Evaluating Best-of-Class Web Service APIs for Today's Multi-platform Video Management Solutions

By Alan Ramaley, CTO, and Nick Rossi, VP Engineering
thePlatform for Media, Inc.

ABSTRACT

Video management and publishing platforms are evolving to meet the market's need for reaching consumers with reliable, high-capacity services – anytime, anywhere, on any device. As such, solution providers have to integrate their technology with a vast set of devices, systems, and environments—includingauthenticated syndication, third-party websites, mobile devices with vastly differing specs, set-topboxes, connectedTVs, smart over-the-top devices, andthird-party services, such as ad networks and content discovery engines.

Web service application programming interfaces (APIs) play an integral role in enabling content providers and distributors to succeed in a consumer driven market that's in constant flux. Developers at media companies and TV service providersneed flexibility and open APIs to adapt to changes in TV, online, and mobile video publishing.

This paper provides an in-depth evaluation of the most important features web service APIs should offer and explains why those features are important. It also examines the evolution of APIs and recommends best practices for a flexible, reliable and easily managed API set.

Several areasfor evaluation are examined and explained, all with an eye towards how APIs informed by service-oriented architecture (SOA) can be used to decoupleand safeguard business-critical services in a deployment and scale them independently.

Areas of focus will include:

- ***Breadth** – an API should expose all the functionality in the underlying service*
- ***Cohesion**– a given service should have a single area of responsibility*
- ***Security** – we will compare and contrast five common models*
- ***Web standards**– support for REST,Atom, RSS, and JSON for data services, and REST and SOAP for business services.*
- ***Data access** – APIs should provide very flexible read and write access to service data*
- ***Notifications** – with a comparison of push vs. pull notification models.*
- ***Extending the schema**– what to look for to make sure a service can support your custom data.*
- ***Scalability**– how to build scalability into an API at the core, to allow for a 99.99% read SLA*

Lastly, the paper focuses on some of the best developer support practices, including API clients and documentation.

INTRODUCTION

The recent introduction of Time Warner Cable's iPad application is just one example of the kind of services and applications that service providers and media companies can develop in-house by taking advantage of open web service APIs.

Going forward, web service APIs will continue to play a crucial role in enabling developers at content companies and TV service providers the flexibility to develop new services and respond to the changes in multi-platform video publishing.

When video management systems were in their infancy, few offered a set ofAPIs that anybody could use to build a media business.Most solution providers incorporated user interfaces on top of proprietary systems that could only expand when in-house developers felt like adding features. If anoutside user wanted to conduct their own development on top of such systems, they were out of luck.

The industry has since learned that web service APIs are a critical component for content providers and distributors, as it enables them to adapt to a fluid marketplace where consumer demand and IP-connected technologies are in constant flux. For this reason, APIs are now a standard part of every video management system. But despite the widespread adoption of APIs, not every system is equal. It begs the question: How good are a given system's APIs, and willthey continue to meet the needs of a media business as it grows?

This paper explores the most importantcapabilities to consider when evaluating the effectiveness of a system's APIs.

BREADTH

First, APIs should expose as much of the video management system's functionality as possible. It's very hard to predict what parts of your system you'll need to automate, based on where customer needs take your business. So,the more elements are available via the API, the more flexibility you have to respond to a changing marketplace.

Verification Process

A good ad-hocapproach to testing an API's breadth is to go to the management console or user interface and ascertain whether the technology vendor uses its own published API. If the vendor is not using it, not only is that a sign that they haven't built their system for maximum adaptability, but it also demonstrates that thevendordoesn't rely on its own APIs to support its product.

This can often be checkedby watching a network trace while using the system'smanagement console. If there areprivate protocols or undocumented payloads going back and forth, then it's likely the public APIs aren't complete enough or powerful enough for general usage.

COHESION

Each API endpoint should focus on a single area of responsibility within the system and use consistent operations and serialization methods for everyobjecttype. With one set of rules to interact with the system, developers can more easily integrate with it.

Multiple Services Versus a Single Monolithic Service

In a provider's API, if every call goes against a single "api.provider.com" or "services.provider.com" endpoint, with some kind of "command" or "service" parameter as a switchboard, that means the API provider has implemented a single monolithic endpoint that contains all APIs.

