Falcon Documentation

Release 2.0.0dev1

Kurt Griffiths et al.

Mar 23, 2019

Contents

Release v2.0dev1 (*Installation*)

Falcon is a minimalist WSGI library for building speedy web APIs and app backends. We like to think of Falcon as the *Dieter Rams* of web frameworks.

When it comes to building HTTP APIs, other frameworks weigh you down with tons of dependencies and unnecessary abstractions. Falcon cuts to the chase with a clean design that embraces HTTP and the REST architectural style.

```
class QuoteResource:

    def on_get(self, req, resp):
        """Handles GET requests"""
        quote = {
            'quote': (
                "I've always been more interested in "
                "the future than in the past."
            ),
            'author': 'Grace Hopper'
        }

        resp.media = quote


api = falcon.API()
api.add_route('/quote', QuoteResource())
```

CHAPTER 1

What People are Saying

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

"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." "Falcon is rock solid and it's fast."

"I'm loving #falconframework! Super clean and simple, I finally have the speed and flexibility I need!"

"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 2

Quick Links

- Read the docs
- Falcon add-ons and complementary packages
- Falcon talks, podcasts, and blog posts
- falconry/user for Falcon users @ Gitter
- falconry/dev for Falcon contributors @ Gitter

CHAPTER 3

Features

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

- Routes based on URI templates RFC
- REST-inspired mapping of URIs to resources
- Global, resource, and method hooks
- Idiomatic HTTP error responses
- Full Unicode support
- Intuitive request and response objects
- Works great with async libraries like gevent
- Minimal attack surface for writing secure APIs
- 100% code coverage with a comprehensive test suite
- No dependencies on other Python packages
- Supports Python 2.7, 3.5+
- Compatible with PyPy

CHAPTER 4

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 5

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 2.7/3.5+ and PyPy.

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.

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.

Debuggable. Falcon eschews magic. It's easy to tell which inputs lead to which outputs. Unhandled exceptions are never encapsulated or masked. Potentially surprising behaviors, such as automatic request body parsing, are well-documented and disabled by default. Finally, when it comes to the framework itself, we take care to keep logic paths simple and understandable. All this makes it easier to reason about the code and to debug edge cases in large-scale deployments.

Features

- Highly-optimized, extensible code base
- Intuitive routing via URI templates and REST-inspired resource classes
- Easy access to headers and bodies through request and response classes
- DRY request processing via middleware components and hooks
- Idiomatic HTTP error responses
- Straightforward exception handling
- Snappy unit testing through WSGI helpers and mocks
- Supports Python 2.7, 3.5+
- Compatible with PyPy

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. Both PyPy2.7 and PyPy3.5 are supported as of PyPy v5.10.

```
$ pip install falcon
```

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

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

CPython

Falcon also fully supports CPython 2.7, and 3.5+.

A universal wheel is available on PyPI for the the Falcon framework. Installing it is as simple as:

```
$ pip install falcon
```

Installing the Falcon wheel is a great way to get up and running quickly in a development environment, but for an extra speed boost when deploying your application in production, Falcon can compile itself with Cython.

The following commands tell pip to install Cython, and then to invoke Falcon's `setup.py`, which will in turn detect the presence of Cython and then compile (AKA cythonize) the Falcon framework with the system's default C compiler.

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

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

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

Installing on OS X

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"
```

Dependencies

Falcon does not require the installation of any other packages, although if Cython has been installed into the environment, it will be used to optimize the framework as explained above.

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.

All Windows developers can use Waitress production-quality pure-Python WSGI server with very acceptable performance. Unfortunately Gunicorn is still not working on Windows and uWSGI need to have Cygwin on Windows installed. Waitress can be good alternative for Windows users if they want quick start using Falcon on it.

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

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:

```
$ cd falcon
$ 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 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
$ 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
$ 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 API:

```
# things.py

# Let's get this party started!
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(object):
    def on_get(self, req, resp):
        """Handles GET requests"""
        resp.status = falcon.HTTP_200  # This is the default status
        resp.body = ('\nTwo things awe me most, the starry sky '
                     'above me and the moral law within me.\n'
                     '\n'
                     '    ~ Immanuel Kant\n\n')

# falcon.API instances are callable WSGI apps
app = falcon.API()

# 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 above example using any WSGI server, such as uWSGI or Gunicorn. For example:

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

On Windows where Gunicorn and uWSGI don't work yet you can use Waitress server

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

Then, in another terminal:

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

Curl is a bit of a pain to use, so let's install HTTPie and use it from now on.

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

More Features

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

```
import json
import logging
import uuid
from wsgiref import simple_server

import falcon
import requests


class StorageEngine(object):

    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):

    @staticmethod
    def handle(ex, req, resp, params):
        description = ('Sorry, couldn\'t write your thing to the '
                       'database. It worked on my box.')

        raise falcon.HTTPError(falcon.HTTP_725,
                               'Database Error',
                               description)


class SinkAdapter(object):

    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 = str(result.status_code) + ' ' + result.reason
        resp.content_type = result.headers['content-type']
        resp.body = result.text


class AuthMiddleware(object):

    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:
```

(continues on next page)

(continued from previous page)

```
            description = ('Please provide an auth token '
                           'as part of the request.')

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

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


class RequireJSON(object):

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


class JSONTranslator(object):
    # NOTE: Starting with Falcon 1.3, you can simply
    # use req.media and resp.media for this instead.

    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('Empty request body',
                                        'A valid JSON document is required.')

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

(continues on next page)

(continued from previous page)

```
        except (ValueError, UnicodeDecodeError):
            raise falcon.HTTPError(falcon.HTTP_753,
                                   'Malformed JSON',
                                   'Could not decode the request body. The '
                                   'JSON was incorrect or not encoded as '
                                   'UTF-8.')

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

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


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.HTTPPayloadTooLarge(
                'Request body is too large', msg)

    return hook


class ThingsResource(object):

    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(
                'Service Outage',
                description,
                30)

        # An alternative way of doing DRY serialization would be to
        # create a custom class that inherits from falcon.Request. This
        # class could, for example, have an additional 'doc' property
        # that would serialize to JSON under the covers.
        #
```

(continues on next page)

(continued from previous page)

```
        # NOTE: Starting with Falcon 1.3, you can simply
        # use resp.media for this instead.
        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(
                'Missing thing',
                'A thing must be submitted in the request body.')

        proper_thing = self.db.add_thing(doc)

        resp.status = falcon.HTTP_201
        resp.location = '/%s/things/%s' % (user_id, proper_thing['id'])


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

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

# If a responder ever raised 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()
```

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

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

5.1.4 Tutorial

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.

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

api = application = falcon.API()
```

This code creates your WSGI application and aliases it as `api`. 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.API* class. Install IPython and fire it up:

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

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

```
In [1]: import falcon

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

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

```
In [3]: help(falcon.API.__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:api
```

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:

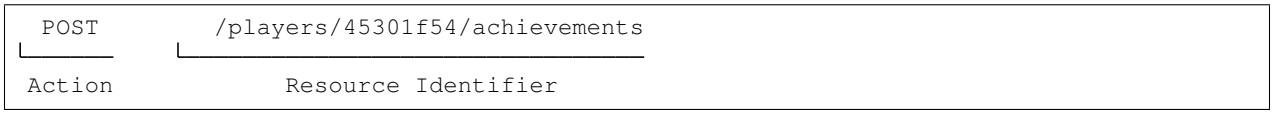
```
$ source .venv/bin/activate
$ pip install httpie
$ http localhost:8000
```

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".

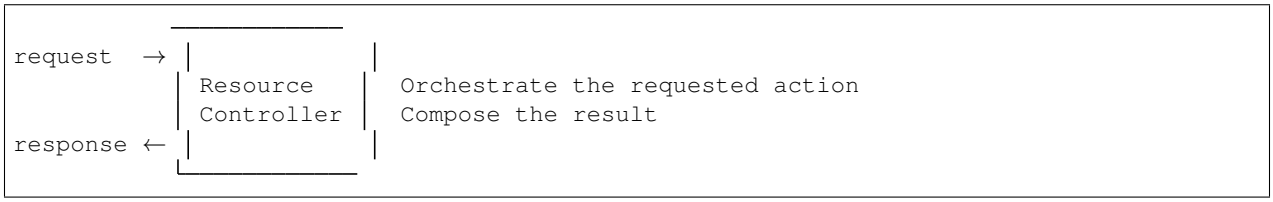


In the REST architectural style, the URL only identifies the resource; it does not specify what action to take on that resource. Instead, users choose from a set of standard methods. For HTTP, these are the familiar GET, POST, HEAD, etc. Clients can query a resource to discover which methods it supports.

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.



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(object):

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

        # Create a JSON representation of the resource
        resp.body = 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
```

(continues on next page)

(continued from previous page)

```
        # 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


api = application = falcon.API()

images = Resource()
api.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.

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 falcon

import msgpack


class Resource(object):

    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.body`. 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
│   ├── app.py
│   └── images.py
└── tests
    ├── __init__.py
    └── test_app.py
```

Falcon supports *testing* its *API* 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 falcon
from falcon import testing
import msgpack
import pytest

from look.app import api


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


# 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` if you are using Python 3, or from `mock` if you are using Python 2.

