Contents

1 Literate Programming with org-mode and restclient TOC_4

1.1 Rest Call Me Maybe

Lately, I've spent a lot of time exploring web APIs. I've tried a couple of tools over the years: Postman — back when it was a chrome extension, Curl, and HTTPie. They were all, ok — they got the job done — but they all ended up missing some feature or some UX. That's when I found restclient, a package for Emacs with a simple `DSL` and could convert the restclient request to an equivalent `curl` request.

```
# Here we will post our base64 encoded secrets to get a bearer token
POST https://api.twitter.com/oauth2/token
Authorization: Basic QmRUTklkSEI5WWZHbVhwSkZ6NTZManpUNjpBMk1rOHJjaVdEWFdva3FCN1pmemZFd
Content-Type: application/x-www-form-urlencoded;charset=UTF-8

grant_type=client_credentials
```

Then once, you've `POSTed` the request a new buffer opens up with the response. #+NAME restclient response

```
{
  "token_type": "bearer",
  "access_token": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA%2FAAAAAAAAAAAAAAAAAAAA%3DAAAA
}
// POST https://api.twitter.com/oauth2/token?grant_type=client_credentials
// HTTP/1.1 200 OK
// cache-control: no-cache, no-store, must-revalidate, pre-check=0, post-check=0
// content-disposition: attachment; filename=json.json
// content-length: 155
// content-type: application/json;charset=utf-8
// date: Thu, 24 Jan 2019 05:54:09 GMT
// expires: Tue, 31 Mar 1981 05:00:00 GMT
// last-modified: Thu, 24 Jan 2019 05:54:09 GMT
// ml: S
// pragma: no-cache
// server: tsa_a
// status: 200 OK
// strict-transport-security: max-age=631138519
// x-connection-hash: b93d89db0ee8f4ea90991c99c8d58449
// x-content-type-options: nosniff
// x-frame-options: DENY
// x-response-time: 20
// x-transaction: 005403e60014e9ee
// x-twitter-response-tags: BouncerCompliant
// x-ua-compatible: IE=edge,chrome=1
// x-xss-protection: 1; mode=block; report=https://twitter.com/i/xss_report
// Request duration: 0.079710s
```

Using restclient as my primary means to play with APIs worked for me for quite a while. It was a simple to read document, that housed all the requests in a single file, and lived alongside the rest of my code in `git`.

I eventually ran into three complications when using it however:

1. My `restclient` files would become a mess and navigating to find a specific endpoint or a set of endpoints was a pain.

2. I was lazy and dealing with authenticated APIs was a pain.

    - I'd need to enter my credentials each time I needed to get a new token

   - Once I authenticated, I would have to copy and paste that token from the response buffer into a variable, to use throughout the restclient file.

3. For exploring APIs I like to take notes, record responses, and record the input.

There was a simple solution for my first and last points, which was to use org-mode. The power and flexibility of Emacs, and its suite of packages, constantly amazes me. I've been a long time fan of `org-mode`. IMO it is a great, in Emacs, replacement for Markdown and a number of other document formats. `org-mode` is a great place to practice literate programming, but it's also a great place to an interactive style of programming, like a REPL or a Jupyter Notebook. Org-mode has a babel extension for restclient. Babel is an extension that runs code contained in SRC block: `#+BEGIN_SRC...#+END_SRC` and saves anything sent to STDOUT or value returned, and then can be used in other SRC code blocks.

If you've never heard of `org-mode`, here's three key things to know before going forward:

1. If you don't know what `org-mode` is, here is a short video exploring some of its features and here is a much longer video on literate programming in Emacs

2. This entire document was written using `org-mode` and `babel`

3. You can execute this document yourself, given you change your credentials ;)

1. **TODO** create a link to the raw file and create a gist for outlining the requirements for this project

1.2 Twitter

A good way to demonstrate the power of `restclient` and `org-mode` would be to post some tweets from Emacs. First, we need to be able to authenticate with twitter. So, let's see how we can use `org-mode` and `restclient` to authenticate with an `OAuth` endpoint.

