Infrastructure and tools to support secure, scalable, and highly available APIs

Agustin Schapira
Comcast

Abstract

This paper describes a layered HTTP infrastructure designed to support web-based APIs in a standard, secure, scalable, and highly available fashion. The efforts described here are based on the observation that large scale distributed systems exhibit greater robustness, flexibility, and extensibility when they are conceived, built, and operated as a set of small independent but interconnected components. Leveraging the power of DNS routing and the HTTP protocol, we have built a platform that makes it easy for engineering teams within the organization to expose their services to other teams as HTTP APIs, and in turn to build their solutions based on other teams' APIs. The adoption of the API platform has reduced duplicate efforts, increased the overall security of our systems, provided greater control and visibility of how components are being used, and ultimately helped us innovate more quickly.

INTRODUCTION

It is by now a widely accepted tenet of web-based application development that de-coupling application logic and core data services from presentation layers enables the independent development of the two pieces and therefore reduces development cycles, increases the maintainability of the code, and allows for the quick migration to new platforms and devices (Burbeck, 1987). More general software engineering principles, such as decomposition (Parnas, 1972) and separation of concerns (Dijkstra, 1982), enhance overall architectural robustness, reduce the duplication of efforts, facilitate systems integration, and encourage the sense of ownership and responsibility over an individual component. The benefits of these practices go beyond pure engineering and into the business realm, as the same mechanisms that encourage the sharing of solutions across engineering teams and subsidiaries may also enable partners to further extend a company's core offerings through syndication arrangements, encouraging development of new, unforeseen solutions based on the existing, highly focused components (Benslimane, Dustdar, & Sheth, 2008). Ultimately, these strategies enable companies to innovate quickly in a technological landscape constantly in flux (Papazoglou & Georgakopoulos, 2003).

In the world of Internet applications and services, reusable solutions are most commonly implemented as components that expose an Application Programming Interface, or API, which other components (internal or external) may invoke in order to extend their own functionality. An API is a well-defined contract that specifies how a consumer should exchange information with a service provider in order to access and use its services (Papazoglou & Georgakopoulos, 2003). In the case of the web, the particular set of rules and data formats specified by the API usually rely on general communications and architectural patterns, such as the SOAP (W3C, 2007) and REST (Fielding, 2000) styles, to specify how the actual messages should be exchanged and interpreted (e.g., how to represent the space of possible actions, how to encode the messages, how to report errors, etc.). Ultimately, all of these architectural styles use HTTP as the underlying communications protocol.

This paper describes a platform to encourage and support, within an organization, the creation and use of web APIs. The starting point for this work was the question: "*What are best ways to encourage the creation, operation, documentation, maintenance, and use of APIs in a large organization, where tens of engineering groups and hundreds of developers are working on their own problems and solutions?*" In other words, if we believe that this particular engineering practice can improve our agility, robustness, and time-to-market, then how can we encourage its adoption in a large organization with multitudes of product lines, deadlines, engineering practices, and goals?

To guide our work, we established five core principles, and then sought to provide answers and solutions to address them:

1. **Exposing APIs should not impose a big burden on the owner of the service** --and, in fact, it should have obvious benefits. To address this issue, we built shared infrastructure that solves the common difficulties involved in supporting APIs (security, access policies, and scalability), and established standard protocols and procedures for incorporating an API onto the infrastructure. By simply "plugging in", API providers get all those issues solved and are freed to focus on their core competencies.

2. **Leveraging existing APIs should not impose a big burden on developers** --the ultimate goal being that a developer who learns how to use one API will immediately also know how to access all other internal APIs. To address this issue, we established well-known protocols for making calls through the infrastructure onto the APIs, and based them on widely used open standards. Furthermore, we created and distributed libraries for several programming languages to take care of the details of making HTTP requests according to the infrastructure's requirements.

