Building an Application: Strategies for Microservices | Swagger

By Janet Wagner

Microservices as an approach to building reliable applications at scale has been gaining popularity in recent years. And major companies like Amazon, Microsoft, and Netflix have been talking about microservices for quite some time now. This post looks at strategies for microservices, specifically starting with a monolith and moving to microservices later and starting with microservices right out of the gate.

What is a Monolith?

When developers talk about a monolithic application, they typically mean an application that is treated as a single unit, is tightly coupled, and usually requires a one-time deployment to a single server. A monolithic application is also scaled as a single unit, likely shares one database (usually relational), and often has one huge code base for the entire app. Everything is defined in the code base, e.g. front-end and server-side logic, etc. and usually only a small set of languages are used to build the app. In some cases, a monolithic application could be deployed to a number of servers, but the app would still be treated as one unit.

What are Microservices?

Microservices have created a lot of buzz among app developers, but there isn't an industry agreed-upon definition as to what microservices are. Microservices are more of a concept for developing distributed apps. However, there is a consensus among many industry leaders as to the characteristics of microservices. Among these characteristics are that the services are loosely coupled but bounded by contexts, can be deployed and scaled independently of each other, and communicate with lightweight mechanisms (e.g. HTTP, REST APIs, WebSockets). A short definition is provided in the book "Building Microservices" by Sam Newman: "Microservices are small, autonomous services that work together."

Building Your App – Monolith or Microservices First?

When it comes to strategies for microservices and APIs, it largely depends on whether you decide to go with a monolithic application first and then later move to microservices or start with microservices right out of the gate.

Industry leaders have different schools of thought regarding monolith or microservices first. Martin Fowler, a well-known author, software engineer, and chief scientist at ThoughtWorks, advocates a monolith first approach. He advises that it is better to start with a monolith and then move to microservices later if needed. Stefan Tilkov, co-founder and principal consultant at innoQ, advocates a microservices first approach on the grounds that the components of a monolith will eventually become very tightly coupled to each other. This makes it very difficult to split the monolith into individual services later.

It should be noted that there are many approaches to application development, it's not just a binary choice- monolith or microservices first. For example, Darby Frey, senior engineering lead, platform, Gamut, explains in an article on Medium that his team decided against doing monolith or microservices first. The team built two apps instead — not really microservices but not a monolith either. The architecture the team built includes an API gateway layer which allows them to have as many backend services as they want.

There are many things to consider before choosing an architecture for your application. If your development team doesn't have any experience building microservices, then a monolith may be a better option. If you want the features of your app to be cutting-edge, then you would need to use the best language, framework, and library for each feature. In this case, microservices would be the better choice. This is not to say that you can't build a monolith with cutting-edge features. But a monolith tends to be limited to only a few languages which means you may end up building features with a language that isn't the best suited for all the features of your app.

Starting with a Monolith

What are some of the reasons you would want to start with a monolith? Compared to microservices, it's typically easier to deploy a monolith because it's usually one deployment on one server. And because the app is treated as a single unit, it is much easier to monitor and perform end-to-end testing. Compared to a monolithic application, monitoring and testing a distributed app is more difficult because the services often have different runtime environments. In addition, when a service needs to be tested, all of the other services that service depends on would need to be launched as well.

So, why would you not want to start with a monolith? One of the biggest reasons you may not want to start with a monolith is that monoliths eventually end up with numerous tightly coupled components. This makes scaling and maintaining the code very difficult. Another reason is that because of all the tightly coupled components, the code base often ends up hard to understand which makes it difficult to onboard new developers.

Starting with Microservices

What are some of the reasons you would want to start with microservices? Because microservices are autonomous, services can be developed, deployed, tested, and scaled independently and quickly. Services can also be developed using any language which means developers can experiment with new technologies and add new features without interfering with other parts of the app. And if there's a problem with one service, the other services will likely remain up and running. The risk that the application will go down because of a problem with one or even multiple services is significantly reduced.

API Evangelist Kin Lane has an interesting take on designing microservices in that the OpenAPI Specification could be used to hammer out all the

components of microservices before writing any code. Documentation, mock representations of services, discovery mechanisms, and many other microservices components could be generated using OpenAPI and OpenAPI-related tools.

So, why would you not want to start with microservices? One of the reasons you may not want to start with microservices is cross-cutting concerns as they tend to pop up as microservices are built. Security and cross-cutting concerns such as logging, caching, and authorization are difficult to tackle when it comes to microservices and often require a lot of development. Some companies will use an API Gateway so that cross-cutting concerns and security are implemented in a centralized place.

Another reason starting with microservices may not be the best option is if your team does not have enough experience building microservices. A microservices architecture is inherently complex, so it is risky to build microservices if your team is not experienced in doing so.

Communication Between Microservices

Microservices are autonomous services, and applications with this type of architecture often have dozens, sometimes hundreds of services. These services need to be able to communicate with each other and with clients. Microservices are often deployed using containers that use an inter-process communication (IPC) mechanism to interact with each other. The most common methods of communication for an IPC mechanism, especially if it involves a request-response cycle, are RESTful. Which often means the use of HTTP/JSON APIs.

Planning a strategy for communication is crucial as services must be able to communicate efficiently and reliably. There are many challenges when it

comes to communication between services such as latency, failures, and routing. A recent article by Christian Posta, chief architect, cloud application development, Red Hat, talks about how calling services and getting services to talk to each other are among the hardest parts of developing microservices.

Choose An Approach That Best Suits Your App

Microservices are a popular approach for application development because it allows developers to build apps that are fast, reliable and highly scalable. But distributed applications tend to be incredibly complex. Choose microservices only if the approach is best suited for your application.

Thanks for reading! Looking for more API resources? Subscribe to the Swagger newsletter. Receive a monthly email with our best API articles, trainings, tutorials, and more. Subscribe