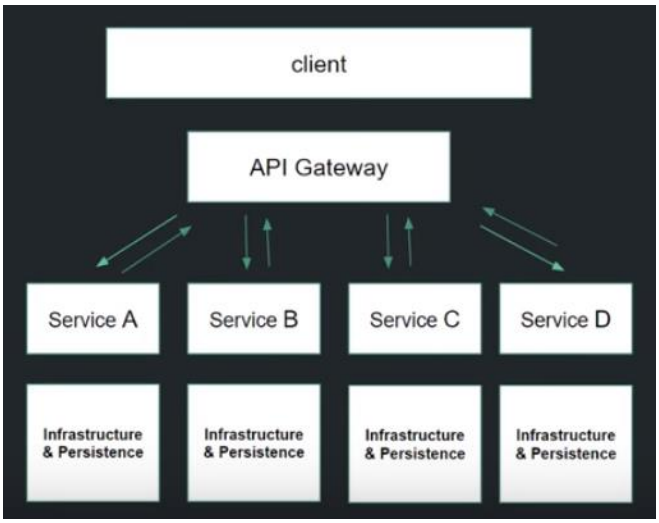
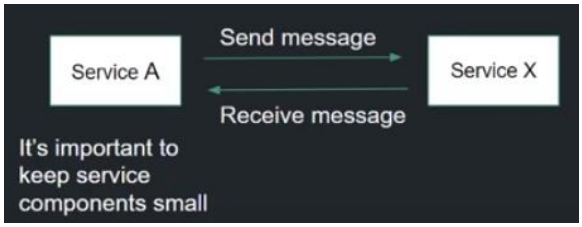


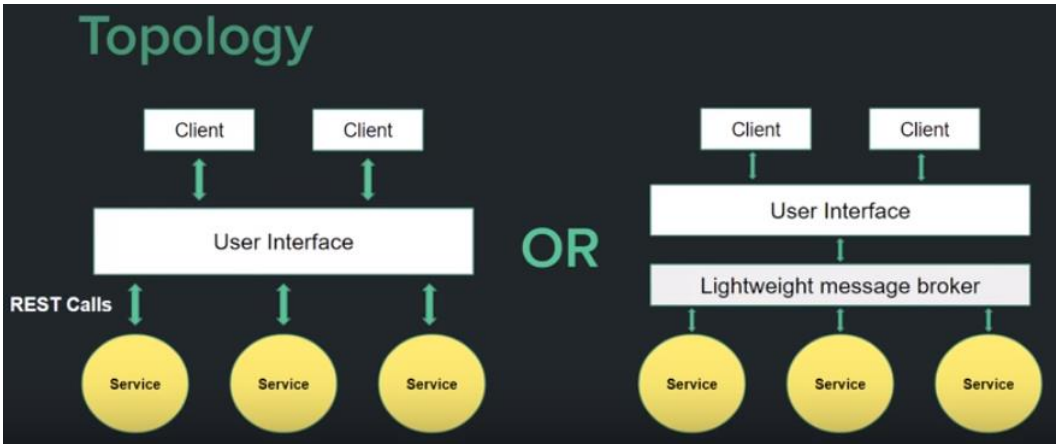


1, For small to medium applications

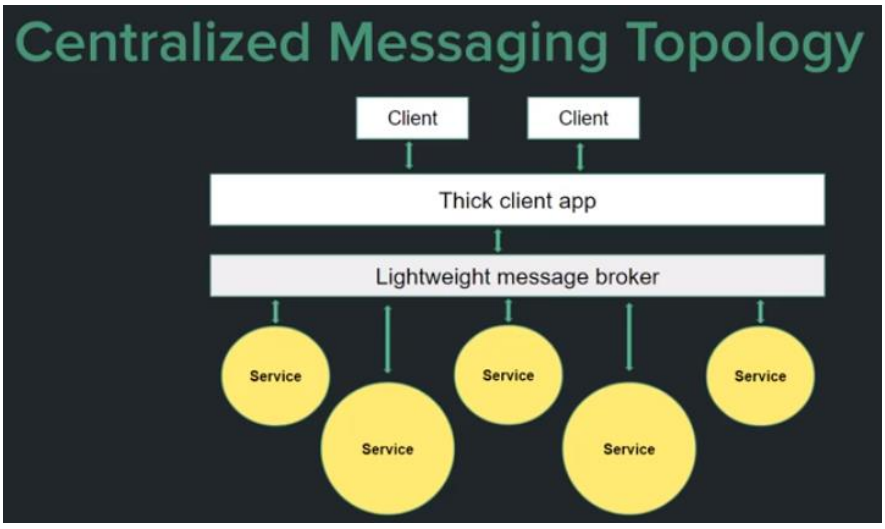
< REST API based Topology >

2, For large applications

< REST Application based Topology >



3, with lightweight message broker



Advantages and Limitations

Pros:

- Scalability & Extensibility
- Scaling at the service level
- Future-proof; Optimization, switching technology, adding features, maintenance...
- Ease of Deployment and Testing
- Reusability
- Default Architectural pattern in the cloud
- Fault Tolerance

Cons:

- Extra work; service contracts maintenance, handling service failures...
- High latency
- Monitoring
- Transactions
- Testing at the application level

Planning for Microservices

1. Is Microservices the right choice? (Evaluate tradeoffs)
2. Identify the main functionalities of the system.
3. Determine service components' scopes; granularity
    - Size of the functionality
    - Type of the functionality
    - Complexity of the functionality
4. Design the interfaces(APIs) of service components
5. Design the API layer for whole the system
6. Determine the communication mechanism
7. Determine a data persistence model(central database vs multiple data stores)

Microservices for Greenfield Projects

Anytime your team develops a new application from scratch, it feels great not to inherit technical debt and be locked into outdated decisions made years ago. Most teams developing new apps today would probably choose to containerize them using Docker and adopt microservices architecture for speed and agility.

However, there are a few things to consider before you start:

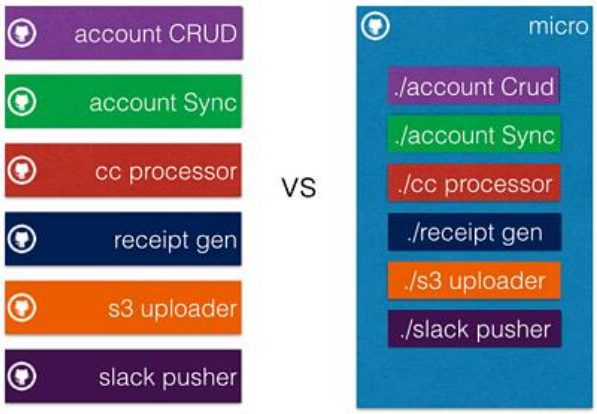
1. Degree of Independence

The first decision is - how independent do you want your services to be? You can choose one of the following options:

- Each service is completely independent with its own Database and UI. This is consistent with an extreme microservices architecture where services really share nothing and are completely decoupled. The nice thing is that the team for each microservice can choose the database that best addresses their requirements. However, this approach makes it much harder to ensure that all data stores are in sync and consistent. For example, you need to verify that the same userIDs exist in all data stores, and there is nothing missing in any of them. Also, database management tasks like backups need to replicated by each team.
- You can choose to share some components, usually your Database. This makes it easier to enforce standards across all teams and ensure data consistency across all services. However, this also means your services are not completely decoupled and if someone updated a table or schema, other services might be affected as well.

Both approaches have pros and cons so you need to decide what you can live with. At Shippable, we chose the second approach since we did not want to deal with inconsistent data and then spend time finding ways to ensure consistency.

2. Code Organization



There are several ways you can organize your codebase. You can create **a repository** for each service or **a** single "**mono repository**" for all services and have a folder for each service.

The pros and cons of both approaches were discussed in our blog post on "Our journey to microservices: mono repo vs multiple repositories" where we explained why we chose the **mono repository** approach.

> **Better developer testing:** Developers can easily run the entire platform on their machine and this helps them understand all services and how they work together. This has led our developers to find more bugs locally before even *sending* a pull request.
>
> **Reduced code complexity**: Senior engineers can easily enforce standardization across all services since it is easy to keep track of pull requests and changes happening across the repository.
>
> **Effective code reviews:** Most developers now understand the end to end platform leading to more bugs being identified and fixed at the code review stage.

Sharing of common components: Developers have a view of what is happening across all services and can effectively carve out common components. Over a few weeks, we actually found that the code for each microservice became smaller, as a lot of common functionality was identified and shared across services.

Easy refactoring: Any time we want to rename something, refactoring is as simple as running a grep command. Restructuring is also easier as everything is neatly in one place and easier to understand.

The results? **Our productivity has increased at least 5x.**

The best thing about our move to a mono repo is that we didn't give up any of the advantages of the microservices architecture.

We believe mono repository are the right choice for teams that want to ship code faster.

3. Technology Stack

It is difficult enough to decide on the technology stack for a monolithic application, but now this decision has to be made for every microservice. What seems attractive in theory can become problematic in practice if your services are too heterogeneous. Standardization becomes a problem and there is potential for cowboy behavior. Also, it's harder for people to move between teams if every team is using a completely different stack.

We recommend a balanced approach where there is a preferred technology stack across the application. If any team wants to override this default stack, they should justify their decision with pros and cons of why a different stack is more suitable for their microservice. Your technology stack should include programming language, testing and logging framework, cloud provider, infrastructure, storage, monitoring, etc.

