FIVE FEATURES OF A SUCCESSFUL API PLATFORM

Effectively building an “API Culture” in your organization

Chris Fauerbach
chris@fauerbachconsulting.com

TABLE OF CONTENTS

INTRODUCTION ABOUT CHRIS 1

"API CULTURE" 1

Example Legacy Case – Not an "api culture"..................................2

"API Culture" – Update to the case! 3

But who maintains it?! 3

DOCUMENTATION.. 4

AUTHENTICATION 6

AUTHORIZATION... 8

SCALABILITY... 10

CONFIGURATION.. 12

WRAP UP 13

INTRODUCTION

ABOUT CHRIS

Chris Fauerbach is a veteran software engineering consultant. He has led corporate and startup engineering teams to success. Startup life has been the most exciting ventures he's been involved with, including the successful growth and exit of nPulse Technologies (acquired by FireEye in 2014), where Chris started as Engineering Employee #1, and the first hire by the Founders.

I try to write about APIs, new/old technology, software integration, cybersecurity and technology careers.

"API CULTURE"

I've coined this phrase to help indicate a healthy technology organization that strives to build a solid set of capabilities that can be leveraged across a wider audience than an engineering team typically gets.

Building an API is more than just writing a web service. It's more than using AJAX or using a REST Framework within your code. Building an "API Culture" is all about providing your development teams the structure and ability to be as effective as they can be. Increase collaboration, increase code quality and increase the reuse of applications that they build. We're not talking about a specific tool or methodology, instead, we're talking about an attitude that your teams will adopt in order to enjoy their day to day working life a lot more.

One of the issues with technology within growing organizations is almost inevitable when different teams build different applications. They get branded, pigeon-holed: whatever word you want to use. They can very easily get stuck. They'll know all about **Widget ABC** but they'll never be able to help **Team GoGetters** fix **Application XYZ** because they're trapped!

EXAMPLE LEGACY CASE – NOT AN "API CULTURE"

** These are real stories. The names of teams and people have been changed. If you find yourself in this story, it MAY be you I'm talking about. Literally or figuratively. If we've worked together, it may LITERALLY be you I'm talking about.

I spent time consulting at a mid-sized financial company. Through various mergers and acquisitions, they had 7 applications that performed the same function in their business. No lie, seriously: 7. That's a huge number! It's completely understandable how they got there. They were growing rapidly and gobbling up complimentary companies. The issue (at least the one I want to harp on here) has to do with how each project became so isolated, they were stepping on each other's toes and slowing the entire technology organization down to a crawl. Each team had a full complement of developers, DBAs, project managers and testers.

One day, the company (enterprise) decided to switch print shops. They used an external/third-party print shop to send out monthly statements to customers. Each of these systems were generators of the statements that had to be sent out. Guess what happened? Seven different teams had to write custom code to change how their system sent out mail lists to the printer. This meant 14 developers (two per), 7 testers, 7 (parts of) project managers and everyone else had to be involved in this 'simple' change! (Ok, changing vendors can be anything but simple, but we'll gloss over this and assume the integration was actually simple).

"API CULTURE" – UPDATE TO THE CASE!

Now let's imagine the same scenario, but, the integration with the third party vendor happened at an 'internal' company owned API. This single API could take requests from all seven systems and broker the data out to the vendor. OH MY GOODNESS!! Think of all the benefits of a single point of integration! (No, don't add the word failure – single point of failure – we'll cover that, so it doesn't happen!)

Once each of the teams integrate with the internal service (Yep, that's work, but hopefully a lot simpler than integrating with an external group!), then to change the external vendor is a single change. If the entire corporation uses the same API to deliver their statements, we have drastically simplified the overall architecture and dependencies.

But who maintains it?!

Everyone! Sure, pick someone who's accountable to make sure the API is maintained, up and running, etc, but allow your entire team the ability to make updates to the API through a defined governance process. This way, you don't have a single team responsible (and capable) of updating this integration point. I'll publish more on this development model, but for now, let's dive into some features of a framework or platform that will make this model successful.

DOCUMENTATION

"Wait a second, you're leading off with documentation? Ugh."

– ***Every developer, ever***

I know, documentation is awful. I personally do everything I can to avoid documenting anything. (Not really, if you hire me, I'll document stuff, I promise). In an effective "API Culture", documenting services is critical. If they aren't documented, they won't be adopted. If they're adopted, then it will be an unhappy integration.

When building a new set of capabilities, or APIs for an organization, documentation is key. In this day and age, a ton of this can be automated, and it's really no more than using a documentation generation tool.

It will be extremely easy to integrate with a service if you could browse a documentation repository (that was auto generated) and find things like:

- *Code samples*
- *URL of dev, test and production environments*
- *Inputs*
- *Outputs*
- *Error conditions*
- *Expected results*
- *Downstream impacts*

That would obviously make life simple! Now do the opposite. Imagine trying to integrate with an API that DIDN'T provide that information?

Think about the time suck on both sides. The integrator and the publisher. It would be miserable.

Document!

Start here:

http://swagger.io/

AUTHENTICATION

It’s very rare to allow open access to data or capabilities inside or outside of an organization. In our case study earlier, we talked about having an API that would print a customer invoice with a third party vendor. Would you allow an unknown party to call that API? Nope, didn’t think so.

Knowing who is calling a service is done via Authentication. Pretty simple in concept, but not necessarily easy to understand and implement. The basic methods of Authentication are “username” and “password”. Each party gets a unique username, sets a password, and they’re off to the races!

