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

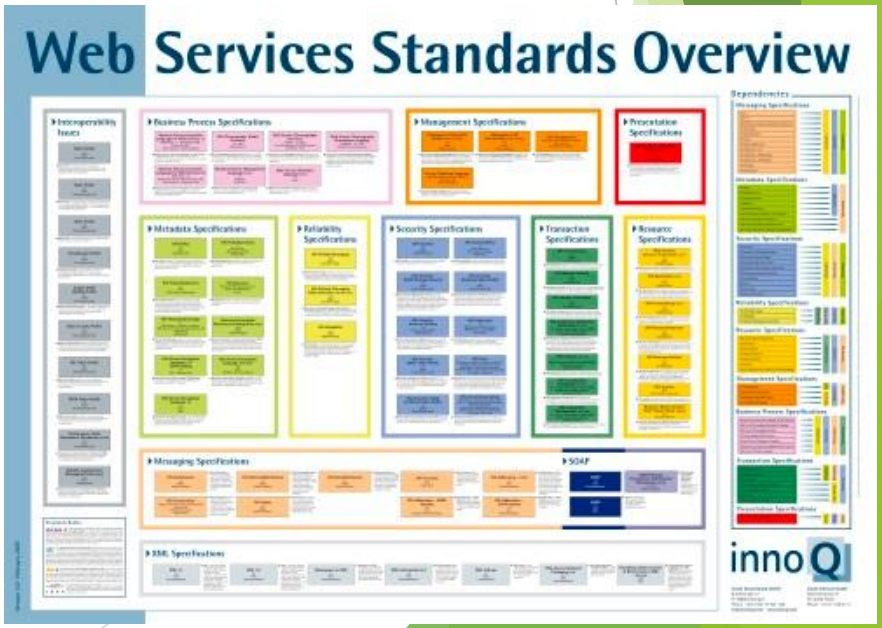
http://www.nitrogen-generators.com/wp-content/uploads/2014/02/grain-silos-nitrogen-generators-960x400.png

…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

```
- <wsdl:definitions name="AdderService" targetNamespace="http://adder.remote.ipojo.felix.apache.org/">
  - <wsdl:types>
    - <xsd:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
      targetNamespace="http://adder.remote.ipojo.felix.apache.org/">
        <xsd:element name="add" type="tns:add"/>
      - <xsd:complexType name="add">
        - <xsd:sequence>
            <xsd:element name="arg0" type="xsd:int"/>
            <xsd:element name="arg1" type="xsd:int"/>
          </xsd:sequence>
        </xsd:complexType>
        <xsd:element name="addResponse" type="tns:addResponse"/>
      - <xsd:complexType name="addResponse">
        - <xsd:sequence>
            <xsd:element name="return" type="xsd:int"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:schema>
    </wsdl:types>
  - <wsdl:message name="addResponse">
      <wsdl:part element="tns:addResponse" name="parameters"> </wsdl:part>
    </wsdl:message>
  - <wsdl:message name="add">
      <wsdl:part element="tns:add" name="parameters"> </wsdl:part>
    </wsdl:message>
  - <wsdl:portType name="AdderServicePortType">
    - <wsdl:operation name="add">
        <wsdl:input message="tns:add" name="add"> </wsdl:input>
        <wsdl:output message="tns:addResponse" name="addResponse"> </wsdl:output>
      </wsdl:operation>
    </wsdl:portType>
  - <wsdl:binding name="AdderServiceSoapBinding" type="tns:AdderServicePortType">
      <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    - <wsdl:operation name="add">
        <soap:operation soapAction="" style="document"/>
      - <wsdl:input name="add">
          <soap:body use="literal"/>
        </wsdl:input>
      - <wsdl:output name="addResponse">
          <soap:body use="literal"/>
        </wsdl:output>
      </wsdl:operation>
```

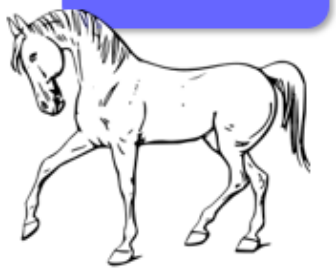


Web services are going back to basics…

- Mobile applications should and need to embrace the principles of the Web
- More interoperable message exchange protocol
- Improved mechanisms for scalability and robustness
- Remove the dependencies on WSDL, SOAP and XML
- Use HTTP as an application protocol by utilizing the full standard
- Simple and Open wins!!!!! (follow the Web's lead) like what Google, Facebook and Twitter have already done with their 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

```
GetCustomer();
UpdateCustomer(CustData);
AddCustomer(CustData)
AddCustomerPolicy(CustData);
DeleteCustomerPolicy();
```

