

[Bhesh Raj Neupane –Design Principle and Microservice Architecture](#)

[Reference](#)

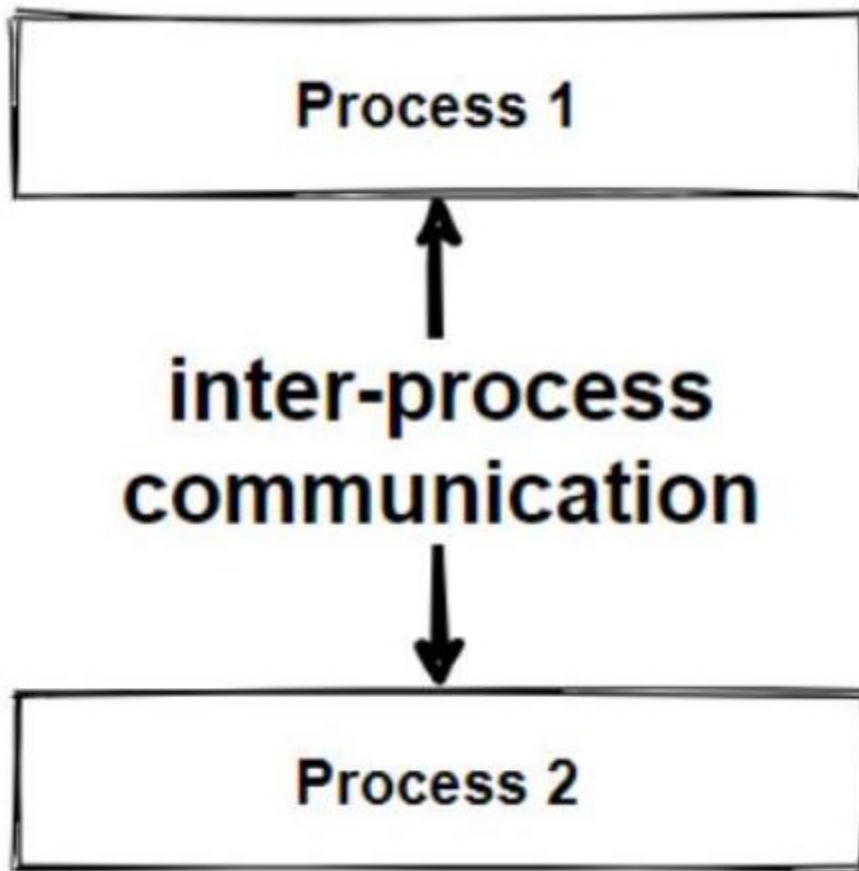
[Mehmet Ozkaya](#)

<https://medium.com/design-microservices-architecture-with-patterns>

Monolithic Architecture <https://medium.com/design-microservices-architecture-with-patterns/communications-in-monolithic-architecture-64f9fa901ba4>

When it comes to **communication in a monolithic architecture**, the application components communicate with each other through **direct method calls** or **function invocations**, often using shared libraries and common databases

- **The communication will be inter-process communication.**



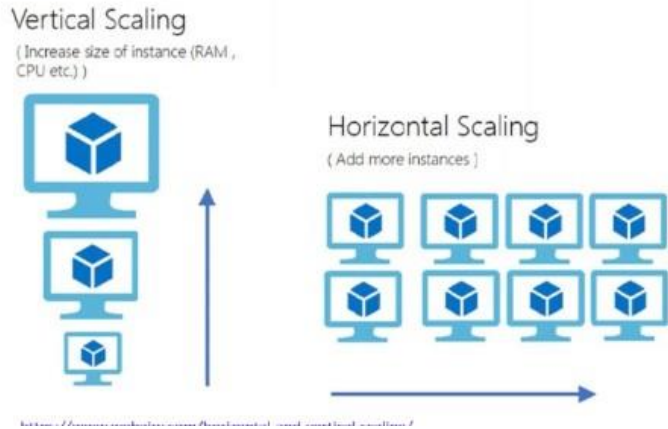
In a monolithic architecture, all of the application's modules are deployed together on a single server or cluster, which means that the **communication** between modules happens within the **same process**, and there is **no need for network communication**.

Scalability

<https://medium.com/design-microservices-architecture-with-patterns/scalability-vertical-scaling-horizontal-scaling-adb52ff679f>

Scalability - Vertical Scaling - Horizontal Scaling

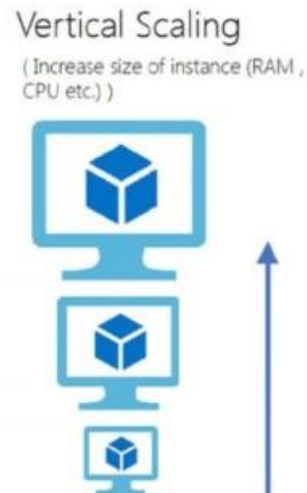
- Vertical Scaling = scaling up
- Horizontal scaling = scaling out
- The number of requests an application can handle
- To prevent downtime, and reduce latency, you must scale
- Horizontal scaling by adding more machines
- Vertical scaling by adding more power



Vertical scaling is basically makes the **nodes stronger**. If you have 1 server, make the server stronger with **adding more hardware**. Make optimization the hardware that will allow you to handle more requests.

Vertical Scaling - Scale up

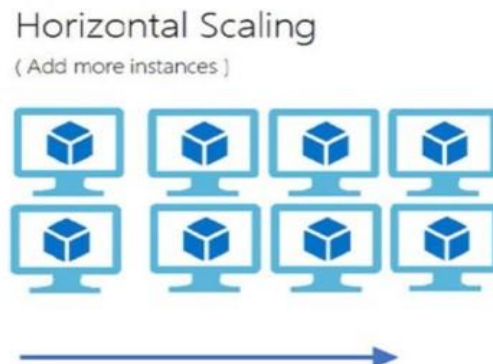
- Makes the nodes stronger
- Adding more computing power
- Same code on machines with better specs
- Adding additional CPU, RAM, and DISK
- Increase power since to **hardware limitations** when you reach the maximum capacity



Horizontal scaling basically means splitting the load between different servers. Horizontal scaling simply **adds more instances** of machines without changing to existing specifications. By scaling out, you can share the processing power and load balancing across multiple machines. **(Best practices- Scale Out , handles millions of requests)**

Horizontal Scaling - Scale out

- Splitting the load between different servers
- Adds more instances of machines
- Share the processing power
- Gives you scalability but also reliability
- **Stateful or Stateless**
- **CAP Theorem**



Load Balancer

load balancers that **balance the traffic** across to all nodes of our applications. Mostly Load balancer is a software application that helps to spread the traffic across a cluster of servers to **improve responsiveness** and availability of the architecture.

