Uplink Documentation

Release 0.5.5

Raj Kumar

Aug 03, 2018

Contents

A Declarative HTTP Client for Python. Inspired by Retrofit.

Note: Uplink is currently in initial development. Until the official release (`v1.0.0`), the public API should be considered provisional. Although we don't expect any considerable changes to the API at this point, please avoid using the code in production, for now.

However, while Uplink is under construction, we invite eager users to install early and provide open feedback, which can be as simple as opening a GitHub issue when you notice a missing feature, latent defect, documentation oversight, etc.

Moreover, for those interested in contributing, checkout the Contribution Guide on GitHub!

Uplink turns your HTTP API into a Python class.

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

class GitHub(Consumer):

    @get("users/{user}/repos")
    def list_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.list_repos("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).

Use decorators and function annotations to describe the HTTP request:

- URL parameter replacement and query parameter support
- Convert response bodies into Python objects (e.g., using `marshmallow` or a custom converter)
- JSON, URL-encoded, and multipart request body and file upload
- Inject functions as **middleware** to define custom response and error handling

CHAPTER 1

The User Manual

Follow this guide to get up and running with Uplink.

1.1 Installation

1.1.1 Using pip

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

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

1.1.2 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
```

1.1.3 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]
```

1.2 Introduction

Uplink delivers reusable and self-sufficient objects for accessing HTTP webservices, with minimal code and user pain. Simply define your consumers using decorators and function annotations, and we'll handle the rest for you... pun intended, obviously

1.2.1 Static Request Handling

Method decorators describe request properties that are relevant to all invocations of a consumer method.

For instance, consider the following GitHub API consumer:

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

Annotated with `timeout`, the method `get_repos()` will build HTTP requests that wait an allotted number of seconds – 60, in this case – for the server to respond before giving up.

As method annotations are simply decorators, you can stack one on top of another for chaining:

```
class GitHub(uplink.Consumer):
    @uplink.headers({"Accept": "application/vnd.github.v3.full+json"})
    @uplink.timeout(60)
    @uplink.get("/repositories")
    def get_repos(self):
        """Dump every public repository."""
```

1.2.2 Dynamic Request Handling

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.

To illustrate, for the method `get_user()` in the following snippet, we have flagged the argument `username` as a URI placeholder replacement using the *`Path`* annotation:

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

Invoking this method on a consumer instance, like so:

```
github.get_user(username="prkumar")
```

Builds an HTTP request that has a URL ending with `users/prkumar`.

Note: As you probably took away from the above example: when parsing the method's signature for argument annotations, `uplink` skips the instance reference argument, which is the leading method parameter and usually named `self`.

1.3 Quickstart

Decorators and function annotations indicate how a request will be handled.

1.3.1 Request Method

Uplink offers decorators that turn any method into a request definition. These decorators provide the request method and relative URL of the intended request: `get`, `post`, `put`, `patch` and `delete`.

The relative URL of the resource is specified in the decorator.

```
@get("users/list")
```

You can also specify query parameters in the URL.

```
@get("users/list?sort=desc")
```

Moreover, request methods must be bound to a `Consumer` subclass.

```
class MyApi(Consumer):
    @get("users/list")
    def list_users(self):
        """List all users."""
```

1.3.2 URL Manipulation

A request URL can be updated dynamically using URI template parameters. 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("group/{id}/users")
def group_list(self, id): pass
```

or use the *`Path`* annotation.

```
@get("group/{id}/users")
def group_list(self, group_id: Path("id")): pass
```

Query parameters can also be added.

```
@get("group/{id}/users")
def group_list(self, group_id: Path("id"), sort: Query): pass
```

For complex query parameter combinations, a mapping can be used:

```
@get("group/{id}/users")
def group_list(self, group_id: Path("id"), options: QueryMap): pass
```

1.3.3 Request Body

An argument's value can be specified for use as an HTTP request body with the *Body* annotation:

```
@post("users/new")
def create_user(self, user: Body): pass
```

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

```
@post("users/new")
def create_user(self, **user_info: Body): pass
```

1.3.4 Form Encoded, Multipart, and JSON

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
@post("user/edit")
def update_user(self, first_name: Field, last_name: Field): pass
```

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

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

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

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

1.3.5 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 be updated dynamically using the *Header* function parameter annotation:

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

1.3.6 Synchronous vs. Asynchronous

By default, Uplink uses the Requests library to make requests. However, the `client` parameter of the `Consumer` constructor offers a way to swap out Requests with another HTTP client:

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

Notably, Requests blocks while waiting for a response from a server. For non-blocking requests, Uplink comes with optional support for `asyncio` and `twisted`. Checkout this example on GitHub for more.

