Academic REST

Paul Zsembik

I ♥ APIs!

REST != HTTP

Overview

At the end of this you should have basic **academic** understanding of:

- **Part 1**: Brief history of web services
- **Part 2**: RESTful architecture
  - Why REST
  - What a resource is
  - How verbs and status codes are used (Hint: Http)
  - What request/response messaging is
  - Query string manipulation (searching, sorting, pagination)
  - Versioning
- **Part 3**:
  - What an API is
  - API Design
  - Path to Better APIs

In the beginning we have web services...

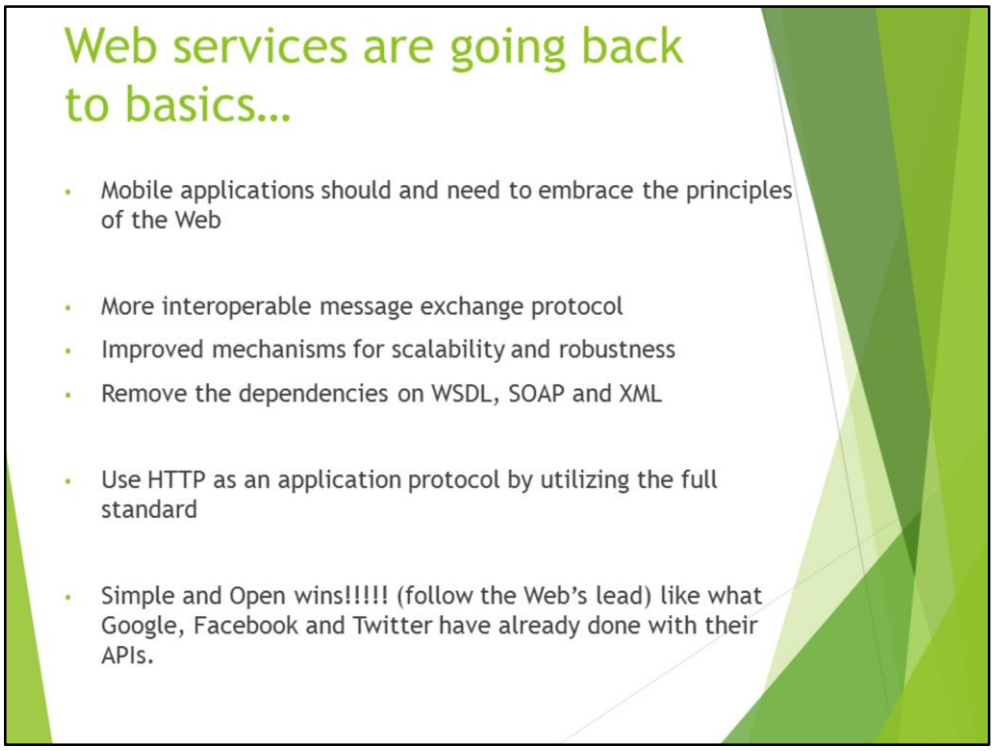
and we used them to build application silos

http://www.nitrogen-generators.com/wp-content/uploads/2014/02/grain-silos-nitrogen-generators-960x400.png

…and the vendor Gods gave us SOAP

What were the challenges?

- A few things:
  - Using HTTP as a transport protocol artificially
  - WS-* over complexity and interoperability
  - WSDL and XML dependency
  - Lack of Service Governance
  - Scalability & Performance
  - Flexibility



SOAP based API and REST based APIs

SOAP v REST

HTTP

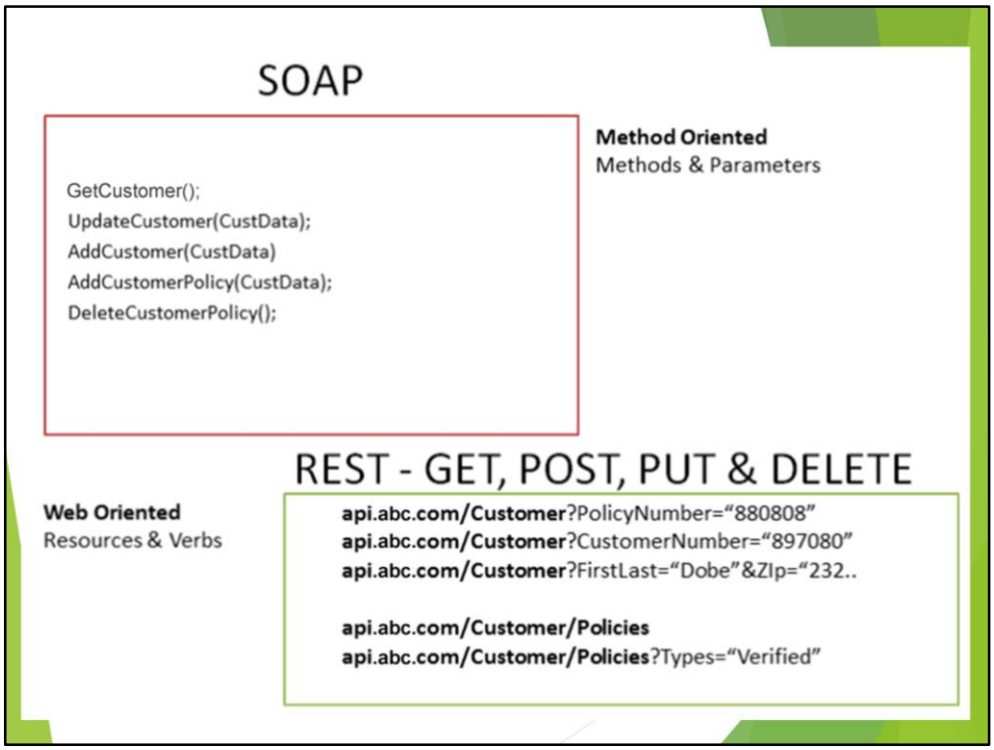
REST

Rides directly on HTTP. Plain and simple. In reality, this is all you need to send data from point A to point B and get the required response. Catch: Until something that represents a service contract is put in to place, it's kinda "anything goes".

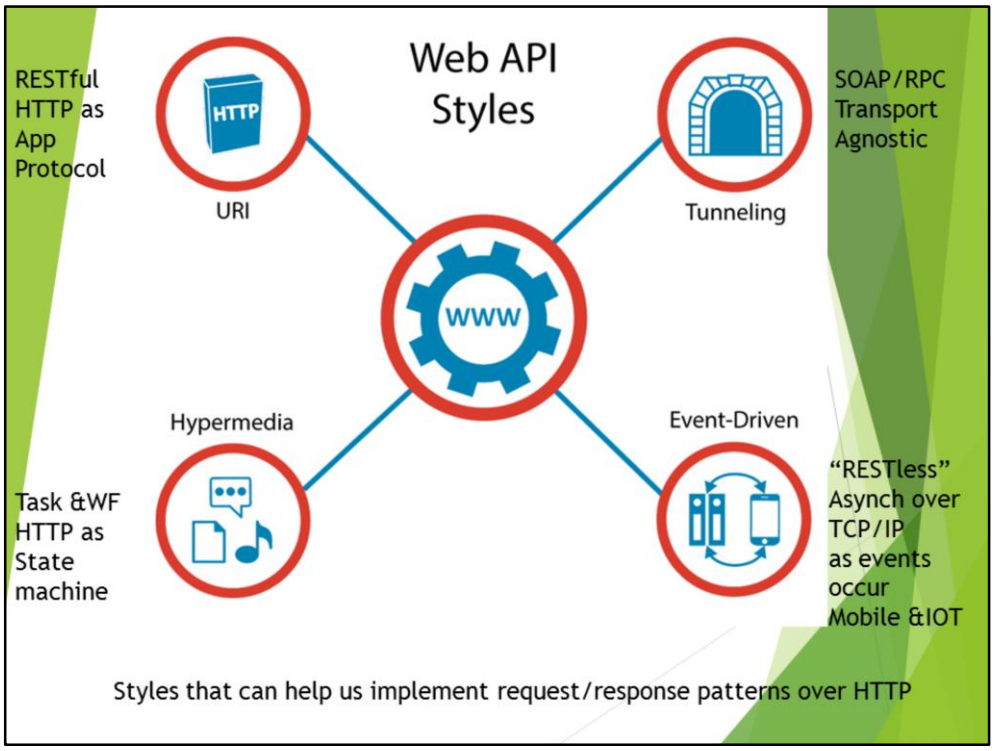
SOAP

The coach is your SOAP envelope: it wraps your data. Main strength is the presence of a contract: the WSDL. Gives you the "comfort" of easily generating artifacts. Catch: look at the complexity and added weight.

http://stackoverflow.com/questions/15805112/using-soap-and-rest



SOAP only Uses POST and sometimes GET.



http://www.apiacademy.co/resources/api-design-201-web-api-architectural-styles/
Tunneling Style
The most well-known implementation of the Tunneling API style is the SOAP messaging standard. SOAP defines an RPC-like interface for application integration and utilizes a standard called WSDL to describe the interface. Client applications can generate proxy code based on a WSDL document and make calls as if the remote component is local.
A Tunneling API normally:
Exposes an RPC-like interface and provides an interface descriptor for binding
Uses an XML-centric message format
Uses HTTP as a transport protocol for a higher application-level protocol
One of the great advantages of this style is that it is transport-agnostic. Applications are free to deliver the same SOAP message over HTTP, JMS or raw TCP/IP connections, greatly increasing flexibility. Additionally, there is a well-defined set of protocol standards in the WS-* specifications able to support a wide array of interaction behaviors and message properties.
However, the use of a transport-agonistic protocol is inefficient when it is known that network interactions will only ever take place over HTTP. In addition, SOAP and the WS-* specification set may be unfamiliar to mobile and Web application developers, resulting in decreased usage and increased development costs in these communities.

URI Style
The URI style, arguably the most familiar to application developers, exhibits these

properties:
An object- or resource-centric API is exposed
URIs and query parameters are used to identify and filter objects
CRUD (create, read, update, delete) operations are mapped to HTTP methods
URI APIs provide an intuitive, simple way for application developers to invoke requests. Well-designed APIs of this style employ “hackable” URI designs, which also act as a form of self-documentation. The reliance on the HTTP protocol can also act as an advantage for client application developers who are familiar with the protocol.
The challenge in designing a URI-style API is that it is difficult to map a complex set of application interactions to the simplified set of four HTTP operations acting upon a resource. This challenge can lead to URI designs that become increasingly complex, resulting in a steeper learning curve for developers.
Another challenge is that this object-based interaction style may require applications to make multiple requests in cases where many collections of data resources are required to achieve a task. In other words, a URI-style API may encourage “chattiness”, which can be detrimental to perceived application responsiveness for end users.