Load Balancer sits between the client and the server.

Load Balancer is **accepting incoming network** and application traffic and **distributing the traffic** across multiple backend servers using different algorithms

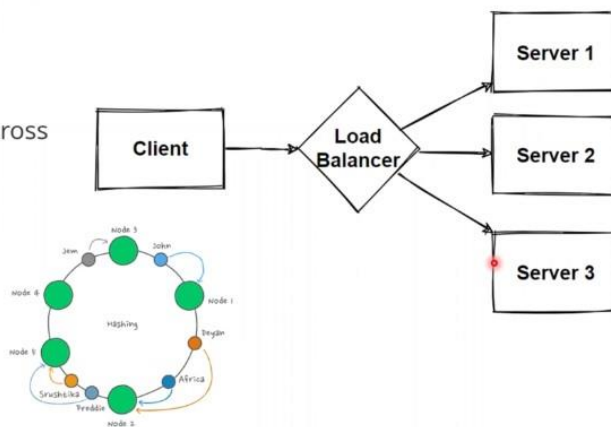
Mostly using consistent hashing algorithms.

NGINX is one of the popular open-source load balancing software that widely using in the software industry.

Main features of Load Balancers should be fault tolerance and improves availability. That means if one of the **backend server** is down, all the traffic will be routed to the rest of the services accordingly from **Load Balancer**. Also if the traffic is growing rapidly, you only need to add more servers and the load balancer will route the traffic for

Load Balancer

- Balance the traffic
- Spread the traffic across a cluster
- Consistent hashing algorithms



you.

Load balancers are using different kind of **distribution algorithms** to optimally distribute the loads. For example, **Round robin** algorithms — works as a **First In First Out (FIFO)**. each server get requests in sequential order.

Why We are using Load Balancer with Consistent Hashing

Consistent hashing is an algorithms for dividing up data between multiple machines. It works particularly well when the number of machines storing data may change. This makes it a useful trick for system design questions involving large, **distributed databases**, which have many machines and must account for machine failure.

Consistent hashing solves the horizontal scalability problem by ensuring that every time we scale up or down, we **don't have to rearrange** all the keys or touch all the database servers. That's why **Consistent hashing** is the best option when working with distributed microservices. In order to increase concurrent request we should **evolve** our **architecture** to **Microservices Architecture**.

N Layered Architecture

N-layered architecture is a design pattern used in software development where the system is organized into multiple layers **to achieve separation of concerns, modularity, and maintainability. Each layer has a specific responsibility, and communication between layers is well-defined.** The number of layers can vary, and "N" is used as a placeholder to represent any number of layers. Commonly, you might encounter architectures with three layers (3-tier) or four layers (4-tier), but more complex systems can have additional layers.

Separation of Concern

Separation of Concerns (SoC) is a fundamental design principle in software development that advocates breaking a computer program into distinct sections, each addressing a different concern or responsibility.

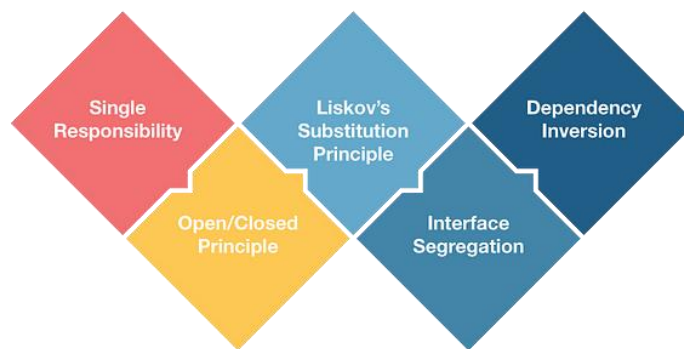
The goal is to isolate different aspects of the software to make it more modular, maintainable, and understandable.

Each concern or module should focus on a specific aspect of the functionality and should be relatively independent of other concerns.

SOLID

<https://medium.com/bgl-tech/what-are-the-solid-design-principles-c61feff33685>

S.O.L.I.D.



The aim of the SOLID principles is to reduce dependencies, enabling the ability to change code in one area of an application without impacting code in other areas.

Single Responsibility

‘There should never be more than one reason for a class to change’

this means that each module or service should have a single responsibility or purpose.

Open/Closed Principle

‘A module should be open for extension but closed for modification’

- Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

- In NestJS, this can be achieved **through the use of dependency injection**, decorators, and the extensibility of modules.

Liskov's Substitution Principle

‘Subclasses should be substitutable for their base classes’

- Subtypes must be substitutable for their base types without altering the correctness of the program.
- In NestJS, this means that derived classes or services should be able to replace the base classes or services without affecting the behavior of the application.

Interface Segregation

‘Many client specific interfaces are better than one general purpose interface’

- A class should not be forced to implement interfaces it does not use.
- In NestJS, this can be achieved by creating small, specific interfaces that are tailored to the needs of the classes that implement them.

Dependency Inversion

‘Depend upon abstractions. Do not depend upon concretions.’

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions

Service Oriented Architecture (SOA)

<https://medium.com/design-microservices-architecture-with-patterns/service-oriented-architecture-1e4716fbca17>

https://docs.oracle.com/cd/E13171_01/alsb/docs30/concepts/introduction.html

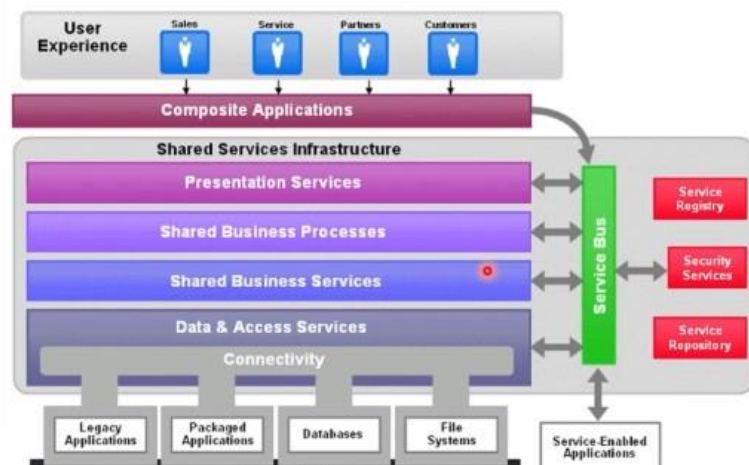
Service-Oriented Architecture (SOA) is a design approach for building software systems that promotes the use of loosely coupled, interoperable, and reusable components or services.