4. Operational Complexity

Microservices greatly increase operational complexity since you need to rethink operations from a very fundamental standpoint. You need to consider the following aspects:

- **Infrastructure**: Defining infrastructure requirements for microservices and then provisioning and maintaining infra for each microservice adds a degree of complexity that most Ops engineers working on monoliths are not accustomed to. Plus, as these services are scaled up and down, infrastructure needs to be provisioned and brought down automatically so you need a very sophisticated level of automation.
- **Load balancing and scaling**: You will likely need a scaling strategy that is much more complicated than for monolithic applications, which are always scaled out (x-axis). With microservices, you will need to figure out if you need to scale all services or just a subset when there is a spike in demand. Your Ops team will need to understand y axis scaling since the microservices approach is consistent with it and z axis scaling to get the benefits of x and y. More on the scalability cube here.
- **Service discovery**: The set of service instances in a microservices world changes dynamically due to scaling and upgrades. Also, services have dynamic network locations, so you need a way for new service instances to be discovered. We recommend a service registry like Consul for this since it has worked very well for us. Read more on client-side service discovery, server-side service discoveryand a list of commonly used Service Registries.
- **Monitoring**: This has to be configured and maintained for every microservice, increasing the complexity for Ops engineer(s). Also, the monitoring solution has to handle scenarios where a subset of services are scaled up and down.

The operational complexity itself should give you some pause before you decide to move to microservices. Unless you are aware of the challenges with microservices and have a plan to address them, it will be a painful transition.

5. Continuous Delivery

We agree with Martin Fowler when he says this in his blog article about microservices tradeoffs:

> **"Being able to swiftly deploy small independent units is a great boon for development, but it puts additional strain on operations as half-a-dozen applications now turn into hundreds of little**

microservices. Many organizations will find the difficulty of handling such a swarm of rapidly changing tools to be prohibitive.

This reinforces the important role of continuous delivery. While continuous delivery is a valuable skill for monoliths, one that's almost always worth the effort to get, it becomes essential for a serious microservices setup."

Organizations that evangelize microservices, like Netflix and Amazon have the resources to build homegrown custom continuous delivery pipelines for their microservices. However, not every organization can afford to do that. Even if you can afford it, you should think about whether your time is best spent building fragile homegrown automation that has to be custom for each microservice, or whether you want to improve your own product instead. You have three choices:

- Decide that you do not want to pay the microservice premium and stay with a monolithic architecture
- Bite the bullet and build homegrown pipelines by cobbling together several different tools that help with parts of the workflow. The problem with this approach is that as the number of microservices increases, the time and effort required to automate them increases at an even faster rate
- Use a CD automation platform like Shippable that will get you 90%+ of the way towards complete continuous delivery for heterogeneous microservices.

I will publish a blog post on the third option in the next few days as part III of this blog series that will also show how Shippable can help with Continuous Delivery as well as infrastructure automation, scaling, and service discovery.

6. Team Organization

Last but not the least, you will need to reorganize your team(s) to ensure that every microservice is really developed, deployed, and maintained independently. You can't have your engineers working on multiple microservices since that will invariably lead to decisions that optimize for factors that don't correlate with what's best for each microservice.

An independent, self-contained team should work on each microservice. Amazon's 2 pizza team approach gives a ballpark team size - it should be small enough that everyone on the team can be fed with 2 pizzas for dinner, i.e. less than 10 people. Each team should be balanced with expertise across Dev, Test, Ops, DB administration, UX, and even product management in some cases. This doesn't mean you need a unique team member for each role, just that they should be addressed within the team. For example, a DevOps engineer satisfies dual roles of Dev and Ops. Similarly, each team can also have a manager who writes specs and defines the UX.

There are variations of the approach above, such as centralizing the Ops, Product Management, or Platform teams. These articles shed further light on the ways teams can be organized.

Microservices for Existing Projects

If you want to move your existing monoliths to microservices, you will still need to consider all the points described above. However, there is an additional challenge specific to your case. You will need a strategy that helps you make the transition in stages.

7. Strategy to Transform from a Monolithic Architecture to Microservices

Just like Rome wasn't built in a day, your transition will take time and dedication. In a nutshell, you need to do the following:

- Implement continuous delivery so that you have the right automation in place as the number of services starts growing
- Move to a distributed source control system like GitHub so that teams can work independently without stepping on each other
- Dockerize your application to get portability and the ability to spin services up and down within seconds
- Always build new functionality as a microservice
- Convert existing components gradually, starting from the least complex business functions with the fewest dependencies and working your way to more complex functions.

Summary

As you can see, adopting microservices is not trivial and should be done only if you see enough value for your applications. I will elaborate further on Continuous Delivery for microservices and strategies for moving existing monoliths to microservices in upcoming posts this month.