Contents

Chapter 1

Platform.sh Git (and/as) Hypermedia

Our job in life is creating silver bullets that fit any foot, right?

We call these conferences API conferences but really its all about CRUD oriented JSONy REST right? With the cool boyz telling you its all about hypermedia.

If you haven't caught up yet its about: Affordance (discoverability), Developer Experience, Reducing Client maintenance cost, Reducing migration headaches.. so its good and we should always use it for everything? We'll see about that.

HTTP as defined in Wikipedia: The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems.

Now this doesn't look at all like whatever we are doing right now, right? When we say do APIs and we, say, create JSON over HTTP interfaces (JSON being not extremely hypertext friendly as a specification....) we are miles away from the definition of HTTP or even its design goals

And, if we go to the definition of API, either Advanced Programming Interface or Application programming interface. This seems to be a whole different level, a differ-

ent abstraction then whatever HTTP is...; An API is a way for one program to program another one. To access it. To mechanize it. It is nothing more then a method to invoke a method on an object. Or to be more late 2014... a way to pass an argument to a function.

The reason we are talking to you guys so much about REST is that in real world applications we have discovered that the HTTP model represents an excellent tradeoff as a kind of universal middleware for invoking *remote* functions. That is has extremely beneficial side effects in terms of system architecture, as it forces developers to think in strongly decoupled, concurrent, fail-tolerant systems. We like the universality of HTTP. We like the simplicity. And we like the fact we have the probably hugest human infrastructure ever that just exclusively with doing this HTTP thing efficiently.

But everything has semantics. What does that phrase mean? it means everything has meaning in it. Everything communicates a worldview. Ideas about the world. Your program has concepts, and it tries to call another program that has other concepts through an intermediary protocol that has its own.

Think about how many times you were yelled at by an API expert/evangelist because "your API does not respect HTTP semantics!" Idempotency! POST is not PATCH which is not PUT you nitwit! This should be "203 Non-Authoritative Information" not "200 OK" ... and the such..

APIs need to bridge gaps between the semantics of two different systems. When we use a middleware representation such as JSON over HTTP it means the semantics are translated twice. So what is usually means is the following conversation :

World View 1 –> World View 1 as represented as Json over HTTP –> World View 1 as represented in World View 2 and World View 2 –> World View 2 as represented as Json over HTTP –> World View 2 as represented in World View 1

Now lets do the trick to pass the preceding twice through

google translate (English->French->English) which gives us:

World View 1 -> World View as represented as a Json over HTTP -> World View one as represented in the world View 2 and World View 2 -> World View 2 as shown as Json over HTTP -> World View 2 as shown in the world View 1

Not bad at all right? we have some type issues (1 became One, an instance became a class World View 2 to World View etc..). In our case French served as the intermediary representation. Now lets use Hebrew! The same trick gives us:

View in 1 -> World View 1 as represented as JSON over HTTP -> World View 1 as represented in View 2 And View the world 2 -> World View 2 as represented about JSON HTTP -> World View 2 as represented in View 1

Well here we are totally broken right? this is not going to work. The semantics of the intermediary system matter. It represents the total Impedance of the system (you know the function over reactance and resistance).

Now lets look at a very important thing in semantics which is "Ontological", how do you determine what a thing "is", its "identity function". Through two classes of underlying technologies that could be uses as an API:

In the semantics of HTTP, as an hypermedia system an object, a thing in the world is the address of the thing. In the semantics of GIT, as an content addressable system a thing in the world is a function over its content.

By the way, this is one reason for which often enough you will see in Hypermedia APIS that an object's serialization will include a "self" relation, which will be a URL.

Referencing things is even more important. A thing referenced in git, being referenced by its real content, will always be at byte-level the "same". If the thing has evolved, it will have a new address, and we will need to update the reference to it in another object "the tree" that will, by itself will be known by the hash of its content. We has a very strict sense of change here.

Referencing in HTTP is totally different, when we use an href, a "link" in programming terms we have something like a "promise" a local representation of a remote function call where the server:port part represents the function and the rest of the URL with its query string represent the arguments to the function. But these are not "pure" functions. Every time we call it we might get a different response. In most cases we won't even be calling the same "function" dependent on oh, so many factors, as we have an enormous stack of technologies to control and represent this changing over time (HTTP caching... and who really returns "203 Non-Authoritative Information").

