SOA Architecture

Wednesday, June 2, 2021 6:50 PM

In short, it provides long-term agility. Microservices enable **better maintainability** in complex, large, and highly-scalable systems by letting you create applications based on many **independently deployable** services that each have granular and autonomous lifecycles.

As an additional benefit, microservices can **scale out independently**. Instead of having a single monolithic application that you must scale out as a unit, you can instead scale out specific microservices. That way, you can scale just the functional area that needs more processing power or network bandwidth to support demand, rather than scaling out other areas of the application that don't need to be scaled. That means cost savings because you need less hardware.

In the traditional monolithic approach, the application scales by cloning the whole app in several servers/VM. In the microservices approach, functionality is segregated in smaller services, so each service can scale independently. The microservices approach allows agile changes and rapid iteration of each microservice, because you can change specific, small areas of complex, large, and scalable applications

Architecting **fine-grained microservices-based application**s enables continuous integration and continuous delivery practices. It also accelerates delivery of new functions into the application. Fine□grained composition of applications also allows you to run and test microservices in i and to

Architecting container and microservice-based applications evolve them autonomously while maintaining clear contracts between them. As long as you don't change the interfaces or contracts, you can change the internal implementation of any microservice or add new functionality without breaking other microservices.

The following are important aspects to enable success in going into production with a microservices based system:

- Monitoring and health checks of the services and infrastructure.
- Scalable infrastructure for the services (that is, cloud and orchestrators).
- Security design and implementation at multiple levels: authentication, authorization, secrets management, secure communication, etc.
- Rapid application delivery, usually with different teams focusing on different microservices.
- DevOps and CI/CD practices and infrastructure

Distributed Data Management

Thursday, June 3, 2021 4:23 PM

Challenge #1: How to define the boundaries of each microservice

your goal should be to get to the most meaningful separation guided by your **domain knowledge**. The emphasis isn't on the size, but instead on **business capabilities**.

When decomposing a traditional data model between bounded contexts, you can have different entities that share the same identity (a buyer is also a user) **with different attributes in each bounded context**

Basically, there's a shared concept of a user that exists in multiple services (domains), which all share the identity of that user. **But in each domain model there might be additional or different details about the user entity**. Therefore, there needs to be a way to map a user entity from one domain (microservice) to another.
There are several benefits to not sharing the same user entity with the same number of attributes across domains. One benefit is to reduce duplication, so that microservice models do not have any
data that they do not need. Another benefit is having a master microservice that owns a certain type
of data per entity so that updates and queries for that type of data are driven only by that microservice

Challenge #2: How to create queries that retrieve data from several Microservices

Challenge #3: How to achieve consistency across multiple microservices

Challenge #4: How to design communication across microservice boundaries

Direct Client to Microservice

Friday, June 4, 2021 11:07 AM

In production environments, you could have an Application Delivery Controller (ADC) like Azure Application Gateway between your microservices and the Internet. This acts as a transparent tier that not only performs load balancing, but secures your services by offering SSL termination. This improves the load of your hosts by offloading CPU-intensive SSL termination and other routing duties to the Azure Application Gateway. In any case, a load balancer and ADC are transparent from a logical application architecture point of view

Good for small scale application

How can client apps minimize the number of requests to the back end and reduce chatty communication to multiple microservices?

How can you handle cross-cutting concerns such as authorization, data transformations, and dynamic request dispatching?

Implementing security and cross-cutting concerns like security and authorization on every microservice can require significant development effort. A possible approach is to have those services within the Docker host or internal cluster to restrict direct access to them from the outside, and to implement those cross-cutting concerns in a centralized place, like an API Gateway.

How can client apps communicate with services that use non-Internet-friendly protocols?

Protocols used on the server side (like AMQP or binary protocols) are usually not supported in client
apps. Therefore, requests must be performed through protocols like HTTP/HTTPS and translated to
the other protocols afterwards. A man-in-the-middle approach can help in this situation

