Falcon Documentation

Release 1.1.0

Kurt Griffiths et al.

Oct 27, 2016

Contents

Release v1.1 (*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.

```python
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())
```

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

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

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

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

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
- Supports Python 2.6, 2.7, 3.3, 3.4 and 3.5
- Compatible with PyPy and Jython

Who's Using Falcon?

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

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

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

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

Useful Links

- Falcon Home
- Falcon @ PyPI
- Falcon @ GitHub

Resources

- An Unladen Web Framework
- The Definitive Introduction to Falcon

Documentation

6.1 Community Guide

6.1.1 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

Chat with fellow community members in the official Falcon IRC channel. It's a great place to ask questions and share your ideas. You can find us in #falconframework on Freenode.

Discussion Group

The Falcon community maintains a discussion group that you can use to share your ideas and ask questions about the framework. To join the discussion, please visit https://groups.google.com/d/forum/falconframework.

Per our Code of Conduct, we expect everyone who participates in community discussions to act professionally, and lead by example in encouraging constructive discussions. Each individual in the community is responsible for creating a positive, constructive, and productive culture.

Submit Issues

If you have an idea for a feature, run into something that is harder to use than it should be, or find a bug, please let the crew know in **#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 on the mailing list, and to help peer-review pull requests. If you use the Chrome browser, we recommend installing the NotHub extension to stay up to date with PRs. If you would like to contribute a new feature or fix a bug in the framework, please check out our Contributor's Guide for more information.

We'd love to have your help!

Code of Conduct

All contributors and maintainers of this project are subject to our Code of Conduct.

6.1.2 Contribute to Falcon

Thanks for your interest in the project! We welcome pull requests from developers of all skill levels. To get started, simply fork the master branch on GitHub to your personal account and then clone the fork into your development environment.

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

Please don't hesitate to reach out if you have any questions, or just need a little help getting started. You can find us in #falconframework on Freenode. It's the best way to discuss ideas, ask questions, and generally stay in touch with fellow contributors.

Please check out our Contributor's Guide for more information.

Thanks!

6.1.3 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:

```python
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 come with batteries included?

The Python ecosystem offers a bunch of great libraries that you are welcome to use from within your responders, hooks, and middleware components. 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.

How do I authenticate requests?

Hooks and middleware components can be used together to authenticate and authorize requests. For example, a middleware component could be used to parse incoming credentials and place the results in `req.context`. Downstream components or hooks could then use this information to authorize the request, taking into account the user's role and the requested resource.

Why 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 serve 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.

My app is setting a cookie, but it isn't being passed back in subsequent requests.

By default, Falcon enables the *secure* cookie attribute. Therefore, if you are testing your app over HTTP (instead of HTTPS), the client will not send the cookie in subsequent requests. See also *the cookie documentation*

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
```

6.2 User Guide

6.2.1 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

Copyright 2012-2016 by Rackspace Hosting, Inc. and other contributors, as noted in the individual source code files.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

By contributing to this project, you agree to also license your source code under the terms of the Apache License, Version 2.0, as described above.

6.2.2 Installation

PyPy

PyPy is the fastest way to run your Falcon app. However, note that only the PyPy 2.7 compatible release is currently supported.

```
$ pip install falcon
```

CPython

Falcon also fully supports CPython 2.6-3.5.

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

```
$ pip install falcon
```

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

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

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

Installing on OS X

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

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

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

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

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

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

Dependencies

Falcon depends on *six* and *python-mimeparse*. *python-mimeparse* is a better-maintained fork of the similarly named *mimeparse* project. Normally the correct package will be selected by Falcon's `setup.py`. However, if you are using an alternate strategy to manage dependencies, please take care to install the correct package in order to avoid errors.

WSGI Server

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

```
$ pip install [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 repo or downloaded a tarball from GitHub, you can install Falcon like this:

```
$ cd falcon
$ pip install .
```

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

```
$ cd falcon
$ pip install -e .
```

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

```
$ cd falcon
$ pip install -r tools/test-requires
$ pytest tests
```

Or, to run the default set of tests:

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

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

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

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

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

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

Or on Linux:

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

6.2.3 Quickstart

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

Learning by Example

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

```python
# things.py

# Let's get this party started!
import falcon


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

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

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

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

You can run the above example using any WSGI server, such as uWSGI or Gunicorn. For example:

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

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.

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

import falcon
import requests


class StorageEngine(object):

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

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


class StorageError(Exception):

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

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


class SinkAdapter(object):

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

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

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


class AuthMiddleware(object):

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

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

        if token is None:
            description = ('Please provide an auth token '
```

```python
                            'as part of the request.')

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

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

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

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


class RequireJSON(object):

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

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


class JSONTranslator(object):

    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 '
```

```python
                                    '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(object):

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

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

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

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

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

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

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

    @falcon.before(max_body(64 * 1024))
```

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

6.2.4 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.

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:

```python
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. For example, you could use [MessagePack](), or any other serialization format.

If you'd like to use MessagePack in the above example, you'll need to install the (de)serializer for Python running `pip install msgpack-python` and then update your responder to set the response data and content_type accordingly:

```python
import falcon

import msgpack


class Resource(object):

    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:

```python
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:

```python
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:

```python
import os
import uuid
import mimetypes

import falcon


class Resource(object):
```

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

    def on_post(self, req, resp):
        ext = mimetypes.guess_extension(req.content_type)
        filename = '{uuid}{ext}'.format(uuid=uuid.uuid4(), ext=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
```

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:

```python
import os
import uuid
import mimetypes

import falcon


class Collection(object):

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

    def on_post(self, req, resp):
        ext = mimetypes.guess_extension(req.content_type)
        filename = '{uuid}{ext}'.format(uuid=uuid.uuid4(), ext=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):
        resp.content_type = mimetypes.guess_type(name)[0]
        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:

```python
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 also supports more complex parameterized path segments containing multiple values. For example, a GH-like API could use the following template to add a route for diffing two branches:

```
/repos/{org}/{repo}/compare/{usr0}:{branch0}...{usr1}:{branch1}
```

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.

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:

```python
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:

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

And then attach the hook to the `on_post` responder like so:

```python
@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 four arguments. Every hook takes, as its first two arguments, a reference to the same `req` and `resp` objects that are passed into responders. `resource` argument is a Resource instance associated with the request. The fourth argument, named `params` by convention, is a reference to the kwarg dictionary Falcon creates for each request. `params` will contain the route's URI template params and their values, if any.

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

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

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

Now, you can imagine that such a hook should apply to all responders for a resource. You can apply hooks to an entire resource like so:

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

    # ...
```

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

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.

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:

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

Or you could handle a bogus filename like this:

```python
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."

6.3 Classes and Functions

6.3.1 API Class

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

```python
import falcon

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

class falcon.**API**(*media_type='application/json; charset=UTF-8', request_type=<class 'fal-con.request.Request'>, response_type=<class 'falcon.response.Response'>, mid-dleware=None, router=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.

   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:

   ```python
   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, params):
            """Process the request and resource *after* routing.

            Note:
                This method is only called when the request matches
                a route to a resource.

            Args:
                req: Request object that will be passed to the
                    routed responder.
                resp: Response object that will be passed to the
                    responder.
                resource: Resource object to which the request was
                    routed. May be None if no route was found for
                    the request.
                params: A dict-like object representing any
                    additional params derived from the route's URI
                    template fields, that will be passed to the
                    resource's responder method as keyword
                    arguments.
            """

        def process_response(self, req, resp, resource, req_succeeded)
            """Post-processing of the response (after routing).

            Args:
                req: Request object.
                resp: Response object.
                resource: Resource object to which the request was
                    routed. May be None if no route was found
                    for the request.
                req_succeeded: True if no exceptions were raised
                    while the framework processed and routed the
                    request; otherwise False.
            """
```

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`)

- **router** (*object, optional*) – An instance of a custom router to use in lieu of the default engine. See also: *Routing*.

req_options

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

add_error_handler(*exception*, *handler=None*)

Registers a handler for a given exception error type.

A handler can raise an instance of `HTTPError` or `HTTPStatus` to communicate information about the issue to the client. Alternatively, a handler may modify *resp* directly.

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

> **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:
>
>   ```python
>   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)
>   ```

add_route(*uri_template*, *resource*, **args*, ***kwargs*)
Associates a templatized URI path with a resource.

Note: The following information describes the behavior of Falcon's default router.

A resource is an instance of a class that defines various "responder" methods, one for each HTTP method the resource allows. Responder names start with *on_* and are named according to which HTTP method they handle, as in *on_get*, *on_post*, *on_put*, etc.

If your resource does not support a particular HTTP method, simply omit the corresponding responder and Falcon will reply with "405 Method not allowed" if that method is ever requested.

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

```python
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.

Note: Since field names correspond to argument names in responder methods, they must be valid Python identifiers.

For example, given the following template:

```
/user/{name}
```

A PUT request to "/user/kgriffs" would be routed to:

```python
def on_put(self, req, resp, name):
    pass
```

Individual path segments may contain one or more field expressions:

```
/repos/{org}/{repo}/compare/{usr0}:{branch0}...{usr1}:{branch1}
```

> **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.

> **Note:** Any additional args and kwargs not defined above are passed through to the underlying router's add_route() method. The default router does not expect any additional arguments, but custom routers may take advantage of this feature to receive additional options when setting up routes.

---

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:

```python
def my_serializer(req, resp, 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)
        resp.body = representation
        resp.content_type = preferred
```

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, resp, exception), where *req* is the request object that was passed to the responder method, *resp* is the response object, and *exception* is an instance of falcon.HTTPError.

class `falcon.RequestOptions`
Defines a set of configurable request options.

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

keep_blank_qs_values
bool – Set to `True` to keep query string fields even if they do not have a value (default `False`). For comma-separated values, this option also determines whether or not empty elements in the parsed list are retained.

auto_parse_form_urlencoded
Set to `True` in order to automatically consume the request stream and merge the results into the request's query string params when the request's content type is *application/x-www-form-urlencoded* (default `False`). In this case, the request's content stream will be left at EOF.

Note: The character encoding for fields, before percent-encoding non-ASCII bytes, is assumed to be UTF-8. The special *_charset_* field is ignored if present.

Falcon expects form-encoded request bodies to be encoded according to the standard W3C algorithm (see also http://goo.gl/6rlcux).

---

auto_parse_qs_csv
Set to `False` to treat commas in a query string value as literal characters, rather than as a comma- separated list (default `True`). When this option is enabled, the value will be split on any non-percent-encoded

---

commas. Disable this option when encoding lists as multiple occurrences of the same parameter, and when values may be encoded in alternative formats in which the comma character is significant.

