Uplink Documentation

Release 0.7.0

Raj Kumar

Jan 07, 2019

Contents

A Declarative HTTP Client for Python. Inspired by Retrofit.

Note: Uplink is in beta development. The public API is still evolving, but we expect most changes to be backwards compatible at this point.

Uplink turns your HTTP API into a Python class.

```
from uplink import Consumer, get, headers, Path, Query

class GitHub(Consumer):
    """A Python Client for the GitHub API."""

    @get("users/{user}/repos")
    def get_repos(self, user: Path, sort_by: Query("sort")):
        """Get user's public repositories."""
```

Build an instance to interact with the webservice.

```
github = GitHub(base_url="https://api.github.com/")
```

Then, executing an HTTP request is as simply as invoking a method.

```
repos = github.get_repos(user="octocat", sort_by="created")
```

The returned object is a friendly `requests.Response`:

```
print(repos.json())
# Output: [{'id': 64778136, 'name': 'linguist', ...
```

For sending non-blocking requests, Uplink comes with support for `aiohttp` and `twisted` (example).

CHAPTER 1

Features

- **Quickly Define Structured API Clients**
  - Use decorators and type hints to describe each HTTP request
  - JSON, URL-encoded, and multipart request body and file upload
  - URL parameter replacement, request headers, and query parameter support
- **Bring Your Own HTTP Library**
  - Non-blocking I/O support for Aiohttp and Twisted
  - *Supply your own session* (e.g., `requests.Session`) for greater control
- **Easy and Transparent Deserialization/Serialization**
  - Define *custom converters* for your own objects
  - Support for `marshmallow` schemas and *handling collections* (e.g., list of Users)
- **Extendable**
  - Install optional plugins for additional features (e.g., protobuf support)
  - Compose *custom response and error handling* functions as middleware
- **Authentication**
  - Built-in support for *Basic Authentication*
  - Use existing auth libraries for supported clients (e.g., `requests-oauthlib`)

Uplink officially supports Python 2.7 & 3.3-3.7.

CHAPTER 2

User Testimonials

Michael Kennedy (@mkennedy), host of Talk Python and Python Bytes podcasts-

> Of course our first reaction when consuming HTTP resources in Python is to reach for Requests. But for *structured* APIs, we often want more than ad-hoc calls to Requests. We want a client-side API for our apps. Uplink is the quickest and simplest way to build just that client-side API. Highly recommended.

Or Carmi (@liiight), notifiers maintainer-

> Uplink's intelligent usage of decorators and typing leverages the most pythonic features in an elegant and dynamic way. If you need to create an API abstraction layer, there is really no reason to look elsewhere.

CHAPTER 3

User Manual

Follow this guide to get up and running with Uplink.

3.1 Installation

3.1.1 Using pip

With **`pip`** (or **`pipenv`**), you can install Uplink simply by typing:

```
$ pip install -U uplink
```

3.1.2 Download the Source Code

Uplink's source code is in a public repository hosted on GitHub.

As an alternative to installing with **`pip`**, you could clone the repository,

```
$ git clone https://github.com/prkumar/uplink.git
```

then, install; e.g., with `setup.py`:

```
$ cd uplink
$ python setup.py install
```

3.1.3 Extras

These are optional integrations and features that extend the library's core functionality and typically require an additional dependency.

When installing Uplink with `pip`, you can specify any of the following extras, to add their respective dependencies to your installation:

Extra	Description
`aiohttp`	Enables *`uplink.AiohttpClient`*, for sending non-blocking requests and receiving awaitable responses.
`marshmallow`	Enables `uplink.MarshmallowConverter`, for converting JSON responses directly into Python objects using `marshmallow.Schema`.
`twisted`	Enables *`uplink.TwistedClient`*, for sending non-blocking requests and receiving `Deferred` responses.

To download all available features, run

```
$ pip install -U uplink[aiohttp, marshmallow, twisted]
```

3.2 Quickstart

Ready to write your first API client with Uplink? This guide will walk you through what you'll need to know to get started.

First, make sure you've *installed (or updated) Uplink*:

```
$ pip install -U uplink
```

3.2.1 Defining an API Client

Writing a **structured** API client with Uplink is very simple.

To start, create a subclass of *`Consumer`*. For example, here's the beginning of our GitHub client (we'll add some methods to this class soon):

```
from uplink import Consumer

class GitHub(Consumer):
    ...
```

When creating an instance of this consumer, we can use the `base_url` constructor argument to identify the target service. In our case, it's GitHub's public API:

```
github = GitHub(base_url="https://api.github.com/")
```

Note: `base_url` is especially useful for creating clients that target separate services with similar APIs; for example, we could use this GitHub consumer to also create clients for any GitHub Enterprise instance for projects hosted outside of the public GitHub.com service. Another example is creating separate clients for a company's production and staging environments, which are typically hosted on separate domains but expose the same API.

So far, this class looks like any other Python class. The real magic happens when you define methods to interact with the webservice using Uplink's HTTP method decorators, which we cover next.

3.2.2 Making a Request

With Uplink, making a request to a webservice is as simple as invoking a method.

Any method of a `Consumer` subclass can be decorated with one of Uplink's HTTP method decorators: `@get`, `@post`, `@put`, `@patch`, `@head`, and `@delete`:

```
class GitHub(Consumer):
    @get("repositories")
    def get_repos(self):
        """List all public repositories."""
```

As shown above, the method's body can be left empty.

The decorator's first argument is the resource endpoint (this is the relative URL path from `base_url`, which we covered above):

```
@get("repositories")
```

You can also specify query parameters:

```
@get("repositories?since=364")
```

Finally, invoke the method to send a request:

```
>>> github = GitHub(base_url="https://api.github.com/")
>>> github.get_repos()
<Response [200]>
>>> _.url
https://api.github.com/repositories
```

By default, uplink uses Requests, so the response we get back from GitHub is wrapped inside a `requests.Response` instance. (If you want, you can *swap out* Requests for a different backing HTTP client, such as *aiohttp*.)

3.2.3 URL Manipulation

Resource endpoints can include URI template parameters that depend on method arguments. A simple URI parameter is an alphanumeric string surrounded by `{` and `}`.

To match the parameter with a method argument, either match the argument's name with the alphanumeric string, like so:

```
@get("users/{username}")
def get_user(self, username): pass
```

or use the *`Path`* annotation.

```
@get("users/{username}")
def get_user(self, name: Path("username")): pass
```

`Query` parameters can also be added dynamically by method arguments.

```
@get("users/{username}/repos")
def get_repos(self, username, sort: Query): pass
```

For "catch-all" or complex query parameter combinations, a *`QueryMap`* can be used:

```
@get("users/{username}/repos")
def get_repos(self, username, **options: QueryMap): pass
```

You can set static query parameters for a method using the *`@params`* decorator.

```
@params({"client_id": "my-client", "client_secret": "****"})
@get("users/{username}")
def get_user(self, username): pass
```

@params can be used as a class decorator for query parameters that need to be included with every request:

```
@params({"client_id": "my-client", "client_secret": "****"})
class GitHub(Consumer):
    ...
```

3.2.4 Header Manipulation

You can set static headers for a method using the *@headers* decorator.

```
@headers({
    "Accept": "application/vnd.github.v3.full+json",
    "User-Agent": "Uplink-Sample-App"
})
@get("users/{username}")
def get_user(self, username): pass
```

@headers can be used as a class decorator for headers that need to be added to every request:

```
@headers({
    "Accept": "application/vnd.github.v3.full+json",
    "User-Agent": "Uplink-Sample-App"
})
class GitHub(Consumer):
    ...
```

A request header can depend on the value of a method argument by using the *Header* function parameter annotation:

```
@get("user")
def get_user(self, authorization: Header("Authorization"):
    """Get an authenticated user."""
```

3.2.5 Request Body

The *Body* annotation identifies a method argument as the the HTTP request body:

```
@post("user/repos")
def create_repo(self, repo: Body): pass
```

This annotation works well with the **keyword arguments** parameter (denoted by the ** prefix):

```
@post("user/repos")
def create_repo(self, **repo_info: Body): pass
```

Moreover, this annotation is useful when using supported serialization formats, such as *JSON* and Protocol Buffers. Take a look at *this guide* for more about serialization with Uplink.

