3]`, the order of execution is as follows:

```
mob1.process_request
    mob2.process_request
        mob3.process_request
            mob1.process_resource
                mob2.process_resource
                    mob3.process_resource
            <route to resource responder method>
        mob3.process_response
    mob2.process_response
mob1.process_response
```

Note that each component need not implement all *process_** methods; in the case that one of the three methods is missing, it is treated as a noop in the stack. For example, if `mob2` did not implement *process_request* and `mob3` did not implement *process_response*, the execution order would look like this:

```
mob1.process_request
    _
        mob3.process_request
            mob1.process_resource
                mob2.process_resource
                    mob3.process_resource
            <route to responder method>
        _
    mob2.process_response
mob1.process_response
```

Short-Circuiting

A *process_request* or *process_resource* middleware method may short-circuit further request processing by setting *falcon.Response.complete* to `True`, e.g.:

```
resp.complete = True
```

After the method returns, setting this flag will cause the framework to skip any remaining *process_request* and *process_resource* methods, as well as the responder method that the request would have been routed to. However, any *process_response* middleware methods will still be called.

In a similar manner, setting *falcon.Response.complete* to `True` from within a *process_resource* method will short-circuit further request processing at that point.

In the example below, you can see how request processing will be short-circuited once *falcon.Response.complete* has been set to `True`, i.e., the framework will prevent `mob3.process_request`, all *process_resource* methods, as well as the routed responder method from processing the request. However, all *process_response* methods will still be called:

```
mob1.process_request
    mob2.process_request  # resp.complete = True
        <skip mob3.process_request>
        <skip mob1/mob2/mob3.process_resource>
        <skip route to resource responder method>
        mob3.process_response
    mob2.process_response
mob1.process_response
```

This feature affords use cases in which the response may be pre-constructed, such as in the case of caching.

Exception Handling

If one of the *process_request* middleware methods raises an exception, it will be processed according to the exception type. If the type matches a registered error handler, that handler will be invoked and then the framework will begin to unwind the stack, skipping any lower layers. The error handler may itself raise an instance of *HTTPError* or *HTTPStatus*, in which case the framework will use the latter exception to update the *resp* object.

Note

By default, the framework installs two handlers, one for *HTTPError* and one for *HTTPStatus*. These can be overridden via *add_error_handler()*.

Regardless, the framework will continue unwinding the middleware stack. For example, if *mob2.process_request* were to raise an error, the framework would execute the stack as follows:

```
mob1.process_request
    mob2.process_request
        <skip mob1/mob2 process_resource>
        <skip mob3.process_request>
        <skip mob3.process_resource>
        <skip route to resource responder method>
        mob3.process_response
    mob2.process_response
mob1.process_response
```

As illustrated above, by default, all *process_response* methods will be executed, even when a *process_request*, *process_resource*, or *on_** resource responder raises an error. This behavior is controlled by the *App class's independent_middleware* keyword argument.

Finally, if one of the *process_response* methods raises an error, or the routed `on_*` responder method itself raises an error, the exception will be handled in a similar manner as above. Then, the framework will execute any remaining middleware on the stack.

5.3.11 CORS

Cross Origin Resource Sharing (CORS) is an additional security check performed by modern browsers to prevent unauthorized requests between different domains.

When developing a web API, it is common to also implement a CORS policy. Therefore, Falcon provides an easy way to enable a simple CORS policy via a flag passed to *falcon.App* or *falcon.asgi.App*.

By default, Falcon's built-in CORS support is disabled, so that any cross-origin requests will be blocked by the browser. Passing `cors_enable=True` will cause the framework to include the necessary response headers to allow access from any origin to any route in the app. Individual responders may override this behavior by setting the `Access-Control-Allow-Origin` header explicitly.

Whether or not you implement a CORS policy, we recommend also putting a robust AuthN/Z layer in place to authorize individual clients, as needed, to protect sensitive resources.

Directly passing the *falcon.CORSMiddleware* middleware to the application allows customization of the CORS policy applied. The middleware allows customizing the allowed origins, if credentials should be allowed and if additional headers can be exposed.

Usage

WSGI

```
import falcon

# Enable a simple CORS policy for all responses
app = falcon.App(cors_enable=True)

# Alternatively, enable CORS policy for example.com and allow
# credentials
app = falcon.App(middleware=falcon.CORSMiddleware(
    allow_origins='example.com', allow_credentials='*'))
```

ASGI

```
import falcon.asgi

# Enable a simple CORS policy for all responses
app = falcon.asgi.App(cors_enable=True)

# Alternatively, Enable CORS policy for example.com and allow
# credentials
app = falcon.asgi.App(middleware=falcon.CORSMiddleware(
    allow_origins='example.com', allow_credentials='*'))
```

Note

Passing the `cors_enable` parameter set to `True` is mutually exclusive with directly passing an instance of *CORSMiddleware* to the application's initializer.

Changed in version 4.0: Attempt to use the combination of `cors_enable=True` and an additional instance of *CORSMiddleware* now results in a `ValueError`.

CORSMiddleware

class falcon.**CORSMiddleware**(*allow_origins: str | Iterable[str] = '*', expose_headers: str | Iterable[str] | None = None, allow_credentials: str | Iterable[str] | None = None, allow_private_network: bool = False*)

CORS Middleware.

This middleware provides a simple out-of-the box CORS policy, including handling of preflighted requests from the browser.

See also:

- https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
- https://www.w3.org/TR/cors/#resource-processing-model

> **Note**
>
> Falcon will automatically add OPTIONS responders if they are missing from the responder instances added to the routes. When providing a custom `on_options` method, the `Allow` headers in the response should be set to the allowed method values. If the `Allow` header is missing from the response, this middleware will deny the preflight request.
>
> This is also valid when using a sink function.

Keyword Arguments

- **allow_origins** (*Union[str, Iterable[str]]*) – List of origins to allow (case sensitive). The string `'*'` acts as a wildcard, matching every origin. (default `'*'`).
- **expose_headers** (*Optional[Union[str, Iterable[str]]]*) – List of additional response headers to expose via the `Access-Control-Expose-Headers` header. These headers are in addition to the CORS-safelisted ones: `Cache-Control`, `Content-Language`, `Content-Length`, `Content-Type`, `Expires`, `Last-Modified`, `Pragma`. (default `None`).

  See also: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Expose-Headers
- **allow_credentials** (*Optional[Union[str, Iterable[str]]]*) – List of origins (case sensitive) for which to allow credentials via the `Access-Control-Allow-Credentials` header. The string `'*'` acts as a wildcard, matching every allowed origin, while `None` disallows all origins. This parameter takes effect only if the origin is allowed by the `allow_origins` argument. (default `None`).
- **allow_private_network** (*bool*) – If `True`, the server includes the `Access-Control-Allow-Private-Network` header in responses to CORS preflight (OPTIONS) requests. This indicates that the resource is willing to respond to requests from less-public IP address spaces (e.g., from public site to private device). (default `False`).

  See also: https://wicg.github.io/private-network-access/#private-network-request-heading

5.3.12 Hooks

Falcon supports *before* and *after* hooks. You install a hook simply by applying one of the decorators below, either to an individual responder or to an entire resource.

For example, consider this hook that validates a POST request for an image resource:

```
def validate_image_type(req, resp, resource, params):
    if req.content_type not in ALLOWED_IMAGE_TYPES:
        msg = 'Image type not allowed. Must be PNG, JPEG, or GIF'
        raise falcon.HTTPBadRequest(title='Bad request', description=msg)
```

You would attach this hook to an `on_post` responder like so:

```
@falcon.before(validate_image_type)
def on_post(self, req, resp):
    pass
```

Or, suppose you had a hook that you would like to apply to *all* responders for a given resource. In that case, you would simply decorate the resource class:

```
def extract_project_id(req, resp, resource, params):
    params['project_id'] = None
    if req.content_type == falcon.MEDIA_JSON:
        params['project_id'] = req.get_media().get('projectId')

@falcon.before(extract_project_id)
class Message:
    def on_post(self, req, resp, project_id):
        pass

    def on_get(self, req, resp, project_id):
        pass
```

In addition to just running your logic before all responders of this resource, the `extract_project_id` hook also injects the *project_id* parameter – see *falcon.before()* for a more elaborate explanation.

Note

When decorating an entire resource class, all method names that resemble responders, including *suffix*ed (see also *add_route()*) ones, are decorated. If, for instance, a method is called `on_get_items`, but it is not meant for handling `GET` requests under a route with the *suffix* `items`, the easiest workaround for preventing the hook function from being applied to the method is renaming it not to clash with the responder pattern.

Note also that you can pass additional arguments to your hook function as needed:

```
def validate_image_type(req, resp, resource, params, allowed_types):
    if req.content_type not in allowed_types:
        msg = 'Image type not allowed.'
        raise falcon.HTTPBadRequest(title='Bad request', description=msg)

@falcon.before(validate_image_type, ['image/png'])
def on_post(self, req, resp):
    pass
```

Falcon supports using any callable as a hook. This allows for using a class instead of a function:

```
class Authorize:
    def __init__(self, roles):
        self._roles = roles

    def __call__(self, req, resp, resource, params):
        pass
```

(continues on next page)

(continued from previous page)

```
@falcon.before(Authorize(['admin']))
def on_post(self, req, resp):
    pass
```

Falcon *middleware components* can also be used to insert logic before and after requests. However, unlike hooks, *middleware components* are triggered **globally** for all requests.

> **Tip**
>
> In order to pass data from a hook function to a resource function use the `req.context` and `resp.context` objects. These context objects are intended to hold request and response data specific to your app as it passes through the framework.

Before Hooks

`falcon.`**`before`**(*action: Callable[Concatenate[wsgi.Request, wsgi.Response, Resource, dict[str, Any], None] | Callable[Concatenate[asgi.Request, asgi.Response, Resource, dict[str, Any], Awaitable[None]], *args: _FN.args, **kwargs: _FN.kwargs*) → Callable[[_R], _R]

Execute the given action function *before* the responder.

The *params* argument that is passed to the hook contains only the fields from the URI template path; it does not include query string values.

Hooks may inject extra params as needed. For example:

```
def do_something(req, resp, resource, params):
    try:
        params['id'] = int(params['id'])
    except ValueError:
        raise falcon.HTTPBadRequest(title='Invalid ID',
                                    description='ID was not valid.')

    params['answer'] = 42
```

Parameters

- **`action`** (*callable*) – A function of the form `func(req, resp, resource, params)`, where *resource* is a reference to the resource class instance associated with the request and *params* is a dict of URI template field names, if any, that will be passed into the resource responder as kwargs.
- **`*args`** – Any additional arguments will be passed to *action* in the order given, immediately following the *req*, *resp*, *resource*, and *params* arguments.

Keyword Arguments

`**kwargs` – Any additional keyword arguments will be passed through to *action*.

After Hooks

`falcon.`**`after`**(*action: Callable[Concatenate[wsgi.Request, wsgi.Response, Resource, None] | Callable[Concatenate[asgi.Request, asgi.Response, Resource, Awaitable[None]], *args: _FN.args, **kwargs: _FN.kwargs*) → Callable[[_R], _R]

Execute the given action function *after* the responder.

Parameters

- **`action`** (*callable*) – A function of the form `func(req, resp, resource)`, where *resource* is a reference to the resource class instance associated with the request

- ***args** – Any additional arguments will be passed to *action* in the order given, immediately following the *req*, *resp* and *resource* arguments.

Keyword Arguments

****kwargs** – Any additional keyword arguments will be passed through to *action*.

5.3.13 Routing

Falcon uses resource-based routing to encourage a RESTful architectural style. Each resource is represented by a class that is responsible for handling all of the HTTP methods that the resource supports.

For each HTTP method supported by the resource, the class implements a corresponding Python method with a name that starts with `on_` and ends in the lowercased HTTP method name (e.g., `on_get()`, `on_patch()`, `on_delete()`, etc.)

> **Note**
>
> Resources in Falcon are represented by a single class instance that is created at application startup when the routes are configured. This minimizes routing overhead and simplifies the implementation of resource classes. In the case of WSGI apps, this also means that resource classes must be implemented in a thread-safe manner (see also: *Is Falcon thread-safe?*).

Falcon routes incoming requests (including *WebSocket handshakes*) to resources based on a set of URI templates. If the path requested by the client matches the template for a given route, the request is then passed on to the associated resource for processing.

Here's a quick example to show how all the pieces fit together:

WSGI

```
import json

import falcon


class ImagesResource:

    def on_get(self, req, resp):
        doc = {
            'images': [
                {
                    'href': '/images/1eaf6ef1-7f2d-4ecc-a8d5-6e8adba7cc0e.png'
                }
            ]
        }

        # Create a JSON representation of the resource; this could
        #   also be done automatically by assigning to resp.media
        resp.text = json.dumps(doc, ensure_ascii=False)

        # The following line can be omitted because 200 is the default
        # status returned by the framework, but it is included here to
        # illustrate how this may be overridden as needed.
        resp.status = falcon.HTTP_200


app = falcon.App()
```

(continues on next page)

(continued from previous page)

```
images = ImagesResource()
app.add_route('/images', images)
```

ASGI

```
import json

import falcon
import falcon.asgi


class ImagesResource:

    async def on_get(self, req, resp):
        doc = {
            'images': [
                {
                    'href': '/images/1eaf6ef1-7f2d-4ecc-a8d5-6e8adba7cc0e.png'
                }
            ]
        }

        # Create a JSON representation of the resource; this could
        #   also be done automatically by assigning to resp.media
        resp.text = json.dumps(doc, ensure_ascii=False)

        # The following line can be omitted because 200 is the default
        # status returned by the framework, but it is included here to
        # illustrate how this may be overridden as needed.
        resp.status = falcon.HTTP_200


app = falcon.asgi.App()

images = ImagesResource()
app.add_route('/images', images)
```

If no route matches the request, control then passes to a default responder that simply raises an instance of *HTTPRouteNotFound*. By default, this error will be rendered as a 404 response for a regular HTTP request, and a 403 response with a 3404 close code for a *WebSocket* handshake. This behavior can be modified by adding a custom error handler (see also *this FAQ topic*).

On the other hand, if a route is matched but the resource does not implement a responder for the requested HTTP method, the framework invokes a default responder that raises an instance of *HTTPMethodNotAllowed*. This class will be rendered by default as a 405 response for a regular HTTP request, and a 403 response with a 3405 close code for a *WebSocket* handshake.

Falcon also provides a default responder for OPTIONS requests that takes into account which methods are implemented for the target resource.

Default Behavior

Falcon's default routing engine is based on a decision tree that is first compiled into Python code, and then evaluated by the runtime. By default, the decision tree is compiled only when the router handles the first request. See *CompiledRouter* for more details.

The *falcon.App.add_route()* and *falcon.asgi.App.add_route()* methods are used to associate a URI template with a resource. Falcon then maps incoming requests to resources based on these templates.

Falcon's default router uses Python classes to represent resources. In practice, these classes act as controllers in your application. They convert an incoming request into one or more internal actions, and then compose a response back to the client based on the results of those actions. (See also: *Tutorial: Creating Resources*)

```
                ┌────────────┐
request   →     │            │
                │ Resource   │ ↻ Orchestrate the requested action
                │ Controller │ ↻ Compose the result
response  ←     │            │
                └────────────┘
```

Each resource class defines various "responder" methods, one for each HTTP method the resource allows. Responder names start with `on_` and are named according to which HTTP method they handle, as in `on_get()`, `on_post()`, `on_put()`, etc.

Note

If your resource does not support a particular HTTP method, simply omit the corresponding responder and Falcon will use a default responder that raises an instance of *HTTPMethodNotAllowed* when that method is requested. Normally this results in sending a 405 response back to the client.

Responders must always define at least two arguments to receive *Request* and *Response* objects, respectively:

```
def on_post(self, req, resp):
    pass
```

For ASGI apps, the responder must be a coroutine function:

```
async def on_post(self, req, resp):
    pass
```

The *Request* object represents the incoming HTTP request. It exposes properties and methods for examining headers, query string parameters, and other metadata associated with the request. A file-like stream object is also provided for reading any data that was included in the body of the request.

The *Response* object represents the application's HTTP response to the above request. It provides properties and methods for setting status, header and body data. The *Response* object also exposes a dict-like *context* property for passing arbitrary data to hooks and middleware methods.

Note

Rather than directly manipulate the *Response* object, a responder may raise an instance of either *HTTPError* or *HTTPStatus*. Falcon will convert these exceptions to appropriate HTTP responses. Alternatively, you can handle them yourself via *add_error_handler()*.

In addition to the standard *req* and *resp* parameters, if the route's template contains field expressions, any responder that desires to receive requests for that route must accept arguments named after the respective field names defined in the template.

A field expression consists of a bracketed field name. For example, given the following template:

```
/user/{name}
```

A PUT request to `'/user/kgriffs'` would cause the framework to invoke the `on_put()` responder method on the route's resource class, passing `'kgriffs'` via an additional *name* argument defined by the responder:

WSGI

```
# Template fields correspond to named arguments or keyword
#   arguments, following the usual req and resp args.
def on_put(self, req, resp, name):
    pass
```

ASGI

```
# Template fields correspond to named arguments or keyword
#   arguments, following the usual req and resp args.
async def on_put(self, req, resp, name):
    pass
```

Because field names correspond to argument names in responder methods, they must be valid Python identifiers.

Individual path segments may contain one or more field expressions, and fields need not span the entire path segment. For example:

```
/repos/{org}/{repo}/compare/{usr0}:{branch0}...{usr1}:{branch1}
/serviceRoot/People('{name}')
```

(See also the *Falcon tutorial* for additional examples and a walkthrough of setting up routes within the context of a sample application.)

Field Converters

Falcon's default router supports the use of field converters to transform a URI template field value. Field converters may also perform simple input validation. For example, the following URI template uses the *int* converter to convert the value of *tid* to a Python `int`, but only if it has exactly eight digits:

```
/teams/{tid:int(8)}
```

If the value is malformed and can not be converted, Falcon will reject the request with a 404 response to the client.

Converters are instantiated with the argument specification given in the field expression. These specifications follow the standard Python syntax for passing arguments. For example, the comments in the following code show how a converter would be instantiated given different argument specifications in the URI template:

```
# IntConverter()
app.add_route(
    '/a/{some_field:int}',
    some_resource
)

# IntConverter(8)
app.add_route(
    '/b/{some_field:int(8)}',
    some_resource
)

# IntConverter(8, min=10000000)
app.add_route(
    '/c/{some_field:int(8, min=10000000)}',
    some_resource
)
```

(See also how `UUIDConverter` is used in Falcon's ASGI tutorial: *Images Resource(s)*.)

Built-in Converters

Identifier	Class	Example
int	*IntConverter*	/teams/{tid:int(8)}
uuid	*UUIDConverter*	/diff/{left:uuid}...{right:uuid}
dt	*DateTimeConverter*	/logs/{day:dt("%Y-%m-%d")}
float	*FloatConverter*	/python/versions/{version:float(min=3.7)}
path	*PathConverter*	/prefix/{other:path}

class falcon.routing.**IntConverter**(*num_digits: int | None = None*, *min: int | None = None*, *max: int | None = None*)

Converts a field value to an int.

Identifier: *int*

Keyword Arguments

- **num_digits** (*int*) – Require the value to have the given number of digits.
- **min** (*int*) – Reject the value if it is less than this number.
- **max** (*int*) – Reject the value if it is greater than this number.

convert(*value: str*) → int | None

Convert a URI template field value to another format or type.

Parameters

value (*str or list[str]*) – Original string to convert. If CONSUME_MULTIPLE_SEGMENTS=True this value is a list of strings containing the path segments matched by the converter.

Returns

Converted field value, or None if the field can not be converted.

Return type

object

class falcon.routing.**FloatConverter**(*min: float | None = None*, *max: float | None = None*, *finite: bool = True*)

Converts a field value to an float.

Identifier: *float*

Keyword Arguments

- **min** (*float*) – Reject the value if it is less than this number.
- **max** (*float*) – Reject the value if it is greater than this number.
- **finite** (*bool*) – Determines whether or not to only match ordinary finite numbers (default: True). Set to False to match nan, inf, and -inf in addition to finite numbers.

Added in version 4.0.

convert(*value: str*) → float | None

Convert a URI template field value to another format or type.

Parameters
value (*str or list[str]*) – Original string to convert. If CONSUME_MULTIPLE_SEGMENTS=True this value is a list of strings containing the path segments matched by the converter.

Returns
Converted field value, or None if the field can not be converted.

Return type
object

class falcon.routing.**UUIDConverter**

Converts a field value to a uuid.UUID.

Identifier: *uuid*

In order to be converted, the field value must consist of a string of 32 hexadecimal digits, as defined in RFC 4122, Section 3. Note, however, that hyphens and the URN prefix are optional.

convert(*value: str*) → UUID | None

Convert a URI template field value to another format or type.

Parameters
value (*str or list[str]*) – Original string to convert. If CONSUME_MULTIPLE_SEGMENTS=True this value is a list of strings containing the path segments matched by the converter.

Returns
Converted field value, or None if the field can not be converted.

Return type
object

class falcon.routing.**DateTimeConverter**(*format_string: str = '%Y-%m-%dT%H:%M:%S%z'*)

Converts a field value to a datetime.

Identifier: *dt*

Keyword Arguments
format_string (*str*) – String used to parse the field value into a datetime. Any format recognized by strptime() is supported (default '%Y-%m-%dT%H:%M:%S%z').

Changed in version 4.0: The default value of *format_string* was changed from '%Y-%m-%dT%H:%M:%SZ' to '%Y-%m-%dT%H:%M:%S%z'.

The new format is a superset of the old one parsing-wise, however, the converted datetime object is now timezone-aware.

convert(*value: str*) → datetime | None

Convert a URI template field value to another format or type.

Parameters
value (*str or list[str]*) – Original string to convert. If CONSUME_MULTIPLE_SEGMENTS=True this value is a list of strings containing the path segments matched by the converter.

Returns
Converted field value, or None if the field can not be converted.

Return type
object

class falcon.routing.**PathConverter**

Field converted used to match the rest of the path.

This field converter matches the remainder of the URL path, returning it as a string.

This converter is currently supported only when used at the end of the URL template.

The classic routing rules of falcon apply also to this converter: considering the template `'/foo/bar/{matched_path:path}'`, the path `'/foo/bar'` will *not* match the route; `'/foo/bar/'` will match, producing `matched_path=''`, when *strip_url_path_trailing_slash* is `False` (the default), while it will *not* match when that option is `True`.

(See also: *How does Falcon handle a trailing slash in the request path?*)

Added in version 4.0.

CONSUME_MULTIPLE_SEGMENTS: ClassVar[bool] = True

When set to `True` it indicates that this converter will consume multiple URL path segments. Currently a converter with `CONSUME_MULTIPLE_SEGMENTS=True` must be at the end of the URL template effectively meaning that it will consume all of the remaining URL path segments.

convert(*value: Iterable[str]*) → str

Convert a URI template field value to another format or type.

Parameters

value (*str or list[str]*) – Original string to convert. If `CONSUME_MULTIPLE_SEGMENTS=True` this value is a list of strings containing the path segments matched by the converter.

Returns

Converted field value, or `None` if the field
can not be converted.

Return type
object

Custom Converters

Custom converters can be registered via the *converters* router option. A converter is simply a class that implements the `BaseConverter` interface:

class falcon.routing.**BaseConverter**

Abstract base class for URI template field converters.

CONSUME_MULTIPLE_SEGMENTS: ClassVar[bool] = False

When set to `True` it indicates that this converter will consume multiple URL path segments. Currently a converter with `CONSUME_MULTIPLE_SEGMENTS=True` must be at the end of the URL template effectively meaning that it will consume all of the remaining URL path segments.

abstractmethod convert(*value: str*) → Any

Convert a URI template field value to another format or type.

Parameters

value (*str or list[str]*) – Original string to convert. If `CONSUME_MULTIPLE_SEGMENTS=True` this value is a list of strings containing the path segments matched by the converter.

Returns

Converted field value, or `None` if the field
can not be converted.

Return type
object

Custom Routers

A custom routing engine may be specified when instantiating *falcon.App()* or *falcon.asgi.App()*. For example:

```
router = MyRouter()
app = App(router=router)
```

Custom routers may derive from the default *CompiledRouter* engine, or implement a completely different routing strategy (such as object-based routing).

A custom router is any class that implements the following interface:

```
class MyRouter:
    def add_route(self, uri_template, resource, **kwargs):
        """Adds a route between URI path template and resource.

        Args:
            uri_template (str): A URI template to use for the route
            resource (object): The resource instance to associate with
                the URI template.

        Keyword Args:
            suffix (str): Optional responder name suffix for this
                route. If a suffix is provided, Falcon will map GET
                requests to ``on_get_{suffix}()``, POST requests to
                ``on_post_{suffix}()``, etc. In this way, multiple
                closely-related routes can be mapped to the same
                resource. For example, a single resource class can
                use suffixed responders to distinguish requests for
                a single item vs. a collection of those same items.
                Another class might use a suffixed responder to handle
                a shortlink route in addition to the regular route for
                the resource.

            **kwargs (dict): Accepts any additional keyword arguments
                that were originally passed to the falcon.App.add_route()
                method. These arguments MUST be accepted via the
                double-star variadic pattern (**kwargs), and ignore any
                unrecognized or unsupported arguments.
        """

    def find(self, uri, req=None):
        """Search for a route that matches the given partial URI.

        Args:
            uri(str): The requested path to route.

        Keyword Args:
             req(Request): The Request object that will be passed to
                the routed responder. The router may use `req` to
                further differentiate the requested route. For
                example, a header may be used to determine the
                desired API version and route the request
                accordingly.

                Note:
                    The `req` keyword argument was added in version
                    1.2. To ensure backwards-compatibility, routers
```

(continues on next page)

(continued from previous page)

```
                    that do not implement this argument are still
                    supported.

        Returns:
            tuple: A 4-member tuple composed of (resource, method_map,
                params, uri_template), or ``None`` if no route matches
                the requested path.

        """
```

Suffixed Responders

While Falcon encourages the REST architectural style, it is flexible enough to accommodate other paradigms. Consider the task of building an API for a calculator which can both add and subtract two numbers. You could implement the following:

```
class Add():
    def on_get(self, req, resp):
        resp.text = str(req.get_param_as_int('x') + req.get_param_as_int('y'))
        resp.status = falcon.HTTP_200


class Subtract():
    def on_get(self, req, resp):
        resp.text = str(req.get_param_as_int('x') - req.get_param_as_int('y'))
        resp.status = falcon.HTTP_200


add = Add()
subtract = Subtract()
app = falcon.App()
app.add_route('/add', add)
app.add_route('/subtract', subtract)
```

However, this approach highlights a situation in which grouping by resource may not make sense for your domain. In this context, adding and subtracting don't seem to conceptually map to two separate resource collections. Instead of separating them based on the idea of "getting" different resources from each, we might want to group them based on the attributes of their function (i.e., take two numbers, do something to them, return the result).

With Suffixed Responders, we can do just that, rewriting the example above in a more procedural style:

```
class Calculator():
    def on_get_add(self, req, resp):
        resp.text = str(req.get_param_as_int('x') + req.get_param_as_int('y'))
        resp.status = falcon.HTTP_200

    def on_get_subtract(self, req, resp):
        resp.text = str(req.get_param_as_int('x') - req.get_param_as_int('y'))
        resp.status = falcon.HTTP_200


calc = Calculator()
app = falcon.App()
app.add_route('/add', calc, suffix='add')
app.add_route('/subtract', calc, suffix='subtract')
```

In the second iteration, using Suffixed Responders, we're able to group responders based on their actions rather than the data they represent. This gives us added flexibility to accommodate situations in which a purely RESTful approach

simply doesn't fit.

Default Router

class falcon.routing.**CompiledRouter**

Fast URI router which compiles its routing logic to Python code.

Generally you do not need to use this router class directly, as an instance is created by default when the falcon.App class is initialized.

The router treats URI paths as a tree of URI segments and searches by checking the URI one segment at a time. Instead of interpreting the route tree for each look-up, it generates inlined, bespoke Python code to perform the search, then compiles that code. This makes the route processing quite fast.

The compilation process is delayed until the first use of the router (on the first routed request) to reduce the time it takes to start the application. This may noticeably delay the first response of the application when a large number of routes have been added. When adding the last route to the application a *compile* flag may be provided to force the router to compile immediately, thus avoiding any delay for the first response.

> **Note**
>
> When using a multi-threaded web server to host the application, it is possible that multiple requests may be routed at the same time upon startup. Therefore, the framework employs a lock to ensure that only a single compilation of the decision tree is performed.

See also *CompiledRouter.add_route()*

add_route(*uri_template: str*, *resource: object*, ***kwargs: Any*) → None

Add a route between a URI path template and a resource.

This method may be overridden to customize how a route is added.

Parameters

- **uri_template** (*str*) – A URI template to use for the route
- **resource** (*object*) – The resource instance to associate with the URI template.

Keyword Arguments

- **suffix** (*str*) – Optional responder name suffix for this route. If a suffix is provided, Falcon will map GET requests to `on_get_{suffix}()`, POST requests to `on_post_{suffix}()`, etc. In this way, multiple closely-related routes can be mapped to the same resource. For example, a single resource class can use suffixed responders to distinguish requests for a single item vs. a collection of those same items. Another class might use a suffixed responder to handle a shortlink route in addition to the regular route for the resource.
- **compile** (*bool*) – Optional flag that can be used to compile the routing logic on this call. By default, *CompiledRouter* delays compilation until the first request is routed. This may introduce a noticeable amount of latency when handling the first request, especially when the application implements a large number of routes. Setting *compile* to `True` when the last route is added ensures that the first request will not be delayed in this case (defaults to `False`).

  > **Note**
  >
  > Always setting this flag to `True` may slow down the addition of new routes when hundreds of them are added at once. It is advisable to only set this flag to `True` when adding the final route.

find(*uri: str*, *req: 'Request' | None = None*) → tuple[object, MethodDict, dict[str, Any], str | None] | None

Search for a route that matches the given partial URI.

Parameters

uri (*str*) – The requested path to route.

Keyword Arguments

req – The *falcon.Request* or *falcon.asgi.Request* object that will be passed to the routed responder. Currently the value of this argument is ignored by *CompiledRouter*. Routing is based solely on the path.

Returns

A 4-member tuple composed of (resource, method_map, params, uri_template), or `None` if no route matches the requested path.

Return type

tuple

map_http_methods(*resource: object*, ***kwargs: Any*) → dict[str, ResponderCallable] | dict[str, AsgiResponderCallable | AsgiResponderWsCallable]

Map HTTP methods (e.g., GET, POST) to methods of a resource object.

This method is called from *add_route()* and may be overridden to provide a custom mapping strategy.

Parameters

resource (*instance*) – Object which represents a REST resource. The default maps the HTTP method `GET` to `on_get()`, `POST` to `on_post()`, etc. If any HTTP methods are not supported by your resource, simply don't define the corresponding request handlers, and Falcon will do the right thing.