3. **API owners should not lose control of their services** --and, in particular, they should be able to make all the choices afforded by the HTTP protocol, as long as those decisions do not interfere with the core security requirements. To address this issue, we chose a highly distributed operations and control structure. We ensured that the API infrastructure imposes as few demands as possible (and that those demands are based on open standards), placed the operational responsibilities on API owners, and gave them complete freedom on issues such as data formats, URL patterns, cache control directives, etc.

4. **The use of APIs should not introduce unnecessary obscurity** in the underlying communication channels --and, in particular, the API infrastructure should preserve as much as possible the transparency and ease-of-use of HTTP and of the architectural styles built on top of it. To address that issue, we introduced HTTP headers that trace the processing of requests as they travel through the infrastructure, and built tools to test and debug APIs from a web browser.

5. **Delays for getting on board, either as an API provider or a consumer, should be minimal** --because people lose interest very quickly when faced with too many hurdles to test a new technology**.** To address this issue for API providers, we created a sandbox version of the infrastructure --with very few limitations compared with the production environment-- where it's possible to get a new API up and running in less than 30 minutes. For API consumers, on the other end, we built a Developer Portal where they can find an API catalog, read detailed documentation, and even access live versions of APIs directly from their web browser, so they can quickly learn about the inputs and outputs of an API without writing a single line of code.

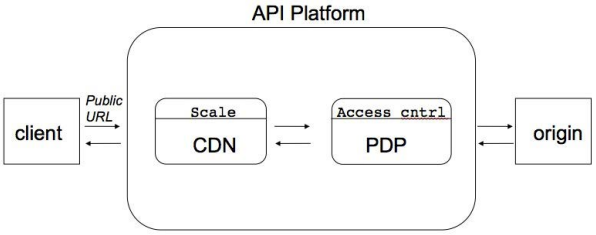
The remainder of the paper describes in more detail the shared API infrastructure that we built, the communications protocols that we imposed on it, and the Developer Portal where we centralize the documentation of existing APIs.

API INFRASTRUCTURE AND ENVIRONMENTS

The most common issues that engineering teams have to face when exposing APIs are security, access control policies, and scalability. In order to address these issues, we built an API infrastructure that enables service providers to simply "plug in their APIs" (following some basic protocols) and effectively delegate those responsibilities to the shared platform.

An API integrated with this infrastructure offers its resources via *public URLs* that bind (via Layer 3 / DNS routing) to the platform infrastructure. Through this level of indirection, an HTTP request from an API client is first routed through the platform so that, by the time it reaches the API's origin servers, most of the scalability and security decisions have been made.

Following the separation of concerns principle, we conceived of the infrastructure as a layered architecture, with a set of HTTP intermediaries tackling the different issues. Internally, the request is routed (through a combination of Layer 3 and Layer 7 mechanisms) through a small set of specialized HTTP intermediaries. Scalability and high-availability are the responsibility of an initial load-bearing layer, implemented through a **Content Distribution Network** (leveraging its very-large-scale edge topology, caching services, and high-availability and failover mechanisms). This outermost layer also acts as a **Policy Agent**, enforcing policy decisions made by a second layer, the **Policy Decision Point**, to which the request is further routed, and which is responsible for authenticating the client, determining whether it has access to the requested resource, applying business policies (such as rate limiting), and logging detailed information about the request for reporting and monitoring. If the request is valid, it is finally routed to the API origin. The response from the origin traverses back downstream though the intermediaries, which may decorate it, and finally back to the client that initiated the request. Figure 1 illustrates this flow.



An important requirement while designing this infrastructure (reflected in the fifth guiding principle) was that developers should be able to "plug" an API and start offering it to others in less than 30 minutes –the underlying assumption being that longer setup times would discourage developers from trying out the platform and would therefore reduce the levels of adoption throughout the organization. This requirement led to the creation of two almost identical but separate environments on top of the platform: a ***Sandbox environment***, with an almost self-service model of API provisioning and where developers can test their APIs in development and QA scenarios, and a ***Production environment***, with SLAs and complete branding for operationally ready APIs --but with a necessarily slower and more involved setup procedure.

