Falcon Documentation

Release 0.2.0rc1

Kurt Griffiths et al.

November 23, 2016

Contents

Release v0.2. (*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 CatalogItem(object):

    # ...

    @falcon.before(hooks.to_oid)
    def on_get(self, id):
        return self._collection.find_one(id)

app = falcon.API(after=[hooks.serialize])
app.add_route('/items/{id}', CatalogItem())
```

CHAPTER 1

What People are Saying

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

"I feel like I'm just talking HTTP at last, with nothing in the middle. Falcon seems like the *requests* of backend."

"The source code for falcon is so good, I almost prefer it to documentation. It basically can't be wrong."

"What other framework has integrated support for '786 TRY IT NOW' ?"

CHAPTER 2

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
- Only depends on six and mimeparse
- Python 2.6, 2.7, 3.3, 3.4 + PyPy

CHAPTER 3

Useful Links

- Falcon Home
- Falcon @ PyPI
- Falcon @ GitHub

CHAPTER 4

Resources

- An Unladen Web Framework
- The Definitive Introduction to Falcon

CHAPTER 5

Documentation

5.1 Community Guide

5.1.1 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!

IRC

While you experiment with Falcon and work to familiarize yourself with the WSGI framework, please consider joining the **#falconframework** IRC channel on Freenode. It's a great place to ask questions, share ideas, and get the scoop on what's new.

Mailing List

The Falcon community maintains a mailing list that you can use to share your ideas and ask questions about the framework. We use the appropriately minimalistic Librelist to host the discussions.

Subscribing is super easy and doesn't require any account setup. Simply send an email to falcon@librelist.com and follow the instructions in the reply. For more information about managing your subscription, check out the Librelist help page.

While we don't have an official code of conduct, we do expect everyone who participates on the mailing list 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.

Discussions are archived for posterity.

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 **#falconframework** and/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 IRC, and to peer-review pull requests. If you use the Chrome browser, we recommend installing the NotHub extension to stay up to date with PRs.

5.1.2 Contribute to Falcon

Kurt Griffiths is the creator and current maintainer of the Falcon framework. He works with a growing team of friendly and stylish volunteers like yourself, who review patches, implement features, fix bugs, and write docs for the project.

Your ideas and patches are always welcome!

IRC

If you are interested in helping out, please join the **#falconframework** IRC channel on Freenode. It's the best way to discuss ideas, ask questions, and generally stay in touch with fellow contributors. We recommend setting up a good IRC bouncer, such as ZNC, which can record and play back any conversations that happen when you are away.

Mailing List

The Falcon community maintains a mailing list that you can use to share your ideas and ask questions about the framework. We use the appropriately minimalistic Librelist to host the discussions.

Subscribing is super easy and doesn't require any account setup. Simply send an email to falcon@librelist.com and follow the instructions in the reply. For more information about managing your subscription, check out the Librelist help page.

While we don't have an official code of conduct, we do expect everyone who participates on the mailing list 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.

Discussions are archived for posterity.

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 **#falconframework** and/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 IRC, and to peer-review pull requests. If you use the Chrome browser, we recommend installing the NotHub extension to stay up to date with PRs.

Pull Requests

Before submitting a pull request, please ensure you have added new tests and updated existing ones as appropriate. We require 100% code coverage. Also, please ensure your coding style follows PEP 8 and doesn't make pyflakes sad.

Additional Style Rules

- Docstrings are required for classes, attributes, methods, and functions.

- Use napolean-flavored dosctrings to make them readable both when using the *help* function within a REPL, and when browsing them on *Read the Docs*.
- Format non-trivial comments using your GitHub nick and an appropriate prefix. Here are some examples:

```
# TODO(riker): Damage report!
# NOTE(riker): Well, that's certainly good to know.
# PERF(riker): Travel time to the nearest starbase?
# APPSEC(riker): In all trust, there is the possibility for betrayal.
```

- Commit messages should be formatted using AngularJS conventions (one-liners are OK for now but bodies and footers may be required as the project matures).
- When catching exceptions, name the variable `ex`.
- Use whitespace to separate logical blocks of code and to improve readability.
- Do not use single-character variable names except for trivial indexes when looping, or in mathematical expressions implementing well-known formulae.
- Heavily document code that is especially complex or clever!
- When in doubt, optimize for readability.

5.1.3 FAQ

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.

Why doesn't Falcon include X?

The Python ecosytem offers a bunch of great libraries that you are welcome to use from within your responder, hooks, and middleware. Falcon doesn't try to dictate what you should use, since that would take away your freedom to choose the best tool for the job.

The Falcon framework lets you decide your own answers to questions like:

- gevent or asyncio?
- JSON or MessagePack?
- konval or jsonschema?
- Mongothon or Monk?
- Storm, SQLAlchemy or peewee?
- Jinja or Tenjin?
- python-multipart or cgi.FieldStorage?

How do I authenticate requests?

Hooks and/or middleware components can be used to to authenticate and authorize requests. For example, you could create a middleware component that parses incoming credentials and places the result in `req.context`. Downstream components or hooks could then use this info to authenticate the user, and then finally authorize the request, taking into account the user's role and the requested resource.

Tip: The Talons project maintains a collection of auth plugins for the Falcon framework.

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. 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 server all requests coming in on that route.

Is Falcon thread-safe?

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. Therefore, as long as you are careful about the way responders access class member variables to avoid conflicts, your WSGI app should be thread-safe.

That being said, Falcon-based services are usually deployed using green threads (via 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 haven't been discovered yet.

Caveat emptor!

How do I implement both POSTing and GETing items for the same resource?

Suppose you wanted to implement the following endpoints:

```
# Resource Collection
POST /resources
GET /resources{?marker, limit}

# 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.

The Falcon community did some experimenting with routing both singleton and collection-based operations to the same Python class, but it turned out to make routing definitions more complicated and less intuitive. That being said, we are always open to new ideas, so please let us know if you discover another way.