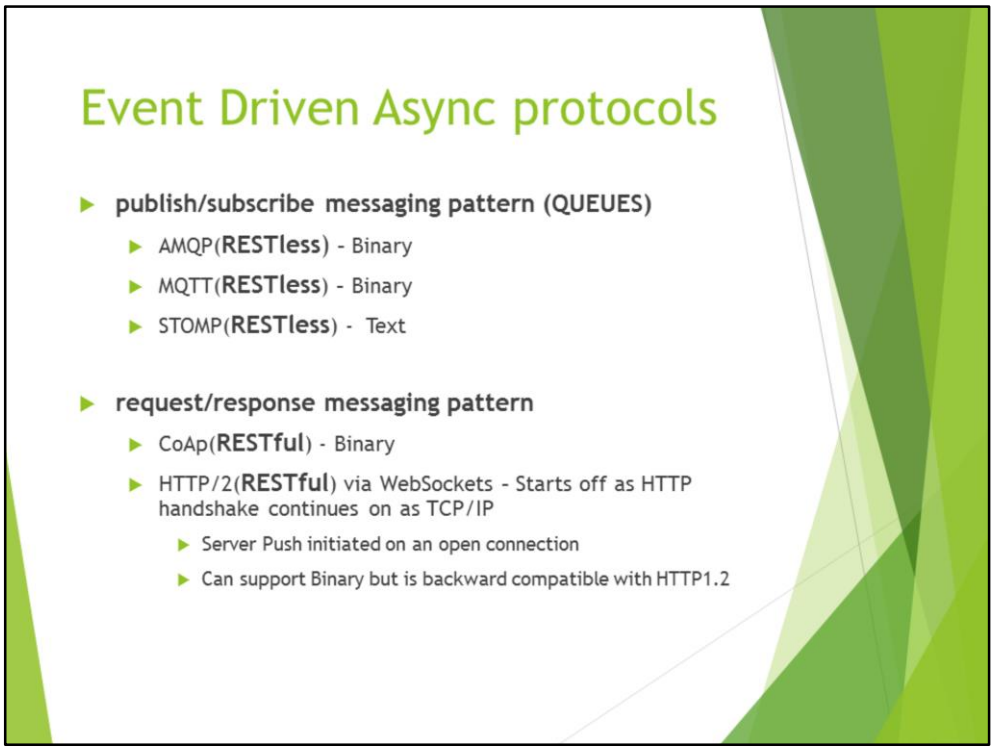
Hypermedia Style
This is similar to the URI style but it utilizes hypermedia to create interactions focused on tasks rather than on objects. Essentially, these are browser-based interaction for machines. In much the same way that you use links to navigate the Web and forms to provide input, a Hypermedia API provides links to navigate a workflow and template input to request information.
A Hypermedia API:
Exposes a task-based interface, often incorporating a workflow or state machine
Uses media to describe link semantics, template-based input and message structures
Provides its own URIs
A key benefit of this style is that it favors long-running services. When designed correctly, a Hypermedia API can evolve over many years and continue to support applications that have been developed during its infancy. However, there is a lack of mature tooling and support for this type of API. Consequently, some developers see hypermedia APIs as excessively complex.

Event-Driven Style
API interactions based on event-driven architectures have gained in popularity recently. A popular example of the event-driven style is the WebSocket protocol standard that has been incorporated into the HTML 5 specification. WebSockets provide a useful way of transmitting data with low overhead, in both directions between a client and server.
A Web API designed using the Event-Driven style will typically exhibit both of the following key properties:
The client and/or the server listen for new events
Events are transmitted via asynchronous messages, as they occur
While some Event-Based APIs utilize the HTTP protocol, there is a growing body of

network-based protocols like WebSocket that favour low overhead, asynchronous communication. This style of API can be very effective for applications that need to frequently update UI widgets or for bi-directional, message-intensive applications such as multiplayer video games.

However, WebSocket APIs require additional infrastructure considerations beyond the traditional HTTP-based API components and are not well-suited to request-reply interactions. Designers should also be conscious that a WebSocket conversation is only initiated in HTTP – once the handshake is complete, the protocol shifts to a TCP/IP WebSocket-based protocol.

While this style is primarily associated with mobile apps, API designers who need to support the multitude of network-connected devices within the Internet of Things (IoT) also have an interest in interactions that reduce overhead and facilitate real-time, event-based communication.

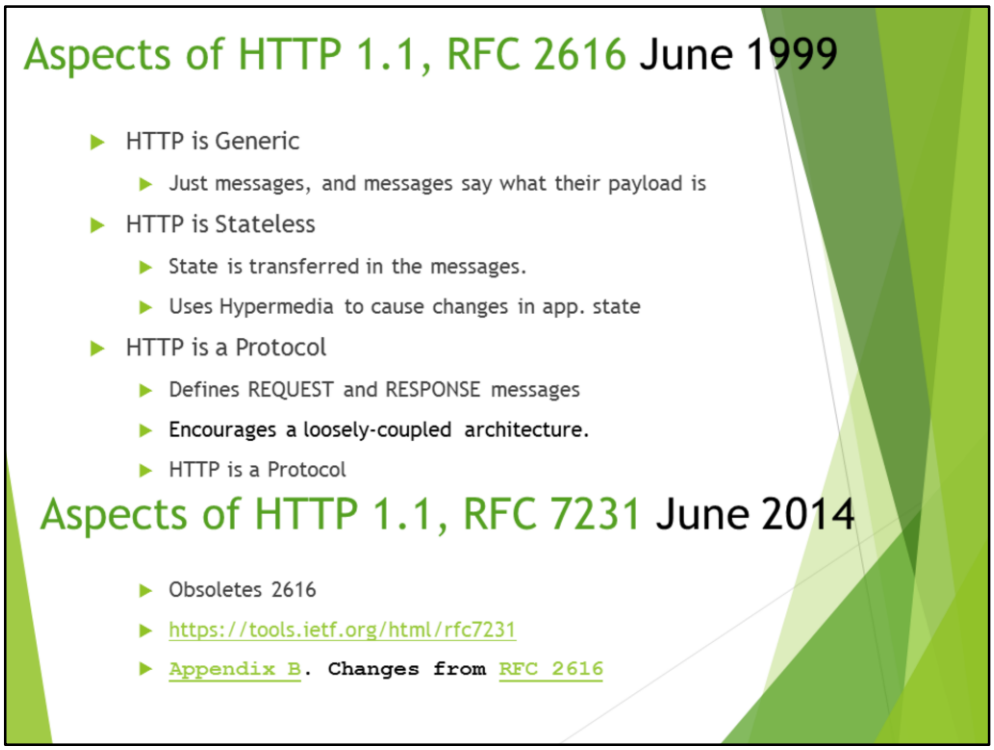


State is implicit in RESTless messages:
The REST architecture makes no such assumptions. There is no such notion as a long-term connection or conversation.

Event Driven Async protocols

Protocol	Sponsor	Blessing	Message Pattern	QoS	Security	"Addressing"	REST?	Constrained Devices?
MQTT	MQTT.org	OASIS	P/S	3 levels**	Best practices	Topic only		Y
CoAp	IETF	IETF	R/R	Optional	DTLS	URL	x	Y
XMPP	XMPP Standards Foundation	IETF	P-P P/S by extension	None (could be done by extension)	TLS/SSL XEP-0198	JID		I think so
AMQP	OASIS	OASIS	P/S	Sophisticated	TLS; SASL	Y		N
DDS	OMG	OMG	P/S	Sophisticated	In beta	Topic/key		Y (no optional features)
SMQ	Real Time Logic	Proprietary (might be opened)	P/S	Limited; mostly via TCP	SSL	Y		Y
HTTP/2	IETF	IETF	R/R	TCP	TLS/SSL	URL	x	Y (HPACK)
AllJoyn	Qualcomm	AllSeen Alliance	RPC	No	Done by app	Y		Thin client option
STOMP	Community		P/S	Simple Server-specific	N	N Server-specific		N

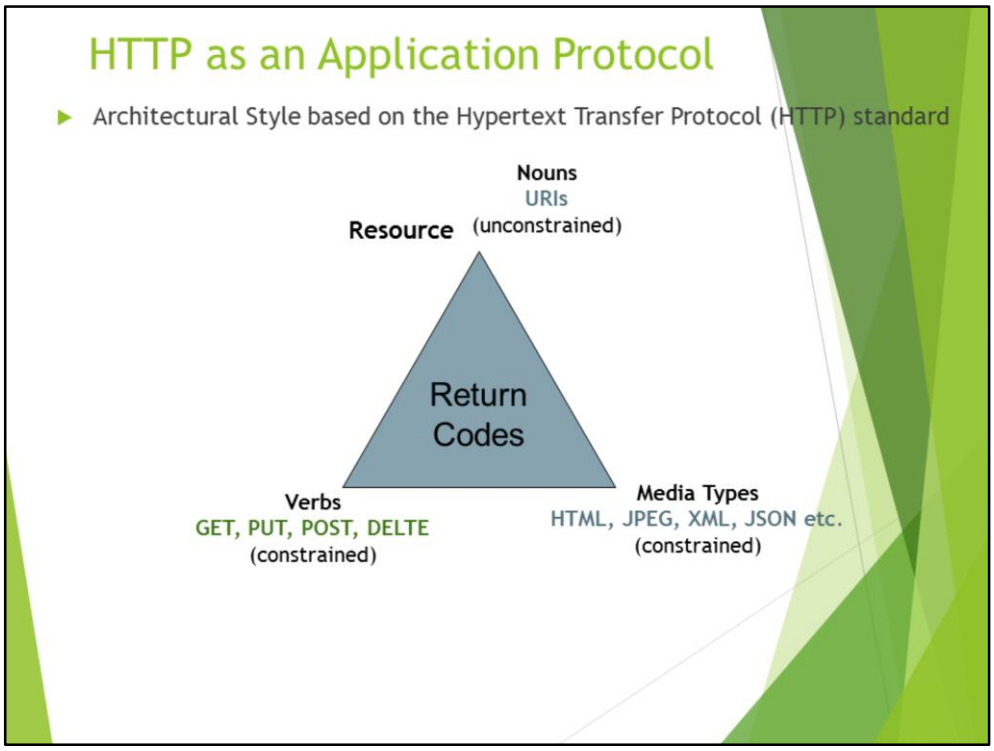
P/S=Publish/Subscribe
R/R=Request/Respond
P-P=Point-to-Point or Peer-to-Peer
RPC=Remote Procedure Call



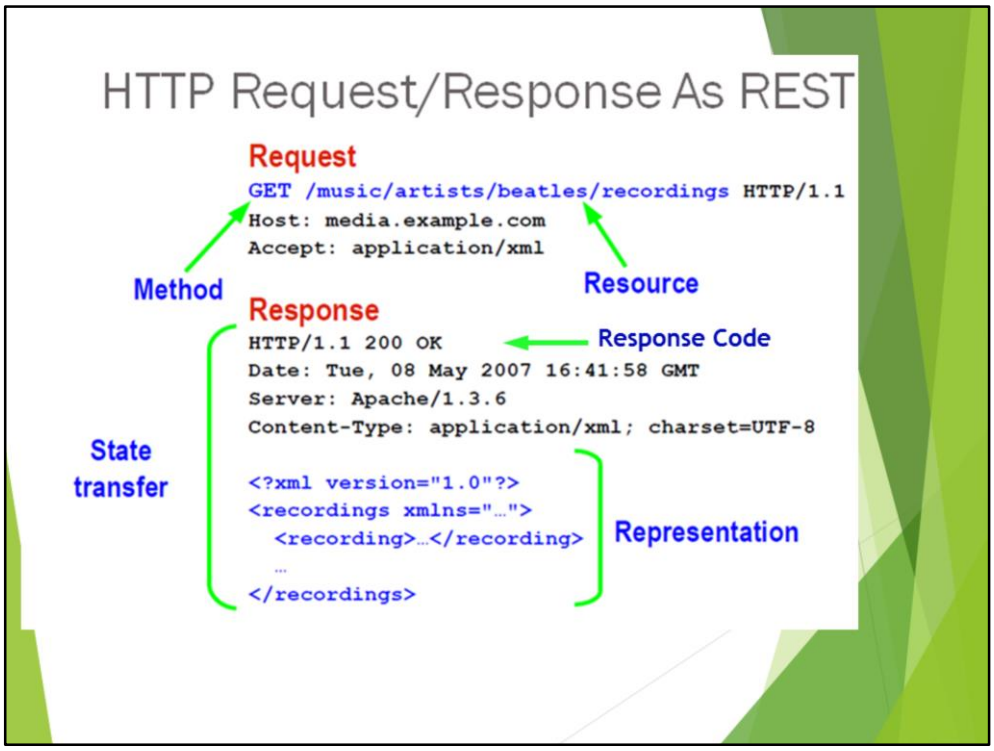
HTTP is simple. It is a message-passing protocol that defines just two basic types of messages - Request Messages and Response Messages. The messages contain headers that can define what the payload of that message is, and give the client or the server hints on how to handle that payload. With such an open, generic, architecture, HTTP is well suited to many types of application development and encourages a loosely-coupled architecture.