3.2.6 Form Encoded, Multipart, and JSON Requests

Methods can also be declared to send form-encoded, multipart, and JSON data.

Form-encoded data is sent when *@form_url_encoded* decorates the method. Each key-value pair is annotated with a *Field* annotation:

```
@form_url_encoded
@patch("user")
def update_user(self, name: Field, email: Field): pass
```

Multipart requests are used when *@multipart* decorates the method. Parts are declared using the *Part* annotation:

```
@multipart
@put("user/photo")
def upload_photo(self, photo: Part, description: Part): pass
```

JSON data is sent when *@json* decorates the method. The *Body* annotation declares the JSON payload:

```
@json
@patch("user")
def update_user(self, **user_info: uplink.Body):
    """Update an authenticated user."""
```

Alternatively, the *Field* annotation declares a JSON field:

```
@json
@patch("user")
def update_user_bio(self, bio: Field):
    """Update the authenticated user's profile bio."""
```

3.2.7 Handling JSON Responses

Many modern public APIs serve JSON responses to their clients.

If your *Consumer* subclass accesses a JSON API, you can decorate any method with *@returns.json* to directly return the JSON response, instead of a response object, when invoked:

```
class GitHub(Consumer):
    @returns.json
    @get("users/{username}")
    def get_user(self, username):
        """Get a single user."""
```

```
>>> github = GitHub("https://api.github.com")
>>> github.get_user("prkumar")
{'login': 'prkumar', 'id': 10181244, ...
```

You can also target a specific field of the JSON response by using the decorator's `key` argument to select the target JSON field name:

```
class GitHub(Consumer):
    @returns.json(key="blog")
    @get("users/{username}")
    def get_blog_url(self, username):
        """Get the user's blog URL."""
```

```
>>> github.get_blog_url("prkumar")
"https://prkumar.io"
```

Note: JSON responses may represent existing Python classes in your application (for example, a `GitHubUser`). Uplink supports this kind of conversion (i.e., deserialization), and we detail this support in *the next guide*.

3.2.8 Persistence Across Requests from a `Consumer`

The `session` property of a *`Consumer`* instance exposes the instance's configuration and allows for the persistence of certain properties across requests sent from that instance.

You can provide default headers and query parameters for requests sent from a consumer instance through its `session` property, like so:

```
class GitHub(Consumer):

    def __init__(self, username, password)
        # Creates the API token for this user
        api_key = create_api_key(username, password)

        # Send the API token as a query parameter with each request.
        self.session.params["api_key"] = api_key

    @get("user/repos")
    def get_user_repos(self, sort_by: Query("sort")):
        """Lists public repositories for the authenticated user."""
```

Headers and query parameters added through the `session` are applied to all requests sent from the consumer instance.

```
github = GitHub("prkumar", "****")

# Both `api_key` and `sort` are sent with the request.
github.get_user_repos(sort_by="created")
```

Notably, in case of conflicts, the method-level headers and parameters override the session-level, but the method-level properties are not persisted across requests.

3.2.9 Response and Error Handling

Sometimes, you need to validate a response before it is returned or even calculate a new return value from the response. Or, you may need to handle errors from the underlying client before they reach your users.

With Uplink, you can address these concerns by registering a callback with one of these decorators: *`@response_handler`* and *`@error_handler`*.

`@response_handler` registers a callback to intercept responses before they are returned (or deserialized):

```
def raise_for_status(response):
    """Checks whether or not the response was successful."""
    if 200 <= response.status_code < 300:
        # Pass through the response.
        return response
```

(continues on next page)

(continued from previous page)

```
    raise UnsuccessfulRequest(response.url)

class GitHub(Consumer):
    @response_handler(raise_for_status)
    @post("user/repo")
    def create_repo(self, name: Field):
        """Create a new repository."""
```

@error_handler registers a callback to handle an exception thrown by the underlying HTTP client (e.g., `requests.Timeout`):

```
def raise_api_error(exc_type, exc_val, exc_tb):
    """Wraps client error with custom API error"""
    raise MyApiError(exc_val)

class GitHub(Consumer):
    @error_handler(raise_api_error)
    @post("user/repo")
    def create_repo(self, name: Field):
        """Create a new repository."""
```

To apply a handler onto all methods of a *Consumer* subclass, you can simply decorate the class itself:

```
@error_handler(raise_api_error)
class GitHub(Consumer):
    ...
```

Notably, the decorators can be stacked on top of one another to chain their behaviors:

```
@response_handler(check_expected_headers)  # Second, check headers
@response_handler(raise_for_status)  # First, check success
class GitHub(Consumer):
    ...
```

Lastly, both decorators support the optional argument `requires_consumer`. When this option is set to `True`, the registered callback should accept a reference to the `Consumer` instance as its leading argument:

```
@error_handler(requires_consumer=True)
def raise_api_error(consumer, exc_type, exc_val, exc_tb):
    """Wraps client error with custom API error"""
    ...

class GitHub(Consumer):
    @raise_api_error
    @post("user/repo")
    def create_repo(self, name: Field):
        """Create a new repository."""
```

3.3 Authentication

This section covers how to do authentication with Uplink.

3.3.1 Basic Authentication

In v0.4, we added the `auth` parameter to the *`uplink.Consumer`* constructor.

Now it's simple to construct a consumer that uses HTTP Basic Authentication with all requests:

```
github = GitHub(BASE_URL, auth=("user", "pass"))
```

3.3.2 Other Authentication

Often, APIs accept credentials as header values (e.g., Bearer tokens) or query parameters. Your request method can handle these types of authentication by simply accepting the user's credentials as an argument:

```
@post("/user")
def update_user(self, access_token: Query, **info: Body):
    """Update the user associated to the given access token."""
```

If several request methods require authentication, you can persist the token through the consumer's *`session`* property:

```
class GitHub(Consumer):

    def __init__(self, access_token):
        self.session.params["access_token"] = access_token
        ...
```

3.3.3 Using Auth Support for Requests and aiohttp

As we work towards Uplink's v1.0 release, improving built-in support for other types of authentication is a continuing goal.

With that said, if Uplink currently doesn't offer a solution for you authentication needs, you can always leverage the available auth support for the underlying HTTP client.

For instance, `requests` offers out-of-the-box support for making requests with HTTP Digest Authentication, which you can leverage like so:

```
from requests.auth import HTTPDigestAuth

client = uplink.RequestsClient(cred=HTTPDigestAuth("user", "pass"))
api = MyApi(BASE_URL, client=client)
```

You can also use other third-party libraries that extend auth support for the underlying client. For instance, you can use requests-oauthlib for doing OAuth with Requests:

```
from requests_oauthlib import OAuth2Session

session = OAuth2Session(...)
api = MyApi(BASE_URL, client=session)
```

3.4 Serialization

Various serialization formats exist for transmitting structured data over the network: JSON is a popular choice amongst many public APIs partly because its human readable, while a more compact format, such as Protocol Buffers, may be more appropriate for a private API used within an organization.

Regardless what serialization format your API uses, Uplink – with a little bit of help – can automatically decode responses and encode request bodies to and from Python objects using the selected format. This neatly abstracts the HTTP layer from your API client, so callers can operate on objects that make sense to your model instead of directly dealing with the underlying protocol.

This document walks you through how to leverage Uplink's serialization support, including integrations for third-party serialization libraries like `marshmallow` and tools for writing custom conversion strategies that fit your unique needs.

