Web API Checklist

Thursday, July 12, 2018 3:21 PM

The Web API Checklist -- 43 Things To Think About When Designing, Testing, and Releasing your API

Posted on April 15, 2013

When you're designing, testing, or releasing a new Web API, you're building a new system on top of an existing complex and sophisticated system. At a minimum, you're building upon HTTP, which is built upon TCP/IP, which is built upon a series of tubes. You're also building upon a web server, an application framework, and maybe an API framework.

Most people, myself included, are not aware of all the intricacies and nuances of every component they're building upon. Even if you deeply understand each component, it's probably going to be too much information to hold in your head at one time.

"We know there are known knowns: there are things we know we know. We also know there are known unknowns: that is to say we know there are things we know we don't know. ***But there are also unknown unknowns – the ones we don't know we don't know.*** *" –Defense Secretary Donald Rumsfeld, Defense Department briefing, Feb 12, 2002*

I don't want you to build an API and only know about your unknown unknowns when they bite you in the ass. So, here's a list of a bunch of things, both obvious and subtle, that can easily be missed when designing, testing, implementing, and releasing a Web API.

HTTP

The HTTP 1.1 specification, RFC2616, is a hefty document at 54,121 words. Here are some select items from the spec that might affect your API design:

#1. Idempotent methods – GET, HEAD, PUT, DELETE, OPTIONS and TRACE are all intended to be idempotent operations; that is, "the side-effects of N > 0 identical requests is the same as for a single request." (RFC2616 §9.1.2)

#2. Authentication – Most APIs will need a way to identify and authenticate the user accessing the API. HTTP provides the Authorization header (RFC2616 §14.8) for this purpose. RFC2617 specifies specific authentication schemes, including the most common, HTTP Basic authentication.

Many popular APIs use HTTP Basic Authentication with an API Key as either the username or the password. This is a simple and effective authentication mechanism.

To implement HTTP Authentication accurately, you should provide a 401 status code with a WWW-Authenticate header if a request is not permitted due to a lack of authentication. Many clients will require this response before sending an Authorization header on a subsequent request.

#3. 201 Created – Use the "201 Created" response code to indicate that the request was processed successfully and resulted in the creation of a new resource. 201 responses can include the new resource URI in the Location header. (RFC2616 §10.2.2)

#4. 202 Accepted – Use the "202 Accepted" response code to indicate that the request is valid and will be processed, but has not been completed. Typically this is used where a processing queue might be present in the background on the server side. (RFC2616 §10.2.3)

#5. 4xx versus 5xx status codes – There's an important distinction between 4xx and 5xx status codes: 4xx codes are intended to indicate a client-side error, and a 5xx code indicates a server-side error. Proper usage of these status code classes can help application developers understand whether they did something wrong (4xx) or something is broken (5xx). (RFC2616 §6.1.1) (*Edit:* Read more about the difference at Is "404 Not Found" really a client error?.)

#6. 410 Gone – The "410 Gone" response code is an under-utilized response code that informs the client that a resource used to be present at this URL, but no longer is. This can be used in your API to indicate deleted, archived, or expired items. (RFC2616 §10.4.11)

#7. Expect: 100-Continue – If an API client is about to send a request with a large entity body, like a POST, PUT, or PATCH, they can send "Expect: 100-continue" in their HTTP headers, and wait for a "100 Continue" response before sending their entity body. This allows the API server to verify much of the validity of the request before wasting bandwidth to return an error response (such as a 401 or a 403). Supporting this functionality is not very common, but it can improve API responsiveness and reduce bandwidth in some scenarios. (RFC2616 §8.2.3)

#8. Connection Keep-Alive – Maintaining a connection with your API server for multiple API requests can be a big performance improvement. Pretty much every web server should support keep-alive connections if configured correctly.

#9. HTTP Compression – HTTP compression can be used both for response bodies (Accept-Encoding: gzip) and for request bodies (Content-Encoding: gzip) to improve the network performance of an HTTP API.

