Engineering

A 6-point plan for implementing a scalable microservices architecture

 Rimantas Benetis11/14/2016

In a previous post, I outlined the many benefits that a microservices architecture brings, if implemented correctly. However, one of the first steps along the path to a microservices architecture is determining if your organization really needs microservices. If your organization has made this determination and is ready to dive in, we find that companies that are successful have a specific, actionable implementation plan. Following is an approach that we find works for many of our clients.

Before delving into the details, I want to emphasize that you don't have to choose the full-blown options outlined in this post. Rather, your organization can abstract your actual implementations and leverage a microservices architecture by defining interfaces for some of your "infrastructure" services. This would allow for you to choose appropriate implementations and decouple you from cloud services as well.

Following are the parts we suggest you consider to ensure that you have a healthy path to a microservices architecture.

1. Serve a business purpose

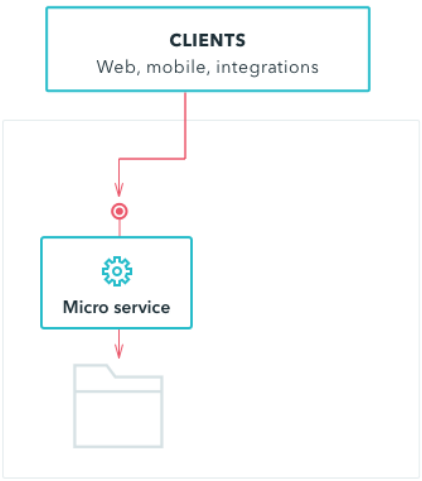
Suppose that you have a single functionality service. Your team should be able to handle this just as easily as a self-sufficient product that should have:

1. Automated deployment (this is good even for monolith applications)
2. Exposure to other systems (accessible endpoint)
3. Provisioned storage (if any)
4. Scalability and load balance (depending on resiliency requirements); just double-up the infrastructure and load balance service

If the service is designed well (stateless) this should be pretty standard and not different from a typical application.

However, let's put this into perspective. Assume that you will have 100 microservices, which changes the above requirements. To manage a large number of services, you would need to automate all of the above requirements. This should already be part of your current development (automated provisioning, deployments, etc.) anyway. However, this is worth noting since it will become an obstacle when you try to deploy hundreds of services. Sorting out this step in advance will actually make your current delivery process better, even if you decide against a microservices architecture.

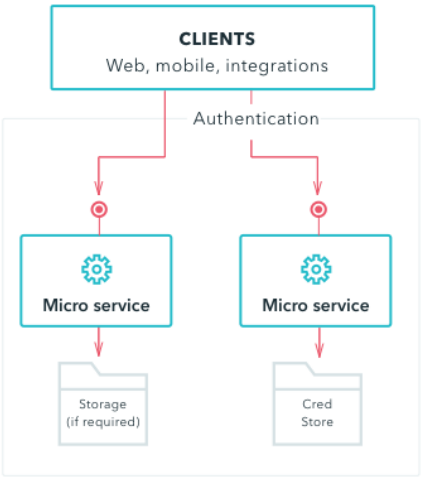
At the conclusion of this step, you end up with something similar to the following:



2. Protect your stuff

Once you have your single service, you will immediately notice that it needs some kind of protection. This could be built using middleware and work pretty well "out of the box." However, when you scale up a hundred times, you will probably want a uniform way to secure your services. For REST API services, for example, you would typically want some kind of SSO. This would require some means of authentication. If your organization already has some kind of SSO that you could leverage, you can simply integrate with that system. If your organization does not have some type of SSO, then you'll need an authentication service to handle your authentication requests. At this point, it's probably safer to go with an open source, tested software. A commercial software, custom building proper (secure) authentication service is expensive, and you need competence.

If you do not yet know what's going to be behind the scene, you may want to abstract your authentication via microservices as well, so that you can easily interchange the underlying product. This also would allow you to easily integrate security in the early stages before a final decision is made.



After you choose an authentication service, it makes sense to have only a single service throughout the entire organization. That quickly becomes a project in its own right!