Keyword Arguments

suffix (*str*) – Optional responder name suffix for this route. If a suffix is provided, Falcon will map GET requests to `on_get_{suffix}()`, POST requests to `on_post_{suffix}()`, etc. In this way, multiple closely-related routes can be mapped to the same resource. For example, a single resource class can use suffixed responders to distinguish requests for a single item vs. a collection of those same items. Another class might use a suffixed responder to handle a shortlink route in addition to the regular route for the resource.

Routing Utilities

The *falcon.routing* module contains the following utilities that may be used by custom routing engines.

falcon.routing.**map_http_methods**(*resource: object*, *suffix: str | None = None*) → MethodDict

Map HTTP methods (e.g., GET, POST) to methods of a resource object.

Parameters

resource – An object with *responder* methods, following the naming convention *on_**, that correspond to each method the resource supports. For example, if a resource supports GET and POST, it should define `on_get(self, req, resp)` and `on_post(self, req, resp)`.

Keyword Arguments

suffix (*str*) – Optional responder name suffix for this route. If a suffix is provided, Falcon will map GET requests to `on_get_{suffix}()`, POST requests to `on_post_{suffix}()`, etc.

Returns

A mapping of HTTP methods to explicitly defined resource responders.

Return type

dict

falcon.routing.**set_default_responders**(*method_map: MethodDict*, *asgi: bool = False*) → None

Map HTTP methods not explicitly defined on a resource to default responders.

Parameters

- **method_map** – A dict with HTTP methods mapped to responders explicitly defined in a resource.
- **asgi** (*bool*) – `True` if using an ASGI app, `False` otherwise (default `False`).

falcon.app_helpers.**prepare_middleware**(*middleware: Iterable[SyncMiddleware[_ReqT, independent_middleware: bool = False, asgi: bool = False*) → PreparedMiddlewareResult | AsyncPreparedMiddlewareResult

Check middleware interfaces and prepare the methods for request handling.

> **Note**
>
> This method is only applicable to WSGI apps.

Parameters

middleware (*iterable*) – An iterable of middleware objects.

Keyword Arguments

- **independent_middleware** (*bool*) – `True` if the request and response middleware methods should be treated independently (default `False`)
- **asgi** (*bool*) – `True` if an ASGI app, `False` otherwise (default `False`)

Returns

A tuple of prepared middleware method tuples

Return type

tuple

falcon.app_helpers.**prepare_middleware_ws**(*middleware: Iterable[AsyncMiddleware]*) → AsyncPreparedMiddlewareWsResult

Check middleware interfaces and prepare WebSocket methods for request handling.

> **Note**
>
> This method is only applicable to ASGI apps.

Parameters

middleware (*iterable*) – An iterable of middleware objects.

Returns

A two-item `(request_mw, resource_mw)` tuple, where *request_mw* is an ordered list of `process_request_ws()` methods, and *resource_mw* is an ordered list of `process_resource_ws()` methods.

Return type

tuple

Static File Routes

Falcon can serve static files directly from a WSGI or ASGI application using the below sink-like *StaticRoute*.

Instances of *StaticRoute* are normally created via *falcon.App.add_static_route()* (please see, however, the documentation of *add_static_route()* for the performance implications of serving files through a Python app).

class falcon.routing.**StaticRoute**(*prefix: str*, *directory: str | Path*, *downloadable: bool = False*, *fallback_filename: str | None = None*)

Represents a static route.

Parameters

- **prefix** (*str*) – The path prefix to match for this route. If the path in the requested URI starts with this string, the remainder of the path will be appended to the source directory to determine the file to serve. This is done in a secure manner to prevent an attacker from requesting a file outside the specified directory.

  Note that static routes are matched in LIFO order, and are only attempted after checking dynamic routes and sinks.

- **directory** (*Union[str, pathlib.Path]*) – The source directory from which to serve files. Must be an absolute path.

- **downloadable** (*bool*) – Set to `True` to include a Content-Disposition header in the response. The "filename" directive is simply set to the name of the requested file.

- **fallback_filename** (*str*) – Fallback filename used when the requested file is not found. Can be a relative path inside the prefix folder or any valid absolute path.

  Note

  If the fallback file is served instead of the requested file, the response Content-Type header, as well as the Content-Disposition header (provided it was requested with the *downloadable* parameter described above), are derived from the fallback filename, as opposed to the requested filename.

match(*path: str*) → bool

Check whether the given path matches this route.

Custom HTTP Methods

While not normally advised, some applications may need to support non-standard HTTP methods, in addition to the standard HTTP methods like GET and PUT. To support custom HTTP methods, use one of the following methods:

- Ideally, if you don't use hooks in your application, you can easily add the custom methods in your application setup by overriding the value of `falcon.constants.COMBINED_METHODS`. For example:

```
import falcon.constants
falcon.constants.COMBINED_METHODS += ['FOO', 'BAR']
```

- Due to the nature of hooks, if you do use them, you'll need to define the FALCON_CUSTOM_HTTP_METHODS environment variable as a comma-delimited list of custom methods. For example:

```
$ export FALCON_CUSTOM_HTTP_METHODS=FOO,BAR
```

Once you have used the appropriate method, your custom methods should be active. You then can define request methods like any other HTTP method:

WSGI

```
# Handle the custom FOO method
def on_foo(self, req, resp):
    pass
```

ASGI

```
# Handle the custom FOO method
async def on_foo(self, req, resp):
    pass
```

5.3.14 Inspect Module

This module can be used to inspect a Falcon application to obtain information about its registered routes, middleware objects, static routes, sinks and error handlers. The entire application can be inspected at once using the *inspect_app()* function. Additional functions are available for inspecting specific aspects of the app.

A `falcon-inspect-app` CLI script is also available; it uses the inspect module to print a string representation of an application, as demonstrated below:

```
# my_module exposes the application as a variable named "app"
$ falcon-inspect-app my_module:app

Falcon App (WSGI)
• Routes:
    ⇒ /foo - MyResponder:
       ├── DELETE - on_delete
       ├── GET - on_get
       └── POST - on_post
    ⇒ /foo/{id} - MyResponder:
       ├── DELETE - on_delete_id
       ├── GET - on_get_id
       └── POST - on_post_id
    ⇒ /bar - OtherResponder:
       ├── DELETE - on_delete_id
       ├── GET - on_get_id
       └── POST - on_post_id
• Middleware (Middleware are independent):
    → MyMiddleware.process_request
      → OtherMiddleware.process_request

        ↣ MyMiddleware.process_resource
          ↣ OtherMiddleware.process_resource

              ├── Process route responder

          ↢ OtherMiddleware.process_response
        ↢ CORSMiddleware.process_response
• Static routes:
    ↦ /tests/ /path/to/tests [/path/to/test/index.html]
    ↦ /falcon/ /path/to/falcon
• Sinks:
    ⇥ /sink_cls SinkClass
    ⇥ /sink_fn sinkFn
• Error handlers:
    ↜ RuntimeError my_runtime_handler
```

The example above shows how `falcon-inspect-app` simply outputs the value returned by the *AppInfo.to_string()* method. In fact, here is a simple script that returns the same output as the `falcon-inspect-app` command:

```
from falcon import inspect
from my_module import app
```

(continues on next page)

(continued from previous page)

```
app_info = inspect.inspect_app(app)

# Equivalent to print(app_info.to_string())
print(app_info)
```

A more verbose description of the app can be obtained by passing `verbose=True` to *`AppInfo.to_string()`*, while the default routes added by the framework can be included by passing `internal=True`. The `falcon-inspect-app` command supports the `--verbose` and `--internal` flags to enable these options.

Using Inspect Functions

The values returned by the inspect functions are class instances that contain the relevant information collected from the application. These objects facilitate programmatic use of the collected data.

To support inspection of applications that use a custom router, the module provides a *`register_router()`* function to register a handler function for the custom router class. Inspection of the default *`CompiledRouter`* class is handled by the *`inspect_compiled_router()`* function.

The returned information classes can be explored using the visitor pattern. To create the string representation of the classes the *`StringVisitor`* visitor is used. This class is instantiated automatically when calling `str()` on an instance or when using the `to_string()` method.

Custom visitor implementations can subclass *`InspectVisitor`* and use the *`InspectVisitor.process()`* method to visit the classes.

Inspect Functions Reference

This module defines the following inspect functions.

`falcon.inspect.`**`inspect_app`**(*app:* App) → *AppInfo*

Inspects an application.

Parameters
app (`falcon.App`) – The application to inspect. Works with both *`falcon.App`* and *`falcon.asgi.App`*.

Returns
The information regarding the application. Call *`to_string()`* on the result to obtain a human-friendly representation.

Return type
AppInfo

`falcon.inspect.`**`inspect_routes`**(*app:* App) → list[*RouteInfo*]

Inspects the routes of an application.

Parameters
app (`falcon.App`) – The application to inspect. Works with both *`falcon.App`* and *`falcon.asgi.App`*.

Returns
A list of route descriptions for the application.

Return type
list[*RouteInfo*]

`falcon.inspect.`**`inspect_middleware`**(*app:* App) → *MiddlewareInfo*

Inspects the middleware components of an application.

Parameters
app (`falcon.App`) – The application to inspect. Works with both *`falcon.App`* and *`falcon.asgi.App`*.

Returns
Information about the app's middleware components.

Return type
MiddlewareInfo

falcon.inspect.**inspect_static_routes**(*app:* App) → list[*StaticRouteInfo*]

Inspects the static routes of an application.

Parameters
app (falcon.App) – The application to inspect. Works with both *falcon.App* and *falcon.asgi.App*.

Returns
A list of static routes that have been added to the application.

Return type
list[*StaticRouteInfo*]

falcon.inspect.**inspect_sinks**(*app:* App) → list[*SinkInfo*]

Inspects the sinks of an application.

Parameters
app (falcon.App) – The application to inspect. Works with both *falcon.App* and *falcon.asgi.App*.

Returns
A list of sinks used by the application.

Return type
list[*SinkInfo*]

falcon.inspect.**inspect_error_handlers**(*app:* App) → list[*ErrorHandlerInfo*]

Inspects the error handlers of an application.

Parameters
app (falcon.App) – The application to inspect. Works with both *falcon.App* and *falcon.asgi.App*.

Returns
A list of error handlers used by the application.

Return type
list[*ErrorHandlerInfo*]

Router Inspection

The following functions enable route inspection.

falcon.inspect.**register_router**(*router_class: type*) → Callable[[...], Callable[[...], list[*RouteInfo*]]]

Register a function to inspect a particular router.

This decorator registers a new function for a custom router class, so that it can be inspected with the function *inspect_routes()*. An inspection function takes the router instance used by the application and returns a list of *RouteInfo*. Eg:

```
@register_router(MyRouterClass)
def inspect_my_router(router):
    return [RouteInfo('foo', 'bar', '/path/to/foo.py:42', [])]
```

Parameters
router_class (*Type*) – The router class to register. If already registered an error will be raised.

falcon.inspect.**inspect_compiled_router**(*router:* CompiledRouter) → list[*RouteInfo*]

Walk an instance of *CompiledRouter* to return a list of defined routes.

Default route inspector for CompiledRouter.

Parameters
router (CompiledRouter) – The router to inspect.

Returns
A list of *RouteInfo*.

Return type
list[*RouteInfo*]

Information Classes

Information returned by the inspect functions is represented by these classes.

class falcon.inspect.**AppInfo**(*routes: list[*RouteInfo*]*, *middleware:* MiddlewareInfo, *static_routes: list[*StaticRouteInfo*]*, *sinks: list[*SinkInfo*]*, *error_handlers: list[*ErrorHandlerInfo*]*, *asgi: bool*)

Describes an application.

Parameters

- **routes** (*list[RouteInfo]*) – The routes of the application.
- **middleware** (MiddlewareInfo) – The middleware information in the application.
- **static_routes** (*list[StaticRouteInfo]*) – The static routes of this application.
- **sinks** (*list[SinkInfo]*) – The sinks of this application.
- **error_handlers** (*list[ErrorHandlerInfo]*) – The error handlers of this application.
- **asgi** (*bool*) – Whether or not this is an ASGI application.

to_string(*verbose: bool = False*, *internal: bool = False*, *name: str = ''*) → str

Return a string representation of this class.

Parameters

- **verbose** (*bool, optional*) – Adds more information. Defaults to False.
- **internal** (*bool, optional*) – Also include internal falcon route methods and error handlers. Defaults to `False`.
- **name** (*str, optional*) – The name of the application, to be output at the beginning of the text. Defaults to `'Falcon App'`.

Returns
A string representation of the application.

Return type
str

class falcon.inspect.**RouteInfo**(*path: str*, *class_name: str*, *source_info: str*, *methods: list[*RouteMethodInfo*]*)

Describes a route.

Parameters

- **path** (*str*) – The path of this route.
- **class_name** (*str*) – The class name of the responder of this route.
- **source_info** (*str*) – The source path where this responder was defined.
- **methods** (*list[RouteMethodInfo]*) – List of methods defined in the route.

class falcon.inspect.**RouteMethodInfo**(*method: str*, *source_info: str*, *function_name: str*, *internal: bool*)

Describes a responder method.

Parameters

- **method** (*str*) – The HTTP method of this responder.
- **source_info** (*str*) – The source path of this function.
- **function_name** (*str*) – Name of the function.
- **internal** (*bool*) – Whether or not this was a default responder added by the framework.

class falcon.inspect.**MiddlewareInfo**(*middleware_tree: MiddlewareTreeInfo*, *middleware_classes: list[MiddlewareClassInfo]*, *independent: bool*)

Describes the middleware of the app.

Parameters

- **middlewareTree** (*MiddlewareTreeInfo*) – The middleware tree of the app.
- **middlewareClasses** (*list[MiddlewareClassInfo]*) – The middleware classes of the app.
- **independent** (*bool*) – Whether or not the middleware components are executed independently.

class falcon.inspect.**MiddlewareTreeInfo**(*request: list[MiddlewareTreeItemInfo]*, *resource: list[MiddlewareTreeItemInfo]*, *response: list[MiddlewareTreeItemInfo]*)

Describes the middleware methods used by the app.

Parameters

- **request** (*list[MiddlewareTreeItemInfo]*) – The *process_request* methods.
- **resource** (*list[MiddlewareTreeItemInfo]*) – The *process_resource* methods.
- **response** (*list[MiddlewareTreeItemInfo]*) – The *process_response* methods.

class falcon.inspect.**MiddlewareClassInfo**(*name: str*, *source_info: str*, *methods: list[MiddlewareMethodInfo]*)

Describes a middleware class.

Parameters

- **name** (*str*) – The name of the middleware class.
- **source_info** (*str*) – The source path where the middleware was defined.
- **methods** (*list[MiddlewareMethodInfo]*) – List of method defined by the middleware class.

class falcon.inspect.**MiddlewareTreeItemInfo**(*name: str*, *class_name: str*)

Describes a middleware tree entry.

Parameters

- **name** (*str*) – The name of the method.
- **class_name** (*str*) – The class name of the method.

class falcon.inspect.**MiddlewareMethodInfo**(*function_name: str*, *source_info: str*)

Describes a middleware method.

Parameters

- **function_name** (*str*) – Name of the method.
- **source_info** (*str*) – The source path of the method.

class falcon.inspect.**StaticRouteInfo**(*prefix: str*, *directory: str*, *fallback_filename: str | None*)

Describes a static route.

Parameters

- **path** (*str*) – The prefix of the static route.
- **directory** (*str*) – The directory for the static route.
- **fallback_filename** (*str or None*) – Fallback filename to serve.

class falcon.inspect.**SinkInfo**(*prefix: str*, *name: str*, *source_info: str*)

Describes a sink.

Parameters

- **prefix** (*str*) – The prefix of the sink.
- **name** (*str*) – The name of the sink function or class.
- **source_info** (*str*) – The source path where this sink was defined.

class falcon.inspect.**ErrorHandlerInfo**(*error: str*, *name: str*, *source_info: str*, *internal: bool*)

Describes an error handler.

Parameters

- **error** (*name*) – The name of the error type.
- **name** (*str*) – The name of the handler.
- **source_info** (*str*) – The source path where this error handler was defined.
- **internal** (*bool*) – Whether or not this is a default error handler added by the framework.

Visitor Classes

The following visitors are used to traverse the information classes.

class falcon.inspect.**InspectVisitor**

Base visitor class that implements the *process* method.

Subclasses must implement `visit_<name>` methods for each supported class.

process(*instance: _Traversable*) → str

Process the instance, by calling the appropriate visit method.

Uses the *__visit_name__* attribute of the *instance* to obtain the method to use.

Parameters

instance (*_Traversable*) – The instance to process.

class falcon.inspect.**StringVisitor**(*verbose: bool = False*, *internal: bool = False*, *name: str = ''*)

Visitor that returns a string representation of the info class.

This is used automatically by calling `to_string()` on the info class. It can also be used directly by calling `StringVisitor.process(info_instance)`.

Parameters

- **verbose** (*bool, optional*) – Adds more information. Defaults to `False`.
- **internal** (*bool, optional*) – Also include internal route methods and error handlers added by the framework. Defaults to `False`.
- **name** (*str, optional*) – The name of the application, to be output at the beginning of the text. Defaults to `'Falcon App'`.

5.3.15 Utilities

This page describes miscellaneous utilities provided by Falcon.

URI

URI utilities.

This module provides utility functions to parse, encode, decode, and otherwise manipulate a URI. These functions are not available directly in the *falcon* module, and so must be explicitly imported:

```
from falcon import uri

name, port = uri.parse_host('example.org:8080')
```

falcon.uri.**decode**(*encoded_uri: str*, *unquote_plus: bool = True*) → str

Decode percent-encoded characters in a URI or query string.

This function models the behavior of *urllib.parse.unquote_plus*, albeit in a faster, more straightforward manner.

Parameters

encoded_uri (*str*) – An encoded URI (full or partial).

Keyword Arguments

unquote_plus (*bool*) – Set to `False` to retain any plus ('+') characters in the given string, rather than converting them to spaces (default `True`). Typically you should set this to `False` when decoding any part of a URI other than the query string.

Returns

A decoded URL. If the URL contains escaped non-ASCII characters, UTF-8 is assumed per RFC 3986.

Return type

str

falcon.uri.**encode**(*uri: str*) → str

Encodes a full or relative URI according to RFC 3986.

RFC 3986 defines a set of "unreserved" characters as well as a set of "reserved" characters used as delimiters. This function escapes all other "disallowed" characters by percent-encoding them.

> **Note**
>
> This utility is faster in the average case than the similar *quote* function found in `urlib`. It also strives to be easier to use by assuming a sensible default of allowed characters.

Parameters

uri (*str*) – URI or part of a URI to encode.

Returns

An escaped version of *uri*, where all disallowed characters have been percent-encoded.

Return type

str

falcon.uri.**encode_check_escaped**(*uri: str*) → str

Encodes a full or relative URI according to RFC 3986.

RFC 3986 defines a set of "unreserved" characters as well as a set of "reserved" characters used as delimiters. This function escapes all other "disallowed" characters by percent-encoding them unless they appear to have been previously encoded. For example, `'%26'` will not be encoded again as it follows the format of an encoded value.

> **Note**
>
> This utility is faster in the average case than the similar *quote* function found in `urllib`. It also strives to be easier to use by assuming a sensible default of allowed characters.

Parameters
uri (*str*) – URI or part of a URI to encode.

Returns
An escaped version of *uri*, where all disallowed characters have been percent-encoded.

Return type
str

`falcon.uri.`**`encode_value`**(*uri: str*) → str

Encodes a value string according to RFC 3986.

Disallowed characters are percent-encoded in a way that models `urllib.parse.quote(safe="~")`. However, the Falcon function is faster in the average case than the similar *quote* function found in urlib. It also strives to be easier to use by assuming a sensible default of allowed characters.

All reserved characters are lumped together into a single set of "delimiters", and everything in that set is escaped.

> **Note**
>
> RFC 3986 defines a set of "unreserved" characters as well as a set of "reserved" characters used as delimiters.

Parameters
uri (*str*) – URI fragment to encode. It is assumed not to cross delimiter boundaries, and so any reserved URI delimiter characters included in it will be percent-encoded.

Returns
An escaped version of *uri*, where all disallowed characters have been percent-encoded.

Return type
str

`falcon.uri.`**`encode_value_check_escaped`**(*uri: str*) → str

Encodes a value string according to RFC 3986.

RFC 3986 defines a set of "unreserved" characters as well as a set of "reserved" characters used as delimiters. Disallowed characters are percent-encoded in a way that models `urllib.parse.quote(safe="~")` unless they appear to have been previously encoded. For example, `'%26'` will not be encoded again as it follows the format of an encoded value.

All reserved characters are lumped together into a single set of "delimiters", and everything in that set is escaped.

> **Note**
>
> This utility is faster in the average case than the similar *quote* function found in `urllib`. It also strives to be easier to use by assuming a sensible default of allowed characters.

Parameters
uri (*str*) – URI fragment to encode. It is assumed not to cross delimiter boundaries, and so any reserved URI delimiter characters included in it will be percent-encoded.

Returns
An escaped version of *uri*, where all disallowed characters have been percent-encoded.

Return type
str

falcon.uri.**parse_host**(*host: str*, *default_port: int | None = None*) → tuple[str, int | None]

Parse a canonical 'host:port' string into parts.

Parse a host string (which may or may not contain a port) into parts, taking into account that the string may contain either a domain name or an IP address. In the latter case, both IPv4 and IPv6 addresses are supported.

Parameters
host (*str*) – Host string to parse, optionally containing a port number.

Keyword Arguments
default_port (*int*) – Port number to return when the host string does not contain one (default `None`).

Returns
A parsed (*host*, *port*) tuple from the given host string, with the port converted to an `int`. If the host string does not specify a port, *default_port* is used instead.

Return type
tuple

falcon.uri.**parse_query_string**(*query_string: str*, *keep_blank: bool = False*, *csv: bool = False*) → dict[str, str | list[str]]

Parse a query string into a dict.

Query string parameters are assumed to use standard form-encoding. Only parameters with values are returned. For example, given 'foo=bar&flag', this function would ignore 'flag' unless the *keep_blank* option is set.

> **Note**
>
> In addition to the standard HTML form-based method for specifying lists by repeating a given param multiple times, Falcon supports a more compact form in which the param may be given a single time but set to a `list` of comma-separated elements (e.g., 'foo=a,b,c'). This comma-separated format can be enabled by setting the *csv* option (see below) to `True`.
>
> When using this format, all commas uri-encoded will not be treated by Falcon as a delimiter. If the client wants to send a value as a list, it must not encode the commas with the values.
>
> The two different ways of specifying lists may not be mixed in a single query string for the same parameter.

Parameters

- **query_string** (*str*) – The query string to parse.
- **keep_blank** (*bool*) – Set to `True` to return fields even if they do not have a value (default `False`). For comma-separated values, this option also determines whether or not empty elements in the parsed list are retained.
- **csv** – Set to `True` in order to enable splitting query parameters on `,` (default `False`). Depending on the user agent, encoding lists as multiple occurrences of the same parameter might be preferable. In this case, keeping *parse_qs_csv* at its default value (`False`) will cause the framework to treat commas as literal characters in each occurring parameter value.

Returns
A dictionary of (*name*, *value*) pairs, one per query parameter. Note that *value* may be a single `str`, or a `list` of `str`.

Return type
dict

Raises
TypeError – *query_string* was not a `str`.

falcon.uri.**unquote_string**(*quoted: str*) → str

Unquote an RFC 7320 "quoted-string".

Parameters
quoted (*str*) – Original quoted string

Returns
unquoted string

Return type
str

Raises
TypeError – *quoted* was not a `str`.

Date and Time

falcon.**http_now**() → str

Return the current UTC time as an IMF-fixdate.

Returns
The current UTC time as an IMF-fixdate, e.g., 'Tue, 15 Nov 1994 12:45:26 GMT'.

Return type
str

falcon.**dt_to_http**(*dt: datetime*) → str

Convert a `datetime` instance to an HTTP date string.

Parameters
dt (*datetime*) – A `datetime` instance to convert, assumed to be UTC.

Returns
An RFC 1123 date string, e.g.: "Tue, 15 Nov 1994 12:45:26 GMT".

Return type
str

falcon.**http_date_to_dt**(*http_date: str*, *obs_date: bool = False*) → datetime

Convert an HTTP date string to a datetime instance.

Parameters
http_date (*str*) – An RFC 1123 date string, e.g.: "Tue, 15 Nov 1994 12:45:26 GMT".

Keyword Arguments
obs_date (*bool*) – Support obs-date formats according to RFC 7231, e.g.: "Sunday, 06-Nov-94 08:49:37 GMT" (default `False`).

Returns
A UTC datetime instance corresponding to the given HTTP date.

Return type
datetime

Raises

- **ValueError** – http_date doesn't match any of the available time formats
- **ValueError** – http_date doesn't match allowed timezones

Changed in version 4.0: This function now returns timezone-aware `datetime` objects.

class falcon.**TimezoneGMT**

GMT timezone class implementing the `datetime.tzinfo` interface.

Deprecated since version 4.0: *`TimezoneGMT`* is deprecated, use `datetime.timezone.utc` instead. (This class will be removed in Falcon 5.0.)

dst(*dt: datetime | None*) → timedelta

Return the daylight saving time (DST) adjustment.

Parameters
dt (*datetime.datetime*) – Ignored

Returns
DST adjustment for GMT, which is always 0.

Return type
datetime.timedelta

tzname(*dt: datetime | None*) → str

Get the name of this timezone.

Parameters
dt (*datetime.datetime*) – Ignored

Returns
"GMT"

Return type
str

utcoffset(*dt: datetime | None*) → timedelta

Get the offset from UTC.

Parameters
dt (*datetime.datetime*) – Ignored

Returns
GMT offset, which is equivalent to UTC and so is always 0.

Return type
datetime.timedelta

HTTP Status

falcon.**http_status_to_code**(*status: HTTPStatus | int | bytes | str*) → int

Normalize an HTTP status to an integer code.

This function takes a member of `http.HTTPStatus`, an HTTP status line string or byte string (e.g., `'200 OK'`), or an `int` and returns the corresponding integer code.

An LRU is used to minimize lookup time.

Parameters
status – The status code or enum to normalize.

Returns
Integer code for the HTTP status (e.g., 200)

Return type
int

falcon.**code_to_http_status**(*status: int | HTTPStatus | bytes | str*) → str

Normalize an HTTP status to an HTTP status line string.

This function takes a member of `http.HTTPStatus`, an `int` status code, an HTTP status line string or byte string (e.g., `'200 OK'`) and returns the corresponding HTTP status line string.

An LRU is used to minimize lookup time.

> **Note**
>
> This function will not attempt to coerce a string status to an integer code, assuming the string already denotes an HTTP status line.

Parameters
status – The status code or enum to normalize.

Returns

HTTP status line corresponding to the given code. A newline
is not included at the end of the string.

Return type
str

Media types

falcon.**parse_header**(*line: str*) → tuple[str, dict[str, str]]

Parse a Content-type like header.

Return the main content-type and a dictionary of options.

Parameters
line – A header value to parse.

Returns
(the main content-type, dictionary of options).

Return type
tuple

> **Note**
>
> This function replaces an equivalent method previously available in the stdlib as `cgi.parse_header()`. It was removed from the stdlib in Python 3.13.

Added in version 4.0.

falcon.mediatypes.**quality**(*media_type: str*, *header: str*) → float

Get quality of the most specific matching media range.

Media-ranges are parsed from the provided *header* value according to RFC 9110, Section 12.5.1 (the `Accept` header).

The provided *media_type* is matched against each of the parsed media ranges, and the fitness of each match is assessed as follows (in the decreasing priority list of criteria):

1. Do the main types (as in `type/subtype`) match?

   The types must either match exactly, or as wildcard (`*`). The matches involving a wildcard are prioritized lower.

2. Do the subtypes (as in `type/subtype`) match?

   The subtypes must either match exactly, or as wildcard (`*`). The matches involving a wildcard are prioritized lower.

3. Do the parameters match exactly?

   If all the parameter names and values (if any) between the media range and media type match exactly, such match is prioritized higher than matches involving extraneous parameters on either side.

Note that if parameter names match, the corresponding values must also be equal, or the provided media type is considered not to match the media range in question at all.

4. The number of matching parameters.
5. Finally, if two or more best media range matches are equally fit according to all of the above criteria (1) through (4), the highest quality (i.e., the value of the `q` parameter) of these is returned.

> **Note**
>
> With the exception of evaluating the exact parameter match (3), the number of extraneous parameters (i.e. where the names are only present in the media type, or only in the media range) currently does not influence the described specificity sort order.

Parameters

- **media_type** – The Internet media type to match against the provided HTTP `Accept` header value.
- **header** – The value of a header that conforms to the format of the HTTP `Accept` header.

Returns

Quality of the most specific media range matching the provided *media_type*. (If none matches, 0.0 is returned.)

Added in version 4.0.

`falcon.mediatypes.`**`best_match`**(*media_types: Iterable[str]*, *header: str*) → str

Choose media type with the highest *`quality()`* from a list of candidates.

Parameters

- **media_types** – An iterable over one or more Internet media types to match against the provided header value.
- **header** – The value of a header that conforms to the format of the HTTP `Accept` header.

Returns

Best match from the supported candidates, or an empty string if the provided header value does not match any of the given types.

Added in version 4.0.

Async

Aliases

Falcon used to provide aliases for the below functions implemented in `asyncio`, with fallbacks for older versions of Python:

- `falcon.get_running_loop()` → `asyncio.get_running_loop()`
- `falcon.create_task(coro, *, name=None)` → `asyncio.create_task()`

However, as of Falcon 4.0+, these aliases are identical to their `asyncio` counterparts on all supported Python versions. (They are only kept for compatibility purposes.)

Simply use `asyncio.get_running_loop()` or `asyncio.create_task()` directly in new code.

Adapters

These functions help traverse the barrier between sync and async code.

async falcon.**sync_to_async**(*func: Callable[[...], Any]*, **args: Any*, ***kwargs: Any*) → Callable[[...], Awaitable[Any]]

Schedule a synchronous callable on the default executor and await the result.

This helper makes it easier to call functions that can not be ported to use async natively (e.g., functions exported by a database library that does not yet support asyncio).

To execute blocking operations safely, without stalling the async loop, the wrapped callable is scheduled to run in the background, on a separate thread, when the wrapper is called.

The default executor for the running loop is used to schedule the synchronous callable.

Warning

This helper can only be used to execute thread-safe callables. If the callable is not thread-safe, it can be executed serially by first wrapping it with *wrap_sync_to_async()*, and then executing the wrapper directly.

Warning

Calling a synchronous function safely from an asyncio event loop adds a fair amount of overhead to the function call, and should only be used when a native async library is not available for the operation you wish to perform.

Parameters

- **func** (*callable*) – Function, method, or other callable to wrap
- ***args** – All additional arguments are passed through to the callable.

Keyword Arguments

****kwargs** – All keyword arguments are passed through to the callable.

Returns

An awaitable coroutine function that wraps the synchronous callable.

Return type

function

falcon.**wrap_sync_to_async**(*func: Callable[[...], Any]*, *threadsafe: bool | None = None*) → Callable[[...], Any]

Wrap a callable in a coroutine that executes the callable in the background.

