Layering Microservices

Sep 24, 2018

• Architecture • Microservices • Meetup • Patterns • Edge • Distributed Systems •

At Meetup, we are going through the *oh-so-familiar* path of splitting a monolithic system into microservices. The work on this started a few years ago, and the team has made sure that most of the microservices prerequisites were in place before we take any further steps. I joined the team this summer to help with planning and executing the architecture changes that are required to take us to the next level.

As we go through this process, one aspect of software architecture that is constant in our day-to-day is the use of *Layers* to organize our components. Layering is a technique that hasn't been discussed as much when it comes to microservices. In this article, I want to review the application of the Layers pattern in a services architecture, and also discuss two layering strategies and how they have been fundamental to me when migrating from monolithic to microservices architectures.

Layers in Service-Oriented Architecture

I believe that Layers are one of the most useful tools in software architecture. They help group components and define how dependency and communication chains happen between them.

Frank Buschmann and his collaborators wrote the most comprehensive description of Layers in software (that I am aware of) in their seminal work Pattern-Oriented Software Architecture, Volume 1, published in 1996. But even before that, Meilir Page-Jones had previously used the concept to describe an Object-Oriented runtime, although he used the word *domains* to refer to each layer. I particularly like using Martin Fowler's description of Layering from his book Patterns of Enterprise Application Architecture:

> *When thinking of a system in terms of layers, you imagine the principal subsystems in the software arranged in some form of layer cake, where each layer rests on a lower layer. In this scheme the higher layer uses various services defined by the lower layer, but the lower layer is unaware of the higher layer. Furthermore, each layer usually hides its lower layers from the layers above, so layer 4 uses the services of layer 3, which uses the services of layer 2, but layer 4 is unaware of layer 2. (Not all layering architectures are opaque like this, but most are—or rather most are mostly opaque.)*

Layers then are groupings of components stacked on top of each other. The word *component* here is a placeholder for whatever the abstraction unit you are working with, e.g., classes, functions, services, etc.

The most well-known implementation of the Layer pattern is probably the networking stack, including its most popular implementation TCP/IP. This choice is often credited for the flexibility and consequent longevity of TCP/IP, making it possible to extend them in ways unforeseen when they were first designed.

If Layering is about grouping components and stacking them, there is still the question of what criteria to use when grouping components. In fact, in the same book quoted above Fowler says:

> *[…] the hardest part of a layered architecture is deciding what layers to have and what the responsibility of each layer should be.*

Considering our focus on services, one could aggregate components in Layers based on the tech stack they use, their expected availability or many other criteria. Even within an engineering organization, different teams (e.g., infrastructure, appsec, application development, cost management, etc.) will likely have different approaches to these groupings depending on what traits are more interesting to them.

Thus, there are infinite combinations in which one can aggregate services into Layers. It is doubtful that a single layering model will be enough to understand every aspect of your architecture, as each one focus on one particular viewpoint. You will be using a combination of layering models to manage a complex architecture.

After building a few microservices architectures, I've found out two layering schemes that are invaluable in understanding and managing such highly-distributed architectures. They are so widely applicable that I will refer to them as Architecture Patterns.