6.3.2 req/resp

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

```python
import falcon


class Resource(object):

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

Request

class `falcon.`**`Request`**(*env*, *options=None*)
    Represents a client's HTTP request.

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

        Parameters env(*dict*) – A WSGI environment dict passed in from the server. See also PEP-3333.

    Keyword Arguments options (dict): Set of global options passed from the API handler.

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.

---

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

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

access_route
    list – IP address of the original client, as well as any known addresses of proxies fronting the WSGI server.

    The following request headers are checked, in order of preference, to determine the addresses:

- `Forwarded`

- `X-Forwarded-For`

- `X-Real-IP`

If none of these headers are available, the value of *remote_addr* is used instead.

Note: Per RFC 7239, the access route may contain "unknown" and obfuscated identifiers, in addition to IPv4 and IPv6 addresses

---

> **Warning:** Headers can be forged by any client or proxy. Use this property with caution and validate all values before using them. Do not rely on the access route to authorize requests.

remote_addr
  str – IP address of the closest client or proxy to the WSGI server.

  This property is determined by the value of `REMOTE_ADDR` in the WSGI environment dict. Since this address is not derived from an HTTP header, clients and proxies can not forge it.

  Note: If your application is behind one or more reverse proxies, you can use *access_route* to retrieve the real IP address of the client.

  ---

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.

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

  ---

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).

  Note: *req.path* may be set to a new value by a *process_request()* middleware method in order to influence routing.

  ---

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

uri_template
   str – The template for the route that was matched for this request. May be `None` if the request has not yet been routed, as would be the case for *process_request()* middleware methods. May also be `None` if your app uses a custom routing engine and the engine does not provide the URI template when resolving a route.

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

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 input object for reading the body of the request, if any. Since this object is provided by the WSGI server itself, rather than by Falcon, it may behave differently depending on how you host your app. For example, attempting to read more bytes than are expected (as determined by the Content-Length header) may or may not block indefinitely. It's a good idea to test your WSGI server to find out how it behaves.

   This can be particulary problematic when a request body is expected, but none is given. In this case, the following call blocks under certain WSGI servers:

```
# Blocks if Content-Length is 0
data = req.stream.read()
```

   The workaround is fairly straightforward, if verbose:

```
# If Content-Length happens to be 0, or the header is
# missing altogether, this will not block.
data = req.stream.read(req.content_length or 0)
```

   Alternatively, when passing the stream directly to a consumer, it may be necessary to branch off the value of the Content-Length header:

```
if req.content_length:
    doc = json.load(req.stream)
```

   For a slight performance cost, you may instead wish to use *bounded_stream*, which wraps the native WSGI input object to normalize its behavior.

Note: If an HTML form is POSTed to the API using the *application/x-www-form-urlencoded* media type, and the `auto_parse_form_urlencoded` option is set, the framework will consume *stream* in order to parse the parameters and merge them into the query string parameters. In this case, the stream will be left at EOF.

---

bounded_stream

File-like wrapper around *stream* to normalize certain differences between the native input objects employed by different WSGI servers. In particular, *bounded_stream* is aware of the expected Content-Length of the body, and will never block on out-of-bounds reads, assuming the client does not stall while transmitting the data to the server.

For example, the following will not block when Content-Length is 0 or the header is missing altogether:

```
data = req.bounded_stream.read()
```

This is also safe:

```
doc = json.load(req.stream)
```

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.)

range_unit

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

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

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

if_unmodified_since

datetime – Value of the If-Unmodified-Since 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.

cookies
> *dict* – A dict of name/value cookie pairs. See also: *Getting Cookies*

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*)
> Retrieve the raw string value for the given header.

> > **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_header_as_datetime(*header*, *required=False*, *obs_date=False*)
> Return an HTTP header with HTTP-Date values as a datetime.

> > **Parameters**

> > - **name** (*str*) – Header name, case-insensitive (e.g., 'Date')
> > - **required** (*bool, optional*) – Set to `True` to raise `HTTPBadRequest` instead of returning gracefully when the header is not found (default `False`).
> > - **obs_date** (*bool, optional*) – Support obs-date formats according to RFC 7231, e.g.: "Sunday, 06-Nov-94 08:49:37 GMT" (default `False`).

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

> **Return type** datetime
>
> **Raises**
>
> - `HTTPBadRequest` – The header was not found in the request, but it was required.
>
> - `HttpInvalidHeader` – The header contained a malformed/invalid value.

get_param(*name*, *required=False*, *store=None*, *default=None*)
 Return the raw value of a query string parameter as a string.

Note: If an HTML form is POSTed to the API using the *application/x-www-form-urlencoded* media type, Falcon can automatically parse the parameters from the request body and merge them into the query string parameters. To enable this functionality, set `auto_parse_form_urlencoded` to `True` via `API.req_options`.

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.
>
> - **default** (*any, optional*) – If the param is not found returns the given value instead of None
>
> **Returns** The value of the param as a string, or `None` if param is not found and is not required.
>
> **Return type** str
>
> **Raises** `HTTPBadRequest` – A required param is missing from the request.

get_param_as_bool(*name*, *required=False*, *store=None*, *blank_as_true=False*)
 Return the value of a query string parameter as a boolean

 The following boolean strings are supported:

```
TRUE_STRINGS = ('true', 'True', 'yes', '1', 'on')
FALSE_STRINGS = ('false', 'False', 'no', '0', 'off')
```

> **Parameters**
>
> - **name** (*str*) – Parameter name, case-sensitive (e.g., 'detailed').
>
> - **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_date(*name*, *format_string='%Y-%m-%d'*, *required=False*, *store=None*)
Return the value of a query string parameter as a date.

Parameters

- **name** ([`str`]) – Parameter name, case-sensitive (e.g., 'ids').

- **format_string** ([`str`]) – String used to parse the param value into a date. Any format recognized by strptime() is supported. (default `"%Y-%m-%d"`)

- **required** (`bool, 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 and can be converted to a `date` according to the supplied format string. If the param is not found, returns `None` unless required is `True`.

Return type [datetime.date](#)

Raises

- `HTTPBadRequest` – A required param is missing from the request.

- `HTTPInvalidParam` – A transform function raised an instance of `ValueError`.

get_param_as_dict(*name*, *required=False*, *store=None*)
Return the value of a query string parameter as a dict.

Given a JSON value, parse and return it as a dict.

Parameters

- **name** ([`str`]) – Parameter name, case-sensitive (e.g., 'payload').

- **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`.

Return type [dict](#)

Raises

- `HTTPBadRequest` – A required param is missing from the request.

- `HTTPInvalidParam` – The parameter's value could not be parsed as JSON.

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

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

Return type list

Raises

- `HTTPBadRequest` – A required param is missing from the request.

- `HTTPInvalidParam` – A transform 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).

> **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*. If *stream* is set, but *stream_len* is not, Falcon will not supply a Content-Length header to the WSGI server. Consequently, the server may choose to use chunked encoding or one of the other strategies suggested by PEP-3333.

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

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.Response`, and then passing that new class to *falcon.API()* by way of the latter's *response_type* parameter.
>
> ---
>
> **Note:** When overriding *context_type* with a factory function (as opposed to a class), the function is called like a method of the current Response instance. Therefore the first argument is the Response instance itself (self).
>
> ---

accept_ranges
> Sets the Accept-Ranges header.
>
> The Accept-Ranges header field indicates to the client which range units are supported (e.g. "bytes") for the target resource.
>
> If range requests are not supported for the target resource, the header may be set to "none" to advise the client not to attempt any such requests.
>
> ---
>
> **Note:** "none" is the literal string, not Python's built-in `None` type.
>
> ---

add_link (*target*, *rel*, *title=None*, *title_star=None*, *anchor=None*, *hreflang=None*, *type_hint=None*)
> Add a link header to the response.
>
> See also: 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, Set-Cookie being the notable exceptions.

> > **Warning:** For setting cookies, see *set_cookie()*

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

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.

> This value will be URI encoded per RFC 3986. If the value that is being set is already URI encoded it should be decoded first or the header should be set manually using the set_header method.

content_range
> A tuple to use in constructing a value for the Content-Range header.
>
> The tuple has the form (*start*, *end*, *length*, [*unit*]), where *start* and *end* designate the range (inclusive), and *length* is the total length, or '*' if unknown. You may pass `int`'s for these numbers (no need to convert to `str` beforehand). The optional value *unit* describes the range unit and defaults to 'bytes'
>
> ---
> **Note:** You only need to use the alternate form, 'bytes */1234', for responses that use the status '416 Range Not Satisfiable'. In this case, raising `falcon.HTTPRangeNotSatisfiable` will do the right thing.
>
> See also: http://goo.gl/Iglhp
> ---

content_type
> Sets the Content-Type header.

etag
> Sets the ETag header.

get_header(*name*)
> Retrieve the raw string value for the given header.
>
> > **Parameters name** (`str`) – Header name, case-insensitive. Must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.
> >
> > **Returns** The header's value if set, otherwise `None`.
> >
> > **Return type** str

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.
>
> This value will be URI encoded per RFC 3986. If the value that is being set is already URI encoded it should be decoded first or the header should be set manually using the set_header method.

retry_after
> 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_cookie(*name*, *value*, *expires=None*, *max_age=None*, *domain=None*, *path=None*, *secure=True*, *http_only=True*)
> Set a response cookie.
>
> ---
> **Note:** This method can be called multiple times to add one or more cookies to the response.
> ---
>
> **See also:**
>
> To learn more about setting cookies, see *Setting Cookies*. The parameters listed below correspond to those defined in RFC 6265.

Parameters

- **name** (*str*) – Cookie name

- **value** (*str*) – Cookie value

Keyword Arguments

- **expires** (*datetime*) – Specifies when the cookie should expire. By default, cookies expire when the user agent exits.

  (See also: RFC 6265, Section 4.1.2.1)

- **max_age** (*int*) – Defines the lifetime of the cookie in seconds. By default, cookies expire when the user agent exits. If both *max_age* and *expires* are set, the latter is ignored by the user agent.

  Note: Coercion to `int` is attempted if provided with `float` or `str`.

  (See also: RFC 6265, Section 4.1.2.2)

- **domain** (*str*) – Restricts the cookie to a specific domain and any subdomains of that domain. By default, the user agent will return the cookie only to the origin server. When overriding this default behavior, the specified domain must include the origin server. Otherwise, the user agent will reject the cookie.

  (See also: RFC 6265, Section 4.1.2.3)

- **path** (*str*) – Scopes the cookie to the given path plus any subdirectories under that path (the "/" character is interpreted as a directory separator). If the cookie does not specify a path, the user agent defaults to the path component of the requested URI.

  Warning: User agent interfaces do not always isolate cookies by path, and so this should not be considered an effective security measure.

  (See also: RFC 6265, Section 4.1.2.4)

- **secure** (*bool*) – Direct the client to only return the cookie in subsequent requests if they are made over HTTPS (default: `True`). This prevents attackers from reading sensitive cookie data.

  Warning: For the *secure* cookie attribute to be effective, your application will need to enforce HTTPS.

  (See also: RFC 6265, Section 4.1.2.5)

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

  (See also: RFC 6265, Section 4.1.2.6)

Raises

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

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

set_header(*name*, *value*)
Set a header for this response to a given value.

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

> **Warning:** For setting cookies, see instead *set_cookie()*

> **Parameters**
> - **name** (*str*) – Header name (case-insensitive). The restrictions noted below for the header's value also apply here.
> - **value** (*str*) – Value for the header. Must be of type str or StringType and contain only US-ASCII characters. Under Python 2.x, the unicode type is also accepted, although such strings are also limited to US-ASCII.

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 a list of (*name*, *value*) tuples. Both *name* and *value* must be of type str or StringType and contain only US-ASCII characters. Under Python 2.x, the unicode type is also accepted, although such strings are also limited to US-ASCII.
>
> ---
> **Note:** Falcon can process a list of tuples slightly faster than a dict.
> ---

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

set_stream(*stream*, *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 when the length of the stream is known in advance.

Note: If the stream length is unknown, you can set *stream* directly, and ignore *stream_len*. In this case, the WSGI server may choose to use chunked encoding or one of the other strategies suggested by PEP-3333.
---

unset_cookie(*name*)
Unset a cookie in the response

Clears the contents of the cookie, and instructs the user agent to immediately expire its own copy of the cookie.

> **Warning:** In order to successfully remove a cookie, both the path and the domain must match the values that were used when the cookie was created.

vary

> Value to use for the Vary header.
>
> Set this property to an iterable of header names. For a single asterisk or field value, simply pass a single-element `list` or `tuple`.
>
> "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

6.3.3 Cookies

Cookie support is available in Falcon version 0.3 or later.

Getting Cookies

Cookies can be read from a request via the *cookies* request attribute:

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

        cookies = req.cookies

        if 'my_cookie' in cookies:
            my_cookie_value = cookies['my_cookie']
        # ....
```

The *cookies* attribute is a regular `dict` object.