```
# Python 3
from unittest.mock import mock_open, call

# Python 2
from mock import mock_open, call
```

For Python 2, you will also need to install the `mock` package:

```
$ pip install mock
```

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)
```

(continues on next page)

(continued from previous page)

```
    # 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 falcon
import msgpack


class Resource(object):

    _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):
```

(continues on next page)

(continued from previous page)

```
        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 falcon
import msgpack


class Resource(object):

    def __init__(self, image_store):
        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(object):

    _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)
```

(continues on next page)

(continued from previous page)

```
        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, Resource


api = application = falcon.API()

image_store = ImageStore('.')
images = Resource(image_store)
api.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, Resource


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


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 API 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

# Python 3
from unittest.mock import call, MagicMock, mock_open

# Python 2
# from mock import call, MagicMock, mock_open

import falcon
from falcon import testing
import msgpack
import pytest

import look.app
import look.images


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


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


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'
```

(continues on next page)

(continued from previous page)

```
    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], falcon.request_helpers.BoundedStream)
    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

```

(continues on next page)

(continued from previous page)

```
import falcon

from .images import ImageStore, Resource


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


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 falcon
import msgpack


class Collection(object):

    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': [
                {
                    '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(object):

    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(object):

    _CHUNK_SIZE_BYTES = 4096
    _IMAGE_NAME_PATTERN = re.compile(
        '[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
```

(continues on next page)

(continued from previous page)

```
    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 IOError('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.body` 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

import images


def create_app(image_store):
```

(continues on next page)

(continued from previous page)

```
    api = falcon.API()
    api.add_route('/images', images.Collection(image_store))
    api.add_route('/images/{name}', images.Item(image_store))
    return api


def get_app():
    storage_path = os.environ.get('LOOK_STORAGE_PATH', '.')
    image_store = images.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.

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('Bad request', msg)
```

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

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

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(object):

    # ...
```

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: Falcon will re-raise errors that do not inherit from *falcon.HTTPError* unless you have registered a custom error handler for that type.

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(object):

    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 IOError:
            # 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 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.

Why does raising an error inside a resource crash my app?

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.

This approach reduces the number of 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. 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.

Tip: Falcon will re-raise errors that do not inherit from *`HTTPError`* unless you have registered a custom error handler for that type (see also: *falcon.API*).

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.

That being said, IO-bound Falcon APIs are usually scaled via multiple processes and green threads (courtesy of the gevent library or similar) which aren't truly running concurrently, so there may be some edge cases where Falcon is not thread-safe that we aren't aware of. If you run into any issues, please let us know.

Does Falcon support asyncio?

Due to the limitations of WSGI, Falcon is unable to support `asyncio` at this time. However, we are exploring alternatives to WSGI (such as ASGI) that will allow us to support asyncio natively in the future.

In the meantime, we recommend using the battle-tested gevent library via Gunicorn or uWSGI to scale IO-bound services. meinheld has also been used successfully by the community to power high-throughput, low-latency services. 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?

Due to the limitations of WSGI, Falcon is unable to support the WebSocket protocol as stated above.

In the meantime, you might try leveraging uWSGI's native WebSocket support, or implementing a standalone service 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 implementing a middleware component similar to the following:

```
class CORSComponent(object):
    def process_response(self, req, resp, resource, req_succeeded):
        resp.set_header('Access-Control-Allow-Origin', '*')

        if (req_succeeded
            and req.method == 'OPTIONS'
            and req.get_header('Access-Control-Request-Method')
        ):
            # NOTE(kgriffs): This is a CORS preflight request. Patch the
            #   response accordingly.

            allow = resp.get_header('Allow')
            resp.delete_header('Allow')

            allow_headers = req.get_header(
                'Access-Control-Request-Headers',
                default='*'
            )

            resp.set_headers((
                ('Access-Control-Allow-Methods', allow),
                ('Access-Control-Allow-Headers', allow_headers),
                ('Access-Control-Max-Age', '86400'),  # 24 hours
            ))
```

When using the above approach, OPTIONS requests must also be special-cased in any other middleware or hooks you use for auth, content-negotiation, etc. For example, you will typically skip auth for preflight requests because it is simply unnecessary; note that such request do not include the Authorization header in any case.

For 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.

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(object):
    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


# ...


resource = MyResource()
api.add_route('/resources/{id}', resource)
api.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

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(object):

    def on_get(self, req, resp):
        resp.body = '{"message": "pong"}'


class Game(object):

    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(object):

    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


api = falcon.API()

# 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
# layer.
game_dao = myapp.DAL.Game(myconfig)

api.add_route('/game/ping', Ping())
api.add_route('/game/{game_id}', Game(game_dao))
api.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(object):

    def __init__(self, dao):
```

(continues on next page)

(continued from previous page)

```
        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"}'


# ...


api = falcon.API()

game = Game(myapp.DAL.Game(myconfig))

api.add_route('/game/{game_id}', game)
api.add_route('/game/{game_id}/state', game, suffix='state')
api.add_route('/game/ping', game, suffix='ping')
```

Extensibility

How do I use WSGI middleware with Falcon?

Instances of *falcon.API* 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.api)
```

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):
    # Check authentication/authorization
    # ...

    req.context.role = 'root'
```

(continues on next page)

(continued from previous page)

```
    req.context.scopes = ('storage', 'things')
    req.context.uid = 0

# ...

@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 *falcon.HTTPNotFound*. You can use *falcon.API.add_error_handler()* to register a custom error handler for this exception type. Alternatively, you may be able to configure your web server to transform the response for you (e.g., using Nginx's `error_page` directive).

500 errors are typically the result of an unhandled exception making its way up to the web server. To handle these errors more gracefully, you can add a custom error handler for Python's base `Exception` type.

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*.

Why are trailing slashes trimmed from req.path?

By default, Falcon normalizes incoming URI paths to simplify later processing and improve the predictability of application logic. This behavior can be disabled via the *strip_url_path_trailing_slash* request option.

Note also that routing is also normalized, so adding a route for "/foo/bar" also implicitly adds a route for "/foo/bar/". Requests coming in for either path will be sent to the same resource.

Why is my query parameter missing from the req object?

If a query param does not have a value, Falcon will by default ignore that parameter. For example, passing `'foo'` or `'foo='` will result in the parameter being ignored.

If you would like to recognize such parameters, you must set the *keep_blank_qs_values* request option to `True`. Request options are set globally for each instance of *falcon.API* via the *req_options* property. For example:

```
api.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, setting the `auto_parse_form_urlencoded` to `True` on an instance of `falcon.API` will cause the framework to consume the request body when the content type is `application/x-www-form-urlencoded`, making the form parameters accessible via *params*, *get_param()*, etc.

```
api.req_options.auto_parse_form_urlencoded = True
```

Alternatively, POSTed form parameters may be read directly from *stream* and parsed via *falcon.uri.parse_query_string()* or urllib.parse.parse_qs().

How can I access POSTed files?

Falcon does not currently support parsing files submitted by an HTTP form (`multipart/form-data`), although we do plan to add this feature in a future version. In the meantime, you can use the standard `cgi.FieldStorage` class to parse the request:

```
# TODO: Either validate that content type is multipart/form-data
# here, or in another hook before allowing execution to proceed.

# This must be done to avoid a bug in cgi.FieldStorage
env = req.env
env.setdefault('QUERY_STRING', '')

# TODO: Add error handling, when the request is not formatted
# correctly or does not contain the desired field...

# TODO: Consider overriding make_file, so that you can
# stream directly to the destination rather than
# buffering using TemporaryFile (see http://goo.gl/Yo8h3P)
form = cgi.FieldStorage(fp=req.stream, environ=env)

file_item = form[name]
if file_item.file:
    # TODO: It's an uploaded file... read it in
else:
    # TODO: Raise an error
```

You might also try this streaming_form_data package by Siddhant Goel, or searching PyPI for additional options from the community.

How do I consume a query string that has a JSON value?

Falcon defaults to treating commas in a query string as literal characters delimiting a comma separated list. For example, given the query string `?c=1,2,3`, Falcon defaults to adding 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 get added to your `request.params` in a way that you probably don't expect: `{'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 switch the way Falcon handles commas in a query string by setting the *auto_parse_qs_csv* to `False` on an instance of *falcon.API*:

```
api.req_options.auto_parse_qs_csv = False
```

When *auto_parse_qs_csv* is set to `False`, 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 escape commas in the request.

How can I handle forward slashes within a route template field?

In Falcon 1.3 we shipped initial support for field converters. We've discussed building on this feature to support consuming multiple path segments ala Flask. This work is currently planned for 2.0.

In the meantime, the workaround is to percent-encode the forward slash. If you don't control the clients and can't enforce this, you can implement a Falcon middleware component to rewrite the path before it is routed.

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 class subclassing dict 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()
```

Since the default context type derives from dict, the existing code will work unmodified with Falcon 2.0. Nevertheless, it is recommended to change usage as outlined above since the ability to use context as dictionary may be removed in a future release.

> **Warning:** Context attributes are not linked to dict items in any special way, i.e. setting an object attribute does not set the corresponding dict item, nor vice versa.
>
> Furthermore, if you need to mix-and-match both approaches under migration, beware that setting attributes such as *items* or *values* would obviously shadow the corresponding dict functions.

If an existing project is making 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

# ...

api = falcon.API(request_type=RequestWithDictContext,
                 response_type=ResponseWithDictContext)
```

Response Handling

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, 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.

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.body`, `resp.data`, or `resp.stream_len` (whichever is set in the response, checked in that order.)

For dynamically-generated content, you can choose to not set `stream_len`, 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 it to enable keep-alive).

Note: PEP-3333 prohibits apps from setting hop-by-hop headers itself, such as Transfer-Encoding.

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'
```

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.API.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
result = self._engine.execute(some_table.select())
for row in result:
    # ....
result.close()

# ...

# Write to the DB within a transaction
with self._engine.begin() as connection:
    r1 = connection.execute(some_table.select())
    # ...
    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.

What is the recommended approach for making configuration variables available to multiple resource classes?

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.

Other than that, it's pretty much up to you if you want to use the standard library config 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).

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)

5.2 Classes and Functions

5.2.1 The API Class

Falcon's API class is a WSGI "application" that you can host with any standard-compliant WSGI server.

