MK blog

iOS development blog and Usability Guidelines

search this site...

RESTful API Server – Doing it the right way (Part 2)

Posted by Mugunth Kumar on Apr 2, 2012 in Articles, Featured Articles

In the part 1 of the post, I introduced the RESTful principle and explained how to architecture your server code so as to ensure easier versioning and deprecation of your API. In this part, I'm going to talk briefly about HATEOAS and hypermedia and then show you the role it plays in a native mobile client development. But the crux of this post is going to be centered around how to implement caching (or rather server side support for caching). Target audience include, server developers and to some extent, iOS or any mobile platform developers.

Contents

HTTP APIs, REST and HATEOAS

In today's world, HTTP APIs can be classified into

- Web Services
- RPC URI Tunnelling
- HTTP-based Type 1
- HTTP-based Type 2
- REST

Here is a very good explanation by Jon Algermissen. Unfortunately, every API provider calls their service as "RESTful" despite otherwise.

So what really is a RESTful server? Any API server that is hypertext/hypermedia driven is RESTful. That means, the developer/client application should be able to discover "other available resources" from the API's root URL. This, is in fact the most important constraint for implementing RESTful APIs. Additionally, a pure RESTful server is one that adheres to HATEOAS constraint.

HATEOAS – Hypermedia as the Engine of Application State

The two main constraints that you should follow for implementing HATEOAS are,

1. Serve only hypermedia resources.

A hypermedia resource is one that contains content and controls (hyperlinks) to other hypermedia resources. JSON (application/json) is NOT a hypermedia resource. (Again, there are a lot of RESTful, HATEOAS servers that serve JSONs) However, you can add additional keys to the JSON making it behave like a hypermedia resource. (For ex: a href key for linking the resource to an associated thumbnail image)

2. Single point of entry for the client application.

From the home page of the API, subsequent "GET" endpoints are embedded as URLs (or links) and subsequent "POST", "PUT" or "DELETE" endpoints are embedded as forms.

The main advantage of adhering to these two HATEOAS constraint is easier documentation.

HATEOAS for your next API?

A big NO
Why do I say this? In the past, APIs were predominantly written for consumption in a web based application. These servers normally serves XHTML and applications run on a web browser. A web browser is a "client", much similar to your mobile (iOS/Android/Windows Phone) client that can parse hypermedia resources and understand HTML forms and "knows" how to present a form to the user. For mobile client implementations, when your RESTful server's resource has a form that you could submit, you can't (at least easily) convert it to a native control

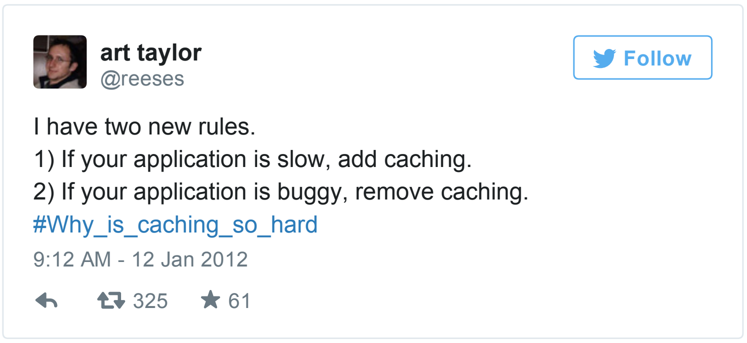
available on the platform. None of the platform vendors, Apple/Google or Microsoft has built in support for converting a XHTML form to a UIViewController (on iOS) or a Intent(on Android) or a Silverlight Page(on Windows Phone).

My recommendation is to go with resource representation for all "GET" entries and expose controller endpoints (instead of forms) for all POST/PUT and DELETE entries. (Example: /friends/add or /venue/checkin) The controller endpoints can (should) be embedded in other responses (so that the client or the client developer can discover it). Of course, that breaks REST, but it's ok. It's better to make products that work, than adhering to standards that offer zero benefits. (This will probably make our API a HTTP-based Type 2 API).

Caching

Let's talk about caching. Caching, as you might think, is predominantly a client side (or intermediate proxy) thing. But do you know that a little bit of support from the server side, you can ensure that your API server adheres to conventions expected by a intermediate caching proxies? This in turn means, you will be benefited from free load balancing provided by them. In fact, caching is described in section 13 of the HTTP spec. Here is a link to it.

Art Taylor, tweeted this couple of months ago.



art taylor
@reeses

Follow

I have two new rules.
1) If your application is slow, add caching.
2) If your application is buggy, remove caching.
#Why_is_caching_so_hard

9:12 AM - 12 Jan 2012

325 61

But believe me, caching can be done just right without getting into trouble like this. Two important principles I would recommend that you adhere to are,

1. Don't rollout custom caching schemes on your client.

2. Understand the basic caching principles explained in HTTP 1.1 RFC specification. The RFC explains two kinds of caching models. Expiration model and Validation model.

