**API Related Standards**

<https://github.com/Leventhan/api-development-tools>

* RESTful Modeling Language
  + RAML
  + Swagger
  + API Blueprint
  + IO Docs
  + Mashape
  + WADL
  + JSON Hyper-Schema
  + ALPS (alps.io)
* Security Standards
  + OAuth, Ping Identity, Open Id, Open Id Connect,
  + Neo-security stack
    - JSON Identity Protocol (JOSE) – (<https://securityblog.redhat.com/2015/04/01/jose-json-object-signing-and-encryption/> )
      * [JWA](http://tools.ietf.org/html/draft-ietf-jose-json-web-algorithms-40) - JSON Web Algorithms, describes cryptographic algorithms used in JOSE
      * [JWK](http://tools.ietf.org/html/draft-ietf-jose-json-web-key-41) - JSON Web Key, describes format and handling of cryptographic keys in JOSE
      * [JWS](http://tools.ietf.org/html/draft-ietf-jose-json-web-signature-41) - JSON Web Signature, describes producing and handling signed messages
      * [JWE](http://tools.ietf.org/html/draft-ietf-jose-json-web-encryption-40) - JSON Web Encryption, describes producing and handling encrypted messages
      * [JWT](http://tools.ietf.org/html/draft-ietf-oauth-json-web-token-32) - JSON Web Token, describes representation of claims encoded in JSON and protected by JWS or JWE
    - Currently being defined in IETF
* MQTT, Web-sockets
* JSON Schema
* Open Data

API Interoperability

* JSON-P: a technique used by web developers to overcome the cross-domain restrictions imposed by browsers

Real-time API Related

* Transport
  + Comet, Websocket (both HTTP)
* Protocols
  + With formal pub/sub semantics
    - XMPP
    - Bayeux
    - STOMP
  + Adhoc, i.e. subscription expressed in the request
    - JSON
* Push
  + PubSubHubbub (google)
  + WebHooks – definition of how to exchange information over internet.

API Discoverability

* APIs.json (<http://apisjson.org/> )
* API Commons (<http://apicommons.org/> )

Tools

<https://github.com/apiaryio/dredd/>

<http://apis.io/> - for API Search

<https://www.apitools.com/>

<https://gelato.io/> - Developer portal as a service. Import API definitions in Swagger. Can integrate with Kong. Can generate sample code from API definitions

<https://tyk.io/> - Open source API gateway with developer portal

<http://cycles.apiops.net/> Thought process around API Ops/Governance/Development

<https://github.com/apickli/apickli> - A node.js API integration testing framework

## RAML (RESTful API Modeling Language)

* About
  + YAML-based language for describing practically-RESTful APIs
  + Encourages reuse, enables discovery and pattern-sharing
  + Trademark of Mulesoft
* Industry backing
  + Lead by Mulesoft, Angular JS, Box, PayPal, SOA Software, Cisco
* Reason for Broader acceptance
  + Open-sourced along with tools and parsers
  + RAML has introduced language features that support structured files, inheritance and address cross cutting concerns
* Resources
  + <https://en.wikipedia.org/wiki/RAML_(software)>
* You can use RAML in a number of ways;
  + It can be used in the design and specification stage to describe the API you plan to build
  + You can use RAML to generate documentation
  + RAML can be used to implement interactive API consoles
  + You can test against RAML
  + It be used to generate mock API responses
* Learnings
  + Biggest plus point over Swagger, it supports OOM (multiple inheritance)
  + Uses variables that can be very helpful when defining multi-part APIs

## Swagger

* About
  + Oldest specification (2011) but has the largest and most active developer community
  + Swagger is a specification and complete framework implementation for describing, producing, consuming, and visualizing RESTful web services.
  + The overarching goal of Swagger is to enable client and documentation systems to update at the same pace as the server
  + Swagger 2.0 takes a YAML based API design-first approach
  + Renamed as Open API Specification. Donated as Open API Initiative
* Industry backing
  + Reverb, 3Scale, SmartBear & Apigee

## API Blueprint

* About
  + API Blueprint is a documentation-oriented API description language
  + API Blueprint, its parser, and most of its tools are completely open sourced
  + Can be used by a variety of tools over the entire lifecycle of the API starting from definition to testing
  + Generate documentation/test suite automatically
* Industry Backing
  + Apiary
* Resources
  + <https://apiblueprint.org/>

