MICRO-SERVICE ARCHITECTURE

BASICS

WHAT IS ARCHITECTURE (SOFTWARE)?

Architecture is the fundamental organization of a system embodied in its components (i.e. Web Server, Application Server, Databases,Storage, Communication layer, etc…), their relationships to each other, and to the environment (i.e. deployment environment shared server, dedicated server, cloud deployment, etc..), and the principles guiding its design and evolution.

WHAT IS MICROSERVICE ARCHITECTURE ?

Microservice means developing a single, small, meaningful functional feature as single service, each service has it’s own process and communicate with lightweight mechanism, deployed in single or multiple servers.

ADVANTAGES OF MICROSERVICE ARCHITECTURE ?

- Each micro service is small and focused on a specific feature / business requirement.
- Microservice can be developed independently by small team of developers (normally 2 to 12 developers).
- Microservice is loosely coupled, means services are independent, in terms of development and deployment both.
- Microservice can be developed using different programming language (Personally I don't suggest to do it).
- Microservice allows easy and flexible way to integrate automatic deployment with Continuous Integration tools (for e.g: Jenkins, Hudson, bamboo etc..).
- The productivity of a new team member will be quick enough.
- Microservice is easy to understand, modify and maintain for a developer because separation of code,small team and focused work.
- Microservice allows you to take advantage of emerging and latest technologies (framework, programming language , programming practice, etc.).
- Microservice has code for business logic only, No mixup with HTML,CSS or other UI component.
- Microservice is easy to scale based on demand.
- Microservice can deploy on commodity hardware or low / medium configuration servers.
- Easy to integrate 3rd party service.
- Every microservice has it's own storage capability but it depends on the project's requirement, you can have common database like MySQL or Oracle for all services.

DISADVANTAGES OF MICROSERVICE ARCHITECTURE ?

- Microservice architecture brings a lot of operations overhead.
- DevOps Skill required (http://en.wikipedia.org/wiki/DevOps).
- Duplication of Effort.
- Distributed System is complicated to manage .
- Default to trace problem because of distributed deployment.
- Complicated to manage whole products when number of services increases.

IN WHICH CASE / REQUIREMENT MICROSERVICE ARCHITECTURE BEST FIT ?

When you need to support desktop, web, mobile, smart appliances, wearable, etc… or you don't know in future which kind of devices you need to support.

WHICH PRODUCTS / COMPANIES ARE USING MICROSERVIE ARCHITECTURE?

Most large scale web based / cloud services providers including Microsoft, Amazon, Twitter, eBay etc. have evolved from monolithic architecture to microservice architecture.

HOW INDEPENDENT MICRO SERVICES COMMUNICATE WITH EACH OTHER?

It depends upon requirement.

