

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...



WSDL Rocks!!!

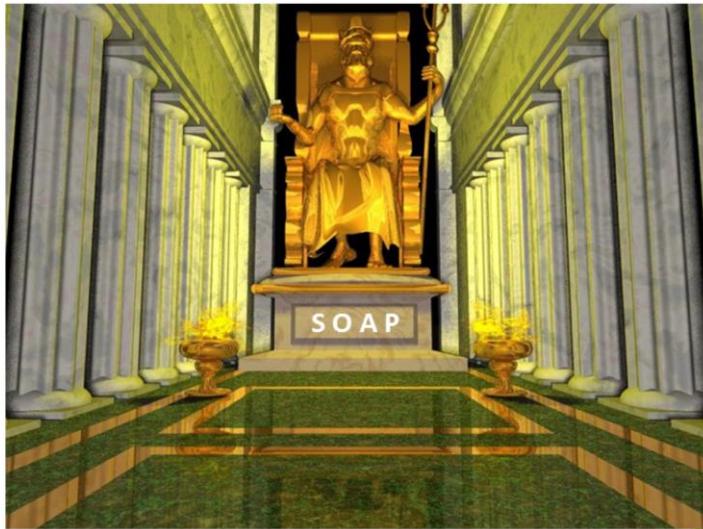
A cartoon illustration of a lion with a large, bushy black mane and a pink face. It is sitting at a desk, looking at a computer monitor. A speech bubble above it contains the text "WSDL Rocks!!!". The background features green diagonal stripes.

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.ipops.felix.apache.org">
  <wsdl:types>
    <xsd:schema attributeFormDefault="unqualified" elementFormDefault="qualified">
      <targetNamespace>http://adder.remote.ipops.felix.apache.org</targetNamespace>
      <xsd:import namespace="http://adder.remote.ipops.felix.apache.org" schemaLocation="AdderService.wsdl"/>
      <xsd:complexType name="add">
        <xsd:sequence>
          <xsd:element name="arg1" type="xsd:int"/>
          <xsd:element name="arg2" type="xsd:int"/>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:complexType name="m-addResponse" type="tns: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:add" name="parameters"/>
  </wsdl:message>
  <wsdl:operation name="add">
    <wsdl:input element="tns:add" name="parameters"/>
    <wsdl:output element="tns:m-addResponse" name="m-addResponse"/>
  </wsdl:operation>
  <wsdl:binding name="AdderServiceSoapBinding" type="tns:AdderServicePortType">
    <wsdl:bindingStyle>document</wsdl:bindingStyle>
    <wsdl:operations>
      <wsdl:operation name="add">
        <wsdl:input soapAction="" style="document">
          <xsd:body use="literal"/>
        </wsdl:input>
        <wsdl:output name="addResponse">
          <xsd:body use="literal"/>
        </wsdl:output>
      </wsdl:operation>
    </wsdl:operations>
  </wsdl:binding>
</wsdl:definitions>
```

Web Services Standards Overview



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 based API and REST based APIs

SOAP v REST



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

SOAP

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

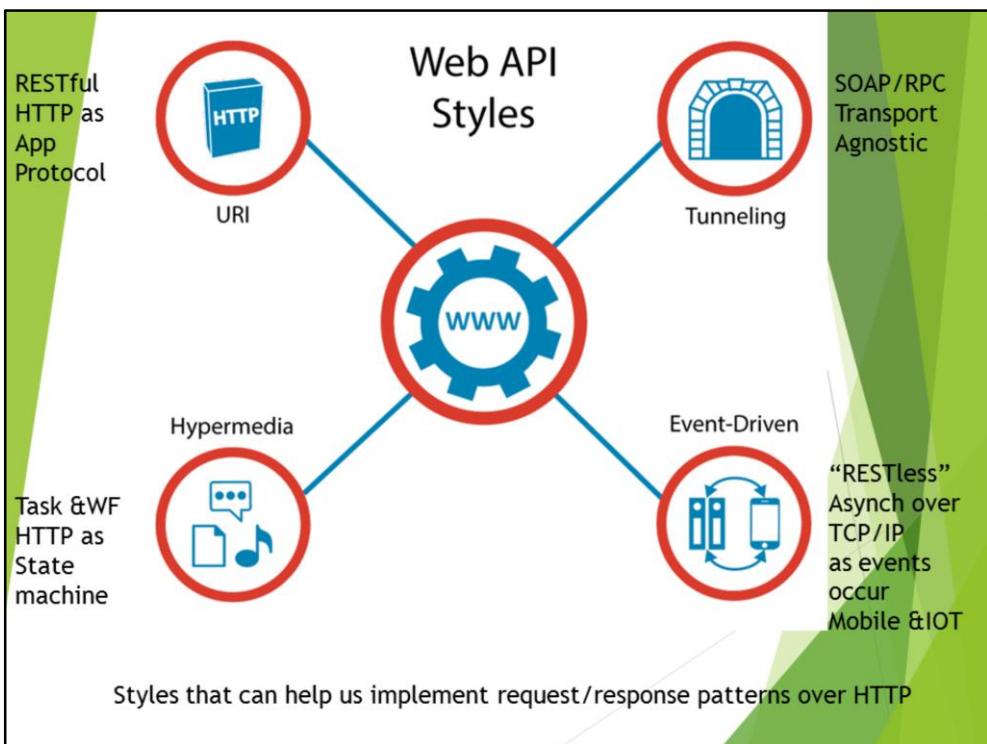
Method Oriented
Methods & Parameters

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"
```