Consider API Gateways instead of direct client-to-microservice Communication

Friday, June 4, 2021 11:15 AM

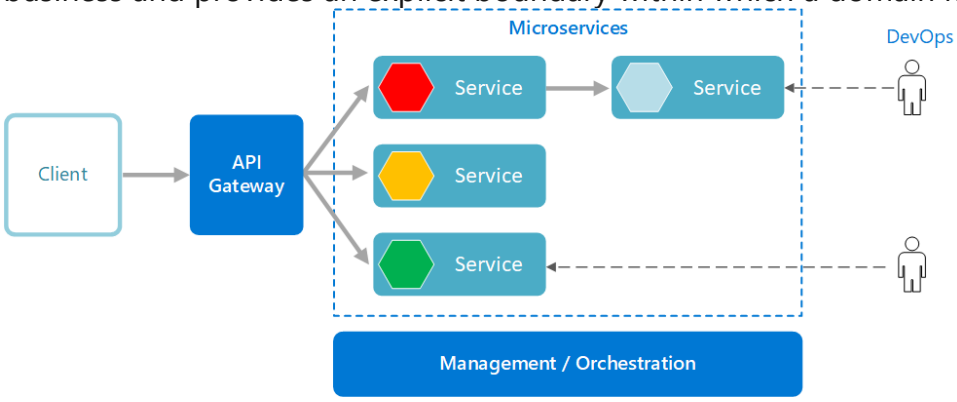
Reverse proxy or gateway routing
Requests aggregation.
Cross-cutting concerns or gateway offloading. Depending on the features offered by each API Gateway product, you can offload functionality from individual microservices to the gateway, which simplifies the implementation of each microservice by consolidating cross-cutting concerns into one tier. This is especially convenient for specialized features that can be complex to implement properly
in every internal microservice, such as the following functionality:

- Authentication and authorization
- Service discovery integration
- Response caching
- Retry policies, circuit breaker, and QoS
- Rate limiting and throttling
- Load balancing
- Logging, tracing, correlation
- Headers, query strings, and claims transformation
- IP whitelisting

Microservices architecture style

Tuesday, July 6, 2021 3:54 PM

A microservices architecture consists of a collection of small, autonomous services. Each service is self-contained and should implement a single business capability within a bounded context. A bounded context is a natural division within a business and provides an explicit boundary within which a domain model exists.



What are microservices?

- Microservices are small, independent, and loosely coupled. A single small team of developers can write and maintain a service.
- Each service is a separate codebase, which can be managed by a small development team.
- Services can be deployed independently. A team can update an existing service without rebuilding and redeploying the entire application.
- Services are responsible for persisting their own data or external state. This differs from the traditional model, where a separate data layer handles data persistence.
- Services communicate with each other by using well-defined APIs. Internal implementation details of each service are hidden from other services.
- Supports polyglot programming. For example, services don't need to share the same technology stack, libraries, or frameworks.

Besides for the services themselves, some other components appear in a typical microservices architecture:

Management/orchestration. This component is responsible for placing services on nodes, identifying failures, rebalancing services across nodes, and so forth. Typically this component is an off-the-shelf technology such as Kubernetes, rather than something custom built.

API Gateway. The API gateway is the entry point for clients. Instead of calling services directly, clients call the API gateway, which forwards the call to the appropriate services on the back end.

Advantages of using an API gateway include:

- It decouples clients from services. Services can be versioned or refactored without needing to update all of the clients.

- Services can use messaging protocols that are not web friendly, such as AMQP.
- The API Gateway can perform other cross-cutting functions such as authentication, logging, SSL termination, and load balancing.
- Out-of-the-box policies, like for throttling, caching, transformation, or validation.

Benefits

- Agility. Because microservices are deployed independently, it's easier to manage bug fixes and feature releases. You can update a service without redeploying the entire application, and roll back an update if something goes wrong. In many traditional applications, if a bug is found in one part of the application, it can block the entire release process. New features may be held up waiting for a bug fix to be integrated, tested, and published.
- Small, focused teams. A microservice should be small enough that a single feature team can build, test, and deploy it. Small team sizes promote greater agility. Large teams tend be less productive, because communication is slower, management overhead goes up, and agility diminishes.
- Small code base. In a monolithic application, there is a tendency over time for code dependencies to become tangled. Adding a new feature requires touching code in a lot of places. By not sharing code or data stores, a microservices architecture minimizes dependencies, and that makes it easier to add new features.
- Mix of technologies. Teams can pick the technology that best fits their service, using a mix of technology stacks as appropriate.
- Fault isolation. If an individual microservice becomes unavailable, it won't disrupt the entire application, as long as any upstream microservices are designed to handle faults correctly (for example, by implementing circuit breaking).
- Scalability. Services can be scaled independently, letting you scale out subsystems that require more resources, without scaling out the entire application. Using an orchestrator such as Kubernetes or Service Fabric, you can pack a higher density of services onto a single host, which allows for more efficient utilization of resources.
- Data isolation. It is much easier to perform schema updates, because only a single microservice is affected. In a monolithic application, schema updates can become very challenging, because different parts of the application may all touch the same data, making any alterations to the schema risky.

