beekeeper Documentation

Release 0.9.2

Jesse Shapiro

April 22, 2016

Contents

Contents:

CHAPTER 1

Why?

1.1 First Principles

In 2000, as his dissertation, Roy Fielding, one of the cofounders of the Apache open source HTTP server project, proposed a new standard for Internet-facing APIs. Rather than having proprietary communications protocols spewing complicated binary data back and forth, Fielding thought that communications should take place over HTTP, using HTTP standard forms of communication, and using representations of the state of the application, rather than actually transmitting the application's state. This document describes what we now know as REST APIs, and it is fundamental to how the Internet works today.

One of the key principles of REST APIs is that each request contains within itself all of the context needed to act on that request – and each response is similar. There are no "sessions" in REST APIs; a request sent one time should do exactly the same thing if sent again. This simplifies some things, but complicates others.

Another key principle written by Fielding in his paper was the concept of "Hypermedia as the Engine of Application State", or HATEOAS. The idea was that a client application could automatically "discover" all of the resources that an API provided, as well as the methods and variables exposed to work with those resources. If the state of the server changed, and one of those methods was no longer available, then the client library would become aware of that as part of the normal course of operations; similarly, a new resource would be immediately available for consumption.

1.2 Simplification

The use of idempotent operations (the same action should have the same result) and of context wholly contained within the request significantly simplifies serverside programming. Rather than maintaining a table full of active session information for its clients, a server needs only to listen for requests, and map them to the relevant data based on the query received. For example, a server that receives a "GET" request at the "Widgets" endpoint, with an ID parameter of 123 can simply know that it should query the Widgets table of its database and return the entry with an internal ID of 123.

This also simplifies things for the client. Rather than having to maintain an idea about "where" in the application it is, it can simply perform requests for particular pieces of information as needed, with the understanding that there won't be any undue side effects. When using a REST API, the application can be confident that, in general, requests don't need to be performed in any particular order.

1.3 Complication

While RESTful principles certainly make communication simpler, there are still some missing pieces. RESTful requests are made over HTTP; we know that. However, there are any number of different ways to do that, and almost

every single one has been used by some API, in some form or another. And, what's worse, HATEOAS has not been widely implemented, even though it would have solved many of these difficulties.

As a result, when working with RESTful APIs, developers have to spend significant periods of time interpreting exactly what the request for a particular resource ought to have in order to be properly interpreted by that particular API. Since the code becomes largely specific to the task at hand, it can't be reused for other purposes, even though the protocols for communication between various REST APIs are, at a base level, quite similar.

The developers of each API might also have a specific idea of what a REST API ought to look like, and might push that agenda by forming their API in a particular way, or with a particular style. This makes the issue even more difficult, by causing APIs to have less and less in common.

1.4 Existing Solutions

There are some solutions today that aim to make working with REST APIs simpler and easier. The largest of these in Python is the "Requests" library. Requests aims to simplify RESTful requests by condensing commonly-used resources into a single library, and stitching them together so that they can be used within a single line of code. For example:

```
resp = requests.get('domain.tld/api/resource').json()
```

The above code succinctly describes exactly what the developer wants to do - and then does it. It uses the HTTP GET method at the web address "domain.tld/api/resource", takes the response, and parses it from a string in JSON format into a native Python dictionary. Requests also provides other useful features, such as cookie persistence and automatic handling of URL parameters and headers.

However, Requests doesn't actually deal with any of the fundamental problems that are at play in modern REST APIs. Does it make it easier to work within the constraints of the existing system? Sure – but because the Requests library is as fundamentally stateless as the APIs it's interacting with, it has no way of eliminating them entirely. A variable that's defined as being passed in a URL parameter must be passed to Requests as a URL parameter, or the request won't work.

1.5 The Problem Remains

What's more, many modern REST APIs don't do a good job of descriptively converting their internal state hierarchy into a set of resource URIs that can be accessed programmatically. This is best exemplified by Wikipedia, whose REST API has but a single endpoint which accesses any resource according to the parameters passed to it:

```
https://en.wikipedia.org/w/api.php
```

To access any resources on the API, the developer has to delve into the (often-lacking) API documentation and figure out exactly what parameters are needed for which resources. She then has to build a class or method that encodes that information into a programmatic form for later use, because knowing that

```
https://en.wikipedia.org/w/api.php?format=json&action=query&prop=revisions&rvprop=content&titles=Wis
```

will return the article for the state of Wisconsin in JSON format is hardly useful or memorable. The Requests method isn't much better:

```
payload = {'format': 'json', 'action': 'query', 'prop': 'revisions', 'rvprop': 'content', 'titles':
resp = requests.get('https://en.wikipedia.org/w/api.php', params=payload).json()
```

This situation is made worse by the fact that a URL parameter is only one type of variable. A given API might not only require the developer to remember (or memorialize in code) that several different variables exist, but also which

of five or more variable types each is, some of which might be handled by Requests natively, but some others of which nmight take some manual labor to get working.