SOAP only Uses POST and sometimes GET.



[http://www.apiacademy.co/resources/api-design-201-web-api-architectural-styles/ Tunneling Style](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-agnostic 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:

Expose 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.

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

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

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 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.

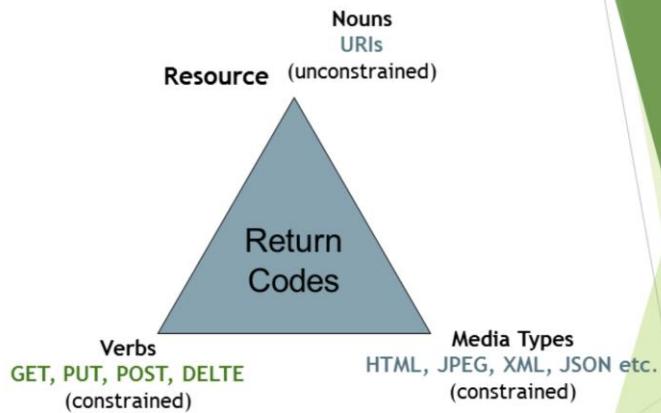
For example, we might be looking at a hypertext document (HTML) that shows the current state of the inventory in a book shop. This document may contain links to each item in inventory. By generating a request by clicking on a link, we are transitioning the state of our application to the state of showing 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.

HTTP as an Application Protocol

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



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/recording HTTP/1.1
Host: media.example.com
Accept: application/xml
```

Method

Resource

Response

```
HTTP/1.1 200 OK
Date: Tue, 08 May 2007 16:41:58 GMT
Server: Apache/1.3.6
Content-Type: application/xml; charset=UTF-8