The internal structure and functioning of the two environments is almost identical. The major difference resides in the fact that sandbox APIs are exposed with *unbranded* public URLs (e.g., as a sub-domain of a

generic *api_plaform.example.com/*), whereas production APIs get their own branded hostnames (e.g. *voice.example.com/*). This is due to the fact that the procedures for establishing new DNS domains and acquiring SSL certificates for them are much more involved than simply creating a sub-domain for an existing domain (and, in particular, require lengthier setup procedures with the load-bearing CDN partner). But beyond that distinction, there are no other differences. This combination of environments allows API owners to get started very quickly (simply by filling out a form with data about the desired edge URL and the location of the origin server and waiting a few minutes for operations staff to initialize an endpoint and turn the API on), while at the same time retaining the ability to easily migrate to a production environment when the APIs and their owners are ready.

The following sub-sections detail different communications protocols imposed on top of the layered architecture in order to guarantee security and offer scalability. Since the differences between the two environments are minimal and have already been detailed above, no further distinctions between them will be made.

Internal Security

The API infrastructure allows API origins to delegate decisions of security. This means that when an API origin receives an incoming request, it should be free to assume that the security and access control policies have already been enforced, and that it may therefore process the request without having to concern itself with those issues. Because API resources are offered via public URLs accessible through the open Internet, however, the API origin must ensure that the request is indeed arriving from a component in the shared infrastructure --otherwise, a malicious client could access it directly and effectively bypass all the security restrictions. For that reason, all layers in the architecture, including the API origins and the Policy Decision Point, are asked to implement what we have called the "*Intermediary Authentication Protocol*". In simple terms, the protocol requests a) that all intermediaries attach HTTP headers to their requests with a ***message authentication code*** (MAC) (Federal Information Processing Standards, 1985) signed using ***shared keys***, and b) that both intermediaries and origins verify the existence and the validity of the code in all incoming requests. (The outermost layer receives requests directly from clients, not intermediaries, and is therefore not expected to verify incoming headers, but it should, on the other hand, provide headers identifying itself as an intermediary before forwarding the request upstream to the next layer).

The protocol requires intermediaries and origins to maintain a list of well-known ***Intermediary IDs*** and their corresponding ***secret keys***. Two internal HTTP headers are attached and verified by all intermediaries and origins involved in the upstream processing of a request. The first header provides information about the request from the intermediary (an ID for the intermediary, a timestamp, and a nonce). The second header provides a hashed MAC (HMAC) computed by concatenating the information from the first header with details about the request (e.g. the *base URL)*, and signing it with a shared key in order to ensure that the information in the other header cannot be tampered with.

Each intermediary should validate an incoming request by 1) making sure that the headers exist and are in the right format, 2) retrieving the secret key that corresponds to the Intermediary ID, 3) creating a signature using the same function as above, 4) comparing the new signature to the value of the signature received in the second header, 5) retrieving the timestamp and ensuring it falls within a certain window (past & future), and finally 6) retrieving the nonce and ensuring

that the request is unique (within that window). If any of these steps fails, the intermediary or origin should immediately return an HTTP 403 response, with a body detailing the reason (e.g. "*signature invalid*"). Otherwise, it should continue the processing of the request in regular fashion, but previously replacing the *Intermediary Authentication* headers that it received with new values, so as to identify itself to the next layer upstream.