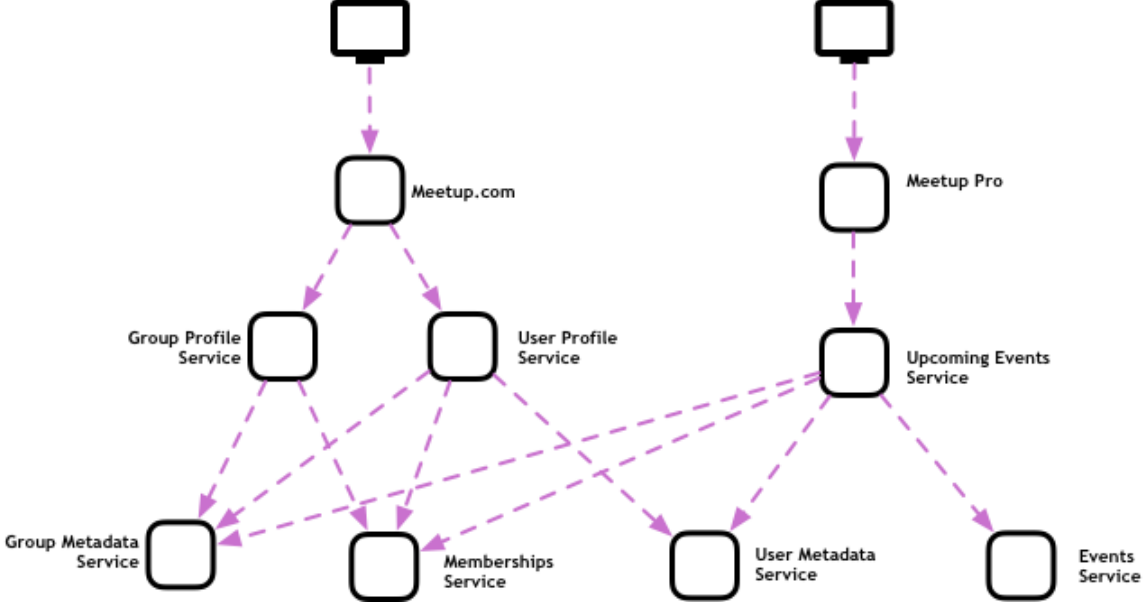
Pattern: Clay-to-Rocks Layering Model

Even amongst services that have similar reliability or security characteristics, e.g. services that implement business logic, we find that they are not all the same in many other vital aspects.

Let's consider a fictional example based on our work at Meetup. At our main consumer website, Meetup.com, there are lots of different user flows. Let's focus on the features touched by our users when someone is looking at a profile. They might be looking at their own user profile or at someone they met at an event. They might also check out a group's profile, to see if they provide the kind of experience that the user is interested in.

In a sophisticated microservices architecture, it is common for each one of the flows above to have their own user-case focused microservices, in turn, invoke lower-level microservices that have data on groups, users, events, etc.

The consumer website isn't the only way users interact with us, however. So that brands and business like Google or DigitalOcean can organize their multiple meetups across the globe, we offer a product called Meetup Pro. One common use case for a Pro user is to get an overview of what events are scheduled across their groups. This is also modeled as a microservice in its own, accessing a few lower-level services.



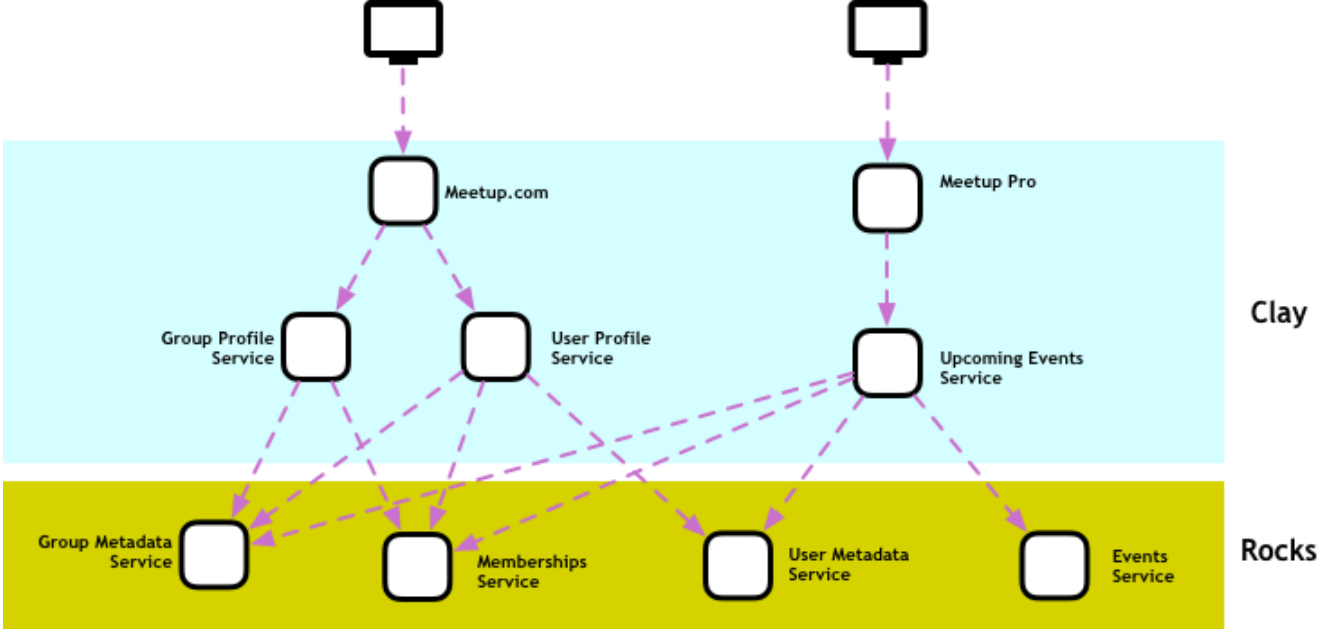
Following this scenario, we have some services that are use-case driven, offering data that corresponds almost one-to-one with what the user sees on screen, and some that are more *raw*, meaning that its data needs to be processed, filtered, and aggregated before it can be presented to users in a meaningful way.