```
import falcon

app = falcon.API()
```

class falcon.**API**(*media_type='application/json'*, *request_type=<class 'falcon.request.Request'>*, *response_type=<class 'falcon.response.Response'>*, *middleware=None*, *router=None*, *independent_middleware=True*)

This class is the main entry point into a Falcon-based app.

Each API instance provides a callable WSGI interface and a routing engine.

Keyword Arguments

- **media_type** (*str*) – Default media type to use as the value for the Content-Type header on responses (default 'application/json'). 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** (*object or list*) – Either a single object or a list of objects (instantiated classes) that implement the following middleware component interface:

```
class ExampleComponent(object):
    def process_request(self, req, resp):
        """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, resp, resource, params):
        """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
                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, resp, resource, req_succeeded)
        """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
```

(continues on next page)

(continued from previous page)

```
                request; otherwise False.
        """
```

(See also: *Middleware*)

- **request_type** (Request) – Request-like class to use instead of Falcon's default class. Among other things, this feature affords inheriting from falcon.request.Request in order to override the context_type class variable. (default falcon.request.Request)
- **response_type** (Response) – Response-like class to use instead of Falcon's default class. (default falcon.response.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.

req_options
A set of behavioral options related to incoming requests. (See also: *RequestOptions*)

resp_options
A set of behavioral options related to outgoing responses. (See also: *ResponseOptions*)

router_options
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*)

add_error_handler(*exception*, *handler=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.

Error handlers are matched in LIFO order. In other words, when searching for an error handler to match a raised exception, and more than one handler matches the exception type, the framework will choose the one that was most recently registered. Therefore, more general error handlers (e.g., for the standard Exception type) should be added first, to avoid masking more specific handlers for subclassed types.

Note: By default, the framework installs two handlers, one for *HTTPError* and one for *HTTPStatus*. 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
        # 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.

add_route(*uri_template*, *resource*, ***kwargs*)

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.

If no route matches the request, control then passes to a default responder that simply raises an instance of *HTTPNotFound*.

This method delegates to the configured router's `add_route()` method. To override the default behavior, pass a custom router object to the *API* 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()*)

- **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.

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.

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*, *prefix='/'*)

Register a sink method for the API.

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.

Parameters

- **sink** (*callable*) – A callable taking the form `func(req, resp)`.
- **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*, *directory*, *downloadable=False*, *fallback_filename=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.

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:

```
api.add_static_route('/foo', foo_path)
api.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** (*str*) – 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.

set_error_serializer(*serializer*)

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: The default serializer will not render any response body for *HTTPError* instances where the *has_representation* property evaluates to `False` (such as in the case of types that subclass *falcon.http_error.NoRepresentation*). However a custom serializer will be called regardless of the property value, and it may choose to override the representation logic.

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, resp, exception):
    representation = None

    preferred = req.client_prefers(('application/x-yaml',
                                    'application/json'))

    if exception.has_representation and preferred is not None:
        if preferred == 'application/json':
            representation = exception.to_json()
        else:
            representation = yaml.dump(exception.to_dict(),
                                       encoding=None)
        resp.body = representation
        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`.

class falcon.**RequestOptions**

Defines a set of configurable request options.

An instance of this class is exposed via *API.req_options* for configuring certain *Request* behaviors.

keep_blank_qs_values

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.

Type bool

auto_parse_form_urlencoded
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 makes the form parameters accessible via *params*, *get_param()*, etc.

> **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 http://goo.gl/6rlcux).

---

auto_parse_qs_csv
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` becomes `['1', '2', '3', '4']`).

strip_url_path_trailing_slash
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.

default_media_type
The default media-type to use when deserializing a response. This value is normally set to the media type provided when a *falcon.API* is initialized; however, if created independently, this will default to the `DEFAULT_MEDIA_TYPE` specified by Falcon.

Type str

media_handlers
A dict-like object that allows you to configure the media-types that you would like to handle. By default, a handler is provided for the `application/json` media type.

Type *Handlers*

class falcon.**ResponseOptions**
Defines a set of configurable response options.

An instance of this class is exposed via *API.resp_options* for configuring certain *Response* behaviors.

secure_cookies_by_default
Set to `False` in development environments to make the *secure* attribute for all cookies default to `False`. This can make testing easier by not requiring HTTPS. Note, however, that this setting can be overridden via *set_cookie()*'s *secure* kwarg.

Type bool

default_media_type
The default Internet media type (RFC 2046) to use when deserializing a response. This value is normally set to the media type provided when a *falcon.API* is initialized; however, if created independently, this will default to the DEFAULT_MEDIA_TYPE specified by Falcon.

Type str

media_handlers
A dict-like object that allows you to configure the media-types that you would like to handle. By default, a handler is provided for the application/json media type.

Type *Handlers*

static_media_types
A mapping of dot-prefixed file extensions to Internet media types (RFC 2046). Defaults to mimetypes.types_map after calling mimetypes.init().

Type dict

class falcon.routing.**CompiledRouterOptions**
Defines a set of configurable router options.

An instance of this class is exposed via *API.router_options* for configuring certain *CompiledRouter* behaviors.

converters
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:

```
api.router_options.converters['mc'] = MyConverter
```

The identifier can then be used to employ the converter within a URI template:

```
api.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*)

5.2.2 Request & Response

Instances of the Request and Response classes are passed into responders as the second and third arguments, respectively.

```
import falcon


class Resource(object):

    def on_get(self, req, resp):
        resp.body = '{"message": "Hello world!"}'
        resp.status = falcon.HTTP_200
```

Request

class falcon.**Request**(*env*, *options=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** (*dict*) – Set of global options passed from the API handler.

env
Reference to the WSGI environ `dict` passed in from the server. (See also PEP-3333.)

Type dict

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: **New in 2.0:** the default *context_type* (see below) was changed from dict to a bare class, and the preferred way to pass request-specific data is now to set attributes directly on the *context* object, for example:

```
req.context.role = 'trial'
req.context.user = 'guest'
```

Type object

context_type
Class variable that determines the factory or type to use for initializing the *context* attribute. By default, the framework will instantiate bare objects (instances of the bare RequestContext class). However, you may override this behavior by creating a custom child class of `falcon.Request`, and then passing that new class to *falcon.API()* by way of the latter's *request_type* parameter.

Note: When overriding *context_type* with a factory function (as opposed to a class), the function is called like a method of the current Request instance. Therefore the first argument is the Request instance itself (self).

Type class

scheme
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.

Type str

forwarded_scheme
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)

Type str

method
HTTP method requested (e.g., 'GET', 'POST', etc.)

Type str

host
Host request header field

Type str

forwarded_host
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)

Type str

port
Port used for the request. If the request URI does not specify a port, the default one for the given schema is returned (80 for HTTP and 443 for HTTPS).

Type int

netloc
Returns the 'host:port' portion of the request URL. The port may be ommitted if it is the default one for the URL's schema (80 for HTTP and 443 for HTTPS).

Type str

subdomain
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.

Type str

app
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.

(Corresponds to the "SCRIPT_NAME" environ variable defined by PEP-3333.)

Type str

uri
The fully-qualified URI for the request.

Type str

url
Alias for *uri*.

Type str

forwarded_uri
Original URI for proxied requests. Uses *forwarded_scheme* and *forwarded_host* in order to reconstruct the original URI requested by the user agent.

Type str

relative_uri
The path and query string portion of the request URI, omitting the scheme and host.

Type str

prefix
The prefix of the request URI, including scheme, host, and WSGI app (if any).

Type str

forwarded_prefix
The prefix of the original URI for proxied requests. Uses *forwarded_scheme* and *forwarded_host* in order to reconstruct the original URI.

Type str

path
Path portion of the request URI (not including query string).

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. 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.

Type str

query_string
Query string portion of the request URI, without the preceding '?' character.

Type str

uri_template
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.

Type str

remote_addr
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.

Type str

access_route
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.

Type list

forwarded
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)

Type list

date
Value of the Date header, converted to a `datetime` instance. The header value is assumed to conform to RFC 1123.

Type datetime

auth
Value of the Authorization header, or `None` if the header is missing.

Type str

user_agent
Value of the User-Agent header, or `None` if the header is missing.

Type str

referer
Value of the Referer header, or `None` if the header is missing.

Type str

accept
Value of the Accept header, or '/' if the header is missing.

Type str

client_accepts_json
`True` if the Accept header indicates that the client is willing to receive JSON, otherwise `False`.

Type bool

client_accepts_msgpack
`True` if the Accept header indicates that the client is willing to receive MessagePack, otherwise `False`.

Type bool

client_accepts_xml
`True` if the Accept header indicates that the client is willing to receive XML, otherwise `False`.

Type bool

cookies
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()*

Type dict

content_type
Value of the Content-Type header, or `None` if the header is missing.

Type str

content_length
Value of the Content-Length header converted to an `int`, or `None` if the header is missing.

Type int

stream
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 particulary 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.

---

bounded_stream

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)
```

expect

Value of the Expect header, or `None` if the header is missing.

Type str

media

Returns a deserialized form of the request stream. When called, it will attempt to deserialize the request stream using the Content-Type header as well as the media-type handlers configured via *falcon.RequestOptions*.

See *Media* for more information regarding media handling.

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.

Type object

range
A 2-member `tuple` parsed from the value of the Range header.

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 continous ranges are supported (e.g., "bytes=0-0,-1" would result in an HTTPBadRequest exception when the attribute is accessed.)