1.6 In Summary

There is a problem with modern REST APIs, and there's no easy solution available right now. Developers have to write thousands of lines of boilerplate code that doesn't do anything but re-implement existing code with slightly different arguments. What's more, because the developers writing that code aren't the ones who created the API in the first place, it's easy to make mistakes: mistakes that have to be fixed by delving into the codebase itself to fix them.

There has to be a better way than this.

CHAPTER 2

What?

2.1 Recap: The problem is state

The problem with existing solutions isn't that there's something fundamentally wrong with them from a technical perspective. They do exactly what they're designed to do, and they do it extremely well. No one is saying that Requests isn't an excellent HTTP client, or that its .json() parsing method isn't excruciatingly convenient. To say that would be to lie.

What's missing from Requests and solutions like it is *state*. Requests is commonly used to access RESTful APIs, but it doesn't actually know anything about those APIs. The developer making use of it has to feed it every variable manually with every request, and there's no clear mapping between a particular request and the actual resource on the remote server that it's trying to access.

Clearly, the solution to the problem isn't to make a "better Requests", or a "better urllib" or a "better cURL". The solution is a library that can both have knowledge about what a RESTful API looks like, and know how to map the variables, endpoints, and data types of that API into an easily-understandable object structure. This means it needs state.

2.2 Beekeeper: The answer

I've been a bit long-winded up to here, so I'll get to the point quickly. Beekeeper is a library that does exactly that. By consuming JSON files that describe the endpoints of an API, the variables that might be in play, and the way those endpoints relate to real objects and real actions on the remote server, beekeeper lets you build a client to work with an entire API with one of code:

```
wiki = API.from_hive_file('wikipedia.json')
```

And once you have that API client, it's just as easy to do things with it:

```
resp = wiki.Articles['Wisconsin'].get()
```

Once you're using beekeeper, you don't need to think in terms of endpoints, or parameters, or headers, or response data formats: you just need to think about the real objects that exist in the system you're trying to use.

2.3 But wait, there's more!

Of course, right now, to get the benefit of this, you typically have to write the JSON "hive" file to describe the API yourself. But, with a bit of help, and a bit of luck, that won't always be the case. The long-term goal is for beekeeper

to become so popular that API providers will write their own hive files and host them on their own websites. In the not-too-distant future, initializing an API interface with beekeeper could be as simple as this:

```
wiki = API.from_domain('wikipedia.org')
```

Simply by using beekeeper, you'll be able to access any API from any site, just by typing in the domain name. And, because the hive file is provided to your application directly by the server, it's constantly up to date with the latest version of the server's API. (Of course, if you want to keep using an older version of the hive, the specification provides for that as well.)

And, since JSON files are easily human-readable too, you can (if you want; you won't need to anymore) navigate to the hive file in your browser and take a look to see what goes into making that particular API work.

CHAPTER 3

How do I use it?

Using beekeeper is pretty simple, but because it can work with almost any RESTful API, it's also a little tricky to describe. Let's take a hypothetical API, for FooBar Ventures.

FooBar Ventures is in the business of widget manufacturing; their API provides tools to help their customers know what kinds of widgets are available, and gives detailed information about them. Each widget is compatible with a variety of products from other vendors, and FooBar Ventures also maintains a list of compatible products which can be accessed via the API.

First, install beekeeper:

```
$ pip install beekeeper
```

Then, from within Python, we'll need to import beekeeper and initialize the FooBar Ventures API:

```
>>> from beekeeper import API

>>> fbv = API.from_domain('foobar.com')
```

Note that if FooBar Ventures served their API over HTTP rather than over HTTPS, you'd need to set the "require_https" keyword argument to False to prevent beekeeper from raising an exception. Because hive files change the behavior of your application, secure transmission is really important. If you're possibly going to be passing sensitive information with your application, and the API provider doesn't host their hive using HTTP, it may be better for you to download their hive yourself, inspect it, include it with your application, and then initialize with a statement like this:

```
>>> fbv = API.from_hive_file(file_location)
```

You can also host the hive yourself securely, and initialize like this:

```
>>> fbv = API.from_remote_hive('https://mydomain.tld/fbv_hive.json')
```

During the initialization, if the API you're accessing requires any variables declared by the hive, you can pass those in as arguments or keyword arguments, and those values will be used on any future requests. This process is similar to what happens when executing a request - more on that later.

Then, let's say we want to get a list of all the widgets that FooBar makes:

```
>>> fbv.Widgets.list()

['RT6330', 'PV46', 'GX280']
```

We didn't need to pass any special variables to the API for this request outside of what's automatically handled already, so it's very simple. Beekeeper also handles parsing the returned data into a Pythonic format, so it's easy to iterate across and subscript into.