#10. HTTP Caching – Provide a Cache-Control header on your API responses. If they're not cacheable, "Cache-Control: no-cache" will make sure proxies and browsers understand that. If they are cacheable, there are a variety of factors to consider, such as whether the cache can be shared by a proxy, or how long a resource is "fresh". (RFC2616 §14.9)

#11. Cache Validation – If you have cacheable API hits, you should provide Last-Modified or ETag headers on your responses, and then support If-Modified-Since or If-None-Match request headers for conditional requests. This will allow clients to check if their cached copy is still valid, and prevent a complete resource download when not required. If implemented properly, you can make your conditional requests more efficient than usual requests, and also save some server-side load. (RFC2616 §13.3)

#12. Conditional Modifications – ETag headers can also be used to enable conditional modifications of your resources. By supplying an ETag header on your GETs, later POST, PATCH or DELETE requests can supply an If-Match header to check whether they're updating or deleting the resource in the same state they last saw it in. (RFC2616 §14.24)

#13. Absolute Redirects – It's a little known requirement of HTTP/1.1 that redirects (eg. 201, 301, 302, 303, 307 response codes) are supposed to contain an absolute URI in the Location response header. Many clients do support relative URIs in Location, but if you want your API to be broadly compatible with many clients, you should use absolute URIs in any redirects. (RFC2616 §14.30)

#14. Link Response Header – In a RESTful API, it's often necessary to provide links to other resources even if the content-type of your response doesn't have a natural way to provide links (for example, a PDF or image representation). RFC5988 specifies a method of providing links in a response header.

#15. Canonical URLs – For resources with multiple URLs, RFC6596 defines a consistent method of providing a canonical URL link.

#16. Chunked Transfer Encoding – If you have large content responses, Transfer-Encoding: Chunked is a great way to stream responses to your client. It will reduce the memory usage requirements (especially for implementing HTTP Compression) of your server and intermediate servers, as well as provide for a faster time-to-first-byte response.

#17. Error Handling in Chunked Transfer Encoding – Before you go and implement chunked transfer encoding, figure out how you're going to handle an error that occurs mid-request. Once you start streaming your response, you can't change your HTTP status code. Typically you'd define a way of representing an error inside your content-type.

#18. X-HTTP-Method-Override – Some HTTP clients can't support anything but GET and POST; you can tunnel other HTTP methods through POST and use the de facto standard X-HTTP-Method-Override header to document the "real" HTTP method.

#19. URL Length – If your API supports complex or arbitrary filtering options as GET parameters, keep in mind that both clients and servers can have compatibility issues with a URL over 2000 characters long.

API Design

#20. Statelessness – There's one place you don't want your API to be storing state, and that's in your application servers. Always keep application servers state-free so that they can be easily and painlessly scaled.

#21. Content Negotiation – If you want to support multiple representations of your resources, you can use content negotiation (eg. Accept headers), or differing URLs for different representations (eg. …?format=json), or you can combine both and have your content negotiation resources redirect to specific formats.

#22. URI Templates – URI Templates are a well-defined mechanism for providing URL composition capabilities to your clients, or for documenting your URL access patterns to your end-user.

#23. Design for Intent – Don't just expose your internal business objects through your API. Design your API to have semantic meaning and to match the use-cases that your users will have. Darrel Miller wrote a great post on API Craft describing this better than I could. (*Edit:* I tried my best in an article entitled Stop Designing Fragile Web APIs.)

#24. Versioning – Theoretically, if you designed a really great API up front, you might never need to create incompatibilities in your API. For the pragmatists in us, put versioning in your API URLs (eg. a /v1/ path), so that you have a safety net in case the API doesn't work out like you wanted. (*Edit:* An expanded justification is my follow-up article: Ain't Nobody Got Time For That: API Versioning)

#25. Authorization – Remember when designing your API that not all users will have access to all objects in the system. It's great if you use or build an API framework with some form of declarative security so that it's easy to assign and modify authorization rights on read and write access to resources.

#26. Bulk Operations – Most clients will perform better if they can issue fewer requests to fetch or modify more data. It's a good idea to build bulk operations into your API to support this kind of use case.