3.4.1 Using Marshmallow Schemas

`marshmallow` is a framework-agnostic, object serialization library for Python. Uplink comes with built-in support for Marshmallow; you can integrate your Marshmallow schemas with Uplink for easy JSON (de)serialization.

First, create a `marshmallow.Schema`, declaring any necessary conversions and validations. Here's a simple example:

```
import marshmallow

class RepoSchema(marshmallow.Schema):
    full_name = marshmallow.fields.Str()

    @marshmallow.post_load
    def make_repo(self, data):
        owner, repo_name = data["full_name"].split("/")
        return Repo(owner=owner, name=repo_name)
```

Then, specify the schema using the *`@returns`* decorator:

```
class GitHub(Consumer):
    @returns(RepoSchema(many=True))
    @get("users/{username}/repos")
    def get_repos(self, username):
        """Get the user's public repositories."""
```

Python 3 users can use a return type hint instead:

```
class GitHub(Consumer):
    @get("users/{username}/repos")
    def get_repos(self, username) -> RepoSchema(many=True)
        """Get the user's public repositories."""
```

Your consumer should now return Python objects based on your Marshmallow schema:

```
github = GitHub(base_url="https://api.github.com")
print(github.get_repos("octocat"))
# Output: [Repo(owner="octocat", name="linguist"), ...]
```

For a more complete example of Uplink's `marshmallow` support, check out this example on GitHub.

3.4.2 Serializing Method Arguments

Most method argument annotations like *Field* and *Body* accept a type parameter that specifies the method argument's expected type or schema, for the sake of serialization.

For example, following the marshmallow example from above, we can specify the RepoSchema as the type of a *Body* argument:

```
from uplink import Consumer, Body

class GitHub(Consumer):
    @json
    @post("user/repos")
    def create_repo(self, repo: Body(type=RepoSchema)):
        """Creates a new repository for the authenticated user."""
```

Then, the repo argument should accept instances of Repo, to be serialized appropriately using the RepoSchema with Uplink's marshmallow integration (see *Using Marshmallow Schemas* for the full setup).

```
repo = Repo(name="my_favorite_new_project")
github.create_repo(repo)
```

3.4.3 Custom JSON Conversion

Recognizing JSON's popularity amongst public APIs, Uplink provides some out-of-the-box utilities to adding JSON serialization support for your objects simple.

Deserialization

@returns.json is handy when working with APIs that provide JSON responses. As its leading positional argument, the decorator accepts a class that represents the expected schema of JSON body:

```
class GitHub(Consumer):
    @returns.json(User)
    @get("users/{username}")
    def get_user(self, username): pass
```

Python 3 users can alternatively use a return type hint:

```
class GitHub(Consumer):
    @returns.json
    @get("users/{username}")
    def get_user(self, username) -> User: pass
```

Next, if your objects (e.g., User) are not defined using a library for which Uplink has built-in support (such as marshmallow), you will also need to register a converter that tells Uplink how to convert the HTTP response into your expected return type.

To this end, we can use *@loads.from_json* to define a simple JSON reader for User:

```
from uplink import loads

@loads.from_json(User)
def user_json_reader(user_cls, json):
    return user_cls(json["id"], json["username"])
```

The decorated function, `user_json_reader()`, can then be passed into the `converter` constructor parameter when instantiating a *`uplink.Consumer`* subclass:

```
github = GitHub(base_url=..., converter=user_json_reader)
```

Alternatively, you can add the `@uplink.install` decorator to register the converter function as a default converter, meaning the converter will be included automatically with any consumer instance and doesn't need to be explicitly provided through the `converter` parameter:

```
from uplink import loads, install

 @install
 @loads.from_json(User)
 def user_json_reader(user_cls, json):
     return user_cls(json["id"], json["username"])
```

At last, calling the `GitHub.get_user()` method should now return an instance of our `User` class:

```
github.get_user("octocat")
# Output: [User(id=583231, name="The Octocat"), ...]
```

Serialization

`@json` is a decorator for *`Consumer`* methods that send JSON requests. Using this decorator requires annotating your arguments with either *`Field`* or *`Body`*. Both annotations support an optional `type` argument for the purpose of serialization:

```
from uplink import Consumer, Body

class GitHub(Consumer):
    @json
    @post("user/repos")
    def create_repo(self, user: Body(type=Repo)):
        """Creates a new repository for the authenticated user."""
```

Similar to deserialization case, we must register a converter that tells Uplink how to turn the `Repo` object to JSON, since the class is not defined using a library for which Uplink has built-in support (such as `marshmallow`).

To this end, we can use *`@dumps.to_json`* to define a simple JSON writer for `Repo`:

```
from uplink import dumps

 @dumps.to_json(Repo)
 def repo_json_writer(repo_cls, repo):
     return {"name": repo.name, "private": repo.is_private()}
```

The decorated function, `repo_json_writer()`, can then be passed into the `converter` constructor parameter when instantiating a *`uplink.Consumer`* subclass:

```
github = GitHub(base_url=..., converter=repo_json_writer)
```

Alternatively, you can add the `@uplink.install` decorator to register the converter function as a default converter, meaning the converter will be included automatically with any consumer instance and doesn't need to be explicitly provided through the `converter` parameter:

```
from uplink import loads, install

 @install
 @dumps.to_json(Repo)
 def repo_json_writer(user_cls, json):
     return {"name": repo.name, "private": repo.is_private()}
```

Now, we should be able to invoke the `GitHub.create_repo()` method with an instance of `Repo`:

```
repo = Repo(name="my_new_project", private=True)
github.create_repo(repo)
```

3.4.4 Converting Collections

Data-driven web applications, such as social networks and forums, devise a lot of functionality around large queries on related data. Their APIs normally encode the results of these queries as collections of a common **type**. Examples include a curated feed of **posts** from subscribed accounts, the top **restaurants** in your area, upcoming *tasks** on a checklist, etc.

You can use the other strategies in this section to add serialization support for a specific type, such as a **post** or a **restaurant**. Once added, this support automatically extends to collections of that type, such as sequences and mappings.

For example, consider a hypothetical Task Management API that supports adding tasks to one or more user-created checklists. Here's the JSON array that the API returns when we query pending tasks on a checklist titled "home":

```
[
    {
       "id": 4139
       "name": "Groceries"
       "due_date": "Monday, September 3, 2018 10:00:00 AM PST"
    },
    {
       "id": 4140
       "title": "Laundry"
       "due_date": "Monday, September 3, 2018 2:00:00 PM PST"
    }
]
```

In this example, the common type could be modeled in Python as a `namedtuple`, which we'll name `Task`:

```
Task = collections.namedtuple("Task", ["id", "name", "due_date"])
```

Next, to add JSON deserialization support for this type, we could create a custom converter using the *`@loads.from_json`* decorator, which is a strategy covered in the subsection *Custom JSON Conversion*. For the sake of brevity, I'll omit the implementation here, but you can follow the link above for details.

Notably, Uplink lets us leverage the added support to also handle collections of type `Task`. The `uplink.types` module exposes two collection types, `List` and *`Dict`*, to be used as function return type annotations. In our example, the query for pending tasks returns a list:

```
from uplink import Consumer, returns, get, types

class TaskApi(Consumer):
   @returns.json
```

(continues on next page)

(continued from previous page)

```
    @get("tasks/{checklist}?due=today")
    def get_pending_tasks(self, checklist) -> types.List[Task]
```

If you are a Python 3.5+ user that is already leveraging the `typing` module to support type hints as specified by **PEP 484** and **PEP 526**, you can safely use `typing.List` and `typing.Dict` here instead of the annotations from `uplink.types`:

```
import typing
from uplink import Consumer, returns, get

class TaskApi(Consumer):
    @returns.json
    @get("tasks/{checklist}?due=today")
    def get_pending_tasks(self, checklist) -> typing.List[Task]
```

Now, the consumer can handle these queries with ease:

```
>>> task_api.get_pending_tasks("home")
[Task(id=4139, name='Groceries', due_date='Monday, September 3, 2018 10:00:00 AM PST
↪'),
 Task(id=4140, name='Laundry', due_date='Monday, September 3, 2018 2:00:00 PM PST')]
```

Note that this feature works with any serialization format, not just JSON.