In an SOA, software functionality is organized into services, which are self-contained, modular units of business logic.

These services can be developed, deployed, and scaled independently, and they communicate with each other over a network to achieve specific business goals.

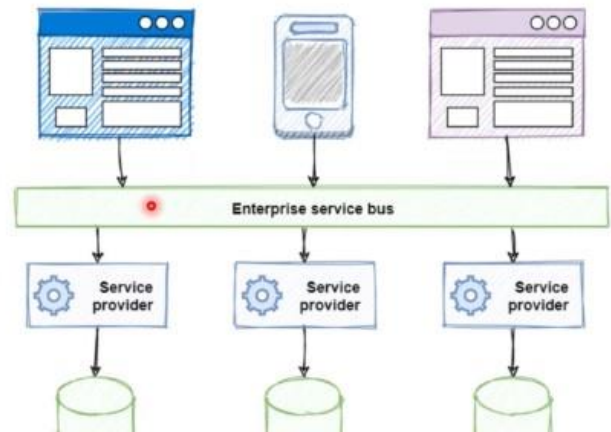
Service-Oriented Architecture

- Service components
- Communicates over the network
- Converged service Infrastructure
- Enterprise applications



Architectural Design patterns - Enterprise Service Bus (ESB)

- Integrations between applications
- Transformations of data models
- Middleware messaging components
- Service orchestration
- Increased complexity and introduced bottlenecks



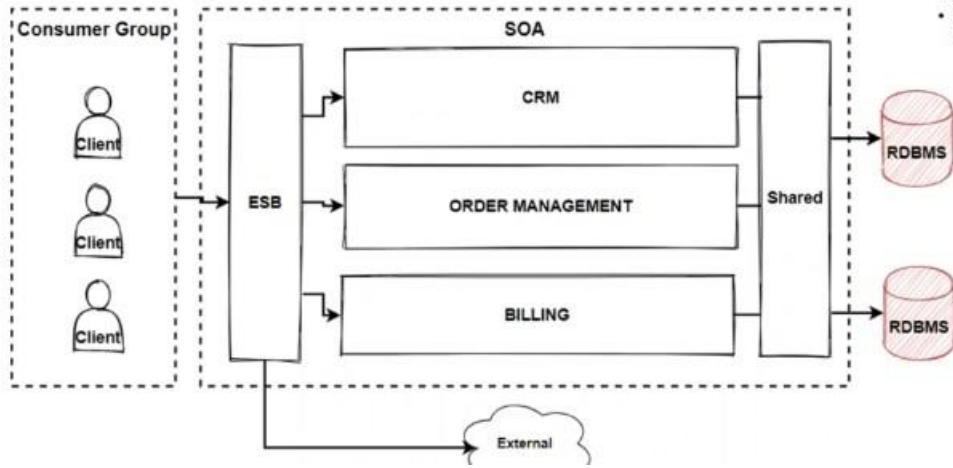
Communication in SOA

We already saw that in **SOA** communications drive by **Enterprise Service Bus — ESB** systems. They performs integrations between applications, handles **connectivity** and **messaging**, performs **routing**, **converts communication protocols** and potentially manages the composition of multiple requests.

So what is the technology that communication handle for **each components** in **SOA**?

First of all, we should say that communication will be the **inter-service communication** because **SOA components** are **distributed**. And these inter-service communication handle by **SOAP-based web services**.

Service-Oriented Architecture



Key Features

- SOAP WS Integrations
- Big Monolithic Applications with using Shared Services

Comparing SOA and Microservices

- Inter-service communication due to distributed services
- SOA, using Enterprise Service Bus
- Microservices, using message brokers
- SOA using Global data model
- Microservices has polyglot databases
- Size of the services different
- SOA increased complexity and introduced bottlenecks
- ESB middleware expensive

Both are using **Inter-service communication** due to **distributed services**. But in SOA, using **Enterprise Service Bus**, using **heavyweight protocols**, such as **SOAP**, **WSDL**, and **XSD protocols**.

But in microservices, using message brokers, or **direct service-to-service** communication, using lightweight protocols such as **REST** or **gRPC**. Also Data models are different, **SOA** using **Global data model** and **shared databases**, But Microservices has polyglot databases with database per service pattern.

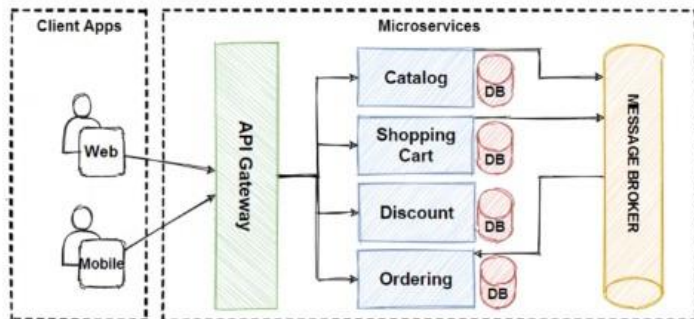
Another **key difference** between SOA and the microservice architecture is the size of the services. SOA is typically used to **integrate large, complex, monolithic applications**. Although services in a microservice architecture almost always **much smaller**.

Microservices Architecture

<https://medium.com/design-microservices-architecture-with-patterns/microservices-architecture-for-enterprise-large-scaled-application-825436c9a78a>

Challenges of Microservices Architecture

- Complexity
- Network problems and Latency
- Development and testing
- Data integrity



Decomposition Microservice Architecture

<https://medium.com/design-microservices-architecture-with-patterns/decomposition-of-microservices-architecture-c8e8cec453e>

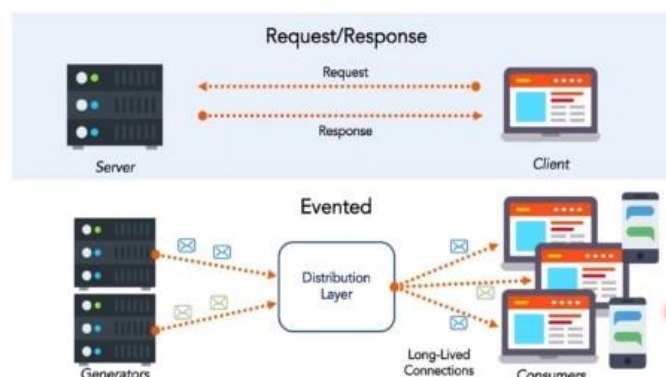
Decompose Microservices with apply various **Microservices Decomposition Patterns** like **Decompose by Business Capability**, **Decompose by Subdomain** and **Bounded Context Pattern** .

