Principles of APIs

What is an API?

An *Application Programming Interface* is a means for two software applications to communicate with each other. If you imagine two people with old-style can-and-string 'phones':

Then the two people are two different software applications, and the can-and-string phone is the API.

APIs, like a conversation, rely on *request* and *response*.

No REST for the wicked

REST - *Representational state transfer*. You may well hear about things being 'RESTful' as well, which just means that object has REST properties. REST describes a concept / series of concepts that pretty much anyone who has used the internet before has already realised, but just didn't know that it had a name. REST is based on URIs (Uniform Resource Identifier, of which a URL is a specific type) and the HTTP protocol - this page has been served up using REST principles. A REST API utilises these sample principles in order to allow a two-way conversation.

There are six main principles behind REST:

1. **Uniform interface -** Individual resources are identified using URLS. The resources are themselves different from the representation (XML, JSON, HTML) sent to the client. The client can manipulate the resource through the representations provided they have the permissions. Each message sent between the client and the server is self-descriptive and includes enough information to describe how it is to be processed. The hypermedia (that is hyperlinks and hypertext) act as the engine for state transfer (HATEOAS - see below)
2. **Cacheable -** Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.
3. **Stateless -** None of the client's context is to be stored on the server side between the request. All of the information necessary to service the request is contained in the URL, query parameters, body or headers.
4. **Client-Server -** A separation of concerns which means that, for example, clients are not concerned with data storage, which remains internal to each server, so that the portability of client code is improved. Servers are not concerned with the user interface or user state, so that servers can be simpler and more scalable. Servers and clients may also be replaced and developed independently, as long as the interface is not altered.
5. **Layered system -** An architecture can be composed of hierarchical layers by constraining component behaviour such that each component cannot “see” beyond the immediate layer with which they are interacting
6. **Code on demand -** An optional constraint where the server temporarily extends the functionality of a client by the transfer of executable code.

Many / most REST APIs are designed to conform to the HAL standard, which forms the basis of HATEOAS (below): This is required to reach level

3 in the Richardson Maturity Model (https://martinfowler.com/articles/richardsonMaturityModel.html), which can be summed up as

Level 0 - The API uses HTTP as the transport mechanism. It's a good idea to make sure you're brushed up on your HTTP status codes as well; a successful API call should pass a HTTP 200 OK in the header - other codes tell you important information as to what happened.

Level 1 - Resources. REST resources are indicated by their URI. An example product catalogue API may have the the URL https://www.river-island.com/api/product/ * When called this may return all of the products in the catalogue. However if I wanted ust one specific product with an ID of '1234' then the resource for this would be https://www.river-island.com/api/product/1234/

**I purposely did not include a version in the URL. See 'versioning' below*

Level 2 - Use of HTTP verbs. See CRUD (below).

Level 3 - Hypermedia Controls. This essentially just means that each API URI will always give you a list of the other available URIs associated with that request; see the HATEOAS section below for a more in-depth overview.

Pile of CRUD

This is simply enough the 4 main operators of persistent storage - Create, Read, Update, Delete. Whenever calling a REST API you'll be doing one of these operations, but as with Single Responsibility Principle you should only execute one of these operations with one request. If you want to do an update and a delete then make two separate requests (to two separate URLs - see above about REST).

The CRUD operators translate into HTTP with the following:

Operation	SQL	HTTP	Idempotent?
Create	INSERT	PUT / POST	No
Read (Retrieve)	SELECT	GET	Yes
Update (Modify)	UPDATE	PUT / POST	No
Delete (Destroy)	DELETE	DELETE	Technically yes

HATEOAS

Hypertext* As The Engione Of Application State essentially just means that hypertext should be used to find your way through the API.

