The HawkOwl Technicals

Release Infinite

HawkOwl

April 30, 2015

Contents

technical, ***noun*** *...an article, or series of articles, which pertain to a subject that requires special knowledge to be understood.*

The Technicals are my attempt at writing a book, one useful technical subject at a time.

They do not intend to be an authoritive guide on anything in particular, but will eventually contain valuable information on the topics of **Twisted**, **Python**, **API design**, and other miscellany.

Contents:

CHAPTER 1

Essays

General articles and essays.

1.1 Testing & I/O

It is said that a watched pot never boils, and I find that the same is with untested code. You write it, you run it, it seems to work, you push it into production, and call it a day. Unfortunately, as soon as you purge the knowledge of that code ever existing from your mind, a user will decide that today is the day that their cat will do some data entry, and everything falls apart.

The most generally accepted approach to preventing your application from having some fatal bug in it is through testing your code – that is, "watching" your code. An automated set of eyes to make sure that all is sane, and that when the user decides to set their display name to "HE COMÉS" that it doesn't explode with a nasty red page and big letters saying `UnicodeDecodeError`.

Generally, this automated testing is done to make sure that the code:

- is syntactically correct – free from syntax errors
- is computationally correct – that given inputs give the correct output
- is "clean" – matches style guides, has no unused variables
- fufils business requirements – that it works, as an element of a system as a whole, to do the task the system was written to do

Different types of testing will do one or more of the things above. Static analysis will make sure that the code is clean and meets style guides, unit tests will make sure the code is both forms of correct as mentioned, and integration tests will make sure that it meets the business requirements it was written to solve. I, here, am talking about unit tests.

1.1.1 What is unit testing?

Unit testing is when individual portions of your code ("units") are tested in isolation of the rest of the system.

haha work in progress

why testing:

- make sure code is clean (eg. pyflakes)
- make sure code is syntactically correct (eg. free from syntax errors)
- make sure code is computationally correct (eg. the input gives the correct output)
- make sure code fufils business requirements (eg. does a certain thing for a customer)
- unit testing is #2 and #3, code analysis is #1, and functional testing is #3 and #4

why unit testing:

- makes sure the units that make up *your* application are correct
- test the pure logic of your application at a near-functional level
- minimises the effect of implementation details on your application's testing (eg. databases)

why i/o in unit tests are bad:

- external dependencies makes it more complex and fragile
- you end up testing the I/O is reliable, rather than your code (and if it's not, you get spurious failures)
- if you're not testing on I/O boundaries, chances are you're not covering certain internal details
- you have to rely on implementation details

what you can do about it:

- write your functions so that they take some state and return some other state, rather than mutating some random state
- write your methods so that they only mutate the state of its object
- have your I/O go through a wrapper that can be faked – eg. rather than talking to memcache, you can use a dictionary
- don't have anything do I/O without telling you – for example, importing a module should NOT do any I/O. you should instantiate an object with some parameters that then does the I/O for you, and is given to the code

what you will get:

- code with unit-testable I/O boundaries (eg. through an interface) is less prone to being stuck with one backend implementation
- tests will run much faster, as it works only in memory, and can be parallelised very easily
- zero-setup tests – no setting up databases or anything to make sure your code is sane
- 100% less sad owls

CHAPTER 2

Twisted Technicals

This contains my technicals about Twisted – using Mail, Web, Deferreds, and a bunch more.

2.1 Twisted Mail

2.1.1 Sending Mail

Sometimes, your application may want to send an email. Below is a function which you can use to send mail over an encrypted connection and authenticating to the server:

esmtpmail.py

```
from cStringIO import StringIO
from twisted.internet import reactor, endpoints, defer
from twisted.mail.smtp import ESMTPSenderFactory

def sendMail(username, password, host, port, msg):
    """
    Send an email message using authenticated and encrypted SMTP.

    @param username: Username to use.
    @param password: Password to use.
    @param host: SMTP server's hostname or IP address.
    @param port: The port to connect to for SMTP.
    @param msg: Email to send.
    @type msg: L{email.message.Message}

    @return: A L{Deferred} that fires with the result of the email being
             sent.
    """
    resultDeferred = defer.Deferred()

    fp = StringIO(msg.as_string(unixfrom=True))

    senderFactory = ESMTPSenderFactory(username, password, msg.get("From"),
        msg.get("To"), fp, resultDeferred)

    endpoints.HostnameEndpoint(reactor, host, port).connect(senderFactory)

    return resultDeferred
```

How can it be encrypted if the reactor connects over TCP, and not SSL/TLS? Not to worry – ESMTP implements STARTTLS (you can find more details on it here). This 'upgrades' the connection from cleartext to an encrypted one before any sensitive data is sent.

The implementation of ESMTPSenderFactory, which this code uses, requires the use of encryption (see ESMTPSender's source).