Type tuple of int

range_unit
Unit of the range parsed from the value of the Range header, or `None` if the header is missing

Type str

if_match
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)

Type list

if_none_match
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)

Type list

if_modified_since
Value of the If-Modified-Since header, or `None` if the header is missing.

Type datetime

if_unmodified_since
Value of the If-Unmodified-Since header, or `None` if the header is missing.

Type datetime

if_range
Value of the If-Range header, or `None` if the header is missing.

Type str

headers
Raw HTTP headers from the request with canonical dash-separated names. 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.

Type dict

params
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.

Type dict

options
Set of global options passed from the API handler.

Type dict

client_accepts(*media_type*)
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

client_prefers(*media_types*)
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

context_type
alias of `RequestContext`

get_cookie_values(*name*)
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*, *required=False*, *default=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*, *required=False*, *obs_date=False*)
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.

get_param(*name*, *required=False*, *store=None*, *default=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 *API.req_options*.

Note: Similar to the way multiple keys in form data is handled, if a query parameter is assigned a comma-separated list of values (e.g., `foo=a,b,c`), only one of those values will be returned, and it is undefined which one. Use *get_param_as_list()* to retrieve all the values.

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*, *required=False*, *store=None*, *blank_as_true=True*, *default=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', 'yes', '1', 'on')
FALSE_STRINGS = ('false', 'False', 'no', '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*, *format_string='%Y-%m-%d'*, *required=False*, *store=None*, *default=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*, *format_string='%Y-%m-%dT%H:%M:%SZ'*, *required=False*, *store=None*, *default=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:%SZ'`).
- **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`.

get_param_as_float(*name*, *required=False*, *min_value=None*, *max_value=None*, *store=None*, *default=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*, *required=False*, *min_value=None*, *max_value=None*, *store=None*, *default=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*, *required=False*, *store=None*, *default=None*)

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.)

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*, *transform=None*, *required=False*, *store=None*, *default=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*.

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 discarded. For example, the following query strings would both result in *['1', '3']*:

```
things=1,,3
things=1&things=&things=3
```

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*, *required=False*, *store=None*, *default=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*)
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

log_error(*message*)
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 or unicode*) – Description of the problem. On Python 2, instances of `unicode` will be converted to UTF-8.

class falcon.**Forwarded**
Represents a parsed Forwarded header.

(See also: RFC 7239, Section 4)

src
The value of the "for" parameter, or `None` if the parameter is absent. Identifies the node making the request to the proxy.

Type str

dest
The value of the "by" parameter, or `None` if the parameter is absent. Identifies the client-facing interface of the proxy.

Type str

host
The value of the "host" parameter, or `None` if the parameter is absent. Provides the host request header field as received by the proxy.

Type str

scheme
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.

Type str

Response

class falcon.**Response**(*options=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 API handler.

status
HTTP status line (e.g., '200 OK'). Falcon requires the full status line, not just the code (e.g., 200). This design makes the framework more efficient because it does not have to do any kind of conversion or lookup when composing the WSGI response.

If not set explicitly, the status defaults to '200 OK'.

Note: Falcon provides a number of constants for common status codes. They all start with the `HTTP_` prefix, as in: `falcon.HTTP_204`.

Type str

media
A serializable object supported by the media handlers configured via *falcon.RequestOptions*.

See *Media* for more information regarding media handling.

Type object

body
String representing response content.

If set to a Unicode type (`unicode` in Python 2, or `str` in Python 3), Falcon will encode the text as UTF-8 in the response. If the content is already a byte string, use the *data* attribute instead (it's faster).

Type str or unicode

data
Byte string representing response content.

Use this attribute in lieu of *body* when your content is already a byte string (`str` or `bytes` in Python 2, or simply `bytes` in Python 3). See also the note below.

Note: Under Python 2.x, if your content is of type `str`, using the *data* attribute instead of *body* is the most efficient approach. However, if your text is of type `unicode`, you will need to use the *body* attribute instead.

Under Python 3.x, on the other hand, the 2.x `str` type can be thought of as having been replaced by what was once the `unicode` type, and so you will need to always use the *body* attribute for strings to ensure Unicode characters are properly encoded in the HTTP response.

Type bytes

stream
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.

---

stream_len
Deprecated alias for *content_length*.

Type int

context
Dictionary to hold any data about the response which is specific to your app. Falcon itself will not interact with this attribute after it has been initialized.

Type dict

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: **New in 2.0:** the default *context_type* (see below) was changed from dict to a bare class, and the preferred way to pass response-specific data is now to set attributes directly on the *context* object, for example:

```
resp.context.cache_strategy = 'lru'
```

Type object

context_type
Class variable that determines the factory or type to use for initializing the *context* attribute. By default, the framework will instantiate bare objects (instances of the bare ResponseContext class). However, you may override this behavior by creating a custom child class of `falcon.Response`, and then passing that new class to *falcon.API()* by way of the latter's *response_type* parameter.

Note: When overriding *context_type* with a factory function (as opposed to a class), the function is called like a method of the current Response instance. Therefore the first argument is the Response instance itself (self).

Type class

options
Set of global options passed from the API handler.

Type dict

headers
Copy of all headers set for the response, sans cookies. Note that a new copy is created and returned each time this property is referenced.

Type dict

complete
Set to `True` from within a middleware method to signal to the framework that request processing should be short-circuited (see also *Middleware*).

Type bool

accept_ranges
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.

add_link (*target*, *rel*, *title=None*, *title_star=None*, *anchor=None*, *hreflang=None*, *type_hint=None*)
Add 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.

Note: So-called "link-extension" elements, as defined by RFC 5988, are not yet supported. See also Issue #288.

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 of 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. If the string contains non-ASCII characters, it should be passed as a `unicode` type string (requires the 'u' prefix in Python 2).

- **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.

append_header (*name*, *value*)
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 restrictions noted below for the header's value also apply here.
- **value** (*str*) – Value for the header. Must be convertable to `str` or be of type `str` or `StringType`. Strings must contain only US-ASCII characters. Under Python 2.x, the `unicode` type is also accepted, although such strings are also limited to US-ASCII.

cache_control
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.

content_length
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 *body* property or the *data* property is set on the response, the framework will force Content-Length to be the length of the given body 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 WSGI server unless *content_length* is explicitly set. Consequently, the server may choose to use chunked encoding or one of the other strategies suggested by PEP-3333.

---

content_location
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.

content_range
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)

content_type
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_type
alias of `ResponseContext`

delete_header(*name*)
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). Must be of type `str` or `StringType` and contain only US-ASCII characters. Under Python 2.x, the `unicode` type is also accepted, although such strings are also limited to US-ASCII.

Raises `ValueError` – *name* cannot be `'Set-Cookie'`.

downloadable_as
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"'`.

etag
Set the ETag header.

The ETag header will be wrapped with double quotes `"value"` in case the user didn't pass it.

expires
Set the Expires header. Set to a `datetime` (UTC) instance.

Note: Falcon will format the `datetime` as an HTTP date string.

get_header(*name*, *default=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

last_modified
Set the Last-Modified header. Set to a datetime (UTC) instance.

Note: Falcon will format the datetime as an HTTP date string.

---

location
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.

retry_after
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*, *value*, *expires=None*, *max_age=None*, *domain=None*, *path=None*, *secure=None*, *http_only=True*)
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.

  (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 *API.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*) – Direct the client to only transfer the cookie with unscripted HTTP requests (default: `True`). This is intended to mitigate some forms of cross-site scripting.

  (See also: RFC 6265, Section 4.1.2.6)

Raises

- `KeyError` – *name* is not a valid cookie name.
- `ValueError` – *value* is not a valid cookie value.

set_header(*name*, *value*)

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 restrictions noted below for the header's value also apply here.
- **value** (*str*) – Value for the header. Must be convertable to `str` or be of type `str` or `StringType`. Strings must contain only US-ASCII characters. Under Python 2.x, the `unicode` type is also accepted, although such strings are also limited to US-ASCII.

Raises `ValueError` – *name* cannot be `'Set-Cookie'`.

set_headers(*headers*)
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** (*dict or list*) – A dictionary 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 contain only US-ASCII characters. Under Python 2.x, the `unicode` type is also accepted, although such strings are also limited to US-ASCII.

Note: Falcon can process a list of tuples slightly faster than a dict.

Raises `ValueError` – *headers* was not a `dict` or `list` of `tuple`.

set_stream(*stream*, *content_length*)
Convenience method for setting 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 WSGI 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.

stream_len
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 *body* property or the *data* property is set on the response, the framework will force Content-Length to be the length of the given body 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 WSGI server unless *content_length* is explicitly set. Consequently, the server may choose to use chunked encoding or one of the other strategies suggested by PEP-3333.

---

unset_cookie(*name*)
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.

> **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.

vary
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)

5.2.3 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.

```
class Resource(object):
    def on_get(self, req, resp):

        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]

            # ...
```

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).

Simple example:

```
class Resource(object):
    def on_get(self, req, resp):

        # 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.