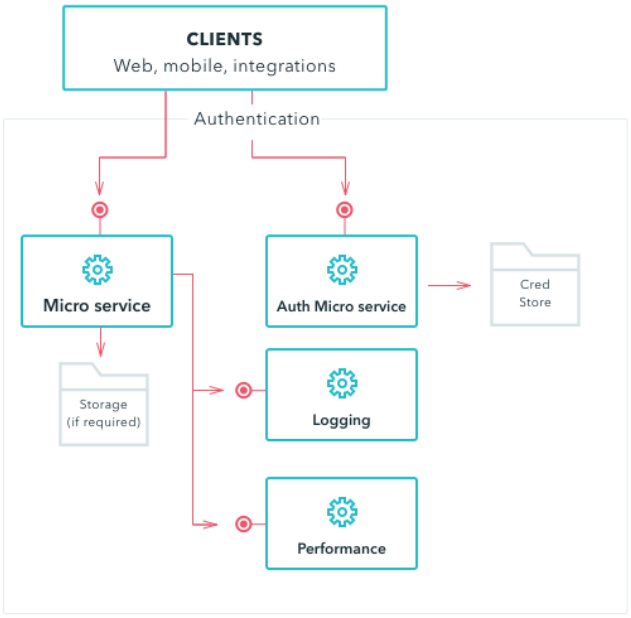
3. See no evil, hear no evil

As soon as one of your microservices is deployed, you immediately need to start monitoring it. At this point, you should already be using some libraries (such as log4j, log4net, etc.) that support proper logging and has various adapters for your convenience. As soon as you scale this to hundreds of microservices, however, you will encounter difficulties unless you have proper log inspection tools, such as cloud-based services loggly or splunk. Your application support team will not appreciate having to dig through mountains of data to find issues (not to mention alerting). I've seen instances where application support had 15,000 issues in their log! Guess if they actually had time to go through all of them.

To avoid this chaos, you should create a uniform way to log. This could be done either by using some libraries in your microservices (most respectable cloud providers have SDKs for myriad languages) or by creating an abstracted logging microservice that would have proper implementation later.

As soon as you define your interfaces, it becomes reasonable to use that service for the entire organization. This again becomes a project in itself!

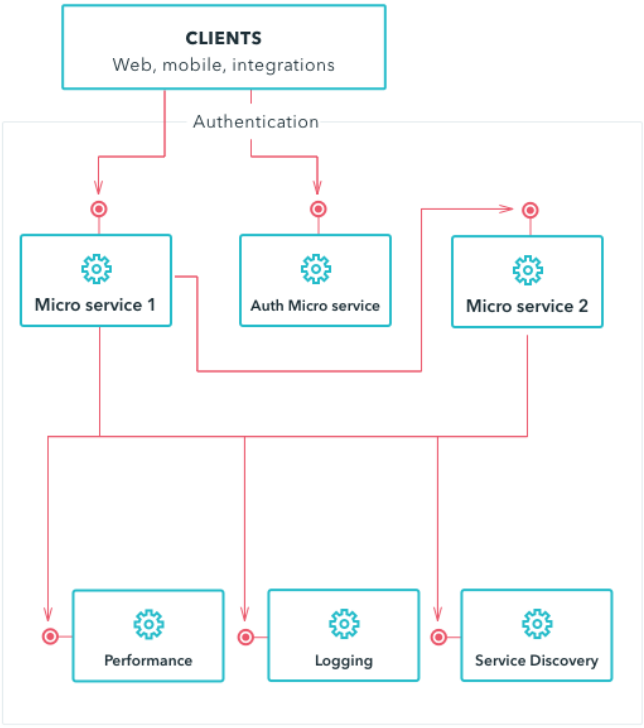
The same approach should be used for performance monitoring. It's fairly easy to profile a single service and run all the performance/load tests. However, as soon as you scale, it becomes impossible to cover all scenarios. You would need to have a discussion with your application and system support teams regarding how you want to manage scalability issues. Again, you can choose to abstract out performance monitoring calls into a microservice and decide on a backing tools later. Yet another project in itself!



You most likely see a pattern emerging that all cross-cutting services can become a huge headache for a small project team. All of those infrastructure (supporting) services are products in their own right. To make them usable by the entire organization involves numerous things. Unless these cross-cutting services are readily available for consumption and a fair share of enterprise thought went into them, a project team will be unable to come up with good solutions at the same time as they are trying to deliver business functionality. What I see typically happen are hacks being created with the intent that they will be replaced with proper solutions in the future. However, during this time, new organization-wide initiatives crop up, resulting in numerous support service hacks, which are usually used by a single product (although the support effort never goes away while the product is active).

4. Find your stuff

When creating products using microservices, you will quickly encounter that dependencies between them needs to be managed somehow. In a monolith configuration, files are describing all the "connection" strings. Looking at one service this sounds like a good approach. However, as soon as you start scaling and adding unsymmetrical scaling (some services are deployed to more nodes than others) you'll notice that configuration files are not feasible. Here comes **service registry**. There are multiple open source tools that can solve these issues (e.g., Consul, Etcd, ZooKeeper). Until you have a tool or process in place, you'll have to improvise. The best suggestion under this scenario is to abstract the service registry as well. As I mentioned previously, abstraction could be done by creating a microservice, or, on a component level, by creating a library that could be used in your services. One additional benefit that you get is that it also could act as your service version tracing repository. It is easy to build additional functionality that would flag services that are not requested anymore. This would create a list of services that need to be decommissioned. The same could be used to create dependency graphs.