So, when a thing references another thing in HTTP it does so through a link, saying this remote system, if you ask it for a "resource" with certain parameters will give you the thing.

This is just fine, not only fine, wonderful, in most use-cases. Which is why you should totally default to a RESTful hypermedia implementation using if possible someone else's standard for it (hint: http://jsonapi.org).

... but sometimes what "gets lost in translation" the fact of having HTTP semantics in the middle is going to present some very important challenges and some very awkward implementations. Enough theory. Let's go real world on this.

We said, an API is a way to call a function on a program.

Let's imagine the software on which we want to call methods on is a PAAS (a platform as a service). Meaning software that should be able to run any other software (The ultimate meta-shit). Now, would we want the invocation system in the middle, our middleware semantics to be something that might have different meanings, different answers depending on the weather? Surely not. In order for a program to be able to run any other program it must be totally predictable, absolutely repeatable.

But lets also take into account that software does want to change over time. We want to be able to add new features right? A guaranty of "sameness" is not enough.

And when you consider these days an application is not

just a single app server that is self contained, but a real-world application is a whole infrastructure. With multiple application servers, databases, caches, message queues.. and data, files etc.. This is means repeatability is becoming non trivial. Because we are not dealing with pure functions with no side effects, but on the contrary something that is extremely state heavy. The stateless HTTP protocol with its dangling "promises" is not only unhelpful here it is going to hinder us greatly.

But git semantics, for this, are really really great; We want a system with this precise state (which means a graph of objects that have known contents) and we want to mutate just a single element which should yield a new "object" with just this difference.

So in http://platform.sh the API is git. And everything is represented in git concepts.

When you do in http://platform.sh

```
git checkout -b "new_cool_feature"
```

tapety tap tap tap . . . code . . . code . . . code . . . tapety tap tap . . . add a search engine . . .

```
git push origin "new_cool_feature"
```

You just used the git protocol, but precisely like with an HTTP API where when you do a "PUT" you are not PUTTING a file on a static WEBDAV server, but manipulating calling a function on an application server, invoking a method with specific semantics, here you will be talking to an application server (which is a highly consistent distributed Pythonian object framework) and telling it "I want to clone an existing infrastructure apply a set of changes and create a new one with just these changes".

So platform.sh will do just that; It will take the running application cluster, get a state representation of every single element in it (again represented as a hash of its content), and create a new application cluster with just the changes (basically mutating just the tree of references to these objects known by their content). Because of this, because everything that is the "same" stays the same, this operation is incredibly fast; We can clone a running appli-

cation, composed of multiple machines, with its databases and caches and what not, in under 30 seconds.

This doesn't mean, good heavens, we dump and commit the gigabytes of content of the database into a git server, madness, no it means we are leveraging the semantics of git. What our "git" server represents is not the application code, but the whole infrastructure that will run it. With the data.

From a developer's point of view this means he can test in isolation a single feature branch, knowing that only this and nothing else changed. That there are no more "dev" and "staging" servers, which are unholy mixtures of broken code, hot-fixes and the next-release branches. It means staging servers become as disposable as topic branches.

From a sysadmin's point of view this means all builds are repeatable. There are no "production only bugs". And if it works on staging it will on production, because they will be the same.

When you want to inspect the history, the whole history of your production system, every single change that was ever made to it. Well `git log`.

Using a protocol that has semantics that marry well our domain makes for the API to be so much easier to understand `git push origin "new_cool_feature"` doesn't mean precisely what you'd expect it to ... because you are used to git as being an API to just store code, but understanding what it does here is easy.... (Hey, with webhooks we already use it to do other stuff right? You expect it also to run tests on Travis, or post a message on Hipchat.)

