



Robert Broeckelmann · Follow

9 min read · Feb 16, 2016

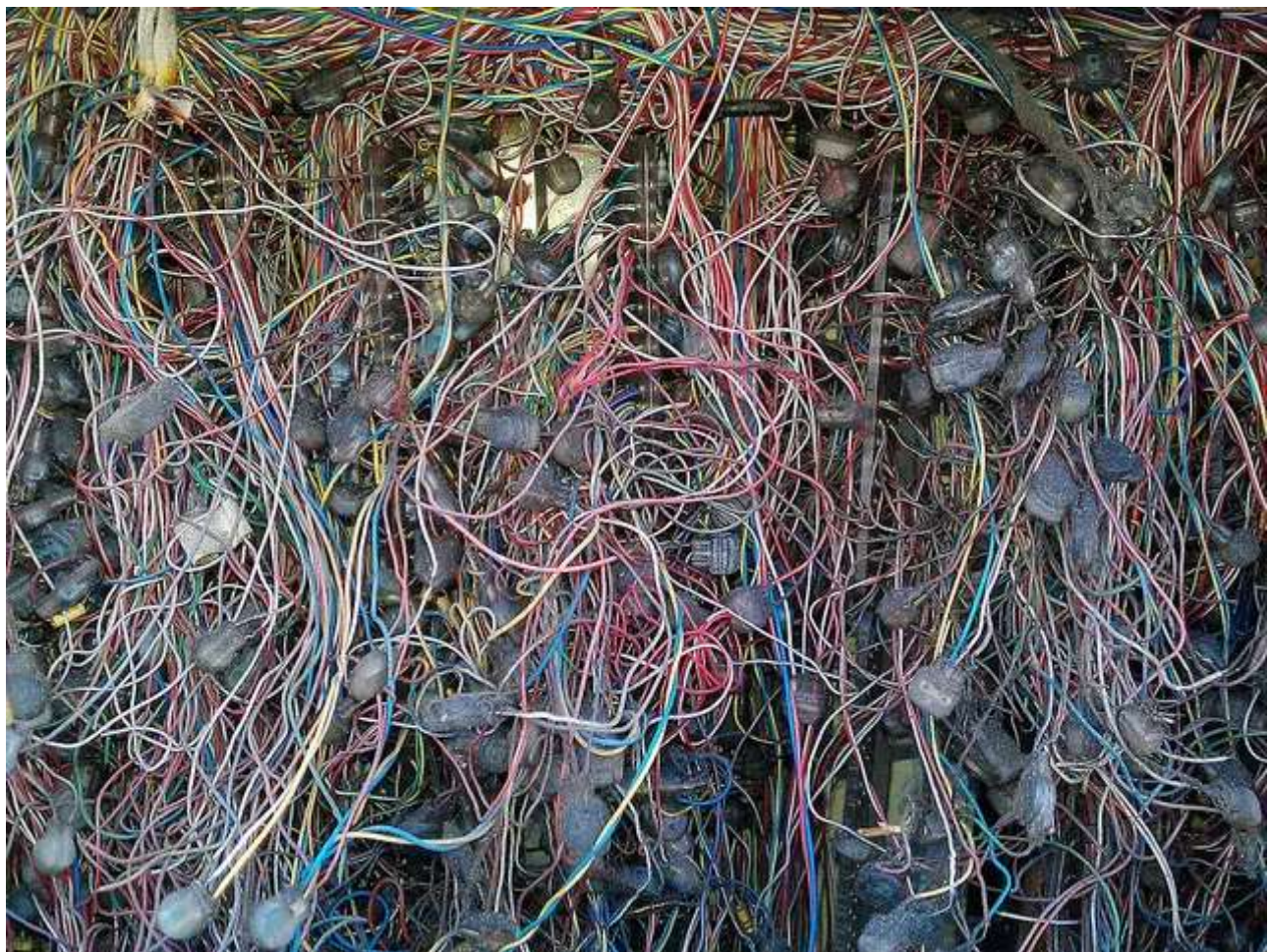


28



What are APIs? (The Technology Perspective)

This post was originally published as “What are APIs? (The Technology Perspective)” on the Levvel Blog.