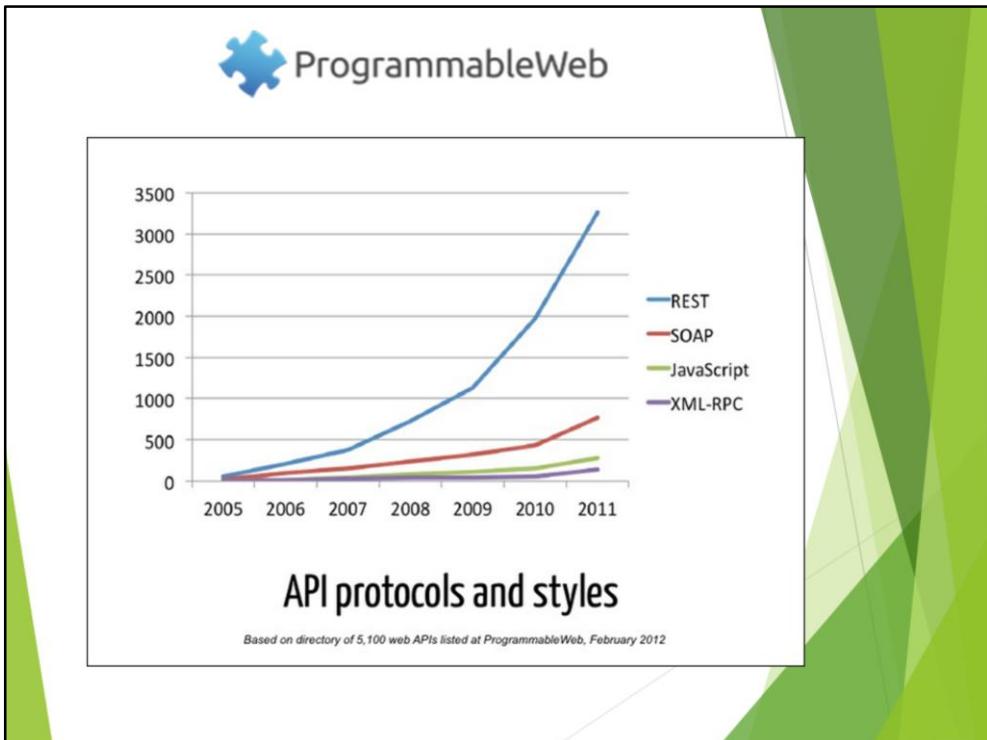
<?xml version="1.0"?>
<recordings xmlns="...">
  <recording>...</recording>
  ...
</recordings>
```

State transfer

Response Code

Representation

The diagram illustrates the components of an HTTP Request/Response as REST. The request line is labeled 'Method' and 'Resource'. The response code is labeled 'Response Code'. The XML representation is labeled 'Representation'.



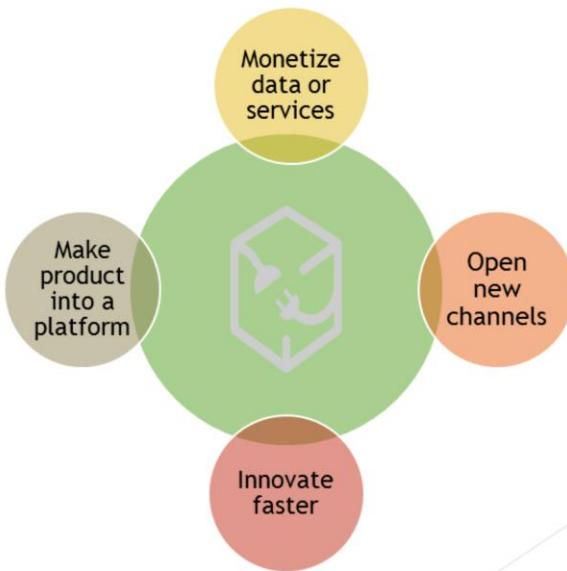
SOAP is not Dead but it is aging.

<http://www.programmableweb.com>

0-5000 = 7 years

5000 to 15,500 = in 4 years it tripled

REST APIs are the new engines of growth



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 “Representational State Transfer” 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 Representational State Transfer.
- ▶ Architectural style described by 6 constraints:

1	<i>Uniform Interface</i>	simplified architecture
2	<i>Stateless</i>	session state is held in the client <ul style="list-style-type: none">• Each request must contain all the information the server needs to correlate the work
3	<i>Cacheable</i>	clients can cache responses to improve performance
4	<i>Client-server</i>	independent implementation of servers and clients
5	<i>Layered System</i>	servers are obfuscated from clients <ul style="list-style-type: none">• No Layer can “see past” the next layer• Ex: API Gateway
6	<i>Code on Demand*</i>	customizable functionality <i>The only optional constraint</i> Allows Clients to be updated by the server

REST over HTTP - Uniform interface

