rubiX
integrating the future

WHITE PAPER

Pim Gaemers
maart 2019

The overambitious API Gateway

Inhoud

The overambitious API Gateway

For a couple of years in a now, Thoughtworks has mentioned to hold the "overambitious API Gateway" in their Techradar (https://www.thought works.com/radar/platforms/overambitious-api-gateways). Yet in a growing market for API Gateways, and in an attempt to differentiate themselves from the competition, vendors keep adding new features and functionality in their product, blurring the line between business domains and infrastructure. In certain cases, the API Gateway even looks and feels like a full fledged integration platform or, dare I say it, ESB.

Being quite unique in featuring a single entry for multiple editions of the Techradar in the hold position, the "overambitious API Gateway" deserves a closer look, together with the concerns and risks that come with it.

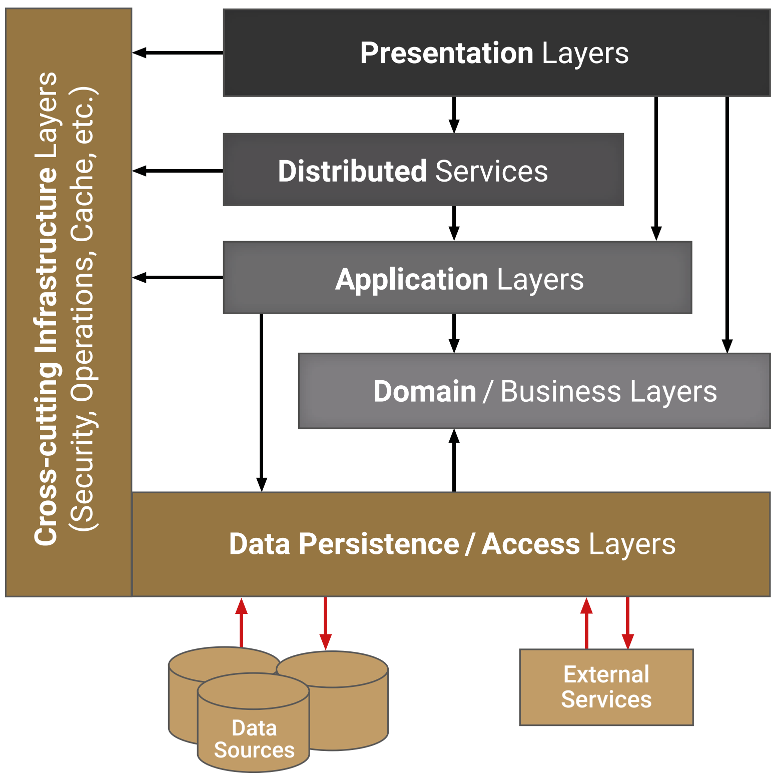
Separation of concerns

The crux of the matter lies in the specificity of certain functions and the way the API Gateway is positioned in the overall IT landscape. Traditionally the API Gateway is used as a shared component in the infrastructure of the IT landscape, providing a narrow set of generic functions. With generic functions, we mean that the functions provided by the API Gateway are not bound to specific business functions or to a specific business domain. Whether used in the HR process or the sales process, is indifferent to the API Gateway. No specific functionality or information regarding the HR or Sales processes is embedded in the API Gateway. Instead, the API Gateway can be viewed similar to an infrastructure component like a corporate firewall or proxy. The result of this, is that the API Gateway can be highly optimized for the functions it needs to implement, most notably API authentication and rate limiting. Operational processes regarding the API Gateway can be optimized, and the required skillset can be specific for the API Gateway. This makes rolling out an API Gateway, operating it and adding to it an optimized and clear cut process, well suited for automation.

However, with the emergence of new capabilities in API Gateway products, like transformation and orchestration, the line between the generic and business functions becomes murky at best. The problem with these functions is they can no longer be separated from the underlying data and/or functionality of the API's in the backend. Having the capability to transform payloads and perform orchestration seems innocuous, but like we have seen time and again in the world of IT, this opens the door to more and more functionality inside the API Gateway, thereby ever increasing the dependency to the different API's in the backend. This makes the software delivery process long and error prone, an occurance sometimes also dubbed as the "integration hell".