Microservices Communications

<https://medium.com/design-microservices-architecture-with-patterns/microservices-communications-f319f8d76b71>

Microservices Communications

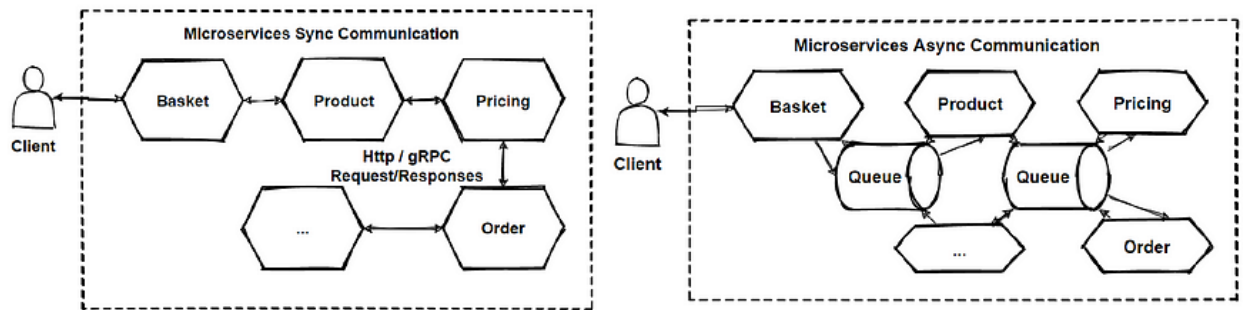
- Monolithic = Inter-Process
- Microservices = Inter-Service communication
- Distributed network calls
- HTTP, gRPC or message brokers
- AMQP protocol
- Synchronous or Asynchronous



One of the biggest challenge when moving to **microservices-based application** is changing the communication mechanism. Because microservices are **distributed** and **microservices communicate** with each other by **inter-service communication** on network level.

Each microservice has its own **instance** and **process**. Therefore, services must interact using an **inter-service communication** protocols like **HTTP**, **gRPC** or **message brokers AMQP** protocol.

Microservices Communication Types — Sync or Async Communication



Synchronous communication

- Synchronous communication is using **HTTP,HTTPS** or **gRPC protocol** for returning sync response.
- The client sends a request and waits for a response from the service. So that means client code block their thread, until the response reach from the server.
- So that means the client call the server and **block client** their operations.
The client code will continue its task when it receives the HTTP server response. So this operation called **Synchronous communication**.

Asynchronous communication

- In Asynchronous communication, the client sends a request but it doesn't wait for a response from the service. **So the key point here is that, the client should not have blocked a thread while waiting for a response.**
- The most popular protocol for this Asynchronous communications is **AMQP (Advanced Message Queuing Protocol)**. So with using **AMQP protocols**, the client sends the message with using message broker systems like **Kafka** and **RabbitMQ queue**. The message producer usually does not wait for a response. This message consume from the subscriber systems in **async** way, and no one waiting for response suddenly.

An **asynchronous communication** also divided by 2 according to implementation. An asynchronous systems can be implemented in a

one-to-one(queue) mode

one-to-many (topic) mode.

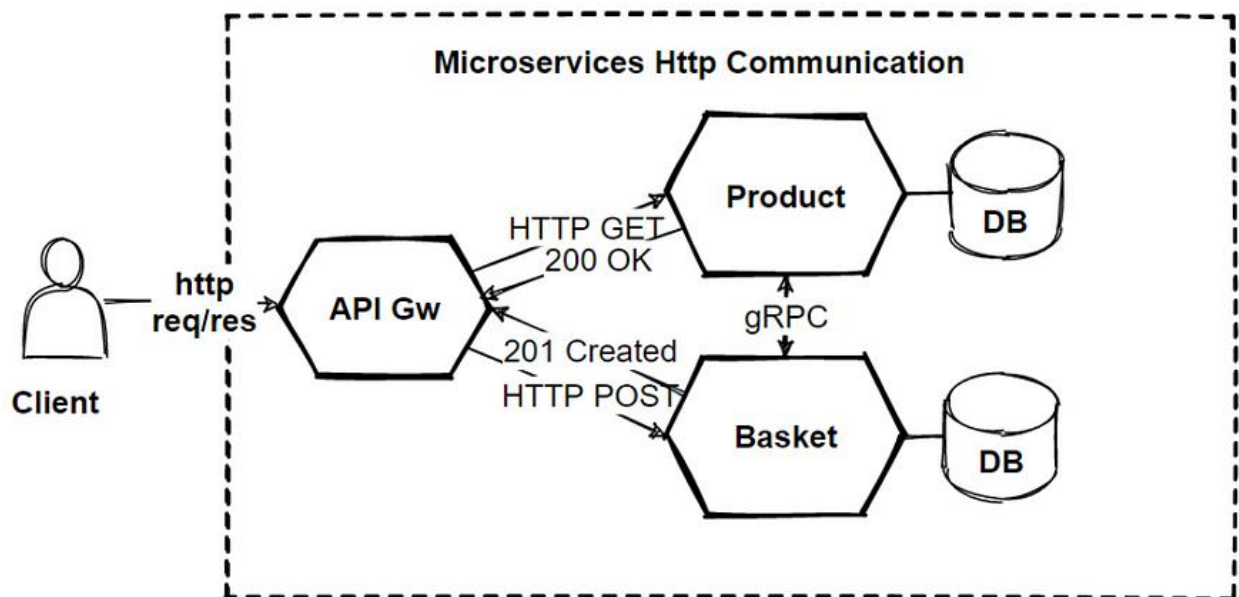
In a **one-to-one(queue)** implementation there is a single producer and single receiver. But in **one-to-many (topic)** implementation has Multiple receivers. Each request can be processed by zero to multiple receivers. **one-to-many (topic)** communications must be asynchronous.

So we will see this communication with the **publish/subscribe** mechanism used in patterns like **Event-driven microservices** architecture in the upcoming articles.