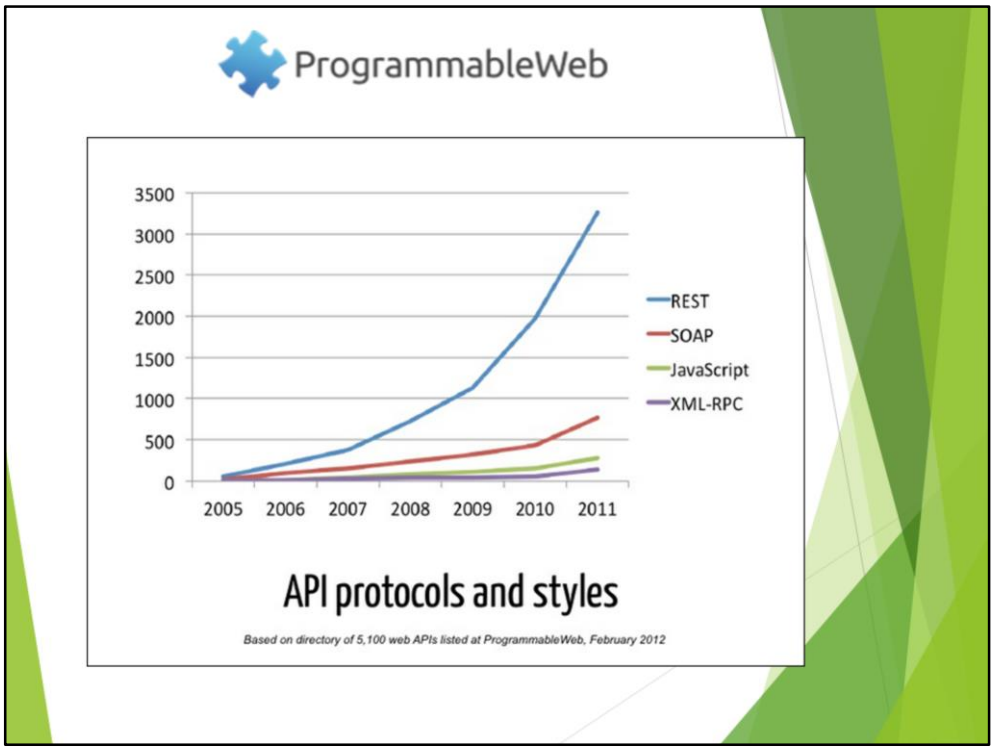
Lack of state == scalability. HTTP does not “hold” application state. State is transferred in the messages. Hypermedia provides application state transitions.
For example, we might be looking at a hypertext document (HTML) that shows the current state of the inventory in a book shop. This document may contain links to each item in inventory. By generating a request by clicking on a link, we are transitioning the state of our application to the state of showing details of that particular item.
In HTTP, we use hypermedia to cause changes in application state.

> This concept, known by the totally awesome acronym of HATEOS - Hypermedia As the Engine of Application State - is one of the fundamental concepts developers must understand to fully utilize HTTP as an application protocol.



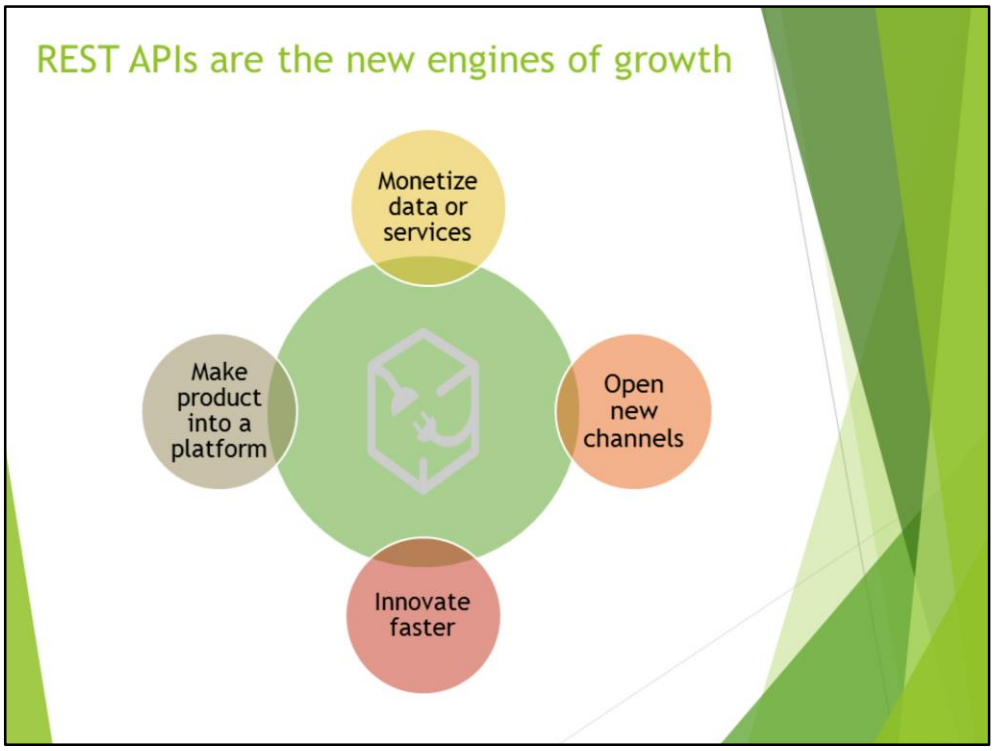
Resources to identify what you are communicating with
Standard verbs to denote what you want to do with that resource
GET, PUT, POST, DELETE, PATCH, HEAD and OPTIONS
Standard status codes to tell you how it went
Returns data in various formats (Media Types)
JSON, XML, HTML, PDF, JPG, etc.
API is a logical grouping of Resources (Like a Menu)


HTTP Request/Response As REST
Request
GET /music/artists/beatles/recordings HTTP/1.1
Host: media.example.com
Accept: application/xml
Method
Resource
Response
HTTP/1.1 200 OK
Response Code
Date: Tue, 08 May 2007 16:41:58 GMT
Server: Apache/1.3.6
Content-Type: application/xml; charset=UTF-8
State transfer
<?xml version="1.0"?>
<recordings xmlns="…">
<recording>…</recording>
…
</recordings>
Representation

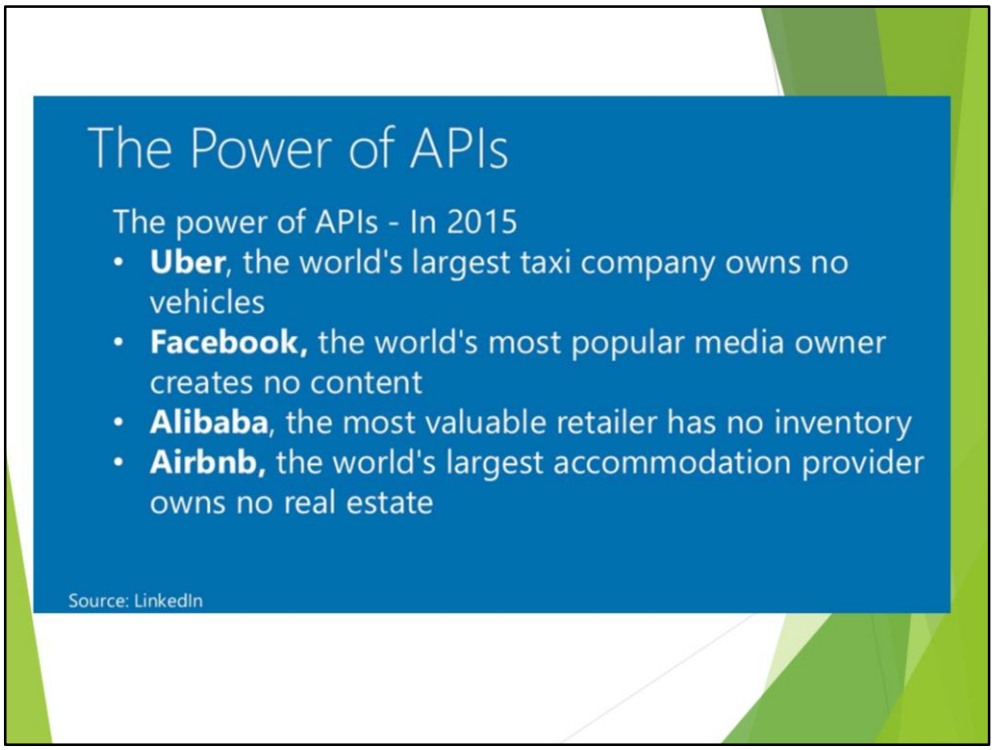


SOAP is not Dead but it is aging.
http://www.programmableweb.com

0-5000 = 7 years
5000 to 15,500 = in 4 years it tripled



Well established industries are being disrupted.


The Power of APIs
The power of APIs - In 2015
• Uber, the world's largest taxi company owns no vehicles
• Facebook, the world's most popular media owner creates no content
• Alibaba, the most valuable retailer has no inventory
• Airbnb, the world's largest accommodation provider owns no real estate
Source: LinkedIn

Superhero of APIs →

- The term "**Re**presentational **S**tate **T**ransfer" was coined by Roy T. Fielding in his Ph.D. thesis, published in 2000: "*Architectural Styles and the Design of Network based Software Architectures*"

CHAPTER 5

Representational State Transfer (REST)

- "Deriving REST can be described by an architectural style consisting of the set of constraints applied to elements within the architecture."
- *These constraints leverage the HTTP standard.*

What is REST?