For example, you might see calls like this in a monolithic API:

- **http://api.provider.com/index.php?service=baseentry&action=list**
- **http://api.provider.com/index.php?service=multirequest&action=null**
- **http://api.provider.com/index.php?service=flavorparams&action=list**
- **http://api.provider.com/index.php?service=accesscontrol&action=list**
- **http://api.provider.com/index.php?service=partner&action=getInfo**

A monolithic API has several drawbacks:

First, it makes federated deployments very difficult,where some data is local to the content or service provider while other data is in the cloud. For example, youmight want to use an API cloud for most services, but store end-user transaction data locally for securitypurposes. A single, monolithic API endpoint does not have this capability.

Second, it puts a limiter on how fast the provider can extend the service. As the feature set grows, provider development will lag as internal teamsare encumbered with the increasing overhead ofcoordinating feature work and deployments in a single code base.

Finally, there's no single scalability strategy that works for all APIs: some get orders of magnitude more traffic than others, and the mix of read vs. write traffic varies, but the deployment of a single switchboard API is limited to an unhappy compromise between traffic capacity and cost.

One must be rigorous about dividing services into areas of responsibility to avoid these pitfalls. A good system will split its APIs into separate, focused services in which each API endpoint has a single job. This ensures that other services aren't affected if unexpected load hits one piece of the service, and the deploymentcan scale each endpoint as appropriate. For example, if there is an abundance of feed requests, administrators can simply add another feeds server instead of spinning up another instance of the entire API stack.

Data Services versus Business Services

There are two basic types of web services:

1. **Data services**, which handle stateful persistence of metadata.
2. **Business services**, which are stateless services with business logic that interactswith the data in data services.

The best approach is to look for a web-service framework that follows the principles of service-oriented architecture (SOA), which decouples data persistence from business logic so that services of each kindcan be deployed and optimized independently.

Base Objects

Optimally, every data service object has a base object with identically named properties for identifiers, modification history, and other common settings.

Properties such as ***title***, ***id***, ***guid***, ***added***, ***updated***, and ***locked***should beconsistent across all services. Consistently identified properties are beneficial, especially when querying for data objects, since the same kinds of queries can be used across all implementations.

If a framework implements these core properties, you can use similar queries across various services. For example, if "updated" is a base property, here's an example of a query you could use inanyservice to get items updated in the month of September 2010:

> http://<service>/data/<objectType>?**byUpdated=2010-09-01T00:00:00Z~2010-10-01T00:00:00Z**

If "id" is common, the following query could be performed in order to get object IDs sorted by when they were added:

> http://<service>/data/<objectType>?**fields=id&sort=added**

Finally, if "title" is common, in order to search for the first five items that have a title starting with "Test",one could execute the following query:

> http://<service>/data/<objectType>?**byTitlePrefix=Test&range=1-5**

In a system that implements base objects and base queries, the only things that need to change in order to perform the queries in these examples are the host name and the object name. Because the pattern repeats across services, once you've learned how one service works, you've learned how all of them work.

SECURITY

APIs must be secure, and calls to APIs from end-user services (such as web form comments) must be completely separated from admin services (such as video publishing).

Admin Security

The level of admin security that is needed depends on what the user is trying to accomplish. Web service API authentication methods tend to fall into one of five models. See the table on the following page, which outlines each security type:

Security Type	**When You'd Use It**	**Drawbacks**
API freely available to any user	There are some cases where no security is desired. For example, read-only, highly constrained RSS feeds that are exposed to end users.	Not secure enough for admin authentication.
User name and clear-text password on every URL	Never	Credentials shouldn't get passed around on calls, as they can be intercepted and used indefinitely in ways that can't be controlled.
Vendor-provided API key	This is typically a random keybound to an organization with a particular set of rights. In many ways, this is similar to a user name and password on URLs, with the token acting as an obfuscated password.	If it's compromised, it can be used indefinitely to make additional calls until one discovers the breach and revokes the key.Revoking this key will typically disable legitimate uses as well, which will need to get updated with a new key.
Non-expiring API keyplus a signature on every URL	The signature is generated by hashing the URL parameters with a private/secret key. The hash is checked on the server before allowing the call. This works fairly well for server-to-server traffic or trusted clients: the URL can be used forever, but cannot be used to create different calls.	To do client-side end-user AJAX-style UI, one needs to push theprivate secret to the client to create this hash for each call, which makes the secreteasy to compromise.
Expiring token	This approach involves making a call to a secure API to generate an expiring token tied to a particular user's permissions, and then including that token on subsequent calls.	These can be captured via "man-in-the-middle" attacks, but this is mitigated by the expiration date.