Basically an **event-bus** or **message broker** system is publishing events between multiple microservices, and communication provide with subscribing these events in an **async way**.

If we're communicating between services internally within our microservices cluster, we might also use **binary format communication mechanisms like gRPC**. gRPC is one of the best way to communicate for internal microservice communication, we will see **gRPC** in the upcoming sections.

Designing HTTP based RESTful APIs for Microservices



There are **2 type of APIs** when designing sync communication in microservices architecture.

- 1- **Public APIs** which is APIs calls from the client applications.
- 2- **Backend APIs** which is used for inter-service communication between backend microservices.

For **Public APIs**, should be align with client request. Clients can be web browser or mobile application requests. So that means the public API should use **RESTful APIs** over **HTTP protocol**. So **RESTful APIs** should use **JSON** payloads for **request-response**, this will easy to check payloads and easy agreement with clients.

For the **backend APIs**, We need to consider network performance instead of easy readable **JSON payloads**. Inter-service communication can result in a lot of network traffic. For that reason, **serialization** speed and payload size become more important. So for the backend APIs, These protocols support **binary serialization** should implement. The protocol alternatives is using **gRPC** or other binary protocols are mandatory.

REST AND gRPC

REST is using HTTP protocol, and request-response structured **JSON** objects. API interfaces design based on **HTTP** verbs like **GET-PUT-POST** and **DELETE**.

gRPC is basically Remote Procedure Call, that basically invoke external system method over the binary network protocols. Payloads are not readable but its faster that **REST APIs**.

What is gRPC ?

gRPC (gRPC Remote Procedure Calls) **is an open source remote procedure call (RPC) system initially developed at Google. gRPC is a framework to efficiently connect services and build distributed systems.**

What is gRPC ?

- gRPC (gRPC Remote Procedure Calls)
- HTTP/2 protocol to transport binary messages
- Protocol Buffers, also known as Protobuf files
- Cross-platform client and server bindings



It is focused on high performance and uses the **HTTP/2** protocol to transport binary messages. It relies on the **Protocol Buffers language** to define service contracts. Protocol Buffers, also known as **Protobuf**, allow you to define the interface to be used in service to service communication regardless of the programming language

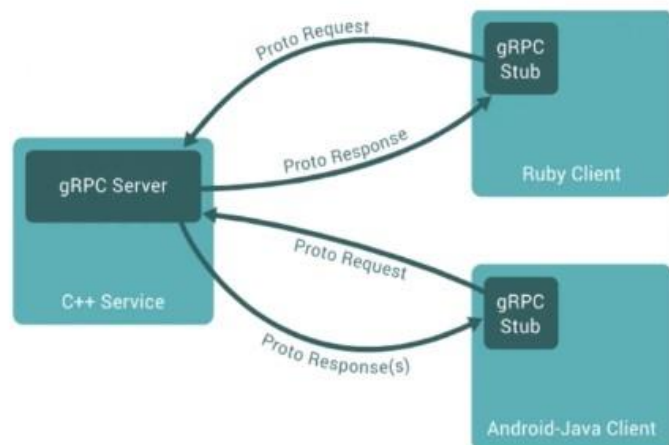
It generates cross-platform client and server bindings for many languages. Most common usage scenarios include connecting services

in microservices style architecture and connect mobile devices, browser clients to backend services. The **gRPC framework** allows developers to create services that can communicate with each other efficiently and independently from their preferred programming language.

Once you define a contract with **Protobuf**, this contract can be used by each service to automatically generate the code that sets up the communication infrastructure. This feature simplifies the creation of service interaction and, together with high performance, makes **gRPC** the ideal framework for creating microservices.

How gRPC works ?

- gRPC directly call a method on a server
- Build distributed applications and services
- Defining a service that specifies methods that can be called remotely
- Can be written in any language that gRPC supports



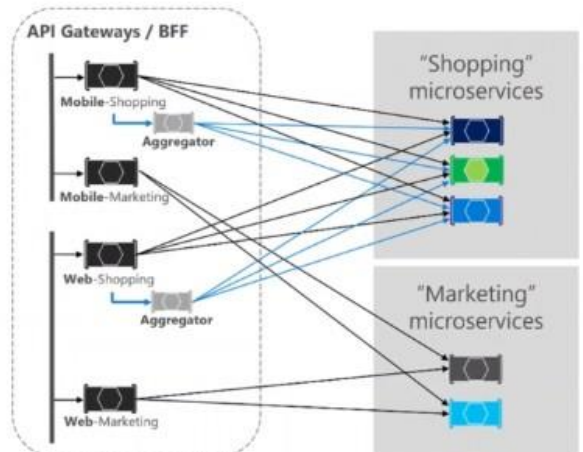
- In **gRPC**, a client application can directly call a method on a server application on a different machine like it were a local

object, making it easy for you to build **distributed applications** and services.

- As with many RPC systems, **gRPC** is based on the idea of defining a service that specifies methods that can be called remotely with their parameters and return types. On the server side, the server implements this interface and runs a **gRPC** server to handle client calls. On the client side, the client has a stub that provides the **same methods** as the server.

gRPC usage of Microservices Communication

- Synchronous backend microservice-to-microservice communication
- Polyglot environments
- Low latency and high throughput communication
- Point-to-point real-time communication
- Network constrained environments