1.3.7 Deserializing the Response Body

Uplink makes it easy and optional to convert HTTP response bodies into data model objects, whether you leverage Uplink's built-in support for libraries such as `marshmallow` (see *uplink.converters.MarshmallowConverter*) or use *uplink.loads* to write custom conversion logic that fits your unique needs.

At the least, you need to specify the expected return type using a decorator from the *uplink.returns* module. *uplink.returns.json* is handy when working with APIs that provide JSON responses:

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

Python 3 users can alternatively use a return type hint:

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

The final step is to register a strategy that converts the HTTP response into the expected return type. To this end, *uplink.loads()* can register a function that handles such deserialization for a particular class and all its subclasses.

```
# The base class for all model types, including User from above.
from models import ModelBase

# Tell Uplink how to deserialize JSON responses into our model classes:
@loads.install  # Make this available to all consumer instances.
@loads.from_json(ModelBase)
def load_model_from_json(model_cls, json_obj):
    return model_cls.from_json(json_obj)
```

This step is not required if you define your data model objects using a library for whom Uplink has built-in support, such as `marshmallow` (see *`uplink.converters.MarshmallowConverter`*).

Note: For API endpoints that return collections (such as a list of users), Uplink just needs to know how to deserialize the element type (e.g., a user), offering built-in support for *Converting Collections*.

1.3.8 Custom Response and Error Handling

New in version 0.4.0.

To register a custom response or error handler, decorate a function with the `response_handler` or `error_handler` decorator.

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

For instance, the function `returns_success()` defined below is a response handler that output whether or not the request was successful:

```
@uplink.response_handler
def returns_success(response):
    return response.status == 200
```

Now, `returns_success()` can be used as a decorator to inject its custom response handling into any request method:

```
@returns_success
@put("/todos")
def create_todo(self, title):
    """Creates a todo with the given title."""
```

To apply the function's handling onto all request methods of a `Consumer` subclass, we can simply use the registered handler as a class decorator:

```
@returns_success
class TodoApp(uplink.Consumer):
    ...
```

Similarly, functions decorated with `error_handler` are registered error handlers. When applied to a request method, these handlers are invoked when the underlying HTTP client fails to execute a request:

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

Notably, handlers can be stacked on top of one another to chain their behavior:

```
@raise_api_error
@returns_success
class TodoApp(uplink.Consumer):
    ...
```

1.3.9 Annotating `__init__()` Arguments

New in version 0.4.0.

Function annotations like `Query` and `Header` can be used with constructor arguments of a `Consumer` subclass. When a new consumer instance is created, the value of these arguments are applied to all requests made through that instance.

For example, the following consumer accepts the API access token as the constructor argument `access_token`:

```
class GitHub(uplink.Consumer):

    def __init__(self, access_token: uplink.Query):
        ...

    @uplink.post("/user")
    def update_user(self, **info: Body):
        """Update the authenticated user"""
```

Now, all requests made from an instance of this consumer class will be authenticated with the access token passed in at initialization:

```
github = TodoApp("my-github-access-token")

# This request will include the above access token as a query parameter.
github.update_user(bio="Beam me up, Scotty!")
```

1.3.10 `_inject()` Request Properties

New in version 0.4.0.

As an alternative to *Annotating __init__() Arguments*, you can achieve a similar behavior with more control by using the `Consumer._inject()` method. With this method, you can calculate request properties within plain old python methods.

```
class TodoApp(uplink.Consumer):

    def __init__(self, username, password)
        # Create an access token
        api_key = create_api_key(username, password)

        # Inject it.
        self._inject(uplink.Query("api_key").with_value(api_key))
```

Similar to the annotation style, request properties added with `_inject()` method are applied to all requests made through the consumer instance.

1.4 Authentication

This section covers how to do authentication with Uplink.