Here is an example, using the above:

email_example.py

```
from __future__ import print_function
from esmtpmail import sendMail # the code above
from email import message
from twisted.internet.task import react

def main(reactor):

    m = message.Message()
    m.add_header("To", "hawkowl@atleastfornow.net")
    m.add_header("From", "hawkowl@atleastfornow.net")
    d = sendMail("hawkowl@atleastfornow.net", "password",
                 "mail.atleastfornow.net", 587, m)
    d.addCallback(print)
    return d

react(main)
```

Upon running it, you will get something like this:

```
$ python email_example.py
(1, [('hawkowl@atleastfornow.net', 250, '2.1.5 Ok')])
```

The printed line is the result of the Deferred that is returned by `sendMail`, once the mail transaction has been completed.

The result is a 2-tuple containing how many addresses the mail was sent successfully to and the sending results. The sending results is a list of 3-tuples containing the email, the SMTP status code (see section 4.2 of the RFC), and the dot-separated ESMTP status code, for each recepient. In the example, the SMTP code of 250 says that everything is OK, and the ESMTP status "2.1.5 Ok" means that the recepient address was valid.

2.2 Klein

Klein is a Flask-like API on top of *Twisted Web*.

2.2.1 Getting Started

Klein is a micro-framework for developing production-ready web services with Python, built off Werkzeug and Twisted. The purpose of this introduction is to show you how to install, use, and deploy Klein-based web applications.

This Introduction

This introduction is meant as a general introduction to Klein concepts.

Everything should be as self-contained, but not everything may be runnable (for example, code that shows only a specific function).

Installing

Klein is available on PyPI. Run this to install it:

```
pip install klein
```

Note: Since Twisted is a Klein dependency, you need to have the requirements to install that as well. You will need the Python development headers and a working compiler - installing `python-dev` and `build-essential` on Debian, Mint, or Ubuntu should be all you need.

Hello World

The following example implements a web server that will respond with "Hello, world!" when accessing the root directory.

```
from klein import run, route

@route('/')
def home(request):
    return 'Hello, world!'

run("localhost", 8080)
```

This imports `run` and `route` from the Klein package, and uses them directly. It then starts a Twisted Web server on port 8080, listening on the loopback address.

This works fine for basic applications. However, by creating a Klein instance, then calling the `run` and `route` methods on it, you are able to make your routing not global.

```
from klein import Klein
app = Klein()

@app.route('/')
def home(request):
    return 'Hello, world!'

app.run("localhost", 8080)
```

By not using the global Klein instance, you can have different Klein routers, each having different routes, if your application requires that in the future.

Adding Routes

Add more decorated functions to add more routes to your Klein applications.

```
from klein import Klein
app = Klein()

@app.route('/')
def pg_root(request):
    return 'I am the root page!'

@app.route('/about')
def pg_about(request):
    return 'I am a Klein application!'

app.run("localhost", 8080)
```

Variable Routes

You can also make *variable routes*. This gives your functions extra arguments which match up with the parts of the routes that you have specified. By using this, you can implement pages that change depending on this – for example, by displaying users on a site, or documents in a repository.

```
from klein import Klein
app = Klein()

@app.route('/user/<username>')
def pg_user(request, username):
    return 'Hi %s!' % (username,)

app.run("localhost", 8080)
```

If you start the server and then visit `http://localhost:8080/user/bob`, you should get `Hi bob!` in return.

You can also define what types it should match. The three available types are `string` (default), `int` and `float`.

```
from klein import Klein
app = Klein()

@app.route('/<string:arg>')
def pg_string(request, arg):
    return 'String: %s!' % (arg,)

@app.route('/<float:arg>')
def pg_float(request, arg):
    return 'Float: %s!' % (arg,)

@app.route('/<int:arg>')
def pg_int(request, arg):
    return 'Int: %s!' % (arg,)

app.run("localhost", 8080)
```

If you run this example and visit `http://localhost:8080/somestring`, it will be routed by `pg_string`, `http://localhost:8080/1.0` will be routed by `pg_float` and `http://localhost:8080/1` will be routed by `pg_int`.

Route Order Matters

But remember: order matters! This becomes very important when you are using variable paths. You can have a general, variable path, and then have hard coded paths over the top of it, such as in the following example.

```
from klein import Klein
app = Klein()

@app.route('/user/<username>')
def pg_user(request, username):
    return 'Hi %s!' % (username,)

@app.route('/user/bob')
def pg_user_bob(request):
    return 'Hello there bob!'

app.run("localhost", 8080)
```

The later applying route for bob will overwrite the variable routing in `pg_user`. Any other username will be routed to `pg_user` as normal.

Static Files

To serve static files from a directory, set the `branch` keyword argument on the route you're serving them from to `True`, and return a t.w.static.File with the path you want to serve.

```
from twisted.web.static import File
from klein import Klein
app = Klein()

@app.route('/', branch=True)
def pg_index(request):
    return File('./')

app.run("localhost", 8080)
```

If you run this example and then visit `http://localhost:8080/`, you will get a directory listing.

Deferreds

Since it's all just Twisted underneath, you can return Deferreds, which then fire with a result.

```
import treq
from klein import Klein
app = Klein()

@app.route('/', branch=True)
def google(request):
    d = treq.get('https://www.google.com' + request.uri)
    d.addCallback(treq.content)
    return d

app.run("localhost", 8080)
```

This example here uses treq (think Requests, but using Twisted) to implement a Google proxy.