1.2.1 Helper functions

First, we need to define a few functions that we are going to use during OAuth authentication. I could use a library or package for this, but when I am writing I become somewhat of a masochist.

```
(defun twitter-signing-key (consumer-secret token-secret)
  "Creates a signing key by combining the consumer-secret and the token secret and perc
  (concat
   (url-encode-url
    consumer-secret)
   "&"
   (url-encode-url
    token-secret)))

(defun twitter-signature-string (method base params)
  "Builds a hex encoded string of the format METHOF&BASE&PARAM1=VALUE1..."
  (let ((sorted-params
         (sort params
               (lambda (first second)
                 (string< (car first) (car second))))))
    (concat
     method
     "&"
     (url-hexify-string base)
     "&"
     (url-hexify-string
      (mapconcat
       (lambda (entry)
         (let ((key (car entry))
               (value (cdr entry)))
           (concat (url-hexify-string key)
                   "="
                   (url-hexify-string value))))
       sorted-params
       "&")))))

(defun build-twitter-header-string (header oauth-headers)
  "Takes in a list of cons cells that represent HTTP headers, as well as the informatio
   the OAUTH response for a Twitter request, and build a restclient style header string
  (concat
   "<<\n"
   (mapconcat
    (lambda (entry)
      (let ((key (car entry))
            (value (cdr entry)))
```

```
        (concat
         key
         ": "
         value
         " ")))
    header
    "")
   "\nAuthorization: OAuth "
   (string-trim-right
    (mapconcat
     (lambda (entry)
       (let ((key (car entry))
             (value (cdr entry)))
         (concat
          key
          "="
          "\"" value "\""
          ",")))
     oauth-headers
     " ")
    ",")
   "\n#"))
```

1.2.2 Shhh It's a Secret

I don't need to store the authentication information in files, and I am to lazy to remember or copy and paste them! I can just use the information that is stored in my environment.

```
echo $TWITTER_CONSUMER_KEY
```

```
echo $TWITTER_CONSUMER_SECRET
```

```
echo $TWITTER_ACCESS_TOKEN
```

```
echo $TWITTER_ACCESS_SECRET
```

1.2.3 Let's Auth it

1. Step 1: Generate a body

   Before we can do all the fun authentication bits that is `OAuth2`, we need to have some content. So, I feel like I need to be on brand for an Emacs user and let everyone know I am using Emacs for something that isn't editing text.

```
(setq twitter-body (list (cons "status" "Hello world! I'm tweeting from Emacs")))
```

2. Step 2: Generating some header data Ok, now that we have our Twitter status, we need to auto generate a few more pieces of information; a nonce, a timestamp, and the signature.

   Emacs doesn't really have a built in crypto library, but do you know who does? Ruby! It is a fun language with a pretty full featured standard library, let's use it to generate our nonce.

   ```
   require 'securerandom'

   nonce = SecureRandom.uuid
   nonce.gsub(/\W/, "")
   ```

   Our request is going to need a time signature.

   ```
   (format-time-string "%s")
   ```

   We need to define the headers that we need for this request.

   ```
   (list
    (cons "Content-Type" "application/x-www-form-urlencoded"))
   ```

   Did I mention Emacs built-in cryptography is kind of lacking? Well, we'll need to let another language do the heavy lifting when signing the request. I like Node and Node has a decent crypto library built into it. In the example below I am defining a code block as a function that I am going to call later and use it in an emacs-lisp source block.

   ```
   let crypto = require('crypto')

   let createSignature = (key, text) => {
     return crypto.createHmac('sha1', key).update(signature_string).digest();
   }

   return createSignature(key, signature_string).toString('base64');
   ```

Now before we can sign anything, and we **do** need to sign things, we need to create a signing key. We can use our consumer-secret and our access-secret to build a Twitter signing key.

```
(twitter-signing-key consumer-secret token-secret)
```

3. Step 3: Creating The Header Next up, we need to build the header, create a string to sign, sign that string and them add that signature to our header. Simple.

```
(let*
    ((twitter-oauth-headers
      (list
       (cons "oauth_consumer_key" consumer-key)
       (cons "oauth_nonce" nonce)
       (cons "oauth_signature_method" "HMAC-SHA1")
       (cons "oauth_timestamp" oauth-time)
       (cons "oauth_token" access-token)
       (cons "oauth_version" "1.0")))
     (signature-string
      (twitter-signature-string "POST"
                                "https://api.twitter.com/1.1/statuses/update.json
                                (append twitter-oauth-headers twitter-body)))
     (signature
      (org-sbe createSignature (signature_string (eval signature-string)) (key (e
  (append twitter-oauth-headers (list (cons "oauth_signature"
                                             (url-hexify-string signature)))))
```

4. Step 4: Encoding Data and Posting To Twitter

Up next, our headers need to be in a string format that `restclient` knows how to read.

```
(build-twitter-header-string header (sort twitter-oauth-headers
                                          (lambda (first second)
                                            (string< (car first) (car second)))))
```