But.. you might say, "great guys this is nifty and cool but as a developer I really, really don't want to have to deal with a complex system such as git, and I don't want to know what it means to rebase an application cluster". This will be true. Because an API is as much about having clear, efficient, semantics, as it is about ease of use and discoverability. APIs are not perfect, they are perfect for a use case. And making a dashboard that will

iterate over stuff from this representation... is well, just not practical. Making a web page that directly speaks git in the browser.... well Tim Caswell (creationix) did just that (https://github.com/creationix/js-git) but well.. this is going to be awkward.

So of course we also expose a Hypermedia API. It represents precisely the same thing; So when you PUT, what will happen is really a `commit` or a `branch` depending on the object you are manipulating. So basically now, you can do whatever you could through git.. also through a REST API. But because this is just a wrapper, a proxy on top of the Git protocol.. it means you are doing it with clear, correct semantics.

We try to apply the "principle of least surprise and maximum immediate joy". So `GET https://eu.platform.sh/api/projects` will give you in a single request the list of your projects, with all the data you need, and the next steps you can take in a convenient bundle.

```
[{
    "id": "jrq5lx44wtatm",
    "_links": {
                "self":  { "href": "/api/projects/jrq5lx44
       "#edit": { "href": "/api/projects/jrq5lx44wtatm"},
                "#ui":   { "href": "https://eu.platform.sh
       "#manage-domains": { "href": "/api/projects/jrq5lx
       "#manage-access": { "href": "/api/projects/jrq5lx4
    },
    "status": {
      "message": "ok",
      "code": "provisioned"
    },
    "repository": {
      "url": "jrq5lx44wtatm@git.eu.platform.sh:jrq5lx44wta
      "client_ssh_key": "ssh-rsa AAAA[...]XdHaaj0]SPwZ jrq
    "title": "Sonata",
    "created_at": "2014-11-13T09:31:31.368000+00:00",
    "updated_at": "2014-11-21T15:44:19.666000+00:00",
    "default_domain": null,
```

```
    "owner": "766e306d-0f9a-4d38-b22f-cc6c5f070a14",
    "subscription": {
      "user_licenses": 5,
      "restricted": false,
      "included_users": 1,
      "storage": 15360,
      "environments": 9,
      "license_uri": "https://marketplace.commerceguys.com/api/platform/licenses/7239",
      "plan": "standard",
      "suspended": false,
      "subscription_management_uri": "https://marketplace.commerceguys.com/user/7/orders"
    }
  }, {
    "id": "wawm7pg7grgew",
    ["..."]
  }, {
    "id": "thpgd2cdvbpm6",
    ["..."]
  }
}]
```

And, we try to bridge the semantics.. git is a statefull connected protocol. While HTTP is a stateless one.. so for example how would you go around handling Long-running operations ?

To keep the API reactive, we handle those with a simple protocol: when an operation is long running, we return `202 Accept` and a link to an activity resource::

```
Status: 202 Accept
Location: https://eu1.c-g.io/activity/xxxxxxxxxxx
```

The activity resource will be polled by the client (or watched through Event Source) and will contain the information about the progress of the task::

```
{
    // [...]

    "status": {
        // Is the operation complete (true/false)?
        "completed": false,
```

```
        // Progress of the operation in percent.
        "progress": 8
    }
}
```

At the end of the processing, the activity will be updated with either a failure state::

```
{
    // [...]

    "status": {
        // Is the operation complete (true/false)?
        "completed": true,
        // Progress of the operation in percent.
        "progress": 100,
        "result": "failure",
        "result message": "Failed to create resource",
        "result data": {
            "xxxx": "yyyyy"
        }
    }
}
```

or a success state::

```
{
    // [...]

    "status": {
        // Is the operation complete (true/false)?
        "completed": true,
        // Progress of the operation in percent.
        "progress": 100,
        "result": "success",
        "result message": "Project {{name}} created succes
        "result data": {
            "xxxx": "yyyyy"
        }
    }
}
```

Let's see this in action.

So... think well on the semantics of your API, default to hypermedia, but if needed don't be ashamed to expose a domain specific API... later when you wrap it in a REST one.. the design of this one ... will be much, much better.

AND.. if ever you wanted a PAAS that can generate a staging server in 30 seconds, check us out. Or.. if it sounds cool and you like GoLang and Python... we are hiring...