Tip: The *cookies* attribute returns a copy of the response cookie dictionary. Assign it to a variable, as shown in the above example, to improve performance when you need to look up more than one cookie.

Setting Cookies

Setting cookies on a response is done via *set_cookie()*.

The *set_cookie()* method should be used instead of *set_header()* or *append_header()*. With *set_header()* you cannot set multiple headers with the same name (which is how multiple cookies are sent to the client). Furthermore, *append_header()* appends multiple values to the same header field in a way that is not compatible with the special format required by the *Set-Cookie* header.

Simple example:

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

        # Set the cookie 'my_cookie' to the value 'my cookie value'
        resp.set_cookie('my_cookie', 'my cookie value')
```

You can of course also set the domain, path and lifetime of the cookie.

```python
class Resource(object):
    def on_get(self, req, resp):
        # Set the maximum age of the cookie to 10 minutes (600 seconds)
        # and the cookie's domain to 'example.com'
        resp.set_cookie('my_cookie', 'my cookie value',
                        max_age=600, domain='example.com')
```

You can also instruct the client to remove a cookie with the *unset_cookie()* method:

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

        # Clear the bad cookie
        resp.unset_cookie('bad_cookie')
```

The Secure Attribute

By default, Falcon sets the *secure* attribute for cookies. This instructs the client to never transmit the cookie in the clear over HTTP, in order to protect any sensitive data that cookie might contain. If a cookie is set, and a subsequent request is made over HTTP (rather than HTTPS), the client will not include that cookie in the request.

> **Warning:** For this attribute to be effective, your application will need to enforce HTTPS when setting the cookie, as well as in all subsequent requests that require the cookie to be sent back from the client.

When running your application in a development environment, you can disable this behavior by passing *secure=False* to *set_cookie()*. This lets you test your app locally without having to set up TLS. You can make this option configurable to easily switch between development and production environments.

See also: RFC 6265, Section 4.1.2.5

6.3.4 Status Codes

Falcon provides a list of constants for common HTTP response status codes.

For example:

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

Or, using the more verbose name:

```python
resp.status = falcon.HTTP_CONFLICT
```

Using these constants helps avoid typos and cuts down on the number of string objects that must be created when preparing responses.

Falcon also provides a generic *HTTPStatus* class. Raise this class from a hook, middleware, or a responder to stop handling the request and skip to the response handling. It takes status, additional headers and body as input arguments.

HTTPStatus

class `falcon.`**`HTTPStatus`**(*status*, *headers=None*, *body=None*)
Represents a generic HTTP status.

Raise an instance of this class from a hook, middleware, or responder to short-circuit request processing in a manner similar to `falcon.HTTPError`, but for non-error status codes.

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

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

`body`
str or unicode – String representing response content. If Unicode, Falcon will encode as UTF-8 in the response.

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

1xx Informational

```
HTTP_CONTINUE = HTTP_100
HTTP_SWITCHING_PROTOCOLS = HTTP_101


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

2xx Success

```
HTTP_OK = HTTP_200
HTTP_CREATED = HTTP_201


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_MULTIPLE_CHOICES = HTTP_300
HTTP_MOVED_PERMANENTLY = HTTP_301
HTTP_FOUND = HTTP_302
HTTP_SEE_OTHER = HTTP_303
```

```
HTTP_NOT_MODIFIED = HTTP_304
HTTP_USE_PROXY = HTTP_305
HTTP_TEMPORARY_REDIRECT = HTTP_307


HTTP_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_BAD_REQUEST = HTTP_400
HTTP_UNAUTHORIZED = HTTP_401  # <-- Really means "unauthenticated"
HTTP_PAYMENT_REQUIRED = HTTP_402
HTTP_FORBIDDEN = HTTP_403  # <-- Really means "unauthorized"
HTTP_NOT_FOUND = HTTP_404
HTTP_METHOD_NOT_ALLOWED = HTTP_405
HTTP_NOT_ACCEPTABLE = HTTP_406
HTTP_PROXY_AUTHENTICATION_REQUIRED = HTTP_407
HTTP_REQUEST_TIMEOUT = HTTP_408
HTTP_CONFLICT = HTTP_409
HTTP_GONE = HTTP_410
HTTP_LENGTH_REQUIRED = HTTP_411
HTTP_PRECONDITION_FAILED = HTTP_412
HTTP_REQUEST_ENTITY_TOO_LARGE = HTTP_413
HTTP_REQUEST_URI_TOO_LONG = HTTP_414
HTTP_UNSUPPORTED_MEDIA_TYPE = HTTP_415
HTTP_REQUESTED_RANGE_NOT_SATISFIABLE = HTTP_416
HTTP_EXPECTATION_FAILED = HTTP_417
HTTP_IM_A_TEAPOT = HTTP_418
HTTP_UNPROCESSABLE_ENTITY = HTTP_422
HTTP_UPGRADE_REQUIRED = HTTP_426
HTTP_PRECONDITION_REQUIRED = HTTP_428
HTTP_TOO_MANY_REQUESTS = HTTP_429
HTTP_REQUEST_HEADER_FIELDS_TOO_LARGE = HTTP_431
HTTP_UNAVAILABLE_FOR_LEGAL_REASONS = HTTP_451


HTTP_400 = '400 Bad Request'
HTTP_401 = '401 Unauthorized'  # <-- Really means "unauthenticated"
HTTP_402 = '402 Payment Required'
HTTP_403 = '403 Forbidden'  # <-- Really means "unauthorized"
HTTP_404 = '404 Not Found'
HTTP_405 = '405 Method Not Allowed'
HTTP_406 = '406 Not Acceptable'
HTTP_407 = '407 Proxy Authentication Required'
HTTP_408 = '408 Request Time-out'
HTTP_409 = '409 Conflict'
HTTP_410 = '410 Gone'
HTTP_411 = '411 Length Required'
HTTP_412 = '412 Precondition Failed'
HTTP_413 = '413 Payload Too Large'
HTTP_414 = '414 URI Too Long'
HTTP_415 = '415 Unsupported Media Type'
```

```
HTTP_416 = '416 Range Not Satisfiable'
HTTP_417 = '417 Expectation Failed'
HTTP_418 = "418 I'm a teapot"
HTTP_422 = "422 Unprocessable Entity"
HTTP_426 = '426 Upgrade Required'
HTTP_428 = '428 Precondition Required'
HTTP_429 = '429 Too Many Requests'
HTTP_431 = '431 Request Header Fields Too Large'
HTTP_451 = '451 Unavailable For Legal Reasons'
```

5xx Server Error

```
HTTP_INTERNAL_SERVER_ERROR = HTTP_500
HTTP_NOT_IMPLEMENTED = HTTP_501
HTTP_BAD_GATEWAY = HTTP_502
HTTP_SERVICE_UNAVAILABLE = HTTP_503
HTTP_GATEWAY_TIMEOUT = HTTP_504
HTTP_HTTP_VERSION_NOT_SUPPORTED = HTTP_505
HTTP_NETWORK_AUTHENTICATION_REQUIRED = HTTP_511


HTTP_500 = '500 Internal Server Error'
HTTP_501 = '501 Not Implemented'
HTTP_502 = '502 Bad Gateway'
HTTP_503 = '503 Service Unavailable'
HTTP_504 = '504 Gateway Time-out'
HTTP_505 = '505 HTTP Version not supported'
HTTP_511 = '511 Network Authentication Required'
```

6.3.5 Error Handling

When a request results in an error condition, you can 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.

All classes are available directly from the *falcon* package namespace:

```python
import falcon

class MessageResource(object):
    def on_get(self, req, resp):