- REST stands for **Re**presentational **S**tate **T**ransfer.
- Architectural style described by 6 constraints:

1	***Uniform Interface***	simplified architecture
2	***Stateless***	session state is held in the client • Each request must contain all the information the server needs to correlate the work
3	***Cacheable***	clients can cache responses to improve performance
4	***Client-server***	independent implementation of servers and clients
5	***Layered System***	servers are obfuscated from clients • No Layer can "see past" the next layer • Ex: API Gateway
6	***Code on Demand****	customizable functionality *The only optional constraint* Allows Clients to be updated by the server

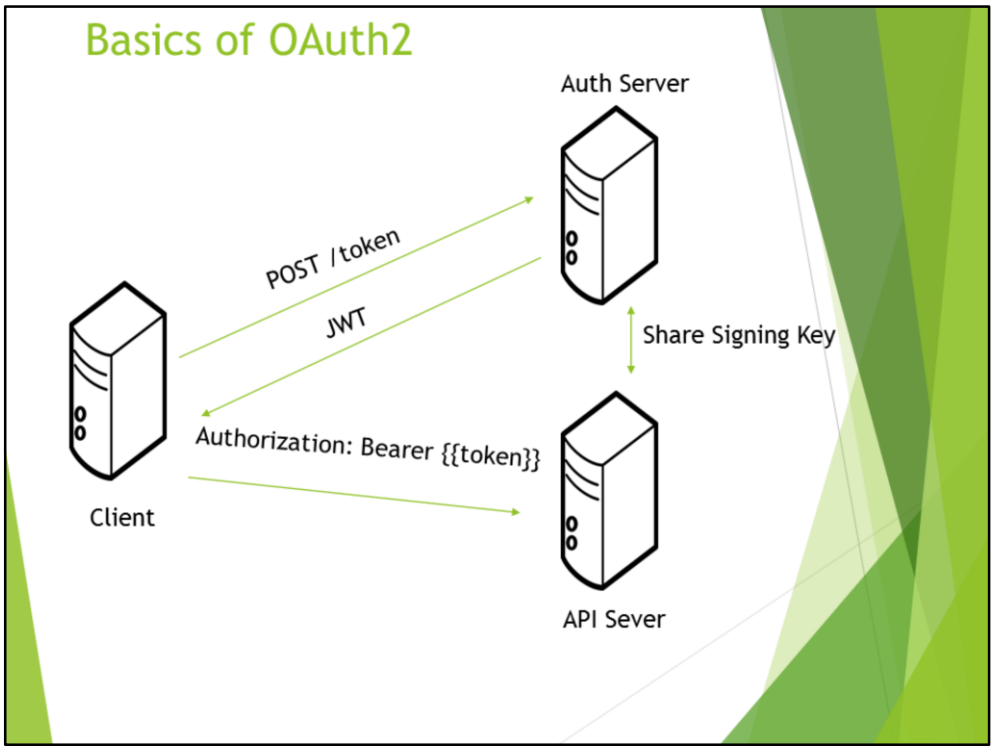
REST over HTTP – Uniform interface

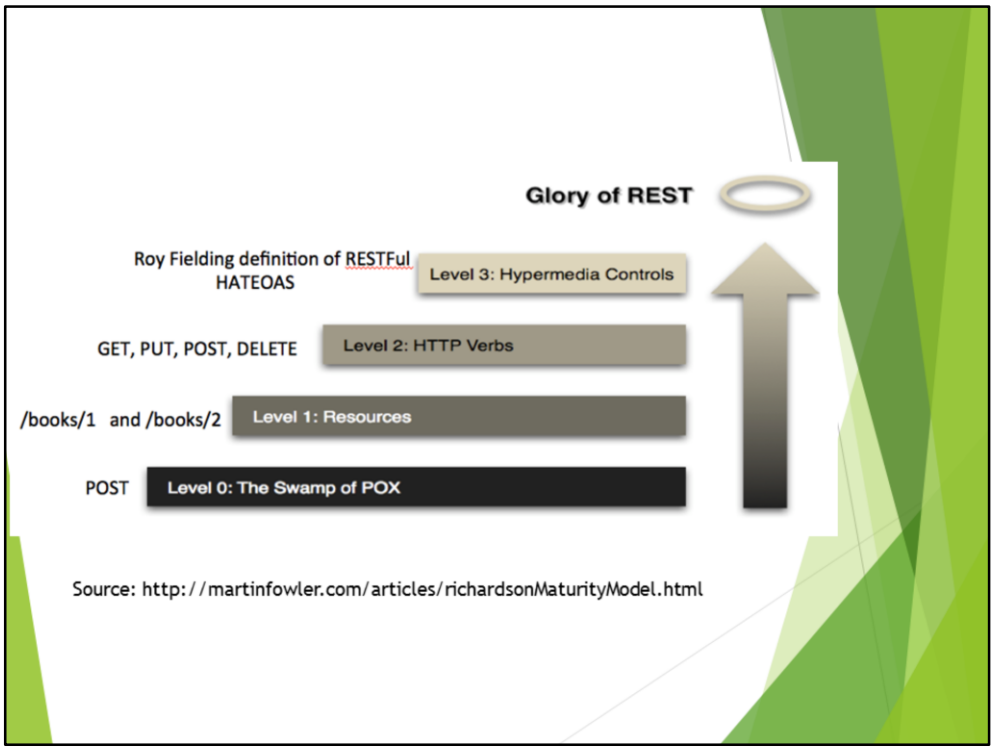
- CRUD operations on addressable resources
  - Create, Read, Update, Delete
- Performed through HTTP methods + URI
- URI standard format: scheme://host:port/path?queryString

CRUD Operation	HTTP Method	Description
Create	POST	Create a new resource
Read	GET	Get a particular resource
Update	PUT	Update an existing resource
Delete	DELETE	Delete a particular resource
Delta Update	PATCH	Update a data change of an existing resource

Code on Demand example

- Some well-known examples for the code on demand paradigm on the web are:
  - Java applets
  - Adobe's ActionScript language for the Flash player
  - JavaScript
- Delegated Authorization
- OAuth2
  - No credentials are passed across the network to the resource server


Basics of OAuth2
Auth Server
POST /token
JWT
Share Signing Key
Authorization: Bearer {{token}}
Client
API Sever

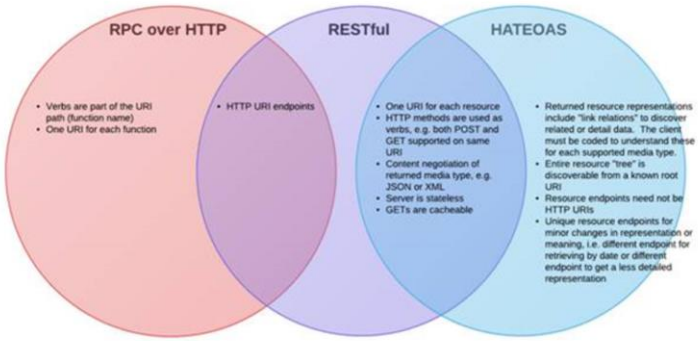


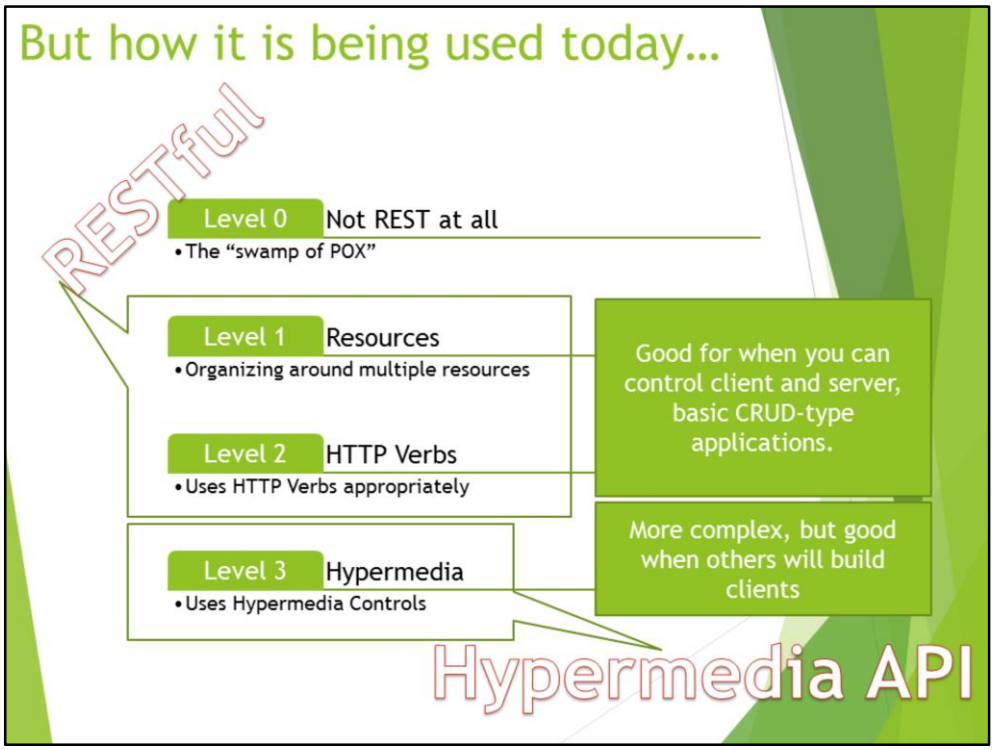
A model (developed by Leonard Richardson) that breaks down the principal elements of a REST approach into three steps. These introduce resources, http verbs, and hypermedia controls.

HATEOAS:
Response include hypermedia links to navigate the restful interface
Each link is presumed to support REST verbs of GET, POST, PUT, DELETE

Level 0: The Swamp of POX – At this level you are dealing with mostly plain old XML, a single endpoint and using just the POST method, most likely with SOAP
Level 1: Resources – At this level you have defined resources such as book/1 and book/2 but still using a single method, again most likely POST
Level 2: HTTP Verbs – At this level you use the HTTP verbs GET,PUT, POST,DELETE properly and use the appropriate response codes.
Level 3: Hypermedia Controls: At this level you are utilizing HATEOAS (Hypermedia as the Engine of Application State). You have reached Roy Fielding definition of RESTFul APIs and the "Glory of REST".

Richardson Maturity Model Spectrum





XML and JSON are data formats, they have no understanding of hyperlinking.

Hypermedia (Self describing APIs)

POST

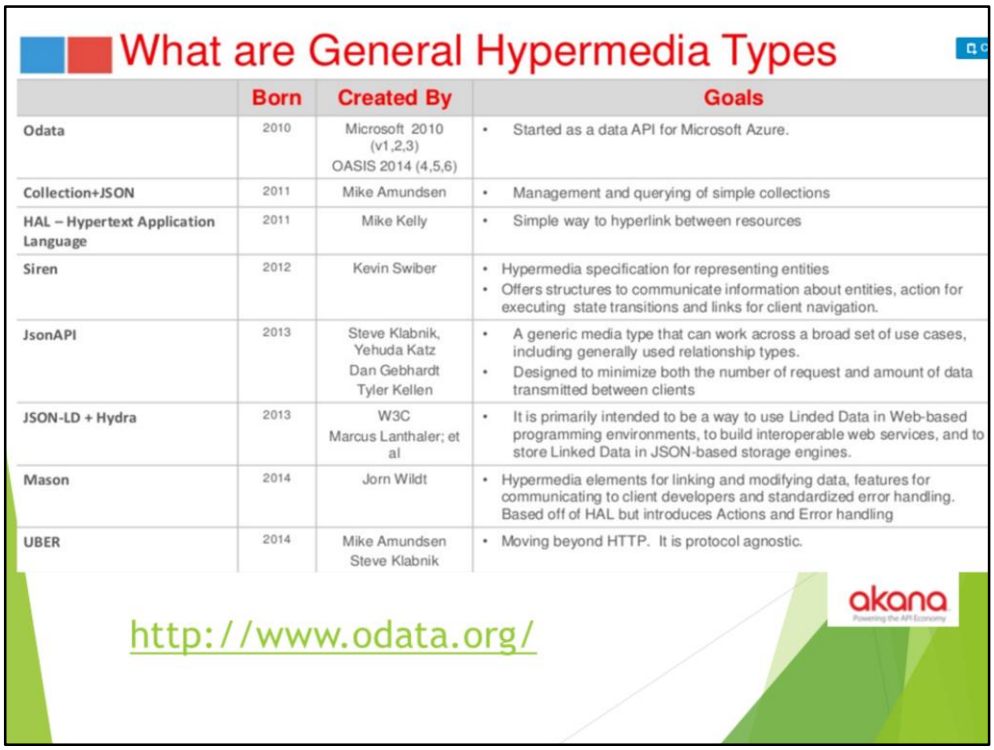
Response

```
HTTP/1.1 201 Created
Content-Length: 652
Content-Type: application/json;odata.metadata=minimal;odata.streaming=true;IEEE754Compatible=false;charset=utf-8
ETag: W/'08D1D3800FC572E3'
Location: http://services.odata.org/V4/(S(34wtn2c0hkuk5ekg0pjr513b))/TripPinServiceRW/People('lewisblack')
OData-Version: 4.0
```