See also *this section of the tutorial*.

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 add custom data to the `req.context` dict, as a way of passing contextual information around.

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 leave off *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-333 prohibits apps from setting hop-by-hop headers itself, such as Transfer-Encoding.

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 you body isn't being returned to the client, it's probably a bug! Let us know in IRC or on the mailing list so we can help.

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 (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. 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 `falcon.HTTPError` unless you have registered a custom error handler for that type (see also: *falcon.API*).

Why are trailing slashes trimmed from req.path?

Falcon normalizes incoming URI paths to simplify later processing and improve the predictability of application logic. In addition to stripping a trailing slashes, if any, Falcon will convert empty paths to "/".

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 are field names in URI templates restricted to certain characters?

Field names are restricted to the ASCII characters in the set `[a-zA-Z_]`. Using a restricted set of characters allows the framework to make simplifying assumptions that reduce the overhead of parsing incoming requests.

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` through the *req_options* attribute. For example:

```
api.req_options.keep_blank_qs_values = True
```

5.2 User Guide

5.2.1 Introduction

Falcon is a minimalist, high-performance web framework for building RESTful services and app backends with Python. Falcon works with any WSGI container that is compliant with PEP-3333, and works great with Python 2.6, Python 2.7, Python 3.3, Python 3.4 and PyPy, giving you a wide variety of deployment options.

How is Falcon different?

First, Falcon is one of the fastest WSGI frameworks available. When there is a conflict between saving the developer a few keystrokes and saving a few microseconds to serve a request, Falcon is strongly biased toward the latter. That being said, Falcon strives to strike a good balance between usability and speed.

Second, Falcon is lean. It doesn't try to be everything to everyone, focusing instead on a single use case: HTTP APIs. Falcon doesn't include a template engine, form helpers, or an ORM (although those are easy enough to add yourself). When you sit down to write a web service with Falcon, you choose your own adventure in terms of async I/O, serialization, data access, etc. In fact, Falcon only has two dependencies: six, to make it easier to support both Python 2 and 3, and mimeparse for handling complex Accept headers. Neither of these packages pull in any further dependencies of their own.

Third, Falcon eschews magic. When you use the framework, it's pretty obvious which inputs lead to which outputs. Also, it's blatantly obvious where variables originate. All this makes it easier to reason about the code and to debug edge cases in large-scale deployments of your application.

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

5.2.2 Installation

Install from PyPI

If available, Falcon will compile itself with Cython for an extra speed boost. The following will make sure Cython is installed first, and that you always have the latest and greatest.

```
$ pip install --upgrade cython falcon
```

Note that if you are running on PyPy, you won't need Cython, so you can just type:

```
$ pip install --upgrade falcon
```

Installing Cython on OS X

In order to get Cython working on OS X Mavericks with Xcode 5.1, you will first need to set up Xcode Command Line Tools. Install them with this command:

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

The Xcode 5.1 CLang compiler treats unrecognized command-line options as errors; this can cause problems under Python 2.6, for example:

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

You can work around errors caused by unused arguments by setting some environment variables:

```
$ export CFLAGS=-Qunused-arguments
$ export CPPFLAGS=-Qunused-arguments
$ pip install cython falcon
```

WSGI Server

Falcon speaks WSGI. If you want to actually serve a Falcon app, you will want a good WSGI server. Gunicorn and uWSGI are some of the more popular ones out there, but anything that can load a WSGI app will do. Gevent is an async library that works well with both Gunicorn and uWSGI.

```
$ pip install --upgrade gevent [gunicorn|uwsgi]
```

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 repro 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 .
```

Did we mention we love pull requests? :)

5.2.3 Quickstart

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

The Big Picture

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:
    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
```

Then, in another terminal:

```
$ curl 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('X-Auth-Token')
        project = req.get_header('X-Project-ID')

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

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

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

            raise falcon.HTTPUnauthorized('Authentication required',
                                          description,
                                          href='http://docs.example.com/auth',
                                          scheme='Token; UUID')

    def _token_is_valid(self, token, project):
        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):

    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'))

        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 'result' not in req.context:
            return

        resp.body = json.dumps(req.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.HTTPRequestEntityTooLarge(
                'Request body is too large', msg)

    return hook


class ThingsResource:

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

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

        try:
```

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

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

            raise falcon.HTTPServiceUnavailable(
                '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.
        req.context['result'] = result

        resp.set_header('X-Powered-By', 'Small Furry Creatures')
        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 KeyError:
            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()
if __name__ == '__main__':
    httpd = simple_server.make_server('127.0.0.1', 8000, app)
    httpd.serve_forever()
```

5.2.4 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.

The Big Picture

First Steps

Before continuing, be sure you've got Falcon *installed*. Then, create a new project folder called "look" and cd into it:

```
$ mkdir look
$ cd look
```

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

```
$ touch app.py
```

Open that file in your favorite text editor and add the following lines:

```
import falcon

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

That 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 expects it to be called, by default.

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. Let's take a look at the falcon.API class.

First, install IPython (if you don't already have it), 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 built-in `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. The team has worked hard to optimize the docstrings for readability, so that you can quickly scan them and find what you need.

Tip: bpython is another super- powered REPL that is good to have in your toolbox when exploring a new library.

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 that you would actually deploy in production.

```
$ pip install gunicorn
$ gunicorn app
```

Now try querying it with curl:

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

You should get a 404. That's actually OK, because we haven't specified any routes yet. Note that Falcon includes a default 404 response handler that will fire for any requested path that doesn't match any routes.