Return Anything

Klein tries to do the right thing with what you return. You can return a result (which can be regular text, a Resource, or a Renderable) synchronously (via `return`) or asynchronously (via `Deferred`). Just remember not to give Klein any `unicode`, you have to encode it into `bytes` first.

Onwards

That covers most of the general Klein concepts. The next chapter is about deploying your Klein application using Twisted's `tap` functionality.

2.2.2 Using `twistd` to Start Your Application

`twistd` (pronounced "twist-dee") is an application runner for Twisted applications. It takes care of starting your app, setting up loggers, daemonising, and providing a nice interface to start it.

Using the `twistd web` Plugin

Exposing a valid IResource will allow your application to use the pre-existing `twistd web` plugin.

To enable this functionality, just expose the `resource` object of your Klein router:

```
from klein import Klein
app = Klein()

@app.route('/')
def hello(request):
```

```
    return "Hello, world!"

resource = app.resource
```

Then run it (in this example, the file above is saved as `twistdPlugin.py`:

```
$ twistd -n web --class=twistdPlugin.resource
```

The full selection of options you can give to `twistd web` can be found in its help page. Here are some relevant entries in it:

```
  -n, --notracebacks      Do not display tracebacks in broken web pages.
                          Displaying tracebacks to users may be security risk!
  -p, --port=             strports description of the port to start the server
                          on.
  -l, --logfile=          Path to web CLF (Combined Log Format) log file.
      --https=            Port to listen on for Secure HTTP.
  -c, --certificate=      SSL certificate to use for HTTPS. [default:
                          server.pem]
  -k, --privkey=          SSL certificate to use for HTTPS. [default:
                          server.pem]
      --class=            Create a Resource subclass with a zero-argument
                          constructor.
```

Using HTTPS via the `twistd web` Plugin

The `twistd web` plugin has inbuilt support for HTTPS, assuming you have TLS support for Twisted.

As an example, we will create some self-signed certs – for the second command, the answers don't really matter, as this is only a demo:

```
$ openssl genrsa > privkey.pem
$ openssl req -new -x509 -key privkey.pem -out cert.pem -days 365
```

We will then run our plugin, specifying a HTTPS port and the relevant certificates:

```
$ twistd -n web --class=twistdPlugin.resource -c cert.pem -k privkey.pem --https=4433
```

This will then start a HTTPS server on port 4433. Visiting `https://localhost:4433` will give you a certificate error – if you add a temporary exception, you will then be given the "Hello, world!" page. Inspecting your browser's URL bar should reveal a little lock – meaning that the connection is encrypted!

Of course, in production, you'd be using a cert signed by a certificate authority – but self-signed certs have their uses.

things we should talk about here:

- set up klein in a class
- set up Options and makeService
- create a twisted tap file
- start it through twistd
- use https (FOR SECURITY)

2.3 Twisted Web

Twisted Web is a Twisted project that aims to provide a useful, stable, and reliable base for web development.

Contents:

2.3.1 Using the Twisted Web `twistd` Plugin

Twisted provides a plugin for the `twistd` application runner, out of the box. It can be run by running `twistd web` from the command line.

`twistd` Options

`twistd` has a few important global options – the most useful of which is `-n` (or `--nodaemon`). This prevents `twistd` from forking to the background, and is handy for when you are testing or playing around with its functionality.

Setting the Port

To set the port that `twistd web` will listen on, specify the `--port` argument to `twistd web` with an endpoint description string understandable by serverFromString . For example, `--port tcp:8080` will make it listen on TCP port 8080.

Serving Static Files

To serve static files, specify the `--path` argument with the path that you wish to serve static files from. As an example, `twistd -n web --port tcp:8080 --path /tmp` will serve your `/tmp` directory on port 8080.

Serving WSGI

WSGI applications can be served by using the `--wsgi` argument with the fully qualified Python name of the WSGI application you wish to serve.

2.3.2 A New Web Framework Proposal

Abstract

Twisted Web, in my opinion, is not so much a web framework, but the building blocks you would create a web server *with*. Outside of Twisted, there are several things which provide this web framework – the Divmod tools, Klein, Cyclone – but they are either complex, imperfect in implementation, or quite heavy-weight. Learning from these past implementations, Twisted should provide a user-facing web framework.