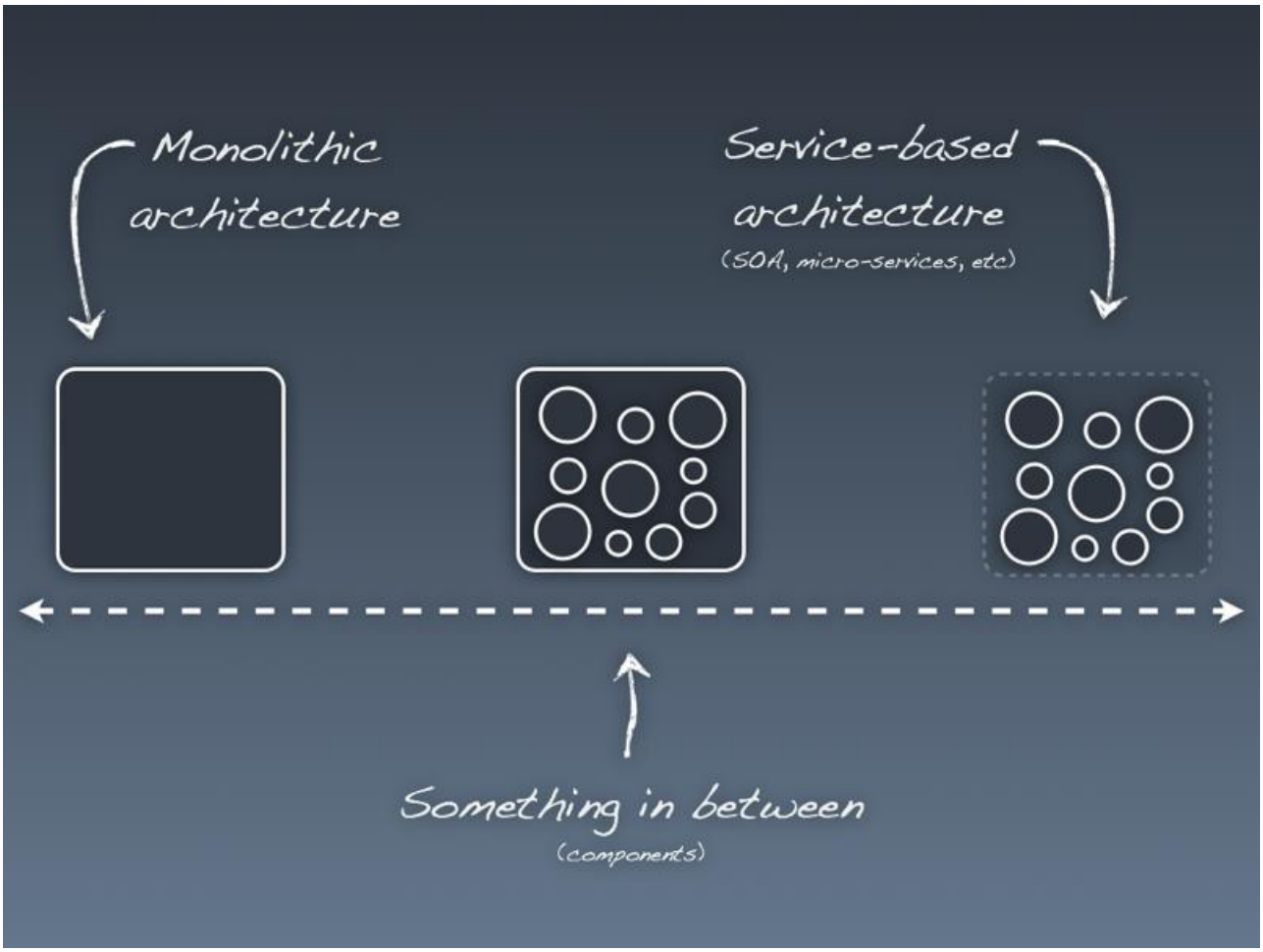
Normally developers use HTTP/REST with JSON or Protobuf (Binary protocol) but are free to use any communication protocol.

WHY EVERYONE TALKS ABOUT MICROSERVICES NOW?

It's been nearly 15 years since the concept of Service Oriented Architecture really took hold.

Improvement of RESTful web service and JSON as a data interchange format has made it easier to build easily interconnectable services simply and quickly.

COMPARISON: MONOLOTHIC – SOA – MSA



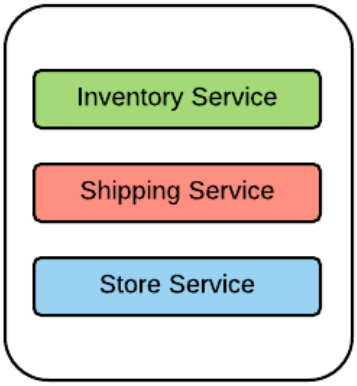
MONOLITHIC ARCHITECTURE

Enterprise software applications are designed to facilitate numerous business requirements.

Hence, a given software application offers hundreds to functionalities and all such functionalities are piled into a single monolithic application.
For examples, ERPs, CRMs, and other various software systems are built as a monolith with several hundreds of functionalities.

The deployment, troubleshooting, scaling, and upgrading of such monstrous software applications is a nightmare.

Figure shows a retail software application which comprises of multiple services.
All these services are deployed into the same application runtime.



CHARACTERISTICS

Here are some of the characteristics of monolithic architecture based applications.
These characteristics of Monolithic Architecture have led to the Microservice Architecture.