GET

```
GET http://services.odata.org/v4/TripPinServiceRW/People('russellwhyte') HTTP/1.1
OData-Version: 4.0
OData-MaxVersion: 4.0
```

```
HTTP/1.1 200 OK
Content-Length: 482
Content-Type: application/json; odata.metadata=minimal
ETag: W/'08D1D5BE987DE78B'
OData-Version: 4.0
{
    "@odata.context":"http://services.odata.org/V4/(S(fo3by51qwd1q124ngi2rxrzc))/TripPinServiceRW/$metadata#People/$entity",
    "@odata.id":"http://services.odata.org/V4/(S(fo3by51qwd1q124ngi2rxrzc))/TripPinServiceRW/People('russellwhyte')",
    "@odata.etag":"W/\"08D285C0E2748213\"",
    "@odata.editLink":"http://services.odata.org/V4/(S(fo3by51qwd1q124ngi2rxrzc))/TripPinServiceRW/People('russellwhyte')",
```



Arguments for Hypermedia APIs
API are more flexible
Won't break the client
Enables discoverability
Replaces the Need for Documentation
Is true Roy Fielding REST

“A RESTafarian is a zealous proponent of the REST software architectural style as defined by Roy T. Fielding in Chapter 5 of his PhD. dissertation at UC Irvine. You can find RESTafarians in the wild but be careful, RESTafarians can be extremely meticulous when discussing the finer points of REST …”

every time you start talking about anything to do with APIs over HTTP, this happens:

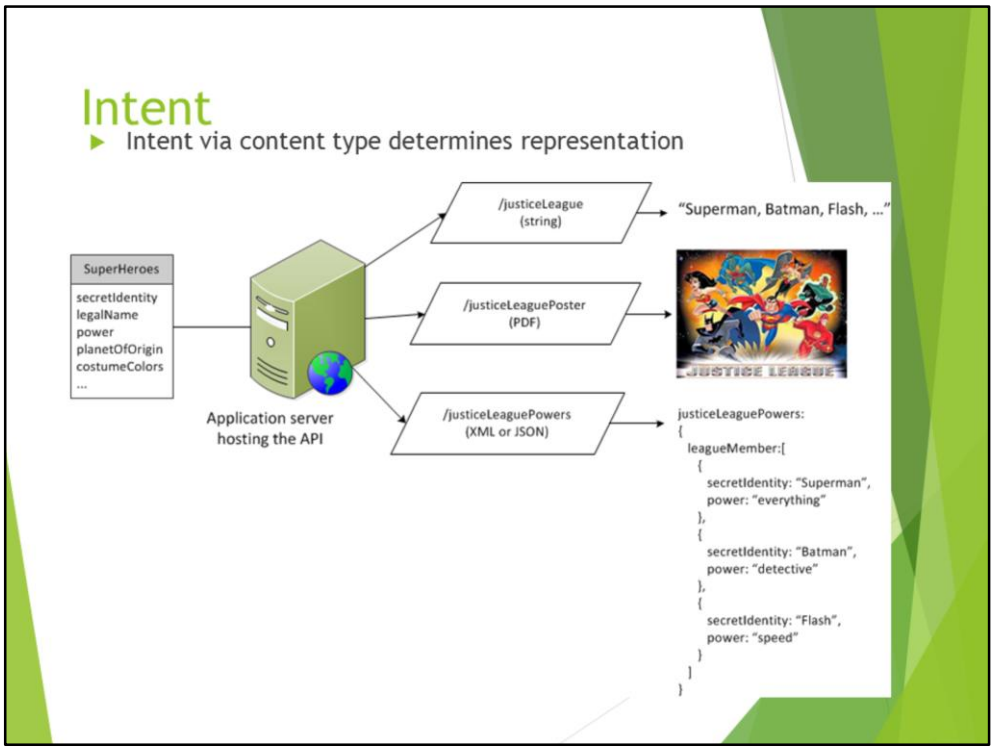
RESTful Insights

- REST is a style rather than a standard and it is surprisingly hard to implement consistently across an organization.
- Developing a REST API involves numerous subtle design decisions around areas such as resource design, method usage and status codes.
- It's easy to get hung up on debates around what is and what is not "RESTful".
  - Microsoft REST API Guidelines 2.3
  - Microsoft REST API Guidelines Are Not RESTful
- There are API design Anti-Patterns.

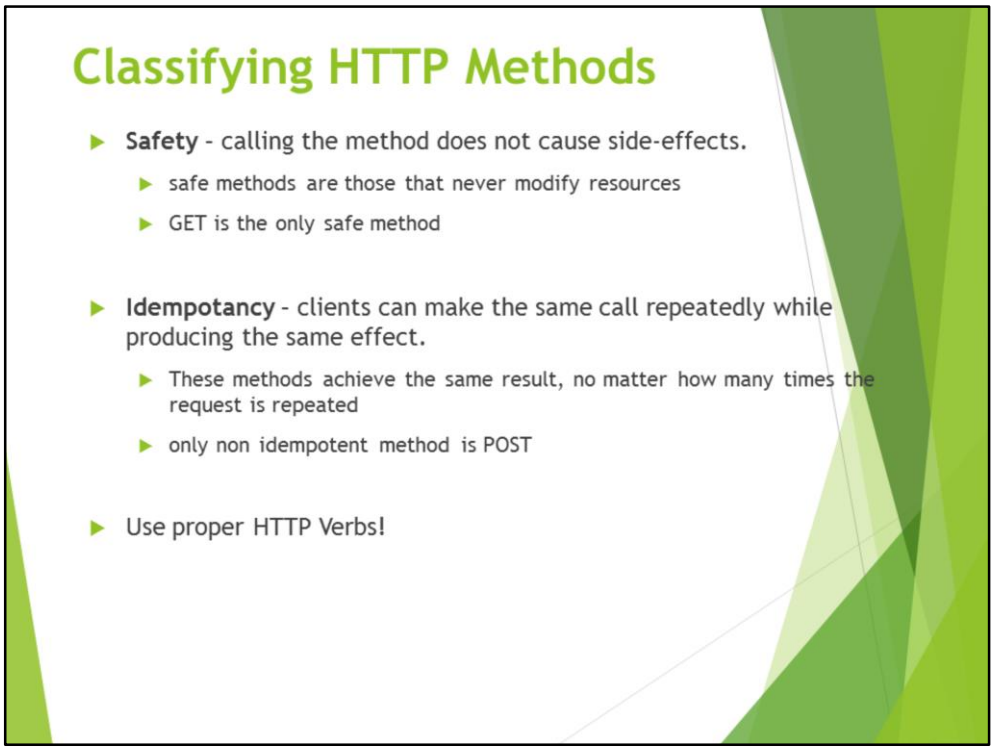
Rock the Mullet API: RESTful in the front, WSDL in the back

Naming Resources

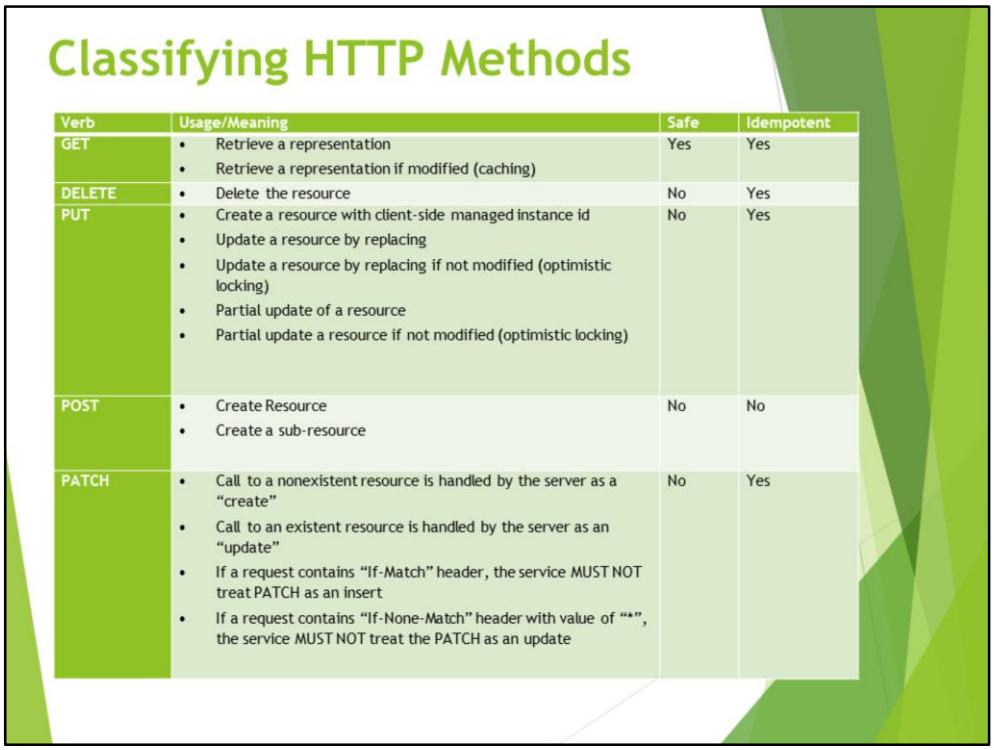
- Use **NOUNS**! Not VERBS (as Http already provides them)
- Meaningful names provides context for consumers
- Simple naming should not hide complexity behind query strings. → Apply *Uniform Interface constraint*
- Resources are hierarchical like in folder structures
  - /parent/child/{id}
- How the industry does it:
  - Foursquare: /*checkins*
  - Groupon: /*deals*
  - Zappos: /*product*