The term, API (Application Programming Interface), was originally used to describe any programming interface for a library or module that could be

composed into larger software systems. In the last few years, it has commonly been used to refer to an architectural style of client-server communication that is meant to be easy to use, relies upon the mechanics of the HTTP protocol, and is stateless.

This architectural paradigm has been characterized by a lack of official-standards until recently; largely due to APIs evolving from a desire to move away from the complexity that was introduced by SOAP and the WS-* (pronounced WS-splat) specs. APIs evolved from REST (REpresentative State Transfer). First described in Roy Fielding's Doctoral Thesis at the University of California, Irvine in 2000, by 2009, REST was becoming a popular alternative to SOAP Web Services, especially for mobile applications. By 2012, the conversation had moved from REST to APIs; though, the key concepts had been evolving for sometime.

The proponents of APIs and REST are quick to call out the simplification and improvements over the predecessor paradigm (SOAP). The industry tends to have a short-term memory. Tongue placed firmly in cheek, I often like to sum up this hype-cycle as:

- APIs formalized REST.
- REST fixed the problems in SOAP
- SOAP fixed the problems in EJBs
- EJBs fixed the problems in CORBA
- CORBA improved DCOM
- DCOM was supposed to be better than RPCs

We also had Tuxedo Services in there somewhere as well (my first job out of college).

Anyone noticing a pattern? Each of these technologies started out as a good idea that were meant to address the short-comings of its predecessor. Each technologies had its club who truly believed it would solve the world's (well, most of them probably just thought the industry's) problems. From my perspective, every one of these technologies did start out as a good idea. The inventors of each set out with the best of intentions (mostly). But, the demands of enterprise IT, across the entire industry, slowly layer more-and-more concepts on top of the original idea. Eventually, that good idea ends up as a hodge-podge of complex standards that cater to various niche corners of the industry.

The current paradigm is definitely APIs; so, let's go with it. We'll start by looking at REST.

From Roy Fielding's Thesis, REST is an architectural style that imposes the following architectural properties:

- Performance
- Scalability
- Simplicity
- Modifiability
- Visibility
- Portability
- Reliability

and the following constraints:

- Client-Server Architecture
- Stateless
- Cache (i.e., responses are cacheable given the right circumstances)
- Uniform Interface
- Layered System
- Code-on-Demand(Optional)

Now, just about everyone wants performance, scalability, simplicity, modifiability, visibility, portability, and reliability in their systems — unless they have some other goal that doesn't include delivery of a robust system(it happens).

Performance can be measured from the stand point of the end user experience, network utilization/latency, server-side system resource utilization(CPU, I/O, memory,etc), and others. Performance tuning all of these aspects is its own discipline. While any system should be built on a set of core best practices derived from the experiences of its designers, it his hard to predict exactly where the performance bottlenecks will appear in a complex system. Get the system working first, define the performance characteristics that are needed, then begin the performance tuning process.

Scalability is the ability of a system or architecture to handle more resources to improve performance or increase the number of users of a system. For example, adding more database servers, application servers, web servers, more VMs, etc.

Simplicity means don't over-engineer it. Make things as simple as possible, but no simpler. Design for generalization(which enables reusability) and separation of concerns. Centuries from now, when technology archeologists are studying the design of turn-of-the-millennium software technology, application of Occam's Razor should be able to explain to our descendant's why your code is the way that it is.

Modifiability is a measure of how easy it is to make changes to the system, test those changes, and understand ahead of time the impact that change will have on the system. This brings together having a well-defined interface published to clients that will not change unless a new version is published and test-driven development methodologies. After the change has been deployed, running through a standard set of automated tests during a build process should give a high-degree of confidence that the change is having the desired effect without any undesired side effects.