In any client/server application, the server is the authoritative source of information. When you download a resource (a page or a response) from a API server, the server sends some "hints" to the client on how to cache them. The server authoritatively dictates if (and when) the client should expire the cache. These "hints" can be sent either programmatically or by a server side configuration. Expiration model is usually a configuration on the server (nginx.conf or equivalent) where as validation model requires some programming effort from the server developer. The server developer should determine when to use validation and when to use expiration based on the kind of resource. Expiration model is normally used when the server can make a reliable guess on how long will a given resource be valid. Validation model is used for everything else. Later, I'll show you how to code your server for both these models. Once you understand the two models, I'll explain when to you use which model.

Expiration model

Let's first look at a very common cache control configuration. If you have configured nginx, you would have added something like this to your nginx.conf. (The Apache equivalent is written into the .htaccess file.)

```
location ~ \.(jpg|gif|png|ico|jpeg|css|swf)$ {
                expires 7d;
        }
```

nginx translates this config to an equivalent HTTP cache header. In this case, the server sends a "Expires" or "Cache-Control: max-age=n" header for all images (amongst others) and expects the client to cache them for 7 days. This means, you normally don't have to re-request the same content again within the next 7 days. Every major web browser (and intermediate proxies) respects this header and works as expected. Unfortunately, a vast majority of the open source image caching frameworks for iOS, including the popular SDWebImage, uses a built in cache control mechanism, that deletes images after n days. The problem here is, these frameworks don't cater to validation model and your mobile client application that uses these frameworks must resort to hacks.

Let me show you an example where this could go wrong. Lets' go back to our "next Facebook" app. When your user uploads an avatar image, he expects to see the changes propagated on all the views. Some clever developers empty the cache after the call to update-profile-image succeeds. (This means, every view controller now have to load the content from the server again). Everything goes well and you have successfully cheated your project manager and every view controller now shows the latest profile picture. However, it doesn't eliminate the problem completely. The user's new profile picture is visible to his friends only after 7 days. Clearly unacceptable. So how do you solve this? As I already told, you have to accept the fact that the server is the only authoritative source for the latest data. Don't use dirty tricks on the client to expire the cache.

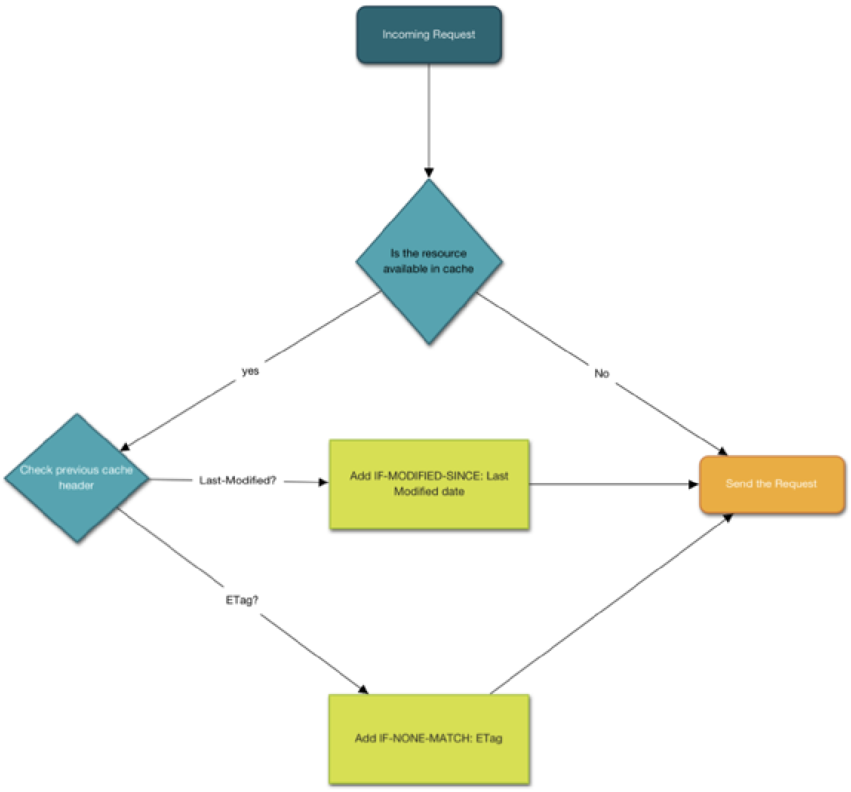
Validation model

Both Facebook and Twitter solve the problem of expiring profile images (after a new image is uploaded) using validation model. In a validation model, the server sends an identifier unique to the requested resource and the client caches both the identifier and the response. In HTTP parlance, this unique identifier is called as an ETag. When you make a second request to the same resource, you should send this ETag. The server uses this identifier to check if the resource you requested has changed (remember, the server is **the** authoritative source). If the resource has indeed changed, it sends you the latest copy. Otherwise, it sends a 304 Not Modified. Validation model requires programming effort on both client and server. I'll walk you through both in the next section.