Method Oriented
Methods & Parameters

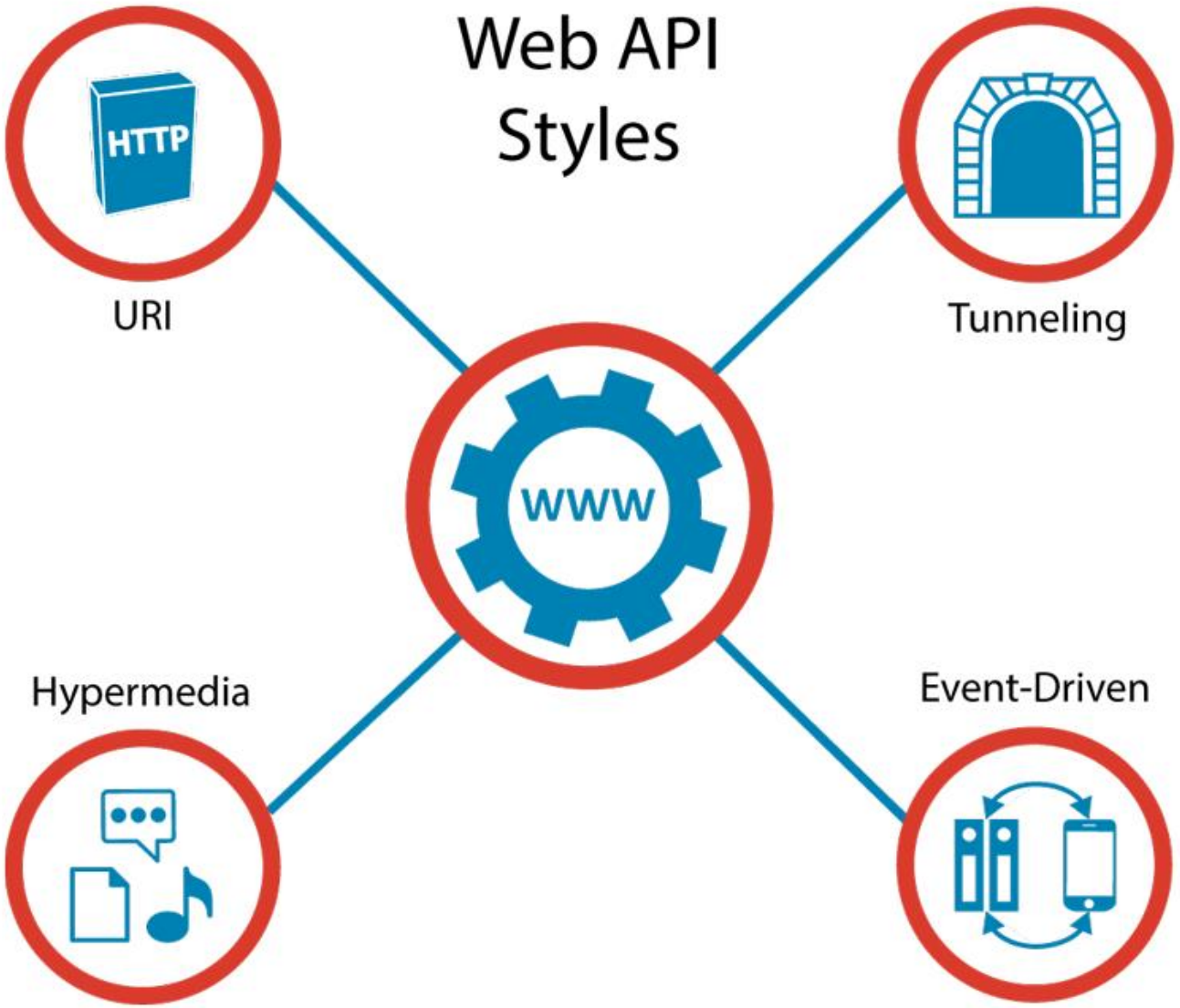
REST - GET, POST, PUT & DELETE

Web Oriented
Resources & Verbs

api.abc.com/Customer?PolicyNumber="880808"
api.abc.com/Customer?CustomerNumber="897080"
api.abc.com/Customer?FirstLast="Dobe"&ZIp="232..

api.abc.com/Customer/Policies
api.abc.com/Customer/Policies?Types="Verified"

RESTful
HTTP as
App
Protocol



SOAP/RPC
Transport
Agnostic

Task &WF
HTTP as
State
machine

"RESTless"
Asynch over
TCP/IP
as events
occur
Mobile &IOT

Styles that can help us implement request/response patterns over HTTP

Event Driven Async protocols

- **publish/subscribe messaging pattern (QUEUES)**
  - AMQP(**RESTless**) – Binary
  - MQTT(**RESTless**) – Binary
  - STOMP(**RESTless**) - Text
- **request/response messaging pattern**
  - CoAp(**RESTful**) - Binary
  - HTTP/2(**RESTful**) via WebSockets – Starts off as HTTP handshake continues on as TCP/IP
    - Server Push initiated on an open connection
    - Can support Binary but is backward compatible with HTTP1.2

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

Aspects of HTTP 1.1, RFC 2616 June 1999

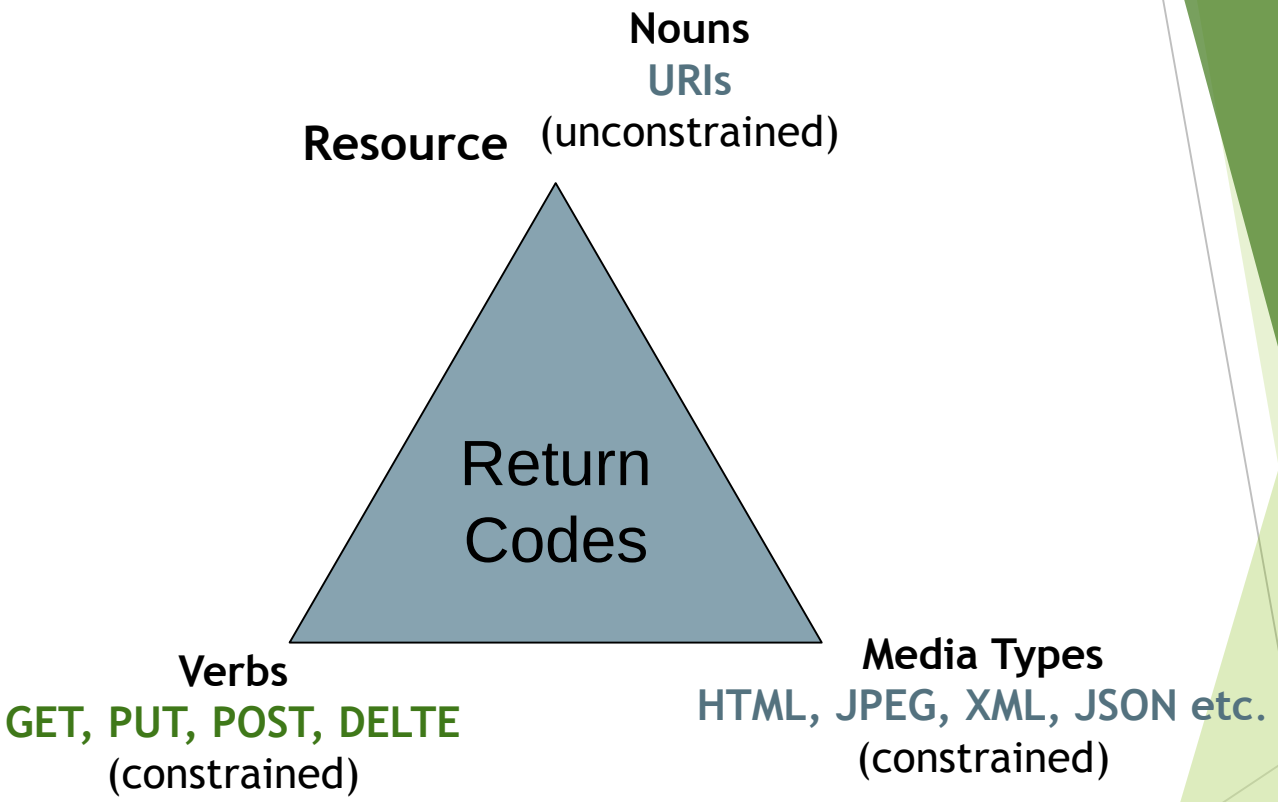
- HTTP is Generic
  - Just messages, and messages say what their payload is
- HTTP is Stateless
  - State is transferred in the messages.
  - Uses Hypermedia to cause changes in app. state
- HTTP is a Protocol
  - Defines REQUEST and RESPONSE messages
  - Encourages a loosely-coupled architecture.
  - HTTP is a Protocol

Aspects of HTTP 1.1, RFC 7231 June 2014

- Obsoletes 2616
- https://tools.ietf.org/html/rfc7231
- **Appendix B. Changes from RFC 2616**