Access Policies - Consumers

Once it is guaranteed that origins will *only* accept requests coming from the shared API infrastructure, it is easier to model and apply security and access control guarantees to APIs. In the most basic model, which we call the "*Consumer Authentication Protocol*", API owners are able to choose who gets access to their services, and under which circumstances. The enforcement of these restrictions (i.e., the guarantee that only authorized users are allowed to access the API) is the responsibility of the Policy Decision Point component of the platform. Through configuration dashboards and out-of-band operations, API owners can grant **Consumer Keys and Secrets** to individual consumers**,** and specify the set of policies (e.g., rate limits, time and geographic restrictions, etc.) that must be respected when each of those consumers issues a request against their API. API consumers, in turn, are asked to sign their requests using those credentials, following in particular the 2-legged version of the widely adopted OAuth 1.0a standard (Internet Engineering Task Force, 2010).

The original OAuth specification should be consulted for full details, but in summary the protocol requires that HTTP requests be decorated with an ***authorization string*** provided in the *Authorization* HTTP header. The authorization string includes the consumer's key along with a hashed message authentication code (based on the method, the hostname, the path, and the GET/POST parameters of the request) signed with a concatenation of the consumer's key and its secret (not to be confused with the Intermediary's key and secret as described in the previous section). The protocol also requires a timestamp and a nonce, to avoid replay attacks.

To authenticate an incoming request, the Policy Decision Point retrieves the shared secret corresponding to the consumer's key provided in the Authorization HTTP Header, and uses it to build the OAuth signature that it should expect given the request that it is actually processing (its hostname, method, path, and parameters). If the expected and received signatures don't match (or if the authorization string is missing or incomplete, the timestamp is older than a certain allowed time window, or the nonce parameter has already been received within that window), the Policy Decision Point instructs the outer layer in the infrastructure (the Policy Enforcement Point) to DENY the request, with an HTTP 403 code and an explanation in the body of the response (e.g., "*Missing Consumer Key"*, "*Invalid Signature"*, etc.), without forwarding the request on to the API origin. If the signature is correct, on the other hand, the Policy Decision Point considers the consumer properly authenticated, and then proceeds to apply business authorization rules such as rate limiting. Eventually, if the request is fully authorized (i.e., if it's coming from an authorized consumer, and all the access policies for that consumer are successfully passed), then it is forwarded to the API origin with an internal header that reveals the identity of the consumer, should the origin need it for tracking or logging purposes.

Access Policies - Resources

The Consumer Authentication protocol ensures that only requests from authorized consumers are allowed, and is an effective

solution when the consumer can be trusted to keep its credentials securely. For cases when the security of the credentials cannot be guaranteed (e.g., when the consumer is an application running on a mobile device), our platform also supports a "*Resource Authorization Protocol*", which gives a particular instance of that consumer access to **a single resource** (e.g., a single user account). With this protocol, if an instance of a consumer application gets compromised (e.g., a particular mobile device gets hacked), then *only the particular resource for which that device had been authorized* is compromised.

The protocol is identical to the 3-legged version of the OAuth 1.0a standard (Internet Engineering Task Force, 2010), and a slight variation over the Consumer Authentication Protocol. In summary, in addition to Consumer Keys and Secrets, valid requests must be signed with an **Access Token and Secret** as well. The protocol defines a set of flows that enable a consumer to acquire an Access Token and Secret for a particular resource (for example, if the resource in question is a user account, the protocol defines a flow by which the consumer application should redirect the user's web browser to an endpoint where the API platform will issue a **Request Token** -- potentially, but not necessarily, first redirecting to a login page and requiring that the user for whose account access will be granted explicitly authorizes the access--, which the consumer application then exchanges for an Access Token and Secret via a backend call to the API platform). Regardless of the flow chosen, the API platform internally correlates the Access Token to the identity of the particular user or resource to which access is being granted. Once the authorization tokens have been acquired, the consumer application will submit API requests signed using a similar mechanism to the one required by the *Consumer Authentication* protocol, with the additional requirement that the Access Token be also provided in the Authorization string, and that the signature be computed using both the Consumer Secret and the Access Token Secret.