**Some people say 'hypermedia' rather than 'hypertext'. Potato / potato.*

Taking a web page as an example, you have hypertext in it (links), the naming of which (should) tell you what they do. So an 'about us' link should go to a page telling you some information about that company, etc. The point here is that you don't need an instruction manual for navigating the website; it's all obvious, straightforward, *has convention*... You can do the same for APIs, and this is HATEOAS: it provides a way of dynamically discovering available actions of an API according to an agreed convention. You can see this in the response from an API call, where you get a 'links' data node. This should contain the relation and the hypertext reference, e.g. (from Spring Data):

```
{
    "content": [ {
        "price": 499.00,
        "description": "Apple tablet device",
        "name": "iPad",
        "links": [ {
            "rel": "self",
            "href": "http://localhost:8080/product/1"
        } ],
        "attributes": {
            "connector": "socket"
        }
    }, {
        "price": 49.00,
        "description": "Dock for iPhone/iPad",
        "name": "Dock",
        "links": [ {
            "rel": "self",
            "href": "http://localhost:8080/product/3"
        } ],
        "attributes": {
            "connector": "plug"
        }
    } ],
    "links": [ {
        "rel": "product.search",
        "href": "http://localhost:8080/product/search"
    } ]
}
```

There are self-referencing links here (rel: self) which just provide the absolute URI to this data, plus at the end the URI to return a product search. If there was an available option from this point to delete a product, it could look like

{

"rel": "product.delete",

"href": "http://localhost:8080/product/delete/3"

}

Another example, this one from Wikipedia. Imagine a banking API. This should list all the possible actions stemming from the account resource:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: ...

<?xml version="1.0"?>
<account>
    <account_number>12345</account_number>
    <balance currency="usd">100.00</balance>
    <link rel="deposit" href="https://bank.example.com/accounts/12345/deposit" />
    <link rel="withdraw" href="https://bank.example.com/accounts/12345/withdraw" />
    <link rel="transfer" href="https://bank.example.com/accounts/12345/transfer" />
    <link rel="close" href="https://bank.example.com/accounts/12345/close" />
</account>
```

Make sense? This is Level 3 of the Richardson Maturity Model (https://martinfowler.com/articles/richardsonMaturityModel.html).

This is a very good article expanding on the Maturity Model: https://8thlight.com/blog/jason-desrosiers/2018/05/30/the-hypermedia-maturity-model.html

Mastering your Swagger

Swagger shouldn't be confused with OpenAPI - they are very closely related, but different things. From swagger.io:

- OpenAPI = Specification
- Swagger = Tools for implementing the specification

Or to put it anopther way, OpenAPI is a format of describing / documenting a REST API. Swagger is a toolset that helps to design, build *and* document REST APIs... OpenAPI provides for REST APIs what WSDL provides for SOAP.

See here for for some examples of APIs documented using OpenAPI: https://github.com/OAI/OpenAPI-Specification/tree/master/examples/v3.0

Swagger builds on this providing

- Swagger Editor – browser-based editor where you can write OpenAPI specs.
- Swagger UI – renders OpenAPI specs as interactive API documentation.
- Swagger Codegen – generates server stubs and client libraries from an OpenAPI spec.

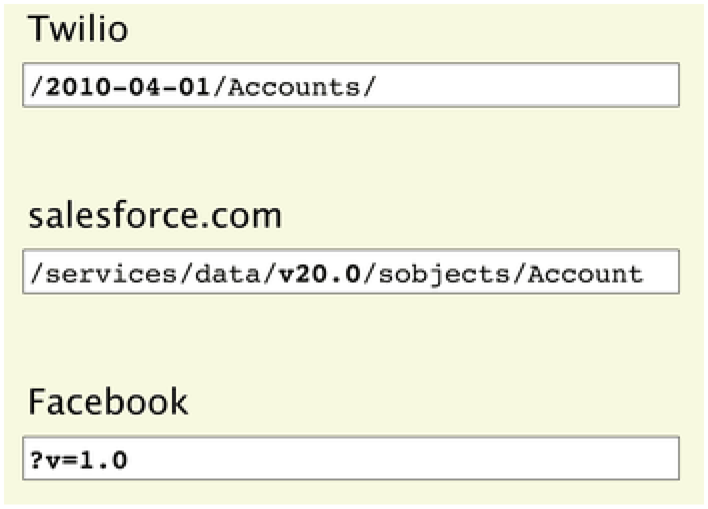
Versioning

APIs will likely go through mutliple versions as features are added / deprecated: It's therefore important to keep track of which version of an API the user is requesting. Where this becomes contentious is how to keep track of this (and the arguments can get quite heated!)

There are two main schools of thought -

Version numbers should be part of the URI, e.g. https://riverisland.com/api/v2/product/

Some people include the version number in the URI in different formats, e.g.


Twilio