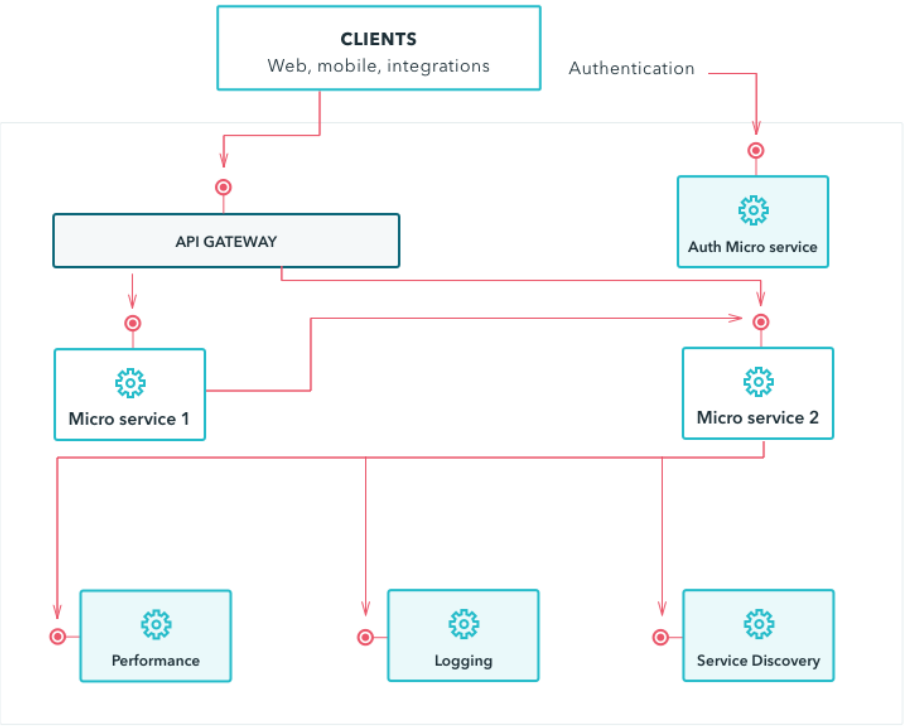


As you can see again this service is organization-wide and can become pretty sophisticated as well (surprise, surprise).

5. Create a gateway

At this stage, you now have a secured, properly monitored single service that is automatically deployed into production and is capable of resolving its dependencies. Congratulations! This is what your current software should look like. However, one certainty about software development is that software will change. Microservices enables you to create new software, update existing software and deploy software more rapidly. However, you do not want to expose your clients to ever-changing internal implementations. Here is where an API gateway comes into play. An API gateway is also a microservice that can have a numerous responsibilities. Firstly, it has to be highly available and its primary function is to route client requests to your underlying microservices. Additional functions that immediately could be identified are:

- **Generalized security (an API could validate all incoming requests, so internal services don't have to recheck)**
- **Performance metrics could be recorded for all requests**
- **Auditing**
- **Transforming single or multiple incoming requests into microservices**
- **Abstracting the client interface from internal interfaces so your clients (including web applications) do not have to change their integrations each time your internal microservices changes**
- **Implementing dynamic request transformations, routing, and combination (e.g., Netflix Zuul)**