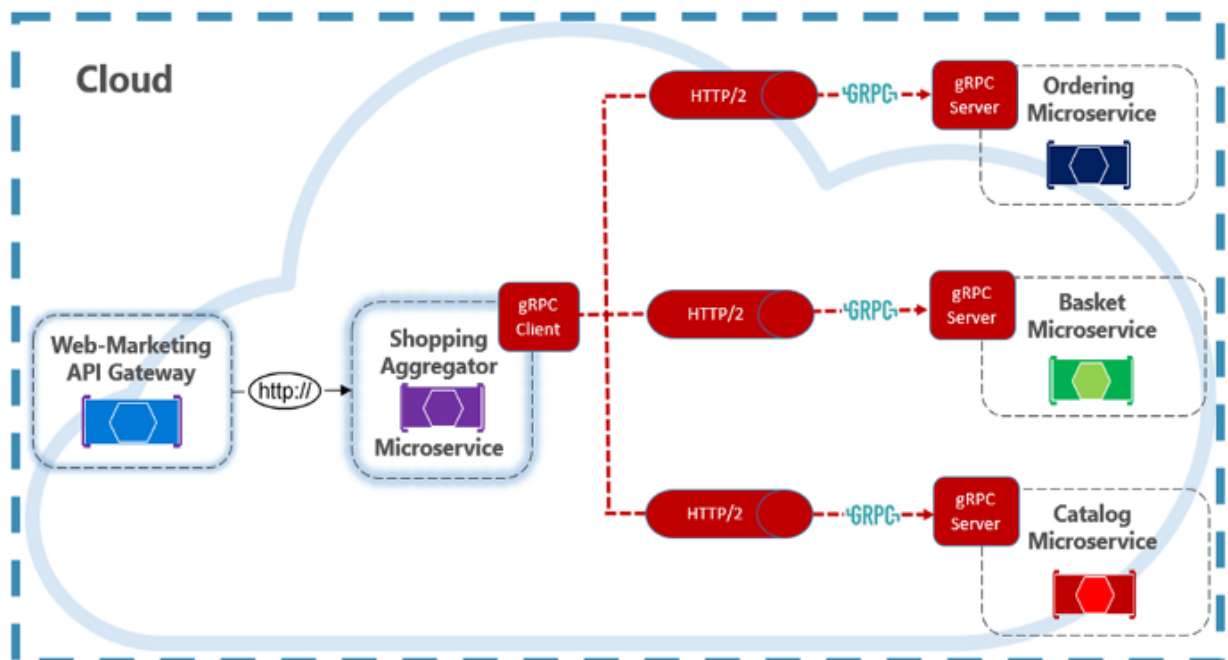


An aggregator

An aggregator microservice is a component of a microservices architecture that is responsible for collecting and combining data from multiple sources or services. Its primary function is to aggregate information and present a unified view or result to the user or other parts of the system.

This **Shopping Aggregator** Microservice receives a single request from a client, dispatches it to various microservices, aggregates the results, and sends them back to the requesting client. Such operations typically require synchronous communication as to produce an immediate response.

Example of gRPC in Microservices Communication



gRPC communication requires both client and server components. You can see that **Shopping Aggregator** implements a gRPC client. The client makes synchronous gRPC calls to backend microservices, this backend microservices are implement a **gRPC server**. As you can see that, The gRPC endpoints must be configured for the **HTTP/2** protocol that is required for gRPC communication.

In microservices world, most of communication use **asynchronous communication patterns** but some operations require direct calls. **gRPC** should be the primary choice for **direct synchronous communication** between microservices. Its high-performance communication protocol, based on **HTTP/2** and protocol buffers, make it a perfect choice.

Drawbacks of the direct client-to-microservices communication

We will compare the **API gateway pattern** and the Direct **client-to-microservice communication**. We have understand how to design Restful APIS for our microservices architecture. So for every microservices should exposes a set of **fine-grained endpoints** to communicate each other.

In this view, each microservice has a **public endpoint**, and when we open **public endpoint** from our microservices, it has lots of **drawbacks** that we should consider.

When you build **large** and **complex microservice-based applications** for example, when handling dozens of microservices, than these **direct-to-microservices communication** can make problems.

The client try to **handle multiple calls** to **microservice endpoints** but this is not **manageable**. Also if we think that new microservices can be add our application, its really hard to manage those from the client application.

If we expand these problems; It can cause to **lots** of **requests** to the **backend services** and it can create possible **chatty communications**. This approach

increases **latency** and **complexity** on the UI side. Ideally, responses should be aggregated in the server side.

Also implementing security and **cross-cutting concerns** like **security** and **authorization** for every microservice is not a good way of implementations. These **cross-cutting concerns** should handle in centralized place that can be in internal cluster. Also if there is a **long-running apis** that need to work on **async communications**, its hard to implement event-driven publish-subscribe model with **message brokers** from the client applications.

So these are the **drawbacks** of the **direct client-to-microservices** communication. Instead of that we should use **API Gateways** between client and internal microservices.

API Gateways can handle that **Cross-cutting concerns** like **authorization** so instead of writing every microservices, authorization can handle in **centralized API gateways** and sent to internal microservices.

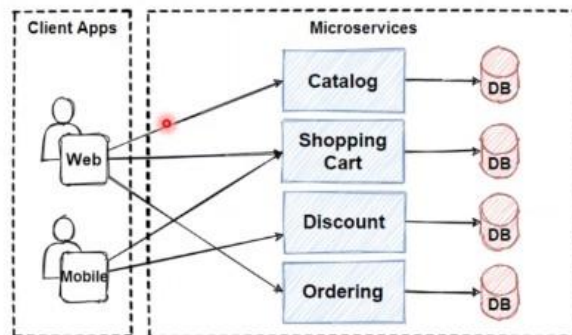
Also API gateway manage routing to internal microservices and able to aggregate several microservice request in 1 response.

problem

Drawbacks of the direct client-to-microservices communication

- Direct client-to-microservice communication problems
- Expose fine-grained endpoints
- Client try to handle multiple call
- Chatty communications
- Increases latency and Complexity on the UI side

Microservices Communications without API Gw

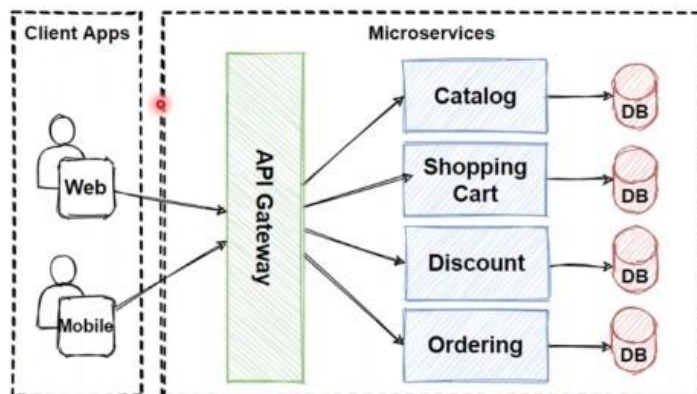


solution

Microservices Communication Design Patterns - API Gateways

- API gateway pattern
- Cross-cutting concerns
- Routing to internal microservices
- Aggregate several microservice

Microservices Architecture - Api Gw



Design API Gateway — Microservices Sync Communications Design Patterns (HTTP, gRPC)