This helper makes it easier to call functions that can not be ported to use async natively (e.g., functions exported by a database library that does not yet support asyncio).

To execute blocking operations safely, without stalling the async loop, the wrapped callable is scheduled to run in the background, on a separate thread, when the wrapper is called.

Normally, the default executor for the running loop is used to schedule the synchronous callable. If the callable is not thread-safe, it can be scheduled serially in a global single-threaded executor.

Warning

Wrapping a synchronous function safely adds a fair amount of overhead to the function call, and should only be used when a native async library is not available for the operation you wish to perform.

Parameters

func (*callable*) – Function, method, or other callable to wrap

Keyword Arguments
threadsafe (*bool*) – Set to `False` when the callable is not thread-safe (default `True`). When this argument is `False`, the wrapped callable will be scheduled to run serially in a global single-threaded executor.

Returns
An awaitable coroutine function that wraps the synchronous callable.

Return type
function

`falcon.`**`wrap_sync_to_async_unsafe`**(*func: Callable[[...], Any]*) → Callable[[...], Any]

Wrap a callable in a coroutine that executes the callable directly.

This helper makes it easier to use synchronous callables with ASGI apps. However, it is considered "unsafe" because it calls the wrapped function directly in the same thread as the asyncio loop. Generally, you should use *`wrap_sync_to_async()`* instead.

> **Warning**
>
> This helper is only to be used for functions that do not perform any blocking I/O or lengthy CPU-bound operations, since the entire async loop will be blocked while the wrapped function is executed. For a safer, non-blocking alternative that runs the function in a thread pool executor, use *`sync_to_async()`* instead.

Parameters
func (*callable*) – Function, method, or other callable to wrap

Returns
An awaitable coroutine function that wraps the synchronous callable.

Return type
function

`falcon.`**`async_to_sync`**(*coroutine: Callable[[...], Awaitable[Result]]*, **args: Any*, ***kwargs: Any*) → Result

Invoke a coroutine function from a synchronous caller.

This method can be used to invoke an asynchronous task from a synchronous context. The coroutine will be scheduled to run on the current event loop for the current OS thread. If an event loop is not already running, one will be created.

> **Warning**
>
> Executing async code in this manner is inefficient since it involves synchronization via threading primitives, and is intended primarily for testing, prototyping or compatibility purposes.

> **Note**
>
> On Python 3.11+, this function leverages a module-wide `asyncio.Runner`.

Parameters
- **coroutine** – A coroutine function to invoke.
- ***args** – Additional args are passed through to the coroutine function.

Keyword Arguments
****kwargs** – Additional args are passed through to the coroutine function.

falcon.**runs_sync**(*coroutine: Callable[[...], Awaitable[Result]]*) → Callable[[...], Result]

Transform a coroutine function into a synchronous method.

This is achieved by always invoking the decorated coroutine function via `async_to_sync()`.

> **Warning**
>
> This decorator is very inefficient and should only be used for adapting asynchronous test functions for use with synchronous test runners such as `pytest` or the `unittest` module.
>
> It will create an event loop for the current thread if one is not already running.

Parameters
coroutine – A coroutine function to masquerade as a synchronous one.

Returns
A synchronous function.

Return type
callable

Other

falcon.util.**deprecated**(*instructions: str*, *is_property: bool = False*, *method_name: str | None = None*) → Callable[[Callable[[...], Any]], Any]

Flag a method as deprecated.

This function returns a decorator which can be used to mark deprecated functions. Applying this decorator will result in a warning being emitted when the function is used.

Parameters

- **instructions** (*str*) – Specific guidance for the developer, e.g.: 'Please migrate to add_proxy(...)'.
- **is_property** (*bool*) – If the deprecated object is a property. It will omit the `(...)` from the generated documentation.
- **method_name** (*str, optional*) – Set to override the name of the deprecated function or property in the generated documentation (default `None`). This is useful when decorating an alias that carries the target's `__name__`.

falcon.util.**deprecated_args**(*, *allowed_positional: int*, *is_method: bool = True*) → Callable[[...], Callable[[...], Any]]

Flag a method call with positional args as deprecated.

Keyword Arguments

- **allowed_positional** (*int*) – Number of allowed positional arguments
- **is_method** (*bool, optional*) – The decorated function is a method. Will add one to the number of allowed positional args to account for `self`. Defaults to True.

falcon.**to_query_str**(*params: Mapping[str, Any] | None*, *comma_delimited_lists: bool = True*, *prefix: bool = True*) → str

Convert a dictionary of parameters to a query string.

Parameters

- **params** (*dict*) – A dictionary of parameters, where each key is a parameter name, and each value is either a `str` or something that can be converted into a `str`, or a list of such values. If a `list`, the value will be converted to a comma-delimited string of values (e.g., 'thing=1,2,3').

- **comma_delimited_lists** (*bool*) – Set to `False` to encode list values by specifying multiple instances of the parameter (e.g., 'thing=1&thing=2&thing=3'). Otherwise, parameters will be encoded as comma-separated values (e.g., 'thing=1,2,3'). Defaults to `True`.
- **prefix** (*bool*) – Set to `False` to exclude the '?' prefix in the result string (default `True`).

Returns
A URI query string, including the '?' prefix (unless *prefix* is `False`), or an empty string if no params are given (the `dict` is empty).

Return type
str

`falcon.`**get_bound_method**(*obj: object*, *method_name: str*) → None | Callable[[...], Any]

Get a bound method of the given object by name.

Parameters

- **obj** – Object on which to look up the method.
- **method_name** – Name of the method to retrieve.

Returns
Bound method, or `None` if the method does not exist on the object.

Raises
AttributeError – The method exists, but it isn't bound (most likely a class was passed, rather than an instance of that class).

`falcon.`**secure_filename**(*filename: str*, *max_length: int | None = None*) → str

Sanitize the provided *filename* to contain only ASCII characters.

Only ASCII alphanumerals, `'.'`, `'-'` and `'_'` are allowed for maximum portability and safety wrt using this name as a filename on a regular file system. All other characters will be replaced with an underscore (`'_'`).

Note

The *filename* is normalized to the Unicode `NKFD` form prior to ASCII conversion in order to extract more alphanumerals where a decomposition is available. For instance:

```
>>> secure_filename('Bold Digit 𝟙')
'Bold_Digit_1'
>>> secure_filename('Ångström unit physics.pdf')
'A_ngstro_m_unit_physics.pdf'
>>> secure_filename('Ångström unit physics.pdf', max_length=19)
'A_ngstro_m_unit.pdf'
```

Parameters

- **filename** (*str*) – Arbitrary filename input from the request, such as a multipart form filename field.
- **max_length** (*Optional[int]*) – Maximum allowed length of the sanitized filename. The sanitized filename is truncated while attempting to preserve its extension. If the provided name has no extension, or the extension is too long, itself, only the head is retained.

  Added in version 4.1.

Returns
The sanitized filename (truncated to *max_length* characters).

Return type
str

Raises
ValueError – the provided filename is an empty string.

falcon.**is_python_func**(*func: Callable | Any*) → bool

Determine if a function or method uses a standard Python type.

This helper can be used to check a function or method to determine if it uses a standard Python type, as opposed to an implementation-specific native extension type.

For example, because Cython functions are not standard Python functions, `is_python_func(f)` will return `False` when f is a reference to a cythonized function or method.

Parameters
func – The function object to check.

Returns
`True` if the function or method uses a standard Python type; `False` otherwise.

Return type
bool

class falcon.**Context**

Convenience class to hold contextual information in its attributes.

This class is used as the default *Request* and *Response* context type (see *Request.context_type* and *Response.context_type*, respectively).

In Falcon versions prior to 2.0, the default context type was `dict`. To ease the migration to attribute-based context object approach, this class also implements the mapping interface; that is, object attributes are linked to dictionary items, and vice versa. For instance:

```
>>> context = falcon.Context()
>>> context.cache_strategy = 'lru'
>>> context.get('cache_strategy')
'lru'
>>> 'cache_strategy' in context
True
```

Although we have decided to maintain the mapping interface in the foreseeable future, new code should prefer the attribute-based approach, as it is more performant.

What is more, if you continue to use the mapping interface (or mix-and-match), care needs to be taken not to overwrite `dict` methods such as `items()`, `values()`, etc.

class falcon.**ETag**

Convenience class to represent a parsed HTTP entity-tag.

This class is simply a subclass of `str` with a few helper methods and an extra attribute to indicate whether the entity-tag is weak or strong. The value of the string is equivalent to what RFC 7232 calls an "opaque-tag", i.e. an entity-tag sans quotes and the weakness indicator.

Note

Given that a weak entity-tag comparison can be performed by using the == operator (per the example below), only a *strong_compare()* method is provided.

Here is an example `on_get()` method that demonstrates how to use instances of this class:

```
def on_get(self, req, resp):
    content_etag = self._get_content_etag()
    for etag in (req.if_none_match or []):
```

(continues on next page)

(continued from previous page)

```
        if etag == '*' or etag == content_etag:
            resp.status = falcon.HTTP_304
            return

    # -- snip --

    resp.etag = content_etag
    resp.status = falcon.HTTP_200
```

(See also: RFC 7232)

dumps() → str

Serialize the ETag to a string suitable for use in a precondition header.

(See also: RFC 7232, Section 2.3)

Returns
An opaque quoted string, possibly prefixed by a weakness indicator `W/`.

Return type
str

is_weak: bool = False

`True` if the entity-tag is weak, otherwise `False`.

classmethod loads(*etag_str: str*) → *ETag*

Deserialize a single entity-tag string from a precondition header.

> **Note**
>
> This method is meant to be used only for parsing a single entity-tag. It can not be used to parse a comma-separated list of values.

(See also: RFC 7232, Section 2.3)

Parameters
etag_str (*str*) – An ASCII string representing a single entity-tag, as defined by RFC 7232.

Returns
An instance of *~.ETag* representing the parsed entity-tag.

Return type
ETag

strong_compare(*other:* ETag) → bool

Perform a strong entity-tag comparison.

Two entity-tags are equivalent if both are not weak and their opaque-tags match character-by-character.

(See also: RFC 7232, Section 2.3.2)

Parameters

- **other** (ETag) – The other *ETag* to which you are comparing
- **one.** (*this*)

Returns
`True` if the two entity-tags match, otherwise `False`.

Return type
bool

5.3.16 Testing Helpers

Functional testing framework for Falcon apps and Falcon itself.

Falcon's testing module contains various test classes and utility functions to support functional testing for both Falcon-based apps and the Falcon framework itself.

The testing framework supports both `unittest` and pytest.

Tests are normally carried out by simulating HTTP requests by calling the corresponding *`TestClient`* methods, e.g., *`simulate_get()`*, *`simulate_post()`*, etc:

```
# -----------------------------------------------------------------
# unittest
# -----------------------------------------------------------------

from falcon import testing
import myapp


class MyTestCase(testing.TestCase):
    def setUp(self):
        super(MyTestCase, self).setUp()

        # Assume the hypothetical `myapp` package has a
        # function called `create()` to initialize and
        # return a `falcon.App` instance.
        self.app = myapp.create()


class TestMyApp(MyTestCase):
    def test_get_message(self):
        doc = {'message': 'Hello world!'}

        result = self.simulate_get('/messages/42')
        self.assertEqual(result.json, doc)


# -----------------------------------------------------------------
# pytest
# -----------------------------------------------------------------

from falcon import testing
import pytest

import myapp


# Depending on your testing strategy and how your application
# manages state, you may be able to broaden the fixture scope
# beyond the default 'function' scope used in this example.

@pytest.fixture()
def client():
    # Assume the hypothetical `myapp` package has a function called
    # `create()` to initialize and return a `falcon.App` instance.
    return testing.TestClient(myapp.create())
```

(continues on next page)

(continued from previous page)

```
def test_get_message(client):
    doc = {'message': 'Hello world!'}

    result = client.simulate_get('/messages/42')
    assert result.json == doc
```

As shown above, the responses rendered by the application are encapsulated by the test `Result`.

Tip

`Result` objects implement a `__rich__` method for facilitating a rich-text representation when used together with the popular rich library.

For instance, provided you have installed both Falcon and `rich` into your environment, you should be able to see a prettier rendition of the below 404-result:

```
>>> import falcon
>>> import falcon.testing
>>> import rich.pretty
>>> rich.pretty.install()
>>> client = falcon.testing.TestClient(falcon.App())
>>> client.get('/endpoint')
Result<404 Not Found application/json b'{"title": "404 Not Found"}'>
```

Simulating Requests

Main Interface

class falcon.testing.**TestClient**(*app: Callable[[...], Any]*, *headers: Mapping[str, str] | None = None*)

Simulate requests to a WSGI or ASGI application.

This class provides a contextual wrapper for Falcon's `simulate_*()` test functions. It lets you replace this:

```
simulate_get(app, '/messages')
simulate_head(app, '/messages')
```

with this:

```
client = TestClient(app)
client.simulate_get('/messages')
client.simulate_head('/messages')
```

For convenience, `TestClient` also exposes shorthand aliases without the `simulate_` prefix. Just as with a typical Python HTTP client, it is possible to simply call:

```
client = TestClient(app)
client.get('/messages')
client.request('LOCK', '/files/first')
```

Note

The methods all call `self.simulate_request()` for convenient overriding of request preparation by child classes.

Note

In the case of an ASGI request, this class will simulate the entire app lifecycle in a single shot, including lifespan and client disconnect events. In order to simulate multiple interleaved requests, or to test a streaming endpoint (such as one that emits server-sent events), *ASGIConductor* can be used to more precisely control the app lifecycle.

An instance of *ASGIConductor* may be instantiated directly, or obtained from an instance of *TestClient* using the context manager pattern, as per the following example:

```
client = falcon.testing.TestClient(app)

# -- snip --

async with client as conductor:
    async with conductor.simulate_get_stream('/events') as result:
        pass
```

Parameters

app (*callable*) – A WSGI or ASGI application to target when simulating requests

Keyword Arguments

headers (*dict*) – Default headers to set on every request (default `None`). These defaults may be overridden by passing values for the same headers to one of the `simulate_*()` methods.

app: *falcon.App*

The app that this client instance was configured to use.

delete(**args: Any*, ***kwargs: Any*) → Any

Simulate a DELETE request to a WSGI application.

(See also: *falcon.testing.simulate_delete()*)

Added in version 3.1.

get(**args: Any*, ***kwargs: Any*) → Any

Simulate a GET request to a WSGI application.

(See also: *falcon.testing.simulate_get()*)

Added in version 3.1.

head(**args: Any*, ***kwargs: Any*) → Any

Simulate a HEAD request to a WSGI application.

(See also: *falcon.testing.simulate_head()*)

Added in version 3.1.

options(**args: Any*, ***kwargs: Any*) → Any

Simulate an OPTIONS request to a WSGI application.

(See also: *falcon.testing.simulate_options()*)

Added in version 3.1.

patch(**args: Any*, ***kwargs: Any*) → Any

Simulate a PATCH request to a WSGI application.

(See also: *falcon.testing.simulate_patch()*)

Added in version 3.1.

post (**args: Any*, ***kwargs: Any*) → Any

Simulate a POST request to a WSGI application.

(See also: `falcon.testing.simulate_post()`)

Added in version 3.1.

put (**args: Any*, ***kwargs: Any*) → Any

Simulate a PUT request to a WSGI application.

(See also: `falcon.testing.simulate_put()`)

Added in version 3.1.

request (**args: Any*, ***kwargs: Any*) → Any

Simulate a request to a WSGI application.

Wraps `falcon.testing.simulate_request()` to perform a WSGI request directly against `self.app`. Equivalent to:

```
falcon.testing.simulate_request(self.app, *args, **kwargs)
```

Added in version 3.1.

simulate_delete (*path: str = '/'*, ***kwargs: Any*) → *Result*

Simulate a DELETE request to a WSGI application.

(See also: `falcon.testing.simulate_delete()`)

simulate_get (*path: str = '/'*, ***kwargs: Any*) → *Result*

Simulate a GET request to a WSGI application.

(See also: `falcon.testing.simulate_get()`)

simulate_head (*path: str = '/'*, ***kwargs: Any*) → *Result*

Simulate a HEAD request to a WSGI application.

(See also: `falcon.testing.simulate_head()`)

simulate_options (*path: str = '/'*, ***kwargs: Any*) → *Result*

Simulate an OPTIONS request to a WSGI application.

(See also: `falcon.testing.simulate_options()`)

simulate_patch (*path: str = '/'*, ***kwargs: Any*) → *Result*

Simulate a PATCH request to a WSGI application.

(See also: `falcon.testing.simulate_patch()`)

simulate_post (*path: str = '/'*, ***kwargs: Any*) → *Result*

Simulate a POST request to a WSGI application.

(See also: `falcon.testing.simulate_post()`)

simulate_put (*path: str = '/'*, ***kwargs: Any*) → *Result*

Simulate a PUT request to a WSGI application.

(See also: `falcon.testing.simulate_put()`)

simulate_request (**args: Any*, ***kwargs: Any*) → *Result*

Simulate a request to a WSGI application.

Wraps `falcon.testing.simulate_request()` to perform a WSGI request directly against `self.app`. Equivalent to:

```
falcon.testing.simulate_request(self.app, *args, **kwargs)
```

class falcon.testing.**ASGIConductor**(*app: Callable[[...], Any]*, *headers: Mapping[str, str] | None = None*)

Test conductor for ASGI apps.

This class provides more control over the lifecycle of a simulated request as compared to *TestClient*. In addition, the conductor's asynchronous interface affords interleaved requests and the testing of streaming protocols such as *Server-Sent Events (SSE)* and *WebSocket*.

ASGIConductor is implemented as a context manager. Upon entering and exiting the context, the appropriate ASGI lifespan events will be simulated.

Within the context, HTTP requests can be simulated using an interface that is similar to *TestClient*, except that all `simulate_*()` methods are coroutines:

```
async with testing.ASGIConductor(some_app) as conductor:
    async def post_events():
        for i in range(100):
            await conductor.simulate_post('/events', json={'id': i}):
            await asyncio.sleep(0.01)

    async def get_events_sse():
        # Here, we will get only some of the single server-sent events
        # because the non-streaming method is "single-shot". In other
        # words, simulate_get() will emit a client disconnect event
        # into the app before returning.
        result = await conductor.simulate_get('/events')

        # Alternatively, we can use simulate_get_stream() as a context
        # manager to perform a series of reads on the result body that
        # are interleaved with the execution of the post_events()
        # coroutine.
        async with conductor.simulate_get_stream('/events') as sr:
            while some_condition:
                # Read next body chunk that was received (if any).
                chunk = await sr.stream.read()

                if chunk:
                    # TODO: Do something with the chunk
                    pass

            # Exiting the context causes the request event emitter to
            # begin emitting ``'http.disconnect'`` events and then awaits
            # the completion of the asyncio task that is running the
            # simulated ASGI request.

    asyncio.gather(post_events(), get_events_sse())
```

Note

Because the *ASGIConductor* interface uses coroutines, it cannot be used directly with synchronous testing frameworks such as pytest.

As a workaround, the test can be adapted by wrapping it in an inline async function and then invoking it via *falcon.async_to_sync()* or decorating the test function with *falcon.runs_sync()*.

Alternatively, you can try searching PyPI to see if an async plugin is available for your testing framework of choice. For example, the `pytest-asyncio` plugin is available for `pytest` users.

Similar to the *TestClient*, *ASGIConductor* also exposes convenience aliases without the `simulate_` pre-

fix. Just as with a typical asynchronous HTTP client, it is possible to simply invoke:

```
await conductor.get('/messages')
await conductor.request('LOCK', '/files/first')
```

Parameters

app (*callable*) – An ASGI application to target when simulating requests.

Keyword Arguments

headers (*dict*) – Default headers to set on every request (default `None`). These defaults may be overridden by passing values for the same headers to one of the `simulate_*()` methods.

app: *asgi.App*

The app that this client instance was configured to use.

async delete(**args: Any*, ***kwargs: Any*) → Any

Simulate a DELETE request to an ASGI application.

(See also: *falcon.testing.simulate_delete()*)

Added in version 3.1.

async get(**args: Any*, ***kwargs: Any*) → Any

Simulate a GET request to an ASGI application.

(See also: *falcon.testing.simulate_get()*)

Added in version 3.1.

get_stream(**args: Any*, ***kwargs: Any*) → Any

Simulate a GET request to an ASGI application with a streamed response.

(See also: *falcon.testing.simulate_get()* for a list of supported keyword arguments.)

This method returns an async context manager that can be used to obtain a managed *StreamedResult* instance. Exiting the context will automatically finalize the result object, causing the request event emitter to begin emitting `'http.disconnect'` events and then await the completion of the task that is running the simulated ASGI request.

In the following example, a series of streamed body chunks are read from the response:

```
async with conductor.get_stream('/events') as sr:
    while some_condition:
        # Read next body chunk that was received (if any).
        chunk = await sr.stream.read()

        if chunk:
            # TODO: Do something with the chunk. For example,
            #   a series of server-sent events could be validated
            #   by concatenating the chunks and splitting on
            #   double-newlines to obtain individual events.
            pass
```

Added in version 3.1.

async head(**args: Any*, ***kwargs: Any*) → Any

Simulate a HEAD request to an ASGI application.

(See also: *falcon.testing.simulate_head()*)

Added in version 3.1.

async options(**args: Any*, ***kwargs: Any*) → Any

Simulate an OPTIONS request to an ASGI application.

(See also: *falcon.testing.simulate_options()*)

Added in version 3.1.

async patch(**args: Any*, ***kwargs: Any*) → Any

Simulate a PATCH request to an ASGI application.

(See also: *falcon.testing.simulate_patch()*)

Added in version 3.1.

async post(**args: Any*, ***kwargs: Any*) → Any

Simulate a POST request to an ASGI application.

(See also: *falcon.testing.simulate_post()*)

Added in version 3.1.

async put(**args: Any*, ***kwargs: Any*) → Any

Simulate a PUT request to an ASGI application.

(See also: *falcon.testing.simulate_put()*)

Added in version 3.1.

async request(**args: Any*, ***kwargs: Any*) → Any

Simulate a request to an ASGI application.

Wraps *falcon.testing.simulate_request()* to perform an ASGI request directly against `self.app`. Equivalent to:

```
falcon.testing.simulate_request(self.app, *args, **kwargs)
```

Added in version 3.1.

async simulate_delete(*path: str = '/'*, ***kwargs: Any*) → *Result*

Simulate a DELETE request to an ASGI application.

(See also: *falcon.testing.simulate_delete()*)

async simulate_get(*path: str = '/'*, ***kwargs: Any*) → *Result*

Simulate a GET request to an ASGI application.

(See also: *falcon.testing.simulate_get()*)

simulate_get_stream(*path: str = '/'*, ***kwargs: Any*) → _AsyncContextManager

Simulate a GET request to an ASGI application with a streamed response.

(See also: *falcon.testing.simulate_get()* for a list of supported keyword arguments.)

This method returns an async context manager that can be used to obtain a managed *StreamedResult* instance. Exiting the context will automatically finalize the result object, causing the request event emitter to begin emitting `'http.disconnect'` events and then await the completion of the task that is running the simulated ASGI request.

In the following example, a series of streamed body chunks are read from the response:

```
async with conductor.simulate_get_stream('/events') as sr:
    while some_condition:
        # Read next body chunk that was received (if any).
        chunk = await sr.stream.read()

        if chunk:
```

(continues on next page)

(continued from previous page)

```
            # TODO: Do something with the chunk. For example,
            #   a series of server-sent events could be validated
            #   by concatenating the chunks and splitting on
            #   double-newlines to obtain individual events.
            pass
```

async simulate_head(*path: str = '/'*, ***kwargs: Any*) → *Result*

Simulate a HEAD request to an ASGI application.

(See also: *falcon.testing.simulate_head()*)

async simulate_options(*path: str = '/'*, ***kwargs: Any*) → *Result*

Simulate an OPTIONS request to an ASGI application.

(See also: *falcon.testing.simulate_options()*)

async simulate_patch(*path: str = '/'*, ***kwargs: Any*) → *Result*

Simulate a PATCH request to an ASGI application.

(See also: *falcon.testing.simulate_patch()*)

async simulate_post(*path: str = '/'*, ***kwargs: Any*) → *Result*

Simulate a POST request to an ASGI application.

(See also: *falcon.testing.simulate_post()*)

async simulate_put(*path: str = '/'*, ***kwargs: Any*) → *Result*

Simulate a PUT request to an ASGI application.

(See also: *falcon.testing.simulate_put()*)

async simulate_request(**args: Any*, ***kwargs: Any*) → *Result* | *StreamedResult*

Simulate a request to an ASGI application.

Wraps *falcon.testing.simulate_request()* to perform an ASGI request directly against `self.app`. Equivalent to:

```
falcon.testing.simulate_request(self.app, *args, **kwargs)
```

simulate_ws(*path: str = '/'*, ***kwargs: Any*) → _WSContextManager

Simulate a WebSocket connection to an ASGI application.

All keyword arguments are passed through to *falcon.testing.create_scope_ws()*.

This method returns an async context manager that can be used to obtain a managed *falcon.testing.ASGIWebSocketSimulator* instance. Exiting the context will simulate a close on the WebSocket (if not already closed) and await the completion of the task that is running the simulated ASGI request.

In the following example, a series of WebSocket TEXT events are received from the ASGI app:

```
async with conductor.simulate_ws('/events') as ws:
    while some_condition:
        message = await ws.receive_text()
```

websocket(**args: Any*, ***kwargs: Any*) → Any

Simulate a WebSocket connection to an ASGI application.

All keyword arguments are passed through to *falcon.testing.create_scope_ws()*.

This method returns an async context manager that can be used to obtain a managed *falcon.testing.ASGIWebSocketSimulator* instance. Exiting the context will simulate a close on the WebSocket (if not already closed) and await the completion of the task that is running the simulated ASGI request.

In the following example, a series of WebSocket TEXT events are received from the ASGI app:

```
async with conductor.websocket('/events') as ws:
    while some_condition:
        message = await ws.receive_text()
```

Added in version 3.1.

class falcon.testing.**Result**(*iterable: Iterable[bytes]*, *status: str*, *headers: Iterable[tuple[str, str]]*)

Encapsulates the result of a simulated request.

Parameters

- **iterable** (*iterable*) – An iterable that yields zero or more bytestrings, per PEP-3333
- **status** (*str*) – An HTTP status string, including status code and reason string
- **headers** (*list*) – A list of (header_name, header_value) tuples, per PEP-3333

property content: bytes

Raw response body, or an `b''` if the response body was empty.

property content_type: str | None

Return the `Content-Type` header or `None` if missing.

property cookies: dict[str, *Cookie*]

A dictionary of *falcon.testing.Cookie* values parsed from the response, by name.

The cookies dictionary can be used directly in subsequent requests:

```
client = testing.TestClient(app)
response_one = client.simulate_get('/')
response_two = client.simulate_post('/', cookies=response_one.cookies)
```

property encoding: str | None

Text encoding of the response body.

Returns `None` if the encoding can not be determined.

property headers: dict[str, str]

A case-insensitive dictionary containing all the headers in the response, except for cookies, which may be accessed via the *cookies* attribute.

> **Note**
>
> Multiple instances of a header in the response are currently not supported; it is unspecified which value will "win" and be represented in *headers*.

property json: Any

Deserialized JSON body.

Will be `None` if the body has no content to deserialize. Otherwise, raises an error if the response is not valid JSON.

property status: str

HTTP status string given in the response.

property status_code: int

The code portion of the HTTP status string.

property text: str

Decoded response body of type `str`.

If the content type does not specify an encoding, UTF-8 is assumed.

class falcon.testing.**StreamedResult**(*body_chunks: Sequence[bytes]*, *status: str*, *headers: Iterable[tuple[str, str]]*, *task: Task*, *req_event_emitter:* ASGIRequestEventEmitter)

Encapsulates the streamed result of an ASGI request.

Parameters

- **body_chunks** (*list*) – A list of body chunks. This list may be appended to after a result object has been instantiated.
- **status** (*str*) – An HTTP status string, including status code and reason string
- **headers** (*list*) – A list of (header_name, header_value) tuples, per PEP-3333
- **task** (*asyncio.Task*) – The scheduled simulated request which may or may not have already finished. *finalize()* will await the task before returning.
- **req_event_emitter** (ASGIRequestEventEmitter) – A reference to the event emitter used to simulate events sent to the ASGI application via its receive() method. *finalize()* will cause the event emitter to simulate an `'http.disconnect'` event before returning.

property content_type: str | None

Return the `Content-Type` header or `None` if missing.

property cookies: dict[str, *Cookie*]

A dictionary of *falcon.testing.Cookie* values parsed from the response, by name.

The cookies dictionary can be used directly in subsequent requests:

```
client = testing.TestClient(app)
response_one = client.simulate_get('/')
response_two = client.simulate_post('/', cookies=response_one.cookies)
```

property encoding: str | None

Text encoding of the response body.

Returns `None` if the encoding can not be determined.

async finalize() → None

Finalize the encapsulated simulated request.

This method causes the request event emitter to begin emitting `'http.disconnect'` events and then awaits the completion of the asyncio task that is running the simulated ASGI request.

property headers: dict[str, str]

A case-insensitive dictionary containing all the headers in the response, except for cookies, which may be accessed via the *cookies* attribute.

> **Note**
>
> Multiple instances of a header in the response are currently not supported; it is unspecified which value will "win" and be represented in *headers*.

property status: str

HTTP status string given in the response.

property status_code: int

The code portion of the HTTP status string.

property stream: *ResultBodyStream*

Raw response body, as a byte stream.

class falcon.testing.**ResultBodyStream**(*chunks: Sequence[bytes]*)

Simple forward-only reader for a streamed test result body.

Parameters

chunks (*list*) – Reference to a list of body chunks that may continue to be appended to as more body events are collected.

async read() → bytes

Read any data that has been collected since the last call.

Returns

data that has been collected since the last call, or an empty byte string if no additional data is available.

Return type

bytes

class falcon.testing.**ASGIWebSocketSimulator**

Simulates a WebSocket client for testing a Falcon ASGI app.

This class provides a way to test WebSocket endpoints in a Falcon ASGI app without having to interact with an actual ASGI server. While it is certainly important to test against a real server, a number of functional tests can be satisfied more efficiently and transparently with a simulated connection.

Note

The ASGIWebSocketSimulator class is not designed to be instantiated directly; rather it should be obtained via *simulate_ws()*.

async close(*code: int | None = None*, *reason: str | None = None*) → None

Close the simulated connection.

Keyword Arguments

- **code** (*int*) – The WebSocket close code to send to the application per the WebSocket spec (default: 1000).
- **reason** (*str*) – The WebSocket close reason to send to the application per the WebSocket spec (default: empty string).

property close_code: int | None

The WebSocket close code provided by the app if the connection is closed.

Returns None if the connection is still open.

property close_reason: str | None

The WebSocket close reason provided by the app if the connection is closed.

Returns None if the connection is still open.

property closed: bool

True if the WebSocket connection has been denied or closed by the app, or the client has disconnected.

property headers: list[tuple[bytes, bytes]] | None

An iterable of [name, value] two-item tuples, where *name* is the header name, and *value* is the header value for each header returned by the app when it accepted the WebSocket connection. This property resolves to None if the connection has not been accepted.

property ready: bool

True if the WebSocket connection has been accepted and the client is still connected, False otherwise.

async receive_data() → bytes

Receive a message from the app with a binary data payload.

