DOI:10.1145/3369748

Article development led by acmqueue
queue.acm.org

Application programming interfaces speak louder than words.

BY THOMAS A. LIMONCELLI

API Practices If You Hate Your Customers

DO YOU HAVE disdain for your customers? Do you wish they would go away? When you interact with customers are you silently fantasizing about them switching to your competitor's product? In short, do you hate your customers?

Maybe you should try using your company's external APIs to show your disdain. What? How could you do that?

In this article, I document a number of industry best practices designed to show customers how much you hate them. All of them are easy to implement. Heck, your company may be doing many of them already.

Why would you want to use your company's API to show your hate? I think the answer is quite simple: Customers are jerks.

Darn customers! Always using our services! Bothering our salespeople for quotes! Creating more work for the accounts receivable department by sending us money. Needing customer support for stupid reasons such as: "The documentation is wrong," or "This feature is broken," or "Your product killed my cat."

See? Jerks.

Older readers may long for the good old days when companies that were actual monopolies would pretend to love their customers. Now we all work for companies that don't admit to being monopolies and actually hate their customers. Boy, how times have changed.

Technique #1: Don't Have an API

Not having an API is a good start. It also requires the least effort of all the techniques. All you have to do is think about adding an API, then *not* do it.

What good is an API, anyway? Primarily it allows customers to implement features that you didn't think of. "Look, buddy, if we didn't think of the feature, it couldn't possibly be very good. We hire the best and brightest to think of new features all day long and not implement them. Don't horn in on their turf."

APIs also permit customers to use a lot more of your product. If they have to click, click, click to use your product, they are going to use it only a little. If an API exists, they can automate their use of your product, which would let them use it a lot more. They could automate provisioning for their entire company. They could build entire new applications based on your API. Just think how much more of your product they would be able to consume with an API.

How totally rude! If they use your product more, you will have to buy more servers, spend more time cashing their checks, and, heaven forbid, maybe start hosting conferences where people use terms such as *leverage*, *hackathons*, and *chalk talks*. Conferences? Ick.

Technique #2: Make Signups Difficult

OK, you have lost the battle and your company wants to build an API anyway. At least you can press the brakes a bit

by requiring a complicated signup process. Self-service onboarding is hedonistic and could lead to dancing.

There are a variety of ways to make onboarding arduous for customers. Some companies require that you open a ticket or speak to an actual human being. That will make any introverted developer think twice before using your API.

Some companies want you to fill out an application form to be able to write an application. Making people beg to use your product is a good way to discourage new users.

For best results, the questions on such forms should be written by someone who previously worked as a CIA interrogator: Why do you want to use this API? What will your application do? Where were you on the night of the 12th? What's your mother's maiden name? Can you prove she's really your mother?

One such form I filled out required me to describe the application I planned to write. Six months later a SWAT team of auditors appeared at my house, weapons blazing, demanding I show them my code. They wanted to verify I had not lied. If my application didn't match my application, then I could be sent to application jail.

OK, that's not a true story. I did, however, once see that question on a form. Sadly, I didn't have a particular application in mind. I was going to explore the API and write a few simple Python-based utilities to automate some daily tasks. I didn't want to explain all that, however, for fear my answer would not be good enough for whoever was judging my application. In a panic, I simply described my application as "dark purple with white highlights." A few weeks later my application was approved. So far, I haven't been visited by any auditor SWAT teams, but as a precaution my code editor has been themed in dark purple with white highlights ever since.

Sadly, some companies do not understand how to make signups difficult. They either make the process entirely self-service, or don't require any kind of signup process at all. When will they ever learn?

Technique #3: Charge Extra. A Lot. Another way to send customers packing is to charge a lot for your API.

IMAGE BY ANDRIJ BORYS ASSOCIATES. USING PHOTO FROM SHUTTERSTOCK

Remember when mobile phone companies charged $100 for a $2 data cable? Why can't APIs be like that? You have the potential to inflict a surcharge on anyone who dares to want to make efficient use of your product. Don't squander this opportunity!

It is normal to charge for your service, or include API access only with the enterprise edition. In fact, that's a good way to keep out spammers or others who would abuse the service.

But that's not what I'm talking about here. I mean you should charge a lot extra for the API. A. Lot. Of. Money.

Make it a revenue stream instead of a way to encourage people to use your product. Make API access so expensive that the sales department thinks API stands for additional profit incentive.

This is a lot easier for on-premises software. There the SDK can be sold separately, perhaps using an otherwise unadvertised SKU. Require management approval, a blessing from the pope, and a note from your mother.

Technique #4: Hide the API Docs from Search Engines

Nothing says "We don't actually want you to use our API" like making your API documentation invisible to search engines. The "build, run, debug" cycle of decades ago has been replaced by "run, crash, Google, fix." If your API documentation doesn't appear in search engines, you've sent your customers back to the old days.

Luckily, this can be easily done by putting the documentation behind a login screen. If Google can't crawl it, it can't index it. Googling for answers about your API will be impossible.