While none of these options are impenetrable, the best approach is to use an expiring token. This solves the problem of tokens never expiring, so even if the user or system doesn't realize that a token is compromised, it can't be used after the expiration time specified when it's requested. And because the token is reusable across calls while it's still valid, there is no need to push signature secrets to the client.

API Types

There are three kinds of APIs that a video service should provide:

1. An admin read/write API that requires admin permissions to work with. For example, service providers would use an admin media API to publish or edit premiumvideos.

2. An audience read/write API that might also allow anonymous access. For example, viewers would use an audience API to add or edit comments or ratings.
3. A read-only, highly cached feed API for end-user guide data. For example, viewers would use a feedAPI through a video playerto retrieve lists of content or play videos.

All of these APIs must be separated. Audience users or viewers cannot be allowed to make admin API calls. For scalability, audience users must access different services altogether.

The primary concern is resource contention: if audience members are allowed into a service or content provider's admin APIs, even with an "audience-only" read-only token, they're competing with the provider's admin requests for API resources. This means that service providers run the risk of having a massive spike of audience-originating requests disrupt video publishing. Or, vice versa: the consumer experience could be degraded by admin API activity. Either situation can negatively impact your business with complaints of perceived outages.

Also, each type of API has a different usage pattern and should betuned separately. Admin APIs are used by a relatively small set of users, and get a relatively high volume of writes, while end-user APIs need to support a massive scale of read traffic with relatively few writes They all need to be configured differently to run optimally.

It's important to evaluate how a system implements a secure wall between these three kinds of services. One effective method is to have separate authentication services for administrators vs. audience members and configure a given API to run against the appropriate one. Another method is to physically separate the deployment of such services so that traffic on one cannot affect the others.

WEB STANDARDS

Avoiding any web service with a proprietary serialization format is preferred. When services support web standards, it's easier for developers to find clients and tools that can consume those services.

Data Services: REST withAtom, RSS, and JSON

It's imperative that any platformsupport a diverse set of Web standards, to increase the chance of interoperability with existing solutions. The following are specific examples from thePlatform's web services, but it should be straightforward to determine the pattern that any web service follows to deliver standards-based serializations:

Atom:

http://feed.theplatform.com/f/ZlTfSB/wAeAAAKTtr_L?form=atom

RSS:

http://feed.theplatform.com/f/ZlTfSB/wAeAAAKTtr_L?form=rss

JSON:

http://feed.theplatform.com/f/ZlTfSB/wAeAAAKTtr_L?form=json

JSON should alsosupport "callback" and "context" parameters for **JSONP**-style usage:

http://feed.theplatform.com/f/ZlTfSB/wAeAAAKTtr_L?form=json&callback=parseFeed&context=sample

Business Services: SOAP and RESTful XML /JSON

The service provider should support SOAP APIs for business services, including WSDL URLs for discovery of method signatures.

SOAP can be a heavy protocol to work with, so the service should also support some XML-RPC variant.For cases where you want the smallest serialization possible, the service should support JSON as well.

Finally, for cases where business service calls need to be made cross-domain in web browsers, the service should support a RESTful interface with JSONP responses.

REST Verbs (HTTP methods) and Just GET

Many REST implementations just support HTTP GET. Even when creating, updating, or deleting objects, one has to do a GET with parameters embedded in the URL.

However, following REST to the letter, this is incorrect. HTTP GET is supposed to be idempotent, so no matter how many times you call the same GET URL, it should make no change to the server state. Other verbs like POST, PUT, and DELETE are intended for state changes. That's why there is a prompt by browsers when refreshing a POST, but not when refreshing a GET.

If your code is making JSONP calls in a browser, the nature of cross-site scripting security requires that every request to a remote domain use a GET, and a good API should make an exception to idempotence in this case. But in less constrained clients, it's cleaner and more standards-based to use the available HTTP verbs with their typical interpretation: POST to create, PUT to update, and DELETE to delete.

For example, to delete a media with an ID of 1586532611, the following HTTP call would be made:

DELETE
http://mps.theplatform.com/data/Media/1586532611?schema=1.2.0&token=...

One could also delete everything with a particular title prefix:

DELETE
http://mps.theplatform.com/data/Media?byTitlePrefix=Old+Media&schema=1.2.0&token=...