- Monolithic applications are designed, developed, and deployed as a single unit.
- Monolithic applications are overwhelmingly complex; which leads to nightmares in maintaining, upgrading, and adding new features.
- Hard to practice agile development and delivery methodologies with Monolithic architecture.
- It is required to redeploy the entire application, in order to update a part of it.
- Scaling: Has to be scaled as a single application and difficult to scale with conflicting resource requirements (e.g. one service requires more CPU while the other requires more memory)
- Reliability: One unstable service can bring the whole application down.
- Hard to innovate: It's really difficult to adopt new technologies and frameworks as of all the functionalities have to build on homogeneous technologies/frameworks.

SOA

Service Oriented Architecture (SOA) was designed to overcome some of the aforementioned limitations by introducing the concept of a 'service'
which is an aggregation and grouping of similar functionalities offered from an application.
Hence, with SOA, a software application is designed as a combination of 'coarse-grained' services.

However, in SOA, the scope of a service is very broad.
That leads to complex and mammoth services with several dozens of operations (functionalities) along with complex message formats and standards (e.g: all WS* standards).

ARE MICROSERVICES REALLY JUST SOA?

If SOA is being considered as ESB (Enterprise Service Bus with smart centralized management), in that case microservices is very different.

Fowler defines it as a subset of SOA, one that refers to a particular style of implementing certain concepts.

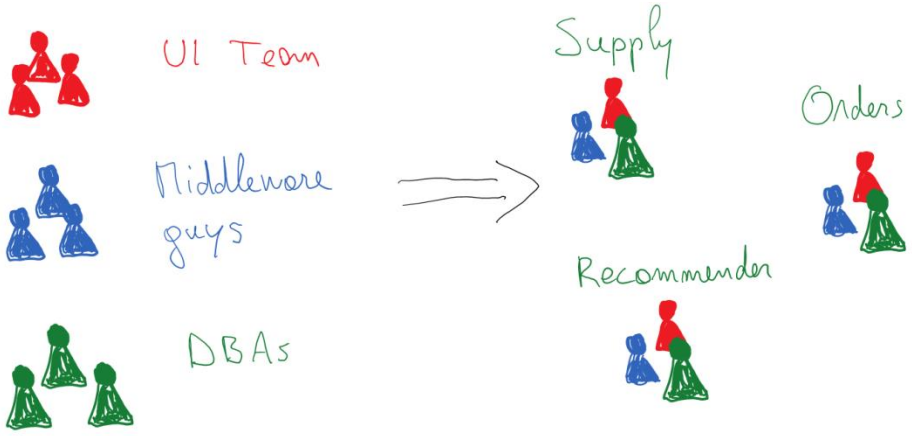
Having "micro" in its name, the question arises about how big a service should be? Actually there's a lot of variation, from 15 people -> 10 services to 4 people -> 200 services. It's not yet exactly clear

Basically a component is something that is independently upgradeable and replaceable.

According to Amazon, these teams should be as big as we're able to feed them with 2 (American) pizzas (a dozen of people).

Fowler highlights that an important fact is that these teams have a direct communication line to the end user or customer and get according feedback how the stuff they build is being used and how well or not it works.

Microservices are much more about team organization rather than software architecture.
Architecture and team organization is always heavily coupled together.



MICROSERVICE ARCHITECTURE

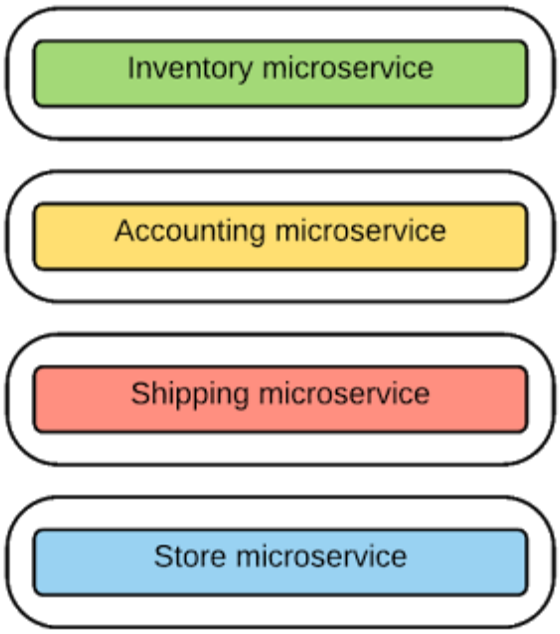
The foundation of microservice architecture (MSA) is about developing a single application as a suite of small and independent services that are running in its own process (different runtime), developed and deployed independently.