If there is Visibility between two components (client and server or two server components, for example), then it is possible to have a third component monitor the interaction, enforce security policy, or provide another service such as caching.

The Portability property reflects the ability to run software and its data in multiple environments. For example, the ability to run the same code base and data on two different cloud providers.

Reliability is fault tolerance. The ability of the system to maintain availability even when things go wrong. Rest assured, things will eventually go wrong in even the most stable of systems. Avoid single points of failure; have redundancy at each level for each components. A single failed component should not bring down the entire system.

Client Server Architecture has been around for a while. In its simplest form, there are two actors: a client and a server. This introduces a basic separation of concerns. The client handles the user interface; the server implements the functionality. Neither is particularly concerned with the “how” of the other.

The Stateless constraint means that the server does not maintain any form state or session for the client. This implies that every request from the client to the server must maintain all information needed to process the request. An interesting example of this that we will see later is JSON Web Tokens(JWT) that are used with OpenID Connect; JWT tokens are a type of bearer token that can contain key:value pairs of attributes that are used to satisfy Role-Based Access Control(RBAC) or Attribute-Based Access Control(ABAC) security models and easily allows an API request to contain all information required to evaluate authentication and authorization decisions. This is a departure from paradigms established in JEE and the Servlet spec where session and state information can be tracked for each client on the server using out-of-the-box functionality.

The Cache constraint says that for situations where it makes sense, a response can be cached either by the client or some type of intermediary. The conditions under which a response can be cached is highly application/situation/domain dependent. A deep understanding of how an application uses data and how that data changes over time is required before using response caching. Both WebSphere DataPower and Apigee Edge offer this functionality out-of-the-box. However, tuning each to correctly cache a response with all dependent variables accounted for (HTTP method, path, authenticated user, query parameters, request headers, request message body fields, response message body fields, response content type) can become very challenging.

The Uniform Interfaces constraint defines the interfaces between clients and servers. There are four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state (referred to as HATEOAS). Identification of Resources is done through URIs (or paths); this is a fundamental building block of a REST or API interface. When a client has a representation of a resource, the information contained in that representation should be sufficient to manipulate (update or delete) that resource on the server. The self-descriptive messages interface constraint means that REST request and response messages should be self-contained. There shouldn't be additional information needed to parse and process the request on the server — this builds on the Stateless constraint described previously. Hypermedia is the use of hyperlinks within hypertext; the HATEOAS interface constraint dictates that hypermedia is used to deliver state between client and server (and vice versa). This means that a REST client uses query parameters, request headers, message body, path, and method to deliver state to the server. Likewise, the server uses a message body, response headers, and response code to return state to the client.

The Layered System constraint basically means that intermediate layers can be placed between the client and server or multiple layers of server components without impacting the functionality of either. This is important because it allows for the introduction of something like an API Gateway — we'll get to that later.

The Code-on-Demand constraint is ability to load code on the client from the server. This is commonly done with Javascript being loaded into a User Agent (such as a browser) for Web Applications, but this constraint isn't really something that I haven't seen used with REST or APIs in the real world; so, I'm not going to discuss that further here.

Moving beyond Fielding's definition of REST, other aspects of our API concept have evolved.

Security for REST has included:

- Basic Authentication
- TLS 1.2(with client authentication required client authentication)
- API Keys(there isn't a spec around this concept that I am aware of)
- OAuth 1.0
- OAuth 2.0(this is a whole family of specifications/RFCs)
- OpenIDConnect(builds on top of OAuth 2.0 and other specs)

Security concerns were outside the Fielding dissertation. And, while a fascinating topic, that, too, for a future post. Going forward, I predict that OpenID Connect will be the API Security Standard that is embraced by the industry.