Example Code

```
from twisted.internet import defer
from twisted.web.veridical import Router, PluggableResource, CSRF, VeridicalSite
from twisted.web.veridical.chunks import TextChunk, IntegerChunk
from twisted.web.veridical.responses import Redirect, TextResponse


class Blog(object):

    router = Router()

    def __init__(self, db):
        self.db = db

    @defer.inlineCallbacks
    @router.route()
```

```
    def root(self, request):
        loginOk = yield self.augments['authentication'].checkLogin(request)

        if loginOk:
            return TextResponse(u"Hi, logged in person!")
        else:
            return TextResponse(u"Hi, logged out person!")

    @defer.inlineCallbacks
    @router.route("posts", IntegerChunk("postID"))
    def postID(self, request, postID=None):
        post = yield self.db.fetchPostByID(postID)
        defer.returnValue(TextResponse(post.content))


    @defer.inlineCallbacks
    @router.route("posts", TextChunk("postSlug"))
    def postSlug(self, request, postSlug=None):
        post = yield self.db.fetchPostBySlug(postSlug)
        defer.returnValue(TextResponse(post.content))


class HumansTXT(object):

    router = Router()

    @router.route("humans.txt")
    def humansTXT(self, request):
        return TextResponse(u"this website was actually made by robots")


class UserAuthenticationService(object):

    router = Router()

    def __init__(self, ID, db):
        self.db = db

    @defer.inlineCallbacks
    def checkLogin(self, request):

        isOk = yield self.db.checkCookie(request.args["user"],
                                         request.getCookie("SESSION"))
        defer.returnValue(isOk)


    @router.route("login")
    def login(self, request):

        CSRF.setToken(request)

        with open("loginpage.html") as f:
            loginPage = f.read()
        return TextResponse(loginPage)


    @defer.inlineCallbacks
    @router.route("login", method="POST")
    def login_POST(self, request):

        if CSRF.checkToken(request):
            authCookie = yield self.db.checkAuth(request.args["user"],
                                                 request.args["password"])
```

```
            if authCookie:
                request.setCookie("SESSION", authCookie)
                defer.returnValue(Redirect())
            else:
                response = TextResponse(u"login failed")
                response.setCode(400)
                defer.returnValue(response)
        else:
            response = TextResponse(u"CSRF failed")
            response.setCode(400)
            defer.returnValue(response)



def authRequiredMiddleware(config):

    @inlineCallbacks
    def _(site, handler, request):

        loginOK = yield self.augments['authentication'].checkLogin(request)

        if loginOK or True in map(request.matches, config["allowed"]):
            return handler(request)
        else:
            return Redirect(*config["redirectTo"])

    return _



db = DBThing()
blog = Blog("base", db)
authentication = UserAuthenticationService("authentication", db)
authenticationRequiredMiddleware = authRequiredMiddleware({
    "allowed": [["accounts", "login"],
                []],
    "reDirectTo": ["accounts", "login"]
})

# Add a humans.txt to the blog service
blog.router.augment(HumansTXT())

# Make a default, empty site.
service = VeridicalSite()

# Add two augments, blog on the root level, and authentication under accounts/
service.router.augment(blog)
service.router.augment("accounts", authentication)

# Register some middleware on the site's router
service.router.middleware.register(authenticationRequiredMiddleware)

# Convenience function
service.router.run('localhost', 8080)

# The routing table will look like this

# / -> blog.root
# /posts/<int:postID> -> blog.postID
# /posts/<str:postSlug> -> blog.postSlug
# /humans.txt -> HumansTXT.humansTXT
# /accounts/login -> authentication.login
```

```
# The order of routing is:
# 1. Routes defined directly on the router.
# 2. Routes defined in augments in the order they were added.
# 3. 404.

# For example, if the router had / and /hi, and it had an augment with the
# routes / and /hello, and that augment had an augment with the routes /hi and
# /there, the request for /hi would be on the first router, /hello would be on
# the second router, and /there would be on the third router.

# As another example, say you have a router with the routes / and /hi, and an
# augment under /hi/there which has the routes / and /foo. If the request /hi
# was given, the first router's /hi would get it. But the request /hi/foo would
# go to the second, as there is no direct match, and it would be passed to the
# second router (albeit as just "/foo", as the "hi" was consumed by the router
# that was augmented).
```

CHAPTER 3

API Design Technicals

This contains my technicals about API design.

Contents:

3.1 Versioning

When updating or changing your API, you need to make sure that existing users are still able to use your service. Updates that change functionality, remove features, or otherwise alter the 'API contract' may have a knock-on effect and cause API clients to stop working. By explicitly versioning your API, you can declare that the changes constitute a new 'API contract', but keep the existing interface operating until users have updated. Unmaintained or hard to update clients can be kept working with a subset of functionality of the new API, by just staying on the old version.

There are several ways of indicating that your API is versioned – some services use a header to specify which version, and some put it directly in the URI. I think that this is the best way of going about it, as it clearly namespaces your APIs, makes it directly obvious to the implementor what version they are using, and allows cleanup or alteration of the resource layout.

This versioning should occur as high up the tree as is possible, along product lines. For example, a system that operates as one whole should be kept on the same version, but two different services of a company may be versioned different if each is accessed as a distinct different API.

3.2 Layout

Being built on top of HTTP, all web APIs use URLs (Uniform Resource Locators) in some way. As the name suggests, they are a standard way of locating *resources*, which is a 'thing' which can accept or provide data.

There are three main patterns in URL layout, which I have termed **Single Endpoint**, **Function Endpoint**, and **RFC-3986 Style**.

3.2.1 Single Endpoint

A *Single Endpoint* API is where all access is performed by communication through a single endpoint – that is, one URL – no matter what you are doing with it. Adding query arguments to requests to this endpoint then indicate what function should be run and what data to operate on.

Since a single endpoint is used, the whole service is a single 'resource'. This is more akin to a remote procedure call interface than a REST-style web interface.

Real World Example: Linode

The Linode API uses this structure, with their API endpoint at `https://api.linode.com/`.