        # ...

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

        # ...
```

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.
>
> **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. If not provided, defaults to the HTTP status line as determined by the status argument.
>
> - **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two (default None).
>
> - **headers** (*dict or list*) – A dict of header names and values to set, or a list of (*name*, *value*) tuples. Both *name* and *value* must be of type str or StringType, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.
>
>   ---
>   **Note:** The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client
>
>   ---
>
>   ---
>   **Note:** Falcon can process a list of tuple slightly faster than a dict.
>
>   ---
>
> - **headers** (*dict*) – 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 (default 'API documentation for this error').

- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

to_dict(*obj_type=<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.
>
> **Return type** dict

to_json()

Returns a pretty-printed JSON representation of the error.

> **Returns** A JSON document for the error.
>
> **Return type** str

to_xml()

Returns an XML-encoded representation of the error.

> **Returns** An XML document for the error.
>
> **Return type** str

Mixins

class falcon.http_error.**NoRepresentation**

Mixin for `HTTPError` child classes that have no representation.

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

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

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

Predefined Errors

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

400 Bad Request.

The server cannot or will not process the request due to something that is perceived to be a client error (e.g., malformed request syntax, invalid request message framing, or deceptive request routing).

(See also: RFC 7231, Section 6.5.1)

> **Keyword Arguments**

- **title** (*str*) – Error title (default '400 Bad Request').

- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.

- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note: The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  ---

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

- **headers** (*dict*) – 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 (default 'API documentation for this error').

- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

exception falcon.**HTTPInvalidHeader**(*msg*, *header_name*, ***kwargs*)
   400 Bad Request.

   One of the headers in the request is invalid.

   Parameters

   - **msg** (`str`) – A description of why the value is invalid.

   - **header_name** (`str`) – The name of the invalid header.

   Keyword Arguments

   - **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

     Note: The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

     ---

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

   - **headers** (*dict*) – 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 (default 'API documentation for this error').

- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

exception falcon.**HTTPMissingHeader**(*header_name*, ***kwargs*)
   400 Bad Request

   A header is missing from the request.

   Parameters header_name (`str`) – The name of the missing header.

   Keyword Arguments

   - **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

      Note: The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

      ---

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

   - **headers** (*dict*) – 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 (default 'API documentation for this error').
   - **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

exception falcon.**HTTPInvalidParam**(*msg*, *param_name*, ***kwargs*)
   400 Bad Request

   A parameter in the request is invalid. This error may refer to a parameter in a query string, form, or document that was submitted with the request.

   Parameters

   - **msg** (`str`) – A description of the invalid parameter.
   - **param_name** (`str`) – The name of the parameter.

   Keyword Arguments

   - **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

      Note: The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

      ---

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

- **headers** (*dict*) – 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 (default 'API documentation for this error').

- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

exception falcon.**HTTPMissingParam**(*param_name*, ***kwargs*)
    400 Bad Request

A parameter is missing from the request. This error may refer to a parameter in a query string, form, or document that was submitted with the request.

> **Parameters** **param_name** (`str`) – The name of the missing parameter.

> **Keyword Arguments**

- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note: The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  ---

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

- **headers** (*dict*) – 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 (default 'API documentation for this error').

- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

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

The request has not been applied because it lacks valid authentication credentials for the target resource.

The server generating a 401 response MUST send a WWW-Authenticate header field containing at least one challenge applicable to the target resource.

If the request included authentication credentials, then the 401 response indicates that authorization has been refused for those credentials. The user agent MAY repeat the request with a new or replaced Authorization header field. If the 401 response contains the same challenge as the prior response, and the user agent has already attempted authentication at least once, then the user agent SHOULD present the enclosed representation to the user, since it usually contains relevant diagnostic information.

(See also: RFC 7235, Section 3.1)

> **Keyword Arguments**

- **title** (*str*) – Error title (default '401 Unauthorized').

- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.

- **challenges** (*iterable of str*) – One or more authentication challenges to use as the value of the WWW-Authenticate header in the response (see also RFC 7235, Section 2.1).

- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

Note: The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

---

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

- **headers** (*dict*) – 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 (default 'API documentation for this error').

- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

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

The server understood the request but refuses to authorize it.

A server that wishes to make public why the request has been forbidden can describe that reason in the response payload (if any).

If authentication credentials were provided in the request, the server considers them insufficient to grant access. The client SHOULD NOT automatically repeat the request with the same credentials. The client MAY repeat the request with new or different credentials. However, a request might be forbidden for reasons unrelated to the credentials.

An origin server that wishes to "hide" the current existence of a forbidden target resource MAY instead respond with a status code of 404 Not Found.

(See also: RFC 7231, Section 6.5.4)

Keyword Arguments

- **title** (*str*) – Error title (default '403 Forbidden').

- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.

- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

Note: The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

---

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

- **headers** (*dict*) – 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 (default 'API documentation for this error').

- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

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

The origin server did not find a current representation for the target resource or is not willing to disclose that one exists.

A 404 status code does not indicate whether this lack of representation is temporary or permanent; the 410 Gone status code is preferred over 404 if the origin server knows, presumably through some configurable means, that the condition is likely to be permanent.

A 404 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls.

(See also: RFC 7231, Section 6.5.3)

Keyword Arguments

- **title** (*str*) – Human-friendly error title. If not provided, and *description* is also not provided, no body will be included in the response.

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

- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

Note: The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

---

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

- **headers** (*dict*) – 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 (default 'API documentation for this error').

- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

exception falcon.**HTTPMethodNotAllowed**(*allowed_methods*, ***kwargs*)

  405 Method Not Allowed.

  The method received in the request-line is known by the origin server but not supported by the target resource.

  The origin server MUST generate an Allow header field in a 405 response containing a list of the target resource's currently supported methods.

  A 405 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls.

  (See also: RFC 7231, Section 6.5.5)

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

  Keyword Arguments

  - **title** (*str*) – Human-friendly error title. If not provided, and *description* is also not provided, no body will be included in the response.

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

  - **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

    Note: The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

    ---

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

  - **headers** (*dict*) – 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 (default 'API documentation for this error').

  - **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

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

  406 Not Acceptable.

  The target resource does not have a current representation that would be acceptable to the user agent, according to the proactive negotiation header fields received in the request, and the server is unwilling to supply a default representation.

  The server SHOULD generate a payload containing a list of available representation characteristics and corresponding resource identifiers from which the user or user agent can choose the one most appropriate. A user

agent MAY automatically select the most appropriate choice from that list. However, this specification does not define any standard for such automatic selection, as described in RFC 7231, Section 6.4.1

(See also: RFC 7231, Section 6.5.6)

Keyword Arguments

- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.

- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note: The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  ---

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

- **headers** (*dict*) – 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 (default 'API documentation for this error').

- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

exception `falcon.`**`HTTPConflict`**(*title=None*, *description=None*, ***kwargs*)
409 Conflict.

The request could not be completed due to a conflict with the current state of the target resource. This code is used in situations where the user might be able to resolve the conflict and resubmit the request.

The server SHOULD generate a payload that includes enough information for a user to recognize the source of the conflict.

Conflicts are most likely to occur in response to a PUT request. For example, if versioning were being used and the representation being PUT included changes to a resource that conflict with those made by an earlier (third-party) request, the origin server might use a 409 response to indicate that it can't complete the request. In this case, the response representation would likely contain information useful for merging the differences based on the revision history.

(See also: RFC 7231, Section 6.5.8)

Keyword Arguments

- **title** (*str*) – Error title (default '409 Conflict').

- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.

- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

> **Note:** The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

> **Note:** Falcon can process a list of `tuple` slightly faster than a `dict`.

- **headers** (*dict*) – 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 (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

exception `falcon.`**`HTTPGone`**(***kwargs*)

   410 Gone.

   The target resource is no longer available at the origin server and this condition is likely to be permanent.

   If the origin server does not know, or has no facility to determine, whether or not the condition is permanent, the status code 404 Not Found ought to be used instead.

   The 410 response is primarily intended to assist the task of web maintenance by notifying the recipient that the resource is intentionally unavailable and that the server owners desire that remote links to that resource be removed. Such an event is common for limited-time, promotional services and for resources belonging to individuals no longer associated with the origin server's site. It is not necessary to mark all permanently unavailable resources as "gone" or to keep the mark for any length of time – that is left to the discretion of the server owner.

   A 410 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls.

   (See also: RFC 7231, Section 6.5.9)

   Keyword Arguments

   - **title** (*str*) – Human-friendly error title. If not provided, and *description* is also not provided, no body will be included in the response.
   - **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two (default `None`).
   - **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

   > **Note:** The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

   > **Note:** Falcon can process a list of `tuple` slightly faster than a `dict`.

- **headers** (*dict*) – 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 (default 'API documentation for this error').

- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

exception falcon.**HTTPLengthRequired**(*title=None*, *description=None*, ***kwargs*)
   411 Length Required.

   The server refuses to accept the request without a defined Content- Length.

   The client MAY repeat the request if it adds a valid Content-Length header field containing the length of the message body in the request message.

   (See also: RFC 7231, Section 6.5.10)

   Keyword Arguments

- **title** (*str*) – Error title (default '411 Length Required').

- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.

- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

   Note: The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

   ---

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

- **headers** (*dict*) – 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 (default 'API documentation for this error').

- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

exception falcon.**HTTPPreconditionFailed**(*title=None*, *description=None*, ***kwargs*)
   412 Precondition Failed.

   One or more conditions given in the request header fields evaluated to false when tested on the server.

   This response code allows the client to place preconditions on the current resource state (its current representations and metadata) and, thus, prevent the request method from being applied if the target resource is in an unexpected state.

   (See also: RFC 7232, Section 4.2)

   Keyword Arguments

- **title** (*str*) – Error title (default '412 Precondition Failed').

- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.

- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note: The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  ---

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

- **headers** (*dict*) – 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 (default 'API documentation for this error').

- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

exception `falcon.`**`HTTPRequestEntityTooLarge`**(*title=None*, *description=None*, *retry_after=None*, ***kwargs*)

413 Request Entity Too Large.

The server is refusing to process a request because the request payload is larger than the server is willing or able to process.

The server MAY close the connection to prevent the client from continuing the request.

If the condition is temporary, the server SHOULD generate a Retry- After header field to indicate that it is temporary and after what time the client MAY try again.

(See also: RFC 7231, Section 6.5.11)

Keyword Arguments

- **title** (*str*) – Error title (default '413 Request Entity Too Large').

- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.

- **retry_after** (*datetime or int*) – Value for the Retry-After header. If a `datetime` object, will serialize as an HTTP date. Otherwise, a non-negative `int` is expected, representing the number of seconds to wait.

- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

  Note: The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

  ---

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

- **headers** (*dict*) – 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 (default 'API documentation for this error').

- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

exception falcon.**HTTPUriTooLong**(*title=None*, *description=None*, ***kwargs*)
    414 URI Too Long.

The server is refusing to service the request because the request- target is longer than the server is willing to interpret.

This rare condition is only likely to occur when a client has improperly converted a POST request to a GET request with long query information, when the client has descended into a "black hole" of redirection (e.g., a redirected URI prefix that points to a suffix of itself) or when the server is under attack by a client attempting to exploit potential security holes.

A 414 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls.

(See also: RFC 7231, Section 6.5.12)

> **Keyword Arguments**
>
> - **title** (*str*) – Error title (default '414 URI Too Long').
>
> - **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two (default `None`).
>
> - **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.
>
>   ---
>   **Note:** The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client
>
>   ---
>
>   ---
>   **Note:** Falcon can process a list of `tuple` slightly faster than a `dict`.
>
>   ---
>
> - **headers** (*dict*) – 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 (default 'API documentation for this error').
>
> - **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

---

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

   415 Unsupported Media Type.

   The origin server is refusing to service the request because the payload is in a format not supported by this method on the target resource.

   The format problem might be due to the request's indicated Content- Type or Content-Encoding, or as a result of inspecting the data directly.

   (See also: RFC 7231, Section 6.5.13)

   Keyword Arguments

- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.

- **headers** (*dict or list*) – A dict of header names and values to set, or a list of (*name*, *value*) tuples. Both *name* and *value* must be of type str or StringType, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

   Note: The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client
   ---

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

- **headers** (*dict*) – 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 (default 'API documentation for this error').

- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default None).

exception falcon.**HTTPRangeNotSatisfiable**(*resource_length*)

   416 Range Not Satisfiable.

   None of the ranges in the request's Range header field overlap the current extent of the selected resource or that the set of ranges requested has been rejected due to invalid ranges or an excessive request of small or overlapping ranges.

   For byte ranges, failing to overlap the current extent means that the first-byte-pos of all of the byte-range-spec values were greater than the current length of the selected representation. When this status code is generated in response to a byte-range request, the sender SHOULD generate a Content-Range header field specifying the current length of the selected representation.

   (See also: RFC 7233, Section 4.4)

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

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

   422 Unprocessable Entity.

The server understands the content type of the request entity (hence a 415 Unsupported Media Type status code is inappropriate), and the syntax of the request entity is correct (thus a 400 Bad Request status code is inappropriate) but was unable to process the contained instructions.

For example, this error condition may occur if an XML request body contains well-formed (i.e., syntactically correct), but semantically erroneous, XML instructions.

(See also: RFC 4918, Section 11.2)

> **Keyword Arguments**
>
> - **title** (*str*) – Error title (default '422 Unprocessable Entity').
>
> - **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.
>
> - **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.
>
> ---
>
> **Note:** The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client
>
> ---
>
> ---
>
> **Note:** Falcon can process a list of `tuple` slightly faster than a `dict`.
>
> ---
>
> - **headers** (*dict*) – 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 (default 'API documentation for this error').
>
> - **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

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

429 Too Many Requests.

The user has sent too many requests in a given amount of time ("rate limiting").

The response representations SHOULD include details explaining the condition, and MAY include a Retry-After header indicating how long to wait before making a new request.

Responses with the 429 status code MUST NOT be stored by a cache.

(See also: RFC 6585, Section 4)

> **Keyword Arguments**
>
> - **title** (*str*) – Error title (default '429 Too Many Requests').
>
> - **description** (*str*) – Human-friendly description of the rate limit that was exceeded.
>
> - **retry_after** (*datetime or int*) – Value for the Retry-After header. If a `datetime` object, will serialize as an HTTP date. Otherwise, a non-negative `int` is expected, representing the number of seconds to wait.

- **headers** (*dict or list*) – A dict of header names and values to set, or a list of (*name*, *value*) tuples. Both *name* and *value* must be of type str or StringType, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

    Note: The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

    ---

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

- **headers** (*dict*) – 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 (default 'API documentation for this error').

- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default None).

exception falcon.**HTTPUnavailableForLegalReasons**(*title=None*, ***kwargs*)
451 Unavailable For Legal Reasons.

The server is denying access to the resource as a consequence of a legal demand.

The server in question might not be an origin server. This type of legal demand typically most directly affects the operations of ISPs and search engines.

Responses using this status code SHOULD include an explanation, in the response body, of the details of the legal demand: the party making it, the applicable legislation or regulation, and what classes of person and resource it applies to.

Note that in many cases clients can still access the denied resource by using technical countermeasures such as a VPN or the Tor network.

A 451 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls.

(See also: RFC 7725, Section 3)

Keyword Arguments

- **title** (*str*) – Error title (default '451 Unavailable For Legal Reasons').

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

- **headers** (*dict or list*) – A dict of header names and values to set, or a list of (*name*, *value*) tuples. Both *name* and *value* must be of type str or StringType, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

    Note: The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

    ---

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

- **headers** (*dict*) – 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 (default 'API documentation for this error').

- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

exception `falcon.`**`HTTPInternalServerError`**(*title=None*, *description=None*, ***kwargs*)
    500 Internal Server Error.

The server encountered an unexpected condition that prevented it from fulfilling the request.

(See also: RFC 7231, Section 6.6.1)

Keyword Arguments

- **title** (*str*) – Error title (default '500 Internal Server Error').

- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.

- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

Note: The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

---

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

- **headers** (*dict*) – 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 (default 'API documentation for this error').

- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

exception `falcon.`**`HTTPBadGateway`**(*title=None*, *description=None*, ***kwargs*)
    502 Bad Gateway.

The server, while acting as a gateway or proxy, received an invalid response from an inbound server it accessed while attempting to fulfill the request.

(See also: RFC 7231, Section 6.6.3)

Keyword Arguments

- **title** (*str*) – Error title (default '502 Bad Gateway').

- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.

- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

Note: The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

---

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

- **headers** (*dict*) – 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 (default 'API documentation for this error').

- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

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

503 Service Unavailable.

The server is currently unable to handle the request due to a temporary overload or scheduled maintenance, which will likely be alleviated after some delay.

The server MAY send a Retry-After header field to suggest an appropriate amount of time for the client to wait before retrying the request.

Note: The existence of the 503 status code does not imply that a server has to use it when becoming overloaded. Some servers might simply refuse the connection.

(See also: RFC 7231, Section 6.6.4)

Keyword Arguments

- **title** (*str*) – Error title (default '503 Service Unavailable').

- **description** (*str*) – Human-friendly description of the error, along with a helpful suggestion or two.

- **retry_after** (*datetime or int*) – Value for the Retry-After header. If a `datetime` object, will serialize as an HTTP date. Otherwise, a non-negative `int` is expected, representing the number of seconds to wait.

- **headers** (*dict or list*) – A `dict` of header names and values to set, or a `list` of (*name*, *value*) tuples. Both *name* and *value* must be of type `str` or `StringType`, and only character values 0x00 through 0xFF may be used on platforms that use wide characters.

Note: The Content-Type header, if present, will be overridden. If you wish to return custom error messages, you can create your own HTTP error class, and install an error handler to convert it into an appropriate HTTP response for the client

---

> **Note:** Falcon can process a list of `tuple` slightly faster than a `dict`.

- **headers** (*dict*) – 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 (default 'API documentation for this error').
- **code** (*int*) – An internal code that customers can reference in their support request or to help them when searching for knowledge base articles related to this error (default `None`).

6.3.6 Redirection

Falcon defines a set of exceptions that can be raised within a middleware method, hook, or responder in order to trigger a 3xx (Redirection) response to the client. Raising one of these classes short-circuits request processing in a manner similar to raising an instance or subclass of *HTTPError*

Redirects

exception `falcon.`**`HTTPMovedPermanently`**(*location*)
301 Moved Permanently.

The 301 (Moved Permanently) status code indicates that the target resource has been assigned a new permanent URI.

> **Note:** For historical reasons, a user agent MAY change the request method from POST to GET for the subsequent request. If this behavior is undesired, the 308 (Permanent Redirect) status code can be used instead.

See also: https://tools.ietf.org/html/rfc7231#section-6.4.2

> **Parameters** `location` (`str`) – URI to provide as the Location header in the response.

exception `falcon.`**`HTTPFound`**(*location*)
302 Found.

The 302 (Found) status code indicates that the target resource resides temporarily under a different URI. Since the redirection might be altered on occasion, the client ought to continue to use the effective request URI for future requests.

> **Note:** For historical reasons, a user agent MAY change the request method from POST to GET for the subsequent request. If this behavior is undesired, the 307 (Temporary Redirect) status code can be used instead.

See also: https://tools.ietf.org/html/rfc7231#section-6.4.3

> **Parameters** `location` (`str`) – URI to provide as the Location header in the response.

exception `falcon.`**`HTTPSeeOther`**(*location*)
303 See Other.

The 303 (See Other) status code indicates that the server is redirecting the user agent to a *different* resource, as indicated by a URI in the Location header field, which is intended to provide an indirect response to the original request.

A 303 response to a GET request indicates that the origin server does not have a representation of the target resource that can be transferred over HTTP. However, the Location header in the response may be dereferenced to obtain a representation for an alternative resource. The recipient may find this alternative useful, even though it does not represent the original target resource.

Note: The new URI in the Location header field is not considered equivalent to the effective request URI.

See also: https://tools.ietf.org/html/rfc7231#section-6.4.4

> **Parameters** `location` (`str`) – URI to provide as the Location header in the response.

exception `falcon.`**`HTTPTemporaryRedirect`**(*location*)
> 307 Temporary Redirect.
>
> The 307 (Temporary Redirect) status code indicates that the target resource resides temporarily under a different URI and the user agent MUST NOT change the request method if it performs an automatic redirection to that URI. Since the redirection can change over time, the client ought to continue using the original effective request URI for future requests.
>
> **Note:** This status code is similar to 302 (Found), except that it does not allow changing the request method from POST to GET.
>
> See also: https://tools.ietf.org/html/rfc7231#section-6.4.7
>
> > **Parameters** `location` (`str`) – URI to provide as the Location header in the response.

exception `falcon.`**`HTTPPermanentRedirect`**(*location*)
> 308 Permanent Redirect.
>
> The 308 (Permanent Redirect) status code indicates that the target resource has been assigned a new permanent URI.
>
> **Note:** This status code is similar to 301 (Moved Permanently), except that it does not allow changing the request method from POST to GET.
>
> See also: https://tools.ietf.org/html/rfc7238#section-3
>
> > **Parameters** `location` (`str`) – URI to provide as the Location header in the response.

6.3.7 Middleware

Middleware components provide a way to execute logic before the framework routes each request, after each request is routed but before the target responder is called, or just before the response is returned for each request. Components are registered with the *middleware* kwarg when instantiating Falcon's *API class*.

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

Falcon's middleware interface is defined as follows:

```python
class ExampleComponent(object):
    def process_request(self, req, resp):
        """Process the request before routing it.
```

```python
        Args:
            req: Request object that will eventually be
                routed to an on_* responder method.
            resp: Response object that will be routed to
                the on_* responder.
        """

    def process_resource(self, req, resp, resource, params):
        """Process the request after routing.

        Note:
            This method is only called when the request matches
            a route to a resource.

        Args:
            req: Request object that will be passed to the
                routed responder.
            resp: Response object that will be passed to the
                responder.
            resource: Resource object to which the request was
                routed.
            params: A dict-like object representing any additional
                params derived from the route's URI template fields,
                that will be passed to the resource's responder
                method as keyword arguments.
        """

    def process_response(self, req, resp, resource, req_succeeded):
        """Post-processing of the response (after routing).

        Args:
            req: Request object.
            resp: Response object.
            resource: Resource object to which the request was
                routed. May be None if no route was found
                for the request.
            req_succeeded: True if no exceptions were raised while
                the framework processed and routed the request;
                otherwise False.
        """
```

Tip: Because *process_request* executes before routing has occurred, if a component modifies `req.path` in its *process_request* method, the framework will use the modified value to route the request.

Tip: The *process_resource* method is only called when the request matches a route to a resource. To take action when a route is not found, a *sink* may be used instead.

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.

6.3.8 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:

```python
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

6.3.9 Routing

The *falcon.routing* module contains utilities used internally by `falcon.API()` to route requests. They are exposed here for use by custom routing engines.

A custom router is any class that implements the following interface:

```
class FancyRouter(object):
    def add_route(self, uri_template, method_map, resource):
        """Adds a route between URI path template and resource.

        Args:
```

```
            uri_template (str): The URI template to add.
            method_map (dict): A method map obtained by calling
                falcon.routing.create_http_method_map.
            resource (object): Instance of the resource class that
                will handle requests for the given URI.
        """


    def find(self, uri):
        """Search for a route that matches the given partial URI.

        Args:
            uri(str): The requested path to route

        Returns:
            tuple: A 4-member tuple composed of (resource, method_map,
                params, uri_template), or ``None`` if no route matches
                the requested path
        """
```

A custom routing engine may be specified when instantiating `falcon.API()`. For example:

```
fancy = FancyRouter()
api = API(router=fancy)
```

Default router and utility functions.

This package implements Falcon's default routing engine, and also includes utility functions to aid in the implementation of custom routers.

`falcon.routing.`**`create_http_method_map`**(*resource*)
    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)`.
>
> **Returns** A mapping of HTTP methods to responders.
>
> **Return type** dict

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

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

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

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

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

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

> **Parameters template** (`str`) – The template to compile. Note that field names are restricted to ASCII a-z, A-Z, and the underscore character.
>
> **Returns** (template_field_names, template_regex)
>
> **Return type** tuple

class `falcon.routing.`**`CompiledRouter`**

Fast URI router which compiles its routing logic to Python code.

Generally you do not need to use this router class directly, as an instance is created by default when the falcon.API class is initialized.

The router treats URI paths as a tree of URI segments and searches by checking the URI one segment at a time. Instead of interpreting the route tree for each look-up, it generates inlined, bespoke Python code to perform the search, then compiles that code. This makes the route processing quite fast.

`add_route`(*uri_template*, *method_map*, *resource*)

Adds a route between a URI path template and a resource.

> **Parameters**
>
> - **uri_template** (`str`) – A URI template to use for the route
>
> - **method_map** (`dict`) – A mapping of HTTP methods (e.g., 'GET', 'POST') to methods of a resource object.
>
> - **resource** (`object`) – The resource instance to associate with the URI template.

`find`(*uri*)

Search for a route that matches the given partial URI.

> **Parameters uri** (`str`) – The requested path to route
>
> **Returns**
>
> > **A 4-member tuple composed of (resource, method_map,** params, uri_template), or `None` if no route matches the requested path
>
> **Return type** tuple

6.3.10 Utilities

URI Functions

URI utilities.

This module provides utility functions to parse, encode, decode, and otherwise manipulate a URI. These functions are not available directly in the *falcon* module, and so must be explicitly imported:

```python
from falcon import uri

name, port = uri.parse_host('example.org:8080')
```

`falcon.uri.`**`encode`**(*uri*)

Encodes a full or relative URI according to RFC 3986.

RFC 3986 defines a set of "unreserved" characters as well as a set of "reserved" characters used as delimiters. This function escapes all other "disallowed" characters by percent-encoding them.

Note: This utility is faster in the average case than the similar *quote* function found in `urlib`. It also strives to be easier to use by assuming a sensible default of allowed characters.

> **Parameters uri** (`str`) – URI or part of a URI to encode. If this is a wide string (i.e., `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.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.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.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.uri.**parse_query_string**(*query_string*, *keep_blank_qs_values=False*, *parse_qs_csv=True*)

> Parse a query string into a dict.
>
> Query string parameters are assumed to use standard form-encoding. Only parameters with values are returned. For example, given 'foo=bar&flag', this function would ignore 'flag' unless the *keep_blank_qs_values* option is set.
>
> ---
>
> **Note:** In addition to the standard HTML form-based method for specifying lists by repeating a given param multiple times, Falcon supports a more compact form in which the param may be given a single time but set to a `list` of comma-separated elements (e.g., 'foo=a,b,c').
>
> When using this format, all commas uri-encoded will not be treated by Falcon as a delimiter. If the client wants to send a value as a list, it must not encode the commas with the values.
>
> The two different ways of specifying lists may not be mixed in a single query string for the same parameter.
>
> ---
>
> > **Parameters**
> >
> > - **query_string** (*str*) – The query string to parse.
> > - **keep_blank_qs_values** (*bool*) – Set to `True` to return fields even if they do not have a value (default `False`). For comma-separated values, this option also determines whether or not empty elements in the parsed list are retained.
> > - **parse_qs_csv** – Set to `False` in order to disable splitting query parameters on `,` (default `True`). Depending on the user agent, encoding lists as multiple occurrences of the same parameter might be preferable. In this case, setting *parse_qs_csv* to `False` will cause the framework to treat commas as literal characters in each occurring parameter value.
> >
> > **Returns** A dictionary of (*name*, *value*) pairs, one per query parameter. Note that *value* may be a single `str`, or a `list` of `str`.
> >
> > **Return type** dict
> >
> > **Raises** `TypeError` – *query_string* was not a `str`.

falcon.uri.**unquote_string**(*quoted*)

> Unquote an RFC 7320 "quoted-string".
>
> > **Parameters** **quoted** (*str*) – Original quoted string
> >
> > **Returns** unquoted string
> >
> > **Return type** str
> >
> > **Raises** `TypeError` – *quoted* was not a `str`.