When we look at services through this lens, we start to see some strong correlation between how user-case driven or raw a service is and how often engineers change them over the product's lifecycle. How many times have you seen the user profile page on social networks like Facebook or Twitter get a facelift since you first joined these networks? They surely look very different now from just a year ago. But, if you think about it, how often has the actual data there changed in a significant way, like when Facebook implemented its "real name" policy, or when Twitter made some profiles "verified"?

In product development, the closer to the customer a piece of software is, the more often it changes. The services on the top of the stack are where product managers and marketers want to improve the experience, where designs need to be refreshed every few months, and where most of the experimentation happens. They naturally experience more *churn* than other services, and this gives us an opportunity to optimize components at this layer for fast-paced change.

Components at the bottom of this diagram, on the other hand, don't change that often. Of course, at some point someone added an attribute to a group or to a user that wasn't there before, but this was often a *big deal*, surrounded by careful change management and a migration strategy from the previous to the new state.

This dichotomy is big enough to justify its own layering model. I like to call this *Clay-to-Rocks*:



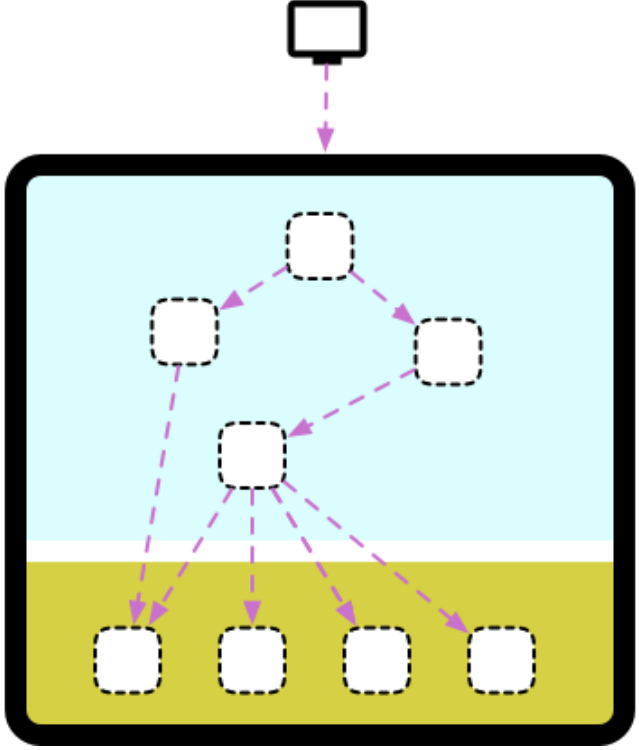
In this model, we group services based on how frequently we expect them to change. *Clay* is a nickname for software that is expected to change often, usually driven by the constant changes that a modern software product requires to stay relevant. Software at this layer isn't meant to be brittle or unreliable, but the people building it will often prioritize iteration speed over performance or resiliency.

Rocks are how we call the underlying software that enables many different use cases, the software that is so close to the core business that it will probably only change if the business model changes. Many other services depend on services from this layer, which means that they should be built and maintained with resiliency and performance in mind.

Services are usually born as *clay*, as the team is experimenting with new products and features. If the experiment finds product/market fit, they are usually moved down as more and more newer products and features start building on them.