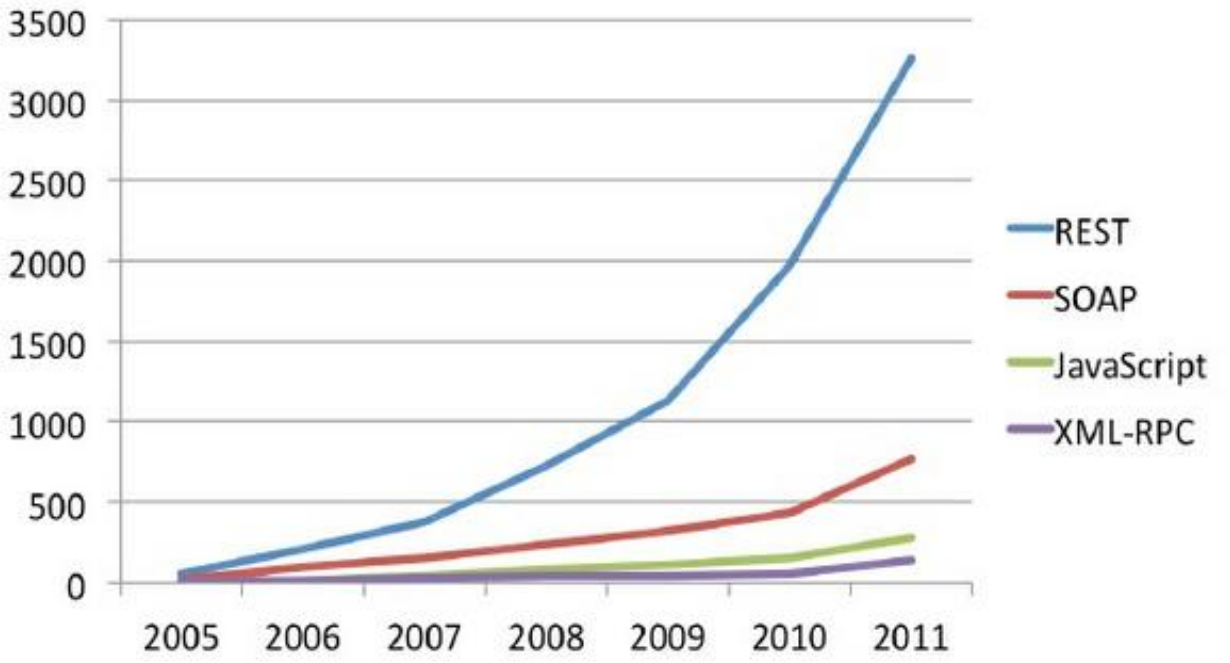
HTTP as an Application Protocol

- Architectural Style based on the Hypertext Transfer Protocol (HTTP) standard



HTTP Request/Response As REST

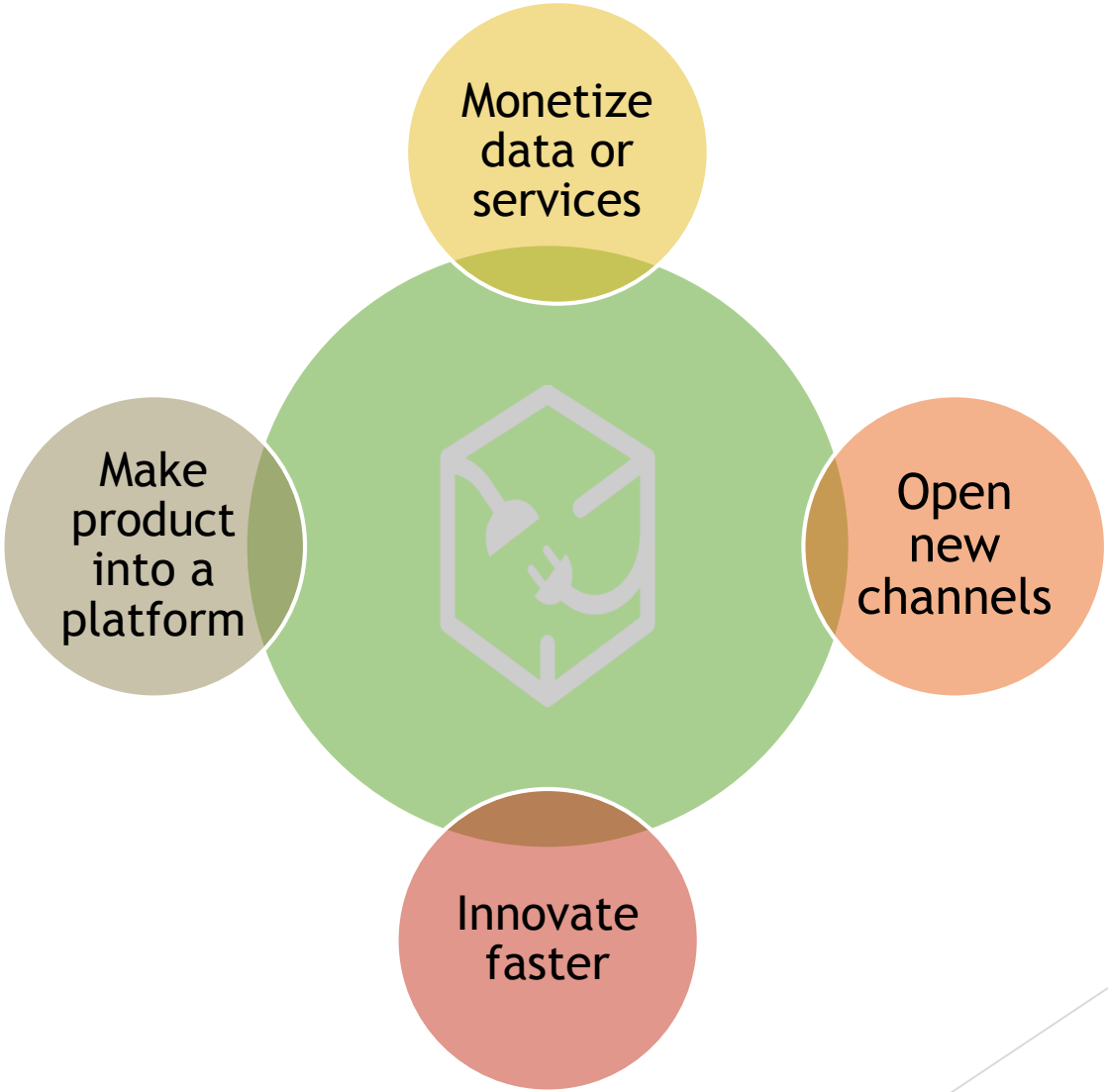
ProgrammableWeb



API protocols and styles

Based on directory of 5,100 web APIs listed at ProgrammableWeb, February 2012

REST APIs are the new engines of growth



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

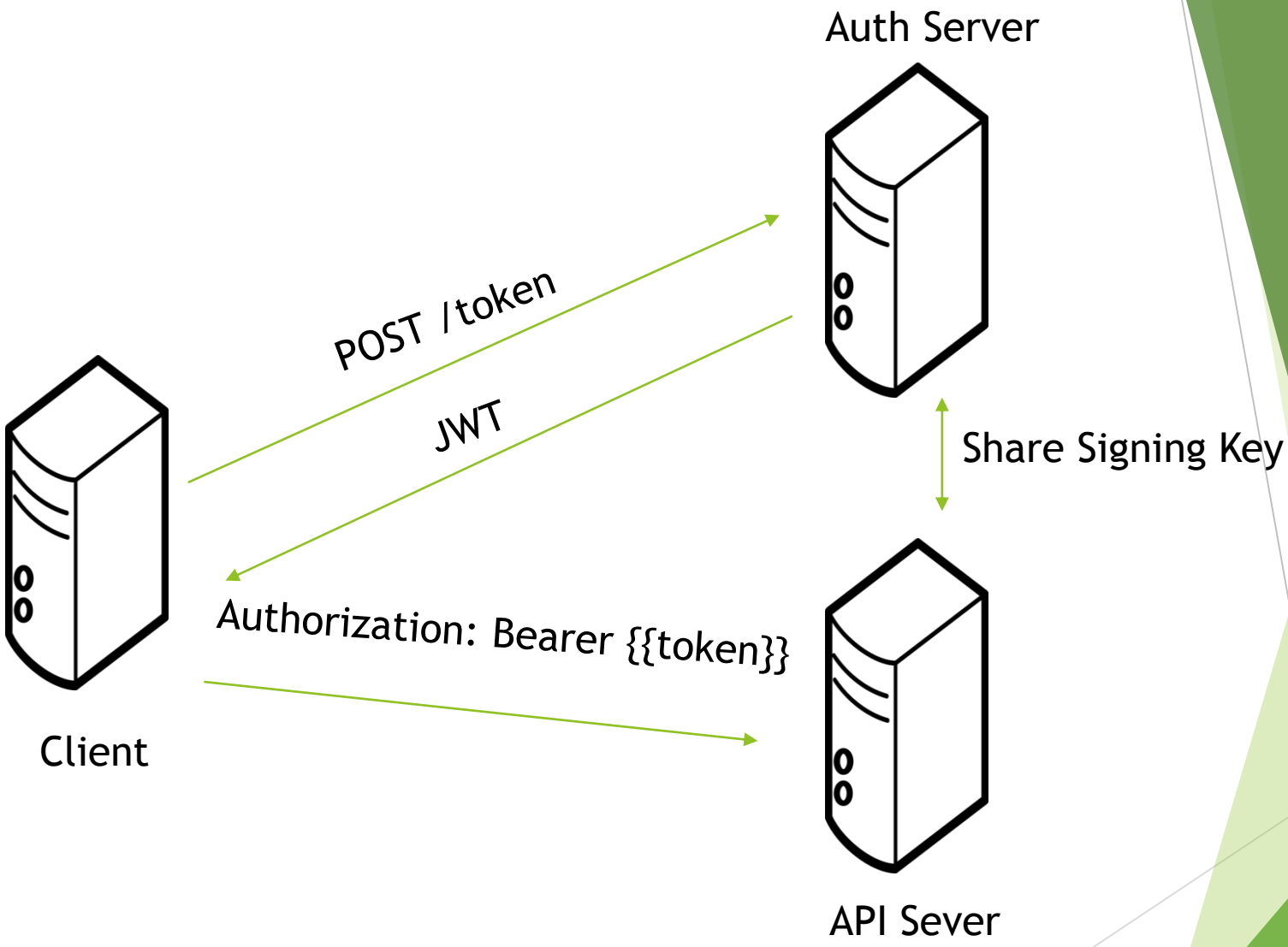