Now, I see one widget I think I'm interested in, called the GX280. Before going further, though, I want to make sure that it's compatible with my system, a HyperStar HS2000.

```
>>> fbv.Widgets['GX280'].compatiblilty_list()

{'manufacturers': {'Athena': {'CompatibleModels': ['AM4000', 'AM236', 'AM236b']}, 'HyperStar': {'Comp
```

Yikes, that's a big response. I could probably parse through it, but a), I'm kind of lazy, and b), maybe there's an easier way. You'll note something interesting about the request first, though; it has a dictionary-style subscription in the middle. This is because FooBar Ventures was kind enough, when they wrote their hive file, to define a ID variable for the Widget object. What this means is that if I know the ID for an object, I can easily get to that particular instance of an object, just by subscripting.

To deal with the response? I mentioned I'm a bit lazy, so I took a quick look at the API documentation, and it looks like FooBar provides a method to direcly check compatibility for a particular model. Let's do that instead:

```
>>> fbv.Widgets['GX280'].compatible_with('HS2000')

{'compatible': True, 'widgetModel': 'GX280', 'systemModel': 'HS2000'}
```

That's easier! Now, it looks like my system is compatible with that widget, so I want to take a closer look at it; make sure it's a good fit. I don't really care about other widgets at the moment, so I'm going to make it a bit easier by assigning the API object instance for the GX280 to its own variable:

```
>>> gx280 = fbv.Widgets['GX280']
```

Note that this isn't downloading any data; it's just binding all the actions that are associated with that particular object, and all the variables that need to be in place for those actions to work, to the name I picked. I can then use any actions as if I had typed out the whole long thing.

```
>>> gx280.description()

{'widgetModel': 'GX280', 'description': 'It's super cool!'}
```

GUYS, IT'S SUPER COOL. I MUST HAVE IT. I think I need 20 of them.

```
>> gx280.order(20)

TypeError: Expected values for variables: ['cc_number', 'quantity']
```

Oh. I guess they want to be paid.

Up until now, we've just been dealing with cases where we need to fill in one variable. When that's the case, beekeeper doesn't even make you tell it the variable name. But when we have more than one variable, you do need to fill that in. Let's try again:

```
>>> gx280.order(quantity=20, cc_number=1234234534564567)

{'status': 'OrderCreated', 'OrderNumber': 5960283}
```

There we go!

Note that I didn't actually need to fill in the name for "quantity". Because I filled in the name for "cc_number" (the only other required variable), beekeeper could have figured out that a variable out on its own without a name should go to the Quantity field. Or, vice versa. If I had filled in "quantity=20", beekeeper would have figured out that the other variable should go into "cc_number".

And that's all there is to using beekeeper! It's simple, fast, and makes working with remote APIs much, much, much easier.

If you're not sure what objects and actions are available for an API, you can easily see the structure by just doing the following:

```
>>> print(fbv)

FooBar Ventures()
|
|---Widgets[widget_id]
|   |   A widget, made by FooBar Ventures!
|   |
|   |---list()
|   |       Get a list of all widgets
|   |
|   |---compatibility_list(widget_id)
|   |       Get a list of systems compatible with the given widget
|   |
|   |---compatible_with(widget_id, system_id)
|   |       Is the system compatible with the widget?
|   |
|   |---description(widget_id)
|   |       Get a description of the widget
|   |
|   |---order(widget_id, cc_number, quantity)
|   |       Order the given quantity of the widget
```

It'll give you a nice printout so you can see where you need to go, and what variable values you need to get there.

CHAPTER 4

How does it work?

4.1 Variables

One of the big problems with existing REST client solutions is that they leave it to the developer to track what types of variables need to be put where. Some of these variable types can be handled automatically, while others need a bit of coaxing to work right. Beekeeper has the built-in ability to handle most of the common variable types, as long as the variable is defined properly within the construct of the Hive file (more details below).

The variable types that beekeeper currently supports are found below:

- HTTP forms
- Headers
- Body data
- URL string replacements
- URL parameters
- HTTP basic authentication
- HTTP bearer authentication
- Multipart/form-data
- Cookies

4.2 Data

Another item of concern is the way that data going back and forth between the client and the server is handled. Beekeeper handles this by having a number of parsers built in; if the data type being sent out matches up with one of the parsers, beekeeper will use it automatically to dump the Python object that's passed to it into a stream of bytes. On the way back, beekeeper reads the response's Content-Type header, and uses the information it finds there to parse the response back into a Python data structure; if it can't, it'll simply return the raw bytes to the developer for further use.

Right now, the MIME types that beekeeper has support for are as follows:

- application/json
- application/x-www-form-urlencoded
- application/octet-stream
- text/plain

- text/html
- application/xml
- text/xml

Note that any response can be handled in a GZipped format as well; like the above formats, this encoding is handled automatically by beekeeper.