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
```

Creating Resources

Falcon borrows some of its terminology from the REST architectural style, so if you are familiar with that mindset, Falcon should be familiar. On the other hand, if you have no idea what REST is, no worries; Falcon was designed to be as intuitive as possible for anyone who understands the basics of HTTP.

In Falcon, you map incoming requests to things called "Resources". A Resource is just a regular Python class that includes some methods that follow a certain naming convention. Each of these methods corresponds to an action that the API client can request be performed in order to fetch or transform the resource in question.

Since we are building an image-sharing API, let's create an "images" resource. Create a new file, `images.py` within your project directory, and add the following to it:

```
import falcon


class Resource(object):

    def on_get(self, req, resp):
        resp.body = '{"message": "Hello world!"}'
        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_x` class method to the resource, where `x` is any one of the standard HTTP methods, lowercased (e.g., `on_get`, `on_put`, `on_head`, etc.).

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. See serialization with MessagePack for example:

```
def on_get(self, req, resp):
    resp.data = msgpack.packb({'message': 'Hello world!'})
    resp.content_type = 'application/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 get a little performance boost by assigning directly to `resp.data`.

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

```
import falcon

import images


api = application = falcon.API()

images = 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.

Restart gunicorn, and then try sending a GET request to the resource:

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

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 help function mentioned earlier to list the Request class members:

```
In [1]: import falcon

In [2]: help(falcon.Request)
```

Each responder also receives a response object that can be used for setting the status code, headers, and body of the response. You can list the Response class members using the same technique used above:

```
In [3]: help(falcon.Response)
```

Let's see how this works. When a client POSTs to our images collection, we want to create a new image resource. First, we'll need to specify where the images will be saved (for a real service, you would want to use an object storage service instead, such as Cloud Files or S3).

Edit your `images.py` file and add the following to the resource:

```
def __init__(self, storage_path):
    self.storage_path = storage_path
```

Then, edit `app.py` and pass in a path to the resource initializer.

Next, let's implement the POST responder:

```
import os
import time
import uuid

import falcon


def _media_type_to_ext(media_type):
    # Strip off the 'image/' prefix
    return media_type[6:]


def _generate_id():
    return str(uuid.uuid4())


class Resource(object):

    def __init__(self, storage_path):
        self.storage_path = storage_path

    def on_post(self, req, resp):
        image_id = _generate_id()
        ext = _media_type_to_ext(req.content_type)
        filename = image_id + '.' + ext

        image_path = os.path.join(self.storage_path, filename)

        with open(image_path, 'wb') as image_file:
            while True:
                chunk = req.stream.read(4096)
                if not chunk:
                    break

                image_file.write(chunk)

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

As you can see, we generate a unique ID and filename for the new 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; Falcon never spools or decodes request data, instead giving you direct access to the incoming binary stream provided by the WSGI server.

Note that we are setting the HTTP response status code to "201 Created". 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.) Note that the Request and Response classes contain convenience 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.

Restart gunicorn, and then try sending a POST request to the resource (substituting test.jpg for a path to any JPEG you like.)

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

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

Serving Images

Now that we have a way of getting images into the service, we 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, like so:

```
$ http GET localhost:8000/images/87db45ff42
```

Now, we could add an `on_get` responder to our images resource, and that is fine for simple resources like this, but that approach can lead to problems when you need to respond differently to the same HTTP method (e.g., GET), depending on whether the user wants to interact with a collection of things, or a single thing.

With that in mind, let's create a separate class to represent a single image, as opposed to a collection of images. We will then add an `on_get` responder to the new class.

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

```
import os
import time
import uuid

import falcon


def _media_type_to_ext(media_type):
    # Strip off the 'image/' prefix
    return media_type[6:]


def _ext_to_media_type(ext):
    return 'image/' + ext


def _generate_id():
    return str(uuid.uuid4())


class Collection(object):

    def __init__(self, storage_path):
        self.storage_path = storage_path

    def on_post(self, req, resp):
        image_id = _generate_id()
        ext = _media_type_to_ext(req.content_type)
        filename = image_id + '.' + ext

        image_path = os.path.join(self.storage_path, filename)

        with open(image_path, 'wb') as image_file:
            while True:
                chunk = req.stream.read(4096)
                if not chunk:
                    break
```

```
                image_file.write(chunk)

        resp.status = falcon.HTTP_201
        resp.location = '/images/' + filename


class Item(object):

    def __init__(self, storage_path):
        self.storage_path = storage_path

    def on_get(self, req, resp, name):
        ext = os.path.splitext(name)[1][1:]
        resp.content_type = _ext_to_media_type(ext)

        image_path = os.path.join(self.storage_path, name)
        resp.stream = open(image_path, 'rb')
        resp.stream_len = os.path.getsize(image_path)
```

As you can see, we renamed `Resource` to `Collection` and added a new `Item` class to represent a single image resource. 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.stream_len`. Whenever using `resp.stream` instead of `resp.body` or `resp.data`, you have to also specify the expected length of the stream 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 the `on_get` responder to do.

Now, let's wire things up and give this a try. Go ahead and edit `app.py` to look something like this:

```
import falcon

import images


api = application = falcon.API()

storage_path = '/usr/local/var/look'

image_collection = images.Collection(storage_path)
image = images.Item(storage_path)

api.add_route('/images', image_collection)
api.add_route('/images/{name}', image)
```

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 currently supports Level 1 URI templates, and support for higher levels is planned.

Now, restart gunicorn and post another picture to the service:

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

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