Using it involves `GET`/`POST` requests to this URL, with an `api_action` query argument that denotes which 'function' to run. Arguments to the function specified are then given by additional query arguments.

An example of an API request that runs the `test.echo` function with the API key `SECRETKEY` is:

```
$ curl "https://api.linode.com/" \
      -d "api_key=SECRETKEY" \
      -d "api_action=test.echo" \
      -d "foo=bar"
```

You can see the full teardown and analysis of the DNS Manager portion of this API at *Deconstructing the Linode DNSManager API*.

3.2.2 Function Endpoint

An API where there are multiple endpoints, each providing a function to run. Query arguments are then added to these endpoints, specifying what data or identifier to operate on.

This layout is similar to **Single Endpoint**, but instead of specifying what function to run by a query argument, it is encoded in the URL.

Example: Blog

Imagine a blog which uses this structure. They have an endpoint for creating blog posts at `https://blog.example.com/api/create_post`, and an endpoint to delete a post at `https://blog.example.com/api/delete_post`.

To create a post, this API request may be used:

```
$ curl "https://blog.example.com/api/create_post?title=test&content=test"
```

This would create a blog post on this service, and somehow return some kind of ID for the new post. To delete it, they would then use this request:

```
$ curl "https://blog.example.com/api/delete_post?id=1"
```

3.2.3 RFC-3986 Style

An API which is laid out in the vein of **RFC 3986**. This is characterised by object types and identifiers being in the URL.

Laid out like this, it allows a single record of data to be referred to entirely in the URL. Most uses of this style of API are data-driven – when clients put information into the system, actions which handle new/changed data are run implicitly. When 'functions' are required, they are usually handled by a resource that accepts `POST` requests.

Real World Example: Stripe

The Stripe API uses this system. Stripe is a payment processor, allowing companies to process credit card payments over the internet.

An example of this API is given in their docs, displayed here. This example creates a new customer, using the API key `sk_test_BQokikJOvBiI2HlWgH4olfQ2` (with no password, as Stripe just use the one key), by `POST`ing at the `customers` object.

```
$ curl "https://api.stripe.com/v1/customers" \
       -u sk_test_BQokikJOvBiI2HlWgH4olfQ2: \
       -d "description=Customer for test@example.com" \
       -d "card=tok_1046XL2eZvKYlo2CsaCAcF5P"
```

Note: `-d` is the argument for adding HTTP `POST` data in cURL. The presence of `-d` changes the verb to `POST` implicitly.

The response then gives the identifier of the created customer.

The main difference between this style of API and the others is that accessing a customer is not done by giving a parameter (eg. in a query argument), but by adding the object type and identifier in the URI. This example, also from the Stripe API docs, fetches a customer by `GET` ting a URI with the customer's identifier.

```
$ curl "https://api.stripe.com/v1/customers/cus_46X1iCm5JBayfU" \
       -u sk_test_BQokikJOvBiI2HlWgH4olfQ2:
```

The URL is built as `object/identifier` – customer `1` would be found at `customers/1`, customer `foo` would be found at `customers/foo`, and so on.

Performing actions on this particular customer becomes changing the HTTP verb from `GET` to the action you want. Stripe's API uses `POST` for updating.

```
$ curl https://api.stripe.com/v1/customers/cus_46X1iCm5JBayfU \
       -u sk_test_BQokikJOvBiI2HlWgH4olfQ2: \
       -d "description=Customer for test@example.com"
```

Note: There exists a `PATCH` verb which developers could implement for updating instead.

Deleting a customer uses the `DELETE` verb:

```
$ curl https://api.stripe.com/v1/customers/cus_46X1iCm5JBayfU \
       -u sk_test_BQokikJOvBiI2HlWgH4olfQ2: \
       -X DELETE
```

Note: Use of `-X` overrides the HTTP verb that cURL uses.

Real World Example: Tesla Model S

The Tesla Model S' API mostly follows this system, where vehicles are referred to by ID in the URI, but a request to a function endpoint under that vehicle will perform an action.

You can fetch resources as you expect:

```
$ curl https://portal.vn.teslamotors.com/vehicles/1/command/gui_settings

{
   "gui_distance_units": "mi/hr",
   "gui_temperature_units": "F",
   "gui_charge_rate_units": "mi/hr",
   "gui_24_hour_time": false,
   "gui_range_display": "Rated"
}
```

Running functions (which, in this case, does something in the real world!) is also possible by POST requests:

```
$ curl -x POST https://portal.vn.teslamotors.com/vehicles/1/command/honk_horn

{
   "result": true,
```

```
    "reason": ""
}
```

Resources, not functions

The benefit of such a layout is that the reference to any particular object always stays the same. There is no 'leaking through' of the framework or the implementation, since you are sending data to a resource, not running a 'function' to mutate/query it.

3.3 Pragmatic REST

There is a lot of debate about what "REST" is, and what makes a REST-ful API. Arguments such as what the `POST`, `PUT`, and `PATCH` methods should do and whether function endpoints are RESTful.

My opinion on all this is that REST, like all other design patterns, is only useful as long as it is pragmatic.