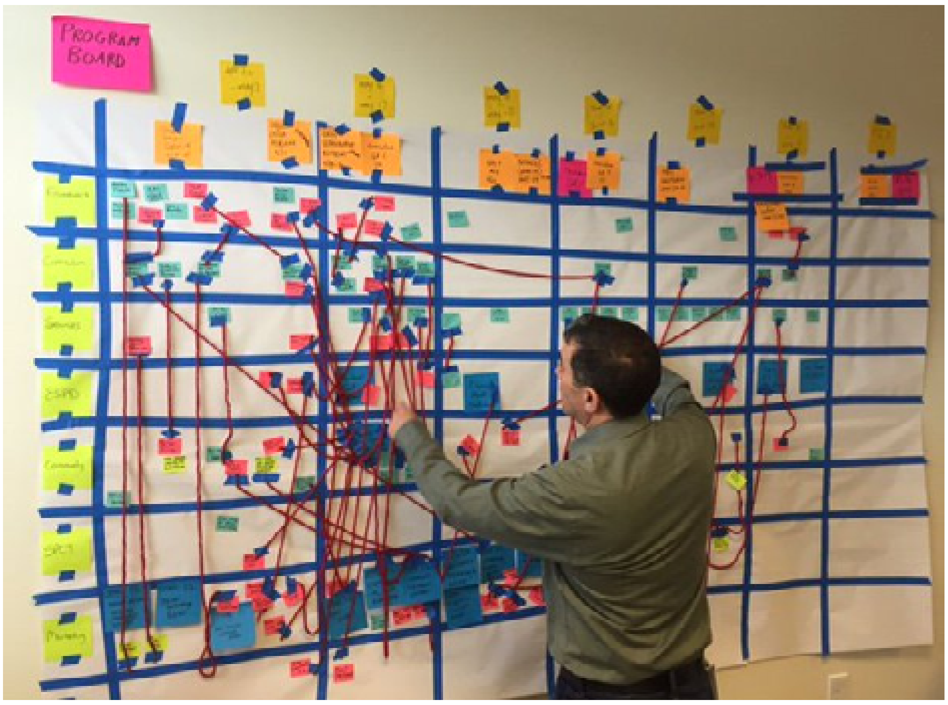
'Implementing business specific logic in a shared infrastructure component is almost never a good idea.'

From a practical point of view, the consequences of these interwoven capabilities can manifest themselves in the form of a more complex software delivery process and organization, resulting in longer time to market of new features and functionality and, arguably even more important, in a longer time-frame regarding fixing bugs and outages.



API Gateway falls into the cross-cutting infrastructure layer and should not contain any functionality of any other layer

These longer and more complex software delivery lead times and operational inefficiencies are usually profoundly increased when the team responsible for the application or API, differs from the team responsible for the API Gateway. When different teams are involved, additional burden is placed on the API Gateway team, who now not only has to operate and maintain the API Gateway itself, but also must have the additional capability and knowledge of implementing and maintaining the business related functions often specific to certain business domains or departments. This additional responsibility often strains the API Gateway team to such an effect that it becomes the bottleneck in the organization for software delivery and maintenance, since it has to be performed for a multitude of applications/API's.



Software delivery involving multiple teams can become complex very quickly

On the support side of things, this can result in a more complex and therefore often longer and more expensive resolution of support calls. Making a quick an clear cut analysis of the problem becomes increasingly difficult when the functionality of certain API's is split in the backend API and the API Gateway.

Even when initially a clear separation is defined between the API's and the API Gateway, when under time pressure to quickly resolve an incident, the API Gateway team can implement a fix in the API Gateway that, based on architecture principles and design practices, ideally should be resolved in the backend API. Thereby introducing architectural and technical debt, resulting in a degrading architecture over time, bringing closer the ever dreaded "integration hell".

Remember ESB's

This, of course, is nothing new in the world of tech and enterprise software. The industry has only just rebounded from numerously failed ESB implementations. And the new trend to move away from centralized managed integration hubs to a more distributed approach. These vendor driven centralized systems had a tendency to turn into gigantic monoliths: difficult to implement and even more difficult to maintain. Often directed from a centralized specific ESB or integration team, vendor specific expertise was required to build and operate the services deployed on the ESB. Creating the very bottleneck we now slowly see emerging in the API Gateway.