```
/2010-04-01/Accounts/
```

salesforce.com

```
/services/data/v20.0/sobjects/Account
```

Facebook

```
?v=1.0
```

But the main popint is that it's part of the URI.

The other main school of thought is that the URI should be immutable, and therefore the versioning should be specified as part of the content negotiation / content-type, e.g.

```
GET /product/123 HTTP/1.1
Accept: application/vnd.riverisland.myapp.product-v2+xml

HTTP/1.1 200 OK
Content-Type: application/vnd.riverisland.myapp.product-v2+xml

<product>

    <productName>some shoes</productName>

    <SKUID>1234</SKUID>

</product>
```

My take is that the second method - the content negotiation one - is the better way of doing it, as changing URIs can break HATEOAS (unless you're updating your URIs dynamically / automatically). That said, most people take the URI route, so what do I know?

Lifted from http://www.lexicalscope.com/blog/2012/03/12/how-are-rest-apis-versioned/ :

Versioning strategies in discussions

POST	VERSIONING
Stack Overflow 1	URI
Stack Overflow 2	Content Negotiation
blog post by Jeremy	Content Negotiation
ycombinator discussion	some opinions both ways
Stack Overflow 3	Content Negotiation
Stack Overflow 4	Content Negotiation
notmessenger blog post	URI (against all headers)
Peter Williams	Content Negotiation (strongly against URIs)
Apigee Blog post on API versioning	URI (some discussion)
Mark Nottingham REST versioning	Recommending essentially versionless extensibility with a HATEOS approach
Nick Berardi on REST versioning	URI
Restify	“Accept-Version” header
Tom Maguire on REST versioning	Content Negotiation
Nicholas Zakas on Rest Versioning	URI
Steve Klabnik on Rest Versioning	Content Negotiation + HATEOS
Luis Rei on Rest Versioning	Content Negotiation with (;version=1.0)
kohana forum discussion on REST versioning	Many Opinions
Troy Hunt on REST versioning	Accept Header but also support custom header and URL
Paul Gear REST versioning	Recommending essentially versionless extensibility with a HATEOS approach

Versioning strategies in popular REST APIs

API NAME	VERSIONING	EXAMPLE
Twillo	date in URI	
Twitter	URI	
Atlassian	URI	
Google Search	URI	
Github API	URI/Media Type in v3	Intention is to remove versioning in favour of hypermedia – current application/vnd.github.v3