```
class Resource(object):
    def on_get(self, req, resp):
        # 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:

```
class Resource(object):
    def on_get(self, req, resp):
        resp.set_cookie('bad_cookie', ':(')

        # Clear the bad cookie
        resp.unset_cookie('bad_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 *API.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

5.2.4 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.

Falcon also provides a generic *HTTPStatus* class. Raise this class from a 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*, *headers=None*, *body=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.

status

HTTP status line, e.g. '748 Confounded by Ponies'.

Type str

headers

Extra headers to add to the response.

Type dict

body

String representing response content. If Unicode, Falcon will encode as UTF-8 in the response.

Type str or unicode

Parameters

- **status** (*str*) – HTTP status code and text, such as '748 Confounded by Ponies'.
- **headers** (*dict*) – Extra headers to add to the response.
- **body** (*str or unicode*) – String representing response content. If Unicode, Falcon will encode as UTF-8 in the response.

1xx Informational

```
HTTP_CONTINUE = HTTP_100
HTTP_SWITCHING_PROTOCOLS = HTTP_101
HTTP_PROCESSING = HTTP_102

HTTP_100 = '100 Continue'
HTTP_101 = '101 Switching Protocols'
HTTP_102 = '102 Processing'
```

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'
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
```

(continues on next page)

(continued from previous page)

```
HTTP_REQUEST_TIMEOUT = HTTP_408
HTTP_CONFLICT = HTTP_409
HTTP_GONE = HTTP_410
HTTP_LENGTH_REQUIRED = HTTP_411
HTTP_PRECONDITION_FAILED = HTTP_412
HTTP_REQUEST_ENTITY_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_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
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 Time-out'
HTTP_409 = '409 Conflict'
HTTP_410 = '410 Gone'
HTTP_411 = '411 Length Required'
HTTP_412 = '412 Precondition Failed'
HTTP_413 = '413 Payload 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_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
```

(continues on next page)

(continued from previous page)

```
HTTP_HTTP_VERSION_NOT_SUPPORTED = HTTP_505
HTTP_INSUFFICIENT_STORAGE = HTTP_507
HTTP_LOOP_DETECTED = HTTP_508
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 Time-out'
HTTP_505 = '505 HTTP Version not supported'
HTTP_507 = '507 Insufficient Storage'
HTTP_508 = '508 Loop Detected'
HTTP_511 = '511 Network Authentication Required'
```

5.2.5 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:

```
import falcon

class MessageResource(object):
```

(continues on next page)

(continued from previous page)

```
    def on_get(self, req, resp):

        # ...

        raise falcon.HTTPBadRequest(
            "TTL Out of Range",
            "The message's TTL must be between 60 and 300 seconds, inclusive."
        )

        # ...
```

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.

Base Class

class falcon.**HTTPError**(*status*, *title=None*, *description=None*, *headers=None*, *href=None*, *href_text=None*, *code=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 *API* 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()`). To also support XML, override the `to_xml()` method.

status
HTTP status line, e.g. '748 Confounded by Ponies'.

Type str

has_representation
Read-only property that determines whether error details will be serialized when composing the HTTP response. In `HTTPError` this property always returns `True`, but child classes may override it in order to return `False` when an empty HTTP body is desired.

(See also: *falcon.http_error.NoRepresentation*)

Note: A custom error serializer (see *set_error_serializer()*) may choose to set a response body regardless of the value of this property.

Type bool

title
Error title to send to the client.

Type str

description
Description of the error to send to the client.

Type str

headers
Extra headers to add to the response.

Type dict

link
An href that the client can provide to the user for getting help.

Type str

code
An internal application code that a user can reference when requesting support for the error.

Type int

Parameters **status** (*str*) – HTTP status code and text, such as "400 Bad Request"

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 '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`).

to_dict(*obj_type=<class 'dict'>*)
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()
Return a pretty-printed JSON representation of the error.

Returns A JSON document for the error.

Return type str

to_xml()
Return an XML-encoded representation of the error.

Returns An XML document for the error.

Return type str

Mixins

class falcon.http_error.**NoRepresentation**
Mixin for `HTTPError` child classes that have no representation.

This class can be mixed in when inheriting from `HTTPError`, in order to override the *has_representation* property such that it always returns `False`. This, in turn, will cause Falcon to return an empty response body to the client.

You can use this mixin when defining errors that either should not have a body (as dictated by HTTP standards or common practice), or in the case that a detailed error response may leak information to an attacker.

Note: This mixin class must appear before `HTTPError` in the base class list when defining the child; otherwise, it will not override the *has_representation* property as expected.

Predefined Errors

exception falcon.**HTTPBadRequest**(*title=None*, *description=None*, *headers=None*, ***kwargs*)
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)

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`).

exception falcon.**HTTPInvalidHeader**(*msg*, *header_name*, *headers=None*, ***kwargs*)

400 Bad Request.

One of the headers in the request is invalid.

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`).

exception falcon.**HTTPMissingHeader**(*header_name*, *headers=None*, ***kwargs*)

400 Bad Request

A header is missing from the request.

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`).

exception falcon.**HTTPInvalidParam**(*msg*, *param_name*, *headers=None*, ***kwargs*)

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.

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`).

exception falcon.**HTTPMissingParam**(*param_name*, *headers=None*, ***kwargs*)

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.

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`).

exception falcon.**HTTPUnauthorized**(*title=None*, *description=None*, *challenges=None*, *headers=None*, ***kwargs*)

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)

Keyword Arguments

- **title** (*str*) – Error title (default '401 Unauthorized').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **challenges** (*iterable of str*) – One or more authentication challenges to use as the value of the WWW-Authenticate header in the response.

  (See also: RFC 7235, Section 2.1)
- **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`).

exception falcon.**HTTPForbidden**(*title=None*, *description=None*, *headers=None*, ***kwargs*)

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)

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`).

exception falcon.**HTTPNotFound**(*headers=None*, ***kwargs*)

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)

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).

exception falcon.**HTTPMethodNotAllowed**(*allowed_methods*, *headers=None*, ***kwargs*)

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)

Parameters **allowed_methods** (*list of str*) – Allowed HTTP methods for this resource (e.g., ['GET', 'POST', 'HEAD']).

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`).

exception falcon.**HTTPNotAcceptable**(*description=None*, *headers=None*, ***kwargs*)

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)

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`).

exception falcon.**HTTPConflict**(*title=None*, *description=None*, *headers=None*, ***kwargs*)

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)

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`).

exception falcon.**HTTPGone**(*headers=None*, ***kwargs*)

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)

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).

exception falcon.**HTTPLengthRequired**(*title=None*, *description=None*, *headers=None*, ***kwargs*)

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)

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`).

exception falcon.**HTTPPreconditionFailed**(*title=None*, *description=None*, *headers=None*, ***kwargs*)

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)

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`).

exception falcon.**HTTPPayloadTooLarge**(*title=None*, *description=None*, *retry_after=None*, *headers=None*, ***kwargs*)

413 Payload 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)

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.
- **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.
- **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`).

exception falcon.**HTTPUriTooLong**(*title=None*, *description=None*, *headers=None*, ***kwargs*)

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)

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`).

exception falcon.**HTTPUnsupportedMediaType**(*description=None*, *headers=None*, ***kwargs*)

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)

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`).

exception falcon.**HTTPRangeNotSatisfiable**(*resource_length*, *headers=None*)

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)

Parameters **resource_length** – The maximum value for the last-byte-pos of a range request. Used to set the Content-Range header.

exception falcon.**HTTPUnprocessableEntity**(*title=None*, *description=None*, *headers=None*, ***kwargs*)

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)

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`).

exception falcon.**HTTPLocked**(*title=None*, *description=None*, *headers=None*, ***kwargs*)

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)

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`).

exception falcon.**HTTPFailedDependency**(*title=None*, *description=None*, *headers=None*, ***kwargs*)

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)

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`).

exception falcon.**HTTPPreconditionRequired**(*title=None*, *description=None*, *headers=None*, ***kwargs*)

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)

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`).

exception falcon.**HTTPTooManyRequests**(*title=None*, *description=None*, *retry_after=None*, *headers=None*, ***kwargs*)

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)

Keyword Arguments

- **title** (*str*) – Error title (default '429 Too Many Requests').
- **description** (*str*) – Human-friendly description of the rate limit that was exceeded.
- **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.
- **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`).

exception falcon.**HTTPRequestHeaderFieldsTooLarge**(*title=None*, *description=None*, *headers=None*, ***kwargs*)

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)

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`).

exception falcon.**HTTPUnavailableForLegalReasons**(*title=None*, *headers=None*, ***kwargs*)

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)

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`).

exception falcon.**HTTPInternalServerError**(*title=None*, *description=None*, *headers=None*, ***kwargs*)

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)

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`).

exception falcon.**HTTPNotImplemented**(*title=None*, *description=None*, *headers=None*, ***kwargs*)

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)

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`).

exception falcon.**HTTPBadGateway**(*title=None*, *description=None*, *headers=None*, ***kwargs*)

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)

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`).

exception falcon.**HTTPServiceUnavailable**(*title=None*, *description=None*, *retry_after=None*, *headers=None*, ***kwargs*)

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)

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.
- **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.
- **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`).

exception falcon.**HTTPGatewayTimeout**(*title=None*, *description=None*, *headers=None*, ***kwargs*)

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)

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`).

exception falcon.**HTTPVersionNotSupported**(*title=None*, *description=None*, *headers=None*, ***kwargs*)

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)

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`).

exception falcon.**HTTPInsufficientStorage**(*title=None*, *description=None*, *headers=None*, ***kwargs*)

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)

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`).

exception falcon.**HTTPLoopDetected**(*title=None*, *description=None*, *headers=None*, ***kwargs*)

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)

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`).

exception falcon.**HTTPNetworkAuthenticationRequired**(*title=None*, *description=None*, *headers=None*, ***kwargs*)

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)

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`).

5.2.6 Media

Falcon allows for easy and customizable internet media type handling. By default Falcon only enables a single JSON handler. However, additional handlers can be configured through the *falcon.RequestOptions* and *falcon.ResponseOptions* objects specified on your *falcon.API*.

Note: To avoid unnecessary overhead, Falcon will only process request media the first time the media property is referenced. Once it has been referenced, it'll use the cached result for subsequent interactions.