4.2.1 Note

beekeeper uses the third-party xmltodict library to handle XML requests by default. Thus, if you're working with an XML API, be aware of the usage implications. You can also set a different XML parser by implementing a custom data handler.

4.3 Hive Location

To be automatically acquired by beekeeper when a developer passes your domain name (for example, facebook.com) to initialize it, your hive file must be located as below:

```
https://facebook.com/api/hive.json
```

It is STRONGLY recommended that you server your hive over HTTPS, as it does change the behavior of client applications. If you serve your hive over HTTP at this location, then, by default, beekeeper will raise an exception. This can be suppressed or handled, but it is not ideal.

You can make your hive accessible on any domain you control; it may be easier to just set up redirects for other domains, though.

Note that subdomains do count as separate domains, so "en.wikipedia.org" would have its own hive, separate from "wikipedia.org".

4.4 Parts of a Hive

Of course, the primary piece of information that beekeeper uses is the Hive file format. This is a JSON file with particular elements that tell beekeeper about the structure of endpoints in the remote API, as well as how those endpoints relate to the system's native objects and the actions that can be performed on those objects.

A hive has several elements; we're going to go into detail on each one. By the time you've read through this section, you should be ready to go out and start writing your own hive files - or ready to read others and understand exactly what they mean.

Note that if a key is not described here, it's not currently in use. This means that, at present, beekeeper won't try to do anything with it, but that's not a permanent promise in all cases.

4.4.1 name

The name of the hive. This is used when printing an API for documentation purposes.

4.4.2 description

Self-explanatory. Like the name key, it's only used when printing out an API, and is optional.

4.4.3 root

The base URL of the API. All API endpoints should exist under this URL; sometimes it's as simple as the domain of the website, and sometimes it can be more complex- for example, an API that might exist on several different subdomains might have a URL replacement entry in the root that can be filled out programmatically later on. By convention, the root URL should not have a closing slash; that should be placed at the beginning of the endpoint "path" keys lower-down.

4.4.4 mimetype

This key should be a valid MIME type; in general, it should be non-functional, but it comes into play if we're unable to extract an MIME type from a server response. It defaults to "application/json".

4.4.5 versioning

Example

```
{
    "versioning" : {
        "version": 8,
        "previousVersions": [
            "version": 7,
            "location": "http://domain.tld/api/hive_v7.json",
            "expires": "2016-12-31T12:00:00Z"
        ]
    }
}
```

The versioning key is completely optional. If you have multiple versions of your API, or if you iterate your API quickly, then it's good to note the current version of the API in the "version" subkey, and the details of any other currently active versions in the "previousVersions" list. An example of the "versioning" key is provided below.

Each item in the "previousVersions" list contains a version identifier, as well as a web path to a hive file that can be used to describe that version of the API. It may also contain an expiry date to indicate that that version of the API is in the process of deprecation, and will be shut off after a certain time. Once a version of the API has been deprecated, it should be removed from the hive file.

If a beekeeper API object is constructed with a version argument, beekeeper will automatically try to fetch the API version described by parsing the hive file it receives, determining if it matches the version given, and if not, loading from the appropriate URL, when available.

4.4.6 variables

The variables key contains any variables that are universally needed or used across the entire API, and which are best to fill when the API interface is constructed. Such variables are passed as arguments during construction, and will apply to every request thereafter, unless overridden manually.

The variables key is an object mapping variable names to variable objects; each variable object can have a number of keys, listed below, in order of how often you'll likely use them:

type

This key can bear a number of possible values describing the different kinds of variables that might be used. Some of them will have special caveats, noted below:

- **url_param** This is the default when a type for the variable is not specified and when a custom default variable type is not set on a hive. It appends a query string to the URL.
- **url_replacement** No caveats; this simply replaces any "format" blocks in the URL (as denoted by curly brackets around a variable name) with the variable's value.
- **http_basic_auth** Handles HTTP basic authorization using a username and password. When doing this, we expect to have variables named both "username" and "password"; if either is missing, beekeeper behaves as though it's an empty string.
- **header** Sets a header with the given name to the given value.
- **bearer_token** Handles HTTP authorization with a bearer token. The name of this variable is not used.
- **data** Sends data in the request body. Only one data-type variable is allowed in a given request.
- **multipart** Handles parsing any number of variables into a multipart/form-data request.
- **cookie** Sends a cookie to the server. By default, beekeeper will use cookies within a session automatically; it'll pull them from server responses, and send them back when needed, without additional definition. Explicitly defining a cookie (which is a single string; if you've got a cookie that's a name-pair value, you'll just need to pass in "name=value" as the string value) will disable any automatic cookie handling for that request and will only send those cookies that are explicitly defined.
- **http_form** Sends the key/value pair as part of an application/x-www-form-urlencoded request body to the server.