Miscellaneous

falcon.**deprecated**(*instructions*)

> Flags a method as deprecated.
>
> This function returns a decorator which can be used to mark deprecated functions. Applying this decorator will result in a warning being emitted when the function is used.

6.3. Classes and Functions 83

Parameters instructions (*str*) – Specific guidance for the developer, e.g.: 'Please migrate to add_proxy(...)''

falcon.**http_now**()

Returns the current UTC time as an IMF-fixdate.

Returns The current UTC time as an IMF-fixdate, e.g., 'Tue, 15 Nov 1994 12:45:26 GMT'.

Return type str

falcon.**dt_to_http**(*dt*)

Converts a datetime instance to an HTTP date string.

Parameters dt (*datetime*) – A datetime instance to convert, assumed to be UTC.

Returns An RFC 1123 date string, e.g.: "Tue, 15 Nov 1994 12:45:26 GMT".

Return type str

falcon.**http_date_to_dt**(*http_date*, *obs_date=False*)

Converts an HTTP date string to a datetime instance.

Parameters

- **http_date** (*str*) – An RFC 1123 date string, e.g.: "Tue, 15 Nov 1994 12:45:26 GMT".

- **obs_date** (*bool, optional*) – Support obs-date formats according to RFC 7231, e.g.: "Sunday, 06-Nov-94 08:49:37 GMT" (default False).

Returns A UTC datetime instance corresponding to the given HTTP date.

Return type datetime

Raises ValueError – http_date doesn't match any of the available time formats

falcon.**to_query_str**(*params*, *comma_delimited_lists=True*, *prefix=True*)

Converts a dictionary of parameters to a query string.

Parameters

- **params** (*dict*) – A dictionary of parameters, where each key is a parameter name, and each value is either a str or something that can be converted into a str, or a list of such values. If a list, the value will be converted to a comma-delimited string of values (e.g., 'thing=1,2,3').

- **comma_delimited_lists** (*bool*) – Set to False to encode list values by specifying multiple instances of the parameter (e.g., 'thing=1&thing=2&thing=3'). Otherwise, parameters will be encoded as comma-separated values (e.g., 'thing=1,2,3'). Defaults to True.

- **prefix** (*bool*) – Set to False to exclude the '?' prefix in the result string (default True).

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

Return type str

falcon.**get_http_status**(*status_code*, *default_reason='Unknown'*)

Gets both the http status code and description from just a code

Parameters

- **status_code** – integer or string that can be converted to an integer

- **default_reason** – default text to be appended to the status_code if the lookup does not find a result

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

Return type str

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

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

Get a bound method of the given object by name.

Parameters

- **obj** – Object on which to look up the method.

- **method_name** – Name of the method to retrieve.

Returns Bound method, or `None` if the method does not exist on the object.

Raises `AttributeError` – The method exists, but it isn't bound (most likely a class was passed, rather than an instance of that class).

class falcon.**TimezoneGMT**

GMT timezone class implementing the `datetime.tzinfo` interface.

dst(*dt*)

Return the daylight saving time (DST) adjustment.

Parameters dt (`datetime.datetime`) – Ignored

Returns DST adjustment for GMT, which is always 0.

Return type datetime.timedelta

tzname(*dt*)

Get the name of this timezone.

Parameters dt (`datetime.datetime`) – Ignored

Returns "GMT"

Return type str

utcoffset(*dt*)

Get the offset from UTC.

Parameters dt (`datetime.datetime`) – Ignored

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

Return type datetime.timedelta

6.3.11 Testing

Testing utilities.

This package contains various test classes and utility functions to support functional testing for both Falcon-based apps and the Falcon framework itself. Both unittest-style and pytest-style tests are supported:

```python
# ----------------------------------------------------------------
# unittest-style
# ----------------------------------------------------------------

from falcon import testing
import myapp
```

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

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


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

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


# ----------------------------------------------------------------------
# pytest-style
# ----------------------------------------------------------------------

from falcon import testing
import pytest

import myapp


@pytest.fixture(scope='module')
def client():
    # Assume the hypothetical `myapp` package has a
    # function called `create()` to initialize and
    # return a `falcon.API` instance.
    return testing.TestClient(myapp.create())


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

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

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

> Encapsulates the result of a simulated WSGI request.
>
> ### Parameters
>
> - **iterable** (*iterable*) – An iterable that yields zero or more bytestrings, per PEP-3333
>
> - **status** (*str*) – An HTTP status string, including status code and reason string
>
> - **headers** (*list*) – A list of (header_name, header_value) tuples, per PEP-3333
>
> **status**
>
> > *str* – HTTP status string given in the response
>
> **status_code**
>
> > *int* – The code portion of the HTTP status string
>
> **headers**

CaseInsensitiveDict – A case-insensitive dictionary containing all the headers in the response, except for cookies, which may be accessed via the *cookies* attribute.

Note: Multiple instances of a header in the response are currently not supported; it is unspecified which value will "win" and be represented in *headers*.

---

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

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

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

text
: *str* – Decoded response body of type `unicode` under Python 2.6 and 2.7, and of type `str` otherwise. If the content type does not specify an encoding, UTF-8 is assumed.

json
: *dict* – Deserialized JSON body. Raises an error if the response is not JSON.

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

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

    name
    : *str* – The cookie's name.

    value
    : *str* – The value of the cookie.

    expires
    : *datetime.datetime* – Expiration timestamp for the cookie, or `None` if not specified.

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

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

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

    secure
    : *bool* – Whether or not the cookie may only only be transmitted from the client via HTTPS.

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

`falcon.testing.``simulate_request`(*app*, *method='GET'*, *path='/'*, *query_string=None*, *headers=None*, *body=None*, *file_wrapper=None*, *params=None*, *params_csv=True*)
: Simulates a request to a WSGI application.

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

    Keyword Arguments

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

- **method** (*str*) – An HTTP method to use in the request (default: 'GET')
- **path** (*str*) – The URL path to request (default: '/')
- **params** (*dict*) – A dictionary of query string parameters, where each key is a parameter name, and each value is either a `str` or something that can be converted into a `str`, or a list of such values. If a `list`, the value will be converted to a comma-delimited string of values (e.g., 'thing=1,2,3').
- **params_csv** (*bool*) – Set to `False` to encode list values in query string params by specifying multiple instances of the parameter (e.g., 'thing=1&thing=2&thing=3'). Otherwise, parameters will be encoded as comma-separated values (e.g., 'thing=1,2,3'). Defaults to `True`.
- **query_string** (*str*) – A raw query string to include in the request (default: `None`). If specified, overrides *params*.
- **headers** (*dict*) – Additional headers to include in the request (default: `None`)
- **body** (*str*) – A string to send as the body of the request. Accepts both byte strings and Unicode strings (default: `None`). If a Unicode string is provided, it will be encoded as UTF-8 in the request.
- **file_wrapper** (*callable*) – Callable that returns an iterable, to be used as the value for *wsgi.file_wrapper* in the environ (default: `None`). This can be used to test high-performance file transmission when *resp.stream* is set to a file-like object.

> **Returns** The result of the request
>
> **Return type** *Result*

falcon.testing.**simulate_get**(*app*, *path*, ***kwargs*)
Simulates a GET request to a WSGI application.

Equivalent to:

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

Parameters

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

Keyword Arguments

- **params** (*dict*) – A dictionary of query string parameters, where each key is a parameter name, and each value is either a `str` or something that can be converted into a `str`, or a list of such values. If a `list`, the value will be converted to a comma-delimited string of values (e.g., 'thing=1,2,3').
- **params_csv** (*bool*) – Set to `False` to encode list values in query string params by specifying multiple instances of the parameter (e.g., 'thing=1&thing=2&thing=3'). Otherwise, parameters will be encoded as comma-separated values (e.g., 'thing=1,2,3'). Defaults to `True`.
- **query_string** (*str*) – A raw query string to include in the request (default: `None`). If specified, overrides *params*.
- **headers** (*dict*) – Additional headers to include in the request (default: `None`)

- **file_wrapper** (*callable*) – Callable that returns an iterable, to be used as the value for *wsgi.file_wrapper* in the environ (default: `None`). This can be used to test high-performance file transmission when *resp.stream* is set to a file-like object.

falcon.testing.**simulate_head**(*app*, *path*, ***kwargs*)
    Simulates a HEAD request to a WSGI application.

    Equivalent to:

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

    Parameters

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

    Keyword Arguments

- **params** (*dict*) – A dictionary of query string parameters, where each key is a parameter name, and each value is either a `str` or something that can be converted into a `str`, or a list of such values. If a `list`, the value will be converted to a comma-delimited string of values (e.g., 'thing=1,2,3').
- **params_csv** (*bool*) – Set to `False` to encode list values in query string params by specifying multiple instances of the parameter (e.g., 'thing=1&thing=2&thing=3'). Otherwise, parameters will be encoded as comma-separated values (e.g., 'thing=1,2,3'). Defaults to `True`.
- **query_string** (*str*) – A raw query string to include in the request (default: `None`). If specified, overrides *params*.
- **headers** (*dict*) – Additional headers to include in the request (default: `None`)

falcon.testing.**simulate_post**(*app*, *path*, ***kwargs*)
    Simulates a POST request to a WSGI application.

    Equivalent to:

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

    Parameters

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

    Keyword Arguments

- **params** (*dict*) – A dictionary of query string parameters, where each key is a parameter name, and each value is either a `str` or something that can be converted into a `str`, or a list of such values. If a `list`, the value will be converted to a comma-delimited string of values (e.g., 'thing=1,2,3').
- **params_csv** (*bool*) – Set to `False` to encode list values in query string params by specifying multiple instances of the parameter (e.g., 'thing=1&thing=2&thing=3'). Otherwise, parameters will be encoded as comma-separated values (e.g., 'thing=1,2,3'). Defaults to `True`.
- **headers** (*dict*) – Additional headers to include in the request (default: `None`)

- **body** (*str*) – A string to send as the body of the request. Accepts both byte strings and Unicode strings (default: `None`). If a Unicode string is provided, it will be encoded as UTF-8 in the request.

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

> Simulates a PUT request to a WSGI application.
>
> Equivalent to:

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

> **Parameters**
>
> - **app** (`callable`) – The WSGI application to call
> - **path** (`str`) – The URL path to request
>
> **Keyword Arguments**
>
> - **params** (*dict*) – A dictionary of query string parameters, where each key is a parameter name, and each value is either a `str` or something that can be converted into a `str`, or a list of such values. If a `list`, the value will be converted to a comma-delimited string of values (e.g., 'thing=1,2,3').
> - **params_csv** (*bool*) – Set to `False` to encode list values in query string params by specifying multiple instances of the parameter (e.g., 'thing=1&thing=2&thing=3'). Otherwise, parameters will be encoded as comma-separated values (e.g., 'thing=1,2,3'). Defaults to `True`.
> - **headers** (*dict*) – Additional headers to include in the request (default: `None`)
> - **body** (*str*) – A string to send as the body of the request. Accepts both byte strings and Unicode strings (default: `None`). If a Unicode string is provided, it will be encoded as UTF-8 in the request.