An excellent article about the detriments of ESB's was written some time ago by Andy Hedges: https://blog.hedges.net/2014/01/20/why-you-dont-need-an-enterprise-service-bus-esb/

No wonder this led to the trend of microservices with "dump pipes and smart endpoints". Which essentially reduces any shared components to either infrastructure or eliminates them all together. Accompanied with software development approaches like domain driven design and the hexagonal architecture components, systems become smaller and more specific. Which comes as no surprise. Going all the way back to the early days of Unix: it is better to do one thing and do it well. http://dotadiw.com/

In its "pure form", the API Gateway still is an exceptionally good fit in any microservice architecture, as well as in a lot of traditional architectures. It then provides generic infrastructure capabilities to the microservices whose responsibility it is to implement a certain business feature. Not to mention the additional benefit of often having the entire API Management capabilities integrated in the solution. Of course that last benefit still holds true for any API Gateway, overambitious or not.

What about the vendors?

It seems a lot of API Gateway vendors are again opening the doors to the ESB anti-patterns we all were glad to leave behind only a few years ago. This hardly comes as a surprise, since the entire ESB market was highly dominated by large software vendors until recently. Offering specific IDE's and other tooling for implementing and operating services on the ESB, the knowledge and expertise needed for this were highly specific as well. The result being a lack of integration with configuration and automation tools, making delivering software often a manual and archaic undertaking.

What about devops and distributed gateways?

The latest trend in API Management probably is distributed API Gateways. Stepping on the shoulders of the earlier rise of the service mesh, a distributed API Gateway can, among other things, offer reduced latency, more specific configuration, and increased security.
With distributed gateways, the gateway is often deployed right along with the application/API and is tailored to the specific application, as opposed to the traditionally used shared API Gateway. For example, it is deployed as a sidecar container when used in a Kubernetes cluster.

'The latest trend in API Management probably is distributed API Gateways.'

To be clear, there's nothing against having such distributed gateways. Depending on the overall architecture, they often are an excellent idea, not only for east-west traffic, but also for north-south traffic.

But it is important to note that from a component point of view, there still is a distinction between the application itself and the API Gateway. Implementation details, like programming languages and frameworks, as well as required tooling for development and deployment, likely differ in between applications , resulting in different tooling and lifecycles for these components.

Dependencies and migration paths

Even if both the API Gateway and the backend application/API are owned by the same Dev/Ops team, it is still worth considering to keep business related functions out of the API Gateway. When these are interwoven in our overambitious API Gateway, the dependency with the backend application/API containing the business logic, becomes very strong, making a migration path or lifecycle update more complex. For every change, whether technical of business driven, a careful analysis must be performed where to perform it. Not only time to market should be considered, but also operational efficiency. Migrating the API Gateway itself to another platform or vendor, or migrating the underlying application/ API, becomes dramatically more complex since functions are spread out over multiple components.

API Gateway! = API's

When thinking about the overall API landscape, and its realization in what often gets called 'the API program', it is important to distinguish the different aspects and components. To clarify, the overall API landscape can be broadly categorized into four distinct parts:

1. **The API Gateway(s)**
2. **The API Manager**
3. **Developer portal**
4. **API's**

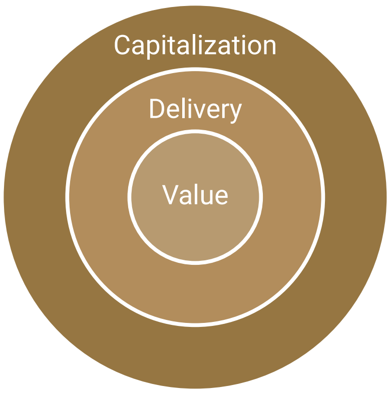
Whereas the API manager provides account-management, analytics and configuration regarding different subscription models, the developer portal is used for outside developers to gain access to the API's documentation and can be a powerful marketing instrument in an API strategy. (Which, perhaps, will be a subject for another time.)