Azure	Custom Header	x-ms-version: 2011-08-18
Facebook	URI/optional versioning	graph.facebook.com/v1.0/me
Bing Maps	URI	
Google maps	unknown/strange	
Netflix	URI parameter	http://api.netflix.com/catalog/titles/series/70023522?v=1.5

Salesforce	URI with version introspection	{
		“label”:”Winter ’10”
		“version”:”20.0,
		“url”:”/services/data/v20.0,
		}

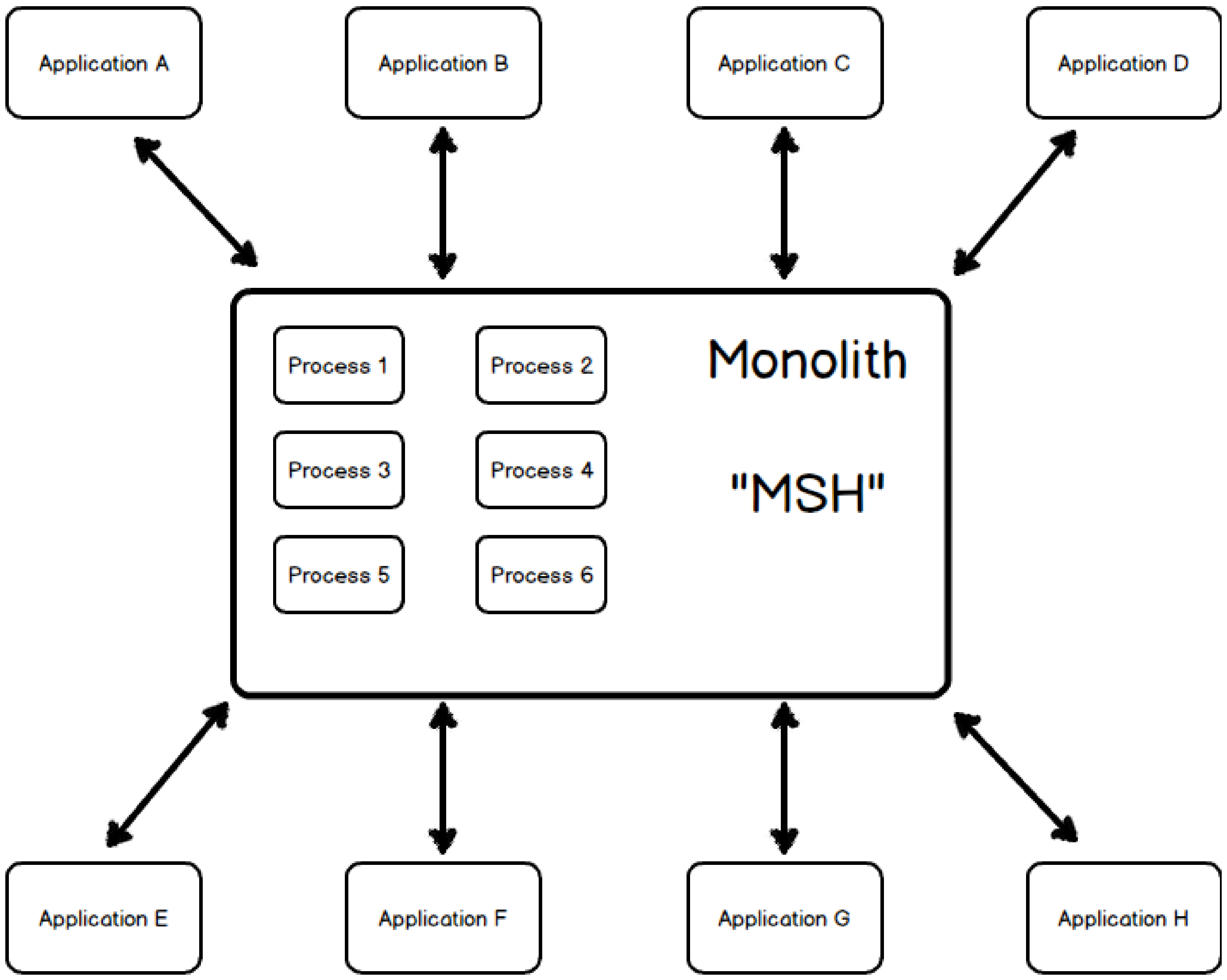
Google data API (youtube/spreadsheets/others)	URI parameter or custom header	“GData-Version: X.0” or “v=X.0”
Flickr	No versioning?	
Digg	URI	http://services.digg.com/2.0/comment.bury
Delicious	URI	https://api.del.icio.us/v1/posts/update
Last FM	URI	http://ws.audioscrobbler.com/2.0/
LinkedIn	URI	http://api.linkedin.com/v1/people/~/connections
Foursquare	URI	https://api.foursquare.com/v2/venues/40a55d80f964a52020f31ee3?oauth_token=XXX&v=Y
Freebase	URI	https://www.googleapis.com/freebase/v1/search?query=nirvana&indent=true
paypal	parameter	&VERSION=XX.0
Twitpic	URI	http://api.twitpic.com/2/upload.format
Etsy	URI	http://openapi.etsy.com/v2
Tropo	URI	https://api.tropo.com/1.0/sessions
Tumblr	URI	api.tumblr.com/v2/user/
openstreetmap	URI and response body	http://server/api/0.6/changeset/create
Ebay	URI (I think)	http://open.api.ebay.com/shopping?version=713
Reddit	No versioning?	
Groupon	URI	http://api.groupon.com/v2/channels//deals{.json|.xml}
Geonames		
Wikipedia	no versioning I think?	
Bitly	URI	https://api-ssl.bitly.com/v3/shorten
Disqus	URI	https://disqus.com/api/3.0/posts/remove.json
Yammer	URI	/api/v1
Drop Box	URI	https://api.dropbox.com/1/oauth/request_token
Amazon Simple Queue Service (Soap)	URI Parameter and WSDL URI	&Version=2011-10-01
Youtube data API versioning	URI	https://www.googleapis.com/youtube/v3

In terms of how to represent version numbers, a good standard is the semantic versioning 'MAJOR.MINOR.PATCH' (e.g. v2.1.1). The MAJOR number should change whenever breaking chnages are introduced.

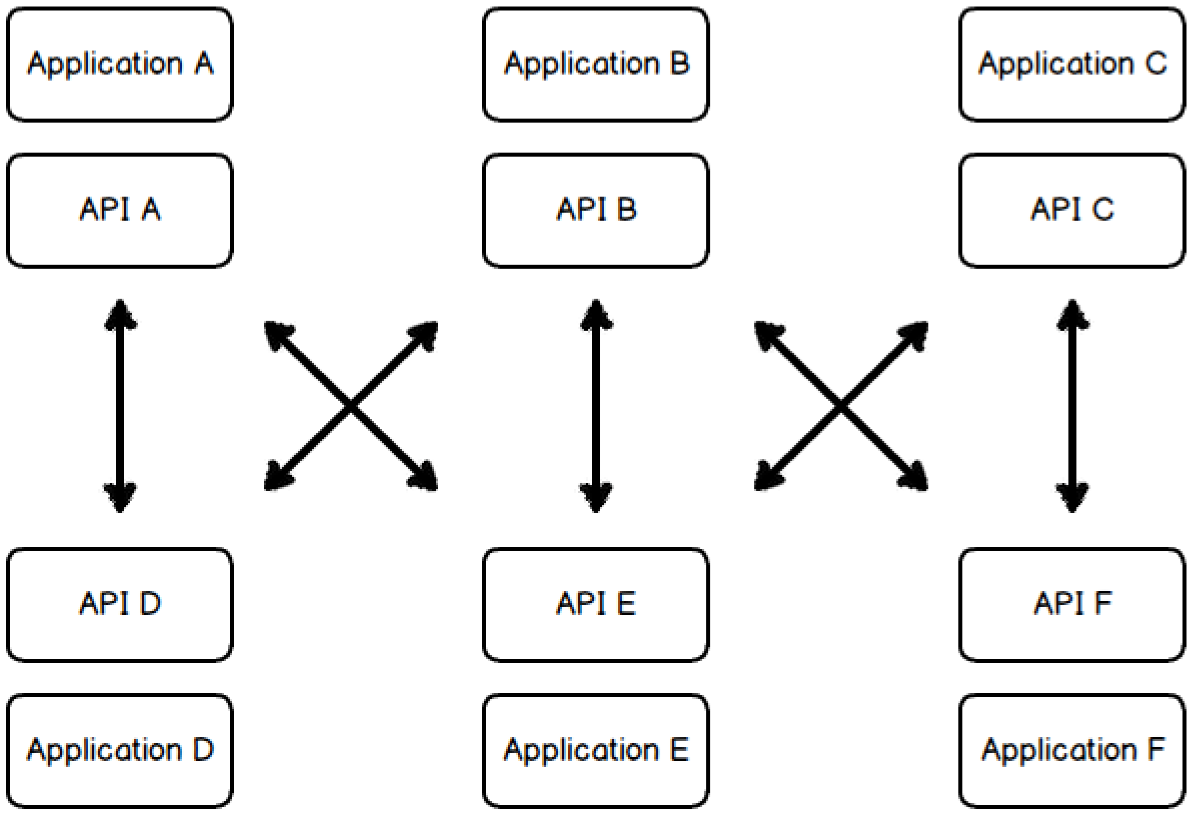
SOA and Microservices

