Microservices

The foundation of microservices architecture(MSA) is about developing a single application as a suite of small and independent services that are running in its own process, developed and deployed independently.

These services are built around business capabilities and independently deployable by fully automated deployment machinery.

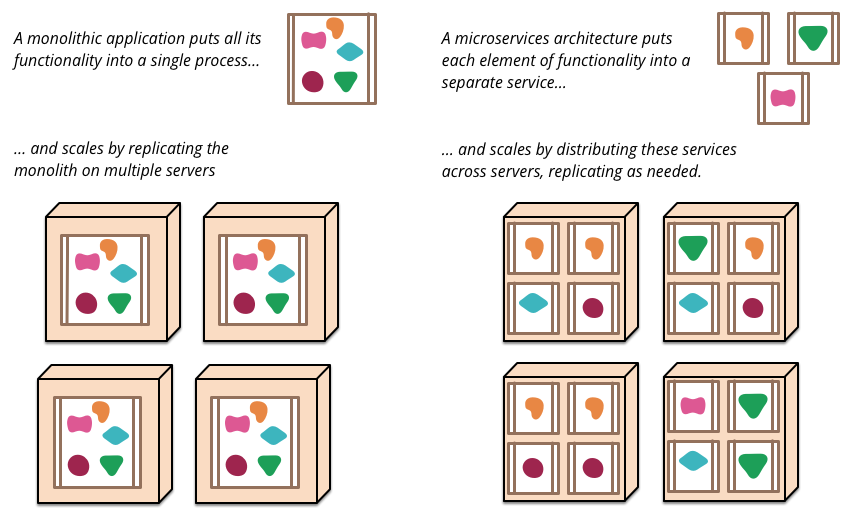
Monolithic architecture :

a monolithic application built as a single unit. Enterprise Applications are often built in three main parts: a client-side user interface (consisting of HTML pages and javascript running in a browser on the user's machine) a database (consisting of many tables inserted into a common, and usually relational, database management system), and a server-side application. The server-side application will handle HTTP requests, execute domain logic, retrieve and update data from the database, and select and populate HTML views to be sent to the browser. This server-side application is a *monolith* - a single logical executable[[2]](https://martinfowler.com/articles/microservices.html" \l "footnote-monolith). Any changes to the system involve building and deploying a new version of the server-side application

You can horizontally scale the monolith by running many instances behind a load-balancer.

Disadvantage : a change made to a small part of the application, requires the entire monolith to be rebuilt and deployed

These frustrations have led to the microservice architectural style: building applications as suites of services

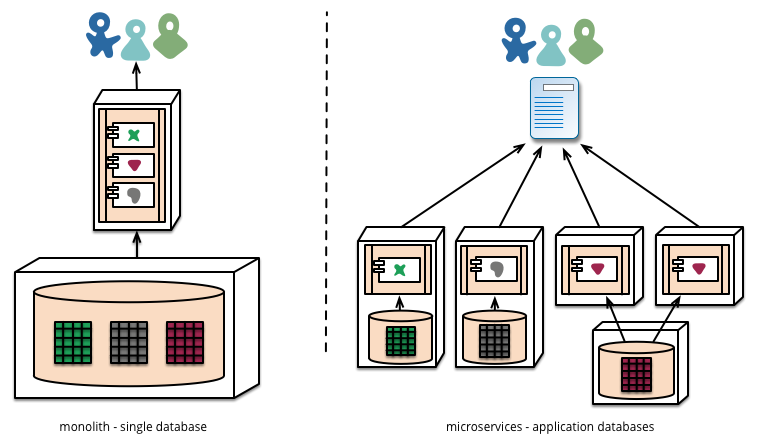
**

It even allows for different services to be written in different programming languages.

if an application is decomposed into multiple services, you can expect many single service changes to only require that service to be redeployed. That's not an absolute, some changes will change service interfaces resulting in some coordination, but the aim of a good microservice architecture is to minimize these through cohesive service boundaries and evolution mechanisms in the service contracts.

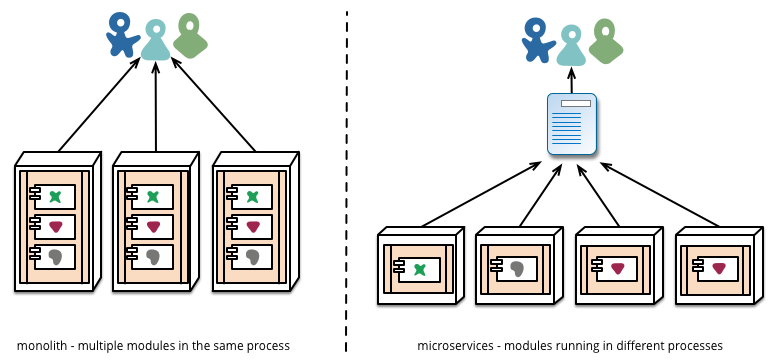
Applications built from microservices aim to be as decoupled and as cohesive as possible - they own their own domain logic and act more as filters in the classical Unix sense - receiving a request, applying logic as appropriate and producing a response.

While monolithic applications prefer a single logical database for persistant data, enterprises often prefer a single database across a range of applications - many of these decisions driven through vendor's commercial models around licensing. Microservices prefer letting each service manage its own database, either different instances of the same database technology, or entirely different database systems - an approach called [Polyglot Persistence](https://martinfowler.com/bliki/PolyglotPersistence.html).