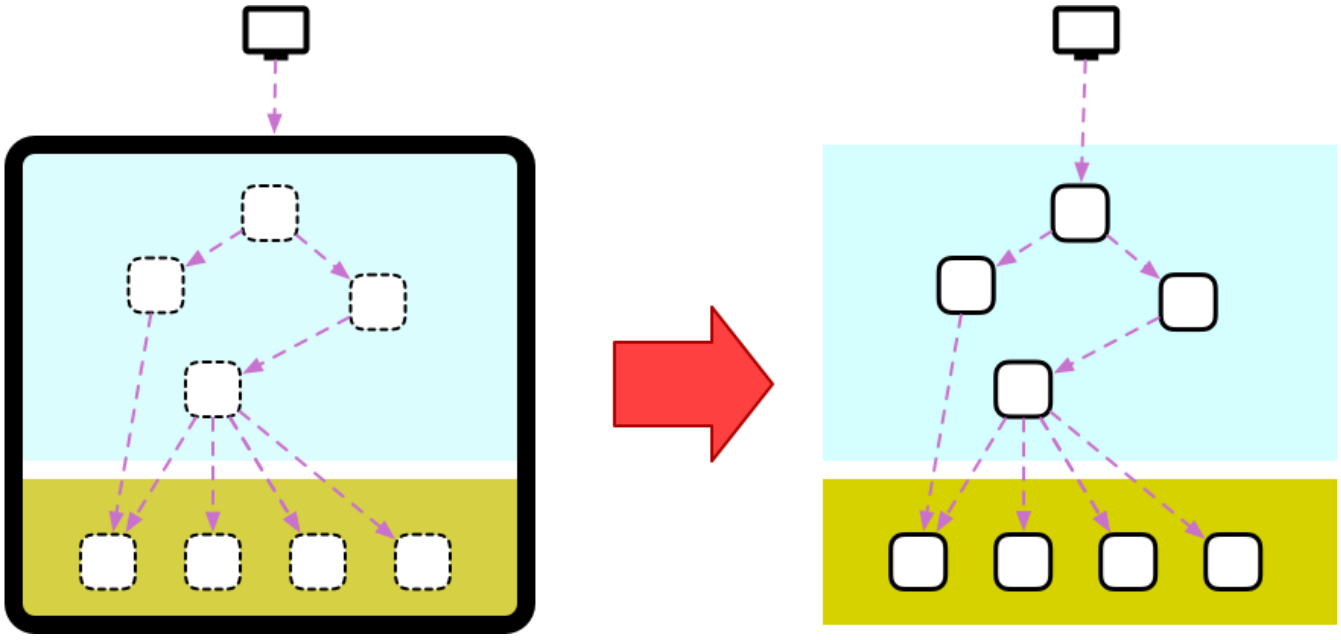
When migrating from monolithic to microservices

Acknowledging the differences between *rocks* and *clay* is a common trait of most successful migration projects I've been part of. When organizations get to a point in their journey where they consider splitting the monolith, they usually have a stable core product but find it hard to iterate on new features or experiments quickly. In most cases, this has to do with how both *clay* and *rocks* share a single system, the monolith.



In such a scenario, the development cycle happens at a slow pace because even the smallest change to a feature at the *clay* layer can inadvertently affect one of the *rocks* and take the whole thing down. Code reviews, manual testing, slow rollouts, and many other change management techniques need to be added to a process, which makes the feedback cycle longer and longer.