#27. Pagination – Pagination serves two big purposes in an API; it reduces the amount of unneeded data delivered to the client, and it reduces the amount of unneeded computation on your application servers. There are many different patterns used for making paginated collection resources; if you don't know where to start, browse through Stackoverflow for some hints. If you want to be my personal hero, implement **consistent pagination** by providing links to additional pages that are timestamped or versioned, such that you will never see duplicate results in pagination requests even if the objects involved change.

#28. Unicode – It's pretty obvious these days that you need to support more than English characters in a web service; just remember to keep this in mind when designing and testing your API. In particular, Unicode characters can be interesting if you use them as natural keys in a URL (eg. /users/jimbob/ becomes /users/싸이/).

#29. Error Logging – Be sure you design how you want your API to perform error logging, rather than just throwing it together. In particular, I find it extremely valuable to distinguish between errors that are caused by the client's input, and errors that are caused by your software. Keep these in two separate logs.

Content

#30. Content Types – Entire books could be written about content types; all I'm going to point out is that they're important. Personally, I like reusing content types that other people have developed, like Atom, Collection+JSON, JSON HAL, or XHTML. Defining your own content type is more work than you expect.

#31. HATEOAS – Hypermedia as the Engine of Application State is a REST constraint that, described simply, means that your content should tell the client what it can do next by via links and forms. If you build your API with this constraint it mind, it will be much more resilient to change… if your clients obey your design approach too.

#32. Date/time – When you provide date/time values in your API, use a format that includes the timezone information. RFC3339 is a subset of ISO 8601 and is the simplest date and time format.

Security

#33. SSL – Consider whether you should offer your API under HTTP and HTTPS, or exclusively HTTPS. Exclusively HTTPS is an option growing in popularity.

#34. Cross-site Request Forgery (CSRF) – If your API accepts the same authentication configuration that your interactive users use, then you might be vulnerable to a CSRF attack. For example, if your interactive users login and get a "SESSIONID" cookie, and that cookie can also be used to invoke API requests, then a carefully composed HTML form could make unexpected API requests on behalf of your users. (*Edit:* Read more about protecting APIs from CSRF attacks.)

#35. Throttling – Make sure one API user can't bring down your system by writing a remarkably stupid API client. It happens, both accidentally and maliciously. If an API user exceeds the generous API request limits you should provide for them, give them a 503 response with a Retry-After header.

#36. Subtle Denial of Service – Throttling should prevent someone from smashing your API in the simplest way, but there are lots of subtle denial-of-service attacks too. Slowloris, Billion laughs, and ReDoS are interesting examples of DoS attacks that don't require a lot of source resources, but they can make your API run out of resources quickly.

Client

Whether you're providing test code to your users, or building an SDK for them, check that you're giving them a client that obeys a few simple rules:

#37. Connection Keep-Alive – Some HTTP client libraries require you to do some extra work to enable persistent connections. Persistent connections can have a significant impact on the perceived performance of your API.

#38. 401 before Authorization – Another quirk of HTTP clients is that they'll often require a "401 Unauthorized" response before they'll issue a request with an Authorization header. This can add a lot of time to your API requests, especially on mobile networks where high latency is a struggle.

#39. Expect: 100-continue – I know at least one API client that defaults to using "Expect: 100-continue"; if it doesn't receive the "100 Continue" response, it'll continue with the request after a 3 second timeout. If you don't support "100 Continue", this will be another performance drag that can be disabled client-side.

Other Stuff

#40. Documentation – Writing API documentation can be a real bore, but hand-written documentation is usually the best documentation. Be sure to include some runnable code or curl command-lines to help get people up-to-speed as quickly as possible. You can also look at documentation tools like apiary.io, Mashery I/O Docs, or Swagger.

#41. Design with a Customer! – Don't design your API in a vacuum; work with a customer and their use-cases. This will help you "Design for Intent" (#23).

#42. Feedback – Make sure you provide your API users with a way to give you feedback on the API. This can be through your support channels, or it can be a hosted forum, or maybe a mailing list. Make it as friction-free as possible for your user.

#43. Automated Testing – Your API should be the easiest thing you've ever had to build automated tests for. It's made for automation, after all. Take advantage of it!

From <https://mathieu.fenniak.net/the-api-checklist/>