**

Promotion of working software 'up' the pipeline means we **automate deployment** to each new environment

A monolithic application will be built, tested and pushed through these environments quite happlily. It turns out that once you have invested in automating the path to production for a monolith, then deploying *more* applications doesn't seem so scary any more. Remember, one of the aims of CD is to make deployment boring, so whether its one or three applications, as long as its still boring it doesn't matter

**

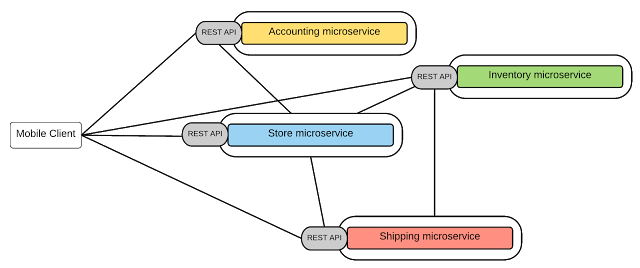
Disadvantage : Any service call could fail due to unavailability of the supplier, the client has to respond to this as gracefully as possible. This is a disadvantage compared to a monolithic design as it introduces additional complexity to handle it.

The traditional monolithic applications use complex binary formats, SOA/Web services-based applications use text messages based on the complex message formats (SOAP) and schemas (xsd). In most microservices-based applications, they use simple text-based message formats such as JSON and XML on top of HTTP resource API style.

Since we build microservices on top of REST architectural style, we can use the same REST API definition techniques to define the contract of the microservices. Therefore, microservices use the standard REST API definition languages such as [Swagger](http://swagger.io/) and [RAML](http://raml.org/) to define the service contracts.

**Point-to-point Style - Invoking Services Directly**

In point to point style, the entirety of the message routing logic resides on each endpoint and the services can communicate directly. Each microservice exposes a REST APIs and a given microservice or an external client can invoke another microservice through its REST API.



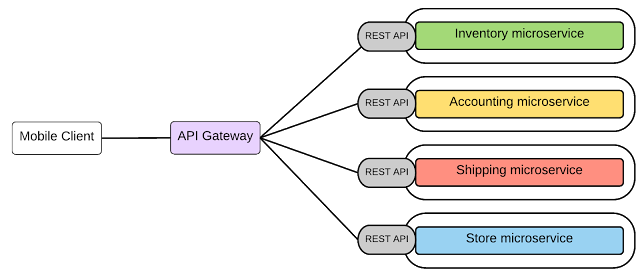
Obviously, this model works for relatively simple microservices-based applications but as the number of services increases, this will become overwhelmingly complex. After all that's the exact same reason for using ESB in the traditional SOA implementation, which is to get rid of the messy point-to-point integration links. Let's try to summarize the key drawbacks of the point-to-point style for microservice communication.

* The non-functional requirements such as end-user authentication, throttling, monitoring, etc. has to be implemented at each and every microservice level.
* As a result of duplicating common functionalities, each microservice implementation can become complex.
* There is no control at all of the communication between the services and clients (even for monitoring, tracing, or filtering)
* Often the direct communication style is considered as a microservice [anti-pattern](http://www.infoq.com/articles/seven-uservices-antipatterns) for large scale microservice implementations.

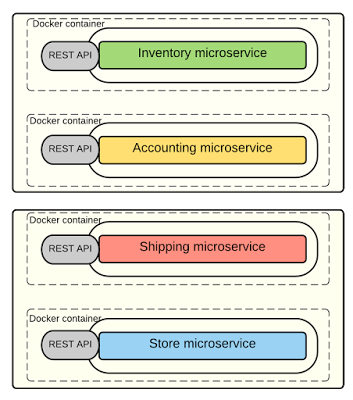
Therefore, for complex Microservices use cases, rather than having point-to-point connectivity or a central ESB, we could have a lightweight central messaging bus which can provide an abstraction layer for the microservices and that can be used to implement various non-functional capabilities. This style is known as API Gateway style.

#### API-Gateway Style

The key idea behind the API Gateway style is that using a lightweight message gateway as the main entry point for all the clients/consumers and implement the common non-functional requirements at the Gateway level. In general, an API Gateway allows you to consume a managed API over REST/HTTP. Therefore, here we can expose our business functionalities which are implemented as microservices, through the API-GW, as managed APIs. In fact, this is a combination of Microservices architecture and API-Management which give you the best of both worlds.

[](http://2.bp.blogspot.com/-mUR7PGsj0eo/Vm7Qd5WvZEI/AAAAAAAADGo/Kc19jdnZbN0/s1600/msa_api_gw.png)

In our retail business scenario, as depicted in figure 5, all the microservices are exposed through an API-GW and that is the single entry point for all the clients. If a microservice wants to consume another microservice that also needs to be done through the API-GW.



In above figure , it shows an overview of the deployment of the microservices of the retail application. Each microservice instance is deployed as a container and there are two containers per each host. You can arbitrarily change the number of containers that you run on a given host.

[Docker](https://www.docker.com/) (an open source engine that lets developers and system administrators deploy self-sufficient application containers in Linux environments) provides a great way to deploy microservices addressing the above requirements. The key steps involved are as follows:

* Package the microservice as a (Docker) container image.
* Deploy each service instance as a container.
* Scaling is done based on changing the number of container instances.
* Building, deploying, and starting microservice will be much faster as we are using Docker containers (which is much faster than  a regular VM)

[Kubernetes](http://kubernetes.io/) is extending Docker's capabilities by allowing to manage a cluster of Linux containers as a single system, managing and running Docker containers across multiple hosts, offering co-location of containers, service discovery, and replication control.