Client side coding

As a matter of fact, on iOS, if you use MKNetworkKit, it does all these automatically for you. But for the sake of Android and Windows Phone developers, I'll explain how this should be implemented.

The validation model uses **ETag** and **Last-Modified** HTTP Headers. Client side support for validation model is easier than server side implementation. If you received a ETag with a resource, when you make a second request to the same resource, send the ETag in "IF-NONE-MATCH" header. Alternatively, if you received a "Last-Modified" with a resource, send it in "IF-MODIFIED-SINCE" header on subsequent requests. Again, the server determines when to use "ETags" and when to use "Last-Modified"


Incoming Request
Is the resource available in cache
yes
No
Check previous cache header
Last-Modified?
Add IF-MODIFIED-SINCE: Last Modified date
Send the Request
ETag?
Add IF-NONE-MATCH: ETag

Implementing expiration model is easy. Just calculate the expiry date based on the headers, "Expires" or "Cache-Control:max-age-n" and delete the cache after this day.

Server side coding

ETag based caching

ETags are usually calculated on the server using object hashing algorithms. (Most high level server side languages like Java/C#/Scala has methods to hash an object). Before serializing the responses, the server should compute the object hash and add this value to the header as ETag. Now, If the client has indeed sent a IF-NONE-MATCH in the request and that ETag matches with what you have computed, send a 304 Not Modified. Otherwise, serialize your response and send the new ETag.

Last-Modified based caching

Last-Modified implementation is slightly tricky. Let's assume that you have an endpoint that points to thelist of friends resource.

http://api.mynextfacebook.com/friends/

When using ETags, you computed the hash of the array of friends. If you are using Last-Modified, you should send the Last-Modified date of this resource. Because this resource is a list; this date could actually mean the date when the user last accepted a new friend. It requires the server developer to store the last modified date for every user in the database. Slightly tricky than ETags, but there is a huge performance advantage. When the client requests this resource for the first time, you send the complete list of friends. Subsequent requests from client will now have a "IF-MODIFIED-SINCE" header. Your server should be programmed to send only the list of friends that are added after this date. The database fetching pseudo code that was previously something like this,

select * from friends

becomes,

select * from friends where friendedDate > IF-MODIFIED-SINCE

If the database select returns zero results, send a 304 Not Modified. So if the user has 300 friends but only two of them are new friends, serialize and send only those two records back to the client. Your database fetching time and the resource payload is reduced by a huge margin.

Of course, this is a super simplified pseudo code. Server developer will probably have a headache when you support unfriending (deletion) or blocking friends. The server should be able to send hints using which the client should be able to tell which all friends were newly added and which all were deleted. This technique in fact requires additional effort on the server side.

Choosing a caching model for your resource

Whew! That was some heavy stuff. I'm now going to establish some ground rules on when to use what type of caching mechanism.

1) All static images should be served using a expiration model.

2) All dynamically changeable data should be served using a validation model.

3) If your dynamic data is a list, you should use Last-Modified based validation model. (For example: /friends).
Other wise, you should use ETag based validation model (For example: /friends/firstname.lastname

4) Images or any resource that might be updated by the user (like profile image) should again use the ETag based validation model. Though they are images, they are not static in your application (like your company logo). Moreover, you cannot reliably calculate the expiry date for such a resource.
The other (slightly easier to implement but a bit hackish) way is to use a URL fault. URL fault works like this.
When you send a avatar URL embedded in a resource, make sure that, some part of the URL is dynamic. That is instead of representing an avatar URL as
http://images.mynextfacebook.com/person/firstname.lastname/avatar

represent it like this
http://images.mynextfacebook.com/person/firstname.lastname/avatar/<randomhash>

Ensure that the random hash changes when the user updates the image. The API that sends the list of friends resource will now have a different URL for friends who updated their image. Changes to profile images happen almost instantly!

If both server and client adheres to already established caching standards, your iOS app and your product can "fly". What I did in this post is a mere explanation of those standards which aren't followed by a vast majority of developers.