If a variable appears in multiple places, you can alternatively use the "types" key, which will let you use a list of different variable types; for example, a variable might need to both be a header and a URL parameter.

If a variable is of two different types in the same "tree", then when executed, it will act as both types. If it's one type by default higher up (by not having a specifically defined type), and then it's defined with a different type lower-down the tree, it'll only have that second type. Finally, if it's defined with one type higher in the tree, and then defined without an explicit type lower down, the original high-level type will remain in place without modification.

optional

This is a simple boolean True or False, defaulting to False if the key isn't present. If "optional" is false, as it is by default, and the variable doesn't receive a value when it needs to have one, then an exception will be raised, and you'll be prompted to fill in this variable, as well as any other variables missing values.

Whether a variable is optional is determined by the lowest-level explicit declaration of such. For example, a variable may be declared as optional for the API as a whole, but then may be explicitly declared as required on a specific endpoint or action.

value

The value key can be set to anything, as long as it's relevant to the variable type being used. Typically, though, you'll be using strings. This should typically not be set inside the hive unless it's being used to control behavior of the API; as an example, it's OK to set the value of an "action" variable to "login". It is **not** OK to set the value of a "password" variable to "hunter2".

Values will be filled in at two times; first, when initializing an API interface, and second, when calling a remote method.

When initializing the API, only variables at the API level will be filled, and will remain filled throughout the session.

When calling a remote method, variable values are not stored, and are only used in that specific request.

Values are determined by the most specifc copy of a variable to have a value explicitly set. From least-specific to most-specific, the levels are API, endpoint, and then action. A higher-level value may be "un-set" by passing a None value during execution of a request (after loading the hive into beekeeper) or by setting a lower-level "null" value within the hive (when writing the JSON file).

mimetype

This key is only used in data-type variables; at present, the "data" type, and the "multipart" type. It's used to determine what parser to use to translate the data into binary before transmission, and how to set the Content-Type header. If not present on "multipart"-type variables, then that specific variable is assumed to be a standard form field variable, rather than data.

Like the value key, mimetype is determined by the lowest-level explicitly declared variable.

filename

This key is only used in the "multipart" variable type; because data, in the context of "multipart" submissions, is assumed to be a file, it may be necessary to set the name of that virtual file to a specific name. If this filename is not defined within the hive, then one of two things will happen. If the object passed to the data handler is a file-like object with a "name" attribute, the value of that attribute will be used. Otherwise, a random filename in UUID form will be assigned.

Like the value key, filename is determined by the lowest-level explicitly declared variable.

name

Sometimes, it's desirable to have the Python name of a variable be different from the API name of that variable. In cases like this, you can set the optional "name" key to have a different string value. If you do so, then within your programming, you'll address this variable using the name it's keyed by in the variables object, but external requests will use the value found in this subkey.

Note that if one of the variables defined in the hive is keyed by a reserved name in Python, the keyed name will be transferred into the "name" key, and an underscore will be added to the key used to access that variable within beekeeper. For example, if a variable is named "from", then to call it, the developer will need to access it as "_from", but it'll still be sent to the remote server with the appropriate name. This is also the case for objects and actions with reserved names.

Like the value key, name is determined by the lowest-level explicitly declared variable.

Example

```
{
    "variables": {
        "FileSubmission": {
            "type": "multipart",
            "optional": false,
            "value": {"key1": "val1", "key2": "val2"},
            "mimetype": "application/json",
            "filename": "myupload.json",
            "name": "OtherFileSubmissionName"
        },
```

```
        "SimpleDefaultedUrlParam": {
        }
    }
}
```

4.4.7 variable_settings

Beekeeper can be configured to have different behaviors around variables. The variable_settings key can be used to do just that. It's optional, and can contain two main keys; first, a default_type key that sets what beekeeper is going to do when it encounters either undefined variables, or a variable with an undefined type. Second, it can contain a custom_types object that has information about the custom variable types that the hive uses. If a handler for each of these types isn't present at initialization of the hive into an API, an exception is raised. Note that the content of each custom_types key isn't mandated, but it should be descriptive, and ideally provide the reader with information about how to get or create the handler.

Example

```
{
    "variable_settings": {
        "default_type": "url_replacement",
        "custom_types": {
            "special_var": {
                "description": "Helps with doing cool stuff!",
                "web_url": "http://mydomain.com/special_var_handler"
            }
        }
    }
}
```

4.4.8 endpoints

The Endpoints key contains definitions of the various different resources available on the API by distinct URLs. The name of these endpoints isn't hugely important, as they're not used on a user-facing level. However, they should still have fairly descriptive names, so that a developer reading your hive file can quickly determine what's happening.