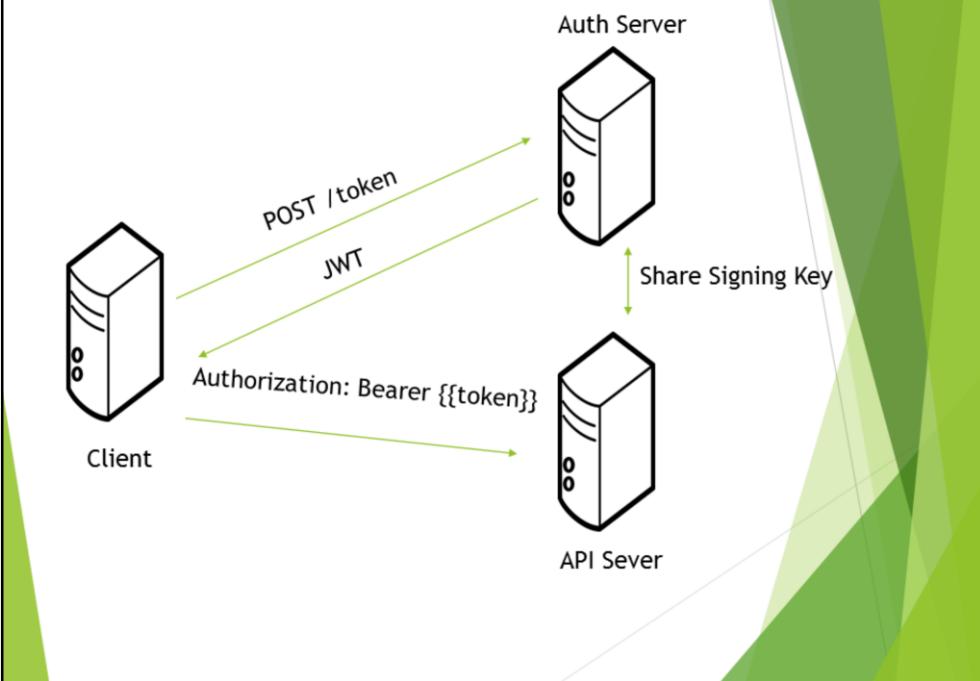
- ▶ CRUD operations on addressable resources
 - ▶ Create, Read, Update, Delete
- ▶ Performed through HTTP methods + URI
- ▶ URI standard format: scheme://host:port/path?queryString

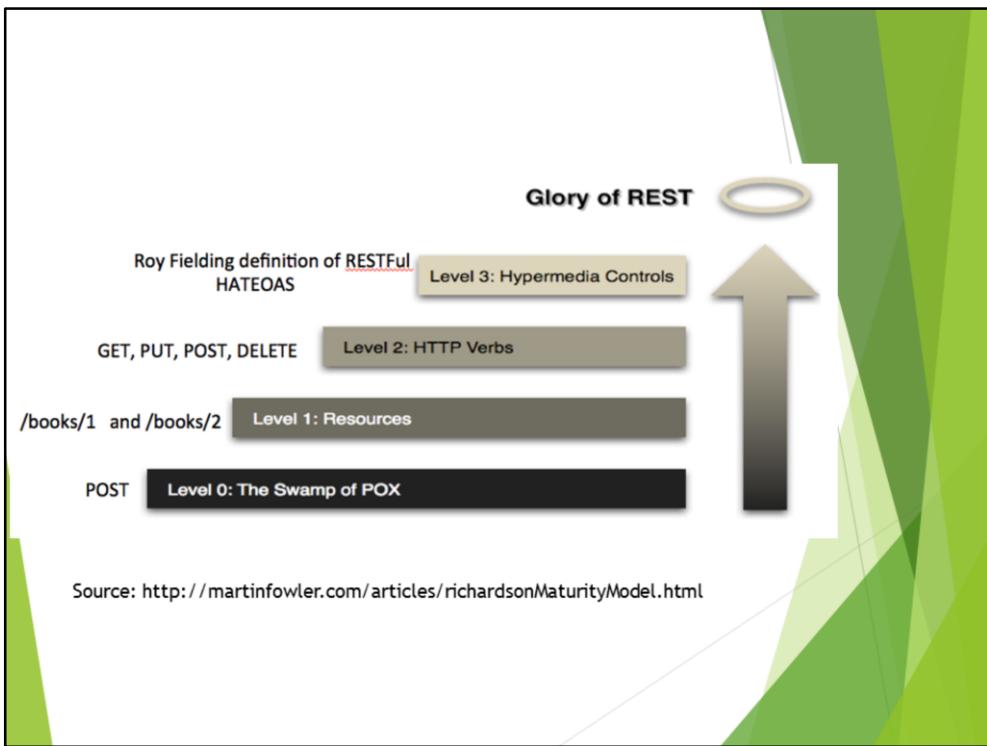
CRUD Operation	HTTP Method	Description
Create	POST	Create a new resource
Read	GET	Get a particular resource
Update	PUT	Update an existing resource
Delete	DELETE	Delete a particular resource
Delta Update	PATCH	Update a data change of an existing resource

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





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:

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

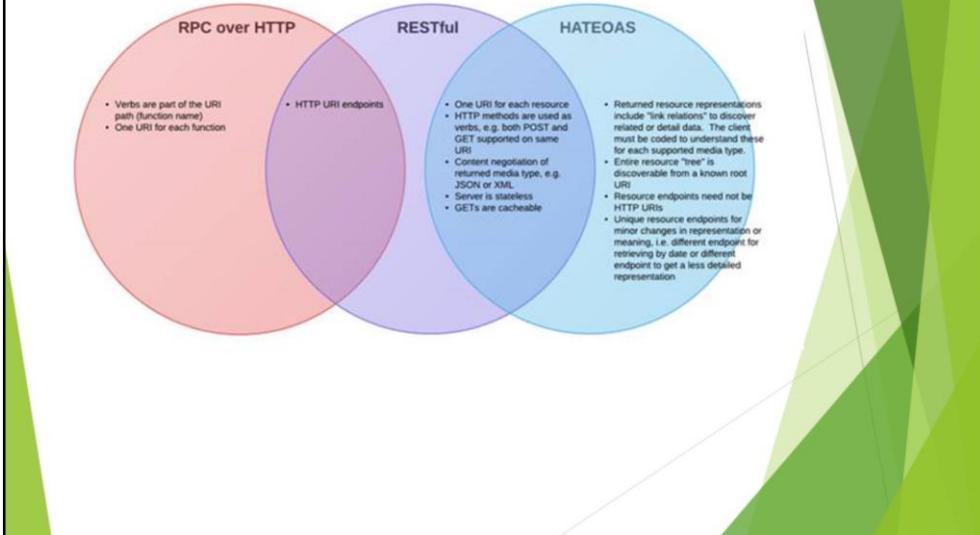
Level 0: The Swamp of POX – At this level you are dealing with mostly plain old XML, a single endpoint and using just the POST method, most likely with SOAP

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's definition of RESTful APIs and the "Glory of REST".

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

Level 3 Hypermedia

- Uses Hypermedia Controls

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

More complex, but good when others will build clients

Hypermedia API

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(34wtn2c0hkuk5ekg0pjrs13b))/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/08D1D58E987DE78B
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)	<ul style="list-style-type: none"> Started as a data API for Microsoft Azure.