3.4.5 Writing A Custom Converter

Extending Uplink's support for other serialization formats or libraries (e.g., XML, Thrift, Avro) is pretty straightforward.

When adding support for a new serialization library, create a subclass of `converters.Factory`, which defines abstract methods for different serialization scenarios (deserializing the response body, serializing the request body, etc.), and override each relevant method to return a callable that handles the method's corresponding scenario.

For example, a factory that adds support for Python's `pickle` protocol could look like:

```
import pickle

from uplink import converters

class PickleFactory(converters.Factory):
    """Adapter for Python's Pickle protocol."""

    def create_response_body_converter(self, cls, request_definition):
        # Return callable to deserialize response body into Python object.
        return lambda response: pickle.loads(response.content)

    def create_request_body_converter(self, cls, request_definition):
        # Return callable to serialize Python object into bytes.
        return pickle.dumps
```

Then, when instantiating a new consumer, you can supply this implementation through the `converter` constructor argument of any `Consumer` subclass:

```
client = MyApiClient(BASE_URL, converter=PickleFactory())
```

If the added support should apply broadly, you can alternatively decorate your `converters.Factory` subclass with the `@uplink.install` decorator, which ensures that Uplink automatically adds the factory to new instances of any `Consumer` subclass. This way you don't have to explicitly supply the factory each time you instantiate a consumer.

```
from uplink import converters, install

@install
class PickleFactory(converters.Factory):
    ...
```

For a concrete example of extending support for a new serialization format or library with this approach, checkout this Protobuf extension for Uplink.

3.5 Clients

To use a common English metaphor: Uplink stands on the shoulders of giants.

Uplink doesn't implement any code to handle HTTP protocol stuff directly; for that, the library delegates to an actual HTTP client, such as Requests or Aiohttp. Whatever backing client you choose, when a request method on a *`Consumer`* subclass is invoked, Uplink ultimately interacts with the backing library's interface, at minimum to submit requests and read responses.

This section covers the interaction between Uplink and the backing HTTP client library of your choosing, including how to specify your selection.

3.5.1 Swapping Out the Default HTTP Session

By default, Uplink sends requests using the Requests library. You can configure the backing HTTP client object using the `client` parameter of the *`Consumer`* constructor:

```
github = GitHub(BASE_URL, client=...)
```

For example, you can use the `client` parameter to pass in your own Requests session object:

```
session = requests.Session()
session.verify = False
github = GitHub(BASE_URL, client=session)
```

Further, this also applies for session objects from other HTTP client libraries that Uplink supports, such as `aiohttp` (i.e., a custom `ClientSession` works here, as well).

Following the above example, the `client` parameter also accepts an instance of any `requests.Session` subclass. This makes it easy to leverage functionality from third-party Requests extensions, such as requests-oauthlib, with minimal integration overhead:

```
from requests_oauthlib import OAuth2Session

session = OAuth2Session(...)
api = MyApi(BASE_URL, client=session)
```

3.5.2 Synchronous vs. Asynchronous

Notably, Requests blocks while waiting for a response from the server. For non-blocking requests, Uplink comes with built-in (but optional) support for `aiohttp` and `twisted`.

For instance, you can provide the *`AiohttpClient`* when constructing a *`Consumer`* instance:

```
from uplink import AiohttpClient

github = GitHub(BASE_URL, client=AiohttpClient())
```

Checkout this example on GitHub for more.

3.5.3 Handling Exceptions From the Underlying HTTP Client Library

Each *`Consumer`* instance has an *`exceptions`* property that exposes an enum of standard HTTP client exceptions that can be handled:

```
try:
    repo = github.create_repo(name="myproject", auto_init=True)
except github.exceptions.ConnectionError:
    # Handle client socket error:
    ...
```

This approach to handling exceptions decouples your code from the backing HTTP client, improving code reuse and testability.

Here are the HTTP client exceptions that are exposed through this property:

- `BaseClientException`: Base exception for client connection errors.
- `ConnectionError`: A client socket error occurred.
- `ConnectionTimeout`: The request timed out while trying to connect to the remote server.
- `ServerTimeout`: The server did not send any data in the allotted amount of time.
- `SSLError`: An SSL error occurred.
- `InvalidURL`: URL used for fetching is malformed.

Of course, you can also explicitly catch a particular client error from the backing client (e.g., `requests.FileModeWarning`). This may be useful for handling exceptions that are not exposed through the *`Consumer.exceptions`* property, for example:

```
try:
    repo = github.create_repo(name="myproject", auto_init=True)
except aiohttp.ContentTypeError:
    ...
```

Handling Client Exceptions within an `@error_handler`

The *`@error_handler`* decorator registers a callback to deal with exceptions thrown by the backing HTTP client.

To provide the decorated callback a reference to the `Consumer` instance at runtime, set the decorator's optional argument `requires_consumer` to `True`. This enables the error handler to leverage the consumer's *`exceptions`* property:

```
@error_handler(requires_consumer=True)
def raise_api_error(consumer, exc_type, exc_val, exc_tb):
    """Wraps client error with custom API error"""
    if isinstance(exc_val, consumer.exceptions.ServerTimeout):
        # Handle the server timeout specifically:
        ...

class GitHub(Consumer):
    @raise_api_error
    @post("user/repo")
    def create_repo(self, name: Field):
        """Create a new repository."""
```

3.6 Tips & Tricks

Here are a few ways to simplify consumer definitions.

3.6.1 Decorating All Request Methods in a Class

To apply a decorator of this library across all methods of a *uplink.Consumer* subclass, you can simply decorate the class rather than each method individually:

```
@uplink.timeout(60)
class GitHub(uplink.Consumer):
    @uplink.get("/repositories")
    def get_repos(self):
        """Dump every public repository."""

    @uplink.get("/organizations")
    def get_organizations(self):
        """List all organizations."""
```

Hence, the consumer defined above is equivalent to the following, slightly more verbose definition:

```
class GitHub(uplink.Consumer):
    @uplink.timeout(60)
    @uplink.get("/repositories")
    def get_repos(self):
        """Dump every public repository."""

    @uplink.timeout(60)
    @uplink.get("/organizations")
    def get_organizations(self):
        """List all organizations."""
```

3.6.2 Adopting the Argument's Name

Several function argument annotations accept a `name` parameter on construction. For instance, the *Path* annotation uses the `name` parameter to associate the function argument to a URI path parameter:

```
class GitHub(uplink.Consumer):
    @uplink.get("users/{username}")
    def get_user(self, username: uplink.Path("username")): pass
```

For such annotations, you can omit the `name` parameter to have the annotation adopt the name of its corresponding method argument.

For instance, from the previous example, we can omit naming the *Path* annotation since the corresponding argument's name, `username`, matches the intended URI path parameter.

```
class GitHub(uplink.Consumer):
    @uplink.get("users/{username}")
    def get_user(self, username: uplink.Path): pass
```

Some annotations that support this behavior include: *Path*, *uplink.Field*, *Part* *Header*, and *uplink.Query*.

3.6.3 Annotating Your Arguments For Python 2.7

There are several ways to annotate arguments. Most examples in this documentation use function annotations, but this approach is unavailable for Python 2.7 users. Instead, you should either utilize the method annotation *args* or use the optional `args` parameter of the HTTP method decorators (e.g., `uplink.get`).