Awaiting this coroutine will block until a message is available or the WebSocket is disconnected.

async receive_json() → Any

Receive a message from the app with a JSON-encoded TEXT payload.

Awaiting this coroutine will block until a message is available or the WebSocket is disconnected.

async receive_msgpack() → Any

Receive a message from the app with a MessagePack-encoded BINARY payload.

Awaiting this coroutine will block until a message is available or the WebSocket is disconnected.

async receive_text() → str

Receive a message from the app with a Unicode string payload.

Awaiting this coroutine will block until a message is available or the WebSocket is disconnected.

async send_data(*payload: bytes | bytearray | memoryview*) → None

Send a message to the app with a binary data payload.

Parameters

payload (*Union[bytes, bytearray, memoryview]*) – The binary data to send.

async send_json(*media: object*) → None

Send a message to the app with a JSON-encoded payload.

Parameters

media – A JSON-encodable object to send as a TEXT (0x01) payload.

async send_msgpack(*media: object*) → None

Send a message to the app with a MessagePack-encoded payload.

Parameters

media – A MessagePack-encodable object to send as a BINARY (0x02) payload.

async send_text(*payload: str*) → None

Send a message to the app with a Unicode string payload.

Parameters

payload (*str*) – The string to send.

property subprotocol: str | None

The subprotocol the app wishes to accept, or `None` if not specified.

async wait_ready(*timeout: int | None = None*) → None

Wait until the connection has been accepted or denied.

This coroutine can be awaited in order to pause execution until the app has accepted or denied the connection. In the latter case, an error will be raised to the caller.

Keyword Arguments

timeout (*int*) – Number of seconds to wait before giving up and raising an error (default: `5`).

class falcon.testing.**Cookie**(*morsel: Morsel*)

Represents a cookie returned by a simulated request.

Parameters

morsel – A `Morsel` object from which to derive the cookie data.

property domain: str

The domain to which this cookie is restricted.

An empty string if not specified.

property expires: datetime | None

Expiration timestamp for the cookie, or `None` if not specified.

Changed in version 4.0: This property now returns timezone-aware `datetime` objects (or `None`).

property http_only: bool

Whether or not the cookie will be visible from JavaScript in the client.

property max_age: int | None

The lifetime of the cookie in seconds, or `None` if not specified.

property name: str

The cookie's name.

property partitioned: bool

Indicates if the cookie has the `Partitioned` flag set.

property path: str

The path prefix to which this cookie is restricted.

An empty string if not specified.

property same_site: str | None

Specifies whether cookies are send in cross-site requests.

Possible values are 'Lax', 'Strict' and 'None'. `None` if not specified.

property secure: bool

Whether or not the cookie may only only be transmitted from the client via HTTPS.

property value: str

The value of the cookie.

Standalone Methods

`falcon.testing.`**simulate_get**(*app: Callable[[...], Any]*, *path: str*, ***kwargs: Any*) → *Result*

Simulate a GET request to a WSGI or ASGI application.

Equivalent to:

```
simulate_request(app, 'GET', path, **kwargs)
```

Note

In the case of an ASGI request, this method will simulate the entire app lifecycle in a single shot, including lifespan and client disconnect events. In order to simulate multiple interleaved requests, or to test a streaming endpoint (such as one that emits server-sent events), *ASGIConductor* can be used to more precisely control the app lifecycle.

Parameters

- **app** (*callable*) – The application to call
- **path** (*str*) – The URL path to request

Note

The path may contain a query string. However, neither *query_string* nor *params* may be specified in this case.

Keyword Arguments

- **root_path** (*str*) – The initial portion of the request URL's "path" that corresponds to the application object, so that the application knows its virtual "location". This defaults to the empty string, indicating that the application corresponds to the "root" of the server.
- **protocol** – The protocol to use for the URL scheme (default: 'http')
- **port** (*int*) – The TCP port to simulate. Defaults to the standard port used by the given scheme (i.e., 80 for 'http' and 443 for 'https'). A string may also be passed, as long as it can be parsed as an int.
- **params** (*dict*) – A dictionary of query string parameters, where each key is a parameter name, and each value is either a `str` or something that can be converted into a `str`, or a list of such values. If a `list`, the value will be converted to a comma-delimited string of values (e.g., 'thing=1,2,3').
- **params_csv** (*bool*) – Set to `True` to encode list values in query string params as comma-separated values (e.g., 'thing=1,2,3'). Otherwise, parameters will be encoded by specifying multiple instances of the parameter (e.g., 'thing=1&thing=2&thing=3'). Defaults to `False`.
- **query_string** (*str*) – A raw query string to include in the request (default: `None`). If specified, overrides *params*.
- **headers** (*dict*) – Extra headers as a dict-like (Mapping) object, or an iterable yielding a series of two-member (*name*, *value*) iterables. Each pair of strings provides the name and value for an HTTP header. If desired, multiple header values may be combined into a single (*name*, *value*) pair by joining the values with a comma when the header in question supports the list format (see also RFC 7230 and RFC 7231). Header names are not case-sensitive.

  Note

  If a User-Agent header is not provided, it will default to:

  ```
  f'falcon-client/{falcon.__version__}'
  ```

- **file_wrapper** (*callable*) – Callable that returns an iterable, to be used as the value for *wsgi.file_wrapper* in the WSGI environ (default: `None`). This can be used to test high-performance file transmission when *resp.stream* is set to a file-like object.
- **host** (*str*) – A string to use for the hostname part of the fully qualified request URL (default: 'falconframework.org')
- **remote_addr** (*str*) – A string to use as the remote IP address for the request (default: '127.0.0.1'). For WSGI, this corresponds to the 'REMOTE_ADDR' environ variable. For ASGI, this corresponds to the IP address used for the 'client' field in the connection scope.
- **http_version** (*str*) – The HTTP version to simulate. Must be either '2', '2.0', 1.1', '1.0', or '1' (default '1.1'). If set to '1.0', the Host header will not be added to the scope.
- **wsgierrors** (*io*) – The stream to use as *wsgierrors* in the WSGI environ (default `sys.stderr`)
- **asgi_chunk_size** (*int*) – The maximum number of bytes that will be sent to the ASGI app in a single `'http.request'` event (default 4096).
- **asgi_disconnect_ttl** (*int*) – The maximum number of seconds to wait since the request was initiated, before emitting an `'http.disconnect'` event when the app calls the receive() function (default 300). Set to `0` to simulate an immediate disconnection without first emitting `'http.request'`.

- **extras** (*dict*) – Additional values to add to the WSGI `environ` dictionary or the ASGI scope for the request (default: `None`)
- **cookies** (*dict*) – Cookies as a dict-like (Mapping) object, or an iterable yielding a series of two-member (*name*, *value*) iterables. Each pair of items provides the name and value for the 'Set-Cookie' header.

Returns
The result of the request

Return type
Result

`falcon.testing.`**simulate_head**(*app: Callable[[...], Any]*, *path: str*, ***kwargs: Any*) → *Result*

Simulate a HEAD request to a WSGI or ASGI application.

Equivalent to:

```
simulate_request(app, 'HEAD', path, **kwargs)
```

> **Note**
>
> In the case of an ASGI request, this method will simulate the entire app lifecycle in a single shot, including lifespan and client disconnect events. In order to simulate multiple interleaved requests, or to test a streaming endpoint (such as one that emits server-sent events), *ASGIConductor* can be used to more precisely control the app lifecycle.

Parameters

- **app** (*callable*) – The application to call
- **path** (*str*) – The URL path to request

> **Note**
>
> The path may contain a query string. However, neither *query_string* nor *params* may be specified in this case.

Keyword Arguments

- **root_path** (*str*) – The initial portion of the request URL's "path" that corresponds to the application object, so that the application knows its virtual "location". This defaults to the empty string, indicating that the application corresponds to the "root" of the server.
- **protocol** – The protocol to use for the URL scheme (default: 'http')
- **port** (*int*) – The TCP port to simulate. Defaults to the standard port used by the given scheme (i.e., 80 for 'http' and 443 for 'https'). A string may also be passed, as long as it can be parsed as an int.
- **params** (*dict*) – A dictionary of query string parameters, where each key is a parameter name, and each value is either a `str` or something that can be converted into a `str`, or a list of such values. If a `list`, the value will be converted to a comma-delimited string of values (e.g., 'thing=1,2,3').
- **params_csv** (*bool*) – Set to `True` to encode list values in query string params as comma-separated values (e.g., 'thing=1,2,3'). Otherwise, parameters will be encoded by specifying multiple instances of the parameter (e.g., 'thing=1&thing=2&thing=3'). Defaults to `False`.
- **query_string** (*str*) – A raw query string to include in the request (default: `None`). If specified, overrides *params*.

- **headers** (*dict*) – Extra headers as a dict-like (Mapping) object, or an iterable yielding a series of two-member (*name*, *value*) iterables. Each pair of strings provides the name and value for an HTTP header. If desired, multiple header values may be combined into a single (*name*, *value*) pair by joining the values with a comma when the header in question supports the list format (see also RFC 7230 and RFC 7231). Header names are not case-sensitive.

  Note

  If a User-Agent header is not provided, it will default to:

  ```
  f'falcon-client/{falcon.__version__}'
  ```

- **host** (*str*) – A string to use for the hostname part of the fully qualified request URL (default: 'falconframework.org')
- **remote_addr** (*str*) – A string to use as the remote IP address for the request (default: '127.0.0.1'). For WSGI, this corresponds to the 'REMOTE_ADDR' environ variable. For ASGI, this corresponds to the IP address used for the 'client' field in the connection scope.
- **http_version** (*str*) – The HTTP version to simulate. Must be either '2', '2.0', 1.1', '1.0', or '1' (default '1.1'). If set to '1.0', the Host header will not be added to the scope.
- **wsgierrors** (*io*) – The stream to use as *wsgierrors* in the WSGI environ (default `sys.stderr`)
- **asgi_chunk_size** (*int*) – The maximum number of bytes that will be sent to the ASGI app in a single `'http.request'` event (default 4096).
- **asgi_disconnect_ttl** (*int*) – The maximum number of seconds to wait since the request was initiated, before emitting an `'http.disconnect'` event when the app calls the receive() function (default 300). Set to `0` to simulate an immediate disconnection without first emitting `'http.request'`.
- **extras** (*dict*) – Additional values to add to the WSGI `environ` dictionary or the ASGI scope for the request (default: `None`)
- **cookies** (*dict*) – Cookies as a dict-like (Mapping) object, or an iterable yielding a series of two-member (*name*, *value*) iterables. Each pair of items provides the name and value for the 'Set-Cookie' header.

Returns

The result of the request

Return type

Result

falcon.testing.**simulate_post**(*app: Callable[[...], Any]*, *path: str*, ***kwargs: Any*) → *Result*

Simulate a POST request to a WSGI or ASGI application.

Equivalent to:

```
simulate_request(app, 'POST', path, **kwargs)
```

Note

In the case of an ASGI request, this method will simulate the entire app lifecycle in a single shot, including lifespan and client disconnect events. In order to simulate multiple interleaved requests, or to test a streaming endpoint (such as one that emits server-sent events), *ASGIConductor* can be used to more precisely control the app lifecycle.

Parameters

- **app** (*callable*) – The application to call
- **path** (*str*) – The URL path to request

Keyword Arguments

- **root_path** (*str*) – The initial portion of the request URL's "path" that corresponds to the application object, so that the application knows its virtual "location". This defaults to the empty string, indicating that the application corresponds to the "root" of the server.
- **protocol** – The protocol to use for the URL scheme (default: 'http')
- **port** (*int*) – The TCP port to simulate. Defaults to the standard port used by the given scheme (i.e., 80 for 'http' and 443 for 'https'). A string may also be passed, as long as it can be parsed as an int.
- **params** (*dict*) – A dictionary of query string parameters, where each key is a parameter name, and each value is either a `str` or something that can be converted into a `str`, or a list of such values. If a `list`, the value will be converted to a comma-delimited string of values (e.g., 'thing=1,2,3').
- **params_csv** (*bool*) – Set to `True` to encode list values in query string params as comma-separated values (e.g., 'thing=1,2,3'). Otherwise, parameters will be encoded by specifying multiple instances of the parameter (e.g., 'thing=1&thing=2&thing=3'). Defaults to `False`.
- **query_string** (*str*) – A raw query string to include in the request (default: `None`). If specified, overrides *params*.
- **content_type** (*str*) – The value to use for the Content-Type header in the request. If specified, this value will take precedence over any value set for the Content-Type header in the *headers* keyword argument. The `falcon` module provides a number of *constants for common media types*.
- **headers** (*dict*) – Extra headers as a dict-like (Mapping) object, or an iterable yielding a series of two-member (*name*, *value*) iterables. Each pair of strings provides the name and value for an HTTP header. If desired, multiple header values may be combined into a single (*name*, *value*) pair by joining the values with a comma when the header in question supports the list format (see also RFC 7230 and RFC 7231). Header names are not case-sensitive.

  Note

  If a User-Agent header is not provided, it will default to:

  ```
  f'falcon-client/{falcon.__version__}'
  ```

- **body** (*str*) – The body of the request (default ''). The value will be encoded as UTF-8 in the WSGI environ. Alternatively, a byte string may be passed, in which case it will be used as-is.
- **json** (*JSON serializable*) – A JSON document to serialize as the body of the request (default: `None`). If specified, overrides *body* and sets the Content-Type header to `'application/json'`, overriding any value specified by either the *content_type* or *headers* arguments.
- **file_wrapper** (*callable*) – Callable that returns an iterable, to be used as the value for *wsgi.file_wrapper* in the WSGI environ (default: `None`). This can be used to test high-performance file transmission when *resp.stream* is set to a file-like object.
- **host** (*str*) – A string to use for the hostname part of the fully qualified request URL (default: 'falconframework.org')

- **remote_addr** (*str*) – A string to use as the remote IP address for the request (default: '127.0.0.1'). For WSGI, this corresponds to the 'REMOTE_ADDR' environ variable. For ASGI, this corresponds to the IP address used for the 'client' field in the connection scope.
- **http_version** (*str*) – The HTTP version to simulate. Must be either '2', '2.0', 1.1', '1.0', or '1' (default '1.1'). If set to '1.0', the Host header will not be added to the scope.
- **wsgierrors** (*io*) – The stream to use as *wsgierrors* in the WSGI environ (default `sys.stderr`)
- **asgi_chunk_size** (*int*) – The maximum number of bytes that will be sent to the ASGI app in a single `'http.request'` event (default 4096).
- **asgi_disconnect_ttl** (*int*) – The maximum number of seconds to wait since the request was initiated, before emitting an `'http.disconnect'` event when the app calls the receive() function (default 300). Set to `0` to simulate an immediate disconnection without first emitting `'http.request'`.
- **extras** (*dict*) – Additional values to add to the WSGI `environ` dictionary or the ASGI scope for the request (default: `None`)
- **cookies** (*dict*) – Cookies as a dict-like (Mapping) object, or an iterable yielding a series of two-member (*name*, *value*) iterables. Each pair of items provides the name and value for the 'Set-Cookie' header.

Returns
The result of the request

Return type
Result

`falcon.testing.`**simulate_put**(*app: Callable[[...], Any]*, *path: str*, ***kwargs: Any*) → *Result*

Simulate a PUT request to a WSGI or ASGI application.

Equivalent to:

```
simulate_request(app, 'PUT', path, **kwargs)
```

Note

In the case of an ASGI request, this method will simulate the entire app lifecycle in a single shot, including lifespan and client disconnect events. In order to simulate multiple interleaved requests, or to test a streaming endpoint (such as one that emits server-sent events), *ASGIConductor* can be used to more precisely control the app lifecycle.

Parameters

- **app** (*callable*) – The application to call
- **path** (*str*) – The URL path to request

Keyword Arguments

- **root_path** (*str*) – The initial portion of the request URL's "path" that corresponds to the application object, so that the application knows its virtual "location". This defaults to the empty string, indicating that the application corresponds to the "root" of the server.
- **protocol** – The protocol to use for the URL scheme (default: 'http')
- **port** (*int*) – The TCP port to simulate. Defaults to the standard port used by the given scheme (i.e., 80 for 'http' and 443 for 'https'). A string may also be passed, as long as it can be parsed as an int.

- **params** (*dict*) – A dictionary of query string parameters, where each key is a parameter name, and each value is either a `str` or something that can be converted into a `str`, or a list of such values. If a `list`, the value will be converted to a comma-delimited string of values (e.g., 'thing=1,2,3').
- **params_csv** (*bool*) – Set to `True` to encode list values in query string params as comma-separated values (e.g., 'thing=1,2,3'). Otherwise, parameters will be encoded by specifying multiple instances of the parameter (e.g., 'thing=1&thing=2&thing=3'). Defaults to `False`.
- **query_string** (*str*) – A raw query string to include in the request (default: `None`). If specified, overrides *params*.
- **content_type** (*str*) – The value to use for the Content-Type header in the request. If specified, this value will take precedence over any value set for the Content-Type header in the *headers* keyword argument. The `falcon` module provides a number of *constants for common media types*.
- **headers** (*dict*) – Extra headers as a dict-like (Mapping) object, or an iterable yielding a series of two-member (*name*, *value*) iterables. Each pair of strings provides the name and value for an HTTP header. If desired, multiple header values may be combined into a single (*name*, *value*) pair by joining the values with a comma when the header in question supports the list format (see also RFC 7230 and RFC 7231). Header names are not case-sensitive.

  Note

  If a User-Agent header is not provided, it will default to:

  ```
  f'falcon-client/{falcon.__version__}'
  ```

- **body** (*str*) – The body of the request (default ''). The value will be encoded as UTF-8 in the WSGI environ. Alternatively, a byte string may be passed, in which case it will be used as-is.
- **json** (*JSON serializable*) – A JSON document to serialize as the body of the request (default: `None`). If specified, overrides *body* and sets the Content-Type header to `'application/json'`, overriding any value specified by either the *content_type* or *headers* arguments.
- **file_wrapper** (*callable*) – Callable that returns an iterable, to be used as the value for *wsgi.file_wrapper* in the WSGI environ (default: `None`). This can be used to test high-performance file transmission when *resp.stream* is set to a file-like object.
- **host** (*str*) – A string to use for the hostname part of the fully qualified request URL (default: 'falconframework.org')
- **remote_addr** (*str*) – A string to use as the remote IP address for the request (default: '127.0.0.1'). For WSGI, this corresponds to the 'REMOTE_ADDR' environ variable. For ASGI, this corresponds to the IP address used for the 'client' field in the connection scope.
- **http_version** (*str*) – The HTTP version to simulate. Must be either '2', '2.0', 1.1', '1.0', or '1' (default '1.1'). If set to '1.0', the Host header will not be added to the scope.
- **wsgierrors** (*io*) – The stream to use as *wsgierrors* in the WSGI environ (default `sys.stderr`)
- **asgi_chunk_size** (*int*) – The maximum number of bytes that will be sent to the ASGI app in a single `'http.request'` event (default 4096).
- **asgi_disconnect_ttl** (*int*) – The maximum number of seconds to wait since the request was initiated, before emitting an `'http.disconnect'` event when the app calls

the receive() function (default 300). Set to `0` to simulate an immediate disconnection without first emitting `'http.request'`.

- **extras** (*dict*) – Additional values to add to the WSGI `environ` dictionary or the ASGI scope for the request (default: `None`)
- **cookies** (*dict*) – Cookies as a dict-like (Mapping) object, or an iterable yielding a series of two-member (*name*, *value*) iterables. Each pair of items provides the name and value for the 'Set-Cookie' header.

Returns
The result of the request

Return type
Result

`falcon.testing.`**simulate_options**(*app: Callable[[...], Any]*, *path: str*, ***kwargs: Any*) → *Result*

Simulate an OPTIONS request to a WSGI or ASGI application.

Equivalent to:

```
simulate_request(app, 'OPTIONS', path, **kwargs)
```

Note

In the case of an ASGI request, this method will simulate the entire app lifecycle in a single shot, including lifespan and client disconnect events. In order to simulate multiple interleaved requests, or to test a streaming endpoint (such as one that emits server-sent events), *ASGIConductor* can be used to more precisely control the app lifecycle.

Parameters

- **app** (*callable*) – The application to call
- **path** (*str*) – The URL path to request

Keyword Arguments

- **root_path** (*str*) – The initial portion of the request URL's "path" that corresponds to the application object, so that the application knows its virtual "location". This defaults to the empty string, indicating that the application corresponds to the "root" of the server.
- **protocol** – The protocol to use for the URL scheme (default: 'http')
- **port** (*int*) – The TCP port to simulate. Defaults to the standard port used by the given scheme (i.e., 80 for 'http' and 443 for 'https'). A string may also be passed, as long as it can be parsed as an int.
- **params** (*dict*) – A dictionary of query string parameters, where each key is a parameter name, and each value is either a `str` or something that can be converted into a `str`, or a list of such values. If a `list`, the value will be converted to a comma-delimited string of values (e.g., 'thing=1,2,3').
- **params_csv** (*bool*) – Set to `True` to encode list values in query string params as comma-separated values (e.g., 'thing=1,2,3'). Otherwise, parameters will be encoded by specifying multiple instances of the parameter (e.g., 'thing=1&thing=2&thing=3'). Defaults to `False`.
- **query_string** (*str*) – A raw query string to include in the request (default: `None`). If specified, overrides *params*.
- **headers** (*dict*) – Extra headers as a dict-like (Mapping) object, or an iterable yielding a series of two-member (*name*, *value*) iterables. Each pair of strings provides the name and value for an HTTP header. If desired, multiple header values may be combined into a

single (*name*, *value*) pair by joining the values with a comma when the header in question supports the list format (see also RFC 7230 and RFC 7231). Header names are not case-sensitive.

Note

If a User-Agent header is not provided, it will default to:

```
f'falcon-client/{falcon.__version__}'
```

- **host** (*str*) – A string to use for the hostname part of the fully qualified request URL (default: 'falconframework.org')
- **remote_addr** (*str*) – A string to use as the remote IP address for the request (default: '127.0.0.1'). For WSGI, this corresponds to the 'REMOTE_ADDR' environ variable. For ASGI, this corresponds to the IP address used for the 'client' field in the connection scope.
- **http_version** (*str*) – The HTTP version to simulate. Must be either '2', '2.0', 1.1', '1.0', or '1' (default '1.1'). If set to '1.0', the Host header will not be added to the scope.
- **wsgierrors** (*io*) – The stream to use as *wsgierrors* in the WSGI environ (default `sys.stderr`)
- **asgi_chunk_size** (*int*) – The maximum number of bytes that will be sent to the ASGI app in a single `'http.request'` event (default 4096).
- **asgi_disconnect_ttl** (*int*) – The maximum number of seconds to wait since the request was initiated, before emitting an `'http.disconnect'` event when the app calls the receive() function (default 300). Set to `0` to simulate an immediate disconnection without first emitting `'http.request'`.
- **extras** (*dict*) – Additional values to add to the WSGI `environ` dictionary or the ASGI scope for the request (default: `None`)

Returns

The result of the request

Return type

Result

`falcon.testing.`**simulate_patch**(*app: Callable[[...], Any]*, *path: str*, ***kwargs: Any*) → *Result*

Simulate a PATCH request to a WSGI or ASGI application.

Equivalent to:

```
simulate_request(app, 'PATCH', path, **kwargs)
```

Note

In the case of an ASGI request, this method will simulate the entire app lifecycle in a single shot, including lifespan and client disconnect events. In order to simulate multiple interleaved requests, or to test a streaming endpoint (such as one that emits server-sent events), *ASGIConductor* can be used to more precisely control the app lifecycle.

Parameters

- **app** (*callable*) – The application to call
- **path** (*str*) – The URL path to request

Keyword Arguments

- **root_path** (*str*) – The initial portion of the request URL's "path" that corresponds to the application object, so that the application knows its virtual "location". This defaults to the empty string, indicating that the application corresponds to the "root" of the server.
- **protocol** – The protocol to use for the URL scheme (default: 'http')
- **port** (*int*) – The TCP port to simulate. Defaults to the standard port used by the given scheme (i.e., 80 for 'http' and 443 for 'https'). A string may also be passed, as long as it can be parsed as an int.
- **params** (*dict*) – A dictionary of query string parameters, where each key is a parameter name, and each value is either a `str` or something that can be converted into a `str`, or a list of such values. If a `list`, the value will be converted to a comma-delimited string of values (e.g., 'thing=1,2,3').
- **params_csv** (*bool*) – Set to `True` to encode list values in query string params as comma-separated values (e.g., 'thing=1,2,3'). Otherwise, parameters will be encoded by specifying multiple instances of the parameter (e.g., 'thing=1&thing=2&thing=3'). Defaults to `False`.
- **query_string** (*str*) – A raw query string to include in the request (default: `None`). If specified, overrides *params*.
- **content_type** (*str*) – The value to use for the Content-Type header in the request. If specified, this value will take precedence over any value set for the Content-Type header in the *headers* keyword argument. The `falcon` module provides a number of *constants for common media types*.
- **headers** (*dict*) – Extra headers as a dict-like (Mapping) object, or an iterable yielding a series of two-member (*name*, *value*) iterables. Each pair of strings provides the name and value for an HTTP header. If desired, multiple header values may be combined into a single (*name*, *value*) pair by joining the values with a comma when the header in question supports the list format (see also RFC 7230 and RFC 7231). Header names are not case-sensitive.

  Note

  If a User-Agent header is not provided, it will default to:

  ```
  f'falcon-client/{falcon.__version__}'
  ```

- **body** (*str*) – The body of the request (default ''). The value will be encoded as UTF-8 in the WSGI environ. Alternatively, a byte string may be passed, in which case it will be used as-is.
- **json** (*JSON serializable*) – A JSON document to serialize as the body of the request (default: `None`). If specified, overrides *body* and sets the Content-Type header to `'application/json'`, overriding any value specified by either the *content_type* or *headers* arguments.
- **host** (*str*) – A string to use for the hostname part of the fully qualified request URL (default: 'falconframework.org')
- **remote_addr** (*str*) – A string to use as the remote IP address for the request (default: '127.0.0.1'). For WSGI, this corresponds to the 'REMOTE_ADDR' environ variable. For ASGI, this corresponds to the IP address used for the 'client' field in the connection scope.
- **http_version** (*str*) – The HTTP version to simulate. Must be either '2', '2.0', 1.1', '1.0', or '1' (default '1.1'). If set to '1.0', the Host header will not be added to the scope.
- **wsgierrors** (*io*) – The stream to use as *wsgierrors* in the WSGI environ (default `sys.stderr`)

- **asgi_chunk_size** (*int*) – The maximum number of bytes that will be sent to the ASGI app in a single `'http.request'` event (default 4096).
- **asgi_disconnect_ttl** (*int*) – The maximum number of seconds to wait since the request was initiated, before emitting an `'http.disconnect'` event when the app calls the receive() function (default 300). Set to `0` to simulate an immediate disconnection without first emitting `'http.request'`.
- **extras** (*dict*) – Additional values to add to the WSGI `environ` dictionary or the ASGI scope for the request (default: `None`)
- **cookies** (*dict*) – Cookies as a dict-like (Mapping) object, or an iterable yielding a series of two-member (*name*, *value*) iterables. Each pair of items provides the name and value for the 'Set-Cookie' header.

Returns
The result of the request

Return type
Result

falcon.testing.**simulate_delete**(*app: Callable[[...], Any]*, *path: str*, ***kwargs: Any*) → *Result*

Simulate a DELETE request to a WSGI or ASGI application.

Equivalent to:

```
simulate_request(app, 'DELETE', path, **kwargs)
```

Note

In the case of an ASGI request, this method will simulate the entire app lifecycle in a single shot, including lifespan and client disconnect events. In order to simulate multiple interleaved requests, or to test a streaming endpoint (such as one that emits server-sent events), *ASGIConductor* can be used to more precisely control the app lifecycle.

Parameters

- **app** (*callable*) – The application to call
- **path** (*str*) – The URL path to request

Keyword Arguments

- **root_path** (*str*) – The initial portion of the request URL's "path" that corresponds to the application object, so that the application knows its virtual "location". This defaults to the empty string, indicating that the application corresponds to the "root" of the server.
- **protocol** – The protocol to use for the URL scheme (default: 'http')
- **port** (*int*) – The TCP port to simulate. Defaults to the standard port used by the given scheme (i.e., 80 for 'http' and 443 for 'https'). A string may also be passed, as long as it can be parsed as an int.
- **params** (*dict*) – A dictionary of query string parameters, where each key is a parameter name, and each value is either a `str` or something that can be converted into a `str`, or a list of such values. If a `list`, the value will be converted to a comma-delimited string of values (e.g., 'thing=1,2,3').
- **params_csv** (*bool*) – Set to `True` to encode list values in query string params as comma-separated values (e.g., 'thing=1,2,3'). Otherwise, parameters will be encoded by specifying multiple instances of the parameter (e.g., 'thing=1&thing=2&thing=3'). Defaults to `False`.

- **query_string** (*str*) – A raw query string to include in the request (default: `None`). If specified, overrides *params*.
- **content_type** (*str*) – The value to use for the Content-Type header in the request. If specified, this value will take precedence over any value set for the Content-Type header in the *headers* keyword argument. The `falcon` module provides a number of *constants for common media types*.
- **headers** (*dict*) – Extra headers as a dict-like (Mapping) object, or an iterable yielding a series of two-member (*name*, *value*) iterables. Each pair of strings provides the name and value for an HTTP header. If desired, multiple header values may be combined into a single (*name*, *value*) pair by joining the values with a comma when the header in question supports the list format (see also RFC 7230 and RFC 7231). Header names are not case-sensitive.

  Note

  If a User-Agent header is not provided, it will default to:

  ```
  f'falcon-client/{falcon.__version__}'
  ```

- **body** (*str*) – The body of the request (default ''). The value will be encoded as UTF-8 in the WSGI environ. Alternatively, a byte string may be passed, in which case it will be used as-is.
- **json** (*JSON serializable*) – A JSON document to serialize as the body of the request (default: `None`). If specified, overrides *body* and sets the Content-Type header to `'application/json'`, overriding any value specified by either the *content_type* or *headers* arguments.
- **host** (*str*) – A string to use for the hostname part of the fully qualified request URL (default: 'falconframework.org')
- **remote_addr** (*str*) – A string to use as the remote IP address for the request (default: '127.0.0.1'). For WSGI, this corresponds to the 'REMOTE_ADDR' environ variable. For ASGI, this corresponds to the IP address used for the 'client' field in the connection scope.
- **http_version** (*str*) – The HTTP version to simulate. Must be either '2', '2.0', 1.1', '1.0', or '1' (default '1.1'). If set to '1.0', the Host header will not be added to the scope.
- **wsgierrors** (*io*) – The stream to use as *wsgierrors* in the WSGI environ (default `sys.stderr`)
- **asgi_chunk_size** (*int*) – The maximum number of bytes that will be sent to the ASGI app in a single `'http.request'` event (default 4096).
- **asgi_disconnect_ttl** (*int*) – The maximum number of seconds to wait since the request was initiated, before emitting an `'http.disconnect'` event when the app calls the receive() function (default 300). Set to `0` to simulate an immediate disconnection without first emitting `'http.request'`.
- **extras** (*dict*) – Additional values to add to the WSGI `environ` dictionary or the ASGI scope for the request (default: `None`)
- **cookies** (*dict*) – Cookies as a dict-like (Mapping) object, or an iterable yielding a series of two-member (*name*, *value*) iterables. Each pair of items provides the name and value for the 'Set-Cookie' header.