Collection+JSON	2011	Mike Amundsen	<ul style="list-style-type: none"> Management and querying of simple collections
HAL – Hypertext Application Language	2011	Mike Kelly	<ul style="list-style-type: none"> Simple way to hyperlink between resources
Siren	2012	Kevin Swiber	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> It is primarily intended to be a way to use Linked 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	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> Moving beyond HTTP. It is protocol agnostic.

<http://www.odata.org/> 

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

RESTafarian



"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

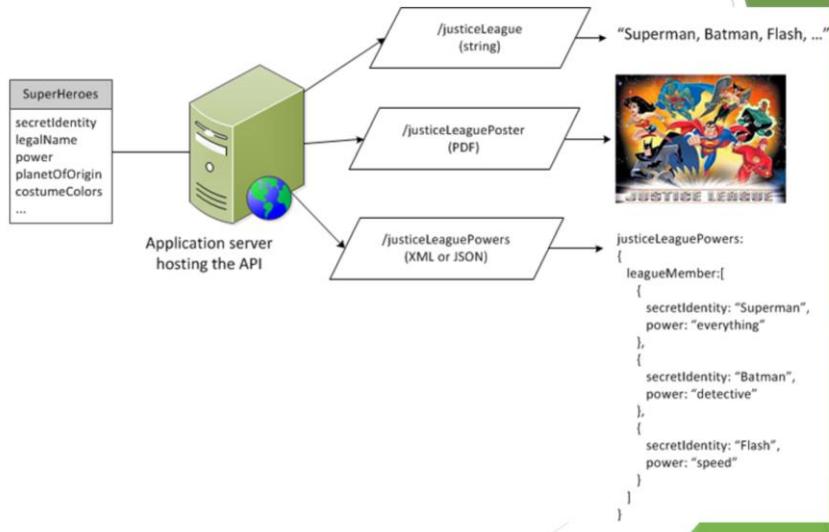


Naming Resources

- ▶ 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

Intent

► Intent via content type determines representation



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

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
- ▶ **Idempotency** - 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!

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

Classifying HTTP Methods

Verb	Usage/Meaning	Safe	Idempotent
GET	<ul style="list-style-type: none">• Retrieve a representation• Retrieve a representation if modified (caching)	Yes	Yes
DELETE	<ul style="list-style-type: none">• Delete the resource	No	Yes
PUT	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• Create Resource• Create a sub-resource	No	No
PATCH	<ul style="list-style-type: none">• 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