## ALPS

* About
  + ALPS = Application Level Profile Semantics
  + ALPS is a Profile format for describing the bounded context of a service.
  + ALPS describes the operations (actions) and data elements of a service. That’s all.
* People Behind
  + Mike Amundsen
  + Mark Foster (Apiary)
* Industry Backing
* Resources
  + <http://apievangelist.com/2015/03/10/what-is-alps/>

## Richardson Maturity Model

* Level 0: Using HTTP as a funnel to route requests which are more like RPC invocations
  + Examples: SOAP
* Level 1: Use of resources. Defining resources and using them as an entry point
* Level 2: Use of HTTP Verbs and HTTP Response codes
  + Get for fetching, etc.
  + HTTP 4XX for errors (not HTTP 200 with other error codes)
* Level 3: Hypermedia controls (HATEOAS)
  + Calls to APIs return with indication on what you can do next and URI of the resources you need to manipulate

## Open Data (OData)

* About
  + An open protocol to allow the creation and consumption of queryable and interoperable RESTful APIs in a simple and standard way
  + An OASIS standard that defines the best practice for building and consuming RESTful APIs
  + OData helps you focus on your business logic while building RESTful APIs without having to worry about the approaches to define request and response headers, status codes, HTTP methods, URL conventions, media types, payload formats and query options etc.
  + OData also guides you about tracking changes, defining functions/actions for reusable procedures and sending asynchronous/batch requests etc.
  + Additionally, OData provides facility for extension to fulfil any custom needs of your RESTful APIs
  + The OData metadata provides a machine-readable description of the data model of the APIs
* Industry Backing
  + Microsoft
  + IBM
* Resources
  + <http://www.odata.org/>

Open Data (<http://theodi.org> )

Open data is data that anyone can access, use and share. Open data has to have a license that says it is open data. Without a license, the data can’t be reused. The license might also say:

* that people who use the data must credit whoever is publishing it (this is called attribution)
* that people who mix the data with other data have to also release the results as open data (this is called share-alike)

**Good open data**

* can be linked to, so that it can be easily shared and talked about
* is available in a standard, structured format, so that it can be easily processed
* has guaranteed availability and consistency over time, so that others can rely on it
* is traceable, through any processing, right back to where it originates, so others can work out whether to trust it

## JSON API

* About
  + A specification for building APIs in JSON – how a client should request that resources be fetched or modified and how servers should respond
  + Designed to minimize the number of requests and the amount of data transmitted between clients and servers, without compromising readability, flexibility or discoverability (smart client that knows how to cache documents)
  + Requires use for the JSON API media type (application/vnd.api+json) for exchanging data
* Industry Backing
* Resources
  + <http://jsonapi.org>

**Security Standards**

## OpenID Connect (OIDC)

* About
  + OpenID Connect is the third generation of OpenID and not really related to earlier versions
  + OpenID Connect was built on top of OAuth 2.0
  + OpenID Connect provides a lightweight framework for identity interactions in a RESTful manner
  + OpenID Connect is a modern federation specification. OpenID Connect is a replacement for SAML and WS-Federation.
  + It simply introduces an identity layer on top of OAuth 2.0. This identity layer is abstracted into an ID token. An OAuth Authorization Server that supports OpenID Connect can return an ID token along with the access token itself.
  + ID tokens allow the client to know, among other things:
    - How the user authenticated (i.e., what type of credential was used)
    - When the user authenticated
    - Various properties about the authenticated user (e.g., first name, last name, shoe size, etc.)
  + If your client needs data about the user, (for customizations), give it an ID token
  + User-Info Endpoint = The endpoint which provides user information to the client as authorized by the user to the AS
* Origin
  + OpenID Foundation
  + Formalized in February 2014
* Industry Backing
* Resources

## Event Driven API Architecture

<http://nordicapis.com/5-protocols-for-event-driven-api-architectures/>

## 5 Types of Event-Driven Protocols for APIs

### 1: WebSockets

**WebSockets** are an interesting event-driven solution, because, for most browsers, they’re actually baked into the application itself. Essentially, WebSocket is a protocol that provides full-duplex communication on a single TCP connection. It was standardized by the Internet Engineering Task Force as [RFC 6455](https://tools.ietf.org/html/rfc6455), and the WebSocket API in Web IDL was later standardized under the W3C banner.