Returns

The result of the request

Return type

Result

falcon.testing.**simulate_request**(*app: Callable[[...], Any]*, *method: str = 'GET'*, *path: str = '/'*, *query_string: str | None = None*, *headers: Mapping[str, str] | Iterable[tuple[str, str]] | None = None*, *content_type: str | None = None*, *body: str | bytes | None = None*, *json: Any | None = None*, *file_wrapper: Callable[[...], Any] | None = None*, *wsgierrors: TextIO | None = None*, *params: Mapping[str, Any] | None = None*, *params_csv: bool = False*, *protocol: str = 'http'*, *host: str = 'falconframework.org'*, *remote_addr: str | None = None*, *extras: Mapping[str, Any] | None = None*, *http_version: str = '1.1'*, *port: int | None = None*, *root_path: str | None = None*, *cookies: Mapping[str, str | Cookie] | None = None*, *asgi_chunk_size: int = 4096*, *asgi_disconnect_ttl: int = 300*) → *Result*

Simulate a request to a WSGI or ASGI application.

Performs a request against a WSGI or ASGI application. In the case of WSGI, uses `wsgiref.validate` to ensure the response is valid.

Note

In the case of an ASGI request, this method will simulate the entire app lifecycle in a single shot, including lifespan and client disconnect events. In order to simulate multiple interleaved requests, or to test a streaming endpoint (such as one that emits server-sent events), `ASGIConductor` can be used to more precisely control the app lifecycle.

Keyword Arguments

- **app** (*callable*) – The WSGI or ASGI application to call
- **method** (*str*) – An HTTP method to use in the request (default: 'GET')
- **path** (*str*) – The URL path to request (default: '/').

  Note

  The path may contain a query string. However, neither *query_string* nor *params* may be specified in this case.

- **root_path** (*str*) – The initial portion of the request URL's "path" that corresponds to the application object, so that the application knows its virtual "location". This defaults to the empty string, indicating that the application corresponds to the "root" of the server.
- **protocol** – The protocol to use for the URL scheme (default: 'http')
- **port** (*int*) – The TCP port to simulate. Defaults to the standard port used by the given scheme (i.e., 80 for 'http' and 443 for 'https'). A string may also be passed, as long as it can be parsed as an int.
- **params** (*dict*) – A dictionary of query string parameters, where each key is a parameter name, and each value is either a `str` or something that can be converted into a `str`, or a list of such values. If a `list`, the value will be converted to a comma-delimited string of values (e.g., 'thing=1,2,3').
- **params_csv** (*bool*) – Set to `True` to encode list values in query string params as comma-separated values (e.g., 'thing=1,2,3'). Otherwise, parameters will be encoded by specifying multiple instances of the parameter (e.g., 'thing=1&thing=2&thing=3'). Defaults to `False`.
- **query_string** (*str*) – A raw query string to include in the request (default: `None`). If specified, overrides *params*.
- **content_type** (*str*) – The value to use for the Content-Type header in the request. If specified, this value will take precedence over any value set for the Content-Type header

in the *headers* keyword argument. The `falcon` module provides a number of *constants for common media types*.

- **headers** (*dict*) – Extra headers as a dict-like (Mapping) object, or an iterable yielding a series of two-member (*name*, *value*) iterables. Each pair of strings provides the name and value for an HTTP header. If desired, multiple header values may be combined into a single (*name*, *value*) pair by joining the values with a comma when the header in question supports the list format (see also RFC 7230 and RFC 7231). Header names are not case-sensitive.

  Note

  If a User-Agent header is not provided, it will default to:

  ```
  f'falcon-client/{falcon.__version__}'
  ```

- **body** (*str*) – The body of the request (default ''). The value will be encoded as UTF-8 in the WSGI environ. Alternatively, a byte string may be passed, in which case it will be used as-is.
- **json** (*JSON serializable*) – A JSON document to serialize as the body of the request (default: `None`). If specified, overrides *body* and sets the Content-Type header to `'application/json'`, overriding any value specified by either the *content_type* or *headers* arguments.
- **file_wrapper** (*callable*) – Callable that returns an iterable, to be used as the value for *wsgi.file_wrapper* in the WSGI environ (default: `None`). This can be used to test high-performance file transmission when *resp.stream* is set to a file-like object.
- **host** (*str*) – A string to use for the hostname part of the fully qualified request URL (default: 'falconframework.org')
- **remote_addr** (*str*) – A string to use as the remote IP address for the request (default: '127.0.0.1'). For WSGI, this corresponds to the 'REMOTE_ADDR' environ variable. For ASGI, this corresponds to the IP address used for the 'client' field in the connection scope.
- **http_version** (*str*) – The HTTP version to simulate. Must be either '2', '2.0', 1.1', '1.0', or '1' (default '1.1'). If set to '1.0', the Host header will not be added to the scope.
- **wsgierrors** (*io*) – The stream to use as *wsgierrors* in the WSGI environ (default `sys.stderr`)
- **asgi_chunk_size** (*int*) – The maximum number of bytes that will be sent to the ASGI app in a single `'http.request'` event (default 4096).
- **asgi_disconnect_ttl** (*int*) – The maximum number of seconds to wait since the request was initiated, before emitting an `'http.disconnect'` event when the app calls the receive() function (default 300).
- **extras** (*dict*) – Additional values to add to the WSGI `environ` dictionary or the ASGI scope for the request (default: `None`)
- **cookies** (*dict*) – Cookies as a dict-like (Mapping) object, or an iterable yielding a series of two-member (*name*, *value*) iterables. Each pair of items provides the name and value for the 'Set-Cookie' header.

Returns

The result of the request

Return type

Result

falcon.testing.**capture_responder_args**(*req: wsgi.Request*, *resp: wsgi.Response*, *resource: object*, *params: Mapping[str, str]*) → None

Before hook for capturing responder arguments.

Adds the following attributes to the hooked responder's resource class:

- *captured_req*
- *captured_resp*
- *captured_kwargs*

In addition, if the capture-req-body-bytes header is present in the request, the following attribute is added:

- *captured_req_body*

Including the capture-req-media header in the request (set to any value) will add the following attribute:

- *capture-req-media*

async falcon.testing.**capture_responder_args_async**(*req:* asgi.Request, *resp:* asgi.Response, *resource: Resource*, *params: Mapping[str, str]*) → None

Before hook for capturing responder arguments.

An asynchronous version of *capture_responder_args()*.

falcon.testing.**set_resp_defaults**(*req: wsgi.Request*, *resp: wsgi.Response*, *resource: Resource*, *params: Mapping[str, str]*) → None

Before hook for setting default response properties.

This hook simply sets the the response body, status, and headers to the *_default_status*, *_default_body*, and *_default_headers* attributes that are assumed to be defined on the resource object.

async falcon.testing.**set_resp_defaults_async**(*req:* asgi.Request, *resp:* asgi.Response, *resource: Resource*, *params: Mapping[str, str]*) → None

Wrap *set_resp_defaults()* in a coroutine.

Low-Level Utils

class falcon.testing.**StartResponseMock**

Mock object representing a WSGI *start_response* callable.

property call_count: int

Number of times *start_response* was called.

headers: Iterable[tuple[str, str]] | None

Raw headers list passed to *start_response*, per PEP-3333.

headers_dict: dict[str, str]

Headers as a case-insensitive `dict`-like object, instead of a `list`.

status: str | None

HTTP status line, e.g. '785 TPS Cover Sheet not attached'.

class falcon.testing.**ASGIRequestEventEmitter**(*body: str | bytes | None = None*, *chunk_size: int | None = None*, *disconnect_at: int | float | None = None*)

Emits events on-demand to an ASGI app.

This class can be used to drive a standard ASGI app callable in order to perform functional tests on the app in question.

> **Note**
>
> In order to ensure the app is able to handle subtle variations in the ASGI events that are allowed by the specification, such variations are applied to the emitted events at unspecified intervals. This includes whether or not the *more_body* field is explicitly set, or whether or not the request *body* chunk in the event is occasionally empty,

Keyword Arguments

- **body** (*str*) – The body content to use when emitting http.request events. May be an empty string. If a byte string, it will be used as-is; otherwise it will be encoded as UTF-8 (default `b''`).
- **chunk_size** (*int*) – The maximum number of bytes to include in a single http.request event (default 4096).
- **disconnect_at** (*float*) – The Unix timestamp after which to begin emitting `'http.disconnect'` events (default now + 30s). The value may be either an `int` or a `float`, depending on the precision required. Setting *disconnect_at* to `0` is treated as a special case, and will result in an `'http.disconnect'` event being immediately emitted (rather than first emitting an `'http.request'` event).

disconnect(*exhaust_body: bool | None = None*) → None

Set the client connection state to disconnected.

Call this method to simulate an immediate client disconnect and begin emitting `'http.disconnect'` events.

Parameters

exhaust_body (*bool*) – Set to `False` in order to begin emitting `'http.disconnect'` events without first emitting at least one `'http.request'` event.

property disconnected: bool

Returns `True` if the simulated client connection is in a "disconnected" state.

class falcon.testing.**ASGILifespanEventEmitter**(*shutting_down: Condition*)

Emits ASGI lifespan events to an ASGI app.

This class can be used to drive a standard ASGI app callable in order to perform functional tests on the app in question.

When simulating both lifespan and per-request events, each event stream will require a separate invocation of the ASGI callable; one with a lifespan event emitter, and one with a request event emitter. An asyncio `Condition` can be used to pause the lifespan emitter until all of the desired request events have been emitted.

Keyword Arguments

shutting_down (*asyncio.Condition*) – An instance of `asyncio.Condition` that will be awaited before emitting the final shutdown event (`'lifespan.shutdown`).

class falcon.testing.**ASGIResponseEventCollector**

Collects and validates ASGI events returned by an app.

Raises

- **TypeError** – An event field emitted by the app was of an unexpected type.
- **ValueError** – Invalid event name or field value.

body_chunks: list[bytes]

An iterable of `bytes` objects emitted by the app via `'http.response.body'` events.

events: list[Mapping[str, Any]]

An iterable of events that were emitted by the app, collected as-is from the app.

headers: list[tuple[str, str]]

An iterable of (str, str) tuples representing the ISO-8859-1 decoded headers emitted by the app in the body of the `'http.response.start'` event.

more_body: bool | None

Whether or not the app expects to emit more body chunks.

Will be `None` if unknown (i.e., the app has not yet emitted any `'http.response.body'` events.)

status: HTTPStatus | str | int | None

HTTP status code emitted by the app in the body of the `'http.response.start'` event.

falcon.testing.**create_environ**(*path: str = '/', query_string: str = '', http_version: str = '1.1', scheme: str = 'http', host: str = 'falconframework.org', port: int | None = None, headers: Mapping[str, str] | Iterable[tuple[str, str]] | None = None, app: str | None = None, body: str | bytes = b'', method: str = 'GET', wsgierrors: TextIO | None = None, file_wrapper: Callable[[...], Any] | None = None, remote_addr: str | None = None, root_path: str | None = None, cookies: Mapping[str, str | Cookie] | None = None*) → dict[str, Any]

Create a mock PEP-3333 environ `dict` for simulating WSGI requests.

Keyword Arguments

- **path** (*str*) – The path for the request (default `'/'`)
- **query_string** (*str*) – The query string to simulate, without a leading `'?'` (default `''`). The query string is passed as-is (it will not be percent-encoded).
- **http_version** (*str*) – The HTTP version to simulate. Must be either `'2'`, `'2.0'`, `'1.1'`, `'1.0'`, or `'1'` (default `'1.1'`). If set to `'1.0'`, the Host header will not be added to the scope.
- **scheme** (*str*) – URL scheme, either `'http'` or `'https'` (default `'http'`)
- **host** (*str*) – Hostname for the request (default `'falconframework.org'`)
- **port** (*int*) – The TCP port to simulate. Defaults to the standard port used by the given scheme (i.e., 80 for `'http'` and 443 for `'https'`). A string may also be passed, as long as it can be parsed as an int.
- **headers** (*dict*) – Headers as a dict-like (Mapping) object, or an iterable yielding a series of two-member (*name*, *value*) iterables. Each pair of strings provides the name and value for an HTTP header. If desired, multiple header values may be combined into a single (*name*, *value*) pair by joining the values with a comma when the header in question supports the list format (see also RFC 7230 and RFC 7231). Header names are not case-sensitive.

  Note

  If a User-Agent header is not provided, it will default to:

  ```
  f'falcon-client/{falcon.__version__}'
  ```

- **root_path** (*str*) – Value for the `SCRIPT_NAME` environ variable, described in PEP-3333: 'The initial portion of the request URL's "path" that corresponds to the application object, so that the application knows its virtual "location". This may be an empty string, if the application corresponds to the "root" of the server.' (default `''`)
- **app** (*str*) – Deprecated alias for *root_path*. If both kwargs are passed, *root_path* takes precedence.

- **body** (*str*) – The body of the request (default `''`). The value will be encoded as UTF-8 in the WSGI environ. Alternatively, a byte string may be passed, in which case it will be used as-is.
- **method** (*str*) – The HTTP method to use (default `'GET'`)
- **wsgierrors** (*io*) – The stream to use as *wsgierrors* (default `sys.stderr`)
- **file_wrapper** – Callable that returns an iterable, to be used as the value for *wsgi.file_wrapper* in the environ.
- **remote_addr** (*str*) – Remote address for the request to use as the `'REMOTE_ADDR'` environ variable (default `None`)
- **cookies** (*dict*) – Cookies as a dict-like (Mapping) object, or an iterable yielding a series of two-member (*name*, *value*) iterables. Each pair of items provides the name and value for the Set-Cookie header.

falcon.testing.**create_scope**(*path: str = '/'*, *query_string: str = ''*, *method: str = 'GET'*, *headers: Mapping[str, str] | Iterable[tuple[str, str]] | None = None*, *host: str = 'falconframework.org'*, *scheme: str | None = None*, *port: int | None = None*, *http_version: str = '1.1'*, *remote_addr: str | None = None*, *root_path: str | None = None*, *content_length: int | None = None*, *include_server: bool = True*, *cookies: Mapping[str, str | Cookie] | None = None*) → dict[str, Any]

Create a mock ASGI scope `dict` for simulating HTTP requests.

Keyword Arguments

- **path** (*str*) – The path for the request (default `'/'`)
- **query_string** (*str*) – The query string to simulate, without a leading `'?'` (default `''`). The query string is passed as-is (it will not be percent-encoded).
- **method** (*str*) – The HTTP method to use (default `'GET'`)
- **headers** (*dict*) – Headers as a dict-like (Mapping) object, or an iterable yielding a series of two-member (*name*, *value*) iterables. Each pair of strings provides the name and value for an HTTP header. If desired, multiple header values may be combined into a single (*name*, *value*) pair by joining the values with a comma when the header in question supports the list format (see also RFC 7230 and RFC 7231). When the request will include a body, the Content-Length header should be included in this list. Header names are not case-sensitive.

  Note

  If a User-Agent header is not provided, it will default to:

  ```
  f'falcon-client/{falcon.__version__}'
  ```

- **host** (*str*) – Hostname for the request (default `'falconframework.org'`). This also determines the value of the Host header in the request.
- **scheme** (*str*) – URL scheme, either `'http'` or `'https'` (default `'http'`)
- **port** (*int*) – The TCP port to simulate. Defaults to the standard port used by the given scheme (i.e., 80 for `'http'` and 443 for `'https'`). A string may also be passed, as long as it can be parsed as an int.
- **http_version** (*str*) – The HTTP version to simulate. Must be either `'2'`, `'2.0'`, `'1.1'`, `'1.0'`, or `'1'` (default `'1.1'`). If set to `'1.0'`, the Host header will not be added to the scope.
- **remote_addr** (*str*) – Remote address for the request to use for the 'client' field in the connection scope (default None)

- **root_path** (*str*) – The root path this application is mounted at; same as SCRIPT_NAME in WSGI (default `''`).
- **content_length** (*int*) – The expected content length of the request body (default `None`). If specified, this value will be used to set the Content-Length header in the request.
- **include_server** (*bool*) – Set to `False` to not set the 'server' key in the scope `dict` (default `True`).
- **cookies** (*dict*) – Cookies as a dict-like (Mapping) object, or an iterable yielding a series of two-member (*name*, *value*) iterables. Each pair of items provides the name and value for the 'Set-Cookie' header.

Added in version 4.1: The raw (i.e., not URL-decoded) version of the provided *path* is now preserved in the returned scope as the `raw_path` byte string. According to the ASGI specification, `raw_path` **does not include** any query string.

`falcon.testing.`**create_scope_ws**(*path: str = '/'*, *query_string: str = ''*, *headers: Mapping[str, str] | Iterable[tuple[str, str]] | None = None*, *host: str = 'falconframework.org'*, *scheme: str | None = None*, *port: int | None = None*, *http_version: str = '1.1'*, *remote_addr: str | None = None*, *root_path: str | None = None*, *include_server: bool = True*, *subprotocols: str | None = None*, *spec_version: str = '2.1'*) → dict[str, Any]

Create a mock ASGI scope `dict` for simulating WebSocket requests.

Keyword Arguments

- **path** (*str*) – The path for the request (default `'/'`)
- **query_string** (*str*) – The query string to simulate, without a leading `'?'` (default `''`). The query string is passed as-is (it will not be percent-encoded).
- **headers** (*dict*) – Headers as a dict-like (Mapping) object, or an iterable yielding a series of two-member (*name*, *value*) iterables. Each pair of strings provides the name and value for an HTTP header. If desired, multiple header values may be combined into a single (*name*, *value*) pair by joining the values with a comma when the header in question supports the list format (see also RFC 7230 and RFC 7231). When the request will include a body, the Content-Length header should be included in this list. Header names are not case-sensitive.

  Note

  If a User-Agent header is not provided, it will default to:

  ```
  f'falcon-client/{falcon.__version__}'
  ```

- **host** (*str*) – Hostname for the request (default `'falconframework.org'`). This also determines the value of the Host header in the request.
- **scheme** (*str*) – URL scheme, either `'ws'` or `'wss'` (default `'ws'`)
- **port** (*int*) – The TCP port to simulate. Defaults to the standard port used by the given scheme (i.e., 80 for `'ws'` and 443 for `'wss'`). A string may also be passed, as long as it can be parsed as an int.
- **http_version** (*str*) – The HTTP version to simulate. Must be either `'2'`, `'2.0'`, or `'1.1'` (default `'1.1'`).
- **remote_addr** (*str*) – Remote address for the request to use for the 'client' field in the connection scope (default None)
- **root_path** (*str*) – The root path this application is mounted at; same as SCRIPT_NAME in WSGI (default `''`).

- **include_server** (*bool*) – Set to `False` to not set the 'server' key in the scope `dict` (default `True`).
- **spec_version** (*str*) – The ASGI spec version to emulate (default `'2.1'`).
- **subprotocols** (*Iterable[str]*) – Subprotocols the client wishes to advertise to the server (default `[]`).

falcon.testing.**create_req**(*options:* RequestOptions | *None* = *None*, ***kwargs:* *Any*) → *Request*

Create and return a new Request instance.

This function can be used to conveniently create a WSGI environ and use it to instantiate a *falcon.Request* object in one go.

The arguments for this function are identical to those of *falcon.testing.create_environ()*, except an additional *options* keyword argument may be set to an instance of *falcon.RequestOptions* to configure certain aspects of request parsing in lieu of the defaults.

falcon.testing.**create_asgi_req**(*body:* *bytes* | *None* = *None*, *req_type:* *type*[Request] | *None* = *None*, *options:* RequestOptions | *None* = *None*, ***kwargs:* *Any*) → *Request*

Create and return a new ASGI Request instance.

This function can be used to conveniently create an ASGI scope and use it to instantiate a *falcon.asgi.Request* object in one go.

The arguments for this function are identical to those of *falcon.testing.create_scope()*, with the addition of *body*, *req_type*, and *options* arguments as documented below.

Keyword Arguments

- **body** (*bytes*) – The body data to use for the request (default b''). If the value is a `str`, it will be UTF-8 encoded to a byte string.
- **req_type** (*object*) – A subclass of *falcon.asgi.Request* to instantiate. If not specified, the standard *falcon.asgi.Request* class will simply be used.
- **options** (falcon.RequestOptions) – An instance of *falcon.RequestOptions* that should be used to determine certain aspects of request parsing in lieu of the defaults.

falcon.testing.**closed_wsgi_iterable**(*iterable:* *Iterable*[*bytes*]) → Iterable[bytes]

Wrap an iterable to ensure its `close()` method is called.

Wraps the given *iterable* in an iterator utilizing a `for` loop as illustrated in the PEP-3333 server/gateway side example. Finally, if the iterable has a `close()` method, it is called upon exception or exhausting iteration.

Furthermore, the first bytestring yielded from iteration, if any, is prefetched before returning the wrapped iterator in order to ensure the WSGI `start_response` function is called even if the WSGI application is a generator.

Parameters

iterable (*iterable*) – An iterable that yields zero or more bytestrings, per PEP-3333

Returns

An iterator yielding the same bytestrings as *iterable*

Return type

iterator

Other Helpers

Test Cases

class falcon.testing.**TestCase**(*methodName='runTest'*)

Extends `unittest` to support WSGI/ASGI functional testing.

> **Note**
>
> If available, uses `testtools` in lieu of `unittest`.

This base class provides some extra plumbing for unittest-style test cases, to help simulate WSGI or ASGI requests without having to spin up an actual web server. Various simulation methods are derived from *falcon.testing.TestClient*.

Simply inherit from this class in your test case classes instead of `unittest.TestCase` or `testtools.TestCase`.

app: *App*

A WSGI or ASGI application to target when simulating requests (defaults to `falcon.App()`). When testing your application, you will need to set this to your own instance of *falcon.App* or *falcon.asgi.App*. For example:

```
from falcon import testing
import myapp


class MyTestCase(testing.TestCase):
    def setUp(self):
        super(MyTestCase, self).setUp()

        # Assume the hypothetical `myapp` package has a
        # function called `create()` to initialize and
        # return a `falcon.App` instance.
        self.app = myapp.create()


class TestMyApp(MyTestCase):
    def test_get_message(self):
        doc = {'message': 'Hello world!'}

        result = self.simulate_get('/messages/42')
        self.assertEqual(result.json, doc)
```

setUp() → None

Hook method for setting up the test fixture before exercising it.

class falcon.testing.**SimpleTestResource**(*status: str | None = None*, *body: str | None = None*, *json: dict[str, str] | None = None*, *headers: HeaderArg | None = None*)

Mock resource for functional testing of framework components.

This class implements a simple test resource that can be extended as needed to test middleware, hooks, and the Falcon framework itself.

Only noop `on_get()` and `on_post()` responders are implemented; when overriding these, or adding additional responders in child classes, they can be decorated with the *falcon.testing.capture_responder_args()* hook in order to capture the *req*, *resp*, and *params* arguments that are passed to the responder. Responders may also be decorated with the *falcon.testing.set_resp_defaults()* hook in order to set *resp* properties to default *status*, *body*, and *header* values.

Keyword Arguments

- **status** (*str*) – Default status string to use in responses
- **body** (*str*) – Default body string to use in responses

- **json** (*JSON serializable*) – Default JSON document to use in responses. Will be serialized to a string and encoded as UTF-8. Either *json* or *body* may be specified, but not both.
- **headers** (*dict*) – Default set of additional headers to include in responses

property called: bool

Whether or not a req/resp was captured.

captured_kwargs: Any | None

The last dictionary of kwargs, beyond `req` and `resp`, that were passed into any one of the responder methods.

captured_req: wsgi.Request | *asgi.Request* | None

The last Request object passed into any one of the responder methods.

captured_req_body: bytes | None

The last Request body provided to any one of the responder methods.

This value is only captured when the `'capture-req-body-bytes'` header is set on the request. The value of the header is the number of bytes to read.

captured_req_media: Any | None

The last Request media provided to any one of the responder methods.

This value is only captured when the `'capture-req-media'` header is set on the request.

captured_resp: wsgi.Response | *asgi.Response* | None

The last Response object passed into any one of the responder methods.

Functions

`falcon.testing.`**rand_string**(*min: int*, *max: int*) → str

Return a randomly-generated string, of a random length.

Parameters

- **min** (*int*) – Minimum string length to return, inclusive
- **max** (*int*) – Maximum string length to return, inclusive

`falcon.testing.`**get_unused_port**() → int

Get an unused localhost port for use by a test server.

> **Warning**
>
> It is possible for a third party to bind to the returned port before the caller is able to do so. The caller will need to retry with a different port in that case.

> **Warning**
>
> This method has only be tested on POSIX systems and may not work elsewhere.

`falcon.testing.`**redirected**(*stdout: ~typing.TextIO = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *stderr: ~typing.TextIO = <_io.TextIOWrapper name='<stderr>' mode='w' encoding='utf-8'>*) → Iterator[None]

Redirect stdout or stderr temporarily.

For instance, this helper can be used to capture output from Falcon reources under tests:

```
import io

import falcon
import falcon.testing


class MediaPrinter:
    def on_post(self, req, resp):
        print(req.get_media())


client = falcon.testing.TestClient(falcon.App())
client.app.add_route('/print', MediaPrinter())

output = io.StringIO()
with falcon.testing.redirected(stdout=output):
    client.simulate_post('/print', json={'message': 'Hello'})

assert output.getvalue() == "{'message': 'Hello'}\n"
```

Tip

The popular pytest also captures and suppresses output from successful tests by default.

falcon.testing.**get_encoding_from_headers**(*headers: Mapping[str, str]*) → str | None

Return encoding from given HTTP Header Dict.

Parameters

headers (*dict*) – Dictionary from which to extract encoding. Header names must either be lowercase or the dict must support case-insensitive lookups.

5.3.17 Typing

Type checking support was introduced in Falcon *4.0*. While most of the library is now typed, further type annotations may be added throughout the 4.x release cycle. To improve them, we may introduce changes to the typing that do not affect runtime behavior, but may surface new or different errors with type checkers.

Note

All undocumented type aliases coming from `falcon._typing` are considered private to the framework itself, and not meant for annotating applications using Falcon. To that end, it is advisable to only use classes from the public interface, and public aliases from *falcon.typing*, e.g.:

```
class MyResource:
    def on_get(self, req: falcon.Request, resp: falcon.Response) -> None:
        resp.media = {'message': 'Hello, World!'}
```

If you still decide to reuse the private aliases anyway, they should preferably be imported inside `if TYPE_CHECKING:` blocks in order to avoid possible runtime errors after an update. Also, make sure to *let us know* which essential aliases are missing from the public interface!

App Types

Falcon's *App* (and *asgi.App*) is a generic type parametrized by its request and response types. Consequently, static type checkers (such as Mypy or Pyright) can correctly infer the specialized *App* type from the *request_type* and/or *response_type* arguments supplied to the initializer.

The use of generics should in most cases require no explicit effort on your side. However, if you annotate your variables or return types as `falcon.App`, the type checker may require you to provide the explicit type parameters when running in the strict mode (Mypy calls the option `--disallow-any-generics`, also part of the `--strict` mode flag).

For instance, the following mini-application will not pass type checking with Mypy in the `--strict` mode:

```
import falcon


class HelloResource:
    def on_get(self, req: falcon.Request, resp: falcon.Response) -> None:
        resp.media = {'message': 'Hello, typing!'}


def create_app() -> falcon.App:
    app = falcon.App()
    app.add_route('/', HelloResource())
    return app
```

In order to address this `type-arg` issue, we could explicitly specify which variant of `App` our `create_app()` is expected to instantiate:

```
import falcon


class HelloResource:
    def on_get(self, req: falcon.Request, resp: falcon.Response) -> None:
        resp.media = {'message': 'Hello, typing!'}


def create_app() -> falcon.App[falcon.Request, falcon.Response]:
    app = falcon.App()
    app.add_route('/', HelloResource())
    return app
```

Alternatively, we could ask ourselves what the purpose of `create_app()` is. If we want to instantiate a WSGI application suitable for a PEP-3333 compliant app server, we could type it accordingly:

```
import wsgiref.simple_server
import wsgiref.types

import falcon


class HelloResource:
    def on_get(self, req: falcon.Request, resp: falcon.Response) -> None:
        resp.media = {'message': 'Hello, typing!'}


def create_app() -> wsgiref.types.WSGIApplication:
    app = falcon.App()
    app.add_route('/', HelloResource())
```

(continues on next page)

(continued from previous page)

```
    return app


if __name__ == '__main__':
    with wsgiref.simple_server.make_server('', 8000, create_app()) as httpd:
        httpd.serve_forever()
```

Both alternatives should now pass type checking in the `--strict` mode.

> **Attention**
>
> For illustration purposes, we also included a `wsgiref.simple_server`-based server in the second revised example, allowing you to run the file directly with Python 3.11+. However, for a real deployment you should always *install* a production-ready WSGI or ASGI server.

Changed in version 4.2: *`falcon.App`* and *`falcon.asgi.App`* are now annotated as `generic types` parametrized by the request and response classes.

Known Limitations

Falcon's emphasis on flexibility and performance presents certain challenges when it comes to adding type annotations to the existing code base.

One notable limitation involves using custom *`Request`* and/or *`Response`* types together with a custom *`context type`*:

```
from falcon import Request


class MyRichContext:
    """My fancy context type with well annotated attributes."""

    ...


class MyRequest(Request):
    context_type = MyRichContext
```

Although a code base employing the above pattern may pass type checking without any warnings even under `--strict` settings, the problem here is that *`MyRequest.context`* is still annotated as *`Context`*, allowing arbitrary attribute access. As a result, this would mask any potential typing issues in the use of `MyRichContext`.

If you make extensive use of a custom context type, and do want to perform type checking against its interface, you can explicitly redefine *context* as having the desired type. In order to convince the type checker, this will require at least one strategically placed `# type: ignore`:

```
from falcon import Request


class MyRichContext:
    """My fancy context type with well annotated attributes."""

    ...


class MyRequest(Request):
    context_type = MyRichContext
```

(continues on next page)

(continued from previous page)

```
    context: MyRichContext  # type: ignore[assignment]
```

Our efforts to work around this issue have so far hit the wall of PEP 526, which states that a `ClassVar` parameter cannot include any type variables, regardless of the level of nesting.

If you come up with an elegant solution to this problem, *let us know*!

Another known inconsistency is the typing of the *`converter interface`*, where certain subclasses (such as *`Path-Converter`*) declare a different input type than the base `convert()` method. (See also the discussions and possible solutions on the GitHub issue #2396.)

Important

The above issues are only typing limitations that have no effect outside of type checking – applications will work just fine at runtime!

