restcli Documentation

Release 0.1.0

Dustin Rohde

Mar 10, 2021

CONTENTS

CHAPTER

ONE

OVERVIEW

In this section we'll get a bird's eye view of **restcli**'s core concepts. After reading this section, you should be ready for the *Tutorial*.

1.1 Collections

restcli understands your API through YAML files called *Collections*. Collections are objects composed of *Groups*, which are again objects composed of *Requests*. A Collection is essentially just a bunch of Requests; Groups are purely organizational.

```
---
weapons:
    equip:
        # <<request>>
    info:
        # <<request>>
potions:
    drink:
        # <<request>>
```

This Collection has two Groups. The first Group, `weapons`, has two Requests, `equip` and `info`. The second has Group is called "potions" and has one Request called "drink". This is a good example of a well-organized Collection — Groups were used to provide context, and even though we're using placeholders, it's easy to infer the purpose of each Request.

1.2 Requests

A Request is a YAML object that describes a particular action against an API. Requests are the bread and butter of **restcli**.

```
method: post
url: "http://httpbin.org/post"
headers:
    Content-Type: application/json
    Authorization: {{ password }}
body: |
    name: bar
    age: {{ cool_number }}
    is_cool: true
```

At a glance, we can get a rough idea of what's going on. This Request uses the POST `method` to send some data (`body`) to the `url` http://httpbin.org/post, with the given Content-Type and Authorization `headers`.

Take note of the stuff in between the double curly brackets: `{{ password }}`, `{{ cool_number }}`. These are template variables, which must be interpolated with concrete values before executing the request, which brings us to our next topic...

1.3 Environments

An Environment is a YAML object that defines values which are used to interpolate template variables in a Collection. Environments can be be modified with *scripts*, which we cover in the *Tutorial*.

This Environment could be used with the Request we looked at in the *previous section*:

```
password: sup3rs3cr3t
cool_number: 25
```

Once the Environment is applied, the Request would look something like this:

```
method: post
url: "http://httpbin.org/post"
headers:
    Content-Type: application/json
    Authorization: sup3rs3cr3t
body: |
    name: bar
    age: 25
    is_cool: true
```

1.3.1 Next Steps

The recommended way to continue learning is the *Tutorial*.

CHAPTER

TWO

USAGE

restcli is invoked from the command-line. To display usage info, supply the `--help` flag:

```
$ restcli --help

Usage: restcli [OPTIONS] COMMAND [ARGS]...

Options:
  -v, --version               Show the version and exit.
  -c, --collection PATH       Collection file.
  -e, --env PATH              Environment file.
  -s, --save / -S, --no-save  Save Environment to disk after changes.
  -q, --quiet / -Q, --loud    Suppress HTTP output.
  --help                      Show this message and exit.

Commands:
  env   View or set Environment variables.
  exec  Run multiple Requests from a file.
  repl  Start an interactive prompt.
  run   Run a Request.
  view  View a Group, Request, or Request Parameter.
```

The available commands are:

Command: run Run a Request.

Command: exec Run multiple Requests from a file.

Command: view Inspect the contents of a Group, Request, or Request attribute.

Command: env View or set Environment variables.

Command: repl Start the interactive prompt.

To display usage info for the different commands, supply the `--help` flag to that particular command.

2.1 Command: run

The `run` command is documented on its own page, in *Making Requests*.

2.2 Command: exec

```
$ restcli exec --help

Usage: restcli exec [OPTIONS] FILE

  Run multiple Requests from a file.

  If '-' is given, stdin will be used. Lines beginning with '#' are ignored.
  Each line in the file should specify args for a single "run" invocation:

      [OPTIONS] GROUP REQUEST [MODIFIERS]...

Options:
  --help  Show this message and exit.
```

The `exec` command loops through the given file, calling `run` with the arguments provided on each line. For example, for the following file:

```
# requests.txt
accounts create -o password:abc123
accounts update password==abc123 -o name:foobar
```

These two invocations are equivalent:

```
$ restcli exec requests.txt
```

```
$ restcli run accounts create -o password:abc123
$ restcli run update password==abc123 -o name:foobar
```

2.3 Command: view

```
$ restcli view --help

Usage: restcli view [OPTIONS] GROUP [REQUEST] [PARAM]

  View a Group, Request, or Request Parameter.

Options:
  -r, --render / -R, --no-render  Render with Environment variables.
  --help                          Show this message and exit.
```

The `view` command selects part of a Collection and outputs it as JSON. It has three forms, described here with examples:

Group view Select an entire Group, e.g.:

```
$ restcli view chordata
```

```
{
  "mammalia": {
    "headers": {
      ...
```

(continues on next page)

(continued from previous page)

```
    },
    "body": ...,
    ...
  },
  "amphibia": {
    ...
  },
  ...
}
```

Request view Select a particular Request within a Group, e.g.:

```
$ restcli view chordata mammalia
```

```
{
  "url": "{{ server }}/chordata/mammalia"
  "method": "get",
  "headers": {
    "Content-Type": "application/json",
    "Accept": "application/json",
  }
}
```

Request Attribute view Select a single Attribute of a Request, e.g.:

```
$ restcli view chordata mammalia url
```

```
"{{ server }}/chordata/mammalia"
```

The output of `view` is just plain JSON, which makes it convenient for scripts that need to programmatically analyze Collections in some way.

Use the `--render` flag to render template variables, e.g.:

```
$ restcli view --render chordata mammalia url
```

```
"https://animals.io/chordata/mammalia"
```

2.4 Command: env

Todo: Write this section

2.5 Command: repl

```
Usage: [OPTIONS] COMMAND [ARGS]...

Options:
  -v, --version               Show the version and exit.
  -c, --collection PATH       Collection file.
  -e, --env PATH              Environment file.
  -s, --save / -S, --no-save  Save Environment to disk after changes.
  -q, --quiet / -Q, --loud    Suppress HTTP output.
  --help                      Show this message and exit.

Commands:
  change_collection  Change to and load a new Collection file.
  change_env         Change to and load a new Environment file.
  env                View or set Environment variables.
  exec               Run multiple Requests from a file.
  reload             Reload Collection and Environment from disk.
  run                Run a Request.
  save               Save the current Environment to disk.
  view               View a Group, Request, or Request Parameter.
```

The `repl` command starts an interactive prompt which allows you to issue commands in a read-eval-print loop. It supports the same set of commands as the regular commandline interface and adds a few repl-specific commands as well.

CHAPTER

THREE

MAKING REQUESTS

```
$ restcli run --help

Usage: restcli run [OPTIONS] GROUP REQUEST [MODIFIERS]...

  Run a Request.

Options:
  -o, --override-env TEXT  Override Environment variables.
  --help                   Show this message and exit.
```

The `run` command runs Requests from a Collection, optionally within an Environment. It roughly executes the following steps:

1. Find the given Request in the given Collection.
2. If `defaults` are given in a Config Document, use it to fill in missing parameters in the Request.
3. If an Environment is given, apply any overrides to it.
4. Render the Request with Jinja2, using the Environment if given.
5. Apply any modifiers to the Request.
6. Execute the Request.
7. If the Request has a `script`, execute it.
8. If `save` is true, write any Environment changes to disk.

Examples:

```
$ restcli -s -c food.yaml -e env.yaml run recipes add -o !foo

$ restcli -c api.yaml run users list-all Authorization:abc123
```

3.1 Environment overrides

When running a Request, the Environment can be overrided on-the-fly with the `-o` option. It supports two types of arguments:

`KEY:VALUE` Set the key `KEY` to the value `VALUE`.

`!KEY` Delete the key `KEY`.

The `-o` option must be specified once for each argument. For example, the following `run` invocation will temporarily set the key `name` to the value `donut` and delete the key `foo`:

```
$ restcli -c food.yaml -e env.yaml run recipes add \
          -o name:donut \
          -o !foo
```

3.2 Request modifiers

In addition to Environment overrides, the Request itself can be modified on-the-fly using a special modifier syntax. In cases where an Environment override changes the same Request parameter, modifiers always take precedence. They must appear later than other options.

Each modifier has a mode and a parameter. The *operation* describes the thing to be modified, and the *mode* describes the way in which it's modified.

Generally, each modifier is written as a commandline flag, specifying the *mode*, followed by an argument, specifying the *operation*. In the following example modifier, its *mode* specified as `-n` (**assign**) and its *operation* specified as `foo:bar`:

```
-n foo:bar
```

Modifiers may omit the *mode* flag as well, in which case *mode* will default to **assign**. Thus, the following modifiers are equivalent:

```
-a foo:bar -n baz=quux
-a foo:bar baz=quux
```