Usage

Zero configuration is needed if you're creating a JSON API. Just access or set the `media` attribute as appropriate and let Falcon do the heavy lifting for you.

```
import falcon


class EchoResource(object):
    def on_post(self, req, resp):
        message = req.media.get('message')

        resp.media = {'message': message}
        resp.status = falcon.HTTP_200
```

Warning: Once *media* is called on a request, it'll consume the request's stream.

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=None*, *resp_schema=None*)

Decorator for validating `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`.

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.

Parameters

- **req_schema** (*dict, optional*) – A dictionary that follows the JSON Schema specification. The request will be validated against this schema.

- **resp_schema** (*dict, optional*) – A dictionary that follows the JSON Schema specification. The response will be validated against this schema.

Example

```
from falcon.media.validators import jsonschema

# -- snip --

@jsonschema.validate(my_post_schema)
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 `media` attribute is used it attempts to de/serialize based on the `Content-Type` header value. The missing link that Falcon doesn't provide is the connection between the *falcon.Request* `Accept` header provided by a user and the *falcon.Response* `Content-Type` header.

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:

```
class NegotiationMiddleware(object):
    def process_request(self, req, resp):
        resp.content_type = req.accept
```

Replacing the Default Handlers

When creating your API object you can either add or completely replace all of the handlers. For example, lets 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 create a new *Handlers* object specifying the desired media type and a handler that can process that data.

```
import falcon
from falcon import media


handlers = media.Handlers({
    'application/msgpack': media.MessagePackHandler(),
})

api = falcon.API(media_type='application/msgpack')

api.req_options.media_handlers = handlers
api.resp_options.media_handlers = handlers
```

Alternatively, if you would like to add an additional handler such as MessagePack, this can be easily done in the following manner:

```
import falcon
from falcon import media


extra_handlers = {
    'application/msgpack': media.MessagePackHandler(),
}

api = falcon.API()

api.req_options.media_handlers.update(extra_handlers)
api.resp_options.media_handlers.update(extra_handlers)
```

Supported Handler Types

class falcon.media.**JSONHandler**(*dumps=None*, *loads=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 *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.JSONHandler(
    dumps=rapidjson.dumps,
    loads=rapidjson.loads,
)
extra_handlers = {
    'application/json': json_handler,
}

api = falcon.API()
api.req_options.media_handlers.update(extra_handlers)
api.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. 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
```

(continues on next page)

(continued from previous page)

```
import rapidjson

json_handler = media.JSONHandler(
    dumps=partial(
        rapidjson.dumps,
        ensure_ascii=False, sort_keys=True
    ),
)
```

Keyword Arguments

- **dumps** (*func*) – Function to use when serializing JSON responses.
- **loads** (*func*) – Function to use when deserializing JSON requests.

deserialize(*stream*, *content_type*, *content_length*)
Deserialize the *falcon.Request* body.

Parameters

- **stream** (*object*) – Input data to deserialize.
- **content_type** (*str*) – Type of request content.
- **content_length** (*int*) – Length of request content.

Returns A deserialized object.

Return type object

serialize(*media*, *content_type*)
Serialize the media object on a *falcon.Response*

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

class falcon.media.**MessagePackHandler**
Handler built using the `msgpack` module.

This handler uses `msgpack.unpackb()` and `msgpack.packb()`. The MessagePack `bin` type is used to distinguish between Unicode strings (`str` on Python 3, `unicode` on Python 2) and byte strings (`bytes` on Python 2/3, or `str` on Python 2).

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
```

---

deserialize(*stream*, *content_type*, *content_length*)
Deserialize the *falcon.Request* body.

Parameters

- **stream** (*object*) – Input data to deserialize.

- **content_type** (*str*) – Type of request content.
- **content_length** (*int*) – Length of request content.

Returns A deserialized object.

Return type object

serialize(*media*, *content_type*)
Serialize the media object on a *falcon.Response*

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

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:

class falcon.media.**BaseHandler**
Abstract Base Class for an internet media type handler

serialize(*media*, *content_type*)
Serialize the media object on a *falcon.Response*

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*, *content_type*, *content_length*)
Deserialize the *falcon.Request* body.

Parameters

- **stream** (*object*) – Input data to deserialize.
- **content_type** (*str*) – Type of request content.
- **content_length** (*int*) – Length of request content.

Returns A deserialized object.

Return type object

Handlers

class falcon.media.**Handlers**(*initial=None*)
A dictionary like object that manages internet media type handlers.

Media Type Constants

The `falcon` module provides a number of constants for common media types, including the following:

```
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
falcon.MEDIA_GIF
```

5.2.7 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*

Redirects

exception falcon.**HTTPMovedPermanently**(*location*, *headers=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*, *headers=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*, *headers=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*, *headers=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*, *headers=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.

5.2.8 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. Components are registered with the *middleware* kwarg when instantiating Falcon's *API class*.

Note: Unlike hooks, middleware methods apply globally to the entire API.

Falcon's middleware interface is defined as follows:

```
class ExampleComponent(object):
    def process_request(self, req, resp):
        """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, resp, resource, params):
        """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.
        """

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

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.API*. 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* middleware method may short-circuit further request processing by setting `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 `complete` to `True` from within a *process_resource* method will short-circuit further request processing at that point.

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 resource responder raises an error. This behavior is controlled by the *API 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.

5.2.9 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('Bad request', 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:

```
@falcon.before(extract_project_id)
class Message(object):
    def on_post(self, req, resp, project_id):
```

(continues on next page)

(continued from previous page)

```
        pass

    def on_get(self, req, resp, project_id):
        pass
```

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('Bad request', 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(object):
    def __init__(self, roles):
        self._roles = roles

    def __call__(self, req, resp, resource, params):
        pass

@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.

`falcon.`**`before`**(*action*, **args*, ***kwargs*)

Decorator to execute the given action function *before* the responder.

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.

  Note: 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('Invalid ID',
                                    'ID was not valid.')

    params['answer'] = 42
```

- ***args** – Any additional arguments will be passed to *action* in the order given, immediately following the *req*, *resp*, *resource*, and *params* arguments.
- ****kwargs** – Any additional keyword arguments will be passed through to *action*.

falcon.**after**(*action*, **args*, ***kwargs*)

Decorator to 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*, *resource*, and *params* arguments.
- ****kwargs** – Any additional keyword arguments will be passed through to *action*.

5.2.10 Routing

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.

If no route matches the request, control then passes to a default responder that simply raises an instance of *HTTPNotFound*. Normally this will result in sending a 404 response back to the client.

Here's a quick example to show how all the pieces fit together:

```
import json

import falcon

class ImagesResource(object):

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

        # Create a JSON representation of the resource
        resp.body = 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
```

(continues on next page)

(continued from previous page)

```
        # illustrate how this may be overridden as needed.
        resp.status = falcon.HTTP_200

api = application = falcon.API()

images = ImagesResource()
api.add_route('/images', images)
```

Default Router

Falcon's default routing engine is based on a decision tree that is first compiled into Python code, and then evaluated by the runtime.

The *add_route()* method is 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
```

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 youself 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 be routed to:

```
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 walkthough 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()
api.add_route(
    '/a/{some_field:int}',
    some_resource
)

# IntConverter(8)
api.add_route(
    '/b/{some_field:int(8)}',
    some_resource
)

# IntConverter(8, min=10000000)
api.add_route(
    '/c/{some_field:int(8, min=10000000)}',
    some_resource
)
```

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")}

class falcon.routing.**IntConverter**(*num_digits=None*, *min=None*, *max=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*)
Convert a URI template field value to another format or type.

Parameters **value** (*str*) – Original string to convert.

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*)
Convert a URI template field value to another format or type.

Parameters **value** (*str*) – Original string to convert.

Returns

Converted field value, or None if the field can not be converted.

Return type object

class falcon.routing.**DateTimeConverter**(*format_string='%Y-%m-%dT%H:%M:%SZ'*)
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:%SZ'`).

convert(*value*)
Convert a URI template field value to another format or type.

Parameters **value** (*str*) – Original string to convert.

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.

convert(*value*)

Convert a URI template field value to another format or type.

Parameters **value** (*str*) – Original string to convert.

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.API()*. For example:

```
router = MyRouter()
api = API(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(object):
    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
```

(continues on next page)

(continued from previous page)

```
                the resource.

            **kwargs (dict): Accepts any additional keyword arguments
                that were originally passed to the falcon.API.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
                    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.

        """
```

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.API 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.

add_route(*uri_template*, *resource*, ***kwargs*)

Adds 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.

find(*uri*, *req=None*)

Search for a route that matches the given partial URI.

Parameters **uri** (*str*) – The requested path to route.

Keyword Arguments **req** (Request) – The 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*, ***kwargs*)

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*, *suffix=None*)

Maps 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*)
Maps 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.

falcon.routing.**compile_uri_template**(*template*)
Compile the given URI template string into a pattern matcher.

This function can be used to construct custom routing engines that iterate through a list of possible routes, attempting to match an incoming request against each route's compiled regular expression.

Each field is converted to a named group, so that when a match is found, the fields can be easily extracted using `re.MatchObject.groupdict()`.

This function does not support the more flexible templating syntax used in the default router. Only simple paths with bracketed field expressions are recognized. For example:

```
/
/books
/books/{isbn}
/books/{isbn}/characters
/books/{isbn}/characters/{name}
```

Also, note that if the template contains a trailing slash character, it will be stripped in order to normalize the routing logic.

Parameters **template** (*str*) – The template to compile. Note that field names are restricted to ASCII a-z, A-Z, and the underscore character.