SOA stands for 'service-oriented architecture': This is essentially a collection of services which communicate with each other, via APIs. The core premise of 'services' is about decoupling - much as you'd design a software application to have a separation of concerns, the SOA is an extrapolation of this, where each service is concerned with performing its set functionality distinctly from the other services. As long as the API between the services adheres to the agreed contract then how the service does what it does is not the concern of any of the other services: Each service is encapsulated.

Below is a diagram of a typical monolith system. There are a number of applications that submit data, and require data back out. The monolith has a number of processes that are all interconnected, sometimes to the point that the actual functioning of the monolith becomes obscure - it can be described as 'MSH': Magic S**t Happens.



The disadvantages to this should be apparent: Changing any of the processes means having to redeploy the entire monolith again, increasing effort and introducing risk. Also because the processes are all tightly coupled then changing one could have unintended / unforseen impacts on some (or all!) of the other services. This again often translates into an increasing level of risk as more changes go in, and potentially lead to a steady increase of technical debt. Eventually the monolith becomes unworkable - there is no-one who has a clear enough oview of all the processes and how they intereact with each other, and therefore any changes are approached with a high degree of trepidation and a large amount of manual testing (and hence a very slow and error-prone release cycle).



Instead a SOA de-couples all the applications / processes, and these communicate with each other via APIs. The advantage to this is that if I want to modify Application A then I can do that without affecting anything else: As long as the API it presents to the world looks and behaves the same (what we call the 'API contract') then this change is entirely transparent to the rest of the estate. There is a separation of concerns.

Further reading:

https://www.nginx.com/blog/introduction-to-microservices/

JSON and the Argonauts, or dealing with the plague of POX and washing away SOAP.