Using `uplink.args`

One approach for Python 2.7 users involves using the method annotation *args*, arranging annotations in the same order as their corresponding function arguments (again, ignore `self`):

```
class GitHub(uplink.Consumer):
    @uplink.args(uplink.Url, uplink.Path)
    @uplink.get
    def get_commit(self, commits_url, sha): pass
```

The `args` argument

New in version v0.5.0.

The HTTP method decorators (e.g., `uplink.get`) support an optional positional argument `args`, which accepts a list of annotations, arranged in the same order as their corresponding function arguments,

```
class GitHub(uplink.Consumer):
    @uplink.get(args=(uplink.Url, uplink.Path))
    def get_commit(self, commits_url, sha): pass
```

or a mapping of argument names to annotations:

```
class GitHub(uplink.Consumer):
    @uplink.get(args={"commits_url": uplink.Url, "sha": uplink.Path})
    def get_commit(self, commits_url, sha): pass
```

Function Annotations (Python 3 only)

When using Python 3, you can use these classes as function annotations (**PEP 3107**):

```
class GitHub(uplink.Consumer):
    @uplink.get
    def get_commit(self, commit_url: uplink.Url, sha: uplink.Path):
        pass
```

CHAPTER 4

API Reference

This guide details the classes and methods in Uplink's public API.

4.1 API

This guide details the classes and methods in Uplink's public API.

4.1.1 The Base Consumer Class

Consumer

class uplink.**Consumer**(*base_url=''*, *client=None*, *converters=()*, *auth=None*, *hooks=()*, ***kwargs*)

Base consumer class with which to define custom consumers.

Example usage:

```
from uplink import Consumer, get

class GitHub(Consumer):

    @get("/users/{user}")
    def get_user(self, user):
        pass

client = GitHub("https://api.github.com/")
client.get_user("prkumar").json()  # {'login': 'prkumar', ... }
```

Parameters

- **base_url** (str, optional) – The base URL for any request sent from this consumer instance.

- **client** (*optional*) – A supported HTTP client instance (e.g., a `requests.Session`) or an adapter (e.g., *RequestsClient*).
- **converters** (`ConverterFactory`, optional) – One or more objects that encapsulate custom (de)serialization strategies for request properties and/or the response body. (E.g., *MarshmallowConverter*)
- **auth** (`tuple` or `callable`, optional) – The authentication object for this consumer instance.
- **hooks** (`TransactionHook`, optional) – One or more hooks to modify behavior of request execution and response handling (see *response_handler* or *error_handler*).

exceptions

An enum of standard HTTP client exceptions that can be handled.

This property enables the handling of specific exceptions from the backing HTTP client.

Example

```
try:
    github.get_user(user_id)
except github.exceptions.ServerTimeout:
    # Handle the timeout of the request
    ...
```

session

The *Session* object for this consumer instance.

Exposes the configuration of this *Consumer* instance and allows the persistence of certain properties across all requests sent from that instance.

Example usage:

```
import uplink

class MyConsumer(uplink.Consumer):
    def __init__(self, language):
        # Set this header for all requests of the instance.
        self.session.headers["Accept-Language"] = language
        ...
```

Returns *Session*

Session

class uplink.session.**Session**

The session of a *Consumer* instance.

Exposes the configuration of a *Consumer* instance and allows the persistence of certain properties across all requests sent from that instance.

auth

The authentication object for this consumer instance.

base_url

The base URL for any requests sent from this consumer instance.

headers
A dictionary of headers to be sent on each request from this consumer instance.

inject(*hook*, **more_hooks*)
Add hooks (e.g., functions decorated with either *response_handler* or *error_handler*) to the session.

params
A dictionary of querystring data to attach to each request from this consumer instance.

4.1.2 Decorators

The method decorators detailed in this section describe request properties that are relevant to all invocations of a consumer method.

headers

class uplink.**headers**(*arg*, ***kwargs*)
A decorator that adds static headers for API calls.

```
@headers({"User-Agent": "Uplink-Sample-App"})
@get("/user")
def get_user(self):
    """Get the current user"""
```

When used as a class decorator, *headers* applies to all consumer methods bound to the class:

```
@headers({"Accept": "application/vnd.github.v3.full+json"})
class GitHub(Consumer):
    ...
```

headers takes the same arguments as dict.

Parameters

- **arg** – A dict containing header values.
- ****kwargs** – More header values.

params

class uplink.**params**(*arg*, ***kwargs*)
A decorator that adds static query parameters for API calls.

```
@params({"sort": "created"})
@get("/user")
def get_user(self):
    """Get the current user"""
```

When used as a class decorator, *params* applies to all consumer methods bound to the class:

```
@params({"client_id": "my-app-client-id"})
class GitHub(Consumer):
    ...
```

params takes the same arguments as dict.

Parameters

- **arg** – A dict containing query parameters.
- ****kwargs** – More query parameters.

json

class uplink.**json**

Use as a decorator to make JSON requests.

You can annotate a method argument with *uplink.Body*, which indicates that the argument's value should become the request's body. *uplink.Body* has to be either a dict or a subclass of py:class:*collections.Mapping*.

Example

```
@json
@patch(/user")
def update_user(self, **info: Body):
    """Update the current user."""
```

You can alternatively use the *uplink.Field* annotation to specify JSON fields separately, across multiple arguments:

Example

```
@json
@patch(/user")
def update_user(self, name: Field, email: Field("e-mail")):
    """Update the current user."""
```

Further, to set a nested field, you can specify the path of the target field with a tuple of strings as the first argument of *uplink.Field*.

Example

Consider a consumer method that sends a PATCH request with a JSON body of the following format:

```
{
    user: {
        name: "<User's Name>"
    },
}
```

The tuple `("user", "name")` specifies the path to the highlighted inner field:

```
@json
@patch(/user")
def update_user(
                self,
                new_name: Field(("user", "name"))
):
    """Update the current user."""
```

form_url_encoded

class uplink.**form_url_encoded**
URL-encodes the request body.

Used on POST/PUT/PATCH request. It url-encodes the body of the message and sets the appropriate `Content-Type` header. Further, each field argument should be annotated with *uplink.Field*.

Example

```
@form_url_encoded
@post("/users/edit")
def update_user(self, first_name: Field, last_name: Field):
    """Update the current user."""
```

multipart

class uplink.**multipart**
Sends multipart form data.

Multipart requests are commonly used to upload files to a server. Further, annotate each part argument with *Part*.

Example

```
@multipart
@put(/user/photo")
def update_user(self, photo: Part, description: Part):
    """Upload a user profile photo."""
```

timeout

class uplink.**timeout**(*seconds*)
Time to wait for a server response before giving up.

When used on other decorators it specifies how long (in secs) a decorator should wait before giving up.

Example

```
@timeout(60)
@get("/user/posts")
def get_posts(self):
    """Fetch all posts for the current users."""
```

When used as a class decorator, *timeout* applies to all consumer methods bound to the class.

Parameters **seconds** (*int*) – An integer used to indicate how long should the request wait.

args

class uplink.**args**(**annotations*, ***more_annotations*)

Annotate method arguments for Python 2.7 compatibility.

Arrange annotations in the same order as their corresponding function arguments.

Example

```
@args(Path, Query)
@get("/users/{username}")
def get_user(self, username, visibility):
    """Get a specific user."""
```

Use keyword args to target specific method parameters.

Example

```
@args(visibility=Query)
@get("/users/{username}")
def get_user(self, username, visibility):
    """Get a specific user."""
```

Parameters

- ***annotations** – Any number of annotations.
- ****more_annotations** – More annotations, targeting specific method arguments.

response_handler

class uplink.**response_handler**(*handler*, *requires_consumer=False*)

A decorator for creating custom response handlers.

