RESTful API Server – Doing it the right way (Part 1)

Posted by Mugunth Kumar on Mar 1, 2012 in Articles, Featured Articles

In 2007, Steve Jobs announced the iPhone that revolutionized the technology industry and changed the way we work and do business. It is 2012 now and increasingly, more and more websites are offering native iOS and Android clients as front ends to their service. Not all startups have the funding to develop apps in addition to their core product. To increase the adoption rate of their product, these companies, release a public API that developers can use to build apps on top. Twitter was probably the first company to be "API first" and now increasingly more number of companies are following this strategy as it is really a great way to build an ecosystem around your product.

Startup life is full of pivots. If you code base cannot support the pivoting decisions you make, you lose. A server code that is nimble enough to adapt to business needs, decides the make or break of a startup. Successful startups are not those that come up with great ideas, but those that are good at executing them. Success of a startup depends on the success of their product, whether it's their iOS app or their service or their API. In my past 3 years, I've worked on a variety of iOS apps (mostly for startups) consuming web services and in this blog, I've tried to consolidate my knowledge and show you the best practices that you should adopt when developing a RESTful API. A good RESTful API is one that is not resistive to changes.

Contents

Target Audience

This blog post is targeted for readers who have intermediate to advanced knowledge in developing RESTful APIs and some basic knowledge of any object oriented (or functional) server side programming language like Java/Ruby/Scala. (Note: I intentionally ignored PHP or Programmable Hyperlinked Pasta)

Structure and Organization

The post is quite detailed and the first part explains basics of REST and the second part explains documenting and versioning your API. The first part is for beginners. The second part is for pros. I know, you are a pro. So, here is a link to jump ahead to API documentation section right away! This is probably where you should start if you think this post is a tl;dr.

RESTful constraints

A RESTful server is one that conforms to the REST constraints. Here is a Wikipedia article on REST. When you develop a API that would predominantly be consumed by a mobile device, following and understanding the three most important constraints would be helpful, not just in developing the API, but in maintaining it and making changes moving forward. Let me explain.

Statelessness

The first constraint is statelessness. Put in simple words, a RESTful server should not contain contextual information about the client. A client, on the other hand, can maintain context of the server's state though. In other words, you shouldn't make your server remember the state of a mobile device using an API.

Let's imagine that your startup is the "next Facebook". A good example of where a developer would potentially make such mistakes is to expose an API that allows a mobile device to set the last read item on a stream (say a Facebook feed). API endpoint that normally returns the feed (say /feed)

will now return items that are new after this last read item. Sounds clever right? You "optimize" the data transfer between the client and server right? Wrong.

What could possibly go wrong in this case is when the user accesses your service from two or three devices and one device sets the last read status and the other device doesn't have a way to download items that was already read on other devices.

> Stateless means, the data returned for a specific API call should not be dependent on calls that are made before it.

The right way to optimize this call, is to pass the timestamp of the last read feed item to the server along with the API call that returns the feed (/feed?lastFeed=20120228). There are other, more standard way of doing this using HTTP Modified Since header. But we will leave that for now. I'll discuss this in part 2 of this post.

The client, however can (should) remember access tokens generated on the server and send it other APIs that require them.

Cacheable and a Layered architecture

The second constraint is to provide a certainty to the client that a response can be cached and reused for a set period of time without making round trips to server. This client can be the real mobile client or any intermediate proxy server. I will explain more about caching later in part 2 of this post.

Client-server separation of concerns and a uniform interface

A RESTful server should abstract and hide away as much implementation details as possible from the client. That is, the client shouldn't bother (or know) about what database the server uses or how many load balancers are currently active and other such stuff. Maintaining a good separation of concerns helps in scaling when your product becomes "viral".

That's probably the three most important constraints you should have in mind while developing a RESTful server. There are three other minor constraints, but they all overlap with what we discussed here.