The use of JSON (Javascript Object Notation) as the de-facto standard data format for REST and APIs evolved later. That, also, wasn't part of the original REST description. JSON earned this distinction in part because its predecessor and earlier defacto-standard, XML, is rather verbose, has many data types, and can be complex to generate and parse. JSON, in contrast is

- simple to parse
- fast to parse
- less verbose(smaller payload/data structure representation)

- has only a small number of types (in comparison to XML anyway) that are supported by most languages in use today
- is popular with developers

The word “lightweight” usually comes up in this discussion. I don’t like that description of anything without a larger context of what is being measured. Size? Number of data types? What if you need the missing data types? Then, maybe the feature set of XML is appropriate. So, you will not see me write the word lightweight in regards to APIs unless there is a criteria and definition included.

Bottom-line, JSON is popular and it has emerged as the de-facto data format standard for APIs. That being said, most APIs I have created or worked with could produce XML and JSON — especially in the enterprise(ie, large organizations)

The final piece of an API (or any interface really) is a mechanism for describing the interface. In the API world, there are several competing standards. I prefer Swagger. Swagger 2.0 is supported by Apigee and IBM. Numerous other vendors have support for it as well. A comparison of alternatives is another blog post. But, I am biased towards Swagger.

So, what is an API? An API is a well-defined, system interface, usually public-facing, that meets the REST criteria laid out in the Fielding Dissertation and generally uses JSON as the input and output data formats (but, could be XML). The term API is also commonly used to refer to the implementation of that interface.

Unlike SOAP and the WS-* world, the API & REST world is far less formal. But, as with all good (or simple) ideas in the IT industry, things mature. That maturing leads to formal specifications and standards — it is inevitable, stop resisting.

For APIs, the following specs exist or are in draft now:

- [JSON Spec](#)
- [JSON Schema Spec draft 4](#)
- [OpenID Connect v1.0](#)
- [Swagger v2.0](#)
- [HTTP 1.1](#)
- [TLS 1.2](#)

Some of my peers will view this type of structure being imposed on the API space as hindering whatever it is they do day-to-day. My primary focus has been enterprise IT. What I am very interested in is how APIs and API Management can be adopted by and help the enterprise (think fortune 100 companies). This type of maturity is needed of anything that one of these organization's is going to adopt. This is a very different situation from what a silicon valley startup or other small organization probably needs — for the most part, these shops are not my focus.

It is interesting to note that the JSON spec is separate from the Javascript language spec. These specifications actually differ a little in what they allow.

The JSON Schema Spec defines a schema language for JSON data structures similar to what XSD does for XML.

Swagger 2.0 uses JSON Schema to define what it calls JSON models.

What would you add to this definition? Leave it in the comments.

Rest

APIs

Definition

**Written by Robert Broeckelmann**

Follow

1.99K Followers · 1 Following

My focus within Information Technology is API Management, Integration, and Identity—especially where these three intersect.

No responses yet



What are your thoughts?