Public Type Aliases

Public Falcon type alias definitions.

class falcon.typing.**AsyncReadableIO**(**args*, ***kwargs*)

Async file-like protocol that defines only a read method, and is iterable.

Added in version 4.0.

falcon.typing.**Headers**

Headers dictionary returned by the framework.

Added in version 4.0.

class falcon.typing.**ReadableIO**(**args*, ***kwargs*)

File-like protocol that defines only a read method.

Added in version 4.0.

falcon.typing.**SSEEmitter**

Async generator or iterator over Server-Sent Events (instances of *`falcon.asgi.SSEvent`*).

Added in version 4.0.

5.4 Changelogs

5.4.1 Changelog for Falcon 4.2.0

Summary

Falcon 4.2.0 primarily contains typing enhancements and performance optimizations. This release also marks the debut of pre-compiled wheels for the *free-threaded* CPython 3.14 build. *Let us know* if you are experimenting with scaling Falcon applications using free-threading!

The typing improvements focus on making the WSGI and ASGI App types *generic* (parametrized by the request and response types). This should make it significantly easier to properly annotate applications that leverage custom request and/or response types.

Additionally, we have fixed a reproducibility issue (thanks to @bmwiedemann from openSUSE for reporting!) in our documentation build process. Regarding packaging Falcon for distributions in general, we would like to remind you of the *Packaging Guide* that was published with the previous Falcon release (4.1.0). We hope this guide proves useful.

This release also incorporates a number of pull requests submitted by our community. Sincere thanks to all 8 contributors who made this release possible!

Changes to Supported Platforms

- The end-of-life Python 3.8 is no longer supported. (#2527)
- Although the Falcon 4.x series is only guaranteed to support Python 3.10+, this release still supports the end-of-life Python 3.9 at runtime using the pure Python wheel.

  (Note that 3.9 is completely unsupported for new development of Falcon apps, as type checking, building documentation, etc, are likely to fail outright.)
- Pre-compiled wheels are no longer provided for the macOS x86_64 (Intel) platform. Falcon will continue to *install* (from the pure Python wheel) and run on macOS Intel just fine. You can also build the binary from *sdist* yourself. (#2536)
- Pre-compiled wheels are now available for the free-threaded CPython 3.14 build on selected Linux platforms. (See also: *Can I run Falcon on free-threaded CPython?*) (#2501)

New & Improved

- The URI encoding functions `encode()`, `encode_check_escaped()`, `encode_value()`, and `encode_value_check_escaped()` have been reimplemented in Cython, leading to a significant speed bump under CPython (provided Falcon is *installed* from a binary wheel, or cythonized from a source distribution).

  The performance improvement can reach an order of magnitude, although the actual numbers vary depending on the input data. For instance, on CPython 3.12, calling `encode_value()` with `'no-reply@falconframework.org'` would be sped up ~9x, and using the already encoded value `'no-reply%40falconframework.org'` with `encode_value_check_escaped()` could yield a ~12x speed boost. (#1169)
- `falcon.App` and `falcon.asgi.App` have been converted to `generic types` parametrized by the request and response types, and popular type checkers such as Mypy and Pyright should now correctly infer the use of custom subclasses passed by *request_type* and *response_type*, respectively. (#2372)
- Falcon now supports the retrieval-like `QUERY` HTTP method; handlers can be implemented using `on_query(req, resp, ...)` on resources. See the IETF draft for details: https://www.ietf.org/archive/id/draft-ietf-httpbis-safe-method-w-body-11.html. (#2539)

Fixed

- Falcon's documentation build process was not fully reproducible, as the output could vary with the build system's date. The issue was addressed by adding support for setting the build date via the SOURCE_DATE_EPOCH standardized environment variable. (#2567)

Contributors to this Release

Many thanks to all of our talented and stylish contributors for this release!

- AyanAhmedKhan
- CaselIT
- kemingy
- MannXo
- sonephyo
- TudorGR
- vytas7
- x612skm

5.4.2 Changelog for Falcon 4.1.0

Summary

This release contains enhancements to media handling, serving static files, and a fix for the WebSockets-sink interaction, alongside performance optimizations and full support for CPython 3.14.

During this release cycle, we have migrated to publishing to PyPI with a Trusted Publisher (thanks to @webknjaz for helping to iron out the workflow details).

For those relying on other package distribution channels than PyPI, we have prepared a brand new *Packaging Guide* for Falcon. Please check it out and let us know what you think! Additionally, we have formalized our security maintenance policy as well as the status of stable releases: *Releases and Versioning*.

This release also incorporates many pull requests submitted by our community. We would like to extend our heartfelt thanks to all 17 contributors who made this release possible!

Changes to Supported Platforms

- CPython 3.14 is now fully supported. (#2413)
- Although the Falcon 4.x series is only guaranteed to support Python 3.10+, this release still supports 3.8 & 3.9 at runtime using the pure Python wheel.

  Falcon 4.2 is expected to drop the end-of-life Python 3.8 completely (but runtime support will continue for 3.9 on a best effort basis).

New & Improved

- *StaticRoute* now renders `Etag` headers. It also checks `If-None-Match` in requests and returns HTTP 304 response if appropriate. (#2243)
- *StaticRoute* now sets the `Last-Modified` header when serving static files. The improved implementation also checks the value of the `If-Modified-Since` header, and renders an HTTP 304 response when the requested file has not been modified. (#2244)
- Similar to *create_environ()*, the *create_scope()* testing helper now preserves the raw URI path, and propagates it to the created ASGI connection scope as the `raw_path` byte string (according to the ASGI specification). (#2262)
- Two new *media_type constants*, `falcon.MEDIA_CSV` and `falcon.MEDIA_PARQUET`, were added in order to provide better support for Python data analysis applications out of the box. (#2335)
- Support for allowing *cross-origin* private network access was added to the built-in *CORSMiddleware*. The new feature is off by default, and can be enabled by passing the keyword argument `allow_private_network=True` to *CORSMiddleware* during initialization. (#2381)
- The *falcon.secure_filename()* utility function can now ensure that the length of the sanitized filename does not exceed the requested limit (passed via the `max_length` argument). In addition, a new option, `max_secure_filename_length`, was added to *MultipartParseOptions* in order to automatically populate this argument when referencing a body part's *secure_filename*. (#2420)
- The *unset_cookie()* method now accepts a `same_site` parameter (with underscore) for consistency with *set_cookie()*. The previous `samesite` parameter (without underscore) is now deprecated (referencing it will emit a deprecation warning). (#2453)
- A new method, `__rich__`, has been added to *falcon.testing.Result* for facilitating a rich-text representation when used together with the popular rich library.

  Provided you have installed both `falcon` and `rich` into your environment, you should be able to see a prettier rendition of the below 404-result:

```
>>> import falcon
>>> import falcon.testing
>>> import rich.pretty
```

(continues on next page)

(continued from previous page)

```
>>> rich.pretty.install()
>>> client = falcon.testing.TestClient(falcon.App())
>>> client.get('/endpoint')
Result<404 Not Found application/json b'{"title": "404 Not Found"}'>
```

(The actual appearance may depend on your terminal and/or REPL settings.) (#2457)

- The *cythonization* process was revised in the light of the performance improvements in newer CPython versions (especially 3.12+), and the compilation is now largely confined to hand-crafted C/Cython code. As a result, the framework should run even faster on modern CPython. (#2470)
- `JSONHandler` can now detect a non-standard (not a subclass of `ValueError`) deserialization error type for a custom *loads* function.

  (Normally, `json.loads()` and third party alternatives do raise a subclass of `ValueError` on invalid input data, however, this is not the case for, e.g., the popular *msgspec* library at the time of writing.) (#2476)

Fixed

- Previously, Falcon's *WebSocket implementation* was not documented to route the request to any `sink`. However, in the case of a missing route, a matching sink was actually invoked, passing `ws` in place of the incompatible `resp`.

  This mismatch has been addressed by introducing a *ws* keyword argument (similar to ASGI `error handlers`) for sink functions meant to accept WebSocket connections.

  For backwards-compatibility, when *ws* is absent from the sink's signature, the `WebSocket` object is still passed in place of the incompatible *resp*. This behavior will change in Falcon 5.0: when draining a WebSocket connection, *resp* will always be set to `None`. (#2414)

Misc

- The readability of the *Contributing docs* was improved by properly rendering GitHub Markdown-flavored checkboxes. (#2318)
- The `falcon.testing.httpnow` compatibility alias is now considered deprecated, and will be removed in Falcon 5.0. Use the `falcon.http_now()` function instead. (#2389)

Contributors to this Release

Many thanks to all of our talented and stylish contributors for this release!

- aarcex3
- AbduazizZiyodov
- Bombaclath97
- bssyousefi
- CaselIT
- Cycloctane
- diegomirandap
- EricGoulart
- jap
- jkmnt
- kemingy
- Krishn1412
- perodriguezl

- Shreshth3
- vytas7
- webknjaz
- x612skm

5.4.3 Changelog for Falcon 4.0.2

Summary

This is a minor point release to fix some missed re-exports for type checkers. In addition, we have also included a couple of documentation improvements.

Fixed

- Running Mypy on code that uses parts of `falcon.testing` would previously lead to errors like:

  ```
  Name "falcon.testing.TestClient" is not defined
  ```

  This has been fixed by explicitly exporting the names that are imported into the `falcon.testing` namespace. (#2387)

Misc

- The printable PDF version of our documentation was enabled on Read the Docs. (#2365)

Contributors to this Release

Many thanks to those who contributed to this bugfix release:

- AkshayAwate
- CaselIT
- chitvs
- jap
- vytas7

5.4.4 Changelog for Falcon 4.0.1

Summary

This is a minor point release addressing a Python distribution issue in Falcon 4.0.0.

Fixed

- Installing Falcon 4.0.0 unexpectedly copies many unintended directories from the source tree to the venv's `site-packages`. This issue has been rectified, and our CI has been extended with new tests (that verify what is actually installed from the distribution) to make sure this regression does not resurface. (#2384)

5.4.5 Changelog for Falcon 4.0.0

Summary

We are happy to present Falcon 4.0, a new major version of the framework that brings a couple of commonly requested features including support for matching multiple path segments (using *`PathConverter`*), and a fully typed codebase. (Please read more about typing in the notes below.)

The timeframe for Falcon 4.0 was challenging due to the need to balance our high standards with the CPython 3.13 timeline. We aimed to deliver the main development branch in this release, without resorting to another compatibility micro update (as we did with Falcon 3.1.1-3.1.3). Following community feedback, we also want to improve our

overall release schedule by shipping smaller increments more often. To support this goal, we have made several tooling and testing improvements: the build process for *binary wheels* has been simplified using cibuildwheel, and our test suite now only requires `pytest` as a hard dependency. Additionally, you can run `pytest` against our tests from any directory. We hope that these changes should also benefit packaging Falcon in Linux distributions.

As with every SemVer major release, we have removed a number of previously deprecated functions, classes, compatibility shims, as well as made other potentially breaking changes that we could not risk in a minor version. If you have been paying attention the deprecation warnings from the 3.x series, the impact should be minimal, but please do take a look at the list of breaking changes below.

This release would not have been possible without the numerous contributions from our community. This release alone comprises a number of pull requests submitted by a group of 30 talented individuals. What is more, we were particularly impressed by the high-quality discussions and code submissions during our EuroPython 2024 Sprint. Some notable sprint contributions include CHIPS support, and a new *WebSocket Tutorial*, among others. In fact, according to the statistics on GitHub, we are thrilled to report that the total number of Falcon contributors has now exceeded 200. We find it fascinating that our framework has become a collaborative effort involving so many individuals, and would like to thank everyone who has made this release possible!

Changes to Supported Platforms

- CPython 3.11 is now fully supported. (#2072)
- CPython 3.12 is now fully supported. (#2196)
- CPython 3.13 is now fully supported. (#2258)
- End-of-life Python 3.5, 3.6 & 3.7 are no longer supported. (#2074, #2273)
- End-of-life Python 3.8 is no longer actively supported, but the framework should still continue to install from the pure-Python wheel or source distribution, and function normally.
- The Falcon 4.x series is guaranteed to support CPython 3.10 and PyPy3.10 (v7.3.16). This means that we may drop the support for Python 3.8 & 3.9 altogether in a later 4.x release, especially if we are faced with incompatible ecosystem changes in typing, Cython, etc.

Typing Support

Type checking support was introduced in version 4.0. While most of the library is now typed, further type annotations may be added throughout the 4.x release cycle. To improve them, we may introduce changes to the typing that do not affect runtime behavior, but may surface new or different errors with type checkers.

Note

All undocumented type aliases coming from `falcon._typing` are considered private to the framework itself, and not meant for annotating applications using Falcon. To that end, it is advisable to only use classes from the public interface, and public aliases from *`falcon.typing`*, e.g.:

```
class MyResource:
    def on_get(self, req: falcon.Request, resp: falcon.Response) -> None:
        resp.media = {'message': 'Hello, World!'}
```

If you still decide to reuse the private aliases anyway, they should preferably be imported inside `if TYPE_CHECKING:` blocks in order to avoid possible runtime errors after an update. Also, make sure to *let us know* which essential aliases are missing from the public interface!

Known typing limitations

Falcon's emphasis on flexibility and performance has presented certain challenges when it comes to adding type annotations to the existing code base. One notable limitation involves using custom *`Request`* and/or *`Response`* types in callbacks that are passed back to the framework, such as when adding an *`error handler`*.

For instance, the following application might unexpectedly not pass type checking:

```
from typing import Any

from falcon import App, HTTPInternalServerError, Request, Response


class MyRequest(Request):
    ...


def handle_os_error(req: MyRequest, resp: Response, ex: Exception,
                    params: dict[str, Any]) -> None:
    raise HTTPInternalServerError(title='OS error!') from ex


app = App(request_type=MyRequest)
app.add_error_handler(OSError, handle_os_error)
```

(Please also see the following GitHub issue: #2372.)

Important

This is only a typing limitation that has no effect outside of type checking – the above `app` will run just fine!

Breaking Changes

- Falcon is no longer vendoring the python-mimeparse library; the relevant functionality has instead been reimplemented in the framework itself, fixing a handful of long-standing bugs in the new implementation.

  If you use standalone python-mimeparse in your project, do not worry! We will continue to maintain it as a separate package under the Falconry umbrella (we took over about 3 years ago).

  The following new behaviors are considered breaking changes:

  - Previously, the iterable passed to *`req.client_prefers`* had to be sorted in the order of increasing desirability. *`best_match()`*, and by proxy *`client_prefers()`*, now consider the provided media types to be sorted in the (more intuitive, we hope) order of decreasing desirability.
  - Unlike `python-mimeparse`, the new *media type utilities* consider media types with different values for the same parameters as non-matching.

    One theoretically possible scenario where this change can affect you is only installing a *media* handler for a content type with parameters; it then may not match media types with conflicting values (that used to match before Falcon 4.0). If this turns out to be the case, also *install the same handler* for the generic `type/subtype` without parameters.

  The new functions, *`falcon.mediatypes.quality()`* and *`falcon.mediatypes.best_match()`*, otherwise have the same signature as the corresponding methods from `python-mimeparse`. (#864)

- A number of undocumented internal helpers were renamed to start with an underscore, indicating they are private methods intended to be used only by the framework itself:

  - `falcon.request_helpers.header_property` → `falcon.request_helpers._header_property`
  - `falcon.request_helpers.parse_cookie_header` → `falcon.request_helpers._parse_cookie_header`
  - `falcon.response_helpers.format_content_disposition` → `falcon.response_helpers._format_content_disposition`
  - `falcon.response_helpers.format_etag_header` → `falcon.response_helpers._format_etag_header`

  - `falcon.response_helpers.format_header_value_list` → `falcon.response_helpers._format_header_value_list`
  - `falcon.response_helpers.format_range` → `falcon.response_helpers._format_range`
  - `falcon.response_helpers.header_property` → `falcon.response_helpers._header_property`
  - `falcon.response_helpers.is_ascii_encodable` → `falcon.response_helpers._is_ascii_encodable`

  If you were relying on these internal helpers, you can either copy the implementation into your codebase, or switch to the underscored variants. (Needless to say, though, we strongly recommend against referencing private methods, as we provide no SemVer guarantees for them.) (#1457)

- A number of previously deprecated methods, attributes and classes have now been removed:
  - In Falcon 3.0, the use of positional arguments was deprecated for the optional initializer parameters of *falcon.HTTPError* and its subclasses.

    We have now redefined these optional arguments as keyword-only, so passing them as positional arguments will result in a `TypeError`:

```
>>> import falcon
>>> falcon.HTTPForbidden('AccessDenied')
Traceback (most recent call last):
  <...>
TypeError: HTTPForbidden.__init__() takes 1 positional argument but 2 were␣
↪given
>>> falcon.HTTPForbidden('AccessDenied', 'No write access')
Traceback (most recent call last):
  <...>
TypeError: HTTPForbidden.__init__() takes 1 positional argument but 3 were␣
↪given
```

    Instead, simply pass these parameters as keyword arguments:

```
>>> import falcon
>>> falcon.HTTPForbidden(title='AccessDenied')
<HTTPForbidden: 403 Forbidden>
>>> falcon.HTTPForbidden(title='AccessDenied', description='No write access
↪')
<HTTPForbidden: 403 Forbidden>
```

  - The `falcon-print-routes` command-line utility is no longer supported; `falcon-inspect-app` is a direct replacement.
  - *falcon.stream.BoundedStream* is no longer re-imported via `falcon.request_helpers`. If needed, import it directly as *falcon.stream.BoundedStream*.
  - A deprecated alias of *falcon.stream.BoundedStream*, `falcon.stream.Body`, was removed. Use *falcon.stream.BoundedStream* instead.
  - A deprecated utility function, `falcon.get_http_status()`, was removed. Please use *falcon.code_to_http_status()* instead.
  - A deprecated routing utility, `compile_uri_template()`, was removed. This function was only employed in the early versions of the framework, and is expected to have been fully supplanted by the *CompiledRouter*. In a pinch, you can simply copy its implementation from the Falcon 3.x source tree into your application.
  - The deprecated `Response.add_link()` method was removed; please use *Response.append_link* instead.

  - The deprecated `has_representation()` method for *`HTTPError`* was removed, along with the `NoRepresentation` and `OptionalRepresentation` classes.
  - An undocumented, deprecated public method `find_by_media_type()` of *`media.Handlers`* was removed. Apart from configuring handlers for Internet media types, the rest of *`Handlers`* is only meant to be used internally by the framework (unless documented otherwise).
  - Previously, the `json` module could be imported via `falcon.util`. This deprecated alias was removed; please import `json` directly from the `standard library`, or another third-party JSON library of choice.

  We decided, on the other hand, to keep the deprecated `falcon.API` alias until Falcon 5.0. (#1853)

- Previously, it was possible to create an *`App`* with the `cors_enable` option, and add additional *`CORSMiddleware`*, leading to unexpected behavior and dysfunctional CORS. This combination now explicitly results in a `ValueError`. (#1977)
- The default value of the `csv` parameter in *`parse_query_string()`* was changed to `False`, matching the default behavior of other parts of the framework (such as *`req.params`*, the test client, etc). If the old behavior fits your use case better, pass the `csv=True` keyword argument explicitly. (#1999)
- The deprecated `api_helpers` was removed in favor of the `app_helpers` module. In addition, the deprecated `body` attributes of the *`Response`*, *`asgi.Response`*, and *`HTTPStatus`* classes were removed. (#2090)
- The function *`falcon.http_date_to_dt()`* now validates HTTP dates to have the correct timezone set. It now also returns timezone-aware `datetime` objects. As a consequence of this change, the return value of *`falcon.Request.get_header_as_datetime()`* (including the derived properties *`date`*, *`if_modified_since`*, *`if_unmodified_since`*, and *`falcon.testing.Cookie.expires`*) has also changed to timezone-aware.

  Furthermore, the default value of the *format_string* parameter in *`falcon.Request.get_param_as_datetime()`* and *`falcon.routing.DateTimeConverter`* has also been updated to use a timezone-aware form. (#2182)
- `setup.cfg` was dropped in favor of consolidating all static project configuration in `pyproject.toml` (`setup.py` is still needed for programmatic control of the build process). While this change should not impact the framework's end-users directly, some `setuptools`-based legacy workflows (such as the obsolete `setup.py test`) will no longer work. (#2314)
- The `is_async` keyword argument was removed from *`validate()`*, as well as the hooks *`before()`* and *`after()`*, since it represented a niche use case that is even less relevant with the recent advances in the ecosystem: Cython 3.0+ will now correctly mark cythonized `async def` functions as coroutines, and pure-Python factory functions that return a coroutine can now be marked as such using `inspect.markcoroutinefunction()` (Python 3.12+ is required). (#2343)

New & Improved

- A new keyword argument, *link_extension*, was added to *`falcon.Response.append_link()`* as specified in RFC 8288, Section 3.4.2. (#228)
- A new `path` *`converter`* capable of matching segments that include `/` was added. (#648)
- The new implementation of *media type utilities* (Falcon was using the `python-mimeparse` library before) now always favors the exact media type match, if one is available. (#1367)
- Type annotations have been added to Falcon's public interface to the package itself in order to better support Mypy (or other type checkers) users without having to install any third-party typeshed packages. (#1947)
- Similar to the existing *`IntConverter`*, a new *`FloatConverter`* has been added, allowing to convert path segments to `float`. (#2022)
- The default error serializer will now use the response media handlers to better negotiate the response content type with the client. The implementation still defaults to JSON if the client does not indicate any preference. (#2023)

- *WebSocket* now supports providing a reason for closing the socket, either directly via *close()* or by configuring *default_close_reasons*. (#2025)
- An informative representation was added to *testing.Result* for easier development and interpretation of failed tests. The form of __repr__ is as follows: `Result<{status_code} {content-type header} {content}>`, where the content part will reflect up to 40 bytes of the result's content. (#2044)
- A new method *falcon.Request.get_header_as_int()* was implemented. (#2060)
- A new property, *headers_lower*, was added to provide a unified, self-documenting way to get a copy of all request headers with lowercase names to facilitate case-insensitive matching. This is especially useful for middleware components that need to be compatible with both WSGI and ASGI. *headers_lower* was added in lieu of introducing a breaking change to the WSGI *headers* property that returns uppercase header names from the WSGI `environ` dictionary. (#2063)
- In Python 3.13, the `cgi` module is removed entirely from the stdlib, including its `parse_header()` method. Falcon addresses the issue by shipping an own implementation; *falcon.parse_header()* can also be used in your projects affected by the removal. (#2066)
- A new `status_code` attribute was added to the *falcon.Response*, *falcon.asgi.Response*, *HTTPStatus*, and *HTTPError* classes. (#2108)
- Following the recommendation from RFC 9239, the *MEDIA_JS* constant has been updated to `text/javascript`. Furthermore, this and other media type constants are now preferred to the stdlib's `mimetypes` for the initialization of *static_media_types*. (#2110)
- A new keyword argument, *samesite*, was added to *unset_cookie()* that allows to override the default `Lax` setting of *SameSite* on the unset cookie. (#2124)
- A new keyword argument, *partitioned*, was added to *set_cookie()* to opt a cookie into partitioned storage, with a separate cookie jar per each top-level site. (See also CHIPS for a more detailed description of this web technology.) (#2213)
- The class `falcon.HTTPPayloadTooLarge` was renamed to *falcon.HTTPContentTooLarge*, together with the accompanying HTTP *status code* update, in order to reflect the newest HTTP semantics as per RFC 9110, Section 15.5.14. (The old class name remains available as a deprecated compatibility alias.)

  In addition, one new *status code constant* was added: `falcon.HTTP_421` (also available as `falcon.HTTP_MISDIRECTED_REQUEST`) in accordance with RFC 9110, Section 15.5.20. (#2276)
- The *CORSMiddleware* now properly handles the missing `Allow` header case, by denying the preflight CORS request. The static file route has been updated to properly support CORS preflight, by allowing `GET` requests. (#2325)
- Added *falcon.testing.Result.content_type* and *falcon.testing.StreamedResult.content_type* as a utility accessor for the `Content-Type` header. (#2349)
- A new flag, *xml_error_serialization*, has been added to *ResponseOptions* that can be used to disable automatic XML serialization of *HTTPError* when using the default error serializer (and the client prefers it).

  This new flag currently defaults to `True`, preserving the same behavior as the previous Falcon versions. Falcon 5.0 will either change the default to `False`, or remove the automatic XML error serialization altogether. If you wish to retain support for XML serialization in the default error serializer, you should add a *response media handler for XML*.

  In accordance with this change, the *falcon.HTTPError.to_xml()* method was deprecated. (#2355)

Fixed

- The web servers used for tests are now run through `sys.executable` in order to ensure that they respect the virtualenv in which tests are being run. (#2047)
- Previously, importing *TestCase* as a top-level attribute in a test module could make `pytest` erroneously attempt to collect its methods as test cases. This has now been prevented by adding a `__test__` attribute (set to `False`) to the *TestCase* class. (#2147)

- Falcon will now raise an instance of *WebSocketDisconnected* from the OSError that the ASGI server signals in the case of a disconnected client (as per the ASGI HTTP & WebSocket protocol version 2.4). It is worth noting though that Falcon's *built-in receive buffer* normally detects the websocket.disconnect event itself prior the potentially failing attempt to send().

  Disabling this built-in receive buffer (by setting *max_receive_queue* to 0) was also found to interfere with receiving ASGI WebSocket messages in an unexpected way. The issue has been fixed so that setting this option to 0 now properly bypasses the buffer altogether, and extensive test coverage has been added for validating this scenario. (#2292)

- Customizing *MultipartParseOptions.media_handlers* could previously lead to unintentionally modifying a shared class variable. This has been fixed, and the *media_handlers* attribute is now initialized to a fresh copy of handlers for every instance of *MultipartParseOptions*. To that end, a proper *copy()* method has been implemented for the media *Handlers* class. (#2293)

- Falcon's multipart form parser no longer requires a CRLF (*'\r\n'*) after the closing `--` delimiter. Although it is a common convention (followed by the absolute majority of HTTP clients and web browsers) to include a trailing CRLF, the popular Undici client (used as Node's default fetch implementation) omits it at the time of this writing. (The next version of Undici will adhere to the convention.) (#2364)

Misc

- The *utility functions* create_task() and get_running_loop() are now deprecated in favor of their standard library counterparts, asyncio.create_task() and asyncio.get_running_loop(). (#2253)

- The *falcon.TimezoneGMT* class was deprecated. Use the UTC timezone (datetime.timezone.utc) from the standard library instead. (#2301)

Contributors to this Release

Many thanks to all of our talented and stylish contributors for this release!

- aarcex3
- aryaniyaps
- bssyousefi
- CaselIT
- cclauss
- chgad
- copalco
- davetapley
- derkweijers
- e-io
- euj1n0ng
- jkapica
- jkklapp
- john-g-g
- kaichan1201
- kentbull
- kgriffs
- M-Mueller
- meetshah133
- mgorny

- mihaitodor
- MRLab12
- myusko
- nfsec
- prathik2401
- RioAtHome
- TigreModerata
- vgerak
- vytas7
- wendy5667

5.4.6 Changelog for Falcon 3.1.3

Summary

This is a minor bugfix release that only pins the `pytest-asyncio` test dependency in order to prevent an incompatible version from interfering with the build workflow.

This release is otherwise identical to *Falcon 3.1.2*.

5.4.7 Changelog for Falcon 3.1.2

Summary

This is a minor point release fixing a couple of high impact bugs, as well as publishing binary wheels for the recently released CPython 3.12.

Changes to Supported Platforms

- Falcon is now supported (including binary wheels) on CPython 3.12. A couple of remaining stdlib deprecations from 3.11 and 3.12 will be addressed in Falcon 4.0.
- As with the previous release, Python 3.5 & 3.6 remain deprecated and will no longer be supported in Falcon 4.0.
- EOL Python 3.7 will no longer be actively supported in 4.0, but the framework should still continue to install from source. We may remove the support for 3.7 altogether later in the 4.x series if we are faced with incompatible ecosystem changes in typing, Cython, etc.

Fixed

- Some essential files were unintentionally omitted from the source distribution archive, rendering it unsuitable to run the test suite off. This has been fixed, and the `sdist` tarball should now be usable as a base for packaging Falcon in OS distributions. (#2051)
- *WebSocket* implementation has been fixed to properly handle *`HTTPError`* and *`HTTPStatus`* exceptions raised by custom *`error handlers`*. The WebSocket connection is now correctly closed with an appropriate code instead of bubbling up an unhandled error to the application server. (#2146)
- Falcon's *`TestClient`* mimics the behavior of real WSGI servers (and the WSGI spec) by presenting the `PATH_INFO` CGI variable already in the percent-decoded form. However, the client also used to indiscriminately set the non-standard `RAW_URI` CGI variable to `/`, which made writing tests for apps *decoding raw URL path* cumbersome. This has been fixed, and the raw path of a simulated request is now preserved in `RAW_URI`. (#2157)

Contributors to this Release

Many thanks to those who contributed to this bugfix release:

- CaselIT
- kgriffs
- liborjelinek
- vytas7

5.4.8 Changelog for Falcon 3.1.1

Summary

This is a minor point release addressing a couple of high impact bugs, and enabling the framework on the recently released CPython 3.11.

Changes to Supported Platforms

- Falcon is now functional on CPython 3.11. Full 3.11 support (including taking care of stdlib deprecations) will be formalized in Falcon 4.0.
- As with the previous release, Python 3.5 & 3.6 remain deprecated and will no longer be supported in Falcon 4.0.

Fixed

- Request attributes *`forwarded_scheme`* and *`forwarded_host`* now no longer raise an `IndexError` while processing an invalid or empty `Forwarded` header. (#2043)
- The `orjson` library now works correctly when used as JSON serializer in the media handlers in the ASGI version of Falcon. (#2100)

Contributors to this Release

Many thanks to those who contributed to this bugfix release:

- CaselIT
- kgriffs
- TBoshoven
- vytas7

5.4.9 Changelog for Falcon 3.1.0

Summary

This release contains several refinements to request validation and error handling, along with some tweaks to response handling for static and downloadable files.

Due to popular demand, *`TestClient`* and *`ASGIConductor`* now expose convenience shorthand aliases for the `simulate_*` methods, i.e., *`simulate_get()`* is now also available as *`get()`*, etc.

Some important bugs were also fixed to ensure applications properly clean up response streams and do not hang when reading request bodies that are streamed using chunked transfer encoding.

This release also adds support for CPython 3.10 and deprecates CPython 3.6.

Changes to Supported Platforms

- CPython 3.10 is now fully supported. (#1966)
- Support for Python 3.6 is now deprecated and will be removed in Falcon 4.0.
- As with the previous release, Python 3.5 support remains deprecated and will no longer be supported in Falcon 4.0.

New & Improved

- The *jsonschema.validate* decorator now raises an instance of *MediaValidationError* instead of the generic *HTTPBadRequest* for request media validation failures. Although the default behavior is kept unaltered in a backwards-compatible fashion (as the specialized exception subclasses the generic one), but it can now be easily customized by adding an error handler for the new class. (#1320)
- Due to popular demand, *TestClient* and *ASGIConductor* now expose convenience shorthand aliases for the `simulate_*` methods, i.e., *simulate_get()* is now also available as *get()*, etc. (#1806)
- The *Request.range* property now has stricter validation:
  - When parsing a byte-range-spec with a last-byte-pos, it must be greater than or equal to first-byte-pos.
  - When parsing a suffix-byte-range-spec, suffix-length must be positive.