The Policy Decision Point, in turn, will receive the request, verify the identity of the consumer, validate the signature, and apply access policies to the consumer, just like in the Consumer Authentication Protocol. Additionally, the Policy Decision Point will retrieve the identity of the resource that corresponds to the Access Token in the Authorization HTTP Header, and apply access policies that apply specifically to that resource. Finally, if all checks pass, the Policy Decision Point will forward the request to the API origin, modifying the original URL to include the identity of the requested resource. For example, if the consumer makes a request to *www.example.com/myaccount* with a given Access Token, and the Access Token corresponds to a user with GUID *123*, then the request to the API origin will be *api_origin.example.com/accounts/123* (the details of URL re-writing can be configured on a per-API basis, of course).

Because access permissions are attached to a single resource, and because it effectively hides the actual identity of the resource, the "Resource Authorization" protocol is also useful in cases where the API owners want to give access to the API to an external partner, without having to reveal internal resource identifiers such as UIDs. It is also useful when business policies demand the explicit approval of the user on whose behalf the consumer application will be issuing requests.

Scalability

In addition to the issues of security and access controls described above, API owners also have to address problems of scalability when they choose to expose their services for

access beyond the applications for which they were originally intended. Fortunately, scalability is another issue that may be well addressed by a common API infrastructure. In particular, our API platform contains a load-bearing tier, implemented on top of a commercially available CDN, which is setup to be the first layer to handle incoming requests (and the last to process outgoing responses). Responses from API origins may be temporarily stored at this layer, which may later choose to respond to subsequent requests directly from its cache, without the need to first forward the requests to the Policy Decision Point and the API origin.

Serving content directly from the outer load-bearing tier, however, introduces a new security issue: if new requests don't have to travel to the Policy Decision Point, then how can the platform guarantee that only authorized consumers will be allowed to access the protected services? The "*Edge Authorization Protocol*" addresses that problem, effectively offering the option of trading fine grain policy decisions for massive scale, high availability, and optimal performance. In this model, the Policy Decision Point is allowed to optionally issue cryptographically secure authorization assertions in the response to requests from a consumer. Those assertions act as logically sessioned tokens: if and only if the consumer replays that authorization token in subsequent requests may the load-bearing edge network assume that the consumer has already been authorized, and that therefore the request may be processed without further engaging the Policy Decision Point.

In terms of actual implementation, the Edge Authorization protocol specifies that the Policy Decision point may decorate the downstream response with a cookie that encodes the IP of the consumer for which authorization should be assumed, the duration of the authorization, a list of paths for which the authorization should be valid, and a signature. Before sending the final response to the client, the edge tier will check for the existence of the cookie and, if found, it will encrypt its value using its own secret key (to make it opaque). The client may then issue another request to the same API. On its end, all the standard rules apply; in particular, the request must still be signed using the *Consumer Authentication* protocol, through which the consumer identifies itself to the API infrastructure. However, the client may also replay the Edge Authorization cookie that it received in the previous response; in that case, the edge processor will decrypt its value, validate it (IP address, expiration time, path, and signature) and, if all checks pass, return content directly from its cache. It is also possible that the edge tier does not find the desired content in its cache (or that the stored response has expired); in that case, the edge tier will have to forward the request all the way to the origin, but it will be allowed to *bypass the Policy Decision Point* and engage the Service Provider interface directly, since the request has already been authorized.

From the point of view of the client, there shouldn’t be any difference in the way that it issues the request (with the exception of the extra cookie) or the way it receives a response (again, with the exception of the cookie). The same is true for API origins: how the request is routed through the API infrastructure is of no concern to them. With minimal effort, then, the shared caching and high-availability layer (along with the Edge Authorization protocol) enables API owners to take advantage of infrastructure that may be too expensive or too complicated to maintain for a single API (thereby reducing the burden of exposing their services, per the first principle) while, at the same time, giving them full control, through the well established HTTP cache control directives, to decide which sections of their APIs should be cached, which ones require refreshes, which ones may be served in a stale state in the case of origin downtimes, etc. (following the third

principle).