While the protocol itself is meant to be used between **web browsers** and **servers**, the protocol can be used in any case where there is a client-server relationship. The protocol itself is based upon TCP, with the additional HTTP interpretation statement that is considered an “Upgrade request” to allow for interoperability.

#### Pros

Because WebSocket is expressly designed for browser operation, it boasts extremely **low overhead** for what it actually does. By establishing a **full-duplex** conversation using a standardized methodology, connection both to and from the two entities can take place simultaneously, resulting in lower overhead and better throughput.

Additionally, the fact that these communications take place over TCP 80/443 means that environments that traditionally block non-web based applications for security reasons can still handle this protocol, as firewalls allow communication to and from this port.

Perhaps the strongest argument for the use of WebSockets are the fact that they are standardized and **natively supported** by all major browsers, ranging from Microsft Edge to Opera, from Firefox to Chrome. This means that any web application that ties into it will be interactable within the vast majority of both browser-based and browser-independent gateways and applications.

#### Cons

WebSockets have one distinct major failing — while it might have support for HTTP-like functionality, **it is not HTTP** whatsoever. This has implications, especially when considering optimizing in HTTP such as caching, proxying, etc., that haven’t quite become apparent.

Because WebSockets are relatively new, having been only officially standardized in 2011, the industry is still understanding what the side effects mean. Most applications that use WebSockets are designed specifically for everything that a WebSocket is — what has yet to be seen, however, is whether or not this solution is better in the long-run than any stateless solution currently available.

There is of course the fact that, as with other architectures on this list, WebSockets create an “always on” connection during the duration of data transfer. While this is fine for many uses such as media streaming and live stream calculations, it also essentially means that, for WebSockets, there is **no scalability**. Ports have hardcoded limitations and bandwidth, and thus in order to “scale”, you must add additional ports to match the maximum load. In stateless systems, this is less of an issue, as requests can wait and be made in such a way as to be independent on the state of the server itself.

### 2: WebHooks

**WebHooks** are a similar concept to the WebSocket. They primarily function using **custom callbacks**, or code that is passed as an argument to another chunk of code and executed at a specified point in time. Essentially, a WebHook is a glorified system of “if this, then do”, allowing for users independent of the event firing to craft a custom response to that event within their own system.

The term was coined by [Jeff Lindsay](https://twitter.com/progrium) in 2007, and quickly became popular amongst users who wished to create automated responses to exterior behaviors. A great example of this would be a developer pushing a new item to GitHub, which causes an event. A user has a system tied into the URI of a WebHook. When the push is published, the user’s system utilizes the URI of the WebHook to integrate the push into a larger build, thereby creating a compiled component.

#### Pros

WebHooks function a lot like WebSockets, but they’re different in some key areas. First and foremost, WebSockets are primarily designed for browser-based communications, and while they can be used regardless in any client-server communication, they do not behave well in a server-to-server setup.

WebHooks, on the other hand, work very well in **server-to-server** systems due to how they operate. Because the system essentially functions as the aforementioned “if this then do”, servers can be configured to tie into pre-formed URIs at any time and execute a given function whenever that event is triggered.

Additionally, WebHooks have the unique benefit of being **based upon HTTP**, unlike WebSockets. This means that the system can be integrated without utilizing any new infrastructure, allowing speedy adoption and relatively simple setup.

#### Cons

The problem with WebHooks is that a lot of their functionality can already be placed on the arguably more powerful REST architectural approach. While adopting event-driven architecture is often a requirement of the service being built, it’s a hard sell when it can be mirrored in REST while also giving the wealth of options that REST gives to the user.

These RESTful solutions such as [RestMS](http://www.restms.org/) are essentially simply message querying services, though, and do require additional infrastructure, which may or may not be doable considering the purpose of the application.

Additionally, WebHooks can be **resource intensive** to both the client and the server. If the client needs to notify many servers that an event has occurred, and a server needs to listen to a great deal of clients notifying of this change, you can very quickly run into a situation where your network grows uncontrollably. While HTTP does scale quite well, this is a definite negative to consider.

However, there are also ways to build a [message queuing service](https://en.wikipedia.org/wiki/Message_queuing_service) on top of HTTP—some RESTful examples include [IronMQ](https://www.iron.io/platform/ironmq/) and RestMS.