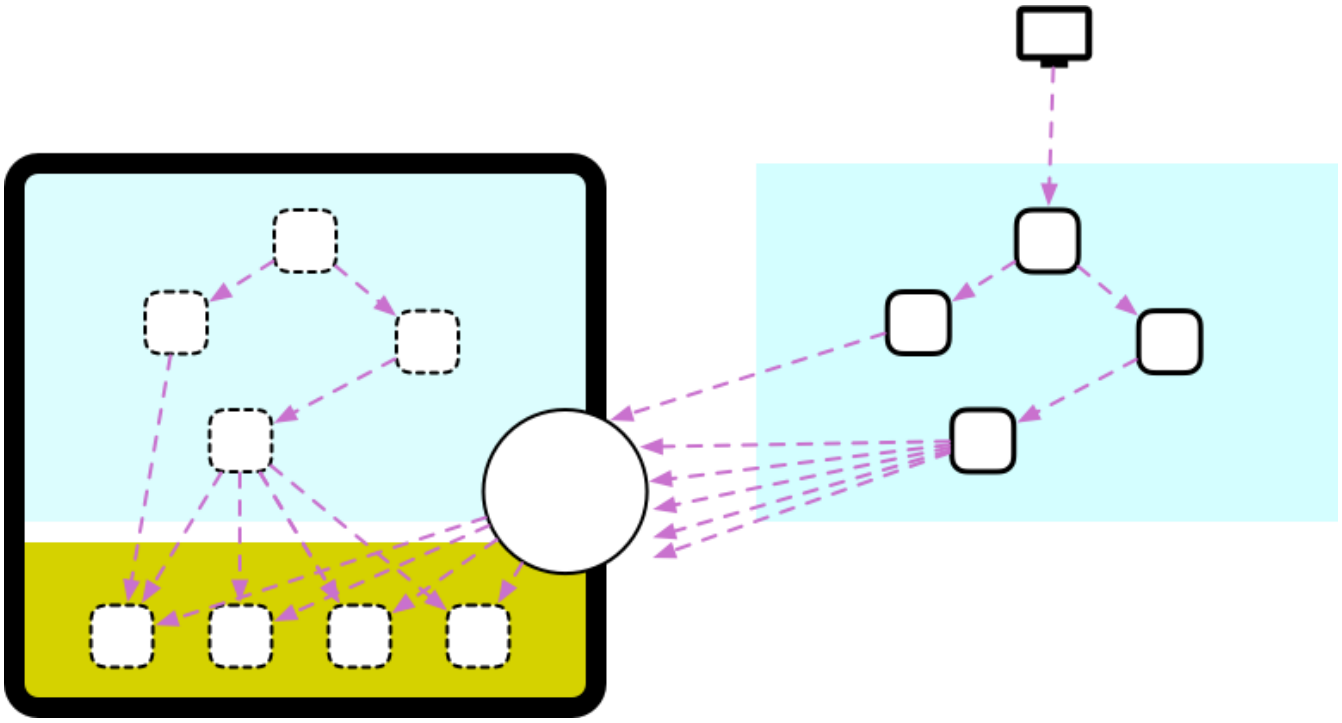
It is very common for organizations at this stage to organize their engineering teams around a big effort to “split the monolith,” extracting services from it. In principle, the plan looks simple:



Unfortunately, I have never seen such an effort go well. Things usually go well as long as we deal with extracting *clay* services. In fact, logic at this layer tends to be so thin and coupled to the user experience that these can be often rewritten with a nice UX refresh.

The real problem reveals itself when people attempt to extract the *rocks*. Not only do these have stricter non-functional requirements, but there are also so many other subsystems that depend on them that it becomes almost impossible to remove one of these things without rewriting half of the monolith.

One approach that I have had more success with, something that classic Monolith-to-Microservices cases such as Twitter or SoundCloud have done, is to focus on extracting your *clay* objects and not worry at first about your *rocks*. What you should do instead is to expose these objects internally, building something that is sometimes called a *backdoor API*.

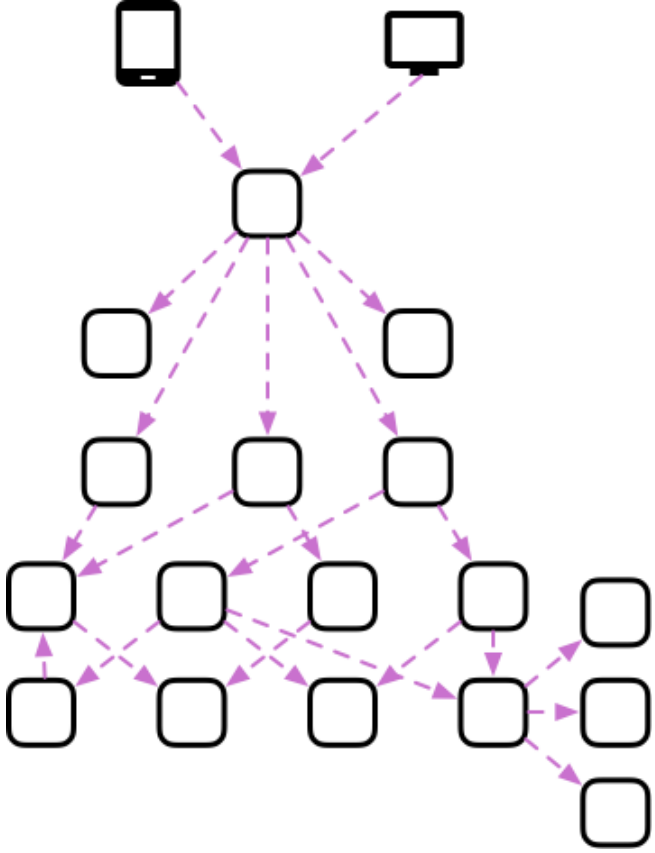


With an approach like this, one can extract the *rocks* over time, while still iterating on your product. It is very common that you never actually get rid of the monolith, but over time it becomes less and less part of the critical path, as the team either extracts objects or the business needs to change, and the new domain is implemented as microservices from the beginning.