- CRUD operations on addressable resources
  - Create, Read, Update, Delete
- Performed through HTTP methods + URI
- URI standard format: scheme://host:port/path?queryString

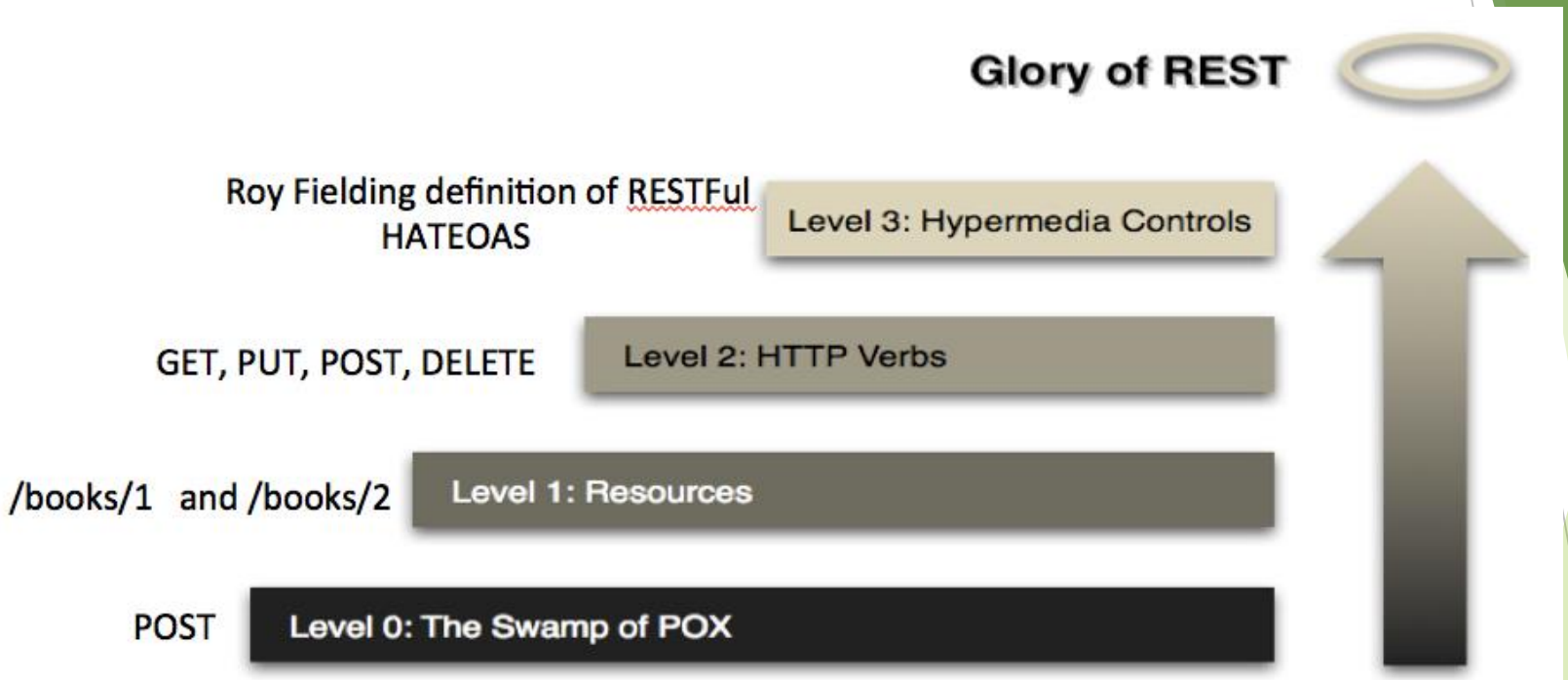
CRUD Operation	HTTP Method	Description
Create	POST	Create a new resource
Read	GET	Get a particular resource
Update	PUT	Update an existing resource
Delete	DELETE	Delete a particular resource
Delta Update	PATCH	Update a data change of an existing resource