If a given URL is too long for server gateways to handle, it's possible to convert this to a POST using the ***application/x-www-form-urlencoded***content type header, and put the parameters in the POST body, along with a ***method*** override:

POST
http://mps.theplatform.com/data/Media
Content-Type: application/x-www-form-urlencoded

method=delete&byTitlePrefix=Old+Media&schema=1.2.0&method=delete&token=...

Finally, if GET must be used for calls from a browser to avoid cross-domain issues, one can do a GET with a method parameter override to delete:

GET
http://mps.theplatform.com/data/Media/1586532611?method=delete&schema=1.2.0&token=...

DATA ACCESS

The ways in which solutions consume data are as varied as the data itself, so a service provider should offer flexible APIs for updating, querying, searching, sorting and paging lists of data, similar to what is possible withdatabase or search queries.

Combining Queries

Many web services don't let you combine queries: if you see method names like ***findByCategory*** or ***findByRating***, that means you'll never be able to do a query that searches for both category and rating.

Any given API should implement a set of base queries and then additional queries, and you should be able to use them in any combination. For example, our media API supports over 20 different queries. Here's a query for objects in the "Action" category:

http://feed.theplatform.com/f/ZlTfSB/wAeAAAKTtr_L?byCategories=Action

And here's a query for anything in ***Action*** OR ***Comedy***:

http://feed.theplatform.com/f/ZlTfSB/wAeAAAKTtr_L?byCategories=Action|Comedy

And here's a query for anything in ***Action*** AND ***Comedy***:

http://feed.theplatform.com/f/ZlTfSB/wAeAAAKTtr_L?byCategories=Action,Comedy

You can combine a category query with a content rating query:

http://feed.theplatform.com/f/ZlTfSB/wAeAAAKTtr_L?byCategories=Comedy&byRatings=G

And you can further combine these with a custom data query:

http://feed.theplatform.com/f/ZlTfSB/wAeAAAKTtr_L?byCategories=Comedy&byRatings=G&byCustomValue={year}{2010}

In a flexible API, there should be no limit to the number of queries you can combine.

API Queries Across Multiple Accounts

A larger organization often needsmultiple accounts in the web service for different groups. Optimally, these organizations will want the APIs to be able to fetch from multiple accounts at once, instead of having to switch users or tokens for each account.

Getting All Objects at Once

Often there is a scenario where the user wants to get every object in one call. For example, you might be synchronizing with a new content management system (CMS), and youwant to get every video in the account(s).

Most web services can't handle this, so they restrict the maximum "page" size a given API call can return, typically to 20 or 100 items. This puts the burden on client code to retrieve successive pages and deal with any errors that come up between them. It also means that the server keeps invoking larger and larger internal queries to skip over the results that were in previous pages, and each subsequent page will therefore be slower to fetch.

A more effective system avoids this by allowing unbounded result sets that stream out to the client. To get every item available in an API, you might make a call like this:

http://feed.theplatform.com/f/ZlTfSB/wAeAAAKTtr_L?range=1-*

Control Over the Sorting of Result Sets

Most web services allow control over API result sorting, but almost all put tight restrictions on what can be sorted and only allow one level of sorting. A more effective API allows sorting on any combination of fields. Here's an example of a sort by title:

http://feed.theplatform.com/f/ZlTfSB/wAeAAAKTtr_L?sort=title

Here's the same sort, but flipped to be in descending order:

http://feed.theplatform.com/f/ZlTfSB/wAeAAAKTtr_L?sort=title|desc

And here's a four-level sort, by locked, then approved (descending), then the ***year*** custom field, and finally publication date:

http://feed.theplatform.com/f/ZlTfSB/wAeAAAKTtr_L?sort=locked,approved|desc,:year,pubDate

NOTIFICATIONS

There are many cases where a client process needs to know when data changes in a service, such as when synchronizing datawith another content management system, invalidatinga customized caching layer, or keeping an audit trail of edits.

Notifications are the optimal way to keep things in sync. Suppose there is the need to synchronize a list of feeds between the provider's web service and the content management systemthat is being used for a provider's web site. When a feed changes, the provider's web service would create a notification that thesynchronization logiccould pick up. Thelogicwould then usethat notification to update thecontent management system. This wouldn't be a human-readable notification like an email, but a special payload designed to be consumed by an API client.

Support for Notifications

Some APIs don't support notifications, and their only mechanism for synchronizing with other systems is to use a polling approach against the actual objects. These APIs require you to check for changes every few minutes, typically with a ***modifiedSince*** query or something similar. If data has been modified, you have to figure out if the change matters to theclient.