Pattern: Edge-Internal-External Layering Model

One important perspective when visualizing distributed application architectures is to be able to place services based on where they live in the network.

Most architectures will have a variation of the model below:



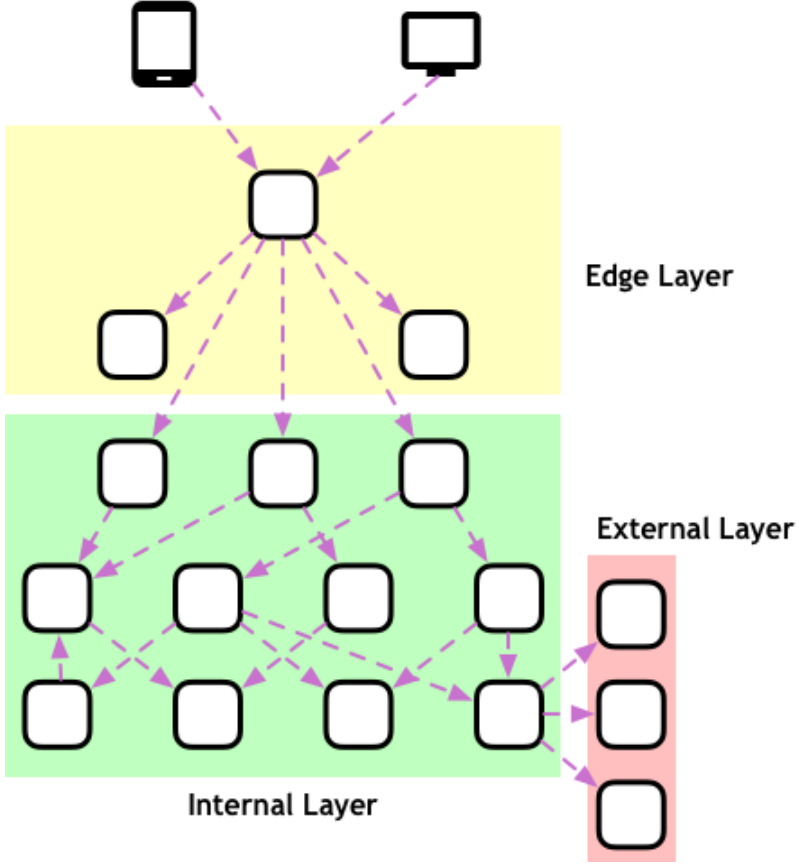
Where the user interacting with an application through something like a web page, mobile app, or API will generate inbound traffic to your services. Irrespective of how many services you might need to fulfill the task at hand; the user request usually hits a single service. This service is often called an *API Gateway*, and it is responsible for figuring out which of your many microservices to call in response to this specific request.

Something that might not be clear from the diagram above is that the API Gateway will often delegate some of its own responsibilities to other components. Unless you want to build a monolithic API, concerns like user authentication, geolocation, rate limiting, and A/B testing should be separate services.

These auxiliary services are different from your typical microservices in many interesting ways. Not only don't they implement application logic, directly related to your core business, but they often have stricter requirements related to availability and scalability. Another trait these components share is that they deal with data coming from the outside world and need to sanitize it before forwarding it to internal service. This means that they have to apply a fair amount of *defensive programming*.

Because these services are under somewhat strict requirements, performing changes to them tends to require a more careful process—e.g., you might want to execute performance tests before deploying modifications to these critical-path systems or a security audit before changes to authentication logic. You will probably need a more sophisticated approach to deploy changes at this layer, maybe with green/blue deployments, as any downtime here will take your whole product offline. All this makes the development cycle of components at this level slower than other services, as there is a higher risk of wide-reaching incidents.