3.2.1 Syntax

The general syntax of modifiers is described here:

```
modifiers      ::=   (mod_append | mod_assign | mod_delete)*
mod_assign     ::=   "-n" operation | operation
mod_append     ::=   "-a" operation
mod_delete     ::=   "-d" operation
operation      ::=   "'" op "'" | '"' op '"'
operation      ::=   op_header | op_query | op_body_str | op_body_nostr
op_header      ::=   <ASCII text> ":" [<ASCII text>]
op_query       ::=   <Unicode text> "==" [<Unicode text>]
op_body_str    ::=   <Unicode text> "=" [<Unicode text>]
op_body_nostr  ::=   <Unicode text> ":=" [<Unicode text>]
```

3.2.2 Modifier modes

There are three modifier modes:

assign Assign the specified value to the specified Request parameter, replacing it if it already exists. This is the default. If no *mode* is specified for a given *modifier*, its *mode* will default to **assign**.

If a header `X-Foo` were set to `bar`, the following would change it to `quux`:

```
$ restcli run actions get -n X-Foo:quux
```

Since **assign** is the default mode, you can omit the `-n`:

```
$ restcli run actions get X-Foo:quux
```

append Append the specified value to the specified Request parameter. This behavior differs depending the type of the Request parameter.

If its a *string*, concenate the incoming value to it as a string. If a string field `nickname` were set to `"foobar"`, the following would change it to `"foobar:quux"`.

```
$ restcli run actions post -a nickname=':quux'
```

If its a *number*, add the incoming value to it as a number. If a json field `age` were set to `27`, the following would change it to `33`.

```
$ restcli run actions post -a age:=6
```

If its an *array*, concatenate the incoming value to it as an array. If a json field `colors` were set to `["red", "yellow"]`, the following would change it to `["red", "yellow", "blue"]`.

```
$ restcli run actions post -a colors:='["blue"]'
```

Other types are not currently supported.

Todo: Add validation for other types.

---

delete Delete the specified Request parameter. This ignores the value completely.

If a url parameter `pageNumber` were set to anything, the following would remove it from the url query completely.

```
$ restcli run actions get -d pageNumber==
```

Todo: Rename `append` mode to `add` and maybe `assign` to `set` or `replace`.

Table 2: Table of modifier modes

Mode	Flag	Usage
assign	`-n`	`-n OPERATION`
append	`-a`	`-a OPERATION`
delete	`-d`	`-d OPERATION`

3.2.3 Modifier operations

Operations

header Operators on a header key-value pair. The *key* and *value* must be valid ASCII. Delimited by `:`.

url param A URL query parameter. Delimited by ==.

string field A JSON object key-value pair. The *value* will be interpreted as a string. Delimited by =.

json field A JSON object key-value pair. The *value* will be interpreted as a string. Delimited by :=.

Table 3: Table of modifier operations

Operation	Delimiter	Usage	Examples
header	:	• `KEY : VALUE` • `KEY :`	• `Authorization:abc` • `Authorization:`
url param	==	• `KEY == VALUE` • `KEY ==`	• `locale==en_US` • `locale==`
string field	=	• `KEY = VALUE` • `KEY =`	• `username=foobar` • `username=`
json field	:=	• `KEY := VALUE` • `KEY :=`	• `age:=15` • `age:=`

3.2.4 Examples

To follow along with the examples, grab the simple example project from the **restcli** source. Then from the example directory, export some environment variables to use the example project's Collection and Environment files:

```
$ export RESTCLI_COLLECTION="simple.collection.yaml"
$ export RESTCLI_ENV="simple.env.yaml"
```

To check your work after each **restcli run** invocation, just inspect the response. All the Requests in this Collection will respond with a JSON blob containing the information about your HTTP request, like this:

```
$ restcli run actions get
```

```
// HTTP response

{
    "args": {
        "fooParam": "10"
    },
    "headers": {
        "Accept": "application/json",
        "Accept-Encoding": "gzip, deflate",
        "Connection": "close",
```

(continues on next page)

(continued from previous page)

```
        "Host": "httpbin.org",
        "User-Agent": "HTTPie/0.9.9",
        "X-Foo": "foo+bar+baz"
    },
    "origin": "75.76.62.109",
    "url": "https://httpbin.org/get?fooParam=10"
}
```

Example 1

Delete the header `"Accept"`.

```
$ run actions get -d Accept:
```

Example 2

Append the string `"420"` to the body value `"nickname"`.

```
$ run actions post -a time=420
```

Example 3

Assign the array `'["red", "yellow", "blue"]'` to the body value `"colors"`.

```
$ run actions post -n colors:='["red", "yellow", "blue"]'
```

CHAPTER

FOUR

TUTORIAL: MODELING AN API

Note: This tutorial assumes that you've read the *Overview* and *Usage* documentation.