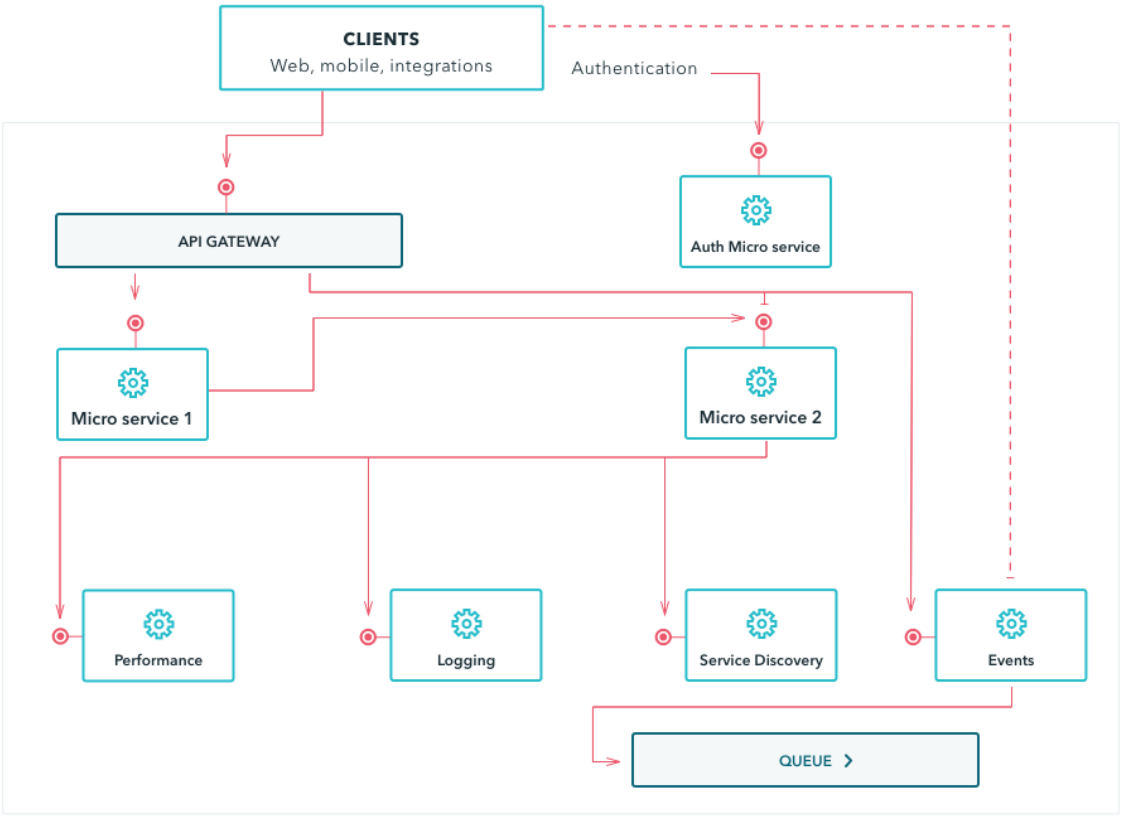


If you need all of the functionality above, this is not a simple task. At this stage, you would have a pretty good foundation for your subsequent services and rolling them out would be a breeze. After all that hard work you will be able to concentrate on business logic only.

6. Construct events

Most microservices solutions requires some kind of asynchronous processing. Also, if you are using CQRS, this piece of the puzzle becomes a must. Its event queues microservice. In this space you can directly use message queues to pass your async tasks. If you do not have a decision here, you could create a microservice event that would implement publish/subscribe endpoints. This way you can:

- **Schedule tasks (simply by dropping events)**
- **Subscribe for specific events (your message handlers becomes simple micro services)**
- **Expose your subscribe functionality to external clients to handle various notifications**



The end result

To get you to a point where adding additional functionality, scaling, and deployment of your products becomes a breeze you need to think about a number of intricate pieces. Some are DevOps, some of are application support, and some are development process changes. If your organization is wanting to go with microservices, my suggestion would be to evaluate if it's really necessary. Answer honestly if it's needed for your product and ensure that you will have support from management and application support. This is a tremendous organizational endeavor and you'll need all the support you can get. Also, some practices such as continuous deployment, monitoring, and alerting is good even if you don't go with a full-scale microservices architecture. As microservices become much smaller in scope and teams have more autonomy, your development process will have to be agile as well to reap its benefits.

So the answer is that for some organization it is not feasible to go with micro services but for some it may bring a lot of benefits and agility. There is no single answer that fits all.

Moving to a serverless architecture

After you get all of your bits and pieces working well in a microservices environment, there is one additional step that could be done--serverless architecture. If you standardize your microservice "run-time" (executable part of a service) you can start deploying single-route processing routines. This would allow you to scale your run-time as required, which hooks up to your processing routine storage, and execute as required. Your team, instead of always writing microservice, would be writing processing routines, which could be enabled (i.e., deployed) at run-time. This would require standardization for processing routines, but a lot of code in a micro service run-time is the same. Also, when you update run-time and redeploy, you get new features for all your processing routines.

Conclusion

The path to microservices may appear to be easy at first glance, but many things need to be taken into account from both an application development and organizational perspective. It is best to take these into consideration in advance lest you be forced to make important decisions mid-project, or worse, abandon the entire project after investing a lot of time and resources. Once you have an idea as to which parts you need to get into place you can start incremental implementation. Keep in mind that you can abstract a lot of pieces initially to get you going, but you'll need to finish them off at some point. Hence, it's always important to have what's next in your line of sight.

Related Posts

Path to microservices: moving away from monolithic architecture in financial services

Want more industry news?Next

- **Privacy Policy**

- **Referral**

- **Sitemap**