REST Requests and the four HTTP methods

- GET
- POST
- PUT
- DELETE

Cacheable constraint and GET requests

Key takeaway from this is, GET method doesn't alter the server state. This inherently means your requests could be cached by any intermediate proxies (think: reduced load). So as a server developer, you shouldn't expose a "GET" method that updates data in your database. It breaks RESTful philosophy specifically the second constraint I talked about earlier. Your "GET" methods should not even update a access log or insert a record that maintains the "last logged in" time. If you are updating your database, it should be always be a POST/PUT method.

That POST vs PUT debate

The HTTP 1.1 specification says, PUT is idempotent. This means, the client can make multiple PUT requests to the same URI and yet doesn't create/update duplicate records.

Assignment operations are a good example of idempotent operation.

```
String userId = this.request["USER_ID"];
```

Even if this operation is executed twice or three times, there is no harm (other than lost CPU cycles).

POST on the other hand is **not** idempotent. It's like an increment operator. You should use POST or PUT based on whether or not the action performed is idempotent. In programmer's parlance, if the client "knows" the URL of the object that would be created, you should use PUT. If the client knows the URL of the creator/factory, use POST.

```
PUT www.example.com/post/1234
```

Use PUT if the client knows the URI that would be created as a result of the call. Even if the client calls this PUT method multiple times, there is no harm or no duplicate records created.

```
POST www.example.com/createpost
```

Use POST if the server creates the unique key and sends results back to the client. Duplicate records will be created when the call is repeated later on with the same parameters.

Read this answer on Stackoverflow for more.

DELETE method

DELETE is straight forward. It's again idempotent like PUT, and should be used to delete a record if it is present.

REST Responses

Responses from your RESTful server can either use XML or JSON. Personally, I would prefer JSON over XML as JSON is less verbose and data transferred is usually less compared to the same response in XML format. The difference might be in the order of a few hundred kilobytes, but given the speed of 3G networks and intermittent mobile data connectivity, these few hundred kilobyte changes can have a huge impact when downloading

the response data.

Authentication

Authentication should be done over https and the client should send the password encrypted using some cryptographic algorithm. Getting a sha1 hash of a NSString in Objective-C is fairly straight forward and the following code illustrates this.

```
- (NSString *) sha1
{
        const char *cstr = [self cStringUsingEncoding:NSUTF8StringEncoding];
        NSData *data = [NSData dataWithBytes:cstr length:self.length];

        uint8_t digest[CC_SHA1_DIGEST_LENGTH];

        CC_SHA1(data.bytes, data.length, digest);

        NSMutableString* output = [NSMutableString stringWithCapacity:CC_SHA1_DIGEST_LENGTH * 2];

        for(int i = 0; i <; CC_SHA1_DIGEST_LENGTH; i++)
                [output appendFormat:@"%02x", digest[i]];

        return output;
}
```

The server should match the encrypted password with the encrypted password stored previously on the server. In any case, you should never transfer passwords in plain text from the client to the server. There is NO EXCEPTION to this rule. The day your users come to know that you are storing passwords as plain text will probably be the day your startup dies. Trust that is once lost can never be gotten back.

RFC 2617 specifies two ways to authenticate with a HTTP server. The first is Basic Access Authentication and the second is Digest Authentication. For internal mobile client use, Basic or Digest authentication is sufficient and most server side (and client side) languages have built in mechanism for implementing this authentication scheme.
If you are planning to make your API public, you should consider using oAuth or better oAuth 2.0. oAuth allows your end users to share the content created within your application with other third party vendors without handling over the keys (username/password). oAuth also allows user to be in full control over what is shared and what is rights do the requesting third party application has.

Facebook Graph API is, by and large, the biggest implementation of oAuth to date. By using oAuth, a Facebook user can share photos with a third party application without sharing other personal information and his access details (username/password). A user can also revoke access to a “rogue” third party application without changing his password.

So far, I talked about the basics of REST. Now lets dive into the meat of the post. In the subsequent sections I’ll talk about best practices that you should follow when documenting, versioning and deprecating your API.