We need to encode our body as a post parameter string.

```
(setq twitter-post-body
      (concat
       ""
       (mapconcat
        (lambda (entry)
          (concat (car entry) "=" (url-hexify-string (cdr entry))))
        twitter-body
        "&")
       ""))
```

Finally, now that we've done all that work to formatting and signing things, we can finish it off by tweeting to the world how much we love Emacs.

```
#
:body := (concat twitter-post-body)
POST https://api.twitter.com/1.1/statuses/update.json?:body
:twitter-headers
```

1.3 To 11

`./images/to_11.gif`

I think using `org-mode` and `restclient` to authenticate and post on Twitter is a little too mundane. Can we do anything more elaborate?

Why, of course we can! This is Emacs, we pretty much have to do something overly complicated.

I'm a big fan of science and I want to share my enthusiasm with the world. So, we're going to use our newly learned skills to talk across several APIs. We're going to:

1. To grab a plant name from trefle.io
2. Find a picture of that plant, using Google Custom Search
3. Make sure that the picture we have is of that plant, using Google vision
4. Tag someone on Twitter and share the plant name and picture with them.

1.3.1 More Helper Functions

We need a function to sanitize the response we get from `restclient`

```
(defun sanitize-restclient-response (string)
 "Trim down a restclient response to JSON, removing the org source block and header in
 (string-trim (replace-regexp-in-string "^#\\+BEGIN_SRC js\\|^#\\+END_SRC\\|^//[[:prin
```

Let's be able to execute an arbitrary source code block

```
(defun run-org-block (&optional code-block-name)
  (save-excursion
    (let ((code-block (or code-block-name
                          (completing-read "Code Block: " (org-babel-src-block-names))
          (goto-char
           (org-babel-find-named-block
            code-block))
          (org-babel-execute-src-block-maybe)))))
```

Here's a couple of functions we're going to use to help us parse a response from Google's API.

```
(defun parse-ml-response (responses)
  "Extracts a Google AI response down to a list of label annotations"
  (let* ((json-response (json-read-from-string responses))
         (label-annotations  (cdr
                              (assoc 'labelAnnotations
                                     (elt
                                      (cdr (assoc 'responses json-response))
                                      0)))))
    label-annotations))

(defun contains-description-p (annotations descriptions)
  "Checks to see if any of the items in the sequence ANNOTATIONS has a description tha
  (let ((annotated-descriptions (mapcar (lambda (item) (cdr (assoc 'description item))
    (reduce (lambda (predicate description)
              (if predicate
                  predicate
                (if (member (downcase description) descriptions)
                    't
                  nil)))
            annotated-descriptions
            :initial-value nil)))
```

1.3.2 Harvesting a name

Let's give our source block a name, `#+NAME: trefle`, so we can easily reference it throughout the rest of our notebook. I am using my Mac's keychain to store and retrieve an access token I have stored for trefle.io.

```
security find-generic-password -gws trefle.io
```

To import a variable from earlier in the file you can use `:var token=trefle` where :var token, specified that you what to insert a variable called token into the proceeding block and the contents of that variable a pull from a block by the name of `trefle`. Now we just need to build the HTTP headers we're going to use for our interaction with `trefle`.

```
(concat
   "<<
Content-Type: application/json
Accept: application/json
Authorization: Bearer " token)
```

As of the last time I looked, trefle.io has over 4000 pages of plants, so we want to get a random plant off of a random page. So to start, we'll generate a page number from 0 to 4000…

```
#
:page := (random 4000)
GET https://trefle.io/api/plants?page=:page
:headers
#
```

Before we can do anything with the output we need to clean it up, `restclient` likes to have all the headers for the response at the bottom of the buffer, so we need to filter those out of the response.

```
(sanitize-restclient-response response)
```

Now we could use emacs-lisp, but everyone has NodeJS installed and NodeJS is pretty much built for parsing JSON, so it only makes sense to use that. We'll grab a random plant from the results and return its name.

```
let index = Math.floor(Math.random() * 30);
return JSON.parse(plants)[index].scientific_name;
```

1.3.3 Imagine

I need to get my Google API key, for this I've been lazy and have just been storing it in my environment.

```
echo $GOOGLE_API_KEY
```

We've got a plant name, now we need image of the plant.

```
GET https://content.googleapis.com/customsearch/v1?cx=009341007550343915479%3Afg_hsgzl
```

