Uplink Documentation

Release 0.1.1

Raj Kumar

Apr 07, 2018

Contents

A Declarative HTTP Client for Python. Inspired by Retrofit.

Note: Uplink is currently in initial development and, therefore, not production ready at the moment. Furthermore, as the package follows a semantic versioning scheme, the public API outlined in this documentation should be considered tentative until the `v1.0.0` release.

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 ('tis **'Hacktoberfest'__**, after all)!

CHAPTER 1

A Quick Walkthrough, with GitHub API v3:

Using decorators and function annotations, you can turn any plain old Python class into a self-describing consumer of your favorite HTTP webservice:

```
from uplink import *

# To register entities that are common to all API requests, you can
# decorate the enclosing class rather than each method separately:
@headers({"Accept": "application/vnd.github.v3.full+json"})
class GitHub(object):

    @get("/users/{username}")
    def get_user(self, username):
        """Get a single user."""

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

To construct a consumer instance, use the helper function `uplink.build()`:

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

To access the GitHub API with this instance, we simply invoke any of the methods that we defined in the interface above. To illustrate, let's update my GitHub profile bio:

```
r = github.update_user(token, bio="Beam me up, Scotty!").execute()
```

Voila, `update_user()` builds the request seamlessly (using the decorators and annotations from the method's definition), and `execute()` sends that synchronously over the network. Furthermore, the returned response `r` is simply a `requests.Response` (documentation):

```
print(r.json()) # {u'disk_usage': 216141, u'private_gists': 0, ...
```

In essence, Uplink delivers reusable and self-sufficient objects for accessing HTTP webservices, with minimal code and user pain .

CHAPTER 2

The User Manual

Follow this guide to get up and running with Uplink.

2.1 Installation

2.1.1 Using `pip`

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

```
$ pip install uplink
```

2.1.2 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
```

2.2 Introduction

Uplink delivers reusable and self-sufficient objects for accessing HTTP webservices, with minimal code and user pain.

Defining similar objects with other Python HTTP clients, such as `requests`, often requires writing boilerplate code and layers of abstraction. With Uplink, simply define your consumers using decorators and function annotations, and we'll handle the REST for you! (Pun intended, obviously.)

2.2.1 Method Annotations: Static Request Handling

Essentially, method annotations describe request properties that are relevant to all invocations of a consumer method.

For instance, consider the following GitHub API consumer:

```
class GitHub(object):
    @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.

Applying Multiple Method Annotations

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

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

A Shortcut for Annotating All Methods in a Class

To apply an annotation across all methods in a class, you can simply annotate the class rather than each method individually:

```
@uplink.timeout(60)
class GitHub(object):
    @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 one:

```
class GitHub(object):
    @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."""
```

2.2.2 Arguments Annotations: Dynamic Request Handling

In programming, parametrization drives 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(object):
    @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`.

Adopting the Argument's Name

When you initialize a named annotation, such as a `Path` or `Field`, without a name (by omitting the `name` parameter), it adopts the name of its corresponding method argument.

For example, in the snippet below, we can omit naming the `Path` annotation since the corresponding argument's name, `username`, matches the intended URI path parameter:

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

Annotating Your Arguments

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 can use argument annotations as decorators or utilize the method annotation `args`.

Argument Annotations as Decorators

For one, annotations can work as function decorators. With this approach, annotations are mapped to arguments from "bottom-up".

For instance, in the below definition, the `Url` annotation corresponds to `commits_url`, and `Path` to `sha`.

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

Using `uplink.args`

The second approach involves using the method annotation `args`, arranging annotations in the same order as their corresponding function arguments (again, ignore `self`):

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

Function Annotations (Python 3 only)

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

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

2.2.3 Integration with `python-requests`

Experienced users of Kenneth Reitz's well-established Requests library might be happy to read that Uplink uses `requests` behind-the-scenes and bubbles `requests.Response` instances back up to the user.

2.3 Getting Started

In this section, we'll cover the basics with Uplink. To illustrate usage with an existent service, we mainly provide examples with GitHub's API. To try them out yourself, you can simply copy the code snippets into a script or the Python console.

2.3.1 Making a Request

The simplest API consumer method requires only an HTTP method decorator.

For example, let's define an GitHub API consumer that can retrieve all the public repositories hosted on the site:

```
import uplink

class GitHub(object):
    @uplink.get("/repositories")
    def get_repos(self): pass
```

Now, to fetch the list of public repositories hosted on `github.com`, we simply invoke the `get_repos()` with a consumer instance:

```
github = uplink.build(GitHub, base_url="https://api.github.com")
response = github.get_repos().execute()
print(response.json()) # [{u'issues_url': u'https://api.github.com/repos/mojombo/grit/
↪issues{/number}', ...
```

To summarize, we used the `get` decorator to indicate that the `get_repos()` method handles an HTTP `GET` request targeting the `/repositories` endpoint.

Further, Uplink currently supports `get`, `post`, `put`, `patch`, and `delete`.

Creating Consumer Instances with `uplink.build`

As illustrated in the previous example, to create consumer instances, use the `uplink.build()` function. Notably, this helper function affords us the ability to reuse consumers in different contexts.

For instance, by simply changing the function's `base_url` parameter, we could use the same GitHub API consumer against the main website, `github.com`, and any GitHub Enterprise instance, since they offer identical APIs.