Returns (template_field_names, template_regex)

Return type tuple

5.2.11 Utilities

URI Functions

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.**encode**(*uri*)
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. If this is a wide string (i.e., `compat.text_type`), it will be encoded to a UTF-8 byte array and any multibyte sequences will be percent-encoded as-is.

Returns An escaped version of *uri*, where all disallowed characters have been percent-encoded.

Return type str

`falcon.uri.`**`encode_value`**(*uri*)

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 escaped. If *value* is a wide string (i.e., `compat.text_type`), it will be encoded to a UTF-8 byte array and any multibyte sequences will be percent-encoded as-is.

Returns An escaped version of *uri*, where all disallowed characters have been percent-encoded.

Return type str

`falcon.uri.`**`decode`**(*encoded_uri*, *unquote_plus=True*)

Decodes 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.`**`parse_host`**(*host*, *default_port=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*, *keep_blank=False*, *csv=True*)
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_qs_values* 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').

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 `False` in order to disable splitting query parameters on `,` (default `True`). Depending on the user agent, encoding lists as multiple occurrences of the same parameter might be preferable. In this case, setting *parse_qs_csv* to `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*)
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`.

Miscellaneous

falcon.**deprecated**(*instructions*)
Flags 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(...)'

falcon.**http_now**()
Returns 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*)
Converts 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*, *obs_date=False*)
Converts 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

falcon.**to_query_str**(*params*, *comma_delimited_lists=True*, *prefix=True*)
Converts 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_http_status**(*status_code*, *default_reason='Unknown'*)
Gets both the http status code and description from just a code

Parameters

- **status_code** – integer or string that can be converted to an integer
- **default_reason** – default text to be appended to the status_code if the lookup does not find a result

Returns status code e.g. "404 Not Found"

Return type str

Raises `ValueError` – the value entered could not be converted to an integer

falcon.**get_bound_method**(*obj*, *method_name*)
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).

class falcon.**TimezoneGMT**
GMT timezone class implementing the datetime.tzinfo interface.

dst(*dt*)
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*)
Get the name of this timezone.

Parameters **dt** (*datetime.datetime*) – Ignored

Returns "GMT"

Return type str

utcoffset(*dt*)
Get the offset from UTC.

Parameters **dt** (*datetime.datetime*) – Ignored

Returns GMT offset, which is equivalent to UTC and so is aways 0.

Return type datetime.timedelta

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 []):
        if etag == '*' or etag == content_etag:
            resp.status = falcon.HTTP_304
            return
```

(continues on next page)

(continued from previous page)

```
    # ...

    resp.etag = content_etag
    resp.status = falcon.HTTP_200
```

(See also: RFC 7232)

is_weak
`True` if the entity-tag is weak, otherwise `False`.

Type bool

dumps()
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

classmethod loads(*etag_str*)
Class method that deserializes 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*)
Performs 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

5.2.12 Testing

Reference

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:

```
# -----------------------------------------------------------------
# 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.API` instance.
        self.app = myapp.create()


class TestMyApp(MyTestCase):
    def test_get_message(self):
        doc = {u'message': u'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.API` instance.
    return testing.TestClient(myapp.create())


def test_get_message(client):
    doc = {u'message': u'Hello world!'}

    result = client.simulate_get('/messages/42')
    assert result.json == doc
```

class falcon.testing.**Result**(*iterable*, *status*, *headers*)

Encapsulates the result of a simulated WSGI 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

status
HTTP status string given in the response

Type str

status_code
The code portion of the HTTP status string

Type int

headers
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*.

Type CaseInsensitiveDict

cookies
A dictionary of *falcon.testing.Cookie* values parsed from the response, by name.

Type dict

encoding
Text encoding of the response body, or `None` if the encoding can not be determined.

Type str

content
Raw response body, or `bytes` if the response body was empty.

Type bytes

text
Decoded response body of type `unicode` under Python 2.7, and of type `str` otherwise. If the content type does not specify an encoding, UTF-8 is assumed.

Type str

json
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.

Type JSON serializable

class falcon.testing.**Cookie**(*morsel*)
Represents a cookie returned by a simulated request.

Parameters **morsel** – A `Morsel` object from which to derive the cookie data.

name
The cookie's name.

Type str

value
The value of the cookie.

Type str

expires
Expiration timestamp for the cookie, or `None` if not specified.

Type datetime.datetime

path
The path prefix to which this cookie is restricted, or `None` if not specified.

Type str

domain
The domain to which this cookie is restricted, or `None` if not specified.

Type str

max_age
The lifetime of the cookie in seconds, or `None` if not specified.

Type int

secure
Whether or not the cookie may only only be transmitted from the client via HTTPS.

Type bool

http_only
Whether or not the cookie may only be included in unscripted requests from the client.

Type bool

falcon.testing.**simulate_request**(*app*, *method='GET'*, *path='/'*, *query_string=None*, *headers=None*, *body=None*, *json=None*, *file_wrapper=None*, *wsgierrors=None*, *params=None*, *params_csv=True*, *protocol='http'*, *host='falconframework.org'*, *remote_addr=None*, *extras=None*)

Simulates a request to a WSGI application.

Performs a request against a WSGI application. Uses `wsgiref.validate` to ensure the response is valid WSGI.

Keyword Arguments

- **app** (*callable*) – The WSGI 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.

- **protocol** – The protocol to use for the URL scheme (default: 'http')
- **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 `False` to encode list values in query string params 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`.
- **query_string** (*str*) – A raw query string to include in the request (default: `None`). If specified, overrides *params*.
- **headers** (*dict*) – Additional headers to include in the request (default: `None`)
- **body** (*str*) – A string to send as the body of the request. Accepts both byte strings and Unicode strings (default: `None`). If a Unicode string is provided, it will be encoded as UTF-8 in the request.
- **json** (*JSON serializable*) – A JSON document to serialize as the body of the request (default: `None`). If specified, overrides *body* and the Content-Type header in *headers*.
- **file_wrapper** (*callable*) – Callable that returns an iterable, to be used as the value for *wsgi.file_wrapper* in the 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')
- **wsgierrors** (*io*) – The stream to use as *wsgierrors* (default `sys.stderr`)
- **extras** (*dict*) – Additional CGI variables to add to the WSGI `environ` dictionary for the request (default: `None`)

Returns The result of the request

Return type *Result*

`falcon.testing.`**simulate_get**(*app*, *path*, ***kwargs*)

Simulates a GET request to a WSGI application.

Equivalent to:

```
simulate_request(app, 'GET', path, **kwargs)
```

Parameters

- **app** (*callable*) – The WSGI 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

- **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 `False` to encode list values in query string params 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`.

- **query_string** (*str*) – A raw query string to include in the request (default: `None`). If specified, overrides *params*.
- **headers** (*dict*) – Additional headers to include in the request (default: `None`)
- **file_wrapper** (*callable*) – Callable that returns an iterable, to be used as the value for *wsgi.file_wrapper* in the environ (default: `None`). This can be used to test high-performance file transmission when *resp.stream* is set to a file-like object.
- **protocol** – The protocol to use for the URL scheme (default: 'http')
- **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')
- **extras** (*dict*) – Additional CGI variables to add to the WSGI `environ` dictionary for the request (default: `None`)

`falcon.testing.`**simulate_head**(*app*, *path*, ***kwargs*)

Simulates a HEAD request to a WSGI application.

Equivalent to:

```
simulate_request(app, 'HEAD', path, **kwargs)
```

Parameters

- **app** (*callable*) – The WSGI 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

- **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 `False` to encode list values in query string params 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`.
- **query_string** (*str*) – A raw query string to include in the request (default: `None`). If specified, overrides *params*.
- **headers** (*dict*) – Additional headers to include in the request (default: `None`)
- **protocol** – The protocol to use for the URL scheme (default: 'http')
- **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')
- **extras** (*dict*) – Additional CGI variables to add to the WSGI `environ` dictionary for the request (default: `None`)

`falcon.testing.`**simulate_post**(*app*, *path*, ***kwargs*)

Simulates a POST request to a WSGI application.

Equivalent to:

```
simulate_request(app, 'POST', path, **kwargs)
```

Parameters

- **app** (*callable*) – The WSGI application to call
- **path** (*str*) – The URL path to request

Keyword Arguments

- **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 `False` to encode list values in query string params 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`.
- **headers** (*dict*) – Additional headers to include in the request (default: `None`)
- **body** (*str*) – A string to send as the body of the request. Accepts both byte strings and Unicode strings (default: `None`). If a Unicode string is provided, it will be encoded as UTF-8 in the request.
- **json** (*JSON serializable*) – A JSON document to serialize as the body of the request (default: `None`). If specified, overrides *body* and the Content-Type header in *headers*.
- **protocol** – The protocol to use for the URL scheme (default: 'http')
- **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')
- **extras** (*dict*) – Additional CGI variables to add to the WSGI `environ` dictionary for the request (default: `None`)

`falcon.testing.`**simulate_put**(*app*, *path*, ***kwargs*)

Simulates a PUT request to a WSGI application.

Equivalent to:

```
simulate_request(app, 'PUT', path, **kwargs)
```

Parameters

- **app** (*callable*) – The WSGI application to call

- **path** (*str*) – The URL path to request

Keyword Arguments

- **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 `False` to encode list values in query string params 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`.
- **headers** (*dict*) – Additional headers to include in the request (default: `None`)
- **body** (*str*) – A string to send as the body of the request. Accepts both byte strings and Unicode strings (default: `None`). If a Unicode string is provided, it will be encoded as UTF-8 in the request.
- **json** (*JSON serializable*) – A JSON document to serialize as the body of the request (default: `None`). If specified, overrides *body* and the Content-Type header in *headers*.
- **protocol** – The protocol to use for the URL scheme (default: 'http')
- **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')
- **extras** (*dict*) – Additional CGI variables to add to the WSGI `environ` dictionary for the request (default: `None`)

`falcon.testing.`**simulate_options**(*app*, *path*, ***kwargs*)

Simulates an OPTIONS request to a WSGI application.

Equivalent to:

```
simulate_request(app, 'OPTIONS', path, **kwargs)
```

Parameters

- **app** (*callable*) – The WSGI application to call
- **path** (*str*) – The URL path to request

Keyword Arguments

- **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 `False` to encode list values in query string params 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`.
- **headers** (*dict*) – Additional headers to include in the request (default: `None`)
- **protocol** – The protocol to use for the URL scheme (default: 'http')

- **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')
- **extras** (*dict*) – Additional CGI variables to add to the WSGI `environ` dictionary for the request (default: `None`)

`falcon.testing.`**simulate_patch**(*app*, *path*, ***kwargs*)

Simulates a PATCH request to a WSGI application.

Equivalent to:

```
simulate_request(app, 'PATCH', path, **kwargs)
```

Parameters

- **app** (*callable*) – The WSGI application to call
- **path** (*str*) – The URL path to request

Keyword Arguments

- **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 `False` to encode list values in query string params 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`.
- **headers** (*dict*) – Additional headers to include in the request (default: `None`)
- **body** (*str*) – A string to send as the body of the request. Accepts both byte strings and Unicode strings (default: `None`). If a Unicode string is provided, it will be encoded as UTF-8 in the request.
- **json** (*JSON serializable*) – A JSON document to serialize as the body of the request (default: `None`). If specified, overrides *body* and the Content-Type header in *headers*.
- **protocol** – The protocol to use for the URL scheme (default: 'http')
- **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')
- **extras** (*dict*) – Additional CGI variables to add to the WSGI `environ` dictionary for the request (default: `None`)

`falcon.testing.`**simulate_delete**(*app*, *path*, ***kwargs*)

Simulates a DELETE request to a WSGI application.

Equivalent to:

```
simulate_request(app, 'DELETE', path, **kwargs)
```

Parameters

- **app** (*callable*) – The WSGI application to call
- **path** (*str*) – The URL path to request

Keyword Arguments

- **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 `False` to encode list values in query string params 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`.
- **headers** (*dict*) – Additional headers to include in the request (default: `None`)
- **protocol** – The protocol to use for the URL scheme (default: 'http')
- **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')
- **extras** (*dict*) – Additional CGI variables to add to the WSGI `environ` dictionary for the request (default: `None`)

class falcon.testing.**TestClient**(*app*, *headers=None*)

Simulates requests to a WSGI 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')
```

Note: The methods all call `self.simulate_request()` for convenient overriding of request preparation by child classes.

Parameters **app** (*callable*) – A WSGI 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.

simulate_delete(*path='/'*, ***kwargs*)

Simulates a DELETE request to a WSGI application.

(See also: *falcon.testing.simulate_delete()*)

simulate_get(*path='/'*, ***kwargs*)
Simulates a GET request to a WSGI application.

(See also: *falcon.testing.simulate_get()*)

simulate_head(*path='/'*, ***kwargs*)
Simulates a HEAD request to a WSGI application.

(See also: *falcon.testing.simulate_head()*)

simulate_options(*path='/'*, ***kwargs*)
Simulates an OPTIONS request to a WSGI application.

(See also: *falcon.testing.simulate_options()*)

simulate_patch(*path='/'*, ***kwargs*)
Simulates a PATCH request to a WSGI application.

(See also: *falcon.testing.simulate_patch()*)

simulate_post(*path='/'*, ***kwargs*)
Simulates a POST request to a WSGI application.

(See also: *falcon.testing.simulate_post()*)

simulate_put(*path='/'*, ***kwargs*)
Simulates a PUT request to a WSGI application.

(See also: *falcon.testing.simulate_put()*)

simulate_request(**args*, ***kwargs*)
Simulates 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.**TestCase**(*methodName='runTest'*)
Extends `unittest` to support WSGI 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 calls 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
A WSGI application to target when simulating requests (default: `falcon.API()`). When testing your application, you will need to set this to your own instance of `falcon.API`. For example:

```
from falcon import testing
import myapp


class MyTestCase(testing.TestCase):
    def setUp(self):
        super(MyTestCase, self).setUp()
```

(continues on next page)

(continued from previous page)

```
        # Assume the hypothetical `myapp` package has a
        # function called `create()` to initialize and
        # return a `falcon.API` instance.
        self.app = myapp.create()


class TestMyApp(MyTestCase):
    def test_get_message(self):
        doc = {u'message': u'Hello world!'}

        result = self.simulate_get('/messages/42')
        self.assertEqual(result.json, doc)
```

Type object

setUp()
Hook method for setting up the test fixture before exercising it.

class falcon.testing.**SimpleTestResource**(*status=None*, *body=None*, *json=None*, *headers=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

called
Whether or not a req/resp was captured.

Type bool

captured_req
The last Request object passed into any one of the responder methods.

Type *falcon.Request*

captured_resp
The last Response object passed into any one of the responder methods.

Type *falcon.Response*

captured_kwargs
The last dictionary of kwargs, beyond `req` and `resp`, that were passed into any one of the responder methods.

Type dict

class falcon.testing.**StartResponseMock**
Mock object representing a WSGI *start_response* callable.

call_count
Number of times *start_response* was called.

Type int

status
HTTP status line, e.g. '785 TPS Cover Sheet not attached'.

Type str

headers
Raw headers list passed to *start_response*, per PEP-333.

Type list

headers_dict
Headers as a case-insensitive `dict`-like object, instead of a `list`.

Type dict

falcon.testing.**capture_responder_args**(*req*, *resp*, *resource*, *params*)
Before hook for capturing responder arguments.

Adds the following attributes to the hooked responder's resource class:

- captured_req
- captured_resp
- captured_kwargs

falcon.testing.**rand_string**(*min*, *max*)
Returns 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.**create_environ**(*path='/'*, *query_string=''*, *protocol='HTTP/1.1'*, *scheme='http'*, *host='falconframework.org'*, *port=None*, *headers=None*, *app=''*, *body=''*, *method='GET'*, *wsgierrors=None*, *file_wrapper=None*, *remote_addr=None*)
Creates 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 '')
- **protocol** (*str*) – The HTTP protocol to simulate (default 'HTTP/1.1'). If set to 'HTTP/1.0', the Host header will not be added to the environment.
- **scheme** (*str*) – URL scheme, either 'http' or 'https' (default 'http')
- **host** (*str*) – Hostname for the request (default 'falconframework.org')

- **port** (*str*) – The TCP port to simulate. Defaults to the standard port used by the given scheme (i.e., 80 for 'http' and 443 for 'https').
- **headers** (*dict*) – Headers as a `dict` or an iterable yielding (*key*, *value*) `tuple`'s
- **app** (*str*) – Value for the `SCRIPT_NAME` environ variable, described in PEP-333: '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 '')
- **body** (*str*) – The body of the request (default ''). Accepts both byte strings and Unicode strings. Unicode strings are encoded as UTF-8 in the request.
- **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 (default '127.0.0.1')

`falcon.testing.`**redirected**(*stdout=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, *stderr=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>*)

A context manager to temporarily redirect stdout or stderr

e.g.:

with redirected(stderr=os.devnull): ...

5.3 Deployment Guide

5.3.1 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.

5.3.2 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 BadThings:sup:(tm) 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.

```
# For Python 3
$ python3 -m venv /home/myproject/venv

# For Python 2
$ virtualenv /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_api()` function or a module-level script. This role of this function or script is to supply an instance of *falcon.API*, which implements the standard WSGI callable interface.

You will need to expose the *falcon.API* 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.API* instance.

Listing 1: /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'])
```

(continues on next page)

(continued from previous page)

```
# uWSGI will look for this variable
application = myproject.create_api(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 `12.0.0.1:8080`.

Listing 2: /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 3: /etc/nginx/sites-avaiable/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;

  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.

5.4 Community Guide

5.4.1 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.

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.

5.4.2 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.

Kurt Griffiths (**kgriffs** on GH, Gitter, and Twitter) is the original creator of the Falcon framework, and currently co-maintains the project along with John Vrbanac (**jmvrbanac** on GH and Gitter, and **jvrbanac** on Twitter). Falcon is developed by a growing community of users and contributors just like you.

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 check out our Contributor's Guide for more information.

Thanks!

5.5 Changelogs

5.5.1 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 subclass of dict.
- Response *context_type* was changed from dict to a subclass of dict.
- *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 1.4. 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
```

(continues on next page)

(continued from previous page)

```
req.context.role = 'trial'
req.context.user = 'guest'
```

To ease the migration path, the previous behavior is supported by subclassing dict, however, as of Falcon 2.0, the dict context interface is considered deprecated, and may be removed in a future release. It is also noteworthy that object attributes and dict items are not automagically linked in any special way, and setting one does not affect the other.

Applications can work around this change by explicitly overriding *context_type* to dict.

- 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 subclassing dict, however, as of Falcon 2.0, the dict context interface is considered deprecated, and may be removed in a future release. It is also noteworthy that object attributes and dict items are not automagically linked in any special way, and setting one does not affect the other.

Applications can work around this change by explicitly overriding *context_type* to dict.

- *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.

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

5.5.2 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.

5.5.3 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

5.5.4 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.

5.5.5 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.

5.5.6 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.

5.5.7 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 *Request.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 *Request.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.

5.5.8 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.

5.5.9 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.

Python Module Index

f

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V