Much like our search for a plant name, we need to clean up the response from Google API so it's easily parseable as JSON.

```
(sanitize-restclient-response google-images)
```

We have a nice list of plant images, let's play Google roulette and use the first image from the search.

```
return "" + JSON.parse(plant_images).items[0].link
```

1.3.4 What shall we learn today?

./images/PreciousHoarseFieldspaniel.gif

When we're running our code we don't have time to make sure all it does what it is supposed to do and everyone knows you can't trust Google. Instead, we'll use machine learning provided by the fabulous Google to validate our choice. We'll ask Google for them top 3 labels for an image and see if those labels contain the words "Flower", "Plant", or "Tree".

```
POST https://vision.googleapis.com/v1/images:annotate?key=:api-key
{
  "requests":[
    {
      "image":{
        "source":{
          "imageUri":
          :plant-image
        }
      },
      "features":[
        {
          "type":"LABEL_DETECTION",
          "maxResults":3
        }
      ]
    }
  ]
}
```

```
(sanitize-restclient-response response)
```

If you're curious what talking to the Google Vision API looks like, here it is.

```
{
  "responses": [
    {
      "labelAnnotations": [
        {
          "mid": "/m/04_tb",
          "description": "map",
          "score": 0.9684097,
          "topicality": 0.9684097
        },
        {
          "mid": "/m/03scnj",
          "description": "line",
          "score": 0.734654,
          "topicality": 0.734654
        },
        {
          "mid": "/m/07j7r",
          "description": "tree",
          "score": 0.7276011,
          "topicality": 0.7276011
        }
      ]
    }
  ]
}
```

Let's check to see if the first three descriptors come back as plant, tree, or a flower. If it doesn't match these descriptors, then we rerun this code block. Warning: this could trap us into an infinite loop.

```
(while (not (contains-description-p
             (parse-ml-response response)
             '("plant" "tree" "flower")))
    (org-sbe image-is-plant-p))
t
```

1.3.5 A rose by any other name

We need one last piece of information before we can demonstrate our love of plants to the world: someone to tweet at. Let's ask ourselves for some input.

```
(read-string "What is the twitter handle of someone you want to tweet? ")
```

1.3.6 Content is king

Now we need to build our body into something we can process later together…

```
(setq twitter-body
 (list
  (cons "status" (concat "" twitter_handle " " plant_name " " (replace-regexp-in-string
```

1.3.7 Preparation, creating a new twitter header.

We can use all of the source blocks we created back when we were professing our love for Emacs. However, we need to change a few references. In the source block below we need to change the reference from `body=hello-world` to `body=twitter-plant-body`.

```
(let*
    ((twitter-oauth-headers
      (list
       (cons "oauth_consumer_key" consumer-key)
       (cons "oauth_nonce" nonce)
       (cons "oauth_signature_method" "HMAC-SHA1")
       (cons "oauth_timestamp" oauth-time)
       (cons "oauth_token" access-token)
       (cons "oauth_version" "1.0")))
     (signature-string
      (twitter-signature-string "POST"
                                "https://api.twitter.com/1.1/statuses/update.json"
                                (append twitter-oauth-headers twitter-body)))
     (signature
      (org-sbe createSignature (signature_string (eval signature-string)) (key (eval s
  (append twitter-oauth-headers (list (cons "oauth_signature"
                                            (url-hexify-string signature)))))
```

1.3.8 Repeat: Encoding Data and Posting To Twitter

Similiarly, we need to reassign `twitter-oauth-headers=twitter-oauth-headers` to `twitter-oauth-headers=twitter-oauth-headers-plants`

```
(build-twitter-header-string header (sort twitter-oauth-headers
                                          (lambda (first second)
                                            (string< (car first) (car second)))))
```

Again, we encode our body…

```
(setq twitter-post-body
      (concat
       ""
       (mapconcat
        (lambda (entry)
          (concat (car entry) "=" (url-hexify-string (cdr entry))))
```

```
         twitter-body
         "&")
        ""))
```

Voila! We've can post a cute plant…or tree…or flower…to Twitter!

```
#
:body := (concat twitter-post-body)
POST https://api.twitter.com/1.1/statuses/update.json?:body
:twitter-headers
```

./images/org-mode-low-quality.gif

2 References

1. https://developer.twitter.com/en/docs/basics/authentication/overview/application-only
2. https://cloud.google.com/vision/docs/request
3. https://developer.twitter.com/en/docs/tweets/post-and-engage/api-reference/post-statuses-update.html
4. http://lti.tools/oauth/