3.3.1 What is REST in the first place?

REST – meaning Representational State Transfer – is when state transforms (such as 'create', 'update', or 'delete') or other idempotent operations (such as 'read') are mapped to the standard HTTP methods.

This makes it very useful in CRUD systems ("Create, Read, Update, Delete", the common life cycle of a general data object).

3.3.2 Method in the madness

If you read **RFC 7231**, the HTTP/1.1 Semantics RFC (which obsoletes **RFC 2616**, the original HTTP/1.1 RFC), it says that:

> The GET method requests transfer of a current selected representation for the target resource.
>
> The POST method requests that the target resource process the representation enclosed in the request according to the resource's own specific semantics.
>
> The PUT method requests that the state of the target resource be created or replaced with the state defined by the representation enclosed in the request message payload.
>
> The DELETE method requests that the origin server remove the association between the target resource and its current functionality.

RFC 5789 further extends **RFC 7231**/**RFC 2616** and says:

> The PATCH method requests that a set of changes described in the request entity be applied to the resource identified by the Request-URI.

In the context of a CRUD system, this means:

- `PUT` is used when the resource's ID is known. `POST` can be used to create by sending the desired new data to a 'list resource', which will return the information required.
- `GET` reads.
- `PUT` updates in entirety and `PATCH` updates portions. `POST` can be used to do either, and is defined by the resource.
- `DELETE` does what it says on the tin.

3.3.3 As long as it makes sense, don't let this bind you

The best thing for any service to do is act according to the **principal of least surprise**. That is, provided a list of endpoints, what different HTTP methods do should be obvious, or make sense with minimal effort. Following the REST paradigm to the letter can help fulfill this principal of least surprise for those people that know it, but systems can use less methods where it makes sense, or such fine-grained control will not be required.

3.4 Authentication

For APIs that authenticate directly, these options are HTTP BASIC (*only* to be used if the transport is secure) or HMAC. HTTP Digest requires challenge-response, which in my opinion makes it less useful for REST APIs.

It goes without saying that your API should provide *security* via TLS, to protect these secret access keys.

If you use HTTP BASIC, do **not** use the secret set by the user – generate an API key instead. With an API key, you can:

- track what API consumer accesses what,
- revoke compromised creds without causing the user undue pain,
- not leak the user's password (which has a high probability of being reused!) in case the transport becomes insecure,
- and separate the user's interactive logon creds (eg. your website) with their un-interactive logon creds (eg. smartphone/desktop applications, external web services).

3.5 Security

3.5.1 User data should not be sniffable

Users of your service should be protected by high-grade transport security (TLS/SSL). Your service should get at least an A on Qualys SSL Test, and all access to your API should be over TLS.

3.5.2 User-set secrets should not be recoverable

User passwords should never be stored in plain text. Use PBKDF2, bcrypt, or scrypt to protect them, as users may reuse username/password combinations

API keys should also not be recoverable through your interface – encourage your users to connect to your service using a new key for each device. Since they are random, in the case of a data breach, all API keys can be purged and recreated.

If your API service is for an application, make your app create an API key upon authentication, and use that for authenticating to the service. This prevents having to store user secrets on the device, and allows your users to revoke access from old devices.

3.6 Deconstructing the Linode DNSManager API

Linode provides a service to their customers called DNS Manager. This allows you to create and modify name server records to be served from Linode's DNS infrastructure, without having to maintain your own name server.

Deconstructing, analysing and reconstructing this API is the topic of this Technical.

3.6.1 What does this API achieve?

As mentioned above, this API allows users to create and modify name server records. To achieve this, you should be able to do the following things:

- List, create, update and delete master zones (domains)
- List, create, update and delete slave zones (which replicate from master zones)
- List, create, update and delete records under master zones

Records may also have a type (for example `A`, the IPv4 record, or `AAAA`, the IPv6 record), and types have differing requirements, so the data for each needs to be captured.

3.6.2 How does the API achieve this?

The DNSManager API is a Single Endpoint API (see *Layout* for details) that is shared with Linode's other APIs.

You send it requests to its endpoint (`https://api.linode.com/`) with query arguments that specify the function you wish to run, along with some arguments that match the specific function.

Authentication is done by either providing an API key as the password in a HTTP BASIC Authorization header, or putting it in the `api_key` query argument.

Transport security is provided via TLS, with support for ECDHE (and therefore forward secrecy).

The response to each request is a JSON object, consisting of three keys:

- `DATA` – the data of the response
- `ACTION` – the action that was run
- `ERRORARRAY` – an array of error messages, empty if successful. The errors are in therwise a list of dicts containing the keys `ERRORCODE` and `ERRORMESSAGE`, which is the code and the human readable message respectively.

3.6.3 How is the API used?

An API key is required to use the Linode API. One can be got from their web interface, using the `user.getapikey` function.

```
$ curl "https://api.linode.com/" \
       -d "api_action=user.getapikey" \
       -d "username=hawkowl" \
       -d "password=7yId7UoGhsnYh1k"
```

This will respond with something similar to the following:

```
{
   "ERRORARRAY": [],
   "ACTION": "user.getapikey",
   "DATA": {
      "USERNAME": "hawkowl",
      "API_KEY": "SECRETKEY"
   }
}
```

Note: All instances of `SECRET_KEY` would be where a valid Linode API key would be. It's much too long to display inline (60+ characters).