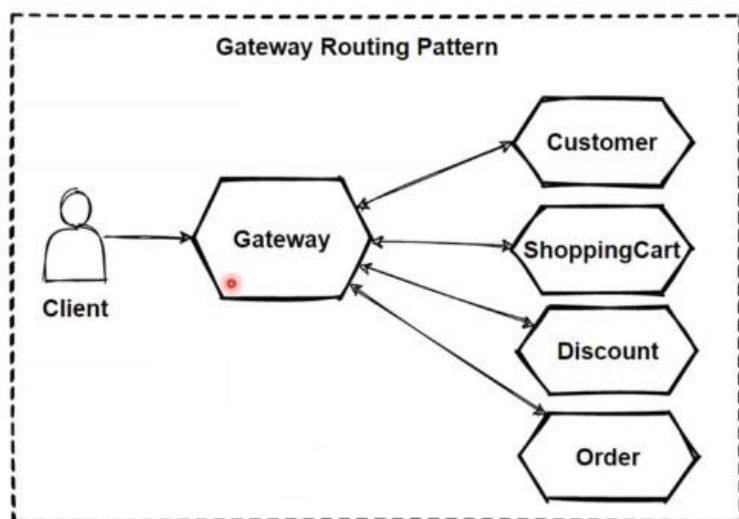
Reference : <https://medium.com/design-microservices-architecture-with-patterns/api-gateway-pattern-8ed0ddfce9df>

3 Main Pattern

- Gateway Routing Pattern
- Gateway Aggregation Pattern
- Gateway Offloading Pattern

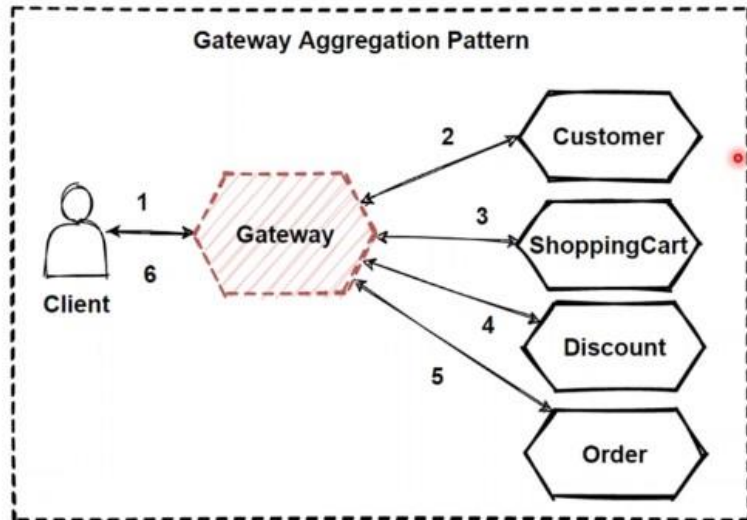
Gateway Routing pattern

- Route requests to multiple microservices
- Exposing a single endpoint
- Layer 7 routing
- Protocol Abstraction
- Centralized Error Management



Gateway Aggregation pattern

- Aggregate multiple internal requests
- Different backend microservices
- Manage direct access services
- Reduce chattiness communication
- Dispatches requests



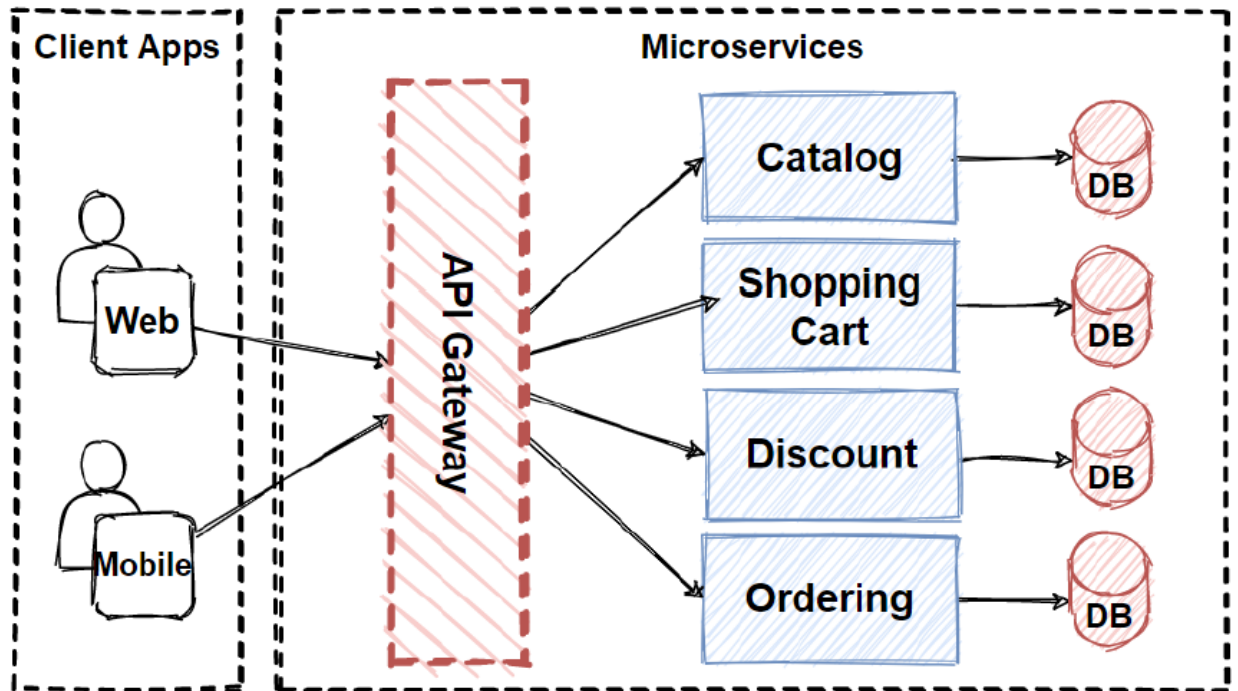
API Gateway Pattern

The **API gateway pattern** is recommended if you want to design and build **complex large microservices-based applications** with multiple client applications. The pattern is similar to the **facade pattern** from object-oriented design, but it is part of a distributed system **reverse proxy** or **gateway routing** for using as a synchronous communication model.

it provides a **single entry point** to the APIs with encapsulating the underlying system architecture. The pattern provides a **reverse proxy** to **redirect** or **route requests** to your internal microservices endpoints. An API gateway provides a single endpoint for the client applications, and it **internally maps** the requests to internal microservices.

the **API gateway** locate between the client apps and the internal microservices. It is working as a **reverse proxy** and routing requests from clients to backend services. It is also provide **cross-cutting concerns** like **authentication**, **SSL termination**, and cache

You can see the image that is collect client request in **single endpoint** and route request to internal microservices.