To register a function as a custom response handler, decorate the function with this class. The decorated function should accept a single positional argument, an HTTP response object:

Example

```
@response_handler
def raise_for_status(response):
    response.raise_for_status()
    return response
```

Then, to apply custom response handling to a request method, simply decorate the method with the registered response handler:

Example

```
@raise_for_status
@get("/user/posts")
def get_posts(self):
    """Fetch all posts for the current users."""
```

To apply custom response handling on all request methods of a *uplink.Consumer* subclass, simply decorate the class with the registered response handler:

Example

```
@raise_for_status
class GitHub(Consumer):
    ...
```

Lastly, the decorator supports the optional argument `requires_consumer`. When this option is set to `True`, the registered callback should accept a reference to the *Consumer* instance as its leading argument:

Example

```
@response_handler(requires_consumer=True)
def raise_for_status(consumer, response):
    ...
```

New in version 0.4.0.

error_handler

class uplink.**error_handler**(*exception_handler*, *requires_consumer=False*)

A decorator for creating custom error handlers.

To register a function as a custom error handler, decorate the function with this class. The decorated function should accept three positional arguments: (1) the type of the exception, (2) the exception instance raised, and (3) a traceback instance.

Example

```
@error_handler
def raise_api_error(exc_type, exc_val, exc_tb):
    # wrap client error with custom API error
    ...
```

Then, to apply custom error handling to a request method, simply decorate the method with the registered error handler:

Example

```
@raise_api_error
@get("/user/posts")
def get_posts(self):
    """Fetch all posts for the current users."""
```

To apply custom error handling on all request methods of a *uplink.Consumer* subclass, simply decorate the class with the registered error handler:

Example

```
@raise_api_error
class GitHub(Consumer):
    ...
```

Lastly, the decorator supports the optional argument `requires_consumer`. When this option is set to `True`, the registered callback should accept a reference to the *Consumer* instance as its leading argument:

Example

```
@error_handler(requires_consumer=True)
def raise_api_error(consumer, exc_type, exc_val, exc_tb):
    ...
```

New in version 0.4.0.

Note: Error handlers can not completely suppress exceptions. The original exception is thrown if the error handler doesn't throw anything.

---

inject

class uplink.**inject**(**hooks*)

A decorator that applies one or more hooks to a request method.

New in version 0.4.0.

returns.*

Converting an HTTP response body into a custom Python object is straightforward with Uplink; the *uplink.returns* modules exposes optional decorators for defining the expected return type and data serialization format for any consumer method.

class uplink.returns.**json**(*type=None*, *key=()*, *model=None*, *member=()*)

Specifies that the decorated consumer method should return a JSON object.

```
# This method will return a JSON object (e.g., a dict or list)
@returns.json
@get("/users/{username}")
def get_user(self, username):
    """Get a specific user."""
```

Returning a Specific JSON Field:

The `key` argument accepts a string or tuple that specifies the path of an internal field in the JSON document.

For instance, consider an API that returns JSON responses that, at the root of the document, contains both the server-retrieved data and a list of relevant API errors:

```
{
    "data": { "user": "prkumar", "id": 140232 },
    "errors": []
}
```

If returning the list of errors is unnecessary, we can use the `key` argument to strictly return the inner field `data`:

```
@returns.json(key="data")
@get("/users/{username}")
def get_user(self, username):
    """Get a specific user."""
```

New in version v0.5.0.

`uplink.returns.`**`from_json`**
alias of *json*

class `uplink.returns.`**`schema`**(*type*)
Specifies that the function returns a specific type of response.

In Python 3, to provide a consumer method's return type, you can set it as the method's return annotation:

```
@get("/users/{username}")
def get_user(self, username) -> UserSchema:
    """Get a specific user."""
```

For Python 2.7 compatibility, you can use this decorator instead:

```
@returns.schema(UserSchema)
@get("/users/{username}")
def get_user(self, username):
    """Get a specific user."""
```

To have Uplink convert response bodies into the desired type, you will need to define an appropriate converter (e.g., using *uplink.loads*).

New in version v0.5.1.

4.1.3 Function Annotations

For programming in general, function parameters drive a function's dynamic behavior; a function's output depends normally on its inputs. With `uplink`, function arguments parametrize an HTTP request, and you indicate the dynamic parts of the request by appropriately annotating those arguments with the classes detailed in this section.

Path

class `uplink.`**`Path`**(*name=None*, *type=None*)
Substitution of a path variable in a URI template.

URI template parameters are enclosed in braces (e.g., `{name}`). To map an argument to a declared URI parameter, use the *Path* annotation:

```
class TodoService(object):
    @get("/todos{/id}")
    def get_todo(self, todo_id: Path("id")): pass
```

Then, invoking `get_todo` with a consumer instance:

```
todo_service.get_todo(100)
```

creates an HTTP request with a URL ending in `/todos/100`.

Note: Any unannotated function argument that shares a name with a URL path parameter is implicitly annotated with this class at runtime.

For example, we could simplify the method from the previous example by matching the path variable and method argument names:

```
@get("/todos{/id}")
def get_todo(self, id): pass
```

Query

class uplink.**Query**(*name=None*, *encoded=False*, *type=None*, *encode_none=None*)

Set a dynamic query parameter.

This annotation turns argument values into URL query parameters. You can include it as function argument annotation, in the format: `<query argument>: uplink.Query`.

If the API endpoint you are trying to query uses `q` as a query parameter, you can add `q: uplink.Query` to the consumer method to set the `q` search term at runtime.

Example

```
@get("/search/commits")
def search(self, search_term: Query("q")):
    """Search all commits with the given search term."""
```

To specify whether or not the query parameter is already URL encoded, use the optional `encoded` argument:

```
@get("/search/commits")
def search(self, search_term: Query("q", encoded=True)):
    """Search all commits with the given search term."""
```

To specify if and how `None` values should be encoded, use the optional `encode_none` argument:

```
@get("/search/commits")
def search(self, search_term: Query("q"),
           search_order: Query("o", encode_none="null")):
    """
    Search all commits with the given search term using the
    optional search order.
    """
```

Parameters

- **encoded** (bool, optional) – Specifies whether the parameter name and value are already URL encoded.
- **encode_none** (str, optional) – Specifies an optional string with which None values should be encoded. If not specified, parameters with a value of None will not be sent.

QueryMap

class uplink.**QueryMap**(*encoded=False*, *type=None*)

A mapping of query arguments.

If the API you are using accepts multiple query arguments, you can include them all in your function method by using the format: `<query argument>: uplink.QueryMap`

Example

```
@get("/search/users")
def search(self, **params: QueryMap):
    """Search all users."""
```

Parameters **encoded** (bool, optional) – Specifies whether the parameter name and value are already URL encoded.

Header

class uplink.**Header**(*name=None*, *type=None*)

Pass a header as a method argument at runtime.

While *uplink.headers* attaches static headers that define all requests sent from a consumer method, this class turns a method argument into a dynamic header value.

Example

```
@get("/user")
def (self, session_id: Header("Authorization")):
    """Get the authenticated user"""
```

HeaderMap

class uplink.**HeaderMap**(*type=None*)

Pass a mapping of header fields at runtime.

Field

class uplink.**Field**(*name=None*, *type=None*)

Defines a form field to the request body.

Use together with the decorator *uplink.form_url_encoded* and annotate each argument accepting a form field with *uplink.Field*.

Example::

```
@form_url_encoded
@post("/users/edit")
def update_user(self, first_name: Field, last_name: Field):
    """Update the current user."""
```

FieldMap

class uplink.**FieldMap**(*type=None*)

Defines a mapping of form fields to the request body.

Use together with the decorator *uplink.form_url_encoded* and annotate each argument accepting a form field with *uplink.FieldMap*.

Example

```
@form_url_encoded
@post("/user/edit")
def create_post(self, **user_info: FieldMap):
    """Update the current user."""
```

Part

class uplink.**Part**(*name=None*, *type=None*)

Marks an argument as a form part.