In most of the definitions of microservices architecture, it is explained as the process of segregating the services available in the monolith into a set of independent services.
However, **Microservices is not just about splitting the services available in monolith into independent services.**

The key idea is that by looking at the functionalities offered from the monolith, we can identify the required business capabilities. Then those business capabilities can be implemented as fully independent, fine-grained, and self-contained (micro) services. They might be implemented on top of different technology stacks and each service is addressing a very specific and limited business scope.

Therefore, the online retail system scenario that we explain above can be realized with microservices architecture as depicted in below figure. With the microservice architecture, the retail software application is implemented as a suite of microservices.

So, as you can see in this figure, based on the business requirements, there is an additional microservice created from the original set of services that are there in the monolith. So, it is quite obvious that using microservices architecture is something beyond the splitting of the services in the monolith.



DESIGNING MICROSERVICES

You may be building your software application from scratch by using Microservices Architecture or you are converting existing applications/services into microservices. Either way, it is quite important that you properly decide the size, scope, and the capabilities of the Microservices. Probably, that is the hardest thing that you initially encounter when you implement Microservices Architecture in practice.

SOME MISCONCEPTIONS

- Lines of Code/Team size are lousy metrics: There are several discussions on deciding the size of the Microservices based on the lines-of-code of its implementation or its team's size (i.e. two-pizza team). However, these are considered to be very impractical and lousy metrics, because we can still develop services with less code/with two-pizza-team size but totally violating the microservice architectural principals.
- 'Micro' is a bit misleading term: Most developers tend to think that they should try to make the service, as small as possible. This is a misinterpretation.
- In the SOA context, services are often implemented as monolithic globs with the support for several dozens of operations/functionalities. So, having SOA-like services and rebranding them as microservices is not going to give you any benefits of microservices architecture.

GUIDELINES FOR DESIGNING MICROSERVICES

- Single Responsibility Principle (SRP): Having a limited and a focused business scope for a microservice helps us to **meet the agility** in development and delivery of services.
- During the designing phase of the microservices, we should find their boundaries and align them with the business capabilities (also known as bounded context in Domain-Driven-Design).
- Make sure the microservices design ensures the agile/independent development and deployment of the service.
- Our focus should be on the scope of the microservice, but not about making the service smaller. The (right) size of the service should be the required size to facilitate a given business capability.
- Unlike service in SOA, a given microservice should have a very few operations/functionalities and simple message format.
- It is often a good practice to start with relatively broad service boundaries to begin with, refactoring to smaller ones (based on business requirements) as time goes on.

In our retail use case, you can find that we have split the functionalities of its monolith into four different microservices, namely 'inventory', 'accounting', 'shipping', and 'store'. They are addressing a limited but focused business scope so that each service is fully decoupled from each other and ensures the agility in development and deployment.

MESSAGING IN MICROSERVICES

In monolithic applications, business functionalities of different processors/components are invoked using function calls or language-level method calls.

In SOA, this was shifted towards a much more loosely coupled web service level messaging, which is primarily based on SOAP on top of different protocols such as HTTP, JMS.
Webservices with several dozens of operations and complex message schemas was a key resistive force for the popularity of web services.

For Microservices architecture, it is required to have a simple and lightweight messaging mechanism.

SYNCHRONOUS MESSAGING - REST, THRIFT

For synchronous messaging (client expects a timely response from the service and waits till it get it) in Microservices Architecture, REST is the unanimous choice as it provides a simple messaging style implemented with HTTP request-response, based on resource API style.

Therefore, most microservices implementations are using HTTP along with resource API based styles (every functionality is represented with a resource and operations carried out on top of those resources).

Thrift is used (in which you can define an interface definition for your microservice), as an alternative to REST/HTTP synchronous messaging.