Throughout this tutorial we will be modeling an API with **restcli**, gradually adding to it as we learn new concepts, until we have a complete API client suite. While the final result will be pasted at the end, I encourage you to follow along and do it yourself as we go. This will give you many opportunities to experiment and learn things you may not have learned otherwise!

4.1 Debriefing

You have been commissioned to build an API for the notorious secret society, the Sons of Secrecy. You were told the following information, in hushed whispers:

1. New members can join by invite only.
2. Each member has a rank within the Society.
3. Your rank determines how many secrets you are told.
4. Only the highest ranking members, called Whisperers, have the ability to recruit and promote members through the ranks.

Your task is to create a membership service for the Whisperers to keep track of and manage their underlings. Using the service, Whisperers must be able to:

1. Invite new members.
2. Promote or demote members' ranks.
3. Send "secrets" to members.

In addition, the service must be guarded by a secret key, and no requests should go through if they do not contain the key.

Let's get started!

4.1.1 Requests

We'll start by modeling the new member invitation service:

```
# secrecy.yaml
---
memberships:
    invite:
        method: post
        url: "{{ server }}/memberships/invite"
        headers:
            Content-Type: application/json
            X-Secret-Key: '{{ secret_key }}'
        body: |
            name: {{ member_name }}
            age: {{ member_age }}
            can_keep_secrets: true
```

We made a new Collection and saved it as `secrecy.yaml`. So far it has one Group called `memberships` with one Request called `invite`.

As requested, we've also added an `X-Secret-Key` header which holds the secret key. It's parameterized so that each Whisperer can have their own personal key. This will be explained later in the *templating* section.

Request Parameters

Let's zoom in a bit on Requests. While we're at it, we'll inspect our `invite` Request more closely as well.

`method` (string, required) HTTP method to use. Case insensitive.

> We chose POST as our method for `invite` since POST is generally used for creating resources. Also, per RFC 7231, the POST method should be used when the request is non-idempotent.

`url` (string, required, templating) Fully qualified URL that will receive the request. Supports *templating*.

> We chose to parameterize the `scheme://host` portion of the URL as `{{ server }}`. As we'll see later, this makes it easy to change the host without a lot of labor, and makes it clear that the path portion of the URL, `/memberships/invite`, is the real subject of this Request.
>
> We'll learn more about template variables later, but for now we know that invitations happen at `/send_invite`.

`headers` (object, ~templating) HTTP headers to add. Keys and values must all be strings. Values support *templating*, but keys don't.

> We're using the standard `Content-Type` header as well as a custom, parameterized header called `X-Secret-Key`. We'll inspect this further in the *templating* section.

`body` (string, templating) The request body. It must be encoded as a string, to facilitate the full power of Jinja2 *templating*. You'll probably want to read the section on *YAML block style* at some point.

> The body string must contain valid YAML, which is converted to JSON before sending the request. Only JSON encoding is supported at this time.
>
> Our `body` parameter has 3 fields, `name`, `age`, and `can_keep_secrets`. The first two are parameterized, but we just set the third to `true` since keeping secrets is pretty much required if you're gonna join the Sons of Secrecy.

`script` (string) A Python script to be executed after the request finishes and a response is received. Scripts can be used to dynamically update the *Environment* based on the response payload. We'll learn more about this later in *scripting*.

Our `invite` Request doesn't have a script.

4.1.2 Templating

restcli supports Jinja2 templates in the `url`, `headers`, and `body` Request Parameters. This is used to parameterize Requests with the help of *Environments*. Any template variables in these parameters, denoted by double curly brackets, will be replaced with concrete values from the given Environment before the request is executed.

During the *Debriefing*, were told that the Whisperers can move members up the ranks if they're deemed worthy. Well it just so happens that Wanda, a fledgling member, has proven herself as a devout secret-keeper.

We'll start by adding another Request to our `memberships` Group:

```
# secrecy.yaml
---
memberships:
    invite: ...

    bump_rank:
        method: patch
        url: '{{ server }}/memberships/{{ member_id }}'
        headers:
            Content-Type: application/json
            X-Secret-Key: '{{ secret_key }}'
        body: |
            title: '{{ titles[rank + 1] }}'
            rank: '{{ rank + 1 }}'
```

Whew, lots of variables! Let's whip up an Environment file for Wanda. This strategy has the advantage that we can seamlessly move between different members without making any changes to the Collection.

```
# wanda.env.yaml
---
server: 'https://www.secrecy.org'
secret_key: sup3rs3cr3t
titles:
    - Loudmouth
    - Seeker
    - Keeper
    - Confidant
    - Spectre
member_id: UGK882I59
rank: 0
#new_secrets:
#    - secret basement room full of kittens
#    - turtles all the way down
```

Todo: add *new_secrets* below, remove from above.