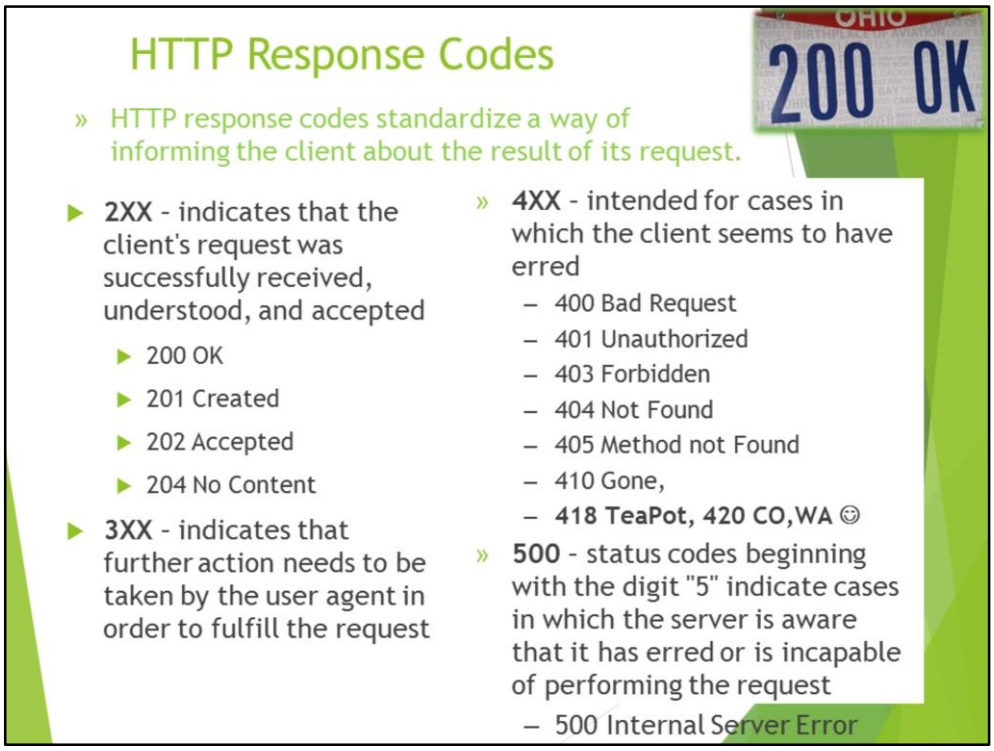
superHero {
superIdentity (string, *optional*),
secretIdentity (string, *optional*),
planetOfOrigin (string, *optional*),
weakness (string, *optional*),
superPowers (array[string], *optional*)
}



You must be careful to apply the HTTP protocol correctly and enforce these semantics yourself.



Verbs have unintended Side Effects if used improperly.



418 was defined in 1998 as one of the traditional IETF April Fools' jokes, in RFC 2324, Hyper Text Coffee Pot Control Protocol, and is not expected to be implemented by actual HTTP servers. The RFC specifies this code should be returned by tea pots requested to brew coffee.

HTTP Response Codes Map

Courtesy of Restlet

HTTP Headers

- They define the operating parameters of the transaction, the so called metadata. In REST style, more important then the body.
- Request Headers

Header	Type	Description
Authorization	String	Authorization header for the request
Date	RFC 1123 date	Time-stamp of the request
Accept	Content type	The request content type for the response, such as: • application/xml • text/xml • application/json • text/javascript

- Response Headers

Header	Description
Date	The date the response was processes, in RFC 1123 format
Content-Type	The content type
ETag	The ETag response-header field provides the current value of the entity tag for the requested variant. Used with If-Match, If-None-Match and If-Range to implement optimistic concurrency control.

Where to pass API parameters?

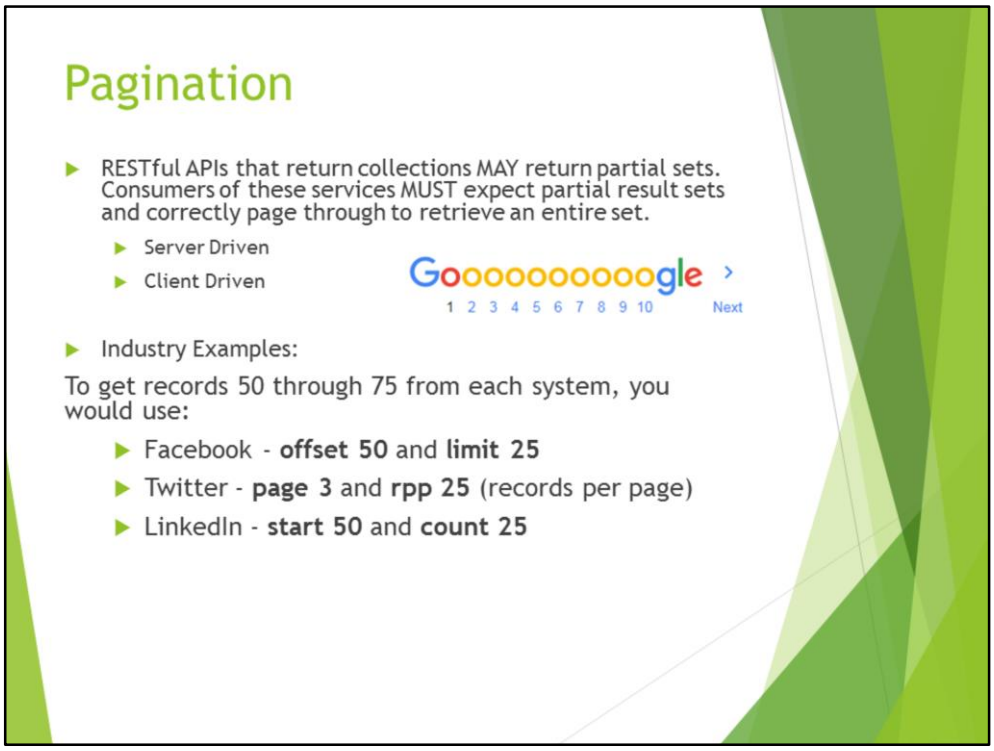
» There are four common places to put information in an HTTP request:

Where	What	Example
URL/Path (if required)	The URL should follow the REST architectural style. It should contain the names of resources and resource IDs only.	v2/policyservicing/policies/{policyNumber}
Headers (Global)	The headers are best used for authentication and specifying the format of your data.	Authorization: Bearer Verbose: Off
Query Parameters (optional)	Query parameters should be used to adjust what kind of data to return – for example, pagination or parameters that filter what data is returned.	/slot10/v1/csc/?loc=2&city=Cleveland&state=OH

POST/PUT body (resource specific)	All data that is used to create or modify resources goes in the body of the request.	{
		"agreementId": " 12345",
		"comment": "This is the optional comment field"
		}

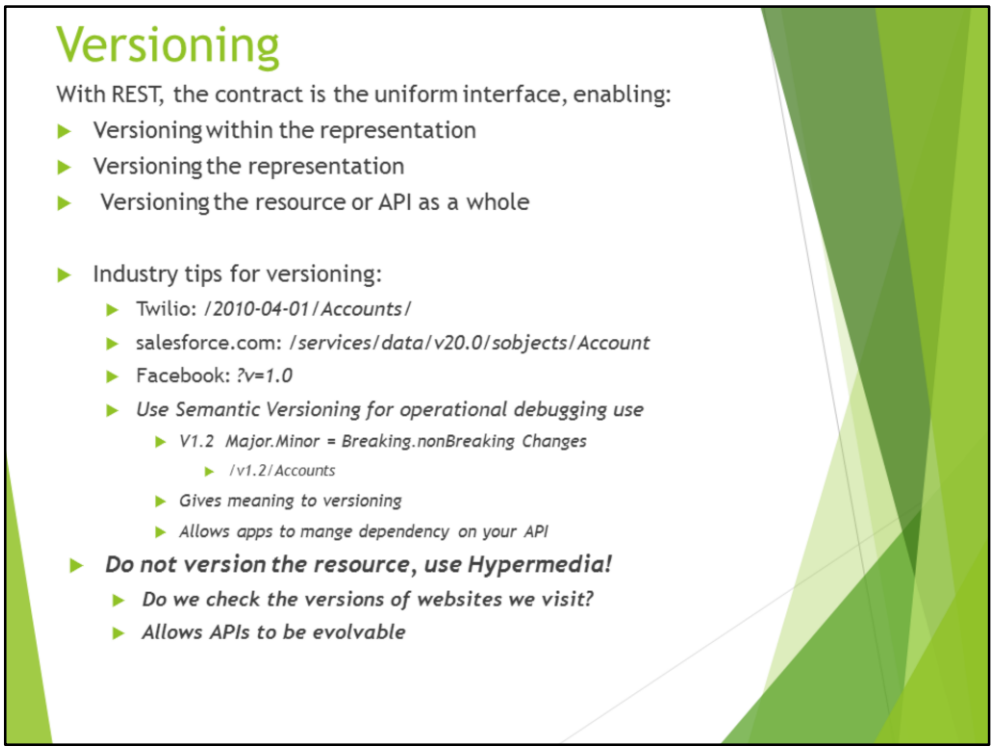
Query String Manipulation

- All operations that can not be performed via the verbs on a resource, can be done in a query string.
- Sweep complexity behind the ? (use only when needed)
- **Sorting** - use an optional parameter *"sort"* with a comma-separated list of fields.
  - */customers?sort=state,county*
- **Searching** - simple search with fields names
  - */claims?status=closed*
  - For more complex queries, a RESTful alias (aka common name) can be used: */claims/monthlyReport*



Server Driven - consumers of these services MUST expect partial result sets and correctly page through to retrieve an entire set.

Client Driven - Clients MAY use *$top* and *$skip* query parameters to specify a number of results to return and an offset.



Versioning within the representation

Optional elements in the response

Versioning the representation

Content negotiation determines representation

Versioning the resource or API as a whole

The most similar approach to the web services used in the past

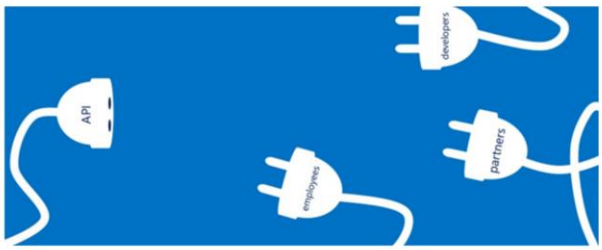
Review

- REST is a new architectural style for web services
- Resources are logical entities that have a have an intent driven representation.
  - Nouns
  - JSON
- Use HTTP protocol to communicate
  - Verbs: GET, PUT, POST, DELETE, PATCH
  - Status Codes: 200, 404, 500, etc.
- Headers
  - Request/Response
- Query string manipulation
  - Sorting/Searching/Pagination
- Versioning - clients should be able to count on services to be stable over time(but evolvable from the provider side)
  Future Proof the APIs.

What Exactly Is an API?