API documentation

The worst documentation that a server developer could write is the one that contains a laundry list of API endpoints, the parameters that are required and the corresponding response from the server. The problem with this is, it’s too difficult to make incremental changes to the server and changing response data as the product evolves becomes a nightmare. I’m proposing some suggestions on how to document your API so that a client developer would understand you better. In the course of time, that will also help you in becoming a better server developer!

Documentation

The first step, I would recommend, is to start thinking about your top level model objects before you start the documentation. Then think about actions that can be done on these objects. The foursquare API documentation is a good example to start with. They have a set of top level objects like venues, users and so on. They also have a set of actions that can be performed on these objects. Once you know the top level objects and actions in your product, designing the endpoints becomes easier and clearer. For example, to “add” a new venue, you would probably have to call a method similar to /venues/add

Document every member of the top level objects in your documentation. Next, document your request and responses using these top level objects rather than raw primitive data types. Instead of writing, this API would return three strings, the first being the id, second name and third description, write that this API would return a venue model.

Documenting Request Parameters

Let’s assume that you have a API that allows the user to login with a Facebook token. Let’s call that api as /login.

Request

/login

Headers

Authorization: Token XXXXX
User-Agent: MyGreatApp/1.0
Accept: application/json
Accept-Encoding: compress, gzip

Parameters

Encoding type – application/x-www-form-urlencoded

token – "Facebook Auth Token" (mandatory)

profileInfo = "json string containing public profile information from Facebook" (optional)

This profileInfo is a *top-level object*. Since you have already documented this object's internal structure, mentioning this alone would suffice.

If your server uses the same Accept, Accept-Encoding and parameter encoding, you can document it separately instead of repeating them everywhere.

Documenting Response Parameters

Responses from API should be documented based on the top level model objects. Quoting from the same foursquare example, the /venue/#venueid# method returns a complete venue model.

In case your model is big and you want to reduce the payload, consider creating a *compact model*. You should use this for APIs that return a list of model objects. Foursquare's API does this as well. Their search API returns an array of compact venue

Exchanging ideas, documenting or letting other developers know what you will return has just gotten easier when you document your API using model objects. The most important takeaway from this section is to treat this document as a contract between you, the server developer and client developers (iOS/Android/Windows Phone/Whatever)

Reasons to version and deprecate your API

Prior to mobile applications, in the era of Web 2.0 applications, API versioning was never a problem. Both the client (Javascript/AJAX front-end) and the server was deployed at the same time. Consumers (your customers) always use the latest front-end client to access your system. Since you are the company that writes both the client and server, you have full control over how to use your API and changes to the API can always be implemented immediately on the front-end. Unfortunately, with native clients this is not possible. You might deploy API version 2 assuming everything will go well, but will blow up on older versions of your iOS apps because there would be still users using the older version of iPhone app even after you pushed an update through App Store. Some companies resort to using push notification to pester users to update their app. This will only end up in losing that customer. I have seen many many iPhones that have more than a 100 app updates pending. There is a pretty good chance that your app might be one of them. You should always be prepared for versioning the API and deprecate them as and when it's proper to do so. But do support your APIs for at least three months.

Versioning

Deploying your server code on a different directory and using a different URL endpoint doesn't automatically mean you have effectively migrated your server code.
That's
http://example.com/api/v1 will be used by version 1.0 of the app and your latest and greatest version 2.0 of the app will use
http://example.com/api/v2

When you make an update, you almost and always make changes to internal data structures, and model objects within your server. That includes changes to the database (adding or removing columns). To make things clear, let's assume that your "next Facebook" app has a API called /feed that returns "Feed" objects.
Today, as of version 1, your Feed object contains a URL to a person's picture (avatarURL), the person name (personName) the feed entry text (feedEntryText) and the timestamp (timeStamp) of the news entry.
Later on, in your version 2, you introduce a feature where you allow advertisers to market their products in the feed. Now, your feed object contains, let's say, a new field called "sourceName" that super cedes person name on the UI. That's, the app should display "sourceName", instead of "personName". Since the UI no longer need to display personName when "sourceName" is present, you decide not to send "personName" when "sourceName" is present. This all sounds good till the older version, version 1 of your application hits your newly deployed server. It starts displaying your advertised entries without a title since "personName" is missing. A "clever" way of handling this, is to send both "personName" and "sourceName". But, my friend, life isn't always that easy. As a developer, you can't keep track of every single change that has ever been made for every single model object in your class. It's just not an efficient way of doing it and 6 months later, you will almost forget why *something* was added to your code.