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

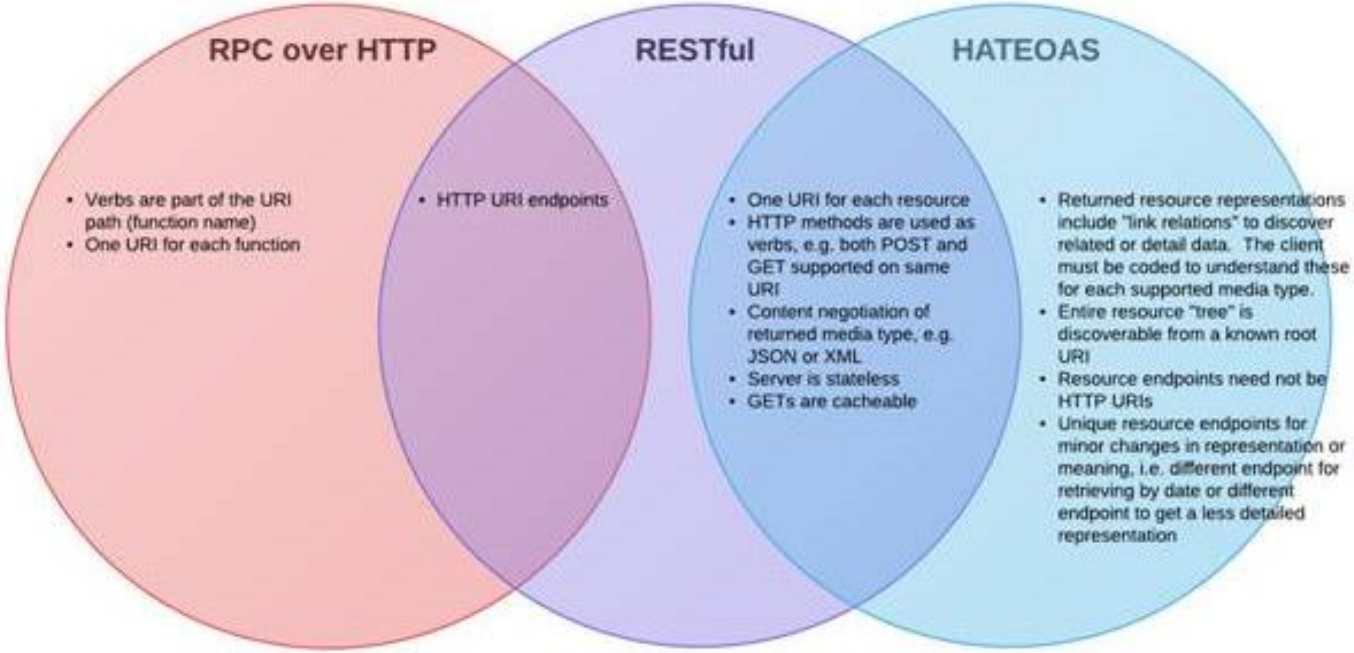
Basics of OAuth2





Source: http://martinfowler.com/articles/richardsonMaturityModel.html

Richardson Maturity Model Spectrum



But how it is being used today…

RESTful

Level 0 Not REST at all

- The "swamp of POX"

Level 1 Resources

- Organizing around multiple resources

Level 2 HTTP Verbs

- Uses HTTP Verbs appropriately

Good for when you can control client and server, basic CRUD-type applications.