Each Endpoint object contains four primary keys:

description

This is an optional key that's only used when printing out an API.

path

The path key is mandatory; it describes the URL of the endpoint in relation to the "root" path given at the API level. Like the root, it may contain URL replacement handlers (variable names within brackets). If you have a number of objects that use syntactically similar paths, it may be useful to define a URL replacement here so that you can use the same endpoint for different objects, and avoid writing the same JSON multiple times.

methods

methods is a list of the HTTP methods that may be used on this endpoint. By default, if no value is given, it is assumed to be a list with a single string "GET". When executing a request, if the HTTP method that's being used isn't one allowed by this key, then an exception will be raised.

variables

variables is an optional key, like at the API level, which contains definitions of variables that are specific to requests on this endpoint. If URL replacements are being used on this endpoint, it's best to define them here so that appropriate errors can be raised if they're missing.

Example

```
{
    "endpoints": {
        "SingleObjectByID": {
            "path": "/{object_type}/{object_id}",
            "methods": [
                "GET",
                "PUT",
                "DELETE"
            ],
            "variables": {
                "object_type": {
                    "type": "url_replacement"
                },
                "object_id": {
                    "type": "url_replacement"
                }
            }
        }
    }
}
```

4.4.9 objects

The Objects key is at the heart of how beekeeper works. Rather than simply handing the developer a list of endpoints, the Objects key allows beekeeper to define the relationship between HTTP endpoints and the actual objects that they represent on the server. It also defines the actions that can be used on those objects.

When an API is initialized in beekeeper, items listed in Objects will be available as attributes on the parent API object, so names should be chosen carefully, and should be solely related to the object itself, rather than to the actions that can be taken with them.

Objects that can be subscripted (more on that later) should be named plurally so that the idea of them as dictionaries to be opened can be thought of more naturally. For example, if the name of a single object is "Widget", the key to that object in the Objects key of the hive should be "Widgets".

As with all objects described so far, any given object will have several keys:

description

This optional key is only used when printing out the API.

id_variable

The id_variable key is a string that defines which variable is filled in when subscription is used with this object type. If the key is not present, the object is not subscriptable.

actions

The "actions" object contains any number of actions that can be taken based on the given object. The actions contained therein define the abstraction between Pythonic object-action pairings and the endpoint-method pairings used by the remote API.

As with a given object, great thought should go into naming these actions. They will be used directly by developers when handling your API, so names should be concise and to the point.

When deciding which Objects to place a given action in, it's best to consider what object the action is being based off, rather than what type of object the action ought to return.

For example, if there's an endpoint that gives a list of the Color objects available with a particular widget, that should exist as the "colors" action on the "Widgets" object, rather than as the "available_options_by_widget" action on the Colors object. If in doubt, ask yourself, "what object is the ID variable I provide this method associated with?"

For fear of repeating myself, as with everything so far, each action has several subkeys:

description

This optional key is only used when printing out the API.

endpoint

The endpoint key is a string referring to the name of the endpoint that will be used when the action is called.

method

The method key is an optional string that defines which HTTP method will be used to hit the given endpoint. If no method is given, the action will default to attempting an HTTP GET.

timeout

The optional timeout key is a number that defines the amount of time beekeeper will wait on data to come from a socket before raising a timeout exception that you can use to retry the request. The default is five seconds.

variables

The variables key here is identical to the variables object that exists on the API and Endpoint levels. In practice, of course, you'll use it for different purposes. For example, if your Action accesses an endpoint that needs to have variables filled in to make it fit for a particular object, the best place to do it is here. You may also need to set other parameters that are specific to the given action.

traverse

The Traverse key here lets you define, on a particular action, which parts of the response data should actually be provided to the program. In some specific cases, it may be useful to pare down the response to specific components.

The Traverse key is a list; each item in that list can be either a string or a list of strings. We start out with the return data in dictionary form, and proceed recursively through the traversal path. For each item in the path, we'll do one of several things:

- If the object we've currently recursed to is a list, we'll return a list of each item, each traversed with the remaining elements of the path.
- If the top item in the path is a list, we'll return a dictionary, with one key for each item in the list. The value of each key in the dictionary will be the traversed value of the item for that key in the object that we're currently recursed to.
- If the top item in the path is a string with value "*", we'll act similarly to what we would do if the top item in the path was a list, but instead of looking at just specific keys, we'll return every key in the current object.
- If the top item in the path is any other string, we'll continue recursively navigating through the dictionary entry with that particular key.

In general, if we reach a node where it isn't possible to navigate to the next path item, we'll raise a TraversalError that contains information about the current path item as well as the state of the object that we've traversed to. The exception is if the previous operation was to split a dictionary (as in the case with a list-type path item, or with a wildcard "*" path item). In this case, if, one of the objects addressed in such a split is not a normally traversable object (a type that inherits from either a dictionary or a list), then we'll just return that object, rather than raising a further exception.