### 3: REST Hooks

Speaking of RESTful examples, **REST Hooks** is essentially “hooking” baked into REST itself. Defined as an initiative from [Zapier](http://resthooks.org/docs/), this is a subject [we’ve covered before](http://nordicapis.com/stop-polling-and-consider-using-rest-hooks/) — hooks are collated to a single target URL as a subscription, which pings the resource requester when a change is noted.

This approach is a response to the practice of **polling**, in which a client constantly checks for changes to a resource. Under the REST Hooks paradigm, the client instead waits for a change, and reacts to it. To put it simply, this is a WebHook in REST.

#### Pros

REST Hooks are obviously super powerful in the correct context — being able to passively receive a resource rather than dedicating processing power to constant polling frees up a lot of the client-side cost.

Perhaps the strongest argument for REST Hooks though, is the fact that it’s so **easy** and **intuitive** to use. While WebHooks utilize HTTP and thus do not need new architecture to set up, they are also limited by the fact that they are built upon HTTP, and can thus be somewhat complex to set up properly and use effectively.

REST Hooks, though, are **subscription based**, and as such, are simply usable by subscribing. This makes it a very easy to use solution while providing a lot of the usability and effectiveness of more complex systems.

#### Cons

Of course, every solution has its negatives, and REST Hooks are no different. It could be viewed that REST Hooks actually fly in the face of what REST is — [session free and stateless](http://nordicapis.com/defining-stateful-vs-stateless-web-services/). REST Hooks essentially create consistent polling, it’s just moved the polling from one side to another.

Then, there’s the arguable problem that REST Hooks might be doing something that has already been solved. Some would argue that TCP already does most of what REST Hooks is trying to do, and simply layering more solutions on top of HTTP to get what TCP already does is a poor approach.

### 4: Pub-Sub

**Pub-Sub** is a slightly different approach. Referred to by its full name as **publish-subscribe**, the concept is where events are published to a class without knowledge of the client subscribing to the class. Basically, a user will join one or more classes, and then will receive event updates without regard or knowledge to the event publisher.

The main difference here is one of conscious choice of provider — in the other solutions noted herein, a user consciously communicates with a given server or provider and receives events as pre-determined. Under the Pub-Sub scheme, the user only specifies which class they wish to be part of and what events they are interested in receiving. From there, they receive these events when one is pushed out.

A way this is often framed in internet discussions is in the frame of a **radio channel**. Record companies, or publishers, issue audio to the station, which then broadcasts this audio to listeners, or subscribers. Pub-sub is the middleman radio station here — listeners don’t know who gave the station the music, nor do the companies know who the listeners are. It is this segmentation that is baked into the pattern.

When we talk about Pub-Sub, we need to keep in mind that we’re actually talking about two different things. Pub-Sub can mean the methodology and general concept in programming terms, but it can also mean specific provider solutions based upon that methodology. For instance, Google’s [Cloud Pub/Sub](https://cloud.google.com/pubsub/docs/overview) is an implementation of the general methodology within their cloud service, and allows for asynchronous many-to-many pub-sub relationships as stated above.

#### Pros

A huge benefit of Pub-Sub is the fact that it’s **loosely coupled**, and thus is extremely **scalable** and flexible. The only thing the event-provider is doing is generating the content — each other step is done through a separated middleman, and so the content is easily scaled and modulated to the architecture and design of the solution.

Additionally, Pub-Sub lends itself very well to **testing**. A subscriber is narrowly limited to a set of events that they have requested under a class, so if a failure occurs, this natural segmentation informs the provider as to where the fault is, and which class of users is experiencing the fault.

#### Cons

Unfortunately, decoupling is also a huge disadvantage for this pattern. By being a **middleman**, Pub-Sub cannot effectively notify the provider that a message has been sent, and the listener is separated from the event and thus may not know if a message wasn’t sent that should have been. Harkening back to the radio explanation, a listener will never know if a song was meant to play on a channel or if the channel is out of range, and once the record executives hand off the music, they’ve got no idea if the user received the songs without direct feedback from them.

Additionally, while the system is extensible and flexible, instability does occur with high traffic load as subclass after subclass might be constructed to handle further segmentation. This of course leads to the aforementioned instability in addition to increased complexity.

You must keep in mind that while the relationship between the publisher and subscriber in this model may be beneficial, it also comes with its own difficulties when these relationships need to be modulated. While you can certainly work around this, at this point, you’re fighting the very basis of the pattern, rather than any secondary natures — you’re trying to make dehydrated water, and fighting against the nature of a pattern suggests the pattern to be inherently poor.