Verbs have unintended Side Effects if used improperly.

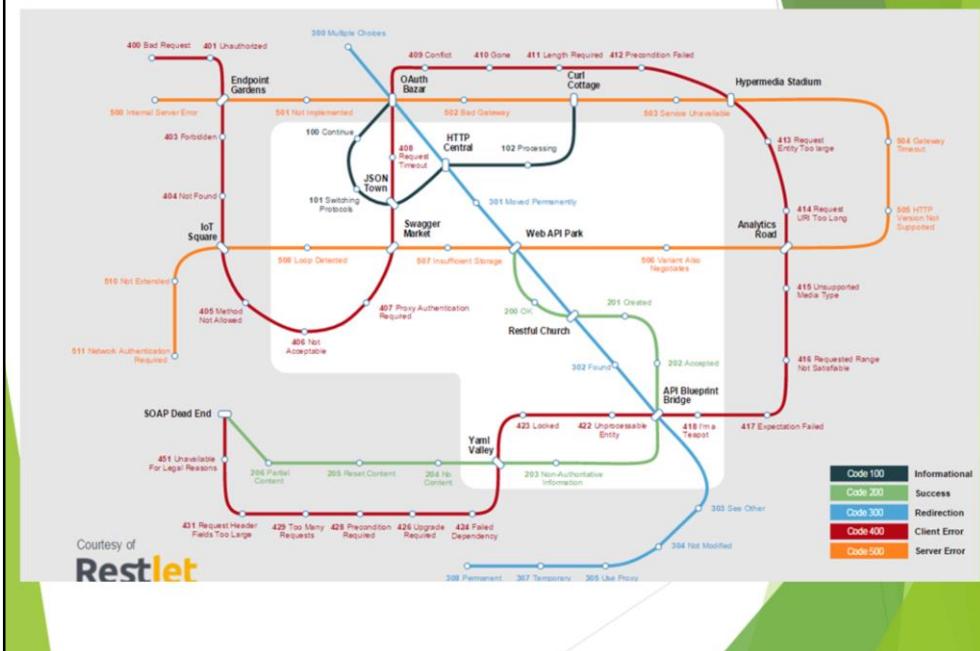
HTTP Response Codes

- » 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



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



HTTP Headers

- ▶ They define the operating parameters of the transaction, the so called metadata. In REST style, more important than 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: <ul style="list-style-type: none">• application/xml• text/xml• application/json• text/javascript

- ▶ Response Headers

Header	Description
Date	The date the response was processed, 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**

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

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 manage 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

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/Search/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)

{JSON}
<xml/>



RAML

apiblueprint



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 in 10,000 production deployments.”





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



The screenshot displays two side-by-side views of the Swagger UI interface.

Left Side (SOAP WSDL):

- A green icon with three curly braces is at the top left.
- The title "SOAP WSDL" is centered above a screenshot of Internet Explorer showing a complex XML document.
- The XML content is a WSDL (Web Services Description Language) document, listing various operations like `getPet`, `updatePet`, and `deletePet`.