1.4.1 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"))
```

1.4.2 Other Authentication

Often, APIs accept credentials as header values 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 more than one request requires authentication, you can make the token an argument of your consumer constructor (see *Annotating __init__() Arguments*):

```
class GitHub(Consumer):

    def __init__(self, base_url, access_token: Query)
        ...
```

1.4.3 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)
```

1.5 Tips & Tricks

Here are a few ways to simplify consumer definitions.

1.5.1 Decorating All Request Methods in a Class

To apply a decorator across all methods in a class, 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."""
```

1.5.2 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`*.

1.5.3 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 2

The Public API

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

2.1 Decorators

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

2.1.1 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.

2.1.2 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: .. code-block:: python

@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."""
```

2.1.3 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."""
```

2.1.4 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."""
```

2.1.5 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.

2.1.6 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.

2.1.7 response_handler

class uplink.**response_handler**(*func*)

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):
    ...
```

New in version 0.4.0.

2.1.8 error_handler

class uplink.**error_handler**(*func*)

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):
    ...
```

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.

2.1.9 inject

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

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

Example

```
@inject(Query("sort").with_value("pushed"))
@get("users/{user}/repos")
def list_repos(self, user):
    """Lists user's public repos by latest pushed."""
```

New in version 0.4.0.

2.1.10 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.

uplink.returns.**from_json**

alias of *json*

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

Specifies that the decorated consumer method should return a JSON object. If a *model* is provided, the resulting JSON object is converted into the *model* object using an appropriate converter (see *uplink.loads.from_json()*).

```
# 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:

This decorator accepts two optional arguments. The `member` 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 `member` argument to strictly return the inner field `data`:

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

Deserialize Objects from JSON:

Often, JSON responses represent models in your application. If an existing Python object encapsulates this model, use the *model* argument to specify it as the return type:

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

For Python 3 users, you can alternatively provide a return value annotation. Hence, the previous code is equivalent to the following in Python 3:

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

Both usages typically require also registering a converter that knows how to deserialize the JSON into your data model object (see *uplink.loads.from_json()*). This step is unnecessary if these objects are defined using a library for whom Uplink has built-in support, such as marshmallow (see *uplink.converters.MarshmallowConverter*).

New in version v0.5.0.

class uplink.returns.**model**(*type*)

Specifies that the function returns a specific class.

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.model(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.

2.2 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.

2.2.1 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
```

2.2.2 Query

class `uplink.`**`Query`**(*name=None*, *encoded=False*, *type=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."""
```

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

`with_value`(*value*)

Creates an object that can be used with the `Consumer._inject` method or *`inject`* decorator to inject request properties with specific values.

New in version 0.4.0.

2.2.3 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.

with_value(*value*)

Creates an object that can be used with the `Consumer._inject` method or *inject* decorator to inject request properties with specific values.

New in version 0.4.0.

2.2.4 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"""
```

with_value(*value*)

Creates an object that can be used with the `Consumer._inject` method or *inject* decorator to inject request properties with specific values.

New in version 0.4.0.

2.2.5 HeaderMap

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

Pass a mapping of header fields at runtime.

with_value(*value*)

Creates an object that can be used with the `Consumer._inject` method or *inject* decorator to inject request properties with specific values.

New in version 0.4.0.

2.2.6 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."""
```

2.2.7 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."""
```

2.2.8 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."""
```

2.2.9 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."""
```

2.2.10 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."""
```

2.2.11 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"""
```

2.3 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=...)
```

2.3.1 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.

2.3.2 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.

2.3.3 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.

2.4 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.

2.4.1 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.

2.4.2 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.

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

2.4.3 Writing a Custom Converter

You can define custom converters by using *`uplink.loads`* and *`uplink.dumps`*.

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

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

To use the converter, you can 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 `uplink.loads.install()` or `uplink.dumps.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:

```
# Register the function as a default loader for the given model class.
@loads.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(ModelBase)
def from_json(model_cls, json_object):
    return model_cls.from_json(json_object)
```

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.json` can leverage the registered strategy to deserialize JSON responses.

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

```
@returns.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.

2.5 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.

2.5.1 0.5.5 - 2018-8-01

Fixed

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

2.5.2 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.

2.5.3 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‘.

2.5.4 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.

2.5.5 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.

2.5.6 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.

2.5.7 0.4.1 - 2018-3-10

Fixed

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

2.5.8 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`.

2.5.9 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.

2.5.10 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.

2.5.11 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.

2.5.12 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.

2.5.13 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