It must be noted that the Edge Authorization protocol introduces an obvious trade-off, in which scalability and performance are gained at the expense of fine-grained security decisions. If responses are served directly from the cache or the origin, bypassing the Policy Decision Point, one loses the ability to immediately revoke access to a consumer (i.e., the edge will continue to grant access until the edge authorization cookie expires). The rate limiting and usage reporting capabilities afforded by the Policy Decision are also lost for requests that are edge authorized. For these reasons, API owners must retain control about which consumers may participate in this Edge Authorization protocol, and for how long the authorizations should be granted. In practice, the Edge Authorization protocol is only used with selected APIs and consumers, and the expiration times for the edge authorization cookie usually set to no more than 1 or 2 minutes.

Traceability

For purposes of traceability and operational awareness, necessary given the various layers involved in the processing of a request through the API platform, intermediaries (including the API origins) are asked to decorate upstream requests (that they process themselves or forward to other layers) and downstream responses (that they return to upper layers) with values that they append to an internal HTTP header. The details of the header are specified by our "*Message Exchange Fingerprint Protocol*".

Specifically, an HTTP header is added by all systems that participate in the handling of a request. Values from different components are separated by a single space, and each value in turn consists of a dash-separated set of a) a direction prefix, either UPSTREAM ("u") or DOWNSTREAM ("d"), b) a component identifier, granular enough to provide uniqueness to the fingerprint (e.g., in a Java system this could some arbitrary "system ID" plus a "process ID" plus a "thread ID", and c) a timestamp, recorded in Unix time (milliseconds since the UTC epoch).

An example value for the internal header, as received by the consumer with the full HTTP response, is shown below (pretty-printed to show processing brackets):

```
u-cdn75209+72.246.30.14+138036545-1287680639000
  u-proxyworkeri6866f301.pdp.com8735-1287680639532
    u-t24005274509060@API-QA-1287680639596
    d-t24005274509060@API-QA-1287680639597
  d-proxyworkeri6866f301.pdp.com8735-1287680639591
d-cdn75209+72.246.30.14+138036545-1287680639800
```

This shows which components participated in the processing of the request (a CDN server at 72.246.30.14, a Policy Decision Point worker with the ID 6866f301, and thread ID t24005274509060 on the API-QA origin). The header is also useful to show the time each bracket took in processing the request (including the calls to the lower layers): 1ms at the origin, 59ms on the Policy Decision Point, and 800ms total at the edge layer (including the time spent waiting for a response from the next layer down).

DEVELOPER PORTAL

The multi-layered architecture of the API infrastructure, along with its protocols and the procedures that providers and consumers must follow, address the first four principles that guided our efforts to encourage the creation of APIs throughout the organization: *minimize effort for API providers*, *minimize effort for API users*, *leave as much control as possible with API providers*, and *do not obscure the communication protocols*. The sandbox platform, a copy of the production environment but running under a default edge URL, tackles the fifth principle of *minimizing unnecessary delays* for API owners. In order

to increase the adoption of existing APIs and address the fifth principle for **API consumers**, we created a Developer Portal, a central place where service providers advertise and document their APIs, and where potential users quickly find out about the APIs offerings and learn how to use them.

Access to the Developer Portal is limited to employees within the organization, and their access credentials tied to the organization's LDAP servers, to minimize the effort required to participate in the portal. Once logged in, visitors can access Wiki pages that explain the inner functioning of the API platform and its protocols, and in particular provide detailed instruction for signing requests using the Consumer Authentication and Resource Authorization protocols (libraries for several programming libraries are also provided here).

More importantly, visitors to the Developer Portal have access to a full API Catalog, listing all the available APIs categorized by group or function, and with links to find further information about each. This is where developers learn about the APIs that they can use to solve the problems they are working on. Each API, in turn, has its own set of documentation pages (which are carved out from the Developer Portal's URL namespace when the API is first exposed through the infrastructure), and where we encourage API owners to provide extensive documentation. Most commonly, the documentation includes the basic definition of endpoints, URLs, path structures, and query parameters, along with the kinds of responses that API consumers should expect (and the error codes that might be returned).