The problem with the overambitious API Gateway primarily lies in the lack of distinction between the API Gateway and the API's themselves. And in the overall confusion that the "API Management" platform is not just for managing API's, but also for delivering API's.

This should be mitigated with proper architecture and design principles. But it serves organizations well to first and foremost think about their entire API program and API Strategy from a more holistic point of view.
In its essence, an API Program describes three aspects of opening up data assets and functionality (often via API's) to a specific, pre-defined, target audience. An API program should answer the:

- Value
- Delivery
- Capitalization

of an API or entire API program.



This forms the initial basis for an API strategy. An API strategy contains specific aspects of the API, like what data assets are made available, what functional use cases can be handled, and what are the more technical/ architectural aspects of the API offering. For example, if GraphQL or REST is being used. Next to the API offering, a target customer segmentation analysis and the overall API Economy should be described. In short, what aspects of the API are made available for whom. Is the API only for internal use, or is it opened up to business partners or the general public. These aspects may seem distant from our discussion of the overambitious API Gateway, however, these considerations do have an impact on the architectural decisions and design practice later on.

'API strategy and program decisions may seem distant from the implementation, but they do have an impact on architectural decisions and design practices later on'

! **Warning: some oversimplified examples coming up!**

For example, an API program and API strategy describe an API which is made available for internal users as well as public use. The architectural decision for serving this API is setting up multiple API Gateways. One inside the corporate network, only to be used by internal teams for consuming the API, and one additional API Gateway setup in a public cloud environment used by the general public. Routing and orchestration functions are likely more prevalent in the internal gateway, as opposed to the public cloud gateway, hereby effecting the design decisions for the functionality in the backend API and the API Gateway. Also, from an operational perspective it most likely is not beneficial to have a different configuration of gateway running, serving the same API.

Another simplified example is when the API is better suited using GraphQL instead of the more traditional REST paradigm. The attributes of a GraphQL, currently, limit certain aspects of API Gateways, for example caching and message transformation.

So the overall API program and API strategy do have an effect later on, on architecture, design, and implementation of the API's and the API Gateways used to serve those API's.

API Strategy and API design practices are part of an API program, and often the API Management solution is as well. However, organizations tend to place this on top of already existing API's and services. Realizing there is a misalignment between their existing services and API's, and the API's in the API Strategy, which are aligned with their business model and based on solid API design practices, bridging that gap is often left to the API Gateway., A task it really is not meant for.

Of course every organization and architecture is different, but as a rule of thumb, there are a couple of don'ts for an API Gateway:

- Do not chain/composite API's in the API Gateway, whether this is done via orchestration or choreography.
- Do not modify HTTP payloads (both request and response) in any shape or form in the API Gateway.
- Do not perform any routing based decisions on domain specific logic or business rules.

Don’t blame the vendors

Yes, we see a lot of features entering API Management products which are questionable at least. But nobody is forcing you to use these features. And it should not be used as an excuse for not doing proper architecture and design. Or for not thinking about the purpose of API’s, their delivery mechanisms, and how they benefit the organization in the first place. But instead leaning solely on the features of an API Management platform and deliver API’s in a unstructured ad hoc manner, aka doing it on the fly.
In the end, it is the organization itself that has to come up with a proper solution which is manageable in the long term. Whether this means keeping the API Gateway as “thin” as possible, or deciding to leverage all the functionality the platform offers.

But do keep in mind the overall value for money. API Gateways come in various pricing brackets and although not always directly correlated to the number of features and functionalities available, it is still worth to critically evaluate the different offerings based on the architecture and design practices you plan to adopt in your API program.

Conclusion

No matter whether you are using an overambitious API Gateway, or even an ESB, it seems that with the increasing amount of features, integration platforms, and enterprise middleware offers, it’s often too tempting for teams not to use these features. For instance for initially delivering a new API quickly, so they don’t have to bother the backend team managing the API, or for quickly resolving that production issue themselves, instead of dispatching the issue to the responsible team. Whilst these overambitious API Gateways can have desirable functionality for certain organizations, be weary of them, as down the line they often become a massive burden for development and operation teams.