Although the overhead is justifiable for these special-case components, we definitely do not want such a slow-moving pace for our regular microservices. One way to help an organization understand which components have the stricter requirements versus which ones are your usual fast-moving pieces is by applying a layering scheme that I like to call *Edge-Internal-External*:



In this model, we explicitly model the services described above as what I call the *Edge* layer. They are the entry point that receives requests from users and does everything required to translate them into requests within your architecture safely.

We then have services in the *Internal* layer. Those will be the vast majority of your microservices, and they can make a lot more assumptions about their clients and environments, including that those requests have been sanitized, have metadata for distributed tracing, etc.

There are also services in the *External* layer. These are services that our systems talk to but are not developed by us or deployed in a way that we can control, usually third-party services.

The Edge Layer itself might be implemented in many different ways. Several vendors offer *all-in-one* options that allow people to outsource this completely, products like API Gateways or Service Meshes. Organizations working at a higher scale or with more complex requirements might want to build and own at least parts of their Edge architecture themselves. This is especially true if they are not happy with the monolithic nature of the products available in the market and want to apply a microservices architecture to this Layer.

Given the focus on availability and performance for components at this Layer, it is common that the team that owns it isn't a Product Engineering team, but falls under *platform* or *infrastructure*.

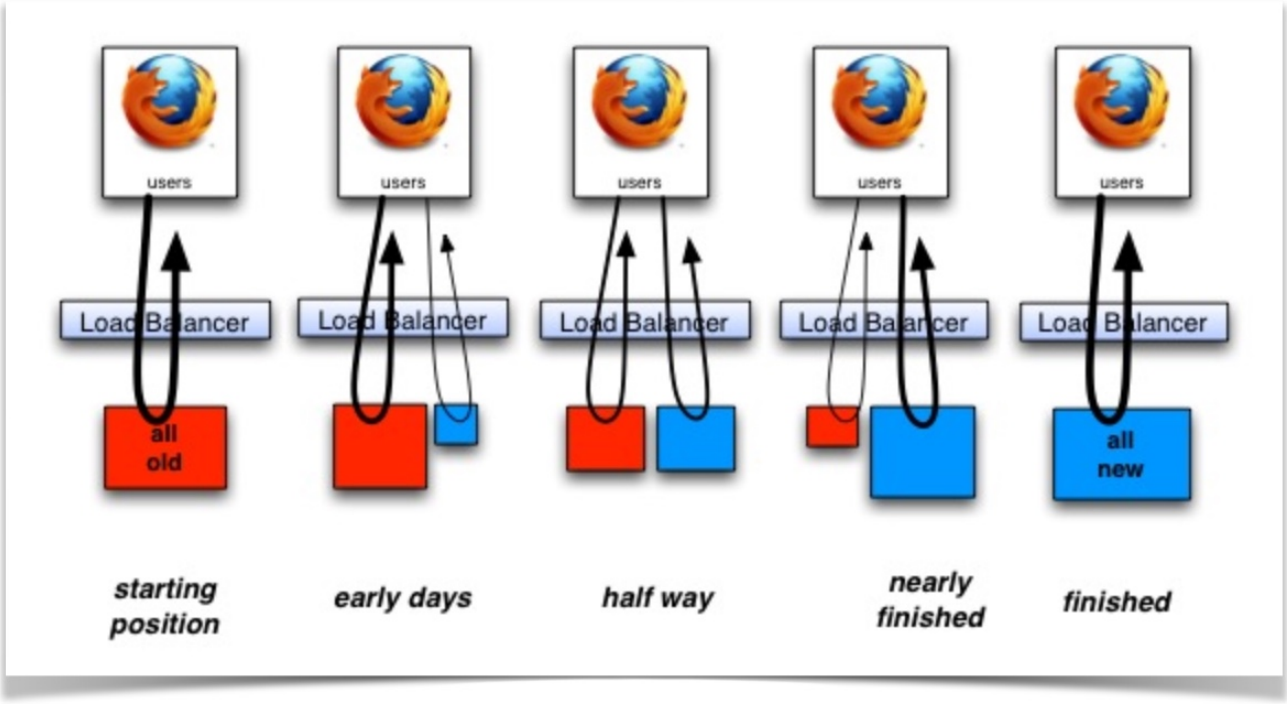
When migrating from monolithic to microservices

Similarly to the Clay-to-Rocks Layering Model, the criteria used by the Edge-Internal-External model has some correlation with grouping components by how hard to change they are. Namely, the Edge layer often requires stricter change management, similar to the *rocks* in the other model. Nevertheless, these models aren't the same, and there are some subtle yet fundamental differences between them.