So that future consumers may get as real a taste for what the API can do, moreover, we allow API owners to attach what we call "*LiveDocs*" to their API's pages. LiveDocs are small, inline boxed forms that can be easily embedded in documentation pages, and through which potential consumers may specify parameters and issue *live* calls (from the browser only) to the corresponding API, without having to worry about getting API keys or having to learn how to sign and issue HTTP requests (a responsibility delegated to the LiveDocs plugins). In addition to displaying the responses returned by the APIs, LiveDocs also present detailed information about how the request was issued (so that potential users can learn about how to sign their requests), the headers that were returned (so they can learn about special protocols such as Edge Authorization), and the intermediaries that participated in the handling of the request (so that they can learn to use the Message Exchange Fingerprints protocol and explore the inner workings of the platform). In this way, general documentation and LiveDocs allow potential consumers to learn as much as possible, and with very little effort, about the set of APIs that are available through the platform, and how they can use them.

The LiveDocs boxes are limited to the parts of the API that owners want to bring attention to in their documentation pages (e.g., they only allow for a small subset of parameters, or provide closed choices for a parameter in the form of a drop-down list). Sometimes developers need more than that. For that purpose, and once they have requested and acquired credentials to access those APIs, developers can access a more powerful on-line tool (also available from the Developer Portal), which we call "*codebug*". Through codebug, developers have full control over all the details of their request: they can specify the full URL of the request, use HTTP verbs other than GET, provide specific headers, add Pragma directives that control the behavior of the platform, and even change signature methods. The codebug console takes care of signing the requests using the developer's credentials for the API, and then presents the same information returned by LiveDocs: the request and

response headers and parameters, the actual response body returned by the API origin, and tracing information showing how the request was handled by the platform.

We believe that being able to interact directly and issue live calls to the API, changing parameters and observing the XML or JSON that the API returns, makes a big difference in terms of the adoption of APIs. Developers can effectively interact with an API without having to write a single line of code, and therefore are able to explore and adopt existing APIs quicker (and based on first-hand experience).

LESSONS LEARNED - FUTURE WORK

Since the release of the API platform within the organization, we have seen an important increase both in the number of APIs offered (for services which would have normally remained closed), and in the adoption of APIs across organizational boundaries (to reuse solutions created by teams from very different groups). More APIs are being incorporated every month, and we may be getting close to a tipping point, from which all (or most) teams will be expected to have solutions running on the platform. Ultimately, this is enabling us to reduce the duplication of efforts, increase the security of our applications, gain deeper understanding and control of who accesses each service, and innovate with shorter development cycles and reduced time to market.

The experience of building and operating the API infrastructure has also given us data to verify our assumptions about what was required to encourage the production and use of APIs, and to discover ways in which we could improve. In particular, we sense the need for improvement in the following areas:

1. **Intra data-center calls need to be treated differently**. The API platform, as detailed in previous sections, consists of an initial CDN layer followed by a Policy Decision Point. In our initial implementation, these two layers are implemented and operated by existing commercial offerings, outside of our internal data-centers. This architecture imposes a significant (and unnecessary) latency overhead when both the API consumer and the API origin are located within the same data-center: requests from the consumer have to travel all the way out of the data-center into the CDN, and then to the Policy Decision Point somewhere in the open Internet, before coming back to the data-center to the processed by the API origin (and the responses need to follow the inverse path).

To address this issue, we are evaluating solutions to bring these layers in-house.

2. **Intermediary Authentication is sometimes hard to deploy**. As we engaged internal teams to incorporate APIs onto the platform, we discovered that some of those services are owned and/or operated by third parties, and it is either difficult or expensive to access their source code. As a consequence, adding the necessary filters so that the origins conform to the Intermediary Authentication protocol is a non-trivial exercise.