Requiring some kind of registration or login to access your API documentation also prevents your competition from examining your API and learning from it. No competitors have ever thought to register using their home address, or to share a password from a friend, right? Never. They are not that smart. It could never possibly happen. You would certainly never do that, so why would they?

If your management refuses to hide documentation behind a login, consider making your documentation a PDF file. This is nearly as frustrating. Most search engines can peer into PDFs, but not if you print the documentation and scan in each page as a bitmap. If search engines OCR such documents, just reformat your text in columns, or tilt the document when you scan it. Be strong, young soldier! With a little elbow grease and a lot of moxie, you can stay one step ahead of anyone who wants to make it easy to access your documentation.

Technique #5: Use a Terrible Protocol

Many APIs use JSON:API (https://jsonapi.org) or JSON-RPC (www.jsonrpc.org). They are lightweight, easy to use, and easy to debug.

Why would anyone do that?

Debugging is boring. Wouldn't you rather appeal to customers who write bug-free code on the first try?

To really show disdain for your customers, use a proprietary protocol so that language support is limited to the client libraries you provide, preferably as binary blobs that are never updated. If you design it carefully, a proprietary protocol can be difficult to understand and impossible to debug, too.

Alternatively, you can use SOAP (Simple Object Access Protocol). According to Wikipedia, SOAP "can be bloated and overly verbose, making it bandwidth-hungry and slow. It is also based on XML, making it expensive to parse and manipulate—especially on mobile or embedded clients" (https://en.wikipedia.org/wiki/SOAPjr). Sounds like a win-win!

Technique #6: Permit Only One API Key

One of my favorite ways to show disdain for a customer is to permit only one API key at a time. Anything you can do to make the customer's operations full of toil and cognitive load says, "I don't care about you."

An API key is basically a password that identifies and authenticates a customer. You have only one password for your email account; why would you need more than one? That would be weird, right?

Eventually, customers will need to change, or "rotate," their API key. Maybe it was leaked. Maybe an employee left the company and has a copy of the key. Maybe they just need to rotate the key yearly as part of their security policy.

Here's the hilarious part. If you permit only one API key at a time, you've created a catch-22 situation. Customers can't change the API key on the server, because the clients will lose access until they've been updated, too. They can't change the clients first because the server won't yet know about the new API key. If there are multiple clients, then you're basically expecting your customers to flash-cut all clients at the exact same time. Basic physics says that can't happen. Even if it could happen, there's no way to canary the new key; you've added deployment risk and complexity where nobody would have expected.

What does your API reveal about your feelings toward your customers?

Technique	Treat customers with disdain	Show customers love
1	Don't have an API	Have an API
2	Make signups difficult, users must justify their request	Self-service onboarding
3	Exorbitant fees for the privilege of API access	Enable API access for free or as part of an "enterprise-level" package
4	API documentation behind login page or otherwise hidden from search engines	API documentation freely accessible and referenced by public search engines
5	Use a proprietary or terrible protocol	Use an industry-standard protocol such as JSON:API or gRPC (https://grpc.io)
6	Permit only one API key	Permit multiple API keys for easy rotation
7	Tempt fate by maintaining documentation manually	Keep documentation in sync with code using automated systems such as Swagger or gRPC
8	Ignore the infrastructure as code (IaC) revolution	Make IaC a top priority: Provide officially supported modules for Terraform, Chef, Puppet, Chocolatey, and similar systems
9	Design APIs to be non-idempotent whenever possible	Design APIs to be idempotent whenever possible

They say the secret to great comedy is timing. Imagine the hilarity of a customer using your product for a year before realizing key rotation is logistically impossible. Surprise!

Hilarity? Or disdain. Whatever.

Some companies do not understand comedy, or how to show disdain for their customers. They permit customers to add a new key on the server, slowly roll out the new key to all clients, testing along the way, then deactivate the old key. Gross!

Technique #7: Maintain Documentation Manually

As your API evolves it is possible for the API and the documentation to get out of sync. Nothing says "I don't care about my users" like building a system that encourages this kind of error.

Or you can go for the trifecta: an API that is out of sync with the documentation, which is out of sync with the client libraries you provide.

Sure, there are systems such as Swagger (https://swagger.io/tools/open-source/) and gRPC (www.grpc.io) that let you define APIs and their documentation in one place, then automatically generate the documentation, server stubs, client SDK bindings in multiple languages, and so on. But what's the fun in doing work once and letting computers generate all the downstream artifacts you need for free? Consistency is for simpletons.

Technique #8: Ignore The IaC Revolution

The ability to treat infrastructure as code (IaC) is becoming a top priority for operational teams. It not only makes operations easier, more testable, and more reliable, but also paves the path to security compliance best practices required by the likes of SOC2 (Service Organization Controls) and PCI (payment card industry).

Some companies waste their time making it easy for customers to do this. They provide officially supported modules for accessing their services from Terraform, Ansible, Chef, Puppet, and similar systems. They make their client-side software easy to consume by hosting repositories for multiple Linux distributions, and they provide a Chocolatey feed for easy installation on Windows.