This is an inferior approach when objects aren't constantly changing, becauseit forces you to make many pointless calls when there's no change just to make sure there is a timely update if something does. This polling taxes the client, the server, and the network.

The better approach is to supporttrue notifications that clients can register for. An API without notifications hasn't taken third-party integrations seriously.

Notifications for All Objects

Many web services are stingy with their notifications. They'll support notifications on some objects but not others, or they'll only send a notification when a video finishes processing, or they'll only notify on add and delete, but not update.

A more effective APIsupports a complete set of notifications. It should notify users on create, update, and delete for every object, and provides updates within seconds of the change being committed.

Notifications through a Firewall

A common but naïve approach to notifications isto let users register a "notification URL", and whenever something changes, the system sends an HTTP message to that URL.

There are limitations to this approach. If there are any hiccups at all between the notificationserver and the customer's server (e.g., network glitch, the customer's server is down for maintenance, exception in the customer's notification handler, etc.), notifications get lost. Most customers that use this approach have to run a monthly "resync everything" process to deal with these lost messages.

This method also requires the user to expose a public URL that anybody could hit. It could be locked down with passwords and IP exclusions, but it providesa vector for attack if hackers found out about it.

A better approach is to store all notifications on the server and serve them with a Comet-style "push" model. A typical exchange starts with a call like this from a client:

http://mps.theplatform.com/notify?token=...

This returns a payload with the ID of the most recent notification available. The programmer then uses this ID and makes another call to open up an HTTP request, which stays open until there arechanges to report:

http://mps.theplatform.com/notify?since=668017728&block=true&token=...

When an object changes, the response gets returned along with the latest notification ID, and the cycle continues. Because the client always initiates this exchange, it can function safely behind a firewall.

Notification Delivery

Another advantage of client-initiated, Comet-style notifications is that this approachcan guarantee that the client never misses a notification.

If clients go offline, for example, they can use their last remembered notification ID and request all notifications since that ID. For example, if the server storesnotificationsfor seven days (or whatever period of time is deemed acceptable), thenas long as the client does not go offline for longerthan that, there's no chance of any notification getting lost.

Notifications can also be delivered in the exact order they were committed to the data store, so they are always received in the proper order.

EXTENDING THE SCHEMA

Because a web service rarely has the exact schema needed for a solution, APIs must allow the providerto extend the object schema with custom fields and data types.

Custom Fields for All Objects

Many videoservices only support adding custom fields to the main media or video object. It's important to have the ability to add custom fields to as many objects as possible in the API. This ensures that in addition to adding custom fields to media, customers can also add them to players, feeds, categories, servers, etc.

Some technology vendors require you to contact their support organization to add custom fields. Others enable only a small number of custom fields to be added. Based on our experience with real-world schemas, a good upper limit is 100 custom fields per object type. It is important to allow users to administer custom fields themselves, so the vendor is never standing in the way of solution development.

Support for Custom Data Types

Many web services don't support the idea of typed custom fields. Instead, custom fields are always strings or string arrays. But typed custom fields are important in all layers of an application for the same reason they're important for native fields:

- The ability to do queries that take advantage of the data type (e.g., date range queries or numeric queries).
- Sorting that works correctly [e.g., the numbers 1, 2, and 11 would sort incorrectly as strings (1, 11, 2), but sort correctly as numbers].
- Data that is correctly serialized. This means that the user doesn't have to write any ***toString()*** and ***parseString()*** functions in their code.
- The ability to choose the right control types in the console UI (e.g., date types get a date picker).

For video services, it's important to look for an API that supports at least these custom data types:

- Boolean (true, false)
- Date (9/19/2010, 2/7/2011, etc.)
- DateTime (9/19/2010 3:27 PM, etc.)
- Decimal (1.01, 2.02, etc.)
- Duration (0:01.5, 1:34:23, etc.)
- Image (an object with an image URL and a hyperlink URL)
- Integer (-3, 5, 10000, etc.)
- Link (an object with a hyperlink URL and a title)
- String ("hello!", etc.)
- Time (9:15 AM, 3:27 PM, etc.)
- URI (any well-formed URI)

It's also important that an API support arrays and maps of anyof these types.

Serialization of Custom Data

Often, web services won't allow control over the namespace, namespace prefix, or tag name for custom data. Instead, they'll use a proprietary serialization.