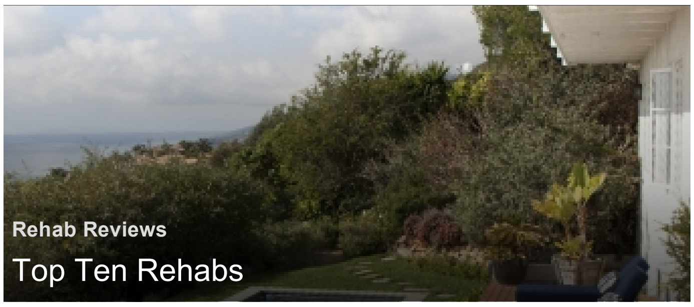
That ends this part of the post. In the last and final part, we will discuss about how to communicate errors and do error handling properly and support internationalization for your application.

Recommended Reading

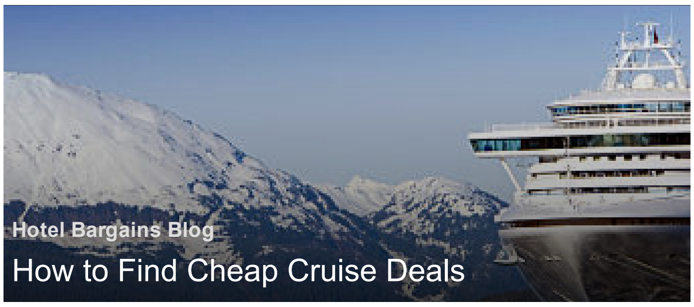
REST API Design Rulebook, By Mark Masse
http://shop.oreilly.com/product/0636920021575.do

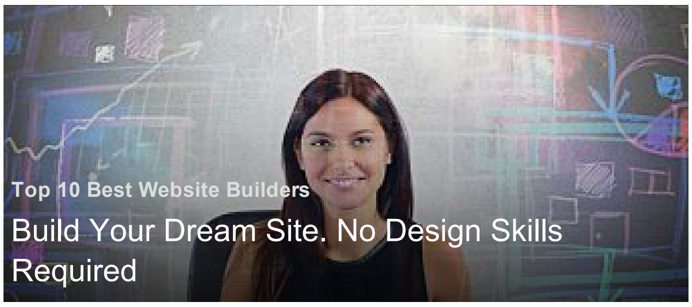
—
Mugunth

Follow me on Twitter



Web2Carz.com
The 2016 Honda Pilot: Honda's Three-Row SUV, Back With a Vengeance




Top 10 Best Website Builders
Build Your Dream Site. No Design Skills Required

4 Comments MK Blog 1 Login

Recommend 3 Share Sort by Best

Join the discussion…

Email · 3 years ago
When we can expect final part?
5 · Reply · Share ›

Gary Bisaga · 9 months ago
Is this really a valid use of these headers? According to the definition of this header (RFC 2616 section 14.25):

"The If-Modified-Since request-header field is used with a method to make it conditional: if the requested variant has not been modified since the time specified in this field, an entity will not be returned from the server; instead, a 304 (not modified) response will be returned without any message-body."

It seems to me that you are advocating using this header in a different way, to modify the returned representation. Now, I fully understand why you want to do this. I have a need myself, which is how I ended up on this page. However, I am concerned that we would be using these headers in a way different from what clients expect. Have you ever seen anybody write an API this way?
· Reply · Share ›

Yo Dirkx · 3 years ago
This is not very stateless and I expect very bad:

select * from friends

becomes,

select * from friends where friendedDate > IF-MODIFIED-SINCE

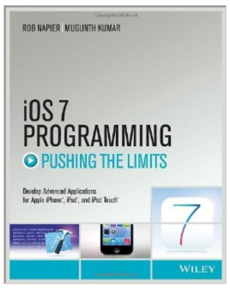
because the exact same resource would return different values depending on caching headers. A resource should return the full resource, not just 2 of the 300 friends.
· Reply · Share ›

Yo Dirkx → Yo Dirkx · 3 years ago
It's also bad because a friend list has much more changes than the friendedDate. A changes friend name for example. Only ETag is reliable then, but if the server has to run all the logic and queries to verify/check an ETag, I don't see the point (unless the response would be huge).
· Reply · Share ›

Subscribe Add Disqus to your site Privacy DISQUS

Get My Book

Want to take your iOS apps to the next level? Then this book is for you.



Socially Speaking

Recent Posts

- On Apple rejecting SeaNav US
- The problem with Cocoapods
- Radar 15159094: UITextView + NSAttributedString is hopelessly broken on iOS 7
- iOS-Framework: Einführung in MKNetworkKit (German Translation of MKNetworkKit documentation)
- MKBlockTimer

Hire

I'm available for new projects. Please visit my hire me page for more details.