Respond

More from Robert Broeckelmann

120	4.681321	172.31.40.187	172.31.41.127	TCP	54 53143 → 88 [ACK] Seq=1 Acc=1 Win=573640 Len=0
121	4.681340	172.31.40.187	172.31.41.127	KRB5	246 AS-REQ
122	4.681759	172.31.41.127	172.31.40.187	KRB5	222 KRB Error: KRB5KDC_ERR_PREAUTH_REQUIRED
123	4.681808	172.31.40.187	172.31.41.127	TCP	54 53143 → 88 [FIN, ACK] Seq=213 Acc=100 Win=573640 Len=0
124	4.682075	172.31.41.127	172.31.40.187	TCP	54 88 → 53143 [ACK] Seq=149 Acc=214 Win=573640 Len=0
125	4.682076	172.31.41.127	172.31.40.187	TCP	54 88 → 53143 [RST, ACK] Seq=149 Acc=214 Win=0 Len=0
126	4.689403	172.31.40.187	172.31.41.127	TCP	66 53144 → 88 [FIN, ECH, CWR] Seq=0 Win=8192 Len=0 RJS=8961 WS=256 SACK_PERM=1
127	4.689440	172.31.41.127	172.31.40.187	TCP	66 88 → 53144 [FIN, ACK, ECH] Seq=0 Acc=1 Win=8192 Len=0 RJS=8961 WS=256 SACK_PERM=1
128	4.689781	172.31.40.187	172.31.41.127	TCP	54 53144 → 88 [ACK] Seq=1 Acc=1 Win=573640 Len=0
129	4.689936	172.31.40.187	172.31.41.127	KRB5	346 AS-REQ
130	4.690334	172.31.41.127	172.31.40.187	KRB5	1501 AS-REP
131	4.690771	172.31.40.187	172.31.41.127	TCP	54 53144 → 88 [FIN, ACK] Seq=220 Acc=1508 Win=571008 Len=0
132	4.690806	172.31.41.127	172.31.40.187	TCP	54 88 → 53144 [ACK] Seq=1508 Acc=208 Win=573640 Len=0
133	4.690893	172.31.41.127	172.31.40.187	TCP	54 88 → 53144 [RST, ACK] Seq=1508 Acc=208 Win=0 Len=0
134	4.691056	172.31.40.187	172.31.41.127	TCP	66 53145 → 88 [FIN, ECH, CWR] Seq=0 Win=8192 Len=0 RJS=8961 WS=256 SACK_PERM=1
135	4.691875	172.31.41.127	172.31.40.187	TCP	66 88 → 53145 [FIN, ACK, ECH] Seq=0 Acc=1 Win=8192 Len=0 RJS=8961 WS=256 SACK_PERM=1
136	4.691379	172.31.40.187	172.31.41.127	TCP	54 53145 → 88 [ACK] Seq=1 Acc=1 Win=573640 Len=0
137	4.691615	172.31.40.187	172.31.41.127	KRB5	1486 TGS-REQ
138	4.691993	172.31.41.127	172.31.40.187	KRB5	1557 TGS-REP
139	4.692943	172.31.40.187	172.31.41.127	TCP	54 53145 → 88 [FIN, ACK] Seq=1433 Acc=1508 Win=571008 Len=0
140	4.693118	172.31.41.127	172.31.40.187	TCP	54 88 → 53145 [ACK] Seq=1508 Acc=1438 Win=573640 Len=0
141	4.693397	172.31.41.127	172.31.40.187	TCP	54 88 → 53145 [RST, ACK] Seq=1508 Acc=1438 Win=0 Len=0

```

as-rep
  prev: 5
  msg-type: krb-as-rep (11)
  padat: 1 item
    pa-data: PA-ETYPE-INFO2
      padat-type: KRB5-PADATA-ETYPE-INFO2 (19)
        padat-value: 30172015a0000012a10e1b0c50434242e0e55472602...
  
```

 Robert Broeckelmann

Kerberos Wireshark Captures: A Windows Login Example


This blog post is the next in my Kerberos and Windows Security series. It describes the...

May 17, 2018

 254

 4



 Robert Broeckelmann

OpenID Connect Logout

The OpenID Connect (OIDC) family of specs supports logout (from a single application)...


Jul 12, 2017

 490

 5



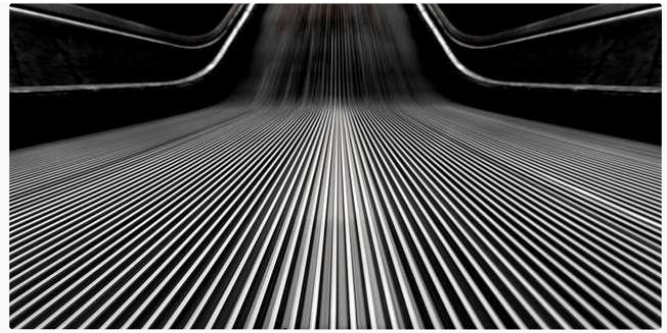



 Robert Broeckelmann

HTTP POST vs GET: Is One More Secure For Use In REST APIs?