But if you aredesigning a solution around web standards that require fields in a particular XML namespace, you need to be able to control all these aspects.

For example, suppose you want to add a "Latitude" custom field to media. If using Geo XML, then the custom field needs to be serialized as follows:

```
<feed
xmlns:geo="http://www.w3.org/2003/01/geo/wgs84_pos#">
<item>
    <title>My Media</title>
    <geo:lat>51.51</geo:lat>
</item>
</feed>
```

Ultimately, custom data serialization should match the consuming client'sneeds: if you have an existing solution that's expecting custom data in a particular namespace, you don't have to change that solution.

Searching by Custom Data

It's also important that the API offers the ability to search by custom data. Otherwise, you would need to implementsolutions where the client pullsback more data than itneeds in order to filter the results, which is inefficient.

Here's how a solution could support this. For example, to see all movies with a "year" value of 2008, the query might look like this:

```
http://feed.theplatform.com/f/ZlTfSB/wAeAAAKTtr_L?byCustomValue={year}{2008}&fields=title,:year
```

To see all movies released between 2008 and 2010, one could perform a ranged query:

```
http://feed.theplatform.com/f/ZlTfSB/wAeAAAKTtr_L?byCustomValue={year}{2008~2010}&fields=title,:year
```

Ideally, you should be able to invoke range queries on any numeric or time-based custom field type, in addition to exact matches.

Sorting by Custom Data

Most web service solutions don't allow for sorting by custom data, but it's an important capability to have. The ability to sort on the server is usually faster than trying to sort on the client. It also allows the ability to page through multiple pages of results with a consistent sort order.

For example, to see all movies sorted by a "year" custom field"with a tiebreaker sort on the native "title" field, the query might look like this:

```
http://feed.theplatform.com/f/ZlTfSB/wAeAAAKTtr_L?sort=:year,title&fields=:year,title
```

SCALABILITY

APIs must be able to handle the provider's site traffic. They should be like a dial tone: always on. It's notoriously hard to support this with hosted services—especially multi-tenant services—and it's a rare organization that's been able to do it: that short list includes Amazon, Google, Salesforce, Yahoo, and a few others.

In order to scale hosted services, it's important that the system has a minimum of 99.99% guaranteed uptime in which the service is available for reads. If using a service with a 99.9% service level agreement (SLA) on reads, that means that the user might not be able to do reads for nearly forty-five minutes each month, and that's forty-five minutes a month during which thesite might be completely dark. A 99.99% read SLA means the service is guaranteed to have unscheduled read downtime of less than five minutes each month.

High Availability

Service reliability depends on a properly engineered deployment that includes redundancy, automatic failover, and 24/7 human response when services experience failures.

It's important to ask a service provider pointed questions about their deployment architecture and the systems they have in place to prevent and respond to outages. A provider should satisfy your questions with evidence of redundant infrastructure, API traffic management, quality engineering in the web services themselves, and a 24/7 support team.

For example, data service APIs should automatically failover to a read-only copy of the data when the primary data source experiences a failure. Data storage in general is unreliable enough that such failover systems are one of the many requirements for achieving 99.99% uptime.

Also, in multi-tenant systems, web services should be designed to prevent heavy traffic from any one tenant from interfering with others.

Response Cache

Most solutions involve repetition of a small set of read-only API calls. Thus, for performance,APIs should provide a response cache for read requests. An API call where the response comes from a cache will respond in a fraction of the time (often under 10 milliseconds) compared with a call that invokesqueries to a database or other remote service.

A good way to check if the API is using a response cache is to look for the ***X-Cache***header in HTTP GET responses. Here's an example from our console data service:

X-Cache: HIT from data.mpx.theplatform.com:80

This tells you whether or not the response came directly from the response cache; there should either be either***HIT*** or ***MISS***. Another indicator of a response cache is a ***Last-Modified*** and a ***Cache-Control*** header. For example, for GET calls, one should see headers like the following:

Last-Modified: Tue, 05 Oct 2010 22:54:57 GMT
Cache-Control: max-age=0

Also, watch what happens when you pass in an***If-Modified-Since*** header with the ***Last-Modified*** value from a previous call. If the service has a response cache, there should be a 304 response:

HTTP/1.1 304 Not Modified

If the API doesn't show any signs of having a response cache, it's not going to hold up under load.