2.3.2 Setting the URL

To set a **static URL**, use the the leading parameter, `path`, of the HTTP method decorator:

```
class GitHub(object):
    @uplink.get("/repositories")
    def get_repos(self): pass
```

Alternatively, you can provide the URL at runtime as a method argument. To set a **dynamic URL**, omit the decorator parameter `path` and annotate the corresponding method argument with `uplink.Url`:

```
class GitHub(object);
    @uplink.get
    def get_commit(self, commit_url: uplink.Url): pass
```

2.3.3 Path Variables

For both static and dynamic URLs, Uplink supports URI templates. These templates can contain parameters enclosed in braces (e.g., `{name}`) for method arguments to handle at runtime.

To map a method argument to a declared URI path parameter for expansion, use the `uplink.Path` annotation. For instance, we can define a consumer method to query any GitHub user's metadata by declaring the path segment parameter `{/username}` in the method's URL.

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

With an instance of this consumer, we can invoke the `get_user` method like so

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

to create an HTTP request with a URL ending in `users/prkumar`.

Implicit `Path` Annotations

When building the consumer instance, `uplink.build()` will try to resolve unannotated method arguments by matching their names with URI path parameters.

For example, consider the consumer defined below, in which the method `get_user()` has an unannotated argument, `username`. Since its name matches the URI path parameter `{username}`, `uplink` will auto-annotate the argument with `Path` for us:

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

Important to note, failure to resolve all unannotated function arguments raises an `InvalidRequestDefinitionError`.

2.3.4 Query Parameters

To set unchanging query parameters, you can append a query string to the static URL. For instance, GitHub offers the query parameter `q` for adding keywords to a search. With this, we can define a consumer that queries all GitHub repositories written in Python:

```
class GitHub(object):
    @uplink.get("/search/repositories?q=language:python")
    def search_python_repos(self): pass
```

Note that we have hard-coded the query parameter into the URL, so that all requests that this method handles include that search term.

Alternatively, we can set query parameters at runtime using method arguments. To set dynamic query parameters, use the `uplink.Query` and `uplink.QueryMap` argument annotations.

For instance, to set the search term `q` at runtime, we can provide a method argument annotated with `uplink.Query`:

```
class GitHub(object):
    @uplink.get("/search/repositories")
    def search_repos(self, q: uplink.Query)
```

Further, the `uplink.QueryMap` annotation indicates that an argument handles a mapping of query parameters. For example, let's use this annotation to transform keyword arguments into query parameters:

```
class GitHub(object):
    @uplink.get("/search/repositories")
    def search_repos(self, **params: uplink.QueryMap)
```

This serves as a nice alternative to adding a `uplink.Query` annotated argument for each supported query parameter. For instance, we can now optionally modify how the GitHub search results are sorted, leveraging the `sort` query parameter:

```
# Search for Python repos and sort them by number of stars.
github.search_repos(q="language:python", sort="stars").execute()
```

Note: Another approach for setting dynamic query parameters is to use *path variables* in the static URL, with "form-style query expansion".

2.3.5 HTTP Headers

To add literal headers, use the `uplink.headers` method annotation, which has accepts the input parameters as `dict`:

```
class GitHub(object):
    # This header explicitly requests version v3 of the GitHub API.
    @uplink.headers({"Accept": "application/vnd.github.v3.full+json"})
    @uplink.get("/repositories")
    def get_repos(self): pass
```

Alternatively, we can use the `uplink.Header` argument annotation to pass a header as a method argument at runtime:

```
class GitHub(object):
    @uplink.get("/users{/username}")
    def get_user(
        self,
        username,
        last_modified: uplink.Header("If-Modified-Since")
    ):
        """Fetch a GitHub user if modified after given date."""
```

Further, you can annotate an argument with `uplink.HeaderMap` to accept a mapping of header fields.

2.3.6 URL-Encoded Request Body

For `POST`/`PUT`/`PATCH` requests, the format of the message body is an important detail. A common approach is to url-encode the body and set the header `Content-Type: application/x-www-form-urlencoded` to notify the server.

To submit a url-encoded form with Uplink, decorate the consumer method with `uplink.form_url_encoded` and annotate each argument accepting a form field with `uplink.Field`. For instance, let's provide a method for reacting to a specific GitHub issue:

```
class GitHub(object):
    @uplink.form_url_encoded
    @uplink.patch("/user")
    def update_blog_url(
        self,
        access_token: uplink.Query,
        blog_url: uplink.Field
    ):
        """Update a user's blog URL."""
```

Further, you can annotate an argument with `uplink.FieldMap` to accept a mapping of form fields.

2.3.7 Send Multipart Form Data

Multipart requests are commonly used to upload files to a server.

To send a multipart message, decorate a consumer method with `uplink.multipart`. Moreover, use the `uplink.Part` argument annotation to mark a method argument as a form part.

Todo: Add a code block that illustrates an example of how to define a consumer method that sends multipart requests.

Further, you can annotate an argument with `uplink.PartMap` to accept a mapping of form fields to parts.

2.3.8 JSON Requests, and Other Content Types

Nowadays, many HTTP webservices nowadays accept JSON requests. (GitHub's API is an example of such a service.) Given the format's growing popularity, Uplink provides the decorator `uplink.json`.

When using this decorator, you should annotate a method argument with `uplink.Body`, which indicates that the argument's value should become the request's body. Moreover, this value is expected to be an instance of `dict` or a subclass of `uplink.Mapping`.

Note that `uplink.Body` can annotate the keyword argument, which often enables concise method signatures:

```
class GitHub(object):
    @uplink.json
    @uplink.patch("/user")
    def update_user(
        self,
        access_token: uplink.Query,
        **info: uplink.Body
    ):
        """Update an authenticated user."""
```

Further, you may be able to send other content types by using `uplink.Body` and setting the `Content-Type` header appropriately with the decorator `uplink.header`.

Index

P