```
$ http localhost:8000/images/6daa465b7b.jpeg
```

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

Query Strings

Coming soon...

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 of all, let's check the incoming media type when something is posted to make sure it is a common image type. We'll do this by using a Falcon `before` hook.

First, let's define a list of media types our 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, 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 like so:

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

Now, before every call to that responder, Falcon will first invoke the `validate_image_type` method. There isn't anything special about that method, other than it must accept three arguments. Every hook takes, as its first two arguments, a reference to the same `req` and `resp` objects that are passed into responders. The third 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 inject extra kwargs for responders in a DRY way, e.g.,:

```
def extract_project_id(req, resp, 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 can imagine that such a hook should apply to all responders for a resource, or even globally to all resources. You can apply hooks to an entire resource like so:

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

    # ...
```

And you can apply hooks globally by passing them into the API class initializer:

```
falcon.API(before=[extract_project_id])
```

To learn more about hooks, take a look at the docstring for the `API` class, as well the docstrings for the `falcon.before` and `falcon.after` decorators.

Now that you've added a hook to validate the media type when an image is POSTed, you can see it in action by passing in something nefarious:

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

That should return a `400 Bad Request` status and a nicely structured error body. 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 API.

Tip: Please take a look at our new sister project, Talons, for a collection of useful Falcon hooks contributed by the community. Also, If you create a nifty hook that you think others could use, please consider contributing to the project yourself.

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. Your code is well-tested, with high code coverage.
3. Errors are anticipated, detected, and handled appropriately within each responder.

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

Speaking of error handling, when something goes horribly (or mildly) wrong, you *could* manually set the error status, appropriate response headers, and even an error body using the `resp` object. However, Falcon tries to make things a bit easier by providing a set of exceptions you can raise when something goes wrong. In fact, if Falcon catches any exception your responder throws that inherits from `falcon.HTTPError`, the framework will convert that exception to an appropriate HTTP error response.

You may raise an instance of `falcon.HTTPError`, or use any one of a number of predefined error classes that try to do "the right thing" in setting appropriate headers and bodies. Have a look at the docs for any of the following to get more information on how you can use them in your API:

```
falcon.HTTPBadGateway
falcon.HTTPBadRequest
falcon.HTTPConflict
falcon.HTTPError
falcon.HTTPForbidden
falcon.HTTPInternalServerError
falcon.HTTPLengthRequired
falcon.HTTPMethodNotAllowed
falcon.HTTPNotAcceptable
falcon.HTTPNotFound
falcon.HTTPPreconditionFailed
falcon.HTTPRangeNotSatisfiable
falcon.HTTPServiceUnavailable
falcon.HTTPUnauthorized
falcon.HTTPUnsupportedMediaType
falcon.HTTPUpgradeRequired
```

For example, you could handle a missing image file like this:

```
try:
    resp.stream = open(image_path, 'rb')
except IOError:
    raise falcon.HTTPNotFound()
```

Or you could handle a bogus filename like this:

```
VALID_IMAGE_NAME = re.compile(r'[a-f0-9]{10}\.(jpeg|gif|png)$')

# ...

class Item(object):

    def __init__(self, storage_path):
        self.storage_path = storage_path

    def on_get(self, req, resp, name):
        if not VALID_IMAGE_NAME.match(name):
            raise falcon.HTTPNotFound()
```

Sometimes you don't have much control over the type of exceptions that get raised. To address this, Falcon lets you create custom handlers for any type of error. For example, if your database throws exceptions that inherit from NiftyDBError, you can install a special error handler just for NiftyDBError, so you don't have to copy-paste your handler code across multiple responders.

Have a look at the docstring for `falcon.API.add_error_handler` for more information on using this feature

to DRY up your code:

```
In [71]: help(falcon.API.add_error_handler)
```

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."

5.3 Classes and Functions

5.3.1 API Class

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

```
import falcon

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

class falcon.**API**(*media_type='application/json; charset=utf-8'*, *before=None*, *after=None*, *request_type=<class 'falcon.request.Request'>*, *response_type=<class 'falcon.response.Response'>*, *middleware=None*)

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

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

> **Warning:** Global hooks (configured using the *before* and *after* kwargs) are deprecated in favor of middleware, and may be removed in a future version of the framework.

Parameters

- **media_type** (*str, optional*) – Default media type to use as the value for the Content-Type header on responses (default 'application/json').
- **middleware** (*object or list, optional*) – One or more 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.

        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):
```

```
                """Process the request after routing.

                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.
                """

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

See also *Middleware*.

- **request_type** (*Request, optional*) – `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, optional*) – `Response`-like class to use instead of Falcon's default class. (default `falcon.response.Response`)

req_options
RequestOptions – A set of behavioral options related to incoming requests.

add_error_handler(*exception*, *handler=None*)
Registers a handler for a given exception error type.

Parameters

- **exception** (*type*) – Whenever an error occurs when handling a request that is an instance of this exception class, the associated handler will be called.
- **handler** (*callable*) – A function or callable object taking the form `func(ex, req, resp, 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(ex, req, resp, params):
        # TODO: Log the error
        # Convert to an instance of falcon.HTTPError
        raise falcon.HTTPError(falcon.HTTP_792)
```

Note: A handler can either raise an instance of `HTTPError` or modify *resp* manually in order to communicate information about the issue to the client.

add_route(*uri_template*, *resource*)

Associates a templatized URI path with a resource.

A resource is an instance of a class that defines various on_* "responder" methods, one for each HTTP method the resource allows. For example, to support GET, simply define an *on_get* responder. If a client requests an unsupported method, Falcon will respond with "405 Method not allowed".

Responders must always define at least two arguments to receive request and response objects, respectively. For example:

```
def on_post(self, req, resp):
    pass
```

In addition, 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
```

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.

add_sink(*sink*, *prefix='/'*)

Registers 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*).

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.