### 5: Server Sent Events

**Server Sent Events**, or SSE, is a communication protocol much like WebSockets, but with the implication of **unidirectional data**. In this architecture, the server is consistently sending updates to the client as an automatic process. This was standardized under HTML5 by the [W3C](https://www.w3.org/TR/2009/WD-eventsource-20091029/), and is thus compatible with any solution that is likewise compatible with HTML5.

Of note is that there is a competing standardization from the [Web Hypertext Application Technology Working Group](https://whatwg.org/) – this is a relic from movement away from “HTML5” and into what WHATWG is calling “HTML Living Standard”. The general working consensus is that, WHATWG’s standardization is prioritized in the rare cases of divergent standards. This could become more of an issue as time marches forward, given that WHATWG was created due to a perceived lack of interest from W3C towards evolving HTML, but for the time being, either standard is generally acceptable.

While simple in theory, Server Sent Events are anything but simple when considering benefits and drawbacks.

#### Pros

SSE is **not bidirectional** in its communications — the server is issuing the events in a steady, predictable method. This is hugely beneficial to applications which do not need the two-way communications baked into WebSockets or other such solutions, as this means **lower bandwidth**, and an allowance for the connection to be temporary rather than always-on during the duration of data transfer. By its nature, the data is being transferred one way, and thus there is no need to wait for data to be returned.

Additionally, at least in theory, SSE is easier to set up in complex situations. You only have to worry about data traveling one direction via one system, thus reducing complexity dramatically. There is no need to define a message exchange protocol, no need to specify data duration or wait times, no need to support bilateral messaging — the single direction saves a lot of complexity.

#### Cons

That simplicity could be where SSE fails for particular use cases. SSE is a very poor solution for situations that require **bidirectional communication**, and while this seems obvious, it would surprise many developers to see how many systems actually depend on bidirectional communication for simple functionality.

While much of this can be fixed with workarounds, a developer’s goal in choosing an event-driven protocol should be to find one that works out of the box, not to find a solution that might work if configured properly and given secondary systems upon which to depend.

There is also the issue of **security** and authentication. While two-way systems can easily use authentication methodologies, SSE handles this using **header forwarding**. While headers can be manipulated and overridden in many languages and applications, the EventSource object in JavaScript does not natively support this, which would cause many adoptees some major headaches.

Finally, there is a concern over loss of efficiency with **over transmitting** data. A two-direction system can determine when a client or server disconnects, but SSE can only determine that a client has disconnected after attempting a full data transmission and receiving a noted failure. Because of this, data can be lost rather quickly, and with many failed connections, this loss can mount dramatically over time.

If you are building for scalability with low overhead in a browser environment, **WebSockets** are a great solution. Conversely, if you’d like those same benefits but are working in a non-browser system, then **WebHooks** should be your approach. **REST Hooks** are not only great for RESTful services, they’re also much easier to set up than either, and thus are great in low-time high-rush situations. **Pub-Sub** can be great if you need to enforce a division between client and server, and this can further be established and controlled in an even stronger way with **Server Sent**.

## TLDR Comparison Table

| **Protocol** | **Related to** | **Standard Body** | **Notes** |
| --- | --- | --- | --- |
| WebSockets | TCP, HTTP-like | IETF, W3C | * Unidirectional communication over TCP * Designed for web browsers & web servers * Good for lower overhead scenarios * Supported in all major browsers |
| Webhooks | URI, HTTP | – | * User defined “HTTP callbacks” * Triggered by an event HTTP * Requests are made to Webhook URI * Enables real-time event triggering |
| REST Hooks | HTTP | Zapier | * Lightweight subscription layer * Manipulated by a REST API * Essentially a WebHook in REST |
| Pub-Sub | – | – | * Client subscribes to classes * Bidirectional * Middleman layer between client and server * Loose coupling |
| Server Sent | HTTP, HTML5 , DOM | WHATWG, W3C | * Server constantly sends updates to the client * Unidirectional push notifications as DOM events |

API Tools

<http://blog.mashape.com/10-api-tools-released-this-year-2015-roundup/>

## JSONP

<http://stackoverflow.com/questions/3839966/can-anyone-explain-what-jsonp-is-in-layman-terms>

<https://www.sitepoint.com/jsonp-examples/>