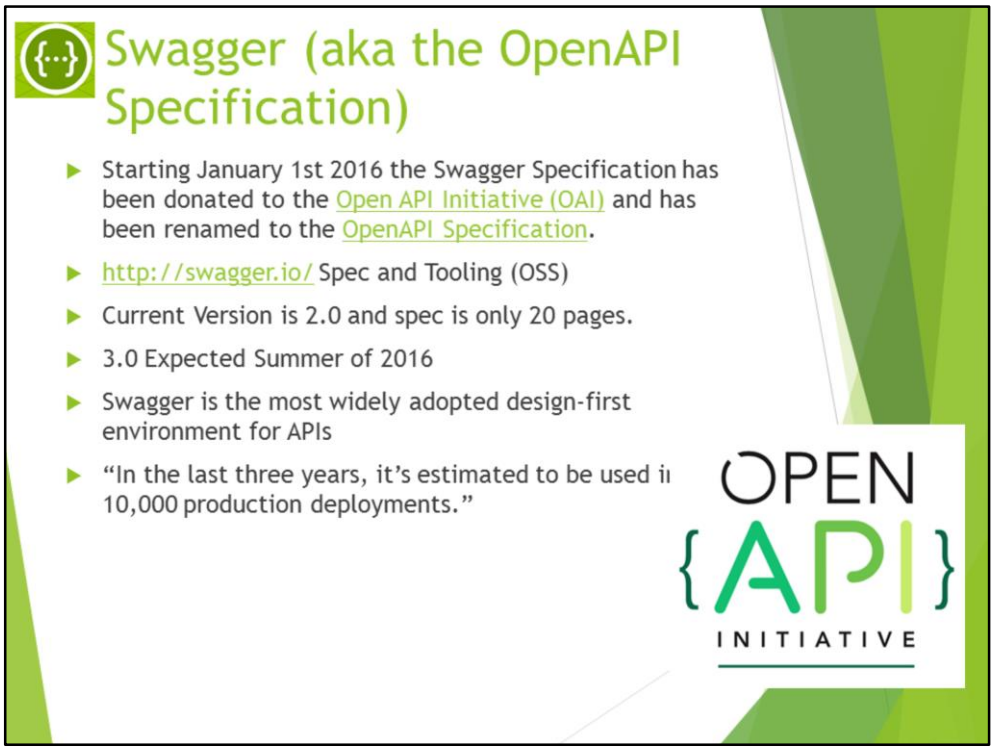
- In the simplest terms, an application programming interface, or API, is a set of requirements that enables one application to talk to another application.
- API is a logical grouping of Resources (Like a Menu)

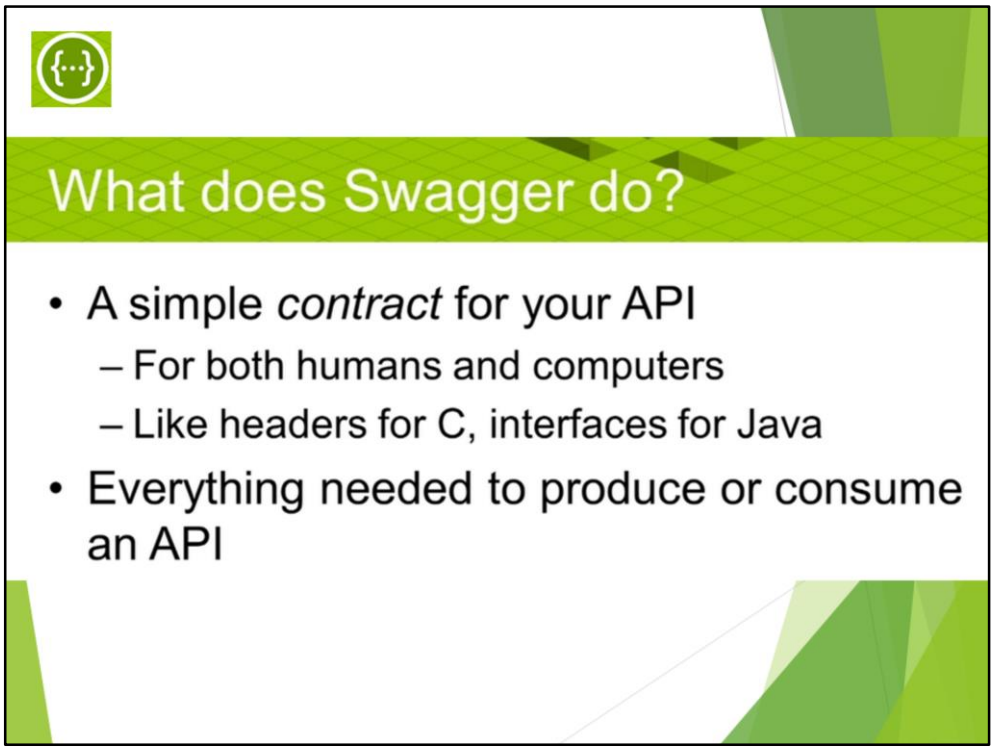
- Another way of thinking about APIs is in the context of a wall socket.

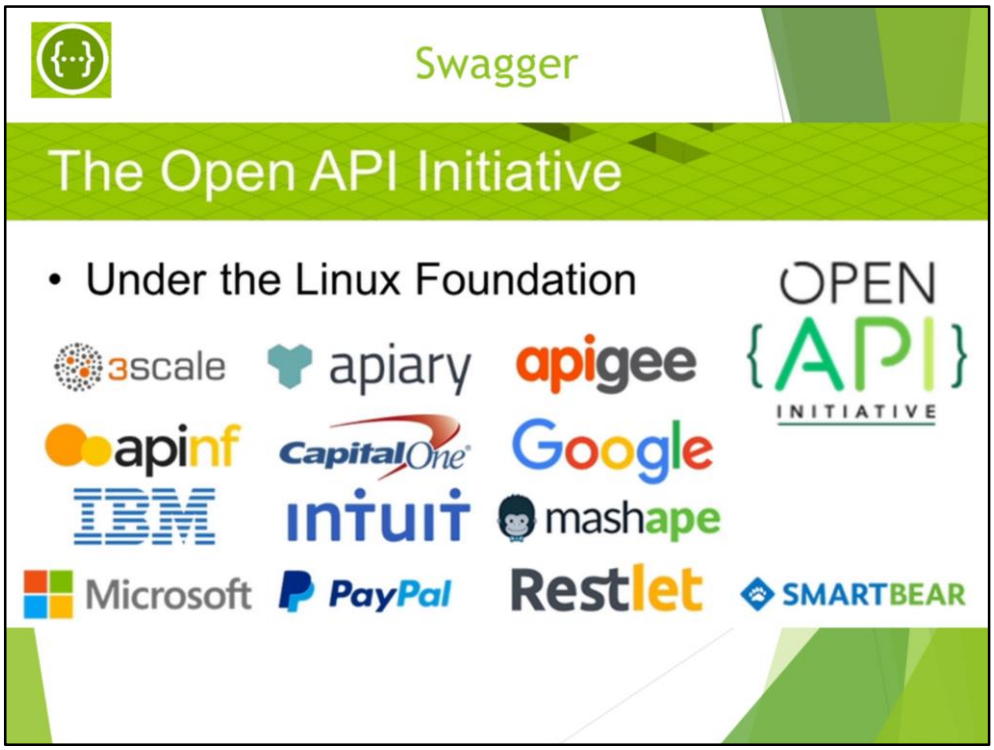


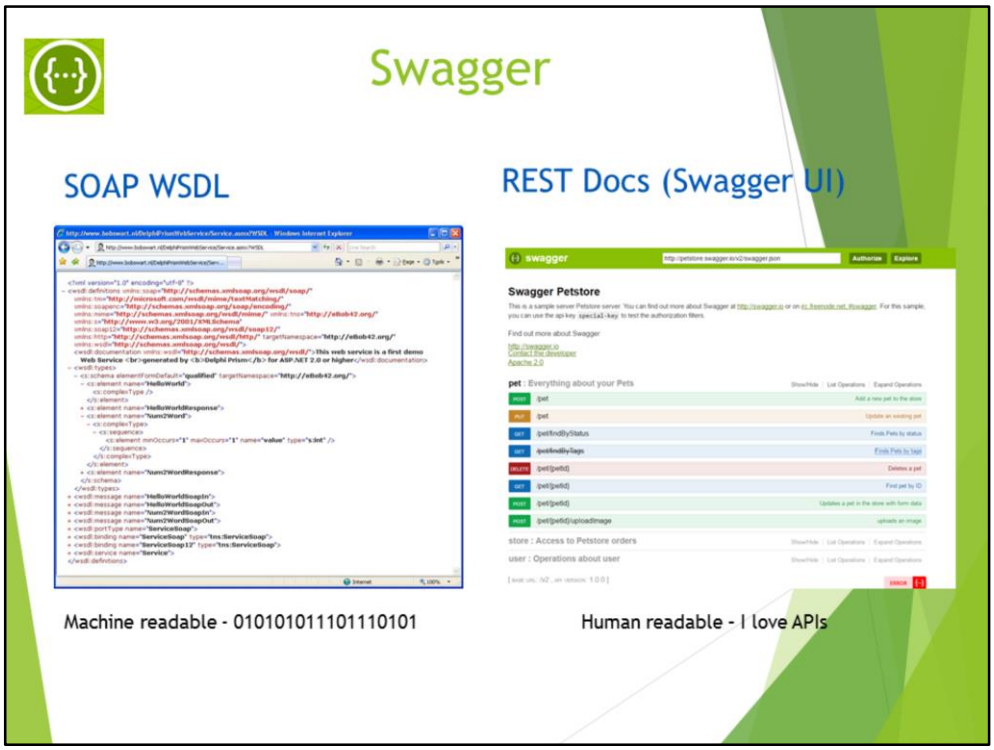
What is a Web API?

- Similar in nature but have a prescribed implementation:
  - HTTP(s)
  - RESTful
  - JSON (preferred)
  - XML (supported)
  - Spec Driven (Swagger, RAML, API Blueprint)

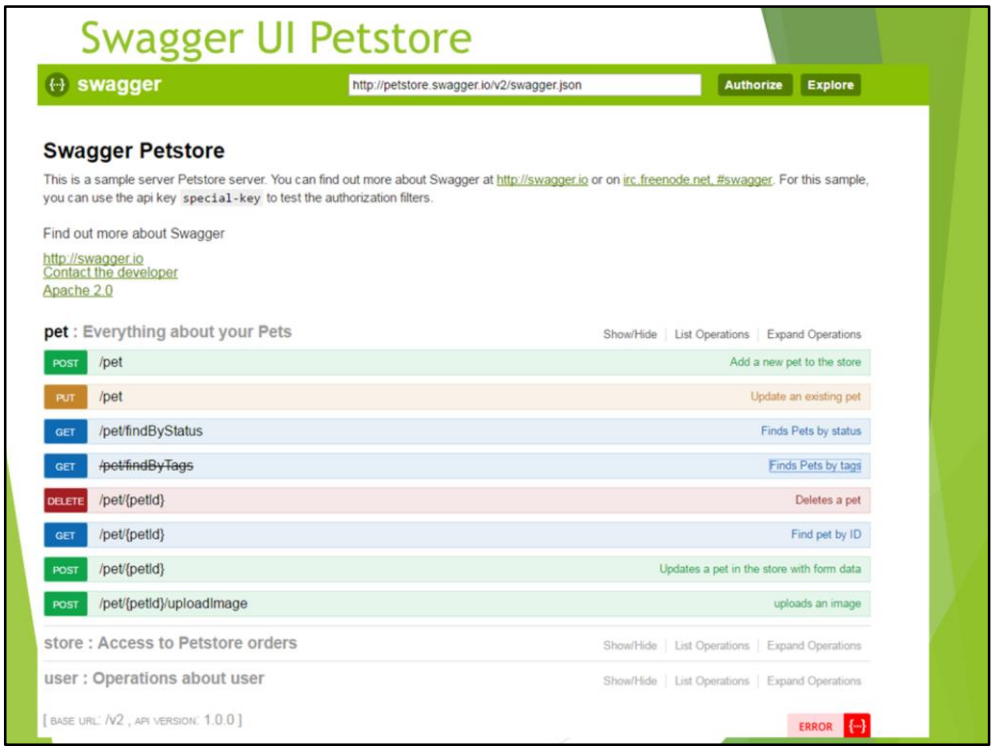

Swagger (aka the OpenAPI Specification)
Starting January 1st 2016 the Swagger Specification has been donated to the Open API Initiative (OAI) and has been renamed to the OpenAPI Specification.
http://swagger.io/ Spec and Tooling (OSS)
Current Version is 2.0 and spec is only 20 pages.
3.0 Expected Summer of 2016
Swagger is the most widely adopted design-first environment for APIs
"In the last three years, it's estimated to be used i
10,000 production deployments."
OPEN
{API}
INITIATIVE