Use together with the decorator *uplink.multipart* and annotate each form part with *uplink.Part*.

Example

```
@multipart
@put(/user/photo")
def update_user(self, photo: Part, description: Part):
    """Upload a user profile photo."""
```

PartMap

class uplink.**PartMap**(*type=None*)

A mapping of form field parts.

Use together with the decorator *uplink.multipart* and annotate each part of form parts with *uplink.PartMap*

Example

```
@multipart
@put(/user/photo")
def update_user(self, photo: Part, description: Part):
    """Upload a user profile photo."""
```

Body

class uplink.**Body**(*type=None*)
Set the request body at runtime.

Use together with the decorator *uplink.json*. The method argument value will become the request's body when annotated with *uplink.Body*.

Example

```
@json
@patch(/user")
def update_user(self, **info: Body):
    """Update the current user."""
```

Url

class uplink.**Url**
Sets a dynamic URL.

Provides the URL at runtime as a method argument. Drop the decorator parameter path from `uplink.get` and annotate the corresponding argument with *uplink.Url*

Example

```
@get
def get(self, endpoint: Url):
    """Execute a GET requests against the given endpoint"""
```

4.1.4 HTTP Clients

The `client` parameter of the `Consumer` constructor offers a way to swap out Requests with another HTTP client, including those listed here:

```
github = GitHub(BASE_URL, client=...)
```

Requests

class uplink.**RequestsClient**(*session=None*, ***kwargs*)
A `requests` client that returns `requests.Response` responses.

Parameters **session** (`requests.Session`, optional) – The session that should handle sending requests. If this argument is omitted or set to `None`, a new session will be created.

Aiohttp

class uplink.**AiohttpClient**(*session=None*, ***kwargs*)
An aiohttp client that creates awaitable responses.

Note: This client is an optional feature and requires the aiohttp package. For example, here's how to install this extra using pip:

```
$ pip install uplink[aiohttp]
```

Parameters **session** (aiohttp.ClientSession, optional) – The session that should handle sending requests. If this argument is omitted or set to None, a new session will be created.

Twisted

class uplink.**TwistedClient**(*session=None*)
Client that returns twisted.internet.defer.Deferred responses.

Note: This client is an optional feature and requires the twisted package. For example, here's how to install this extra using pip:

```
$ pip install uplink[twisted]
```

Parameters **session** (requests.Session, optional) – The session that should handle sending requests. If this argument is omitted or set to None, a new session will be created.

4.1.5 Converters

The converter parameter of the *uplink.Consumer* constructor accepts a custom adapter class that handles serialization of HTTP request properties and deserialization of HTTP response objects:

```
github = GitHub(BASE_URL, converter=...)
```

Starting with version v0.5, some out-of-the-box converters are included automatically and don't need to be explicitly provided through the converter parameter. These implementations are detailed below.

Marshmallow

Uplink comes with optional support for marshmallow.

class uplink.converters.**MarshmallowConverter**
A converter that serializes and deserializes values using marshmallow schemas.

To deserialize JSON responses into Python objects with this converter, define a marshmallow.Schema subclass and set it as the return annotation of a consumer method:

```
@get("/users")
def get_users(self, username) -> UserSchema():
    '''Fetch a single user'''
```

Note: This converter is an optional feature and requires the `marshmallow` package. For example, here's how to install this feature using pip:

```
$ pip install uplink[marshmallow]
```

Note: Starting with version v0.5, this converter factory is automatically included if you have `marshmallow` installed, so you don't need to provide it when constructing your consumer instances.

Converting Collections

New in version v0.5.0.

Uplink can convert collections of a type, such as deserializing a response body into a list of users. If you have `typing` installed (the module is part of the standard library starting Python 3.5), you can use type hints (see **PEP 484**) to specify such conversions. You can also leverage this feature without `typing` by using one of the proxy types defined in `uplink.types`.

The following converter factory implements this feature and is automatically included, so you don't need to provide it when constructing your consumer instance:

class uplink.converters.**TypingConverter**

An adapter that serializes and deserializes collection types from the `typing` module, such as `typing.List`.

Inner types of a collection are recursively resolved, using other available converters if necessary. For instance, when resolving the type hint `typing.Sequence[UserSchema]`, where `UserSchema` is a custom `marshmallow.Schema` subclass, the converter will resolve the inner type using *uplink.converters.MarshmallowConverter*.

```
@get("/users")
def get_users(self) -> typing.Sequence[UserSchema]:
    '''Fetch all users.'''
```

Note: The `typing` module is available in the standard library starting from Python 3.5. For earlier versions of Python, there is a port of the module available on PyPI.

However, you can utilize this converter without the `typing` module by using one of the proxies defined by *uplink.returns* (e.g., *uplink.types.List*).

Here are the collection types defined in `uplink.types`. You can use these or the corresponding type hints from `typing` to leverage this feature:

uplink.types.**List**

list() -> new empty list list(iterable) -> new list initialized from iterable's items A proxy for `typing.List` that is safe to use in type hints with Python 3.4 and below.

```
@returns.from_json
@get("/users")
def get_users(self) -> types.List[str]:
    """Fetches all users"""
```

uplink.types.**Dict**

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's

(key, value) pairs

dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable:

d[k] = v

dict(kwargs) -> new dictionary initialized with the name=value pairs** in the keyword argument list. For example: dict(one=1, two=2)

A proxy for `typing.Dict` that is safe to use in type hints with Python 3.4 and below.

```
@returns.from_json
@get("/users")
def get_users(self) -> types.Dict[str, str]:
    """Fetches all users"""
```

Writing Custom JSON Converters

As a shorthand, you can define custom JSON converters using the *`@loads.from_json`* (deserialization) and *`@dumps.to_json`* (serialization) decorators.

These classes can be used as decorators to create converters of a class and its subclasses:

```
# Creates a converter that can deserialize the given `json` in to an
# instance of a `Model` subtype.
@loads.from_json(Model)
def load_model_from_json(model_type, json):
    ...
```

Note: Unlike consumer methods, these functions should be defined outside of a class scope.

To use the converter, provide the generated converter object when instantiating a *`Consumer`* subclass, through the `converter` constructor parameter:

```
github = GitHub(BASE_URL, converter=load_model_from_json)
```

Alternatively, you can add the `@install` decorator to register the converter function as a default converter, meaning the converter will be included automatically with any consumer instance and doesn't need to be explicitly provided through the :py:obj:`converter` parameter:

```
from uplink import install, loads

# Register the function as a default loader for the given model class.
@install
@loads.from_json(Model)
def load_model_from_json(model_type, json):
    ...
```

class uplink.**loads**(*base_class*, *annotations=()*)

Builds a custom object deserializer.

This class takes a single argument, the base model class, and registers the decorated function as a deserializer for that base class and all subclasses.

Further, the decorated function should accept two positional arguments: (1) the encountered type (which can be the given base class or a subclass), and (2) the response data.

```
@loads(ModelBase)
def load_model(model_cls, data):
    ...
```

New in version v0.5.0.

classmethod from_json(*base_class*, *annotations=()*)

Builds a custom JSON deserialization strategy.

This decorator accepts the same arguments and behaves like *uplink.loads*, except that the second argument of the decorated function is a JSON object:

```
@loads.from_json(User)
def from_json(user_cls, json):
    return user_cls(json["id"], json["username"])
```

Notably, only consumer methods that have the expected return type (i.e., the given base class or any subclass) and are decorated with *uplink.returns.from_json* can leverage the registered strategy to deserialize JSON responses.

For example, the following consumer method would leverage the *from_json()* strategy defined above:

```
@returns.from_json
@get("user")
def get_user(self) -> User: pass
```

New in version v0.5.0.

class uplink.**dumps**(*base_class*, *annotations=()*)

Builds a custom object serializer.