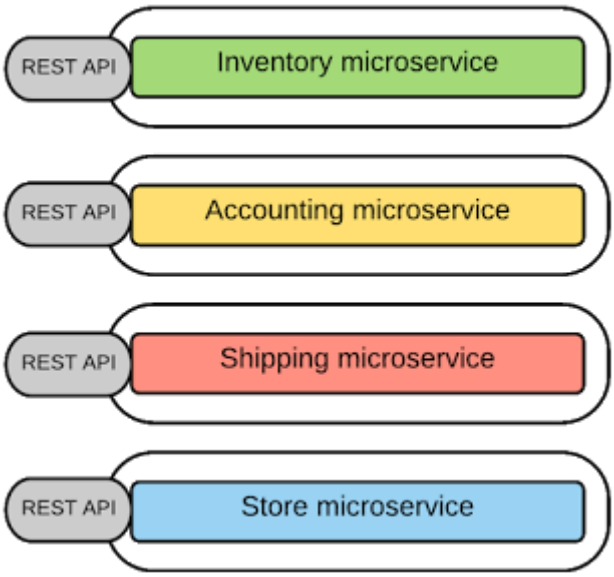


Figure: Using REST interfaces to expose microservices.

ASYNCHRONOUS MESSAGING - AMQP, STOMP, MQTT

For some microservices scenarios, it is required to use asynchronous messaging techniques(client doesn't expect a response immediately, or does not accept a response at all). In such scenarios, asynchronous messaging protocols such as AMQP, STOMP, or MQTT are widely used.

MESSAGE FORMATS - JSON, XML, THRIFT, PROTOBUF, AVRO

Deciding the best-suited message format for microservices is another key factor.

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, we use simple text-based message formats such as JSON and XML on top of HTTP resource API style.

In cases where we need binary message formats (text messages can become verbose in some use cases), microservices can leverage binary message formats such as binary Thrift, ProtoBuf, or Avro.

SERVICE CONTRACTS

Service contracts means defining the service interfaces

When you have a business capability implemented as a service, you need to define and publish the service contract. In traditional monolithic applications, we barely find such feature to define the business capabilities of an application. In SOA/Web services world, WSDL is used to define the service contract, but, as we all know, WSDL is not the ideal solution for defining microservices contract as WSDL is insanely complex and tightly coupled to SOAP.

Since we build microservices on top of REST architectural style, we 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 and RAML to define the service contracts.

For other microservices implementation which are not based on HTTP/REST (such as Thrift), we can use the protocol level 'Interface Definition Languages(IDL)' (e.g.: Thrift IDL).

INTEGRATING MICROSERVICES

Integrating microservices means inter-service communication

In Microservices architecture, the software applications are built as a suite of independent services. So, in order to realize a business use case, it is required to have the communication structures between different microservices/processes. That's why inter-service/process communication between microservices is a such a vital aspect.

In SOA implementations, the inter-service communication between services is facilitated with an Enterprise Service Bus (ESB) and most of the business logic resides in the intermediate layer (message routing, transformation, and orchestration). However, Microservices architecture promotes to eliminate the central message bus/ESB and move the 'smart-ness' or business logic to the services and client (known as 'Smart Endpoints').

Since microservices use standard protocols such as HTTP, JSON, etc. the requirement of integrating with a disparate protocol is minimal when it comes to the communication among microservices. Another alternative approach in Microservice communication is to use a lightweight message bus or gateway with minimal routing capabilities and just acting as a 'dumb pipe' with no business logic implemented on gateway.

Based on these styles there are several communication patterns that have emerged in microservices architecture.

- Point-to-point Style - Invoking Services Directly
- API-Gateway Style

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.