Example

```
{
    "Widget": {
        "description": "A widget!",
        "id_variable": "object_id",
        "actions": {
            "get": {
                "endpoint": "SingleObjectByID",
                "variables": {
                    "object_type": {
                        "value": "widget"
                    }
                }
            },
            "update": {
                "endpoint": "SingleObjectByID",
                "method": "PUT",
                "variables": {
                    "object_type": {
                        "value": "widget"
                    },
                    "widget": {
                        "type": "data",
                        "mimetype": "application/json"
                    }
                }
            },
            "delete": {
                "endpoint": "SingleObjectByID",
```

```
                "method": "DELETE",
                "variables": {
                    "object_type": {
                        "value": "widget"
                    }
                }
            },
            "list": {
                "endpoint": "ListObjectInstances",
                "method": "GET",
                "variables": {
                    "object_type": {
                        "value": "widget"
                    }
                },
                "traverse": [
                    "data",
                    "results",
                    "widgets"
                ]
            }
        }
    }
}
```

CHAPTER 5

Advanced Usage

beekeeper is designed to be easy-to-use, but its structure also makes it incredibly adaptable and powerful for advanced users. Right now, there are three key advanced feature that you may want to take advantage of.

5.1 Methods as Variables

When initializing a hive, you can pass callable objects like methods and functions into beekeeper to use as values for variables, as long as there aren't any variables that need to be used to call them. To avoid this, you can do a few different things. First, and most easily, they can be object instance methods. beekeeper will keep those methods linked to their original contexts, and so their results will be based on the state of their parent object. You can also use alternate forms of callables, like Python's built-in "partial" object, which lets you fill in variable values ahead of time.

Either way you do it, you let your beekeeper-generated API move from a simple static platform to a more robust and dynamic system. Perhaps the most common use of this feature would be to tie into an OAuth authentication scheme, or some other system that requires credentials to change automatically over time.

To set this up, just pass in an uncalled method or function as a variable when you initialize the API. Every time you make an API call, your method will be executed, and the returned value will be used as the value for that variable.

5.2 Custom Data Handlers

beekeeper has built-in support for a number of different data types, and automatically chooses between them based on the defined MIME type of data being sent, or the Content-Type header of the data being received. Right now, we can read JSON, plaintext, binary streams, and XML.

But, you might decide that you need something different. For example, you might want to automatically parse a "text/csv" response into a set of lists, or you might want to do that the other way around to post "text/csv" data to a server.

First, you'll need to define a data handler class. It should have at least one of two possible static methods; "dump" and "load". "dump" takes a Python object and encodes it to a bytes object in the appropriate format; "load" does exactly the opposite.

You should inherit your class from beekeeper.DataHandler; this will automatically load it into beekeeper without any further action on your part. To make sure that it's handling the right data, you'll need to set at least one of two class variables; *mimetype* or *mimetypes*; *mimetype* should be a single string with the MIME type you want your class to handle, while if your class can handle multiple MIME types (as in the case of an XML parser that handles both 'application/xml' and 'text/xml'), you'll set *mimetypes* to be a list of those MIME types.

All told, it should look something like this:

```
class CSVHandler(beekeeper.DataHandler):

mimetype = 'text/csv'

@staticmethod
def dump(python_object, encoding):
    """
    Logic goes here - take the Python object the method receives, and parse it
    into a bytes() object. Be sure to use the text encoding passed to the
    "encoding" argument.
    """

@staticmethod
def load(response, encoding):
    """
    Again, logic goes here. beekeeper will pass you a bytes() object, as well as the
    encoding the bytes were sent in, and will expect to receive in response a Python
    object that's relevant to the data received.
    """
```

Because you've informed beekeeper about the specific MIME type that the data handler should be associated with, beekeeper now knows exactly when to use it: when you define a data variable that has the defined MIME type of "text/csv", or when a response is received from the server with "text/csv" in the "Content-Type" header. If a MIME type doesn't have a data handler associated with it, beekeeper will just return the raw bytes received.

5.3 Custom Variable Types

Sometimes, you have a variable that's a little particular in its needs, and which you might want to make a bit easier to use. To do that, you can define a custom variable type and handler to make things a bit simpler.

For example, when updating contact properties using the Hubspot API, a JSON object in the following format is required:

```
{
    "properties": [
        {
            "property": "firstname",
            "value": "John"
        },
        {
            "property": "lastname",
            "value": "Smith"
        }
    ]
}
```

It's a little bit verbose. And the whole goal of beekeeper is to make your life easier, so you can put in a little work to make it easier still, just by defining a custom variable handler and sticking it into beekeeper.