The `falcon.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, exception):
    representation = None

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

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

    return (preferred, representation)
```

> **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. If this is not desirable, a custom error serializer may be used to override this behavior.

Parameters **serializer** (*callable*) – A function taking the form `func(req, exception)`, where *req* is the request object that was passed to the responder method, and *exception* is an instance of `falcon.HTTPError`. The function must return a `tuple` of the form (*media_type*, *representation*), or (`None`, `None`) if the client does not support any of the available media types.

class `falcon.`**RequestOptions**
This class is a container for `Request` options.

keep_blank_qs_values
bool – Set to `True` in order to retain blank values in query string parameters (default `False`).

5.3.2 Req/Resp

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.
- **`options`** (*dict*) – Set of global options passed from the API handler.

`protocol`
str – Either 'http' or 'https'.

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

`host`
str – Hostname requested by the client

`subdomain`
str – 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.

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

`app`
str – Name of the WSGI app (if using WSGI's notion of virtual hosting).

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

`context`
dict – Dictionary to hold any data 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.

`context_type`
class – Class variable that determines the factory or type to use for initializing the *context* attribute. By default, the framework will instantiate standard `dict` objects. 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.

`uri`
str – The fully-qualified URI for the request.

`url`
str – alias for *uri*.

relative_uri
str – The path + query string portion of the full URI.

path
str – Path portion of the request URL (not including query string).

query_string
str – Query string portion of the request URL, without the preceding '?' character.

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

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

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

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

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

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

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

stream
File-like object for reading the body of the request, if any.

Note: If an HTML form is POSTed to the API using the *application/x-www-form-urlencoded* media type, Falcon 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.

Note also that 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).

---

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

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

range
tuple of int – 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.)

if_match
str – Value of the If-Match header, or `None` if the header is missing.

if_none_match
str – Value of the If-None-Match header, or `None` if the header is missing.

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

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

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

headers
dict – 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. This parsing can be costly, so unless you need all the headers in this format, you should use the *get_header* method or one of the convenience attributes instead, to get a value for a specific header.

params
dict – 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.

options
dict – Set of global options passed from the API handler.

client_accepts(*media_type*)
Determines 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*)
Returns 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

get_header(*name*, *required=False*)
Return a raw header value as a string.

Parameters

- **name** (*str*) – Header name, case-insensitive (e.g., 'Content-Type')
- **required** (*bool, optional*) – Set to `True` to raise `HTTPBadRequest` instead of returning gracefully when the header is not found (default `False`).

Returns

The value of the specified header if it exists, or `None` if the header is not found and is not required.

Return type str

Raises `HTTPBadRequest` – The header was not found in the request, but it was required.

`get_param`(*name*, *required=False*, *store=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, the parameters from the request body will be merged into the query string parameters.

If a key appears more than once in the form data, one of the values will be returned as a string, but it is undefined which one. Use *req.get_param_as_list()* to retrieve all the values.

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 *req.get_param_as_list()* to retrieve all the values.

Parameters

- **`name`** (*`str`*) – Parameter name, case-sensitive (e.g., 'sort').
- **`required`** (*`bool, optional`*) – Set to `True` to raise `HTTPBadRequest` instead of returning `None` when the parameter is not found (default `False`).
- **`store`** (*`dict, optional`*) – A `dict`-like object in which to place the value of the param, but only if the param is present.

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=False*)

Return the value of a query string parameter as a boolean

The following boolean strings are supported:

```
TRUE_STRINGS = ('true', 'True', 'yes')
FALSE_STRINGS = ('false', 'False', 'no')
```

Parameters

- **`name`** (*`str`*) – Parameter name, case-sensitive (e.g., 'detailed').
- **`required`** (*`bool, optional`*) – 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, optional`*) – 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*) – If `True`, an empty string value will be treated as `True`. Normally empty strings are 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` through the *req_options* attribute.

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.

get_param_as_int(*name*, *required=False*, *min=None*, *max=None*, *store=None*)

Return the value of a query string parameter as an int.

Parameters

- **name** (*str*) – Parameter name, case-sensitive (e.g., 'limit').
- **required** (*bool, optional*) – Set to `True` to raise `HTTPBadRequest` instead of returning `None` when the parameter is not found or is not an integer (default `False`).
- **min** (*int, optional*) – Set to the minimum value allowed for this param. If the param is found and it is less than min, an `HTTPError` is raised.
- **max** (*int, optional*) – Set to the maximum value allowed for this param. If the param is found and its value is greater than max, an `HTTPError` is raised.
- **store** (*dict, optional*) – A `dict`-like object in which to place the value of the param, but only if the param is found (default `None`).

Returns

The value of the param if it is found and can be converted to an integer. 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. Also raised if the param's value falls outside the given interval, i.e., the value must be in the interval: min <= value <= max to avoid triggering an error.

get_param_as_list(*name*, *transform=None*, *required=False*, *store=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').
- **transform** (*callable, optional*) – 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, optional*) – Set to `True` to raise `HTTPBadRequest` instead of returning `None` when the parameter is not found (default `False`).

- **store** (*dict, optional*) – A `dict`-like object in which to place the value of the param, but only if the param is found (default `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 a query string containing this:

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

or a query string containing this:

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

would both result in:

```
['1', '3']
```

Return type list

Raises

- `HTTPBadRequest` – A required param is missing from the request.
- `HTTPInvalidParam` – A tranform function raised an instance of `ValueError`.

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.

Response

class `falcon.`**Response**

Represents an HTTP response to a client request.

Note: *Response* is not meant to be instantiated directly by responders.

---

status

str – 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`.

---

body

str or unicode – String representing response content. If Unicode, Falcon will encode as UTF-8 in the response. If data is already a byte string, use the data attribute instead (it's faster).

body_encoded
bytes – Returns a UTF-8 encoded version of *body*.

data
bytes – 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.

---

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.

stream_len
int – Expected length of *stream* (e.g., file size).

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: https://tools.ietf.org/html/rfc5988

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://goo.gl/618GHr for a list of registered link relation types.

Kwargs:

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.

Warning: If the header already exists, the new value will be appended to it, delimited by a comma. Most header specifications support this format, Cookie and Set-Cookie being the notable exceptions.

Parameters

- **name** (*str*) – Header name to set (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.
- **value** (*str*) – Value for the header. Must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

cache_control
Sets 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_location
Sets the Content-Location header.

content_range
A tuple to use in constructing a value for the Content-Range header.

The tuple has the form (*start*, *end*, *length*), where *start* and *end* designate the byte range (inclusive), and *length* is the total number of bytes, or '*' if unknown. You may pass `int`'s for these numbers (no need to convert to `str` beforehand).

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: http://goo.gl/Iglhp

content_type
Sets the Content-Type header.

etag
Sets the ETag header.

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

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

location
Sets the Location header.

retry_after
Sets 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_header(*name*, *value*)
Set a header for this response to a given value.

> **Warning:** Calling this method overwrites the existing value, if any.

Parameters

- **name** (*str*) – Header name to set (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.
- **value** (*str*) – Value for the header. Must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

set_headers(*headers*)
Set several headers at once.

> **Warning:** Calling this method overwrites existing values, if any.

Parameters **headers** (*dict or list*) – A dictionary of header names and values to set, or `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:** 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*, *stream_len*)
Convenience method for setting both *stream* and *stream_len*.

Although the *stream* and *stream_len* properties may be set directly, using this method ensures *stream_len* is not accidentally neglected.

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`.

"Tells downstream proxies how to match future request headers to decide whether the cached response can be used rather than requesting a fresh one from the origin server."

(Wikipedia)

See also: http://goo.gl/NGHdL

5.3.3 Status Codes

Falcon provides a list of constants for common HTTP response status codes that you can use like so:

```
# Override the default "200 OK" response status
resp.status = falcon.HTTP_409
```

1xx Informational

```
HTTP_100 = '100 Continue'
HTTP_101 = '101 Switching Protocols'
```

2xx Success

```
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_226 = '226 IM Used'
```

3xx Redirection

```
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'
```

4xx Client Error

```
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_426 = '426 Upgrade Required'
```

5xx Server Error