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

> Simulates an OPTIONS request to a WSGI application.
>
> Equivalent to:

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

> **Parameters**
>
> - **app** (`callable`) – The WSGI application to call
> - **path** (`str`) – The URL path to request
>
> **Keyword Arguments**
>
> - **params** (*dict*) – A dictionary of query string parameters, where each key is a parameter name, and each value is either a `str` or something that can be converted into a `str`, or a list of such values. If a `list`, the value will be converted to a comma-delimited string of values (e.g., 'thing=1,2,3').
> - **params_csv** (*bool*) – Set to `False` to encode list values in query string params by specifying multiple instances of the parameter (e.g., 'thing=1&thing=2&thing=3'). Otherwise, parameters will be encoded as comma-separated values (e.g., 'thing=1,2,3'). Defaults to `True`.
> - **headers** (*dict*) – Additional headers to include in the request (default: `None`)

`falcon.testing.``**simulate_patch**``(`*app*, *path*, *\*\*kwargs*`)`
Simulates a PATCH request to a WSGI application.

Equivalent to:

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

> **Parameters**
> - **app** (`callable`) – The WSGI application to call
> - **path** (`str`) – The URL path to request
>
> **Keyword Arguments**
> - **params** (*dict*) – A dictionary of query string parameters, where each key is a parameter name, and each value is either a `str` or something that can be converted into a `str`, or a list of such values. If a `list`, the value will be converted to a comma-delimited string of values (e.g., 'thing=1,2,3').
> - **params_csv** (*bool*) – Set to `False` to encode list values in query string params by specifying multiple instances of the parameter (e.g., 'thing=1&thing=2&thing=3'). Otherwise, parameters will be encoded as comma-separated values (e.g., 'thing=1,2,3'). Defaults to `True`.
> - **headers** (*dict*) – Additional headers to include in the request (default: `None`)
> - **body** (*str*) – A string to send as the body of the request. Accepts both byte strings and Unicode strings (default: `None`). If a Unicode string is provided, it will be encoded as UTF-8 in the request.

`falcon.testing.``**simulate_delete**``(`*app*, *path*, *\*\*kwargs*`)`
Simulates a DELETE request to a WSGI application.

Equivalent to:

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

> **Parameters**
> - **app** (`callable`) – The WSGI application to call
> - **path** (`str`) – The URL path to request
>
> **Keyword Arguments**
> - **params** (*dict*) – A dictionary of query string parameters, where each key is a parameter name, and each value is either a `str` or something that can be converted into a `str`, or a list of such values. If a `list`, the value will be converted to a comma-delimited string of values (e.g., 'thing=1,2,3').
> - **params_csv** (*bool*) – Set to `False` to encode list values in query string params by specifying multiple instances of the parameter (e.g., 'thing=1&thing=2&thing=3'). Otherwise, parameters will be encoded as comma-separated values (e.g., 'thing=1,2,3'). Defaults to `True`.
> - **headers** (*dict*) – Additional headers to include in the request (default: `None`)

class `falcon.testing.``**TestClient**``(`*app*`)`
Simulates requests to a WSGI application.

This class provides a contextual wrapper for Falcon's *simulate_** test functions. It lets you replace this:

```
simulate_get(app, '/messages')
simulate_head(app, '/messages')
```

with this:

```
client = TestClient(app)
client.simulate_get('/messages')
client.simulate_head('/messages')
```

> **Parameters app** (*callable*) – A WSGI application to target when simulating requests

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

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

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

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

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

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

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

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

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

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

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

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

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

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

simulate_request (**args, **kwargs*)
 Simulates a request to a WSGI application.

> Wraps *falcon.testing.simulate_request()* to perform a WSGI request directly against
> self.app. Equivalent to:
>
> ```
> falcon.testing.simulate_request(self.app, *args, **kwargs)
> ```

class falcon.testing.**TestCase** (*methodName='runTest'*)
 Extends unittest to support WSGI functional testing.

Note: If available, uses testtools in lieu of unittest.

This base class provides some extra plumbing for unittest-style test cases, to help simulate WSGI calls without having to spin up an actual web server. Various simulation methods are derived from *falcon.testing.TestClient*.

Simply inherit from this class in your test case classes instead of `unittest.TestCase` or `testtools.TestCase`.

app
> *object* – A WSGI application to target when simulating requests (default: `falcon.API()`). When testing your application, you will need to set this to your own instance of `falcon.API`. For example:

```python
from falcon import testing
import myapp


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

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


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

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

api
> *object* – Deprecated alias for `app`

api_class
> *callable* – Deprecated class variable; will be removed in a future release.

class falcon.testing.**SimpleTestResource**(*status=None*, *body=None*, *json=None*, *headers=None*)
Mock resource for functional testing of framework components.

This class implements a simple test resource that can be extended as needed to test middleware, hooks, and the Falcon framework itself.

Only noop `on_get()` and `on_post()` responders are implemented; when overriding these, or adding additional responders in child classes, they can be decorated with the *falcon.testing.capture_responder_args()* hook in order to capture the *req*, *resp*, and *params* arguments that are passed to the responder. Responders may also be decorated with the `falcon.testing.set_resp_defaults()` hook in order to set *resp* properties to default *status*, *body*, and *header* values.

> **Keyword Arguments**
> - **status** (*str*) – Default status string to use in responses
> - **body** (*str*) – Default body string to use in responses
> - **json** (*dict*) – Default JSON document to use in responses. Will be serialized to a string and encoded as UTF-8. Either *json* or *body* may be specified, but not both.
> - **headers** (*dict*) – Default set of additional headers to include in responses

captured_req
> *falcon.Request* – The last Request object passed into any one of the responder methods.

captured_resp
> *falcon.Response* – The last Response object passed into any one of the responder methods.

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

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

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

> • captured_req

> • captured_resp

> • captured_kwargs

`falcon.testing.`**`rand_string`**(*min*, *max*)
> Returns a randomly-generated string, of a random length.

> > **Parameters**

> > • **min** (*int*) – Minimum string length to return, inclusive

> > • **max** (*int*) – Maximum string length to return, inclusive

`falcon.testing.`**`create_environ`**(*path='/'*, *query_string=''*, *protocol='HTTP/1.1'*, *scheme='http'*, *host='falconframework.org'*, *port=None*, *headers=None*, *app=''*, *body=''*, *method='GET'*, *wsgierrors=None*, *file_wrapper=None*)
> Creates a mock PEP-3333 environ `dict` for simulating WSGI requests.

> > **Keyword Arguments**

> > • **path** (*str*) – The path for the request (default '/')

> > • **query_string** (*str*) – The query string to simulate, without a leading '?' (default '')

> > • **protocol** (*str*) – The HTTP protocol to simulate (default 'HTTP/1.1'). If set to 'HTTP/1.0', the Host header will not be added to the environment.

> > • **scheme** (*str*) – URL scheme, either 'http' or 'https' (default 'http')

> > • **host** (*str*) – Hostname for the request (default 'falconframework.org')

> > • **port** (*str*) – The TCP port to simulate. Defaults to the standard port used by the given scheme (i.e., 80 for 'http' and 443 for 'https').

> > • **headers** (*dict*) – Headers as a `dict` or an iterable yielding (*key*, *value*) `tuple`'s

- **app** (*str*) – Value for the `SCRIPT_NAME` environ variable, described in PEP-333: 'The initial portion of the request URL's "path" that corresponds to the application object, so that the application knows its virtual "location". This may be an empty string, if the application corresponds to the "root" of the server.' (default '')

- **body** (*str*) – The body of the request (default ''). Accepts both byte strings and Unicode strings. Unicode strings are encoded as UTF-8 in the request.

- **method** (*str*) – The HTTP method to use (default 'GET')

- **wsgierrors** (*io*) – The stream to use as *wsgierrors* (default `sys.stderr`)

- **file_wrapper** – Callable that returns an iterable, to be used as the value for *wsgi.file_wrapper* in the environ.

Deprecated

class `falcon.testing.`**`TestBase`**(*methodName='runTest'*)
   Extends `unittest` to support WSGI functional testing.

> **Warning:** This class has been deprecated and will be removed in a future release. Please use [`TestCase`](#)
> instead.

> **Note:** If available, uses `testtools` in lieu of `unittest`.

This base class provides some extra plumbing for unittest-style test cases, to help simulate WSGI calls without having to spin up an actual web server. Simply inherit from this class in your test case classes instead of `unittest.TestCase` or `testtools.TestCase`.

`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.

`api_class`
   alias of `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*.

> **srmock_class**
>> alias of *StartResponseMock*

> **tearDown**()
>> Destructor, unittest-style

class falcon.testing.**TestResource**
> Mock resource for functional testing.

> > **Warning:** This class is deprecated and will be removed in a future release. Please use *SimpleTestResource* instead.

> 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.

6.4 Changelogs

6.4.1 Changelog for Falcon 1.1.0

Breaking Changes

(None)

New & Improved

- A new *bounded_stream* property was added to `falcon.Request` that can be used in place of the *stream* property to mitigate the blocking behavior of input objects used by some WSGI servers.

- A new *uri_template* property was added to `Request` to expose the template for the route corresponding to the path requested by the user agent.

- A *context* property was added to `Response` to mirror the same property that is already available for `Request`.

- JSON-encoded query parameter values can now be retrieved and decoded in a single step via `get_param_as_dict()`.

- CSV-style parsing of query parameter values can now be disabled.

- `get_param_as_bool()` now recognizes "on" and "off" in support of IE's default checkbox values.

- An *accept_ranges* property was added to `Response` to facilitate setting the Accept-Ranges header.

- Added the `HTTPUriTooLong` and `HTTPGone` error classes.

- When a title is not specified for `HTTPError`, it now defaults to the HTTP status text.

- All parameters are now optional for most error classes.

- Cookie-related documentation has been clarified and expanded

- The `falcon.testing.Cookie` class was added to represent a cookie returned by a simulated request. `falcon.testing.Result` now exposes a *cookies* attribute for examining returned cookies.

- pytest support was added to Falcon's testing framework. Apps can now choose to either write unittest- or pytest-style tests.

- The test runner for Falcon's own tests was switched from nose to pytest.

- When simulating a request using Falcon's testing framework, query string parameters can now be specified as a `dict`, as an alternative to passing a raw query string.

- A flag is now passed to the *process_request* middleware method to signal whether or not an exception was raised while processing the request. A shim was added to avoid breaking existing middleware methods that do not yet accept this new parameter.

- A new CLI utility, *falcon-print-routes*, was added that takes in a module:callable, introspects the routes, and prints the results to stdout. This utility is automatically installed along with the framework:

```
$ falcon-print-routes commissaire:api
-> /api/v0/status
-> /api/v0/cluster/{name}
-> /api/v0/cluster/{name}/hosts
-> /api/v0/cluster/{name}/hosts/{address}
```

- Custom attributes can now be attached to instances of `Request` and `Response`. This can be used as an alternative to adding values to the *context* property, or implementing custom subclasses.

- `get_http_status()` was implemented to provide a way to look up a full HTTP status line, given just a status code.

Fixed

- When `auto_parse_form_urlencoded` is set to `True`, the framework now checks the HTTP method before attempting to consume and parse the body.

- Before attempting to read the body of a form-encoded request, the framework now checks the Content-Length header to ensure that a non-empty body is expected. This helps prevent bad requests from causing a blocking read when running behind certain WSGI servers.

- When the requested method is not implemented for the target resource, the framework now raises `HTTPMethodNotAllowed`, rather than modifying the `Request` object directly. This improves visibility for custom error handlers and for middleware methods.

- Error class docstrings have been updated to reflect the latest RFCs.

- When an error is raised by a resource method or a hook, the error will now always be processed (including setting the appropriate properties of the `Response` object) before middleware methods are called.

- A case was fixed in which middleware processing did not continue when an instance of `HTTPError` or `HTTPStatus` was raised.

- The `encode()` method will now attempt to detect whether the specified string has already been encoded, and return it unchanged if that is the case.

- The default OPTIONS responder now explicitly sets Content-Length to zero in the response.

- `falcon.testing.Result` now assumes that the response body is encoded as UTF-8 when the character set is not specified, rather than raising an error when attempting to decode the response body.

- When simulating requests, Falcon's testing framework now properly tunnels Unicode characters through the WSGI interface.

- `import falcon.uri` now works, in addition to `from falcon import uri`.

- URI template fields are now validated up front, when the route is added, to ensure they are valid Python identifiers. This prevents cryptic errors from being raised later on when requests are routed.

- When running under Python 3, `inspect.signature()` is used instead of `inspect.getargspec()` to provide compatibility with annotated functions.

6.4.2 Changelog for Falcon 1.0.0

Breaking Changes

- The deprecated global hooks feature has been removed. `API` no longer accepts *before* and *after* kwargs. Applications can work around this by migrating any logic contained in global hooks to reside in middleware components instead.

- The middleware method `process_resource()` must now accept an additional *params* argument. This gives the middleware method an opportunity to interact with the values for any fields defined in a route's URI template.

- The middleware method `process_resource()` is now skipped when no route is found for the incoming request. This avoids having to include an `if resource is not None` check when implementing this method. A sink may be used instead to execute logic in the case that no route is found.

- An option was added to toggle automatic parsing of form params. Falcon will no longer automatically parse, by default, requests that have the content type "application/x-www-form-urlencoded". This was done to avoid unintended side-effects that may arise from consuming the request stream. It also makes it more straightforward for applications to customize and extend the handling of form submissions. Applications that require this functionality must re-enable it explicitly, by setting a new request option that was added for that purpose, per the example below:

```
app = falcon.API()
app.req_options.auto_parse_form_urlencoded = True
```

- The `HTTPUnauthorized` initializer now requires an additional argument, *challenges*. Per RFC 7235, a server returning a 401 must include a WWW-Authenticate header field containing at least one challenge.

- The performance of composing the response body was improved. As part of this work, the `Response.body_encoded` attribute was removed. This property was only intended to be used by the framework itself, but any dependent code can be migrated per the example below:

```python
# Before
body = resp.body_encoded

# After
if resp.body:
    body = resp.body.encode('utf-8')
else:
    body = b''
```

New & Improved

- A code of conduct was added to solidify our community's commitment to sustaining a welcoming, respectful culture.

- CPython 3.5 is now fully supported.

- The constants HTTP_422, HTTP_428, HTTP_429, HTTP_431, HTTP_451, and HTTP_511 were added.

- The `HTTPUnprocessableEntity`, `HTTPTooManyRequests`, and `HTTPUnavailableForLegalReasons` error classes were added.

- The `HTTPStatus` class is now available directly under the *falcon* module, and has been properly documented.

- Support for HTTP redirections was added via a set of `HTTPStatus` subclasses. This should avoid the problem of hooks and responder methods possibly overriding the redirect. Raising an instance of one of these new redirection classes will short-circuit request processing, similar to raising an instance of `HTTPError`.

- The default 404 responder now raises an instance of `HTTPError` instead of manipulating the response object directly. This makes it possible to customize the response body using a custom error handler or serializer.

- A new method, `get_header()`, was added to `Response`. Previously there was no way to check if a header had been set. The new `get_header()` method facilitates this and other use cases.

- `falcon.Request.client_accepts_msgpack()` now recognizes "application/msgpack", in addition to "application/x-msgpack".

- New `access_route` and `remote_addr` properties were added to `Request` for getting upstream IP addresses.

- `Request` and `Response` now support range units other than bytes.

- The `API` and `StartResponseMock` class types can now be customized by inheriting from `TestBase` and overriding the *api_class* and *srmock_class* class attributes.

- Path segments with multiple field expressions may now be defined at the same level as path segments having only a single field expression. For example:

```python
api.add_route('/files/{file_id}', resource_1)
api.add_route('/files/{file_id}.{ext}', resource_2)
```

- Support was added to `API.add_route()` for passing through additional args and kwargs to custom routers.

- Digits and the underscore character are now allowed in the `falcon.routing.compile_uri_template()` helper, for use in custom router implementations.

- A new testing framework was added that should be more intuitive to use than the old one. Several of Falcon's own tests were ported to use the new framework (the remainder to be ported in a subsequent release.) The new testing framework performs wsgiref validation on all requests.

- The performance of setting `Response.content_range` was improved by ~50%.

- A new param, *obs_date*, was added to `falcon.Request.get_header_as_datetime()`, and defaults to `False`. This improves the method's performance when obsolete date formats do not need to be supported.

Fixed

- Field expressions at a given level in the routing tree no longer mask alternative branches. When a single segment in a requested path can match more than one node at that branch in the routing tree, and the first branch taken happens to be the wrong one (i.e., the subsequent nodes do not match, but they would have under a different branch), the other branches that could result in a successful resolution of the requested path will now be subsequently tried, whereas previously the framework would behave as if no route could be found.

- The user agent is now instructed to expire the cookie when it is cleared via `unset_cookie()`.

- Support was added for hooks that have been defined via `functools.partial()`.

- Tunneled UTF-8 characters in the request path are now properly decoded, and a placeholder character is substituted for any invalid code points.

- The instantiation of `Request.context_type` is now delayed until after all other properties of the `Request` class have been initialized, in case the context type's own initialization depends on any of `Request`'s properties.

- A case was fixed in which reading from `Request.stream` could hang when using `wsgiref` to host the app.

- The default error serializer now sets the Vary header in responses. Implementing this required passing the `Response` object to the serializer, which would normally be a breaking change. However, the framework was modified to detect old-style error serializers and wrap them with a shim to make them compatible with the new interface.

- A query string containing malformed percent-encoding no longer causes the framework to raise an error.

- Additional tests were added for a few lines of code that were previously not covered, due to deficiencies in code coverage reporting that have since been corrected.

- The Cython note is no longer displayed when installing under Jython.

- Several errors and ambiguities in the documentation were corrected.

6.4.3 Changelog for Falcon 0.3.0

Breaking Changes

- Date headers are now returned as `datetime.datetime` objects instead of strings.

New & Improved

- This release includes a new router architecture for improved performance and flexibility.

- A custom router can now be specified when instantiating the `API` class.

- URI templates can now include multiple parameterized fields within a single path segment.

- Falcon now supports reading and writing cookies.

- Falcon now supports Jython 2.7.

- A method for getting a query param as a date was added to the `Request` class.

- Date headers are now returned as `datetime.datetime` objects.

- A default value can now be specified when calling `Request.get_param()`. This provides an alternative to using the pattern:

```
value = req.get_param(name) or default_value
```

- Friendly constants for status codes were added (e.g., `falcon.HTTP_NO_CONTENT` vs. `falcon.HTTP_204`.)

- Several minor performance optimizations were made to the code base.

Fixed

- The query string parser was modified to improve handling of percent-encoded data.

- Several errors in the documentation were corrected.

- The `six` package was pinned to 1.4.0 or better. `six.PY2` is required by Falcon, but that wasn't added to `six` until version 1.4.0.

6.4.4 Changelog for Falcon 0.2.0

Breaking Changes

- The deprecated util.misc.percent_escape and util.misc.percent_unescape functions were removed. Please use the functions in the util.uri module instead.

- The deprecated function, API.set_default_route, was removed. Please use sinks instead.

- HTTPRangeNotSatisfiable no longer accepts a media_type parameter.

- When using the comma-delimited list convention, req.get_param_as_list(...) will no longer insert placeholders, using the None type, for empty elements. For example, where previously the query string "foo=1,,3" would result in ['1', None, '3'], it will now result in ['1', '3'].

New & Improved

- Since 0.1 we've added proper RTD docs to make it easier for everyone to get started with the framework. Over time we will continue adding content, and we would love your help!

- Falcon now supports "wsgi.filewrapper". You can assign any file-like object to resp.stream and Falcon will use "wsgi.filewrapper" to more efficiently pipe the data to the WSGI server.

- Support was added for automatically parsing requests containing "application/x-www-form-urlencoded" content. Form fields are now folded into req.params.

- Custom Request and Response classes are now supported. You can specify custom types when instantiating falcon.API.

- A new middleware feature was added to the framework. Middleware deprecates global hooks, and we encourage everyone to migrate as soon as possible.

- A general-purpose dict attribute was added to Request. Middleware, hooks, and responders can now use req.context to share contextual information about the current request.

- A new method, append_header, was added to falcon.API to allow setting multiple values for the same header using comma separation. Note that this will not work for setting cookies, but we plan to address this in the next release (0.3).

- A new "resource" attribute was added to hooks. Old hooks that do not accept this new attribute are shimmed so that they will continue to function. While we have worked hard to minimize the performance impact, we recommend migrating to the new function signature to avoid any overhead.

- Error response bodies now support XML in addition to JSON. In addition, the HTTPError serialization code was refactored to make it easier to implement a custom error serializer.

- A new method, "set_error_serializer" was added to falcon.API. You can use this method to override Falcon's default HTTPError serializer if you need to support custom media types.

- Falcon's testing base class, testing.TestBase was improved to facilitate Py3k testing. Notably, TestBase.simulate_request now takes an additional "decode" kwarg that can be used to automatically decode bytestring PEP-3333 response bodies.

- An "add_link" method was added to the Response class. Apps can use this method to add one or more Link header values to a response.

- Added two new properties, req.host and req.subdomain, to make it easier to get at the hostname info in the request.

- Allow a wider variety of characters to be used in query string params.

- Internal APIs have been refactored to allow overriding the default routing mechanism. Further modularization is planned for the next release (0.3).

- Changed req.get_param so that it behaves the same whether a list was specified in the query string using the HTML form style (in which each element is listed in a separate 'key=val' field) or in the more compact API style (in which each element is comma-separated and assigned to a single param instance, as in 'key=val1,val2,val3')

- Added a convenience method, set_stream(...), to the Response class for setting the stream and its length at the same time, which should help people not forget to set both (and save a few keystrokes along the way).

- Added several new error classes, including HTTPRequestEntityTooLarge, HTTPInvalidParam, HTTPMissingParam, HTTPInvalidHeader and HTTPMissingHeader.

- Python 3.4 is now fully supported.

- Various minor performance improvements

Fixed

- Ensure 100% test coverage and fix any bugs identified in the process.

- Fix not recognizing the "bytes=" prefix in Range headers.

- Make HTTPNotFound and HTTPMethodNotAllowed fully compliant, according to RFC 7231.

- Fixed the default on_options responder causing a Cython type error.

- URI template strings can now be of type unicode under Python 2.

- When SCRIPT_NAME is not present in the WSGI environ, return an empty string for the req.app property.

- Global "after" hooks will now be executed even when a responder raises an error.

- Fixed several minor issues regarding testing.create_environ(...)

- Work around a wsgiref quirk, where if no content-length header is submitted by the client, wsgiref will set the value of that header to an empty string in the WSGI environ.

- Resolved an issue causing several source files to not be Cythonized.

- Docstrings have been edited for clarity and correctness.

f

A

B

C

D

E