What does Swagger do?
• A simple *contract* for your API
– For both humans and computers
– Like headers for C, interfaces for Java
• Everything needed to produce or consume an API

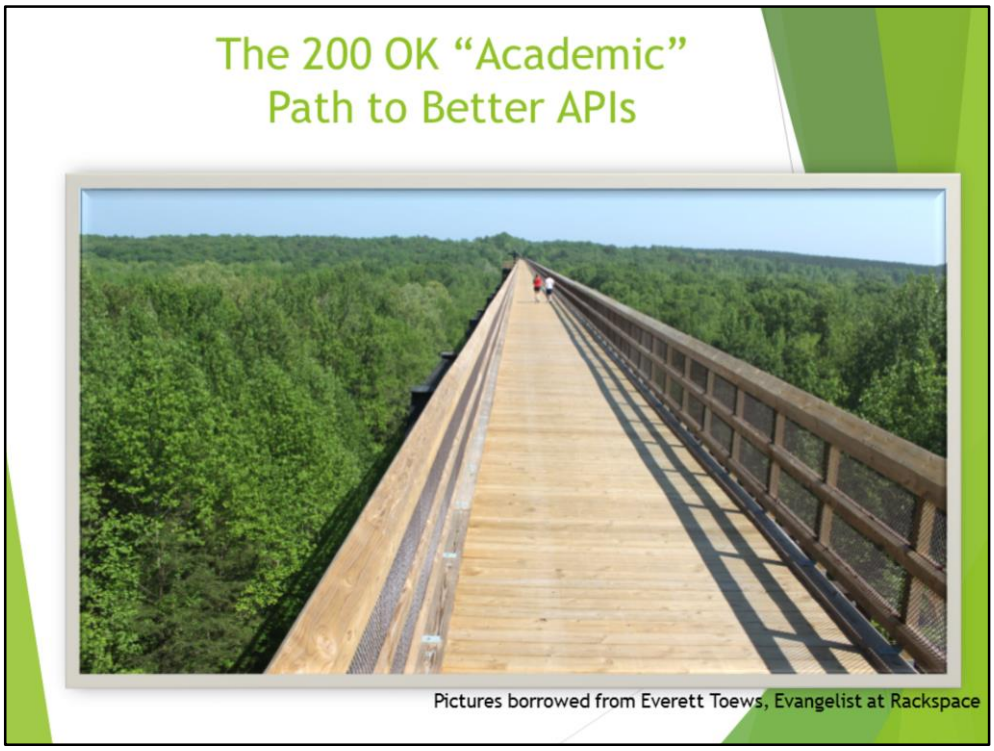

Swagger
The Open API Initiative
• Under the Linux Foundation
3scale
apiary
apigee
OPEN
{API}
INITIATIVE
apinf
CapitalOne
Google
IBM
intuit
mashape
Microsoft
PayPal
Restlet
SMARTBEAR



Simple and Human readable


Swagger UI Petstore
swagger
http://petstore.swagger.io/v2/swagger.json
Authorize
Explore
Swagger Petstore
This is a sample server Petstore server. You can find out more about Swagger at http://swagger.io or on irc.freenode.net, #swagger. For this sample, you can use the api key special-key to test the authorization filters.
Find out more about Swagger
http://swagger.io
Contact the developer
Apache 2.0
pet : Everything about your Pets
Show/Hide | List Operations | Expand Operations
POST /pet Add a new pet to the store
PUT /pet Update an existing pet
GET /pet/findByStatus Finds Pets by status
GET /pet/findByTags Finds Pets by tags
DELETE /pet/{petId} Deletes a pet
GET /pet/{petId} Find pet by ID
POST /pet/{petId} Updates a pet in the store with form data
POST /pet/{petId}/uploadImage uploads an image
store : Access to Petstore orders
Show/Hide | List Operations | Expand Operations
user : Operations about user
Show/Hide | List Operations | Expand Operations
[BASE URL: /v2 , API VERSION: 1.0.0]
ERROR

Show how Postman can import Swagger File



How to tie all the info together. The bigger perspective and how this will help with designing APIs.

API Personas

API Consumers
Use API's from Portal

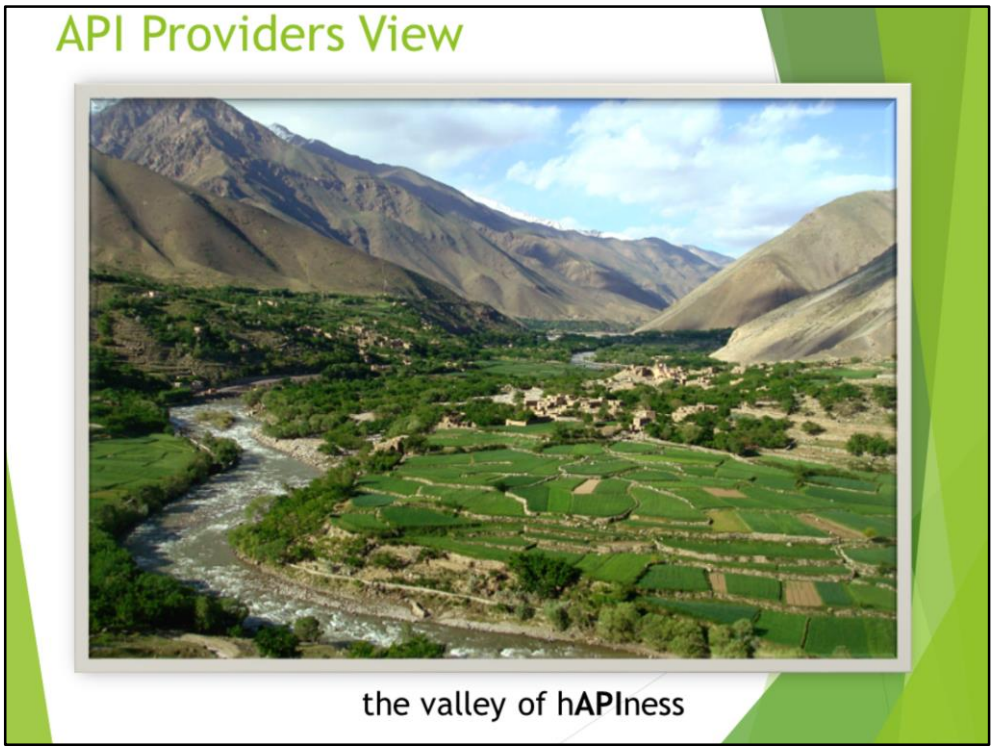
- Where do I access APIs?
- What Apps do I register ?
- What is available today?
- How do I understand APIs?
- How do I request access?

API Designers
Create API Docs

- How can I create APIs consumers love?
- How can I rapidly release & update my APIs?
- How do I evolve my API?
- How do I measure success?

API Providers
Publish API's to Portal

- How do I on board APIs?
- How do I assemble APIs?
- How do I manage security?
- Will the infrastructure scale?
- How do I measure performance?



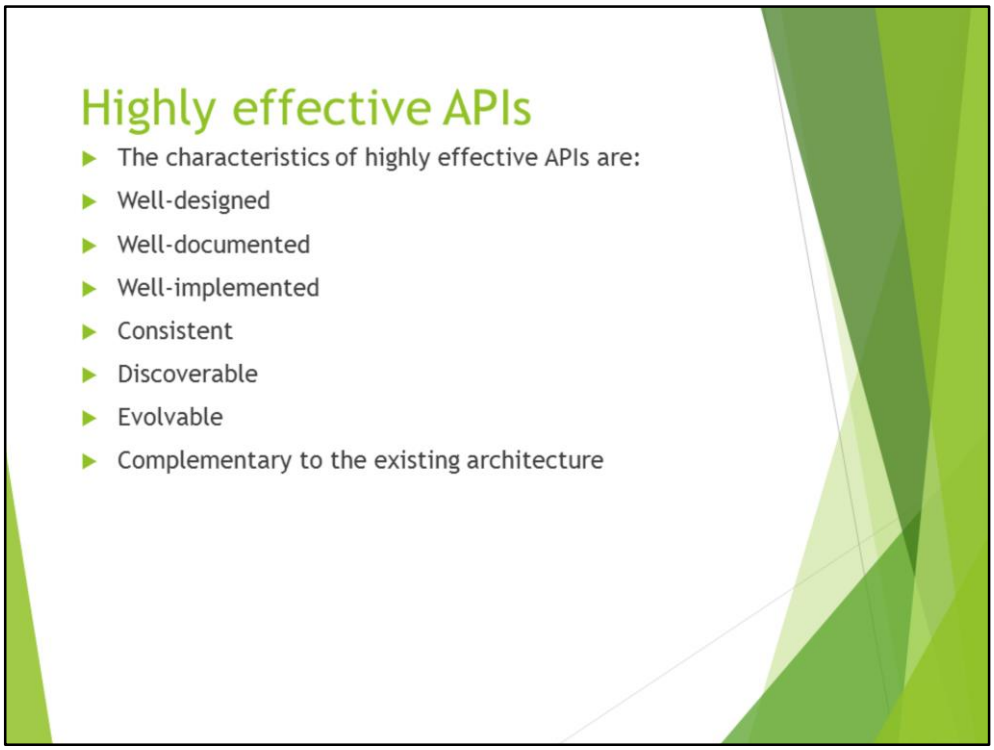
Look at all the APIs we have!, just walk down to the valley of happiness to start consuming them.

They are focusing on just the API they need to produce.

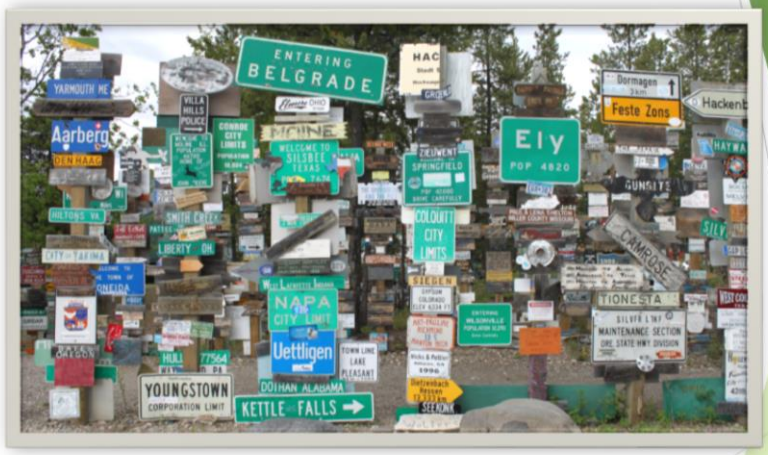
There are many inconsistencies that can happen when standards and best practices are not followed. Consumers become confused when trying to implement multiple APIs effectively.

Where we do not want to be.
"The bog of bad design"

The beautiful path above the ground where we can all see the resources we want to use for our applications.


Highly effective APIs
The characteristics of highly effective APIs are:
Well-designed
Well-documented
Well-implemented
Consistent
Discoverable
Evolvable
Complementary to the existing architecture

DX via Documentation



Well documented APIs enhance the experience for developers and have become an essential requirement for defining an API's success.