Thinking back, in web 2.0, this wasn't a problem at all. The Javascript front-end would have been immediately updated to cater to the API changes. However iOS apps are disconnected unlike a web application. It's the user's prerogative to update it.

I have a very elegant solution to propose for this kind of tricky situation.

Versioned URL paradigm

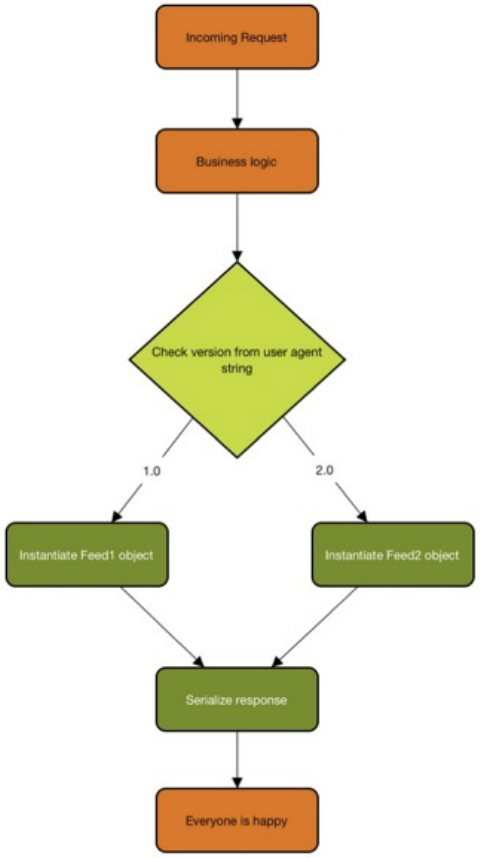
First is to differentiate multiple versions using the URL.
http://api.example.com/v1/feeds will be consumed by version 1 of the iOS app and
http://api.example.com/v2/feeds will be consumed by version 2 of the iOS app.

While this method sounds good, you can't go on creating duplicate copies of your deployed code base for every single change you make to the output format. I recommend this only when you make a huge breaking release/change. For minor changes, consider versioning your models.

Versioned model paradigm

I showed you how to document your models a while ago. Consider this document as a contractual agreement between the server and client developer. You should never make a change to this model without changing the version. This means, in our previous case, there would be two models, Feed1 and Feed2.
Feed2 has sourceName and outputs sourceName and removes personName when sourceName is present.
Feed1 behavior remains same like how it was agreed upon when documented.

The request controller code flow will look closely similar to this.



RESTful controller pseudo-code flow chart

You should consider moving the instantiation code into the class as a factory method. The controller pseudo-code should look similar to this

```
Feed myFeedObject = Feed.createFeedObject("1.0");
myFeedObject.populateWithDBObject(FeedDao* feedDaoObject);
```

Whether it is 1.0 or 2.0 is decided by the controller from the UserAgent string.

Update:
Rather than depending on version numbers in UserAgent string, the client should send the version number in Accept header.
So instead of sending

Accept: application/json

you should send

Accept: application/myservice.1.0+json

This way, you have the ability to request a different version of response object for every REST resource you request.
Thanks for hacker news readers who sent this to me.

The controller asks the Feed factory method to create the correct feed object based on the incoming request (all requests have UserAgent that looks like AppName/1.0) and based on the version of the client. When you implement your server like this, *any* change is easy. Making a change to your server without breaking existing contracts will be a breeze. Just create new models, make change to the factory method to instantiate this new model for newer versions and you are set to go!