One way in which such differences can arise is when migrating from monolithic to microservices architectures. The Clay-To-Rocks model suggests that you leave your *rocks* inside the monolith for as

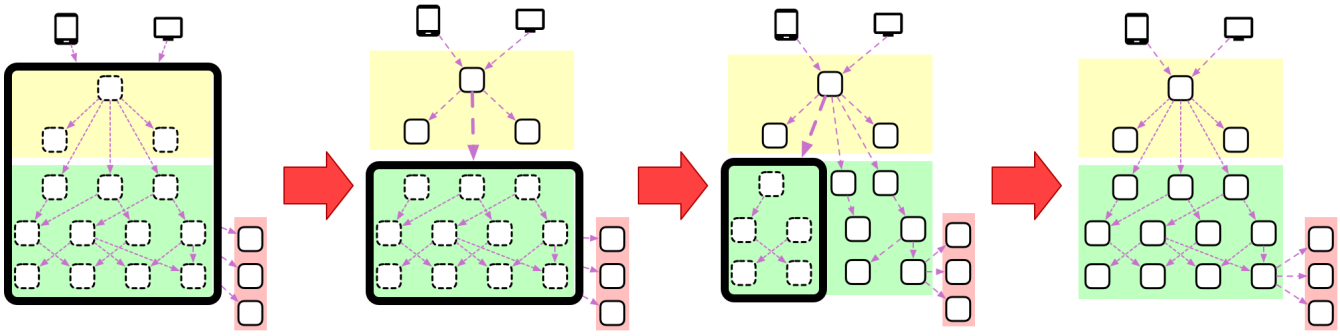
long as you have to. In an Edge-Internal-External layering scheme, though, we have the Edge as a high leverage point, meaning that a small effort applied here can cause a massive improvement throughout the whole systems.

A very popular approach when using the Edge to drive an organization away from monolithic systems is by using the Strangler Pattern, first cataloged by Martin Fowler and described in more detail in a seminal article (and diagram) by Paul Hammant:



The basic idea behind a strangler is that you put a piece of middleware between the user and the legacy system. At first, the middleware will redirect all requests it receives to the legacy system, and return its responses to the user. You can then incrementally write replacements for subsystems of the legacy and deploy them into production. The middleware is smart enough to redirect traffic destined to that subsystem to the new implementation (often by inspecting the URL requested) while still redirecting all other traffic to the legacy system. Eventually, more and more subsystems get written, and the new versions replace the whole legacy application.

In our Edge-Internal-External model, the Edge layer offers an intuitive place for such a strangling point. A widespread approach to microservices migration is to start by removing this layer from the monolith. At this stage not only you can slowly extract logic from the monolith in their own microservices without changing any of your client applications.



Another advantage of this strategy is that you can also make sure that any new features are already implemented as microservices and still have access to vital features such as authentication and caching. In my experience, the biggest challenge in a large refactoring effort such as microservices adoption is to

make sure that while a team is extracting logic from the monolith you don't have other teams adding to it. This pattern offers you a way to clean-up your old systems without blocking people from working on new features.

The complexity of managing complexity

So much of software architecture is about keeping complexity under control. Layers can be a great way to contain entropy around your system, but sometimes it happens that teams fall in love with the pattern and start overdoing it. When using Layers, I recommend that you first start by applying a few simple models like the above. Any model with more than three or four layers is a bad smell to me—maybe you are trying to bundle together two different layering models?

Another aspect of using an architectural pattern is making sure that all engineers understand the *why* and *how* of it. It does not matter how many fancy diagrams you have buried down some Confluence page if your team doesn't appreciate your layers they will either completely ignore it or spend a lot of time debating if a given service should be on layer X or Y.

Like any other tool in enterprise architecture, layers are only useful when they are simple and widely understood.

Acknowledgments

Etel Sverdlov, Vitor Pellegrino, José Muanis, Thompson Marzagão, Brian Gruber, Danilo Sato, and Douglas Campos gave feedback on drafts of this article.

Revision History

- 09/24/2018 - First published

Phil Calçado

Phil Calçado

pcalcado
pcalcado