UI Edits

Some vendors implement their console application separately from their API. The result is that when changes happen in their console, it can take some time—sometimes up to five minutes—for the changes to appear in the API response cache.

This doesn't occur when every part of the system is run off of the underlying API, including the console. If a change is made in the console, it will show up in the next admin API call. But if nothing changes, then the API will hold on to the cached response, and the console user willexperience fasterinteraction.

Support for Sparse Objects

A rich object definition will have many fields on it, but it's unlikely that youneed every field when you make a request. Minimizing the actual set of fields returned improves performance at the server (less to query and serialize), over the network (less to transmit), and on the client (less to parse).

It's important that APIs support sparse objects with a ***fields*** parameter or something similar. For example, the following would just return ***title*** and ***id*** fields.

http://feed.theplatform.com/f/ZlTfSB/wAeAAAKTtr_L?fields=title,id

Asking for all fields in a particular namespace would return everything in the media RSS namespace:

http://feed.theplatform.com/f/ZlTfSB/wAeAAAKTtr_L?fields=media:

If an API supports dependent objects—objects that are contained in other objects, like files inside of media—the system shouldalso support field lists for the dependent objects:

http://feed.theplatform.com/f/ZlTfSB/wAeAAAKTtr_L?**fields=title,content.url,content.bitrate**

If these kinds of nested objects are supported, it means that youcan make fewer calls to get the data you need.

Sparse objects are important for create and update calls as well. An API should allow you to specify the fields to update in a call, rather than requirefull objects. Such sparse updates should be less expensive to invoke than a full update.

Multi-Item Create, Update and Delete

Many APIs only allow write operations on a single item at a time. For example, ifyou want to add 100 media, youneed to make 100 separate calls over the network. So, for every call, you are penalized for network transit time as well as the time to do an individual insertin the data store:

Total Time = (# of items) * ((network latency) + (time to add 1 item to data store))

Some APIs try to work around this via "boxcarring," where you package multiple API calls into a single request. However, this methodonlyreduces the network latency because each call typically still gets evaluated separately on the server, with a separate data store add for each one:

Total Time = (network latency) + ((# of items) * (time to add 1 item to data store))

A better solution is to allow for true multi-item create, update, and delete. With this method, youcan send multiple items in a single feed, and the web service updates them in the data storein a single atomic operation:

Total Time = (network latency) + (time to add N items to data store)

Adding 20 items to a data store in a single operation is significantly faster than adding20 individual items separately.

DEVELOPER SUPPORT PRACTICES

APIs should be easy to develop against, and they should make iteasy for you to get help if youget stuck.

Browser-basedAPI Client

It can be tricky to build a REST URL for an API call. That's why companies like Flickr and Twitter have pages that offer help in accomplishing this.

It's best to look for a system where the data services have a built-in web client to help the user build your API calls. For example, in thePlatform's services, you can go to the root of anydata service and add ***/client***., This will return an HTML page that lets a programmer construct ad hoc REST calls and test functionality.

Supported API Clients

When you see notes on a technology vendor's web site like "API clients are not maintained or supported and are used at your own risk," or documentation that points you to community forums for support, you know that API clients have gotten short shrift. They've either been abandoned or crowd-sourced.Either way, if there are bugs in them, you'll need to depend on the community or fix them yourself.

Instead, you should look for a technology vendor whose API clients are maintained and officially supported. Optimally, these clients should get built as part of the core services, and not as an add-on by a different team.

For example, if you're a Java programmer using thePlatform's services, we provide JAR bundles with Java classes that implement calls to the service APIs, which can be downloaded from our Technical Resource Center. We also provide client DLLs for .NET.

Finally, if you're using another framework, you can use the web client to compose your particular REST calls and then add the URLs to your code.

API Documentation and Support

Everything in the system's APIs should be documented and available through an online technical resource center.

It's also important that any API have a team of support engineers who are trained in the API and capable of resolving even the most difficult problems.

CONCLUSION

Today, web service APIs are critical for enabling content companies and TV service providers to build systems that can toleratecontinuous change inIP-connected technologies and consumer behavior.

Despite the widespread adoption of APIs in video management systems, not every system is equal, and service providers must evaluate them carefully to ensure theyaccommodate the needs of developers working with a wide variety of technologies, partners, and types of content.This paper explores a baseline of characteristics that any robust API should provide, but only through a detailed evaluation of a system based on aprovider's unique requirements canyou fully determine the suitability of any technology.