Linode provides an "echo" function for testing.

```
$ curl "https://api.linode.com/" \
       -d "api_key=SECRETKEY" \
       -d "api_action=test.echo" \
       -d "foo=bar"
```

Since `test.echo` function simply responds with what it was given, that request will respond with this on success:

```
{
   "ERRORARRAY": [],
   "ACTION": "test.echo",
   "DATA": {
      "foo": "bar"
   }
}
```

If something goes wrong, it will respond with an error instead:

```
 {
     "ERRORARRAY": [{
         "ERRORCODE": 4,
         "ERRORMESSAGE": "Authentication failed"
     }],
     "ACTION": "test.echo",
     "DATA": {}
}
```

3.6.4 Using the API in context

To create a domain, we need to use the `domain.create` method. This takes a number of arguments, but a working command is below.

Note: The API docs for Linode's `domain.create` method say that `CustomerID` is required. This is wrong.

```
$ curl "https://api.linode.com/" \
       -d "api_key=SECRETKEY" \
       -d "api_action=domain.create" \
       -d "Domain=mycoolawesomesite.net" \
       -d "Type=master" \
       -d "SOA_Email=hawkowl@atleastfornow.net"

{
    "ERRORARRAY": [],
    "ACTION": "domain.create"
    "DATA": {
        "DomainID": 12345
    }
}
```

`DomainID` is what you want to hold onto. This is the ID of your new domain, and you will need it to query it, delete it, or add entries to it.

We can query it like this:

```
$ curl "https://api.linode.com/" \
       -d "api_key=SECRETKEY" \
       -d "api_action=domain.list" \
       -d "DomainID=12345"

{
    "ERRORARRAY": [],
    "ACTION": "domain.list",
```

```
    "DATA": [{
        "DOMAINID": 12345,
        "DESCRIPTION": "",
        "EXPIRE_SEC": 0,
        "RETRY_SEC": 0,
        "STATUS": 1,
        "LPM_DISPLAYGROUP": "",
        "MASTER_IPS": "",
        "REFRESH_SEC": 0,
        "SOA_EMAIL": "hawkowl@atleastfornow.net",
        "TTL_SEC": 0,
        "DOMAIN": "mycoolawesomesite.net",
        "AXFR_IPS": "none",
        "TYPE": "master"
    }]
}
```

Note: Not giving the `DomainID` key will make it return all domains under your account.

We can then add what Linode calls "resources" to this domain, such as subdomains.

```
$ curl "https://api.linode.com/" \
       -d "api_key=SECRETKEY" \
       -d "api_action=domain.resource.create" \
       -d "DomainID=12345" \
       -d "Type=A" \
       -d "Name=www" \
       -d "Target=203.0.113.27"

{
     "ERRORARRAY": [],
     "ACTION": "domain.resource.create",
     "DATA": {
         "ResourceID": 7654321
     }
}
```

There are several kinds of types of resources – `A`, `AAAA`, `TXT`, `MX`, `SRV`, `NS` and `CNAME`. They all share the same resource creation function, and some of the meanings of the parameters are overloaded. None of the parameters other than `Type` or the `DomainID` are marked as universally required in the documentation, requiring you to read the description to see if it applies to the type you are creating.

For instance, the Target parameter has the following docs:

> *When Type=MX the hostname. When Type=CNAME the target of the alias. When Type=TXT the value of the record. When Type=A or AAAA the token of '[remote_addr]' will be substituted with the IP address of the request.*

The full documentation for this function can be found on Linode's site.

Listing resources works more or less the same as `domain.list`. A `DomainID` is given to `domain.resources.list`, with an optional `ResourceID` to display only a single resource. Otherwise, all resources under that domain are given.

```
$ curl "https://api.linode.com/" \
       -d "api_key=SECRETKEY" \
       -d "api_action=domain.resource.list" \
       -d "DomainID=12345"

{
   "ERRORARRAY": [],
   "ACTION": "domain.resource.list",
   "DATA": [{
```

```
        "DOMAINID": 12345,
        "PORT": 80,
        "RESOURCEID": 7654321,
        "NAME": "www",
        "WEIGHT": 5,
        "TTL_SEC": 0,
        "TARGET": "203.0.113.27",
        "PRIORITY": 10,
        "PROTOCOL": "",
        "TYPE": "A"
    }]
}
```

3.6.5 Shortfalls of the API

As I see it, the current Linode API has the following shortfalls:

- The single endpoint is shared between all Linode API services, and there is no easy or quick way to restrict an API key to only access the DNSManager API. The docs say that you can achieve this by creating users and restricting their permissions, but I've not researched this further.
- There is no versioning of the API.
- The function-based approach makes it more complex to use the API by splitting up the reference to the object you wish to access over a method name (eg. `domain.resource.list`) and then a set of parameters, rather than having it directly in the URI.
- Creating domains and resources are more complex than required due to meanings of parameters being overloaded. The documentation isn't great at explaining what you exactly need, either.