A variable handler takes keyword arguments of the defined type, processes them, and sets "final variables" of one of the types native to HTTP requests. Those four types are as follows:

- "url_param"
- "header"
- "url_replacement"

- "data"

Beekeeper is designed to be able to handle (on a structural level; not necessarily with built-in code) pretty much any variable type you can throw at it, as long as it can be simplified into those four variable types. The way it does this is by passing the request object along with the request to parse a variable; the function that eventually handles the variable can then decide how to apply the necessary changes to the request.

This is done via four callback methods on the Request object:

- set_headers(**headers)
- set_data(data)
- set_url_params(**params)
- set_url_replacements(**replacements)

Each of these callback methods can take any number of keyword arguments paired with the final values for those variables. The exception is the *set_data()* method, which can take a single value, since each HTTP request can only have a single request body (to get around this, use the multipart variable type).

You can also use the *beekeeper.render_variables* method if your data needs more processing as one of the built-in types.

Now that we've got some principles down, let's look at our original case. We want a simpler way to write Hubspot contacts, so let's implement a custom variable type to handle getting them into the right format:

```
@beekeeper.VariableHandler('hs_contact')
def hubspot_contact_handler(rq, **values):
    #Typically, because this is a data-type object, we only receive one variable.
    for _, contact in values.items():
        x = {
                'properties': [
                    {'property':prop, 'value': val} for prop, val in contact.items()
                ]
            }
        beekeeper.render_variables(rq, 'data', data={'value': x, 'mimetype': 'application/json'})
```

Note the *beekeeper.VariableHandler('hs_contact')* decorator. This decorator wraps up your function and automatically attaches it to any variable types that you include in the decorator parameters. You can use a custom variable name, like we did here, or you can bind a custom handler to a built-in variable type by using its name.

This simple function will perform the transformation we're looking for (we can simply pass in a dictionary containing the new variable values), and then pass it into the data-rendering pipeline, which will handle setting both the body data we need, and the appropriate "Content-Type" header. Note that there isn't a return statement; this is because each function applies its settings directly to the request.

If you're writing a hive for general distribution, carefully consider the implications of using custom variable types. Unlike custom data types, beekeeper has no way to handle hives that use custom variables unless a handler has been bound. Thus, it's best to create two versions of a hive; one that uses the custom handlers you want, and one that uses only the standard variable types. You can then use the versioning data in the standard hive to point to the customized hive in an opt-in manner for consumers who have either implemented or downloaded an appropriate variable handler.

CHAPTER 6

Readme

6.1 Description

beekeeper is a Python library designed around dynamically generating a RESTful client interface based on a minimal JSON hive.

The hive specification is designed to provide beekeeper (or other applications consuming hive files) with programmatically-designed insight into the structure of both the REST endpoints that are available and the objects and methods that those endpoints represent.

While the classes available in beekeeper can be used manually to create Pythonic representations of REST endpoints, it is strongly preferred that the library be used as a whole with a constructed hive file. As APIs become larger in scale (in terms of the number of endpoints and represented objects), the time benefit of beekeeper becomes more pronounced, as adding additional objects and endpoints is a trivial process.

6.2 Requirements

beekeeper requires Python 2.7.9/3.4.3 or higher and their built-in modules, as well as xmltodict.

6.3 Installation

```
pip install beekeeper
```

6.4 Usage

The usage of beekeeper will depend on what features are provided by the person who wrote the hive file. There are a number of ways that the hive writer can make your life easier. Regardless, at a base level, usage will look something like this:

```
from beekeeper import API
myAPI = API.from_hive_file('fname.json')
x = myAPI.Widgets.action(id='foo', argument='bar')
```

If the hive developer defines an ID variable for the object you're working with, you can subscript, dictionary style:

```
x = myAPI.Widgets['foo'].action(argument='bar')
```

If you've only got one remaining argument in the method call, you don't even need to name it! You can do something like this:

```
x = myAPI.Widgets['foo'].action('bar')
```

This also holds true if you have multiple variables, but the other ones are assigned by name:

```
x = myAPI.Widgets['foo'].action('bar', var2='baz')
```

If you're using a hive file, then it should define which variables are needed. If you try to call a function without filling in that variable, it should automatically yell at you and tell you what variables are missing. Since these variables are defined within the hive, beekeeper will do the work for you, automatically determine what data type a particular variable is, and put it exactly where it needs to go.

beekeeper will also automatically handle parsing data. When you send data, beekeeper will read the MIME type that was defined in the variable for that data, and try to automatically move it from a "Python" format (e.g., a dictionary) to the right REST API format (e.g., JSON).

This holds true in the other direction as well; beekeeper will read the MIME type of the response data, and hand it back to you in a Pythonic format! If beekeeper doesn't know how to handle the data, it'll just give you the raw bytes so that you can do what you need to with them.

6.5 Notes

beekeeper does not currently do SSL certificate verification when used on Python versions earlier than 2.7.9 or 3.4.3.