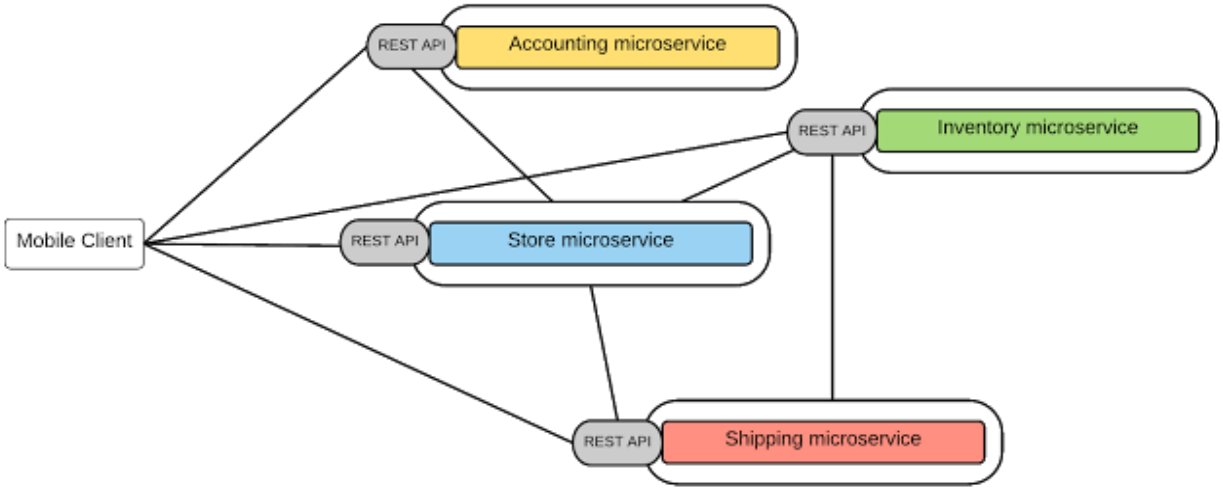


Figure: Inter-service communication with point-to-point connectivity.

Obviously, this model works for relatively simple microservices-based applications but as the number of services increases, this model becomes 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.

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 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.

In our retail business scenario, as depicted in this figure, 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.

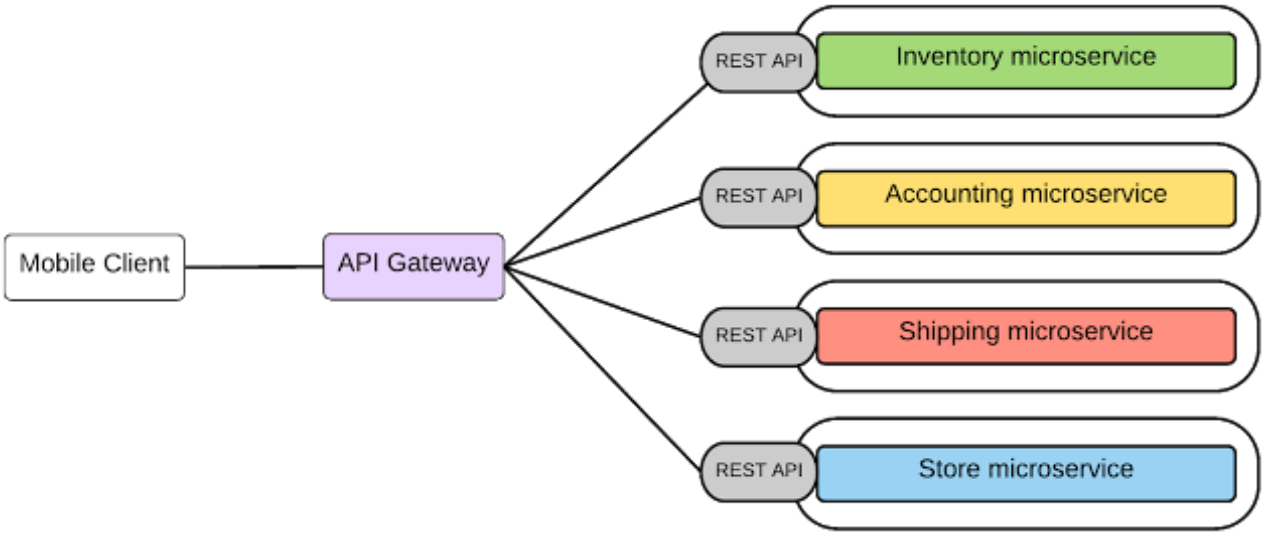


Figure: All microservices are exposed through an API-GW.

ADVANTAGES OF API-GW STYLE

Ability to provide the required abstractions at the gateway level for the existing microservices. For example, rather than provide a one-size-fits-all style API, the API gateway can expose a different API for each client.

- Lightweight message routing/transformations at gateway level.
- Central place to apply non-functional capabilities such as security, monitoring and throttling
- With the use of API-GW pattern, the microservice becomes even more lightweight as all the non-functional requirements are implemented at the Gateway level.

The API-GW style becomes the most widely used pattern in most microservice implementations.