This decorator takes a single argument, the base model class, and registers the decorated function as a serializer for that base class and all subclasses.

Further, the decorated function should accept two positional arguments: (1) the encountered type (which can be the given base class or a subclass), and (2) the encountered instance.

```
@dumps(ModelBase)
def deserialize_model(model_cls, model_instance):
    ...
```

New in version v0.5.0.

classmethod to_json(*base_class*, *annotations=()*)

Builds a custom JSON serialization strategy.

This decorator accepts the same arguments and behaves like *uplink.dumps*. The only distinction is that the decorated function should be JSON serializable.

```
@dumps.to_json(ModelBase)
def to_json(model_cls, model_instance):
    return model_instance.to_json()
```

Notably, only consumer methods that are decorated with py:class:*uplink.json* and have one or more argument annotations with the expected type (i.e., the given base class or a subclass) can leverage the registered strategy.

For example, the following consumer method would leverage the *to_json()* strategy defined above, given `User` is a subclass of `ModelBase`:

```
@json
@post("user")
def change_user_name(self, name: Field(type=User): pass
```

New in version v0.5.0.

CHAPTER 5

Miscellaneous

5.1 Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to the Semantic Versioning scheme.

5.1.1 0.7.0 - 2018-12-06

Added

- `Consumer.exceptions` property for handling common client exceptions in a client-agnostic way. (#117)
- Optional argument `requires_consumer` for `response_handler` and `error_handler`; when set to `True`, the registered callback should accept a reference to a `Consumer` instance as its leading argument. (#118)

Changed

- For a `Query`-annotated argument, a `None` value indicates that the query parameter should be excluded from the request. Previous behavior was to encode the parameter as `...?name=None`. To retain this behavior, specify the new `encode_none` parameter (i.e., `Query(..., encode_none="None")`). (#126 by @nphilipp)

Fixed

- Support for changes to `Schema().load` and `Schema().dump` in `marshmallow` v3. (#109)

5.1.2 0.6.1 - 2018-9-14

Changed

- When the `type` parameter of a function argument annotation, such as `Query` or `Body`, is omitted, the type of the annotated argument's value is no longer used to determine how to convert the value before it's passed to the backing client; the argument's value is converted only when its `type` is explicitly set.

5.1.3 0.6.0 - 2018-9-11

Added

- The `session` property to the `Consumer` base class, exposing the consumer instance's configuration and allowing for the persistence of certain properties across requests sent from that instance.
- The `params` decorator, which when applied to a method of a `Consumer` subclass, can add static query parameters to each API call.
- The `converters.Factory` base class for defining integrations with other serialization formats and libraries.
- The `uplink.install` decorator for registering extensions, such as a custom `converters.Factory` implementation, to be applied broadly.

Fixed

- Issue with detecting `typing.List` and `typing.Dict` for converting collections on Python 3.7.
- `RuntimeWarning` that "`ClientSession.close` was never awaited" when using `aiohttp >= 3.0`.

Changed

- When using the `marshmallow` integration, Uplink no longer suppresses `Schema` validation errors on deserialization; users can now handle these exceptions directly.

5.1.4 0.5.5 - 2018-8-01

Fixed

- Issue with sending JSON list `Body` using `@json` annotation.

5.1.5 0.5.4 - 2018-6-26

Fixed

- When using `uplink.AiohttpClient` with `aiohttp>=3.0`, the underlying `aiohttp.ClientSession` would remain open on program exit.

5.1.6 0.5.3 - 2018-5-31

Fixed

- Issue where adding two or more response handlers (i.e., functions decorated with `uplink.response_handler`) to a method caused a `TypeError`.

5.1.7 0.5.2 - 2018-5-30

Fixed

- Applying `returns.json` decorator without arguments should produce JSON responses when the decorated method is lacking a return value annotation.

5.1.8 0.5.1 - 2018-4-10

Added

- Decorator `uplink.returns.model` for specifying custom return type without indicating a specific data deserialization format.

Fixed

- Have `uplink.Body` decorator accept any type, not just mappings.
- Reintroduce the `uplink.returns` decorator.

5.1.9 0.5.0 - 2018-4-06

Added

- Decorators for convenient registration of custom serialization. (`uplink.dumps`) and deserialization (`uplink.loads`) strategies.
- Support for setting nested JSON fields with `uplink.Field` and `uplink.json`.
- Optional `args` parameter to HTTP method decorators (e.g., `uplink.get`) for another Python 2.7-compatible alternative to annotating consumer method arguments with function annotations.
- Decorator `uplink.returns.json` for converting HTTP response bodies into JSON objects or custom Python objects.
- Support for converting collections (e.g., converting a response body into a list of users).

Changed

- Leveraging built-in converters (such as `uplink.converters.MarshmallowConverter`) no longer requires providing the converter when instantiating an `uplink.Consumer` subclass, as these converters are now implicitly included.

Fixed

- `uplink.response_handler` and `uplink.error_handler` properly adopts the name and docstring of the wrapped function.

5.1.10 0.4.1 - 2018-3-10

Fixed

- Enforce method-level decorators override class-level decorators when they conflict.

5.1.11 0.4.0 - 2018-2-10

Added

- Support for Basic Authentication.
- The `response_handler` decorator for defining custom response handlers.
- The `error_handler` decorator for defining custom error handlers.
- The `inject` decorator for injecting other kinds of middleware.
- The `Consumer._inject` method for adding middleware to a consumer instance.
- Support for annotating constructor arguments of a `Consumer` subclass with built-in function annotations like `Query` and `Header`.

5.1.12 0.3.0 - 2018-1-09

Added

- HTTP HEAD request decorator by @brandonio21.
- Support for returning deserialized response objects using `marshmallow` schemas.
- Constructor parameter for `Query` and `QueryMap` to support already encoded URL parameters.
- Support for using `requests.Session` and `aiohttp.ClientSession` instances with the `client` parameter of the `Consumer` constructor.

Changed

- `aiohttp` and `twisted` are now optional dependencies/extras.

Fixed

- Fix for calling a request method with `super`, by @brandonio21.
- Fix issue where method decorators would incorrectly decorate inherited request methods.

5.1.13 0.2.2 - 2017-11-23

Fixed

- Fix for error raised when an object that is not a class is passed into the `client` parameter of the `Consumer` constructor, by @kadrach.

5.1.14 0.2.0 - 2017-11-03

Added

- The class `uplink.Consumer` by @itstehkman. Consumer classes should inherit this base. class, and creating consumer instances happens through instantiation.
- Support for `asyncio` for Python 3.4 and above.
- Support for `twisted` for all supported Python versions.

Changed

- **BREAKING**: Invoking a consumer method now builds and executes the request, removing the extra step of calling the `execute` method.

Deprecated

- Building consumer instances with `uplink.build`. Instead, Consumer classes should inherit `uplink.Consumer`.

Fixed

- Header link for version 0.1.1 in changelog.

5.1.15 0.1.1 - 2017-10-21

Added

- Contribution guide, `CONTRIBUTING.rst`.
- "Contributing" Section in README.rst that links to contribution guide.
- `AUTHORS.rst` file for listing project contributors.
- Adopt Contributor Covenant Code of Conduct.

Changed

- Replaced tentative contributing instructions in preview notice on documentation homepage with link to contribution guide.

5.1.16 0.1.0 - 2017-10-19

Added

- Python ports for almost all method and argument annotations in Retrofit.
- Adherence to the variation of the semantic versioning scheme outlined in the official Python package distribution tutorial.
- MIT License
- Documentation with introduction, instructions for installing, and quick getting started guide covering the builder and all method and argument annotations.
- README that contains GitHub API v3 example, installation instructions with `pip`, and link to online documentation.

Python Module Index

u

Index

U