```
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'
```

5.3.4 Error Handling

When a request results in an error condition, you *could* manually set the error status, appropriate response headers, and even an error body using the `resp` object. However, Falcon tries to make things a bit easier and more consistent by providing a set of error classes you can raise from within your app. Falcon catches any exception that inherits from `falcon.HTTPError`, and automatically converts it to an appropriate HTTP response.

You may raise an instance of `falcon.HTTPError` directly, or use any one of a number of predefined error classes that try to be idiomatic in setting appropriate headers and bodies.

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 this or a child class to have Falcon automagically return pretty error responses (with an appropriate HTTP status code) to the client when something goes wrong.

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

`has_representation`
bool – 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 the `falcon.http_error.NoRepresentation` mixin.

`title`
str – Error title to send to the client. Will be `None` if the error should result in an HTTP response with an empty body.

`description`
str – Description of the error to send to the client.

`headers`
dict – Extra headers to add to the response.

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

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

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

Keyword Arguments

- **title** (*str*) – Human-friendly error title (default `None`).
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two (default `None`).
- **headers** (*dict*) – 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`.

- **headers** – Extra headers to return in the response to the client (default `None`).
- **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 (defaults to "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=<type 'dict'>*)
Returns 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.

to_json()
Returns a pretty-printed JSON representation of the error.

Returns A JSON document for the error.

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

Returns An XML document for the error.

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.`**`HTTPInvalidHeader`**(*msg*, *header_name*, ***kwargs*)
HTTP header is invalid. Inherits from `HTTPBadRequest`.

Parameters

- **`msg`** (*`str`*) – A description of why the value is invalid.
- **`header_name`** (*`str`*) – The name of the header.
- **`kwargs`** (*`optional`*) – Same as for `HTTPError`.

exception `falcon.`**`HTTPMissingHeader`**(*header_name*, ***kwargs*)
HTTP header is missing. Inherits from `HTTPBadRequest`.

Parameters

- **`header_name`** (*`str`*) – The name of the header.
- **`kwargs`** (*`optional`*) – Same as for `HTTPError`.

exception `falcon.`**`HTTPInvalidParam`**(*msg*, *param_name*, ***kwargs*)
HTTP parameter is invalid. Inherits from `HTTPBadRequest`.

Parameters

- **`msg`** (*`str`*) – A description of the invalid parameter.
- **`param_name`** (*`str`*) – The name of the paramameter.
- **`kwargs`** (*`optional`*) – Same as for `HTTPError`.

exception `falcon.`**`HTTPMissingParam`**(*param_name*, ***kwargs*)
HTTP parameter is missing. Inherits from `HTTPBadRequest`.

Parameters

- **`param_name`** (*`str`*) – The name of the paramameter.
- **`kwargs`** (*`optional`*) – Same as for `HTTPError`.

exception `falcon.`**`HTTPBadRequest`**(*title*, *description*, ***kwargs*)
400 Bad Request.

The request could not be understood by the server due to malformed syntax. The client SHOULD NOT repeat the request without modifications. (RFC 2616)

Parameters

- **`title`** (*`str`*) – Error title (e.g., 'TTL Out of Range').

- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **kwargs** (*optional*) – Same as for `HTTPError`.

exception `falcon.`**`HTTPUnauthorized`**(*title*, *description*, ***kwargs*)
401 Unauthorized.

Use when authentication is required, and the provided credentials are not valid, or no credentials were provided in the first place.

Parameters

- **title** (*str*) – Error title (e.g., 'Authentication Required').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **scheme** (*str*) – Authentication scheme to use as the value of the WWW-Authenticate header in the response (default `None`).
- **kwargs** (*optional*) – Same as for `HTTPError`.

exception `falcon.`**`HTTPForbidden`**(*title*, *description*, ***kwargs*)
403 Forbidden.

Use when the client's credentials are good, but they do not have permission to access the requested resource.

If the request method was not HEAD and the server wishes to make public why the request has not been fulfilled, it SHOULD describe the reason for the refusal in the entity. If the server does not wish to make this information available to the client, the status code 404 (Not Found) can be used instead. (RFC 2616)

Parameters

- **title** (*str*) – Error title (e.g., 'Permission Denied').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **kwargs** (*optional*) – Same as for `HTTPError`.

exception `falcon.`**`HTTPNotFound`**(***kwargs*)
404 Not Found.

Use this when the URL path does not map to an existing resource, or you do not wish to disclose exactly why a request was refused.

exception `falcon.`**`HTTPMethodNotAllowed`**(*allowed_methods*, ***kwargs*)
405 Method Not Allowed.

The method specified in the Request-Line is not allowed for the resource identified by the Request-URI. The response MUST include an Allow header containing a list of valid methods for the requested resource. (RFC 2616)

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

exception `falcon.`**`HTTPNotAcceptable`**(*description*, ***kwargs*)
406 Not Acceptable.

The client requested a resource in a representation that is not supported by the server. The client must indicate a supported media type in the Accept header.

The resource identified by the request is only capable of generating response entities which have content characteristics not acceptable according to the accept headers sent in the request. (RFC 2616)

Parameters

- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **kwargs** (*optional*) – Same as for `HTTPError`.

exception `falcon.`**`HTTPConflict`**(*title*, *description*, ***kwargs*)

409 Conflict.

The request could not be completed due to a conflict with the current state of the resource. This code is only allowed in situations where it is expected that the user might be able to resolve the conflict and resubmit the request. The response body SHOULD include enough information for the user to recognize the source of the conflict. Ideally, the response entity would include enough information for the user or user agent to fix the problem; however, that might not be possible and is not required.

Conflicts are most likely to occur in response to a PUT request. For example, if versioning were being used and the entity being PUT included changes to a resource which conflict with those made by an earlier (third-party) request, the server might use the 409 response to indicate that it can't complete the request. In this case, the response entity would likely contain a list of the differences between the two versions in a format defined by the response Content-Type.

(RFC 2616)

Parameters

- **title** (*str*) – Error title (e.g., 'Editing Conflict').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **kwargs** (*optional*) – Same as for `HTTPError`.

exception `falcon.`**`HTTPLengthRequired`**(*title*, *description*, ***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. (RFC 2616)

Parameters

- **title** (*str*) – Error title (e.g., 'Missing Content-Length').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **kwargs** (*optional*) – Same as for `HTTPError`.

exception `falcon.`**`HTTPPreconditionFailed`**(*title*, *description*, ***kwargs*)

412 Precondition Failed.

The precondition given in one or more of the request-header fields evaluated to false when it was tested on the server. This response code allows the client to place preconditions on the current resource metainformation (header field data) and thus prevent the requested method from being applied to a resource other than the one intended. (RFC 2616)

Parameters

- **title** (*str*) – Error title (e.g., 'Image Not Modified').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **kwargs** (*optional*) – Same as for `HTTPError`.

exception falcon.**HTTPUnsupportedMediaType**(*description*, ***kwargs*)
415 Unsupported Media Type.

The client is trying to submit a resource encoded as an Internet media type that the server does not support.

Parameters

- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **kwargs** (*optional*) – Same as for HTTPError.

exception falcon.**HTTPRangeNotSatisfiable**(*resource_length*)
416 Range Not Satisfiable.

The requested range is not valid. See also: http://goo.gl/Qsa4EF

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

exception falcon.**HTTPInternalServerError**(*title*, *description*, ***kwargs*)
500 Internal Server Error.

Parameters

- **title** (*str*) – Error title (e.g., 'This Should Never Happen').
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **kwargs** (*optional*) – Same as for HTTPError.

exception falcon.**HTTPBadGateway**(*title*, *description*, ***kwargs*)
502 Bad Gateway.

Parameters

- **title** (*str*) – Error title, for example: 'Upstream Server is Unavailable'.
- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
- **kwargs** (*optional*) – Same as for HTTPError.

exception falcon.**HTTPServiceUnavailable**(*title*, *description*, *retry_after*, ***kwargs*)
503 Service Unavailable.

Parameters

- **title** (*str*) – Error title (e.g., 'Temporarily 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. See also: http://goo.gl/DIrWr .
- **kwargs** (*optional*) – Same as for HTTPError.

5.3.5 Middleware Components

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.

        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):
        """Process the request after routing.

        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.
        """

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

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.

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 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
```

If one of the *process_request* middleware methods raises an error, it will be processed according to the error 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, in which case the framework will use the latter exception to update the *resp* object. 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, mob3, and routing>
    mob2.process_response
mob1.process_response
```

Finally, if one of the *process_response* methods raises an error, or the routed on_* responder method itself raises an error, the exception will be handled in a similar manner as above. Then, the framework will execute any remaining middleware on the stack.

5.3.6 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):
```

```
        pass

    def on_get(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.

falcon.**before**(*action*)

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
```

---

falcon.**after**(*action*)

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

5.3.7 Routing

The *falcon.routing* module contains utilities used internally by `falcon.API` to route requests. They are exposed here for use by classes that inherit from `falcon.API` to implement custom routing logic, and in anticipation of a future version of the framework that will afford customization of routing via composition in lieu of inheritance.

falcon.routing.**compile_uri_template**(*template*)

Compile the given URI template string into a pattern matcher.