With this architecture in place, your version 1 and version 2 of the app can still talk to the same server. Your controller will render the version 1 object to the older client and version 2 object to the newer client.

Deprecation

With the versioned model paradigm I proposed, deprecating your API gets a lot more easier. This is very important when you make your API public at a later stage.
When you make a major version update, cleanup all the factory methods in your models based on your business decisions.
If you decide not to support version 1 of the iOS app with the release of version 3 of the API, remove the associated models, remove the lines that instantiate version 1 of the model (in your factory method) and you are good to go.

Versioning and deprecation goes a long way in ensuring the longevity of your company and the product by ensuring that you will always be nimble enough for most of the pivoting decisions made by the business owner. Businesses die when they cannot pivot. Usually resistance to pivoting comes internally from the technical team. This technique should solve that problem.

Caching

The next important performance improvement that you should focus on when you build an API is to support caching. If you are like most other and think caching is a client side thing, think again. Part 2 of this blog post explains how to support caching based on HTTP 1.1 standards.

Error handling & Internationalization of your API

Notifying your client of the kind of errors that happened on server is as important as sending the correct data. I'll explain about error handling and Internationalization of your API in part 3 of this post. Not making any promises, but this will surely take some time.

—
Mugunth

Follow me on Twitter

No related posts.

Luke Stokes · 3 years ago

Great post!

What are your thoughts on using the X-API-VERSION header along with a version tag on the individual resource media types instead of a url version? As an example, the API I'm working on has a required header of X-API-VERSION and each resource also has their own version information such as vnd.foxycart.com.user.v1+json. The flexibility this gives us is that if we need to add or change an aspect of a resource for a really important client, we can just create a new resource likeso: vnd.foxycart.com.user.v1.1+json. Then later down the road if we have a business requirement change (i.e. the resource didn't actually change, but creating one might do something different like send an email or update data somewhere else), we can then bump up the X-API-VERSION to 2 and roll our v1.1 media type into a new v2 media type for everyone. Does that sound like a good plan?

MugunthKumar Mod Luke Stokes · 3 years ago

You are right. A couple others also suggested something like this and I updated the post to reflect them.

• Reply • Share ›

Martin Wawrusch ➚ MugunthKumar • 3 years ago

Just stumbled upon your post, very well done. Take a look at this article regarding versioning the mime type: http://www.informit.com/articl...

The gist: There is a not so well know version option for the mime type, like so:
vnd.example-com.foo+json; version=2.0

1 • Reply • Share ›

opyh • 3 years ago

Some thoughts:
- Don't just use _any_ encryption for your password. You should NEVER just hash your password with SHA-1. Think of rainbow tables! Use something like PBKDF2 (http://en.wikipedia.org/wiki/P... with an application-specific (or better user-specific) random salt.
- Don't use that versioning antipattern. Use this instead: http://blog.steveklabnik.com/p...

2 • Reply • Share ›

Graham Lee ➚ opyh • 3 years ago

The hash algorithm is a bad choice (PBKDF2 is great for key generation, bcrypt is great for hashing), but the hash code also contains a bug that's likely to lead to a crash. How long is the string @"💩";? Foundation thinks it's two unichars (which it is, even though it's only one character) and the C library thinks it's four chars (which it also is). The conversion to an NSData is not going to work.

1 • Reply • Share ›

MugunthKumar Mod ➚ opyh • 3 years ago

I wrote about two versioning patterns. Which one did you mean when you said don't use that versioning anti pattern?

1 • Reply • Share ›

Jason Moore ➚ MugunthKumar • 3 years ago

The linked article states: "The first thing that people do when they want a versioned API is to shove a /v1/ in the URL. THIS IS BAD!!!!!" So, looks like he doesn't like the version in the URL. ;)

But, I think it has a use. If the API changes drastically, say from '/login' to '/user/login', then it would be cleaner in the code to branch off to a different set of logic for 'v1' or 'v2' - and later (to your point) it would be easier to just remove the v1 code altogether.