Note: The `env.yaml` extension in `wanda.env.yaml` is just a convention to identify the file as an Environment. Any extension may be used.

We're almost ready to run it, but let's change `server` to something real so we don't get any errors:

```
server: http://httpbin.org/anything
```

Now we'll run the request:

```
$ restcli -c secrecy.yaml -e wanda.env.yaml run memberships bump_rank
```

Here's what **restcli** does when we hit enter:

1. Load the Collection (`secrecy.yaml`) and locate the Request `memberships.bump_rank`.
2. Load the Environment (`wanda.yaml`).
3. Use the Environment to execute the contents of the `url`, `headers`, and `body` parameters as Jinja2 Templates,.
4. Run the resulting HTTP request.

If we could view the finalized Request object before running it in #4, this is what it would look like:

```
# secrecy.yaml

method: post
url: 'https://www.secrecy.org/memberships/12345/bump_rank'
headers:
    Content-Type: application/json
    X-Secret-Key: sup3rs3cr3t
body: |
    rank: 1
    title: Seeker
```

Here's a piece-by-piece breakdown of what happened:

- **In the `url` section:**
  - `{{ server }}` was replaced with the value of Environment variable `server`.
  - `{{ member_id }}` was replaced with the value of Environment variable `member_id`.
- In the `headers` section, `{{ secret_key }}` was replaced with the value of Environment variable `secret_key`.
- **In the `body` section:**
  - `{{ rank }}` was replaced with the value of Environment variable `rank`, incremented by 1.
  - `{{ title }}` was replaced by an item from the Environment variable `titles`, an array, by indexing it with the incremented rank value.

Note: When it gets a request, http://httpbin.org/anything echoes back the URL, headers, and request body in the response. You can use this to check your work. If something is off, be sure to fix it before we continue.

Congrats on your new rank Wanda!

What we just learned should cover most use cases, but if you need more power or just want to explore, there's much more to templating than what we just covered! **restcli** supports the entire Jinja2 template language, so check out the official Template Designer Documentation for the whole scoop.

4.1.3 Scripting

Templating is a powerful feature that allows you to make modular, reusable Requests which encapsulate particular functions of your API without being tied to specifics. We demonstrated this by modeling a function to increase a member's rank, and created an Environment file to use it on Wanda. If we wanted to do the same for another member, we'd simply create a new Environment.

However, what happens when it's time for Wanda's second promotion? We know her current rank is 1, but the Environment still says 0. If we ran the `bump_rank` Request on the same Environment again, we'd get the same result:

```
# secrecy.yaml

body: |
    rank: 1
    title: Seeker
```

We need a way to update the Environment automatically after we run the Request.

This is achieved through scripting. As mentioned earlier in *Request Parameters*, each Request supports an optional `script` parameter which contains Python code. It is evaluated after the request is ran, and can modify the current Environment.

Let's add a script to our `bump_rank` Request:

```
# secrecy.yaml

bump_rank:
    ...
    script: |
        env['rank'] += 1
```

Now each time we run `bump_rank` it will update the Environment with the new value. Let's run it again to see the changes in action:

```
$ restcli --save -c secrecy.yaml -e wanda.env.yaml run memberships bump_rank
```

Notice that we added the `--save` flag. Without this, changes to the Environment would not be saved to disk.

Open up your Environment file and make sure `rank` was updated successfully.

Note: All script examples were written for Python3.7, but most will probably work in Python3+. To get version info, including the Python version, use the `--version` flag:

```
$ restcli --version
```

Under the hood, scripts are executed with the Python builtin `exec()`, which is called with a code object containing the script as well as a `globals` dict containing the following variables:

response A Response object from the Python requests library, which contains the status code, response headers, response body, and a lot more. Check out the Response API for a detailed list.

env A Python dict which contains the entire hierarchy of the current Collection. It is mutable, and editing its contents may result in one or both of the following effects:

A. If running in interactive mode, any changes made will persist in the active Environment until the session ends.

B. If `autosave` is enabled, the changes will be saved to disk.

Any functions or variables imported in the `lib` section of the *Config document* will be available in your scripts as well. We'll tackle the *Config document* in the next section.