3.6.6 Re-engineering the API

Now that we have analysed how the API works and used it in context, I will now re-engineer it from the ground up, providing a proof in concept using the Twisted asynchonous networking framework and the Saratoga API development framework.

Models

The API needs to handle a few particular data models:

- Master Zones (which can have resources)
- Slave Zones (which can not have resources)
- Resources (individual records, under a zone)

I these can be better termed as **domains**, **zone mirrors**, and **records**, respectively.

Layout

The API will be in the RFC-3986 Style, with an explicit version in the path. The whole API for this example will be dedicated to the DNSManager API. An example of the root URI for v1 would be something like `dns.api.linode.com/v1/`.

Since we have two top level models, we should have them at the root:

```
/domains
/zonemirrors
```

You can then refer to individual domains and mirrors with an ID:

```
/domains
/domains/<ID>
/zonemirrors
/zonemirrors/<ID>
```

As domains can have records, we need to be able to refer to them too:

```
/domains
/domains/<ID>
/domains/<ID>/records
/domains/<ID>/records/<ID>
/zonemirrors
/zonemirrors/<ID>
```

But since different records have incredibly disparate data models depending on the type, it might be good to keep them seperate:

```
/domains
/domains/<ID>
/domains/<ID>/A
/domains/<ID>/A/<ID>
/domains/<ID>/MX
/domains/<ID>/MX/<ID>
/domains/<ID>/NS
/domains/<ID>/NS/<ID>
/domains/<ID>/AAAA
/domains/<ID>/AAAA/<ID>
/domains/<ID>/TXT
/domains/<ID>/TXT/<ID>
/domains/<ID>/SRV
/domains/<ID>/SRV/<ID>
/domains/<ID>/CNAME
/domains/<ID>/CNAME/<ID>
/domains/<ID>/records
/zonemirrors
/zonemirrors/<ID>
```

This lets us get all of the records of a domain in one go, or all the records of a specific type on the domain. Accessing a record individually has to be done through the correct type.

This map looks a bit complicated. However, since every record type has different parameters, it makes a lot more sense to split them up. It also makes it easier to document and use, as you don't have overloaded meanings of each option.

Domains

Domains ("master zones") are core to the API – they are what everything else sits under.

Adapting from the Linode API docs, this is the domain 'model':

- Domain – required (eg. `atleastfornow.net`)
- Start of Authority email – required (eg. `hawkowl@atleastfornow.net`)
- Default Time To Live (for records that don't have a TTL specified)
- Status (eg. disabled, active)
- AXFR IPs (IP addresses allowed to transfer the zone)

From this, we can develop the following JSON Schema for creating a Domain:

```
{
    "description": "Domain -- Create",
    "type": "object",
```

```
    "required": ["domain", "soa"],
    "properties": {
        "domain": {
            "title": "The base for this domain.",
            "type": "string",
            "format": "hostname"
        },
        "soa": {
            "title": "Start Of Authority Email.",
            "type": "string",
            "format": "email"
        },
        "default_ttl": {
            "title": "Default TTL for records, in seconds.",
            "type": "integer"
        },
        "status": {
            "title": "The status of the domain.",
            "type": "string",
            "enum": ["active", "inactive"]
        },
        "axfr": {
            "title": "IP addresses which may AXFR the domain.",
            "oneOf": [
                {
                    "type": "array",
                    "uniqueItems": true,
                    "items": {
                        "anyOf": [
                            {
                                "type": "string",
                                "format": "ipv4"
                            },
                            {
                                "type": "string",
                                "format": "ipv6"
                            }
                        ]
                    }
                },
                {
                    "type": "string",
                    "length": 0
                }
            ]

        }
    }
}
```

To put it simply, this means that a domain is an `object` (`dict`), and can have these properties. Out of those properties, `domain` and `soa` **must** be given. The rest are optional, and have defaults if they are not provided.

But since we also want to validate *outputs* as well as inputs, lets also write a JSON Schema for the response. (It's generally good to respond as if they immediately did a `GET` request on the new resource.)

```
{
    "description": "Domain -- Create Response",
    "type": "object",
    "properties": {
        "id": {
            "title": "The ID of this domain.",
            "type": "integer"
```

```
        },
        "domain": {
            "title": "The base for this domain.",
            "type": "string",
            "format": "hostname"
        },
        "soa": {
            "title": "Start Of Authority Email.",
            "type": "string",
            "format": "email"
        },
        "default_ttl": {
            "title": "Default TTL for records, in seconds.",
            "type": "integer"
        },
        "status": {
            "title": "The status of the domain.",
            "type": "string",
            "enum": ["active", "inactive"]
        },
        "axfr": {
            "title": "IP addresses which may AXFR the domain.",
            "type": "array",
            "uniqueItems": true,
            "items": {
                "oneOf": [
                    { "format": "ipv4" },
                    { "format": "ipv6" }
                ]
            }
        }
    }
}
```

They are nearly exactly similar, barring the inclusion of `id` in the response. By checking both the input and output, it is less likely that a bug will cause the API to return incorrect data or data that it shouldn't.

R