Level 3 Hypermedia

- Uses Hypermedia Controls

More complex, but good when others will build clients

Hypermedia API

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')",
```

What are General Hypermedia Types

	Born	Created By	Goals
Odata	2010	Microsoft 2010 (v1,2,3) OASIS 2014 (4,5,6)	• Started as a data API for Microsoft Azure.
Collection+JSON	2011	Mike Amundsen	• Management and querying of simple collections
HAL – Hypertext Application Language	2011	Mike Kelly	• Simple way to hyperlink between resources
Siren	2012	Kevin Swiber	• Hypermedia specification for representing entities • Offers structures to communicate information about entities, action for executing state transitions and links for client navigation.
JsonAPI	2013	Steve Klabnik, Yehuda Katz Dan Gebhardt Tyler Kellen	• A generic media type that can work across a broad set of use cases, including generally used relationship types. • Designed to minimize both the number of request and amount of data transmitted between clients
JSON-LD + Hydra	2013	W3C Marcus Lanthaler; et al	• It is primarily intended to be a way to use Linded Data in Web-based programming environments, to build interoperable web services, and to store Linked Data in JSON-based storage engines.
Mason	2014	Jorn Wildt	• Hypermedia elements for linking and modifying data, features for communicating to client developers and standardized error handling. Based off of HAL but introduces Actions and Error handling
UBER	2014	Mike Amundsen Steve Klabnik	• Moving beyond HTTP. It is protocol agnostic.

http://www.odata.org/

akana
Powering the API Economy

RESTafarian


BUT IT'S NOT
"RESTful" IF YOU...
ENOUGH!

RESTful Insights

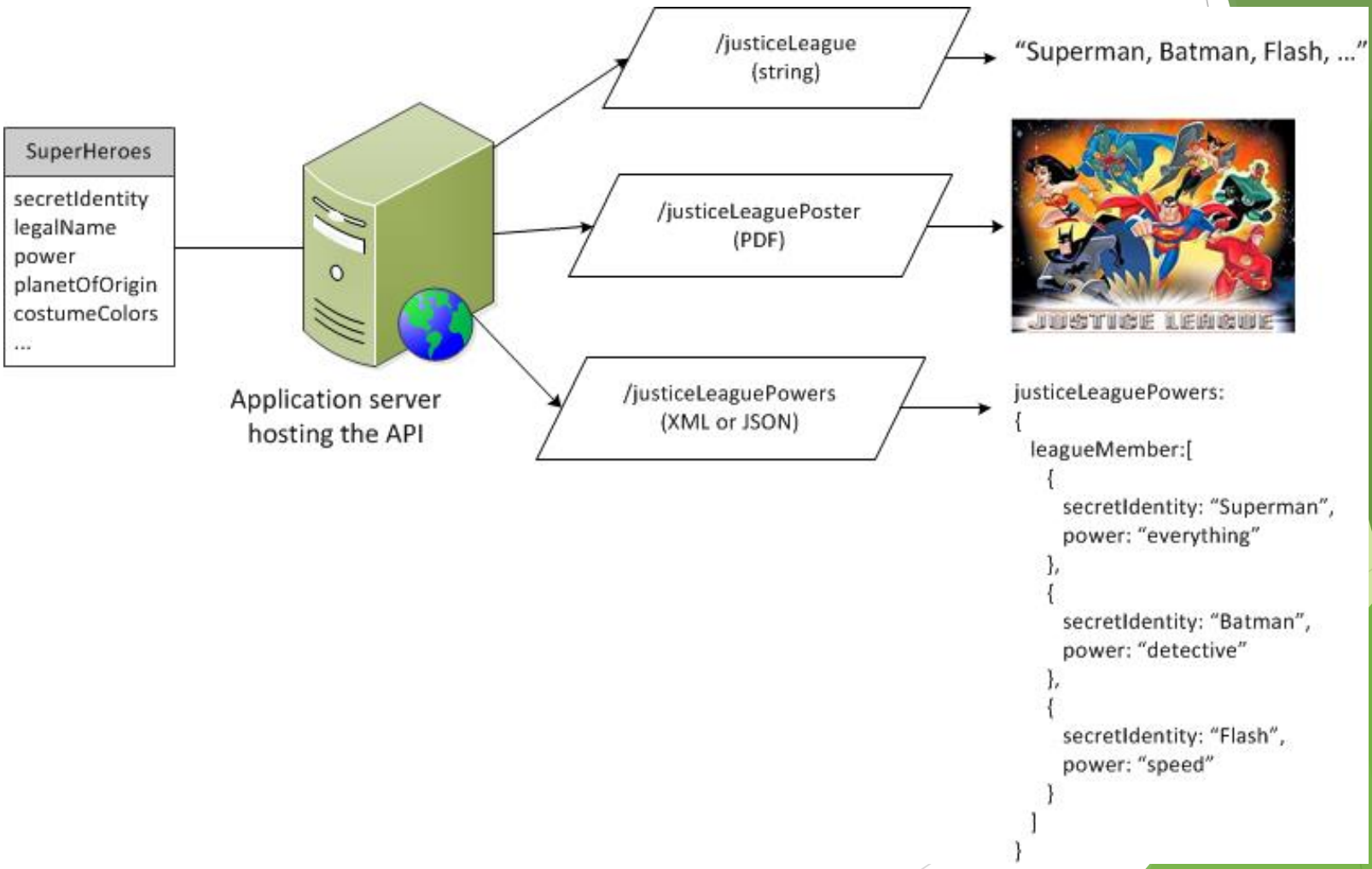
- REST is a style rather than a standard and it is surprisingly hard to implement consistently across an organization.
- Developing a REST API involves numerous subtle design decisions around areas such as resource design, method usage and status codes.
- It's easy to get hung up on debates around what 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.

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*

Intent

- Intent via content type determines representation



Classifying HTTP Methods

- **Safety** – calling the method does not cause side-effects.