The use of HTTP POST vs HTTP GET for read-only (or query) operations in REST APIs...

Feb 6, 2021  78



 Robert Broeckelmann

Authentication vs. Federation vs. SSO

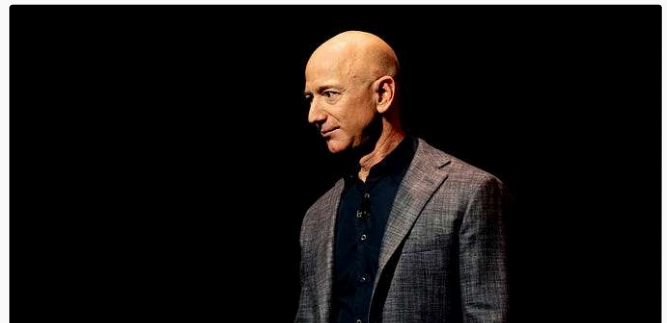
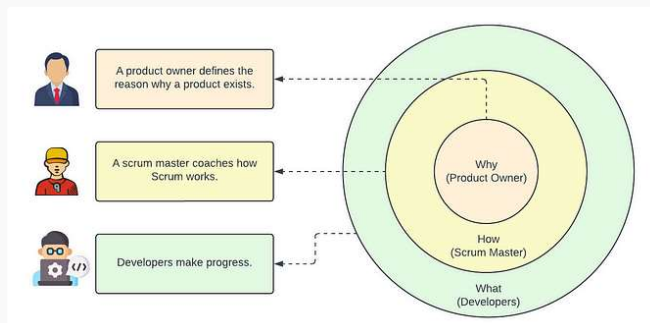
Authentication. Federation. Single Sign On (SSO). I've mentioned these concepts many...

Sep 24, 2017  859  6



See all from Robert Broeckelmann

Recommended from Medium





In Beyond Agile Leadership by Eiki Takeuchi



Jessica Stillman

Why Hiring High-Performance Developers is the Biggest Mistake...

When I was working as a software engineer in Japan, I saw numerous smart and talented...



Nov 11



368



32



Oct 30



13.7K



322

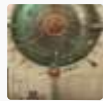


Lists



Coding & Development

11 stories · 926 saves



data science and AI

40 stories · 296 saves



Company Offsite Reading List

8 stories · 170 saves



Natural Language Processing

1841 stories · 1463 saves



In Stackademic by Crafting-Code

I Stopped Using Kubernetes. Our DevOps Team Is Happier Than Ever

Why Letting Go of Kubernetes Worked for Us



Nov 19



3.5K



113



In DataDrivenInvestor by Austin Starks

I used OpenAI's o1 model to develop a trading strategy. It is...

It literally took one try. I was shocked.



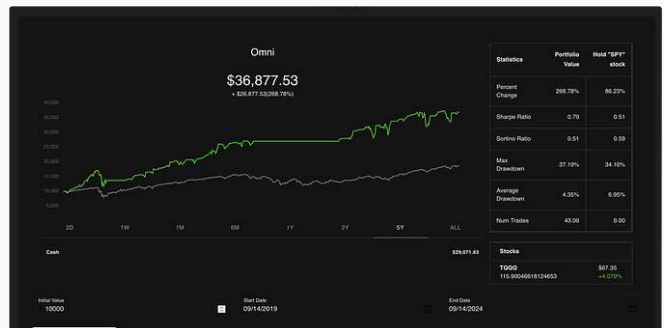
Sep 16




6.8K



174





 Adaobi Adibe

How to do things if you're not that smart and don't have any talent

This is a blog post aimed at people who want to do important work or make meaningful...

6d ago  4.8K  137



 Mark Shrike, MD, PhD 

The dumbest decision I ever made (and the Nobel Prize that explains...

Decision science, Family Guy, and why the "safe" choice is often the riskiest.

Nov 21  7.4K  167



See more recommendations