This function currently only recognizes Level 1 URI templates, and only for the path portion of the URI.

See also: http://tools.ietf.org/html/rfc6570

Parameters **template** – A Level 1 URI template. Method responders must accept, as arguments, all fields specified in the template (default '/'). Note that field names are restricted to ASCII a-z, A-Z, and the underscore '_'.

Returns (template_field_names, template_regex)

Return type tuple

falcon.routing.**create_http_method_map**(*resource*, *uri_fields*, *before*, *after*)

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)`.
- **uri_fields** – A set of field names from the route's URI template that a responder must support in order to avoid "method not allowed".
- **before** – An action hook or `list` of hooks to be called before each *on_** responder defined by the resource.
- **after** – An action hook or `list` of hooks to be called after each *on_** responder defined by the resource.

Returns A mapping of HTTP methods to responders.

Return type dict

5.3.8 Utilities

URI Functions

`falcon.util.uri.`**decode**(*encoded_uri*)

Decodes percent-encoded characters in a URI or query string.

This function models the behavior of *urllib.parse.unquote_plus*, but is faster. It is also more robust, in that it will decode escaped UTF-8 mutibyte sequences.

Parameters **encoded_uri** (*str*) – An encoded URI (full or partial).

Returns

A decoded URL. Will be of type unicode on Python 2 IFF the URL contained escaped non-ASCII characters, in which case UTF-8 is assumed per RFC 3986.

Return type str

`falcon.util.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., `six.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.util.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., `six.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.util.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.
- **`default_port`** (*`int, optional`*) – 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.util.uri.`**`parse_query_string`**(*query_string*, *keep_blank_qs_values=False*)

Parse a query string into a dict.

Query string parameters are assumed to use standard form-encoding. Only parameters with values are parsed. 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').

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_qs_values`** (*`bool`*) – If set to `True`, preserves boolean fields and fields with no content as blank strings.

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`.

Testing

class `falcon.testing.`**`TestBase`**(*methodName='runTest'*)
Extends `testtools.TestCase` to support WSGI integration testing.

`TestBase` provides a base class that provides some extra plumbing to help simulate WSGI calls without having to actually host your API in a server.

Note: If `testtools` is not available, `unittest` is used instead.

`api`
falcon.API – An API instance to target when simulating requests. Defaults to `falcon.API()`.

`srmock`
falcon.testing.StartResponseMock – Provides a callable that simulates the behavior of the *start_response* argument that the server would normally pass into the WSGI app. The mock object captures various information from the app's response to the simulated request.

`test_route`
str – A simple, generated path that a test can use to add a route to the API.

`setUp`()
Initializer, unittest-style

`simulate_request`(*path*, *decode=None*, ***kwargs*)
Simulates a request to *self.api*.

Parameters

- **`path`** (*`str`*) – The path to request.
- **`decode`** (*`str, optional`*) – If this is set to a character encoding, such as 'utf-8', *simulate_request* will assume the response is a single byte string, and will decode it as the result of the request, rather than simply returning the standard WSGI iterable.
- **`kwargs`** (*`optional`*) – Same as those defined for *falcon.testing.create_environ*.

`tearDown`()
Destructor, unittest-style

class `falcon.testing.`**`TestResource`**
Mock resource for integration testing.

This class implements the *on_get* responder, captures request data, and sets response body and headers.

Child classes may add additional methods and attributes as needed.

`sample_status`
str – HTTP status to set in the response

`sample_body`
str – Random body string to set in the response

resp_headers
dict – Sample headers to use in the response

req
falcon.Request – Request object passed into the *on_get* responder.

resp
falcon.Response – Response object passed into the *on_get* responder.

kwargs
dict – Keyword arguments passed into the *on_get* responder, if any.

called
bool – `True` if *on_get* was ever called; `False` otherwise.

on_get(*req*, *resp*, ***kwargs*)
GET responder.

Captures *req*, *resp*, and *kwargs*. Also sets up a sample response.

Parameters

- **req** – Falcon `Request` instance.
- **resp** – Falcon `Response` instance.
- **kwargs** – URI template *name=value* pairs, if any, along with any extra args injected by middleware.

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

call_count
int – Number of times *start_response* was called.

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

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

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

falcon.testing.**httpnow**()
Returns the current UTC time as an RFC 1123 date.

Returns An HTTP date string, e.g., "Tue, 15 Nov 1994 12:45:26 GMT".

Return type str

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*)
Creates a mock PEP-3333 environ `dict` for simulating WSGI requests.

Parameters

- **path** (*str, optional*) – The path for the request (default '/')
- **query_string** (*str, optional*) – The query string to simulate, without a leading '?' (default '')
- **protocol** (*str, optional*) – 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 or int, optional*) – 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 or list, optional*) – Headers as a `dict` or an iterable collection of (*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 or unicode*) – The body of the request (default '')
- **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.

Miscellaneous

`falcon.util.`**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.util.`**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.util.`**http_date_to_dt**(*http_date*)

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".

Returns

A UTC datetime instance corresponding to the given HTTP date.

Return type datetime

falcon.util.**to_query_str**(*params*)
Converts a dictionary of params 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`. If *params* is a `list`, it will be converted to a comma-delimited string of values (e.g., 'thing=1,2,3')

Returns

A URI query string including the '?' prefix, or an empty string if no params are given (the `dict` is empty).

Return type str

5.4 Changelogs

5.4.1 Changelog for Falcon 0.2.0

New

- 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

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'].

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.

f

I

K

L

M

N

O

P

Q

R

S

T

U

V