  Static routes now support `Range` requests. This is useful for streaming media and resumable downloads. (#1858)
- Added a *Response.viewable_as* property; this is similar to *Response.downloadable_as* but with an "inline" disposition type so the response will still be displayed in the browser. (#1951)
- Added support for passing `pathlib.Path` objects as *directory* in the *add_static_route()* method on all targeted Python versions. (#1962)
- *Static routes* now set the `Content-Length` header indicating a served file's size (or length of the rendered content range). (#1991)
- When called with deprecated positional arguments, methods and class initializers (such as *falcon.HTTPError*) will now emit a user-friendlier warning indicating the fully qualified name of the method in question. (#2010)

Fixed

- If provided, the `close()` method of an ASGI *resp.stream* is now guaranteed to be called even in the case of an exception raised while iterating over the data. (#1943)
- Previously, files could be left open when serving via an ASGI static route (depending on the underlying GC implementation). This has been fixed so that a file is closed explicitly after rendering the response. (#1963)
- When a request was streamed using the chunked transfer encoding (with no `Content-Length` known in advance), iterating over *req.stream* could hang until the client had disconnected. This bug has been fixed, and iteration now stops upon receiving the last body chunk as expected. (#2024)

Misc

- The `compile_uri_template()` utility method has been deprecated and will be removed in Falcon 4.0. This function was only employed in the early versions of the framework, and is expected to have been fully supplanted by the *CompiledRouter*. In the unlikely case it is still in active use, its source code can be simply copied into an affected application. (#1967)

Contributors to this Release

Many thanks to all the contributors for this release!

- abidahmadq
- andriyor

- CaselIT
- Contessina
- dflss
- dimucciojonathan
- forana
- kgriffs
- laurent-chriqui
- maxking
- mgorny
- mihaitodor
- nix010
- signalw
- the-bets
- tipabu
- treharne
- vgerak
- vytas7

5.4.10 Changelog for Falcon 3.0.1

Summary

This is a minor point release to take care of a couple of bugs that we did not catch for 3.0.0.

Fixed

- The `api_helpers` module was re-added, since it was renamed to `app_helpers` (and effectively removed) without announcing a corresponding breaking change. This module is now considered deprecated, and will be removed in a future Falcon version. (#1902)
- ASGI HTTP headers were treated as UTF-8 encoded, not taking the incompatibility with WSGI and porting of WSGI applications into consideration. This was fixed, and ASGI headers are now decoded and encoded as ISO-8859-1. (#1911)

Contributors to this Release

Many thanks to those who contributed to this bugfix release:

- CaselIT
- vytas7

5.4.11 Changelog for Falcon 3.0.0

Summary

We are pleased to present Falcon 3.0, a major new release that includes *ASGI-based* `asyncio` and *WebSocket* support, fantastic *multipart/form-data parsing*, better error handling, enhancements to existing features, and the usual assortment of bug fixes.

This is easily the biggest release—in terms of both hours volunteered and code contributed—that we have ever done. We sincerely thank our stupendous group of 38 contributors who submitted pull requests for this release, as well as all those who have generously provided financial support to the project.

When we began working on this release, we knew we wanted to not only evolve the framework's existing features, but also to deliver first-class, user-friendly `asyncio` support alongside our existing *WSGI* feature set.

On the other hand, we have always fought the temptation to expand Falcon's scope, in order to leave room for community projects and standards to innovate around common, self-contained capabilities. And so when ASGI arrived on the scene, we saw it as the perfect opportunity to deliver long-requested `asyncio` and *WebSocket* features while still encouraging sharing and reuse within the Python web community.

It can be painful to migrate a large code base to a major new version of a framework. Therefore, in 3.0 we went to great lengths to minimize breaking changes, although a number of methods and attributes were deprecated. That being said, everyone will likely run up against at least one or two items in the breaking changes list below. Please carefully review the list of changes and thoroughly test your apps with Falcon 3.0 before deploying to production.

Leading up to this release, members of the core maintainers team spent many hours (and not a few late nights and weekends) prototyping, tuning, and testing in order to uphold the high standards of correctness and reliability for which Falcon is known. That being said, no code is perfect, so please don't hesitate to reach out on falconry/user or GitHub if you run into any issues.

Again, thanks so much to everyone who supported this release! Over the years we like to think that our little framework has had a positive impact on the Python community, and has even helped nudge the state of the art forward. And it is all thanks to our amazing supporters and contributors.

Changes to Supported Platforms

- Python 3.8 and 3.9 are now fully supported.
- Python 3.6+ is only required when using the new ASGI interface. WSGI is still supported on Python 3.5+.
- Python 3.5 support is deprecated and may be removed in the next major release.
- Python 3.4 is no longer supported.
- The Falcon 2.x series was the last to support Python language version 2. As a result, support for CPython 2.7 and PyPy2.7 was removed in Falcon 3.0.

Breaking Changes

- The class `OptionalRepresentation` and the attribute `has_representation` were deprecated. The default error serializer now generates a representation for every error type that derives from *falcon.HTTPError*. In addition, Falcon now ensures that any previously set response body is cleared before handling any raised exception. (#452)
- The class `NoRepresentation` was deprecated. All subclasses of *falcon.HTTPError* now have a media type representation. (#777)
- In order to reconcile differences between the framework's support for WSGI vs. ASGI, the following breaking changes were made:
  - *falcon.testing.create_environ()* previously set a default User-Agent header, when one was not provided, to the value `'curl/7.24.0 (x86_64-apple-darwin12.0)'`. As of Falcon 3.0, the default User-Agent string is now `f'falcon-client/{falcon.__version__}'`. This value can be overridden for the sake of backwards-compatibility by setting `falcon.testing.helpers.DEFAULT_UA`.
  - The *falcon.testing.create_environ()* function's *protocol* keyword argument was renamed to *http_version* and now only includes the version number (the value is no longer prefixed with `'HTTP/'`).
  - The *falcon.testing.create_environ()* function's *app* keyword argument was renamed to *root_path*.
  - The *writeable* property of *falcon.stream.BoundedStream* was renamed to *writable* per the standard file-like I/O interface (the old name was a misspelling)
  - If an error handler raises an exception type other than *falcon.HTTPStatus* or *falcon.HTTPError*, remaining middleware *process_response* methods will no longer be executed before bubbling up the unhandled exception to the web server.

  - `falcon.get_http_status()` no longer accepts floats, and the method itself is deprecated.
  - *falcon.app_helpers.prepare_middleware()* no longer accepts a single object; the value that is passed must be an iterable.
  - *falcon.Request.access_route* now includes the value of the *remote_addr* property as the last element in the route, if not already present in one of the headers that are checked.
  - When the `'REMOTE_ADDR'` field is not present in the WSGI environ, Falcon will assume `'127.0.0.1'` for the value, rather than simply returning `None` for *falcon.Request.remote_addr*.

  The changes above were implemented as part of the ASGI+HTTP work stream. (#1358)

- Header-related methods of the *Response* class no longer coerce the passed header name to a string via `str()`. (#1497)
- An unhandled exception will no longer be raised to the web server. Rather, the framework now installs a default error handler for the `Exception` type. This also means that middleware *process_response* methods will still be called in this case, rather than being skipped as previously. The new default error handler simply generates an HTTP 500 response. This behavior can be overridden by specifying your own error handler for `Exception` via `add_error_handler()`. (#1507)
- Exceptions are now handled by the registered handler for the most specific matching exception class, rather than in reverse order of registration. "Specificity" is determined by the method resolution order of the raised exception type. (See *add_error_handler()* for more details.) (#1514)
- The deprecated `stream_len` property was removed from the *Response* class. Please use *set_stream()* or *content_length* instead. (#1517)
- If *RequestOptions.strip_url_path_trailing_slash* is enabled, routes should now be added without a trailing slash. Previously, the trailing slash was always removed as a side effect of a bug regardless of the *strip_url_path_trailing_slash* option value. See also: *How does Falcon handle a trailing slash in the request path?* (#1544)
- Rename `falcon.Response.body` and `falcon.HTTPStatus.body` to `text`. The old name is deprecated, but still available. (#1578)
- Referencing the class *falcon.stream.BoundedStream* through the `falcon.request_helpers` module is deprecated. It is now accessible from the module `falcon.stream`. (#1583)
- General refactoring of internal media handler:
  - Deserializing an empty body with a handler that does not support it will raise *falcon.MediaNotFoundError*, and will be rendered as a `400 Bad Request` response. This error may be suppressed by passing a default value to `get_media` to be used in case of empty body. See also *falcon.Request.get_media()* for details. Previously `None` was returned in all cases without calling the handler.
  - Exceptions raised by the handlers are wrapped as *falcon.MediaMalformedError*, and will be rendered as a `400 Bad Request` response.
  - Subsequent calls to *falcon.Request.get_media()* or *falcon.Request.media* will re-raise the same exception, if the first call ended in an error, unless the exception was a *falcon.MediaNotFoundError* and a default value is passed to the `default_when_empty` attribute of the current invocation. Previously `None` was returned.

  External handlers should update their logic to align to the internal Falcon handlers. (#1589)
- The *falcon.Response.data* property now just simply returns the same data object that it was set to, if any, rather than also checking and serializing the value of the *falcon.Response.media* property. Instead, a new *render_body()* method has been implemented, which can be used to obtain the HTTP response body for the request, taking into account the *text*, *data*, and *media* attributes. (#1679)
- The `params_csv` parameter now defaults to `False` in *falcon.testing.simulate_request()*. The change was made to match the default value of the request option *auto_parse_qs_csv* (`False` since Falcon 2.0). (#1730)

- The *falcon.HTTPError.to_json()* now returns `bytes` instead of `str`. Importing `json` from `falcon.util` is deprecated. (#1767)
- The private attributes for *JSONHandler* were renamed, and the private attributes used by *MessagePackHandler* were replaced. Subclasses that refer to these variables will need to be updated. In addition, the undocumented `falcon.media.Handlers.find_by_media_type()` method was deprecated and may be removed in a future release. (#1822)

New & Improved

- ASGI+WebSocket support was added to the framework via *falcon.asgi.App* and *falcon.asgi.WebSocket*. (#321)
- The error classes in `falcon.errors` were updated to have the `title` and `description` keyword arguments and to correctly handle headers passed as list of tuples (#777)
- *MultipartFormHandler* was added to enable support for multipart forms (of content type `multipart/form-data`) through *falcon.Request.get_media()*. (#953)
- The *falcon.Response.status* attribute can now be also set to an `http.HTTPStatus` instance, an integer status code, as well as anything supported by the *falcon.code_to_http_status()* utility method. (#1135)
- A new kwarg, `cors_enable`, was added to the *falcon.App* initializer. `cors_enable` can be used to enable a simple blanket CORS policy for all responses. (See also: *CORS*.) (#1194)
- ASGI+HTTP support was added to the framework via a new class, *falcon.asgi.App*. The *testing* module was also updated to fully support ASGI apps, including two new helper functions: *falcon.testing.create_scope()* and *falcon.testing.create_asgi_req()*. WSGI users also get a new *falcon.testing.create_req()* method. As part of the ASGI work, several additional utility functions were added, including *falcon.is_python_func()*, *falcon.http_status_to_code()* and *falcon.code_to_http_status()*; as well as sync/async helpers `falcon.get_running_loop()`, `falcon.create_task()`, *falcon.sync_to_async()*, *falcon.wrap_sync_to_async()*, and *falcon.wrap_sync_to_async_unsafe()*. (#1358)
- The *falcon.App* class initializer now supports a new argument `sink_before_static_route` (default `True`, maintaining 2.0 behavior) to specify if *sinks* should be handled before or after *static routes*. (#1372)
- The *falcon.Response.append_link()* method now supports setting the *crossorigin* link CORS settings attribute. (#1410)
- Falcon now supports all WebDAV methods (RFC 2518 and RFC 4918), such as COPY, LOCK, MKCOL, MOVE, PROPFIND, PROPPATCH and UNLOCK. (#1426)
- Added inspect module to collect information about an application regarding the registered routes, middleware, static routes, sinks and error handlers (See also: *Inspect Module*.) (#1435)
- WSGI path decoding in *falcon.Request* was optimized, and is now significantly faster than in Falcon 2.0. (#1492)
- The *set_headers()* method now accepts an instance of any dict-like object that implements an `items()` method. (#1546)
- Change *falcon.routing.CompiledRouter* to compile the routes only when the first request is routed. This can be changed by passing `compile=True` to *falcon.routing.CompiledRouter.add_route()*. (#1550)
- The *set_cookie()* method now supports setting the *SameSite* cookie attribute. (#1556)
- The `falcon.API` class was renamed to *falcon.App*. The old `API` class remains available as an alias for backwards-compatibility, but it is now considered deprecated and will be removed in a future release. (#1579)
- *URLEncodedFormHandler* was added to enable support for URL-encoded forms (of content type `application/x-www-form-urlencoded`) through *falcon.Request.get_media()*. The

auto_parse_form_urlencoded option is now deprecated in favor of *URLEncodedFormHandler*. (See also: *How can I access POSTed form params?*). (#1580)

- *get_param_as_bool()* now supports the use of `'t'` and `'y'` values for `True`, as well as `'f'` and `'n'` for `False`. (#1606)
- *falcon.testing.simulate_request()* now accepts a *content_type* keyword argument. This provides a more convenient way to set this common header vs. the *headers* argument. (#1646)
- When no route matches a request, the framework will now raise a specialized subclass of *HTTPNotFound* (*HTTPRouteNotFound*) so that a custom error handler can distinguish that specific case if desired. (#1647)
- *Default media handlers* were simplified by removing a separate handler for the now-obsolete `application/json; charset=UTF-8`. As a result, providing a custom JSON media handler will now unambiguously cover both `application/json` and the former `Content-type`. (#1717)

Fixed

- Previously, the default *CompiledRouter* was erroneously stripping trailing slashes from URI templates. This has been fixed so that it is now possible to add two different routes for a path with and without a trailing forward slash (see also: *RequestOptions.strip_url_path_trailing_slash*). (#1544)
- *falcon.uri.decode()* and *falcon.uri.parse_query_string()* no longer explode quadratically for a large number of percent-encoded characters. The time complexity of these utility functions is now always close to $O(n)$. (#1594)
- When *auto_parse_qs_csv* is enabled, the framework now correctly parses all occurrences of the same parameter in the query string, rather than only splitting the values in the first occurrence. For example, whereas previously `t=1,2&t=3,4` would become `['1', '2', '3,4']`, now the resulting list will be `['1', '2', '3', '4']` (#1597)
- The *parse_query_string()* utility function is now correctly parsing an empty string as `{}`. (#1600)
- Previously, response serialization errors (such as in the case of a faulty custom media handler, or because an instance of *HTTPUnsupportedMediaType* was raised for an unsupported response content type) were unexpectedly bubbled up to the application server. This has been fixed, and these errors are now handled exactly the same way as other exceptions raised in a responder (see also: *Error Handling*). (#1607)
- *falcon.Request.forwarded_host* now contains the port when proxy headers are not set, to make it possible to correctly reconstruct the URL when the application is not behind a proxy. (#1678)
- The *Response.downloadable_as* property is now correctly encoding non-ASCII filenames as per RFC 6266 recommendations. (#1749)
- The *falcon.routing.CompiledRouter* no longer mistakenly sets route parameters while exploring non matching routes. (#1779)
- The *to_query_str()* method now correctly encodes parameter keys and values. As a result, the *params* parameter in *simulate_request()* will now correctly pass values containing special characters (such as `'&'`) to the application. (#1871)
- *falcon.uri.encode* and *falcon.uri.encode_value* now escape all percent characters by default even if it appears they have already been escaped. The *falcon.uri.encode_check_escaped* and *falcon.uri.encode_value_check_escaped* methods have been added to give the option of retaining the previous behavior where needed. These new methods have been applied to the *falcon.Response.location*, *falcon.Response.content_location*, *falcon.Response.append_link()* attrs and methods to retain previous behavior. (#1872)
- Previously, methods marked with the `deprecated()` utility wrapper could raise an unexpected `AttributeError` when running under certain applications servers such as Meinheld. This has been fixed so that `deprecated()` no longer relies on the availability of interpreter-specific stack frame introspection capabilities. (#1882)

Misc

- Deprecate the use of positional arguments for the optional kw args of the `falcon.HTTPError` subclasses (#777)
- Setup towncrier to make CHANGES reporting much easier. (#1461)
- Fix test errors on Windows (#1656)
- A new method, `get_media()`, was added that can now be used instead of the `falcon.Request.media` property to make it more clear to app maintainers that getting the media object for a request involves a side-effect of consuming and deserializing the body stream. The original property remains available to ensure backwards-compatibility with existing apps. (#1679)
- Falcon now uses the `falcon.Response` media handlers when serializing to JSON `falcon.HTTPError` and `falcon.asgi.SSEvent`. `falcon.Request` will use its defined media handler when loading a param as JSON with `falcon.Request.get_param_as_json()`. (#1767)
- The *add_link()* method of the `falcon.Request` class was renamed to `falcon.Response.append_link()`. The old name is still available as a deprecated alias. (#1801)

Contributors to this Release

Many thanks to all of our talented and stylish contributors for this release!

- adsahay
- AR4Z
- ashutoshvarma
- bibekjoshi54
- BigBlueHat
- brunneis
- CaselIT
- Ciemaar
- Coykto
- cozyDoomer
- cravindra
- csojinb
- danilito19
- edmondb
- flokX
- grktsh
- hackedd
- jmvrbanac
- karlhigley
- kemingy
- kgriffs
- mattdonders
- MinesJA
- minrock
- mivade

- mosi-kha
- myusko
- nagaabhinaya
- nZac
- pbjr23
- rmyers
- safaozturk93
- screamingskulls
- seanharrison
- timgates42
- vytas7
- waghanza
- withshubh

5.4.12 Changelog for Falcon 2.0.0

Summary

Many thanks to all of our awesome contributors (listed down below) who made this release possible!

In 2.0 we added a number of new convenience methods and properties. We also made it a lot cleaner and less error-prone to assign multiple routes to the same resource class via suffixed responders.

Also noteworthy is the significant effort we invested in improving the accuracy, clarity, and breadth of the docs. We hope these changes will help make the framework easier to learn for newcomers.

Middleware methods can now short-circuit request processing, and we improved cookie and ETag handling. Plus, the testing framework received several improvements to make it easier to simulate certain types of requests.

As this is the first major release that we have had in quite a while, we have taken the opportunity to clean up many parts of the framework. Deprecated variables, methods, and classes have been removed, along with all backwards-compatibility shims for old method signatures. We also changed the defaults for a number of request options based on community feedback.

Please carefully review the list of breaking changes below to see what you may need to tweak in your app to make it compatible with this release.

Changes to Supported Platforms

- CPython 3.7 is now fully supported.
- Falcon 2.x series is the last to support Python language version 2. As a result, support for CPython 2.7 and PyPy2.7 will be removed in Falcon 3.0.
- Support for CPython 3.4 is now deprecated and will be removed in Falcon 3.0.
- Support for CPython 2.6, CPython 3.3 and Jython 2.7 has been dropped.

Breaking Changes

- Previously, several methods in the *Response* class could be used to attempt to set raw cookie headers. However, due to the Set-Cookie header values not being combinable as a comma-delimited list, this resulted in an incorrect response being constructed for the user agent in the case that more than one cookie was being set. Therefore, the following methods of `falcon.Response` now raise an instance of `ValueError` if an attempt is made to use them for Set-Cookie: *set_header()*, *delete_header()*, *get_header()*, *set_headers()*.

- *falcon.testing.Result.json* now returns `None` when the response body is empty, rather than raising an error.
- *get_param_as_bool()* now defaults to treating valueless parameters as truthy, rather than falsy. `None` is still returned by default when the parameter is altogether missing.
- *get_param_as_bool()* no longer raises an error for a valueless parameter when the `blank_as_true` keyword argument is `False`. Instead, `False` is simply returned in that case.
- *keep_blank_qs_values* now defaults to `True` instead of `False`.
- *auto_parse_qs_csv* now defaults to `False` instead of `True`.
- `independent_middleware` kwarg on `falcon.API` now defaults to `True` instead of `False`.
- The `stream_len` property of the *Response* class was changed to be an alias of the new *content_length* property. Please use *set_stream()* or *content_length* instead, going forward, as `stream_len` is now deprecated.
- Request *context_type* was changed from dict to a bare class implementing the mapping interface. (See also: *How do I adapt my code to default context type changes in Falcon 2.0?*)
- Response *context_type* was changed from dict to a bare class implementing the mapping interface. (See also: *How do I adapt my code to default context type changes in Falcon 2.0?*)
- *JSONHandler* and *HTTPError* no longer use *ujson* in lieu of the standard *json* library (when *ujson* is available in the environment). Instead, *JSONHandler* can now be configured to use arbitrary `dumps()` and `loads()` functions. If you also need to customize *HTTPError* serialization, you can do so via `set_error_serializer()`.
- The `find()` method for a custom router is now required to accept the `req` keyword argument that was added in a previous release. The backwards-compatible shim was removed.
- All *middleware* methods and *hooks* must now accept the arguments as specified in the relevant interface definitions as of Falcon 2.0. All backwards-compatible shims have been removed.
- Custom error serializers are now required to accept the arguments as specified by `set_error_serializer()` for the past few releases. The backwards-compatible shim has been removed.
- An internal function, `make_router_search()`, was removed from the `api_helpers` module.
- An internal function, `wrap_old_error_serializer()`, was removed from the `api_helpers` module.
- In order to improve performance, the *falcon.Request.headers* and *falcon.Request.cookies* properties now return a direct reference to an internal cached object, rather than making a copy each time. This should normally not cause any problems with existing apps since these objects are generally treated as read-only by the caller.
- The *falcon.Request.stream* attribute is no longer wrapped in a bounded stream when Falcon detects that it is running on the wsgiref server. If you need to normalize stream semantics between wsgiref and a production WSGI server, *bounded_stream* may be used instead.
- *falcon.Request.cookies* now gives precedence to the first value encountered in the Cookie header for a given cookie name, rather than the last.
- The ordering of the parameters passed to custom error handlers was adjusted to be more intuitive and consistent with the rest of the framework:

```
# Before
def handle_error(ex, req, resp, params):
  pass

# Falcon 2.0
def handle_error(req, resp, ex, params):
  pass
```

See also: `add_error_handler()`

- *`strip_url_path_trailing_slash`* now defaults to `False` instead of `True`.
- The deprecated `falcon.testing.TestCase.api` property was removed.
- The deprecated `falcon.testing.TestCase.api_class` class variable was removed.
- The deprecated `falcon.testing.TestBase` class was removed.
- The deprecated `falcon.testing.TestResource` class was removed.
- The deprecated `protocol` property was removed from the *`Request`* class.
- The deprecated `get_param_as_dict()` method alias was removed from the *`Request`* class. Please use *`get_param_as_json()`* instead.
- Routers were previously allowed to accept additional args and keyword arguments, and were not required to use the variadic form. Now, they are only allowed to accept additional options as variadic keyword arguments, and to ignore any arguments they don't support. This helps overridden router logic be less fragile in terms of their interface contracts, which also makes it easier to keep Falcon backwards-compatible in the face of any future changes in this area.
- `add_route()` previously accepted **args*, but now no longer does.
- The `add_route()` method for custom routers no longer takes a *method_map* argument. Custom routers should, instead, call the *`map_http_methods()`* function directly from their `add_route()` method if they require this mapping.
- The `serialize()` media handler method now receives an extra *content_type* argument, while the `deserialize()` method now takes *stream*, *content_type*, and *content_length* arguments, rather than a single *raw* argument. The raw data can still be obtained by executing `raw = stream.read()`.

  See also: *`BaseHandler`*
- The deprecated `falcon.routing.create_http_method_map()` method was removed.
- The keyword arguments for *`parse_query_string()`* were renamed to be more concise:

```
# Before
parsed_values = parse_query_string(
    query_string, keep_blank_qs_values=True, parse_qs_csv=False
)

# Falcon 2.0
parsed_values = parse_query_string(
    query_string, keep_blank=True, csv=False
)
```

- *`auto_parse_qs_csv`* now defaults to `False` instead of `True`.
- The `HTTPRequestEntityTooLarge` class was renamed to `HTTPPayloadTooLarge`.
- Two of the keyword arguments for *`get_param_as_int()`* were renamed to avoid shadowing built-in Python names:

```
# Before
dpr = req.get_param_as_int('dpr', min=0, max=3)

# Falcon 2.0
dpr = req.get_param_as_int('dpr', min_value=0, max_value=3)
```

- The *`falcon.media.validators.jsonschema.validate()`* decorator now uses `functools.wraps()` to make the decorated method look like the original.

- Previously, *HTTPError* instances for which the *has_representation* property evaluated to `False` were not passed to custom error serializers (such as in the case of types that subclass `NoRepresentation`). This has now been fixed so that custom error serializers will be called for all instances of *HTTPError*.
- Request cookie parsing no longer uses the standard library for most of the parsing logic. This may lead to subtly different results for archaic cookie header formats, since the new implementation is based on RFC 6265.
- The *if_match* and *if_none_match* properties now return a list of *falcon.ETag* objects rather than the raw value of the If-Match or If-None-Match headers, respectively.
- When setting the *etag* header property, the value will now be wrapped with double-quotes (if not already present) to ensure compliance with RFC 7232.
- The default error serializer no longer sets the *charset* parameter for the media type returned in the Content-Type header, since UTF-8 is the default encoding for both JSON and XML media types. This should not break well-behaved clients, but could impact test cases in apps that assert on the exact value of the Content-Type header.
- Similar to the change made to the default error serializer, the default JSON media type generally used for successful responses was also modified to no longer specify the *charset* parameter. This change affects both the `falcon.DEFAULT_MEDIA_TYPE` and `falcon.MEDIA_JSON` *constants*, as well as the default value of the *media_type* keyword argument specified for the `falcon.API` initializer. This change also affects the default value of the *RequestOptions.default_media_type* and *ResponseOptions.default_media_type* options.

New & Improved

- Several performance optimizations were made to hot code paths in the framework to make Falcon 2.0 even faster than 1.4 in some cases.
- Numerous changes were made to the docs to improve clarity and to provide better recommendations on how to best use various parts of the framework.
- Added a new *headers* property to the *Response* class.
- Removed the `six` and `python-mimeparse` dependencies.
- Added a new *complete* property to the *Response* class. This can be used to short-circuit request processing when the response has been pre-constructed.
- Request *context_type* now defaults to a bare class allowing to set attributes on the request context object:

```
# Before
req.context['role'] = 'trial'
req.context['user'] = 'guest'

# Falcon 2.0
req.context.role = 'trial'
req.context.user = 'guest'
```

  To ease the migration path, the previous behavior is supported by implementing the mapping interface in a way that object attributes and mapping items are linked, and setting one sets the other as well. However, as of Falcon 2.0, the dict context interface is considered deprecated, and may be removed in a future release.

  Applications can work around this change by explicitly overriding *context_type* to dict. (See also: *How do I adapt my code to default context type changes in Falcon 2.0?*)

- Response *context_type* now defaults to a bare class allowing to set attributes on the response context object:

```
# Before
resp.context['cache_strategy'] = 'lru'

# Falcon 2.0
resp.context.cache_strategy = 'lru'
```

To ease the migration path, the previous behavior is supported by implementing the mapping interface in a way that object attributes and mapping items are linked, and setting one sets the other as well. However, as of Falcon 2.0, the dict context interface is considered deprecated, and may be removed in a future release.

Applications can work around this change by explicitly overriding *context_type* to dict. (See also: *How do I adapt my code to default context type changes in Falcon 2.0?*)

- *JSONHandler* can now be configured to use arbitrary `dumps()` and `loads()` functions. This enables support not only for using any of a number of third-party JSON libraries, but also for customizing the keyword arguments used when (de)serializing objects.
- Added a new method, *get_cookie_values()*, to the *Request* class. The new method supports getting all values provided for a given cookie, and is now the preferred mechanism for reading request cookies.
- Optimized request cookie parsing. It is now roughly an order of magnitude faster.
- *append_header()* now supports appending raw Set-Cookie header values.
- Multiple routes can now be added for the same resource instance using a suffix to distinguish the set of responders that should be used. In this way, multiple closely-related routes can be mapped to the same resource while preserving readability and consistency.

  See also: `add_route()`
- The *falcon.media.validators.jsonschema.validate()* decorator now supports both request and response validation.
- A static route can now be configured to return the data from a default file when the requested file path is not found.

  See also: `add_static_route()`
- The ordering of the parameters passed to custom error handlers was adjusted to be more intuitive and consistent with the rest of the framework:

```
# Before
def handle_error(ex, req, resp, params):
  pass

# Falcon 2.0
def handle_error(req, resp, ex, params):
  pass
```

  See also: `add_error_handler()`.
- All error classes now accept a *headers* keyword argument for customizing response headers.
- A new method, *get_param_as_float()*, was added to the *Request* class.
- A new method, *has_param()*, was added to the *Request* class.
- A new property, *content_length*, was added to the *Response* class. Either *set_stream()* or *content_length* should be used going forward, as `stream_len` is now deprecated.
- All `get_param_*()` methods of the *Request* class now accept a *default* argument.
- A new header property, *expires*, was added to the *Response* class.
- The *CompiledRouter* class now exposes a *map_http_methods* method that child classes can override in order to customize the mapping of HTTP methods to resource class methods.
- The `serialize()` media handler method now receives an extra *content_type* argument, while the `deserialize()` method now takes *stream*, *content_type*, and *content_length* arguments, rather than a single *raw* argument. The raw data can still be obtained by executing `raw = stream.read()`.

  See also: *BaseHandler*
- The *get_header()* method now accepts a *default* keyword argument.

- The *simulate_request()* method now supports overriding the host and remote IP address in the WSGI environment, as well as setting arbitrary additional CGI variables in the WSGI environment.
- The *simulate_request()* method now supports passing a query string as part of the path, as an alternative to using the *params* or *query_string* keyword arguments.
- Added a deployment guide to the docs for uWSGI and NGINX on Linux.
- The `decode()` method now accepts an *unquote_plus* keyword argument. The new argument defaults to `False` to avoid a breaking change.
- The *if_match()* and *if_none_match()* properties now return a list of *falcon.ETag* objects rather than the raw value of the If-Match or If-None-Match headers, respectively.
- `add_error_handler()` now supports specifying an iterable of exception types to match.
- The default error serializer no longer sets the *charset* parameter for the media type returned in the Content-Type header, since UTF-8 is the default encoding for both JSON and XML media types.
- Similar to the change made to the default error serializer, the default JSON media type generally used for successful responses was also modified to no longer specify the *charset* parameter. This change affects both the `falcon.DEFAULT_MEDIA_TYPE` and `falcon.MEDIA_JSON` *constants*, as well as the default value of the *media_type* keyword argument specified for the `falcon.API` initializer. This change also affects the default value of the *RequestOptions.default_media_type* and *ResponseOptions.default_media_type* options.