1 • Reply • Share ›

MugunthKumar Mod ➚ Jason Moore • 3 years ago

That's is exactly what I wrote. Use URL versioning only when you make a huge breaking release/change

2 • Reply • Share ›

Martin Wawrusch ➚ Jason Moore • 3 years ago

If you change from /login to /user/login then you should respond with a 301 moved permanently. Versioning other than at the resource level through mime type version is really evil and one the reasons why the API ecosystem is in a bad shape.

1 • Reply • Share ›

Lars • 3 years ago

Why do you want to hash/encrypt the password if you're using HTTPS?

2 • Reply • Share ›

Sean ➚ Lars • 3 years ago

What if the server is compromised? A hacker could easily see the plain text password if they had access to the server

3 • Reply • Share ›

Brais Gabín ➚ Sean • 3 years ago

If you send the encrypted password and this is what you store in DDBB when a hacker access to your server may copy the DDBB. The hacker has all the information necessary to login as any of your users at any time even if you fix the security hole. But if you send the password in plain text and the server stores the password encrypted, the hacker may only do harm while the security hole persis.

4 • Reply • Share ›

ismriv ➚ Brais Gabín • 2 years ago

This is clearly the way to go.

One more remark: you shouldn't be using SHA-(1|2) for encrypting the passwords, but a more expensive cryptographic algorithm such as bcrypt. And think of salting them as well.

1 • Reply • Share ›

valugi • 3 years ago
PHP hater...
1 • Reply • Share ›

Mansour • 3 years ago
Hmm, if the authentication is over HTTPS, what is wrong with plain password? After all, when you sign in to a website in your browser, passwords are sent plain. Server doesn't have to store the it plain, it can hash what the client sends and compare it with hash stored in db (to protect users who use one password for several websites).
1 • Reply • Share ›

Shibukoden • 9 months ago
Thank you. This saves lot of time.
• Reply • Share ›

Stavros • 2 years ago
Thank you for the article, ti is very helpful!

In the section of "Cacheable constraint and GET requests" you mention that we should not make any update to the DB even the last login time. Do you have any suggestions on that? What is a good practice to follow?
• Reply • Share ›

Shideon • 3 years ago
You have good points but you didn't mention hypermedia as the engine of application state (HATEOAS). It's one of the most important parts of REST and the most overlooked.
• Reply • Share ›

MugunthKumar Mod ➚ Shideon • 3 years ago
It's in the Part 2 of this post.
1 • Reply • Share ›

MugunthKumar Mod ➚ Shideon • 3 years ago
It's in the Part 2 of this post.
• Reply • Share ›

John Yeglinski • 3 years ago
Cant wait for the next installment in this series!!!
• Reply • Share ›

Jsjdjs • 3 years ago
/login is a verb in URL. You should PUT to /session/id
• Reply • Share ›

laeagle ➚ Jsjdjs • 3 years ago
"login" is a noun. "Log in", or "logIn" in camel case, is a verb.
• Reply • Share ›

Steve Klabnik ➚ Jsjdjs • 3 years ago
As far as REST is concerned, URIs are opaque. It's totally irrelevant what the URI is.

Also, some people use 'login' as a noun.
• Reply • Share ›

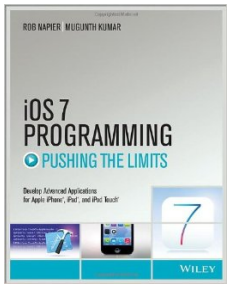
NewmanOZ • 3 years ago
Thank you very much, it was very useful and easy to understand.
This article will be my instruction today for API refactoring issues :)
• Reply • Share ›

Dan • 3 years ago
Why did you intentionally ignore PHP?
• Reply • Share ›

jarod stewart • 3 years ago
What's wrong with PHP?
• Reply • Share ›

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.