To address this issue, we deployed an internal gateway whose sole purpose is to front those closed systems: the gateway handles Intermediary Authentication, and the API origins sit behind a firewall that only enables requests from the gateway. While adding (minimal) latency, this enables us to expose services that otherwise would have to remain closed (or would be very expensive to open up without security risks).

3. **API owners need to have complete access to traffic logs**. When an API origin sits behind the API platform, its web server access logs lose richness. For example, it is no longer possible (or easy) to find our which IP addresses are hitting it (since they are all

coming from the Policy Decision Point), or filter out requests that had to be dropped because of security violations (since they are dropped by the Policy Decision Point), or which ones resulted in cache hits (since caching is handled by the CDN layer). This is useful data that API owners should not lose access to.

To address this issue, we are currently working on a prototype solution that fetches and combines log files from the three infrastructure layers (the CDN, the Policy Decision Point, and the API origin) and makes them available to API owners. In the future we will consider integrating this tool with the existing API management dashboards.

4. **Exposing APIs implies a cultural shift, which takes time and education.** Early adopters of the API platform were easy to get on board: they were already interested in exposing their APIs, they felt comfortable with the security implications, they understood the details of layered HTTP architectures, and they were able and willing to use low-level tools (and cumbersome procedures) to configure their endpoint. As we push for adoption beyond this initial group, however, it is becoming more important to be able to explain in less technical terms the advantages of opening up APIs, to detail more explicitly the security guarantees that the platform provides, and in general to provide more guidance about the best ways to go about doing it.

To address these issues, we are working on Best Practices documents, open security reports and assessments, easier to use (and more accessible) forms and procedures for creating API endpoints in the sandbox, and simplified key management policies. We are discovering that, in order to gain wide acceptance and adoption, the existence of these materials is as important as the technical merits of the platform.

ACKNOWLEDGEMENTS

This API project was initiated at Comcast by Matt Stevens, who produced the original design of the architecture and core protocols. It was later maintained and expanded by Benjamin Schmaus, with further help from Peter Cline, Michajlo Matijkiw, and Matt Hawthorne.

During the implementation phase, we got invaluable help from Philip Grabner, Satish Narayanan, and Cory Sakakeeny from Akamai, and Bill Lim and Peter Nehrer from Mashery. Also fundamental were the contributions of Bahar Limaye, from Software Security Consultants, who performed in-depth reviews of the security of the platform.

Many people within Comcast pushed for the adoption of the infrastructure throughout the organization, and I'd like to thank in particular Rich Woundy, Rich Ferrise, and Hai Thai.

REFERENCES

W3C. (2007). SOAP version 1.2. Retrieved from W3C: **http://www.w3.org/TR/soap/**

Burbeck, S. (1987). *How to use Model-View-Controller (MVC)*. Retrieved from http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html

Benslimane, D., Dustdar, S., & Sheth, A. (2008). Services Mashups: The New Generation of Web Applications. *IEEE Internet Computing , 12* (5), 13-15.

Dijkstra, E. W. (1982). On the role of scientific thought. In *Selected writings on Computing: A Personal Perspective* (pp. 60–66). New York, NY: Springer-Verlag New York, Inc.

Federal Information Processing Standards. (1985). *Publication 113 - Computer DataAuthentication.* Federal Information Processing Standards.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures.* Irvine, CA: University of California, Irvine.

Internet Engineering Task Force. (2010). *RFC 5849: The OAuth 1.0 Protocol.* (E. E. Hammer-Lahav, Ed.)

Papazoglou, M., & Georgakopoulos, D. (2003, October). Service-Oriented Computing. *Communications of the ACM , 46* (10), pp. 24-28.

Parnas, D. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM , 15(12)*, pp. 1053-1058.