Fixed

- Fixed a docs issue where with smaller browser viewports, the API documentation will start horizontal scrolling.
- The color scheme for the docs was modified to fix issues with contrast and readability when printing the docs or generating PDFs.
- The *simulate_request()* method now forces header values to *str* on Python 2 as required by PEP-3333.
- The `HTTPRequestEntityTooLarge` class was renamed to `HTTPPayloadTooLarge` and the reason phrase was updated per RFC 7231.
- The `falcon.CaseInsensitiveDict` class now inherits from `collections.abc.MutableMapping` under Python 3, instead of `collections.MutableMapping`.
- The `\ufffd` character is now disallowed in requested static file paths.
- The *falcon.media.validators.jsonschema.validate()* decorator now uses `functools.wraps()` to make the decorated method look like the original.
- The `falcon-print-routes` CLI tool no longer raises an unhandled error when Falcon is cythonized.
- The plus character (`'+'`) is no longer unquoted in the request path, but only in the query string.
- Previously, *HTTPError* instances for which the *has_representation* property evaluated to `False` were not passed to custom error serializers (such as in the case of types that subclass `NoRepresentation`). This has now been fixed so that custom error serializers will be called for all instances of *HTTPError*.
- When setting the *etag* header property, the value will now be wrapped with double-quotes (if not already present) to ensure compliance with RFC 7232.
- Fixed `TypeError` being raised when using Falcon's testing framework to simulate a request to a generator-based WSGI app.

Contributors to this Release

Many thanks to all of our talented and stylish contributors for this release!

- Bertrand Lemasle
- CaselIT
- DmitriiTrofimov

- KingAkeem
- Nateyo
- Patrick Schneeweis
- TheMushrr00m
- ZDBioHazard
- alysivji
- aparkerlue
- astonm
- awbush
- bendemaree
- bkcsfi
- brooksryba
- carlodri
- grktsh
- hugovk
- jmvrbanac
- kandziu
- kgriffs
- klardotsh
- mikeylight
- mumrau
- nZac
- navyad
- ozzzik
- paneru-rajan
- safaozturk93
- santeyio
- sbensoussan
- selfvin
- snobu
- steven-upside
- tribals
- vytas7

5.4.13 Changelog for Falcon 1.4.1

Breaking Changes

(None)

Changes to Supported Platforms

(None)

New & Improved

(None)

Fixed

- Reverted the breaking change in 1.4.0 to *falcon.testing.Result.json*. Minor releases should have no breaking changes.
- The README was not rendering properly on PyPI. This was fixed and a validation step was added to the build process.

5.4.14 Changelog for Falcon 1.4.0

Breaking Changes

- *falcon.testing.Result.json* now returns None when the response body is empty, rather than raising an error.

Changes to Supported Platforms

- Python 3 is now supported on PyPy as of PyPy3.5 v5.10.
- Support for CPython 3.3 is now deprecated and will be removed in Falcon 2.0.
- As with the previous release, Python 2.6 and Jython 2.7 remain deprecated and will no longer be supported in Falcon 2.0.

New & Improved

- We added a new method, `add_static_route()`, that makes it easy to serve files from a local directory. This feature provides an alternative to serving files from the web server when you don't have that option, when authorization is required, or for testing purposes.
- Arguments can now be passed to hooks (see *Hooks*).
- The default JSON media type handler will now use ujson, if available, to speed up JSON (de)serialization under CPython.
- Semantic validation via the *format* keyword is now enabled for the *validate()* JSON Schema decorator.
- We added a new helper, *get_param_as_uuid()*, to the *Request* class.
- Falcon now supports WebDAV methods (RFC 3253), such as UPDATE and REPORT.
- We added a new property, *downloadable_as*, to the *Response* class for setting the Content-Disposition header.
- `create_http_method_map()` has been refactored into two new methods, *map_http_methods()* and *set_default_responders()*, so that custom routers can better pick and choose the functionality they need. The original method is still available for backwards-compatibility, but will be removed in a future release.
- We added a new *json* param to *simulate_request()* et al. to automatically serialize the request body from a JSON serializable object or type (for a complete list of serializable types, see json.JSONEncoder).
- *TestClient*'s `simulate_*()` methods now call *simulate_request()* to make it easier for subclasses to override *TestClient*'s behavior.
- *TestClient* can now be configured with a default set of headers to send with every request.
- The *FAQ* has been reorganized and greatly expanded.

- We restyled the docs to match https://falconframework.org

Fixed

- Forwarded headers containing quoted strings with commas were not being parsed correctly. This has been fixed, and the parser generally made more robust.
- *JSONHandler* was raising an error under Python 2.x when serializing strings containing Unicode code points. This issue has been fixed.
- Overriding a resource class and calling its responders via `super()` did not work when passing URI template params as positional arguments. This has now been fixed.
- Python 3.6 was generating warnings for strings containing `'\s'` within Falcon. These strings have been converted to raw strings to mitigate the warning.
- Several syntax errors were found and fixed in the code examples used in the docs.

Contributors to this Release

Many thanks to all of our talented and stylish contributors for this release!

- GriffGeorge
- hynek
- kgriffs
- rhemz
- santeyio
- timc13
- tyronegroves
- vytas7
- zhanghanyun

5.4.15 Changelog for Falcon 1.3.0

Breaking Changes

(None)

Changes to Supported Platforms

- CPython 3.6 is now fully supported.
- Falcon appears to work well on PyPy3.5, but we are waiting until that platform is out of beta before officially supporting it.
- Support for both CPython 2.6 and Jython 2.7 is now deprecated and will be discontinued in Falcon 2.0.

New & Improved

- We added built-in resource representation serialization and deserialization, including input validation based on JSON Schema. (See also: *Media*)
- URI template field converters are now supported. We expect to expand this feature over time. (See also: *Field Converters*)
- A new method, *get_param_as_datetime()*, was added to *Request*.
- A number of attributes were added to *Request* to make proxy information easier to consume. These include the *forwarded*, *forwarded_uri*, *forwarded_scheme*, *forwarded_host*, and *forwarded_prefix* attributes. The *prefix* attribute was also added as part of this work.

- A *referer* attribute was added to *Request*.
- We implemented `__repr__()` for *Request*, *Response*, and *HTTPError* to aid in debugging.
- A number of Internet media type constants were defined to make it easier to check and set content type headers. (See also: *Media Type Constants*)
- Several new 5xx error classes were implemented. (See also: *Error Handling*)

Fixed

- If even a single cookie in the request to the server is malformed, none of the cookies will be parsed (all-or-nothing). Change the parser to simply skip bad cookies (best-effort).
- `API` instances are not pickleable. Modify the default router to fix this.

5.4.16 Changelog for Falcon 1.2.0

Breaking Changes

(None)

New & Improved

- A new *default* kwarg was added to *get_header()*.
- A *delete_header()* method was added to *falcon.Response*.
- Several new HTTP status codes and error classes were added, such as *falcon.HTTPFailedDependency*.
- If *ujson* is installed it will be used in lieu of *json* to speed up error serialization and query string parsing under CPython. PyPy users should continue to use *json*.
- The *independent_middleware* kwarg was added to `falcon.API` to enable the execution of *process_response()* middleware methods, even when *process_request()* raises an error.
- Single-character field names are now allowed in URL templates when specifying a route.
- A detailed error message is now returned when an attempt is made to add a route that conflicts with one that has already been added.
- The HTTP protocol version can now be specified when simulating requests with the testing framework.
- The *falcon.ResponseOptions* class was added, along with a *secure_cookies_by_default* option to control the default value of the "secure" attribute when setting cookies. This can make testing easier by providing a way to toggle whether or not HTTPS is required.
- *port*, *netloc* and *scheme* properties were added to the *falcon.Request* class. The *protocol* property is now deprecated and will be removed in a future release.
- The *strip_url_path_trailing_slash* was added to *falcon.RequestOptions* to control whether or not to retain the trailing slash in the URL path, if one is present. When this option is enabled (the default), the URL path is normalized by stripping the trailing slash character. This lets the application define a single route to a resource for a path that may or may not end in a forward slash. However, this behavior can be problematic in certain cases, such as when working with authentication schemes that employ URL-based signatures. Therefore, the *strip_url_path_trailing_slash* option was introduced to make this behavior configurable.
- Improved the documentation for *falcon.HTTPError*, particularly around customizing error serialization.
- Misc. improvements to the look and feel of Falcon's documentation.
- The tutorial in the docs was revamped, and now includes guidance on testing Falcon applications.

Fixed

- Certain non-alphanumeric characters, such as parenthesis, are not handled properly in complex URI template path segments that are comprised of both literal text and field definitions.
- When the WSGI server does not provide a *wsgi.file_wrapper* object, Falcon wraps `Response.stream` in a simple iterator object that does not implement *close()*. The iterator should be modified to implement a *close()* method that calls the underlying stream's *close()* to free system resources.
- The testing framework does not correctly parse cookies under Jython.
- Whitespace is not stripped when parsing cookies in the testing framework.
- The Vary header is not always set by the default error serializer.
- While not specified in PEP-3333 that the status returned to the WSGI server must be of type *str*, setting the status on the response to a *unicode* string under Python 2.6 or 2.7 can cause WSGI servers to raise an error. Therefore, the status string must first be converted if it is of the wrong type.
- The default OPTIONS responder returns 204, when it should return 200. RFC 7231 specifically states that Content-Length should be zero in the response to an OPTIONS request, which implies a status code of 200 since RFC 7230 states that Content-Length must not be set in any response with a status code of 204.

5.4.17 Changelog for Falcon 1.1.0

Breaking Changes

(None)

New & Improved

- A new *bounded_stream* property was added to *falcon.Request* that can be used in place of the *stream* property to mitigate the blocking behavior of input objects used by some WSGI servers.
- A new *uri_template* property was added to *Request* to expose the template for the route corresponding to the path requested by the user agent.
- A *context* property was added to *Response* to mirror the same property that is already available for *Request*.
- JSON-encoded query parameter values can now be retrieved and decoded in a single step via `get_param_as_dict()`.
- CSV-style parsing of query parameter values can now be disabled.
- *get_param_as_bool()* now recognizes "on" and "off" in support of IE's default checkbox values.
- An *accept_ranges* property was added to *Response* to facilitate setting the Accept-Ranges header.
- Added the *HTTPUriTooLong* and *HTTPGone* error classes.
- When a title is not specified for *HTTPError*, it now defaults to the HTTP status text.
- All parameters are now optional for most error classes.
- Cookie-related documentation has been clarified and expanded
- The *falcon.testing.Cookie* class was added to represent a cookie returned by a simulated request. *falcon.testing.Result* now exposes a *cookies* attribute for examining returned cookies.
- pytest support was added to Falcon's testing framework. Apps can now choose to either write unittest- or pytest-style tests.
- The test runner for Falcon's own tests was switched from nose to pytest.
- When simulating a request using Falcon's testing framework, query string parameters can now be specified as a `dict`, as an alternative to passing a raw query string.
- A flag is now passed to the *process_request* middleware method to signal whether or not an exception was raised while processing the request. A shim was added to avoid breaking existing middleware methods that do not yet accept this new parameter.

- A new CLI utility, *falcon-print-routes*, was added that takes in a module:callable, introspects the routes, and prints the results to stdout. This utility is automatically installed along with the framework:

```
$ falcon-print-routes commissaire:api
-> /api/v0/status
-> /api/v0/cluster/{name}
-> /api/v0/cluster/{name}/hosts
-> /api/v0/cluster/{name}/hosts/{address}
```

- Custom attributes can now be attached to instances of *Request* and *Response*. This can be used as an alternative to adding values to the *context* property, or implementing custom subclasses.
- `get_http_status()` was implemented to provide a way to look up a full HTTP status line, given just a status code.

Fixed

- When *auto_parse_form_urlencoded* is set to `True`, the framework now checks the HTTP method before attempting to consume and parse the body.
- Before attempting to read the body of a form-encoded request, the framework now checks the Content-Length header to ensure that a non-empty body is expected. This helps prevent bad requests from causing a blocking read when running behind certain WSGI servers.
- When the requested method is not implemented for the target resource, the framework now raises *HTTPMethodNotAllowed*, rather than modifying the *Request* object directly. This improves visibility for custom error handlers and for middleware methods.
- Error class docstrings have been updated to reflect the latest RFCs.
- When an error is raised by a resource method or a hook, the error will now always be processed (including setting the appropriate properties of the *Response* object) before middleware methods are called.
- A case was fixed in which middleware processing did not continue when an instance of *HTTPError* or *HTTPStatus* was raised.
- The *encode()* method will now attempt to detect whether the specified string has already been encoded, and return it unchanged if that is the case.
- The default OPTIONS responder now explicitly sets Content-Length to zero in the response.
- *falcon.testing.Result* now assumes that the response body is encoded as UTF-8 when the character set is not specified, rather than raising an error when attempting to decode the response body.
- When simulating requests, Falcon's testing framework now properly tunnels Unicode characters through the WSGI interface.
- `import falcon.uri` now works, in addition to `from falcon import uri`.
- URI template fields are now validated up front, when the route is added, to ensure they are valid Python identifiers. This prevents cryptic errors from being raised later on when requests are routed.
- When running under Python 3, `inspect.signature()` is used instead of `inspect.getargspec()` to provide compatibility with annotated functions.

5.4.18 Changelog for Falcon 1.0.0

Breaking Changes

- The deprecated global hooks feature has been removed. `API` no longer accepts *before* and *after* kwargs. Applications can work around this by migrating any logic contained in global hooks to reside in middleware components instead.
- The middleware method `process_resource()` must now accept an additional *params* argument. This gives the middleware method an opportunity to interact with the values for any fields defined in a route's URI template.

- The middleware method `process_resource()` is now skipped when no route is found for the incoming request. This avoids having to include an `if resource is not None` check when implementing this method. A sink may be used instead to execute logic in the case that no route is found.
- An option was added to toggle automatic parsing of form params. Falcon will no longer automatically parse, by default, requests that have the content type "application/x-www-form-urlencoded". This was done to avoid unintended side-effects that may arise from consuming the request stream. It also makes it more straightforward for applications to customize and extend the handling of form submissions. Applications that require this functionality must re-enable it explicitly, by setting a new request option that was added for that purpose, per the example below:

```
app = falcon.API()
app.req_options.auto_parse_form_urlencoded = True
```

- The *HTTPUnauthorized* initializer now requires an additional argument, *challenges*. Per RFC 7235, a server returning a 401 must include a WWW-Authenticate header field containing at least one challenge.
- The performance of composing the response body was improved. As part of this work, the `Response.body_encoded` attribute was removed. This property was only intended to be used by the framework itself, but any dependent code can be migrated per the example below:

```
# Before
body = resp.body_encoded

# After
if resp.body:
    body = resp.body.encode('utf-8')
else:
    body = b''
```

New & Improved

- A code of conduct was added to solidify our community's commitment to sustaining a welcoming, respectful culture.
- CPython 3.5 is now fully supported.
- The constants HTTP_422, HTTP_428, HTTP_429, HTTP_431, HTTP_451, and HTTP_511 were added.
- The *HTTPUnprocessableEntity*, *HTTPTooManyRequests*, and *HTTPUnavailableForLegalReasons* error classes were added.
- The `HTTPStatus` class is now available directly under the *falcon* module, and has been properly documented.
- Support for HTTP redirections was added via a set of `HTTPStatus` subclasses. This should avoid the problem of hooks and responder methods possibly overriding the redirect. Raising an instance of one of these new redirection classes will short-circuit request processing, similar to raising an instance of *HTTPError*.
- The default 404 responder now raises an instance of *HTTPError* instead of manipulating the response object directly. This makes it possible to customize the response body using a custom error handler or serializer.
- A new method, *get_header()*, was added to *Response*. Previously there was no way to check if a header had been set. The new *get_header()* method facilitates this and other use cases.
- *falcon.Request.client_accepts_msgpack()* now recognizes "application/msgpack", in addition to "application/x-msgpack".
- New *access_route* and *remote_addr* properties were added to *Request* for getting upstream IP addresses.
- *Request* and *Response* now support range units other than bytes.
- The `API` and *StartResponseMock* class types can now be customized by inheriting from `TestBase` and overriding the *api_class* and *srmock_class* class attributes.

- Path segments with multiple field expressions may now be defined at the same level as path segments having only a single field expression. For example:

```
api.add_route('/files/{file_id}', resource_1)
api.add_route('/files/{file_id}.{ext}', resource_2)
```

- Support was added to `API.add_route()` for passing through additional args and kwargs to custom routers.
- Digits and the underscore character are now allowed in the `falcon.routing.compile_uri_template()` helper, for use in custom router implementations.
- A new testing framework was added that should be more intuitive to use than the old one. Several of Falcon's own tests were ported to use the new framework (the remainder to be ported in a subsequent release.) The new testing framework performs wsgiref validation on all requests.
- The performance of setting `Response.content_range` was improved by ~50%.
- A new param, *obs_date*, was added to *falcon.Request.get_header_as_datetime()*, and defaults to `False`. This improves the method's performance when obsolete date formats do not need to be supported.

Fixed

- Field expressions at a given level in the routing tree no longer mask alternative branches. When a single segment in a requested path can match more than one node at that branch in the routing tree, and the first branch taken happens to be the wrong one (i.e., the subsequent nodes do not match, but they would have under a different branch), the other branches that could result in a successful resolution of the requested path will now be subsequently tried, whereas previously the framework would behave as if no route could be found.
- The user agent is now instructed to expire the cookie when it is cleared via *unset_cookie()*.
- Support was added for hooks that have been defined via `functools.partial()`.
- Tunneled UTF-8 characters in the request path are now properly decoded, and a placeholder character is substituted for any invalid code points.
- The instantiation of *context_type* is now delayed until after all other properties of the *Request* class have been initialized, in case the context type's own initialization depends on any of *Request*'s properties.
- A case was fixed in which reading from *stream* could hang when using `wsgiref` to host the app.
- The default error serializer now sets the Vary header in responses. Implementing this required passing the *Response* object to the serializer, which would normally be a breaking change. However, the framework was modified to detect old-style error serializers and wrap them with a shim to make them compatible with the new interface.
- A query string containing malformed percent-encoding no longer causes the framework to raise an error.
- Additional tests were added for a few lines of code that were previously not covered, due to deficiencies in code coverage reporting that have since been corrected.
- The Cython note is no longer displayed when installing under Jython.
- Several errors and ambiguities in the documentation were corrected.

5.4.19 Changelog for Falcon 0.3.0

Breaking Changes

- Date headers are now returned as `datetime.datetime` objects instead of strings.
- The expected signature for the `add_route()` method of custom routers no longer includes a *method_map* parameter. Custom routers should, instead, call the `falcon.routing.util.map_http_methods()` function directly from their `add_route()` method if they require this mapping.

New & Improved

- This release includes a new router architecture for improved performance and flexibility.
- A custom router can now be specified when instantiating the `API` class.
- URI templates can now include multiple parameterized fields within a single path segment.
- Falcon now supports reading and writing cookies.
- Falcon now supports Jython 2.7.
- A method for getting a query param as a date was added to the `Request` class.
- Date headers are now returned as `datetime.datetime` objects.
- A default value can now be specified when calling `Request.get_param()`. This provides an alternative to using the pattern:

```
value = req.get_param(name) or default_value
```

- Friendly constants for status codes were added (e.g., `falcon.HTTP_NO_CONTENT` vs. `falcon.HTTP_204`.)
- Several minor performance optimizations were made to the code base.

Fixed

- The query string parser was modified to improve handling of percent-encoded data.
- Several errors in the documentation were corrected.
- The `six` package was pinned to 1.4.0 or better. `six.PY2` is required by Falcon, but that wasn't added to `six` until version 1.4.0.

5.4.20 Changelog for Falcon 0.2.0

Breaking Changes

- The deprecated util.misc.percent_escape and util.misc.percent_unescape functions were removed. Please use the functions in the util.uri module instead.
- The deprecated function, API.set_default_route, was removed. Please use sinks instead.
- HTTPRangeNotSatisfiable no longer accepts a media_type parameter.
- When using the comma-delimited list convention, req.get_param_as_list(…) will no longer insert placeholders, using the None type, for empty elements. For example, where previously the query string "foo=1,,3" would result in ['1', None, '3'], it will now result in ['1', '3'].

New & Improved

- Since 0.1 we've added proper RTD docs to make it easier for everyone to get started with the framework. Over time we will continue adding content, and we would love your help!
- Falcon now supports "wsgi.filewrapper". You can assign any file-like object to resp.stream and Falcon will use "wsgi.filewrapper" to more efficiently pipe the data to the WSGI server.
- Support was added for automatically parsing requests containing "application/x-www-form-urlencoded" content. Form fields are now folded into req.params.
- Custom Request and Response classes are now supported. You can specify custom types when instantiating falcon.API.
- A new middleware feature was added to the framework. Middleware deprecates global hooks, and we encourage everyone to migrate as soon as possible.
- A general-purpose dict attribute was added to Request. Middleware, hooks, and responders can now use req.context to share contextual information about the current request.

- A new method, append_header, was added to falcon.API to allow setting multiple values for the same header using comma separation. Note that this will not work for setting cookies, but we plan to address this in the next release (0.3).
- A new "resource" attribute was added to hooks. Old hooks that do not accept this new attribute are shimmed so that they will continue to function. While we have worked hard to minimize the performance impact, we recommend migrating to the new function signature to avoid any overhead.
- Error response bodies now support XML in addition to JSON. In addition, the HTTPError serialization code was refactored to make it easier to implement a custom error serializer.
- A new method, "set_error_serializer" was added to falcon.API. You can use this method to override Falcon's default HTTPError serializer if you need to support custom media types.
- Falcon's testing base class, testing.TestBase was improved to facilitate Py3k testing. Notably, TestBase.simulate_request now takes an additional "decode" kwarg that can be used to automatically decode bytestring PEP-3333 response bodies.
- An "add_link" method was added to the Response class. Apps can use this method to add one or more Link header values to a response.
- Added two new properties, req.host and req.subdomain, to make it easier to get at the hostname info in the request.
- Allow a wider variety of characters to be used in query string params.
- Internal APIs have been refactored to allow overriding the default routing mechanism. Further modularization is planned for the next release (0.3).
- Changed req.get_param so that it behaves the same whether a list was specified in the query string using the HTML form style (in which each element is listed in a separate 'key=val' field) or in the more compact API style (in which each element is comma-separated and assigned to a single param instance, as in 'key=val1,val2,val3')
- Added a convenience method, set_stream(…), to the Response class for setting the stream and its length at the same time, which should help people not forget to set both (and save a few keystrokes along the way).
- Added several new error classes, including HTTPRequestEntityTooLarge, HTTPInvalidParam, HTTPMissingParam, HTTPInvalidHeader and HTTPMissingHeader.
- Python 3.4 is now fully supported.
- Various minor performance improvements

Fixed

- Ensure 100% test coverage and fix any bugs identified in the process.
- Fix not recognizing the "bytes=" prefix in Range headers.
- Make HTTPNotFound and HTTPMethodNotAllowed fully compliant, according to RFC 7231.
- Fixed the default on_options responder causing a Cython type error.
- URI template strings can now be of type unicode under Python 2.
- When SCRIPT_NAME is not present in the WSGI environ, return an empty string for the req.app property.
- Global "after" hooks will now be executed even when a responder raises an error.
- Fixed several minor issues regarding testing.create_environ(…)
- Work around a wsgiref quirk, where if no content-length header is submitted by the client, wsgiref will set the value of that header to an empty string in the WSGI environ.
- Resolved an issue causing several source files to not be Cythonized.
- Docstrings have been edited for clarity and correctness.

5.5 Deployment Guide

5.5.1 Preamble & Disclaimer

Falcon conforms to the standard WSGI protocol that most Python web applications have been using since 2003. If you have deployed Python applications like Django, Flask, or others, you will find yourself quite at home with Falcon and your standard Apache/mod_wsgi, gunicorn, or other WSGI servers should suffice.

There are many ways to deploy a Python application. The aim of these quickstarts is to simply get you up and running, not to give you a perfectly tuned or secure environment. You will almost certainly need to customize these configurations for any serious production deployment.

5.5.2 Deploying Falcon on Linux with NGINX and uWSGI

NGINX is a powerful web server and reverse proxy and uWSGI is a fast and highly-configurable WSGI application server. Together, NGINX and uWSGI create a one-two punch of speed and functionality which will suffice for most applications. In addition, this stack provides the building blocks for a horizontally-scalable and highly-available (HA) production environment and the configuration below is just a starting point.

This guide provides instructions for deploying to a Linux environment only. However, with a bit of effort you should be able to adapt this configuration to other operating systems, such as OpenBSD.

Running your Application as a Different User

It is best to execute the application as a different OS user than the one who owns the source code for your application. The application user should *NOT* have write access to your source. This mitigates the chance that someone could write a malicious Python file to your source directory through an upload endpoint you might define; when your application restarts, the malicious file is loaded and proceeds to cause any number of Bad Things™ to happen.

```
$ useradd myproject --create-home
$ useradd myproject-runner --no-create-home
```

It is helpful to switch to the project user (myproject) and use the home directory as the application environment.

If you are working on a remote server, switch to the myproject user and pull down the source code for your application.

```
$ git clone git@github.com/myorg/myproject.git /home/myproject/src
```

Note

You could use a tarball, zip file, scp or any other means to get your source onto a server.

Next, create a virtual environment which can be used to install your dependencies.

```
$ python3 -m venv /home/myproject/venv
```

Then install your dependencies.

```
$ /home/myproject/venv/bin/pip install -r /home/myproject/src/requirements.txt
$ /home/myproject/venv/bin/pip install -e /home/myproject/src
$ /home/myproject/venv/bin/pip install uwsgi
```

Note

The exact commands for creating a virtual environment might differ based on the Python version you are using and your operating system. At the end of the day the application needs a virtualenv in /home/myproject/venv with the project dependencies installed. Use the `pip` binary within the virtual environment by `source venv/bin/activate` or using the full path.

Preparing your Application for Service

For the purposes of this tutorial, we'll assume that you have implemented a way to configure your application, such as with a `create_app()` function or a module-level script. The role of this function or script is to supply an instance of *falcon.App*, which implements the standard WSGI callable interface.

You will need to expose the *falcon.App* instance in some way so that uWSGI can find it. For this tutorial we recommend creating a `wsgi.py` file. Modify the logic of the following example file to properly configure your application. Ensure that you expose a variable called `application` which is assigned to your *falcon.App* instance.

Listing 5: /home/myproject/src/wsgi.py

```
import os
import myproject

# Replace with your app's method of configuration
config = myproject.get_config(os.environ['MYPROJECT_CONFIG'])

# uWSGI will look for this variable
application = myproject.create_app(config)
```

Note that in the above example, the WSGI callable is simple assigned to a variable, `application`, rather than being passed to a self-hosting WSGI server such as *wsgiref.simple_server.make_server*. Starting an independent WSGI server in your `wsgi.py` file will render unexpected results.

Deploying Falcon behind uWSGI

With your `wsgi.py` file in place, it is time to configure uWSGI. Start by creating a simple `uwsgi.ini` file. In general, you shouldn't commit this file to source control; it should be generated from a template by your deployment toolchain according to the target environment (number of CPUs, etc.).

This configuration, when executed, will create a new uWSGI server backed by your `wsgi.py` file and listening at `127.0.0.1:8080`.

Listing 6: /home/myproject/src/uwsgi.ini

```
[uwsgi]
master = 1
vacuum = true
socket = 127.0.0.1:8080
enable-threads = true
thunder-lock = true
threads = 2
processes = 2
virtualenv = /home/myproject/venv
wsgi-file = /home/myproject/src/wsgi.py
chdir = /home/myproject/src
uid = myproject-runner
gid = myproject-runner
```

Note

Threads vs. Processes

There are many questions to consider when deciding how to manage the processes that actually run your Python code. Are you generally CPU bound or IO bound? Is your application thread-safe? How many CPU's do you have? What system are you on? Do you need an in-process cache?

The configuration presented here enables both threads and processes. However, you will have to experiment and do some research to understand your application's unique requirements, and then tailor your uWSGI configuration

accordingly. Generally speaking, uWSGI is flexible enough to support most types of applications.

Note

TCP vs. UNIX Sockets

NGINX and uWSGI can communicate via normal TCP (using an IP address) or UNIX sockets (using a socket file). TCP sockets are easier to set up and generally work for simple deployments. If you want to have finer control over which processes, users, or groups may access the uWSGI application, or you are looking for a bit of a speed boost, consider using UNIX sockets. uWSGI can automatically drop privileges with `chmod-socket` and switch users with `chown-socket`.

The `uid` and `gid` settings, as shown above, are critical to securing your deployment. These values control the OS-level user and group the server will use to execute the application. The specified OS user and group should not have write permissions to the source directory. In this case, we use the *myproject-runner* user that was created earlier for this purpose.

You can now start uWSGI like this:

```
$ /home/myproject/venv/bin/uwsgi -c uwsgi.ini
```

If everything goes well, you should see something like this:

```
*** Operational MODE: preforking+threaded ***
...
*** uWSGI is running in multiple interpreter mode ***
...
spawned uWSGI master process (pid: 91828)
spawned uWSGI worker 1 (pid: 91866, cores: 2)
spawned uWSGI worker 2 (pid: 91867, cores: 2)
```

Note

It is always a good idea to keep an eye on the uWSGI logs, as they will contain exceptions and other information from your application that can help shed some light on unexpected behaviors.

Connecting NGINX and uWSGI

Although uWSGI may serve HTTP requests directly, it can be helpful to use a reverse proxy, such as NGINX, to offload TLS negotiation, static file serving, etc.

NGINX natively supports the uwsgi protocol, for efficiently proxying requests to uWSGI. In NGINX parlance, we will create an "upstream" and direct that upstream (via a TCP socket) to our now-running uWSGI application.

Before proceeding, install NGINX according to the instructions for your platform.

Then, create an NGINX conf file that looks something like this:

Listing 7: /etc/nginx/sites-available/myproject.conf

```
server {
  listen 80;
  server_name myproject.com;

  access_log /var/log/nginx/myproject-access.log;
  error_log  /var/log/nginx/myproject-error.log  warn;
```

(continues on next page)

(continued from previous page)

```
  location / {
    uwsgi_pass 127.0.0.1:8080
    include uwsgi_params;
  }
}
```

Finally, start (or restart) NGINX:

```
$ sudo service start nginx
```

You should now have a working application. Check your uWSGI and NGINX logs for errors if the application does not start.

Further Considerations

We did not explain how to configure TLS (HTTPS) for NGINX, leaving that as an exercise for the reader. However, we do recommend using Let's Encrypt, which offers free, short-term certificates with auto-renewal. Visit the Let's Encrypt site to learn how to integrate their service directly with NGINX.

In addition to setting up NGINX and uWSGI to run your application, you will of course need to deploy a database server or any other services required by your application. Due to the wide variety of options and considerations in this space, we have chosen not to include ancillary services in this guide. However, the Falcon community is always happy to help with deployment questions, so please don't hesitate to ask.

PYTHON MODULE INDEX

f