That works for simple things like web users, people reading a blog, or updating their mailing information for their magazine subscription.   Is that good enough for accessing medical records or viewing credit card transactions? In this day and age, we have various ‘things’ that need to be authenticated. Another concern we have is extra data around an authentication action. Can your system tell if someone who usually logs in from Kansas is now trying to log in from China?

We also have to authenticate the application that’s trying to call the service. Not just the user that the application is trying to authenticate on behalf of.

That’s a little tricky sometimes, and we’ll get more into “authorization” next (dang, I gave it away), but knowing the calling client is important too, especially on ‘internal’ applications when there’s not necessarily an end user involved. Do you want any application (hacker on your network with a python script) printing and sending account statements?

Authenticate:

- *Users*
- *Clients (applications)*
- *Requesting Party (UMA: User managed access , 3rd party)*

Start here:

https://oauth.net/2/

AUTHORIZATION

Once we know “who” is performing an action, we need to figure out if they’re “allowed” to do what they want to do.

Do they have access to the data (e.g. customer data)?

- *Job function*
- *Job title*
- *Paygrade*
- *Confirmed email address*
- *Time of day*
- *Past behavior*

Are they allowed to perform the action (e.g. pay an invoice)?

- *Check Writing Authority*
- *Authorization Levels*
- *Responsible Vendor List*

In many instances, it’s not only the User that needs to be authorized, but it’s also the ‘client/application’ that needs to be authorized. Some applications can keep secrets, some cannot. For instance, a publicly accessible web site cannot be trusted to keep a secret. Even if traffic is flowing over an encrypted channel (HTTPS, TLS), the data needs to be visible to the web client and therefore, the end user. This can be compared to a custom application, that runs in your private network layer and can be compiled and obfuscated. That application can very well keep it a secret. One of the secrets it will need to keep is its own identity, to identify itself during API calls. In OAUTH2/OIDC language, that’s the “client secret”. This is a long, unique and hard to guess string that identifies the client. Your API platform will then ‘know’ that the request is coming from a valid client, and can make authorization decisions based on that.

In the example of our invoice printing, let's assume we have the Real Software System 1 that is trusted to send invoices. It has a 'client secret' of "123456789" (obviously, not a real secret key, but easy to type). Every time it calls an API end point, it passes that ID. Now, even if a malicious client "knows" the end point and the parameters that are needed to send a customer invoice, it can't do it unless it also has a trusted "client secret".

Start here:

https://tools.ietf.org/html/rfc6749#section-2

SCALABILITY

When you build something that others will use, others will use it. “If you build it..”, or something like that. That’s the point of building easy to use, enterprise services. The more people use an API, the more computing resources you’re going to need.

Gone are the days of having to procure and provision a server, months in advance, and having limited rack space. Gone are the days of “vertically” scaling application servers. That game is costly and prohibitive. Enterprises had to purchase according to their peak performance, and systems would sit idle 90% of the time.

I did some work at another mega corporation that did some major data center consolidations. The company had approximately 9 data centers spread out across the continental US. These weren’t small collocated racks. These were gigantic warehouses that cost millions of dollars to run annually. Now, they’re shrinking the data center foot prints.   It’s time to move away from vertically scaling, and embrace horizontally scaling applications in the cloud.

Looking at a service provider such as Amazon Web Services (AWS), there are technologies that will automatically scale the deployment of your application. If there is a growth in usage, the service can temporarily add another virtual server that’s running your API. When the load comes down, that server will disappear. Pay for what you use, not what you had to plan for at peak time.

Always scale back down! If you don’t set your scaling policies to come back down, you’ll probably end up paying MORE in the long run. Scale up quickly, scale down a little slower.   As an example, at AWS you can start a simple API with a micro server. If CPU gets above 50% over a five-minute period, add another server or two, up to a max (watch that budget!).   Every 10 minutes under a 50% load, drop a server. Make sure all of your provisioning is automatic and you don’t have to attend adding a new server. That would be kind of pointless. Pick your technology to auto provision. Options include user scripts, Puppet; there are lots of methods out here.

CONFIGURATION

The coveted fifth slot in the (unordered) list of five things to make sure you cover in your API platform. Configuration. Mundane! Hear me out though, this can bite you.

Through the life of an API, or any application for that matter, there are tons of items that will be configured. For instance:

- *Number of threads to run*
- *Client permissions*
- *Downstream server names*
- *Database credentials*

Find a tool that makes updating these items easy. Manual updates, automatic updates, updates that persist to scaling groups, etc.   This is hyper critical.

The first assumption here is that you're properly using configuration in a lot of your code. Not configuration files typically, but 'external' data stores (Redis, Database, etc), or best of all, environment variables. If you have a custom application that's deployed in your network, use environment variables all over the place to avoid hard coded strings, or hard to update configuration files. In your code, re-read those environment variables regularly, not just on startup.

Environment variables are excellent, because they allow your code and configuration files to be the same no matter what environment you're deployed to. Dev, QA, Prod etc all 'act' the same, just read environment variables to find out how to behave.

WRAP UP

This may still seem overwhelming. There are so many things that are required and can be missed when setting up a new API Platform. There are technical aspects that you can't mess up. There are organizational aspects that you'll probably overlook. There are culture components that can make or break your journey into an "API Culture".

Please don't hesitate to reach out for consultation. I'd be happy to work with you and your organization. I can always be found at:

Site - https://fauerbachconsulting.com

Blog – https://fauie.com

Email – chris@fauerbachconsulting.com

chris@fauie.com