Challenges

The benefits of microservices don't come for free. Here are some of the challenges to consider before embarking on a microservices architecture.

- Complexity. A microservices application has more moving parts than the equivalent monolithic application. Each service is simpler, but the entire system as a whole is more complex.
- Development and testing. Writing a small service that relies on other

dependent services requires a different approach than a writing a traditional monolithic or layered application. Existing tools are not always designed to work with service dependencies. Refactoring across service boundaries can be difficult. It is also challenging to test service dependencies, especially when the application is evolving quickly.

- Lack of governance. The decentralized approach to building microservices has advantages, but it can also lead to problems. You may end up with so many different languages and frameworks that the application becomes hard to maintain. It may be useful to put some project-wide standards in place, without overly restricting teams' flexibility. This especially applies to cross-cutting functionality such as logging.
- Network congestion and latency. The use of many small, granular services can result in more interservice communication. Also, if the chain of service dependencies gets too long (service A calls B, which calls C...), the additional latency can become a problem. You will need to design APIs carefully. Avoid overly chatty APIs, think about serialization formats, and look for places to use asynchronous communication patterns like queue-based load leveling.
- Data integrity. With each microservice responsible for its own data persistence. As a result, data consistency can be a challenge. Embrace eventual consistency where possible.
- Management. To be successful with microservices requires a mature DevOps culture. Correlated logging across services can be challenging. Typically, logging must correlate multiple service calls for a single user operation.
- Versioning. Updates to a service must not break services that depend on it. Multiple services could be updated at any given time, so without careful design, you might have problems with backward or forward compatibility.
- Skill set. Microservices are highly distributed systems. Carefully evaluate whether the team has the skills and experience to be successful.

Best practices

- Model services around the business domain.
- Decentralize everything. Individual teams are responsible for designing and building services. Avoid sharing code or data schemas.
- Data storage should be private to the service that owns the data. Use the best storage for each service and data type.
- Services communicate through well-designed APIs. Avoid leaking implementation details. APIs should model the domain, not the internal implementation of the service.
- Avoid coupling between services. Causes of coupling include shared database schemas and rigid communication protocols.
- Offload cross-cutting concerns, such as authentication and SSL termination, to the gateway.
- Keep domain knowledge out of the gateway. The gateway should handle and route client requests without any knowledge of the business rules or domain logic. Otherwise, the gateway becomes a dependency and can cause

coupling between services.

- Services should have loose coupling and high functional cohesion. Functions that are likely to change together should be packaged and deployed together. If they reside in separate services, those services end up being tightly coupled, because a change in one service will require updating the other service. Overly chatty communication between two services may be a symptom of tight coupling and low cohesion.
- Isolate failures. Use resiliency strategies to prevent failures within a service from cascading. See Resiliency patterns and Designing reliable applications.

From <https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices>

API Management

Friday, June 4, 2021 7:17 PM

Azure API Management not only solves your API Gateway needs but provides features like gathering insights from your APIs.

If you're using an API management solution, an API Gateway is only a component within that full API management solution

Gateway is only a component within that full API management solution.
Figure 4-14. Using Azure API Management for your API Gateway
Azure API Management solves both your API Gateway and Management needs like logging, security,
metering, etc. In this case, when using a product like Azure API Management, the fact that you might
have a single API Gateway is not so risky because these kinds of API Gateways are "thinner", meaning
that you don't implement custom C# code that could evolve towards a monolithic component.
The API Gateway products usually act like a reverse proxy for ingress communication, where you can
also filter the APIs from the internal microservices plus apply authorization to the published APIs in this single tier.
The insights available from an API Management system help you get an understanding of how your APIs are being used and how they are performing. They do this by letting you view near real-time analytics reports and identifying trends that might impact your business. Plus, you can have logs about request and response activity for further online and offline analysis.
With Azure API Management, you can secure your APIs using a key, a token, and IP filtering. These features let you enforce flexible and fine-grained quotas and rate limits, modify the shape and behavior of your APIs using policies, and improve performance with response caching.
In this guide and the reference sample application (eShopOnContainers), the architecture is limited to
a simpler and custom-made containerized architecture in order to focus on plain containers without 49 CHAPTER 4 | Architecting container and microservice-based applications
using PaaS products like Azure API Management. But for large microservice-based applications that are deployed into Microsoft Azure, we encourage you to evaluate Azure API Management as the base
for your API Gateways in production