  - safe methods are those that never modify resources
  - GET is the only safe method
- **Idempotancy** – clients can make the same call repeatedly while producing the same effect.
  - These methods achieve the same result, no matter how many times the request is repeated
  - only non idempotent method is POST
- Use proper HTTP Verbs!

Classifying HTTP Methods

Verb	Usage/Meaning	Safe	Idempotent
GET	• Retrieve a representation • Retrieve a representation if modified (caching)	Yes	Yes
DELETE	• Delete the resource	No	Yes
PUT	• Create a resource with client-side managed instance id • Update a resource by replacing • Update a resource by replacing if not modified (optimistic locking) • Partial update of a resource • Partial update a resource if not modified (optimistic locking)	No	Yes
POST	• Create Resource • Create a sub-resource	No	No
PATCH	• Call to a nonexistent resource is handled by the server as a “create” • Call to an existent resource is handled by the server as an “update” • If a request contains “If-Match” header, the service MUST NOT treat PATCH as an insert • If a request contains “If-None-Match” header with value of “*”, the service MUST NOT treat the PATCH as an update	No	Yes

HTTP Response Codes

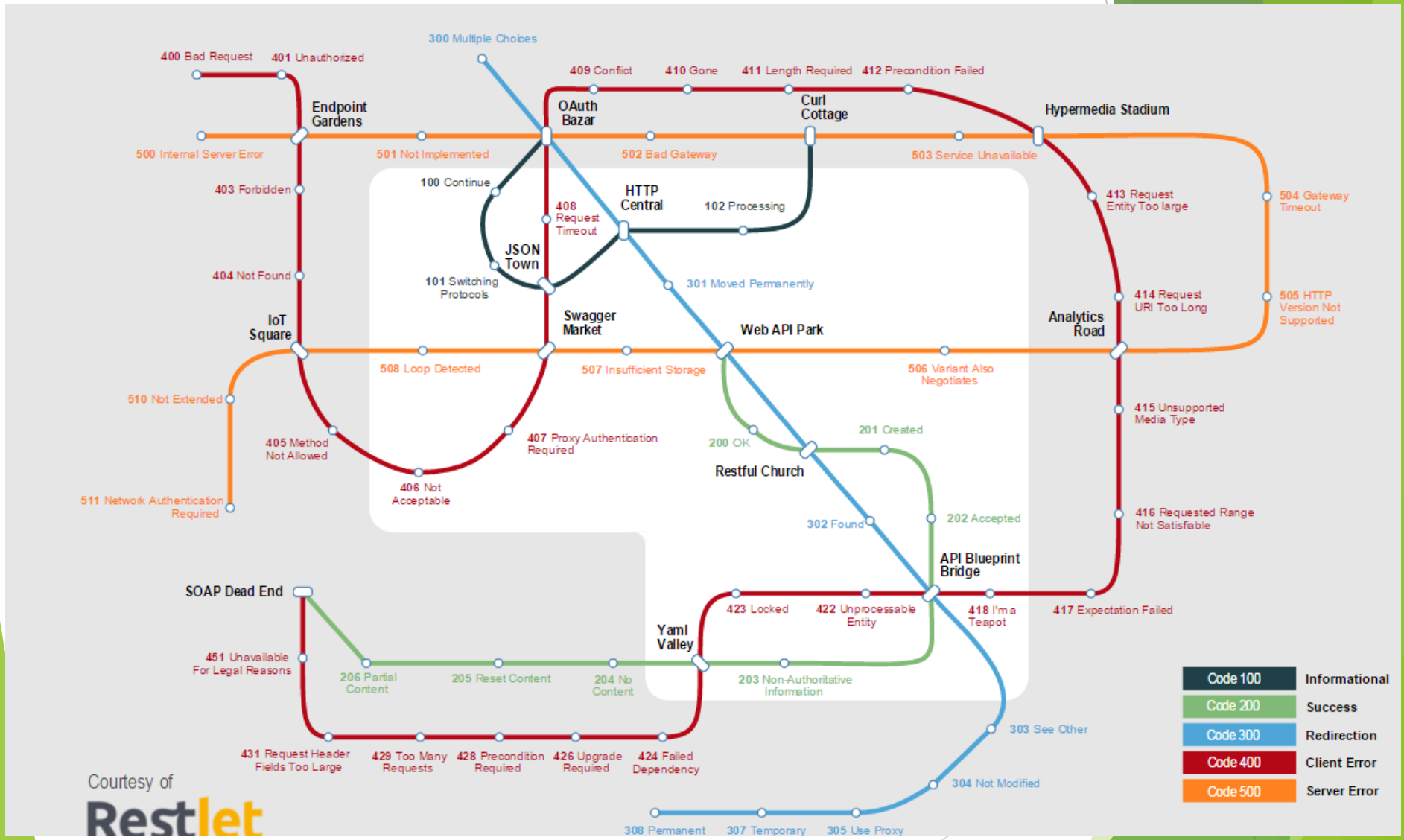
» HTTP response codes standardize a way of informing the client about the result of its request.

- **2XX** – indicates that the client's request was successfully received, understood, and accepted
  - 200 OK
  - 201 Created
  - 202 Accepted
  - 204 No Content
- **3XX** – indicates that further action needs to be taken by the user agent in order to fulfill the request

» **4XX** – intended for cases in which the client seems to have erred
  - 400 Bad Request
  - 401 Unauthorized
  - 403 Forbidden
  - 404 Not Found
  - 405 Method not Found
  - 410 Gone,
  - **418 TeaPot, 420 CO,WA** ☺

» **500** – status codes beginning with the digit "5" indicate cases in which the server is aware that it has erred or is incapable of performing the request
  - 500 Internal Server Error

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*

Pagination

- RESTful APIs that return collections MAY return partial sets. Consumers of these services MUST expect partial result sets and correctly page through to retrieve an entire set.