Not *all* APIs use JSON, but *all the good ones* do. XML was useful for a while, but its had its day, and now JSON is king. Essentially XML was just too 'bloated' - there's a lot of extra payload you need to craft out and send around in order to convey the data you actually want sent - JSON is a lot more compact, and that translates as faster to model, faster to parse, ceasier to cache and a simple name / value pair tanslates better from / into an application structure. Correspondingly unless there is a good reason not to then you should aim to craft out all your APIs using JSON. However you will come across older APIs that use alternatives for their transmission methods, Plain Old XML (POX) and Simple Object Access Protocol (SOAP). These have their places, and SOAP supporters will mumble things around the interoperability of WSDL, and ACID compliance, but often these are simply not good enough reasons to put up with the extra complexity and transmission overhead that a SOAP API has compared to a JSON REST API.

The 8 Fallacies of Distributed Computing

There are a few common assumptions that people tend to make when dealing with APIs / microservices / distributed computing in general... These are:

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

These aren't as relevant nowadays as they used to be but it's still good to bear them in mind, if only just to check that they aren't going to be an issue.

One aspect of SOA / microservices that should always be taken into account is what to do if a service fails: You need to design so that applications can tolerate the failure of services gracefully. (Indeed Netflix created an application called 'Chaos Monkey' speficially for testing this - Chaos Monkey randomly terminates services, and you see how well everything handles that. This later evolved into the Simian Army - a whole suite of tools which randomly terminates Amazon services, introduces latency, keeps threads locked, prevents auto-scaling, can take down entire Availability Zones or indeed Regions, etc). See also the concept of 'Chaos Engineering'.

Should I use *any* of this?

File under 'lengths of pieces of string'. Not all APIs are RESTful. Not all REST APIs use JSON. A lot of people hate HATEOAS... There is no 100% defined standard here: always use the best tool for the job.