Note: Since Python is whitespace sensitive, you'll probably want to read the section on *YAML block style*.

4.1.4 The Config Document

So far our Collections have been composed of a single YAML document. **restcli** supports an optional second document per Collection as well, called the Config Document.

Note: If you're not sure what "document" means in YAML, here's a quick primer:

Essentially, documents allow you to have more than one YAML "file" (document) in the same file. Notice that `---` that appears at the top of each example we've looked at? That's how you tell YAML where your document begins.

Technically, the spec has more rules than that for documents but PyYAML, the library **restcli** uses, isn't that strict. Here's the spec anyway if you're interested: http://yaml.org/spec/1.2/spec.html#id2800132

If present, the Config Document must appear *before* the Requests document. Breaking it down, a Collection must either:

- contain exactly one document, the Requests document, or
- contain exactly two documents; the Config Document and the Requests document, in that order.

Let's add a Config Document to our Secretmasons Collection. We'll take a look and then jump into explanations after:

```
# secrecy.yaml
---
defaults:
    headers:
        Content-Type: application/json
        X-Secret-Key: '{{ secret_key }}'
lib:
    - restcli.contrib.scripts

---
memberships:
    invite: ...

    upgrade: ...
```

Config Parameters

The Config Document is used for global configuration in general, so the parameters defined here don't have much in common.

`defaults` (object) Default values to use for each Request parameter when not specified in the Request. `defaults` has the same structure as a Request, so each parameters defined here must also be valid as a Request parameter.

`lib` (array) `lib` is an array of Python module paths. Each module here must contain a function with the signature `define(request, env, *args, **kwargs)` which returns a dict. That dict will be added to the execution environment of any script that gets executed after a Request is completed.

restcli ships with a pre-baked `lib` module at `restcli.contrib.scripts`. It provides some useful utility functions to use in your scripts. It can also be used as a learning tool.

4.2 Appendix

4.2.1 A. YAML Block Style

Writing multiline strings for the `body` and `script` Request parameters without losing readability is easy with YAML's block style. I recommend using literal style since it preserves whitespace and is the most readable. Adding to the example above:

```
body: |
    name: bar
    age: {{ foo_age }}
    attributes:
        fire_spinning: 32
        basket_weaving: 11
```

The vertical bar (|) denotes the start of a literal block, so newlines are preserved, as well as any *additional* indentation. In this example, the result is that the value of `body` is 5 lines of text, with the last two lines indented 4 spaces.

Note that it is impossible to escape characters within a literal block, so if that's something you need you may have to try a different

---

restcli is a terminal web API client written in Python. It draws inspiration from Postman and HTTPie, and offers some of the best features of both.

CHAPTER

FIVE

FEATURES

- save requests as YAML files
- scripting
- parameterized requests using Jinja2 templating
- expressive commandline syntax, inspired by HTTPie
- first-class JSON support
- interactive prompt with autocomplete
- colored output

CHAPTER

SIX

CLI USAGE

Command-line usage is documented in the Usage manual.

CHAPTER

SEVEN

DOCUMENTATION

CHAPTER

EIGHT

INSTALLATION

With `pip`:

```
$ pip install -r requirements.txt
$ pip install .
```

With `setup.py`:

```
$ python setup.py install
```

With `setup.py` but allow edits to the files under `restcli/` and reflect those changes without having to reinstall `restcli`:

```
$ python setup.py develop
```

If you have `invoke`, you can use it for running the tests and installation. If not, you can install it with `pip install invoke`.

```
$ invoke test     # Run the tests
$ invoke install  # Install it
$ invoke build    # Run the whole build workflow
```

8.1 Docker

Assuming Docker is installed, **restcli** can run inside a container. To build the Docker container, run the following from the project root:

```
$ docker build -t restcli .
```

Then you can run commands from within the container:

```
$ docker run -it restcli -c foobar.yaml run foo bar
$ docker run -it restcli --save -c api.yaml -e env.yaml env foo:bar
```

CHAPTER

NINE

ROADMAP

9.1 Short-term

Here's what we have in store for the foreseeable future.

- autocomplete Group and Request names in the command prompt
- support for other formats (plaintext, forms, file uploads)
- convert to/from Postman collections

9.2 Long-term

Here are some longer-term feature concepts that may or may not get implemented.

- full screen terminal UI via python_prompt_toolkit
- in-app request editor (perhaps using pyvim)

CHAPTER

TEN

LICENSE

This software is distributed under the Apache License, Version 2.0.