  - Server Driven
  - Client Driven

- Industry Examples:

To get records 50 through 75 from each system, you would use:

- Facebook - **offset 50** and **limit 25**
- Twitter - **page 3** and **rpp 25** (records per page)
- LinkedIn - **start 50** and **count 25**

Versioning

With REST, the contract is the uniform interface, enabling:

- Versioning within the representation
- Versioning the representation
- Versioning the resource or API as a whole

- Industry tips for versioning:
  - Twilio: */2010-04-01/Accounts/*
  - salesforce.com: */services/data/v20.0/sobjects/Account*
  - Facebook: *?v=1.0*
  - *Use Semantic Versioning for operational debugging use*
    - *V1.2 Major.Minor = Breaking.nonBreaking Changes*
      - */v1.2/Accounts*
    - *Gives meaning to versioning*
    - *Allows apps to mange dependency on your API*
  - ***Do not version the resource, use Hypermedia!***
    - ***Do we check the versions of websites we visit?***
    - ***Allows APIs to be evolvable***

Review

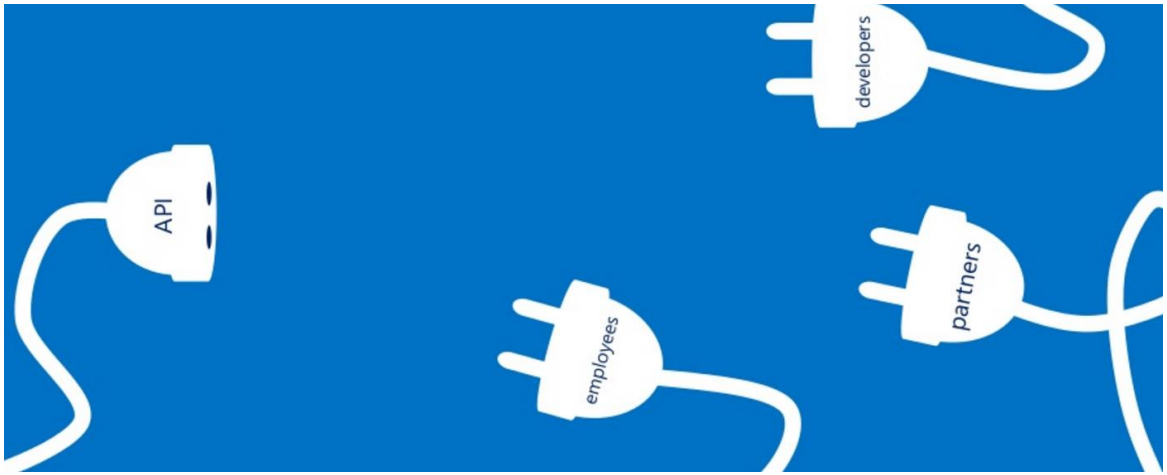
- REST is a new architectural style for web services
- Resources are logical entities that have a have an intent driven representation.
  - Nouns
  - JSON
- Use HTTP protocol to communicate
  - Verbs: GET, PUT, POST, DELETE, PATCH
  - Status Codes: 200, 404, 500, etc.
- Headers
  - Request/Response
- 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

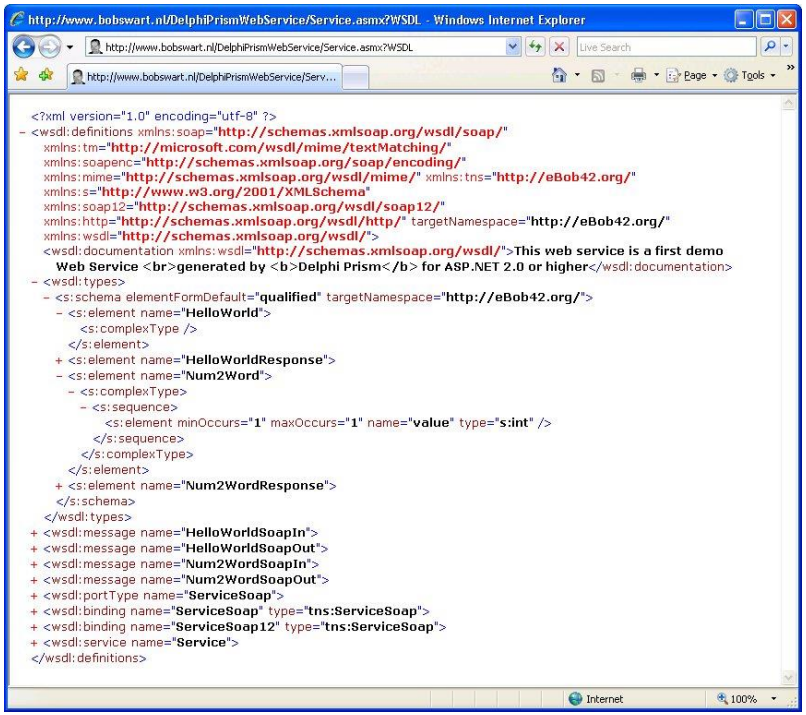
apinf CapitalOne Google

IBM intuit mashape

Microsoft PayPal Restlet SMARTBEAR

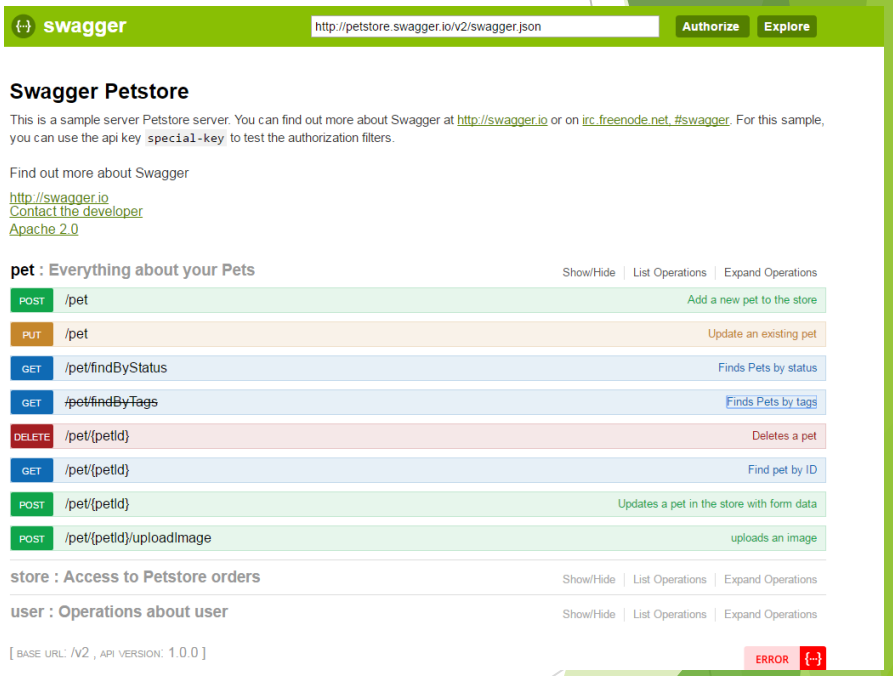
Swagger

SOAP WSDL



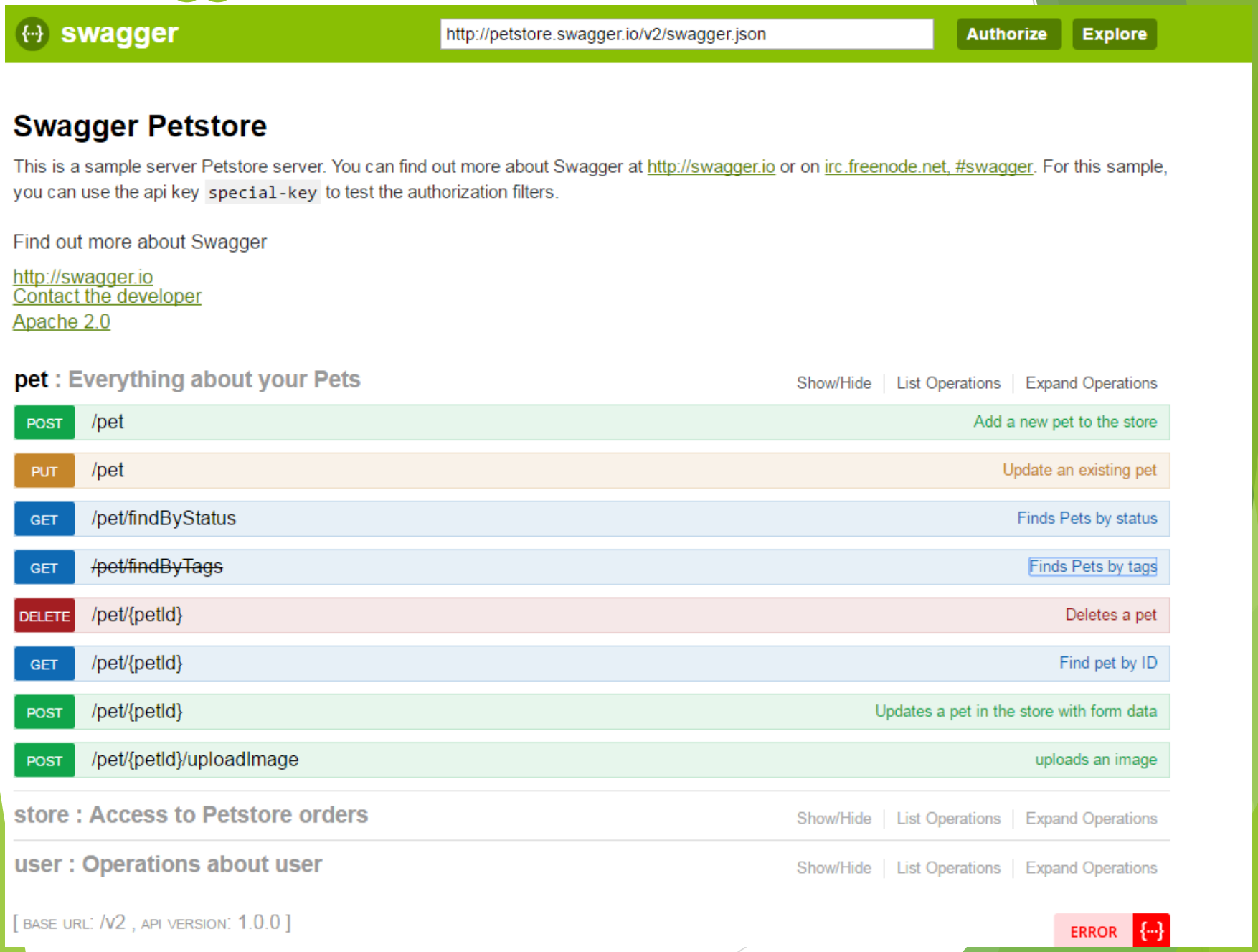
Machine readable - 010101011101110101

REST Docs (Swagger UI)



Human readable – I love APIs

Swagger UI Petstore



HTTP messaging/testing tools

Postman

Advanced REST client

JSON Editor

The 200 OK "Academic" Path to Better APIs

Pictures borrowed from Everett Toews, Evangelist at Rackspace

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?

API Providers View

the valley of h**API**ness

API Designers View

API Consumers view

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

How do we get here?

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