Nothing says "We don't actually want you to use our API" like making your API documentation invisible to search engines.

It's much simpler to ignore all of these technologies and hope that the open-source community will provide. Yes, this may result in a confusing array of incompatible options, but you can trumpet the benefits of "user choice."

Technique #9: Don't Be Idempotent

I've saved the nerdiest technique for last.

An operation is idempotent if performing it multiple times yields the same result as performing it exactly once.

Suppose there's an API call that creates a virtual machine (VM). If this API call is idempotent, the first time we call it the VM is created. The second time it is called the system detects that the VM already exists and simply returns without error. If this API call is non-idempotent, calling it 10 times will result in 10 VMs being created. (Note: the opposite of *idempotent* isn't *potent*.)

Similarly, an idempotent delete call will remove the object; subsequent calls will quietly do nothing and return a success status code. If the call were non-idempotent, the second call would return a "not found" error, which would confuse the developers and potentially make them question the meaning of existence.

Why would anyone issueo the same API call more than once? When dealing with RPCs (remote procedure calls), the response may be success, failure, or no reply at all. If you don't hear back from the server, you have to retry the request.

With an idempotent protocol you can simply resend the request. With a non-idempotent protocol, every action must be followed by code that discovers the current state and does the right thing to recover. Putting all that recovery logic in the client is a layering violation.

In the VM example, you would have to query the inventory and see if the VM you asked to create exists. If it does exist, you must make sure it was created properly or is in a good state. If it is in a bad state, you repair it or delete it and start over. The list of potential conditions and edge cases goes on and on.

That was a simple example. Recovery from other API calls can be even more complex. The attempts to recover from failures may also fail. Now you are faced with an infinitely recursive world of failures, failed re-

covery attempts, and on and on. Code that looks sane on first glance ends up creating zero VMs, or three VMs, or more. With multiple simultaneous clients, you must deal with timing, locking problems, crossed messages, and a nest of heisenbugs.

Putting this logic in the client library ensures the client will need more frequent updating. Requiring the user to implement the recovery logic is delightfully evil: how would they even know what they should implement?

These problems are reduced or eliminated when the API is idempotent.

Why not simply use a more reliable network? Oh, that's just adorable. Networks are never reliable. They can't be. Thinking that networks are reliable is the first fallacy of distributed computing (https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing).

If networks are unreliable, then a network API is inherently unreliable, too. A request can be lost on its way to the server and never executed. Execution may be complete, but the reply back to us gets lost. The server may reboot during the operation. The client might reboot while sending the request, while waiting for the request, or after receiving the request but before local state is stored on stable storage. So many edge cases!

In distributed computing everything can fail. If you hate your customers, you can make sure that dealing with failure is burdensome, error prone, and just plain impossible to get 100% right. Customers will always be fixing edge cases instead of doing productive work.

Don't spoil the fun. Show your disdain for customers with non-idempotent APIs.

Summary. The accompanying table includes a summary of these techniques along with ways that companies may accidentally provide good service to their API customers.

Getting buy-in. Your coworkers may resist some of these techniques. How do you get them on board?

You could have them read this article, although that could backfire. If the wrong person reads it, he or she might push back and do the opposite.

If that happens, you might end up with a great API that is easy to get started with, easy to use, has great documentation that is easy to access, and helps people write code that works the first time and every time.

If networks are unreliable, then a network API is inherently unreliable, too. A request can be lost on its way to the server and never executed.

Shirley, You Can't Be Serious!

This article is written in jest to make a point. Although some companies do the bad things set forth here, they don't do them to hurt customers. In my experience, engineers take pride in doing good work and impressing customers with well-made systems. I trust that when companies do the naughty things in this article, it is out of ignorance, lack of resources, or an impossible deadline.

Luckily, in some cases the good practice is easier to implement than the bad practice. Creating an authentication system to restrict access to documentation is more difficult than making the documentation freely available. Putting all documentation on one long page so that it can be searched using Ctrl-F is easier than putting each API call on a separate page.

Sadly, some of these good practices do require a lot of work. Creating a self-service onboarding system is not easy. It requires usability testing and revisions. Ease of use is never achieved on the first guess.

Justifying the resources required for all these good practices may be difficult, especially when an API isn't used by many of your customers. "What's the ROI when hardly anyone uses our API?" your management may ask. I look at it differently: Maybe usage is low because you haven't done these things. C

Related articles on queue.acm.org

Programmers are People, Too
https://queue.acm.org/detail.cfm?id=1071731

Forked Over
Kode Vicious
https://queue.acm.org/detail.cfm?id=2611431

Managing Technical Debt
Eric Allman
https://queue.acm.org/detail.cfm?id=2168798

Thomas A. Limoncelli is the SRE manager at Stack Overflow Inc. in New York City. His books include *The Practice of System and Network Administration*, *The Practice of Cloud System Administration*, and *Time Management for System Administrators*. He blogs at EverythingSysadmin.com and tweets at @YesThatTom.