- Below the browser window, the text "Machine readable - 010101011101110101" is displayed.

Right Side (REST Docs (Swagger UI)):

- The title "REST Docs (Swagger UI)" is centered above a screenshot of the Swagger Petstore UI.
- The UI shows a list of operations under the "pet" resource, such as `getPet`, `updatePet`, `deletePet`, `petStatus`, `petUploadImage`, `petInfo`, and `petPhoto`.
- Each operation has a brief description and a "Try it out" button.
- Below the UI, the text "Human readable - I love APIs" is displayed.

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

Method	Path	Description
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

user : Operations about user

[BASE URL: /V2 , API VERSION: 1.0.0]

ERROR { }

Show/Hide | List Operations | Expand Operations

Show/Hide | List Operations | Expand Operations

HTTP messaging/testing tools



Postman



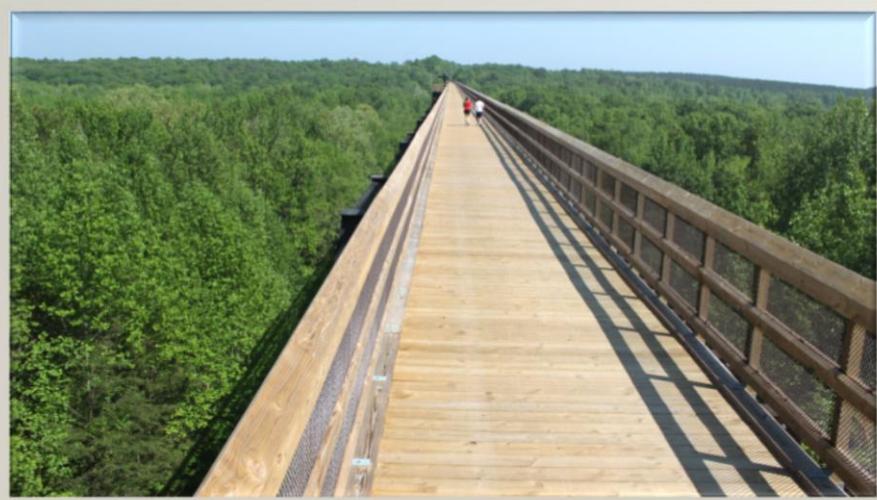
Advanced REST client



JSON Editor

Show how Postman can import Swagger File

The 200 OK “Academic” Path to Better APIs



Pictures borrowed from Everett Toews, Evangelist at Rackspace

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?

API Providers View



the valley of hAPIness

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

API Designers View



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

API Consumers view



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”



How do we get here?



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.