DATA MANAGEMENT

CENTRALIZED DATA MANAGEMENT

In monolithic architecture, the application stores data in a single and centralized databases to implement various functionalities/capabilities of the application.

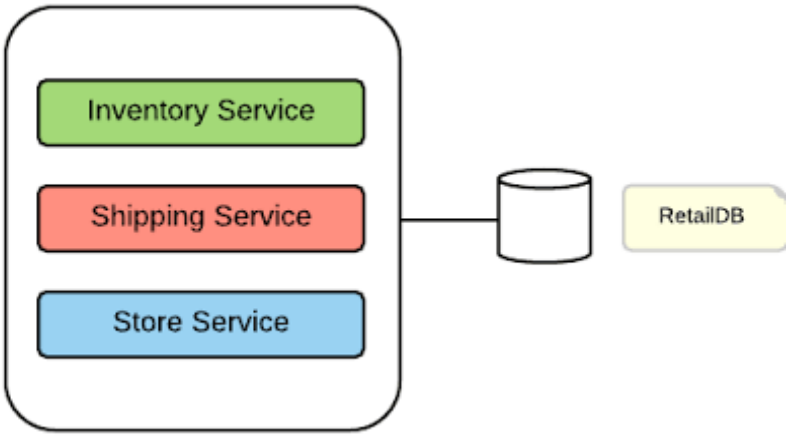


Figure: Monolithic application uses a centralized database to implement all its features.

DECENTRALIZED DATA MANAGEMENT

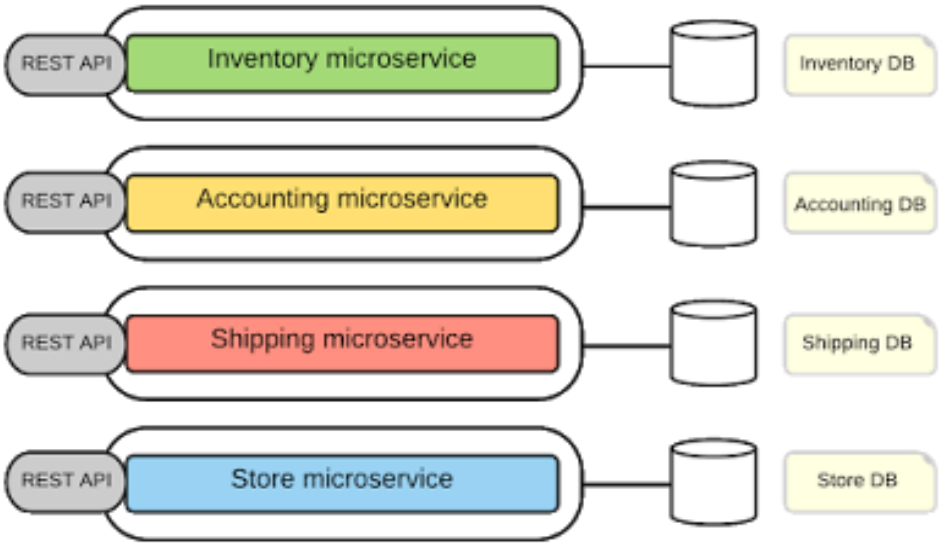


Figure: Microservices has its own private database and they can't directly access the database owned by other microservices.

In Microservices architecture, the functionalities are dispersed across multiple microservices and, if we use the same centralized database, then the microservices will no longer be independent from each other (for instance, if the database schema has changed from a given microservice, that will break several other services). Therefore, each microservice has to have its own database.

Here are the key aspects of implementing decentralized data management in microservices architecture.

- Each microservice can have a private database to persist the data that requires to implement the business functionality offered from it.
- A given microservice can only access the dedicated private database but not the databases of other microservices.
- In some business scenarios, you might have to update several database for a single transaction. In such scenarios, the databases of other microservices should be updated through its service API only (not allowed to access the database directly)

The de-centralized data management gives you the fully decoupled microservices and the liberty of choosing disparate data management techniques (SQL or NoSQL etc., different database management systems for each service). However, for complex transactional use cases that involve multiple microservices, the transactional behavior has to be implemented using the APIs offered from each service and the logic resides either at the client or intermediary (GW) level.