So there are several client applications connect to **single API Gateway** in here. We should careful about this situation, because if we put here a single API Gateway, that means its possible to **single-point-of-failure risk** in here. If these client applications increase, or adding more logic to **business complexity** in API Gateway, it would be **anti-pattern**.

e need to **careful** about using **single API Gateway**, it should be segregated based on **business boundaries** of the client applications and not be a **single aggregator** for all the internal microservices.

Main Features of API Gateway Pattern

Ocelot Features	
Routing	Authentication
Request Aggregation	Authorization
Service Discovery with Consul & Eureka	Throttling
Load Balancing	Logging, Tracing
Correlation Pass-Through	Headers/Query String Transformation
Quality of Service	Custom Middleware

Reverse proxy or gateway routing

This is part of gateway routing pattern features. The API Gateway provides reverse proxy to redirect requests to the endpoints of the internal microservices. Usually, It is using **layer 7 routing** for HTTP requests for request redirections. This routing feature provides to decouple client applications from the internal microservices. So it is separating responsibilities on **network layer**. Another benefit is abstracting internal operations, API GW provide abstraction over the backend microservices, so even there is changes on backend microservices, it won't be affect to client applications. That means don't need to update client applications when changing backend services.

Requests aggregation

This is part of gateway aggregation pattern features. API Gateway can **aggregate multiple internal microservices** into a single

client request. With this approach, the client application sends a single request to the API Gateway. After that API Gateway **dispatches several requests** to the internal microservices and then aggregates the results and sends everything back to the client application in 1 single response. The main benefit of this gateway aggregation pattern is to reduce chattiness communication between the client applications and the backend microservices.

Cross-cutting concerns and gateway offloading

This is part of gateway offloading pattern features. Since API Gateway handle client request in centralized placed, its best practice to implement cross cutting functionality on the API Gateways.

The cross-cutting functionalities can be;

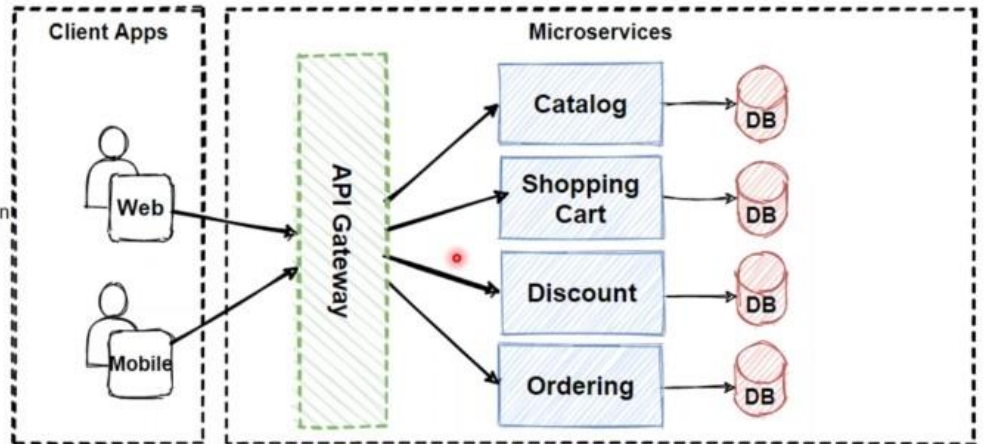
- **Authentication** and **authorization**
- **Service discovery** integration
- Response **caching**
- **Retry policies, circuit breaker**, and QoS
- **Rate limiting** and throttling
- **Load balancing**
- Logging, tracing, correlation

- Headers, query strings, and claims transformation
- IP allowlisting

Microservices Architecture - Api Gw

Principles

- KISS
- YAGNI
- SoC
- SOLID
- Gateway Routing
- Gateway Aggregation
- Gateway Offloading
- Api Gw
- Database per Microservices

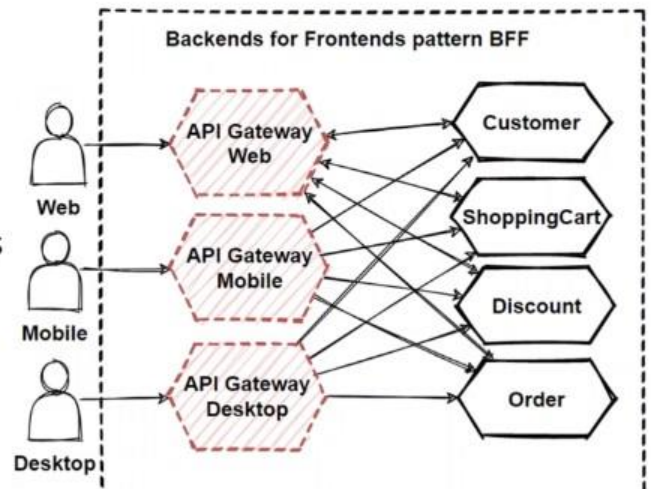


Backends FOR Frontends Patterns BFF

<https://medium.com/design-microservices-architecture-with-patterns/backends-for-frontends-pattern-bff-7ccd9182c6a1>

Backends for Frontends pattern BFF

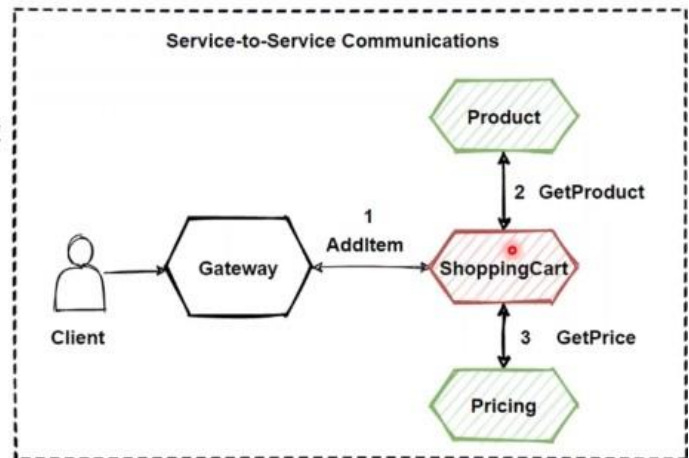
- Separate api gateways
- API Gateway for handling to routing and aggregate operations
- Grouping the client applications
- Avoid bottleneck of 1 API Gw
- Several api gateways as per user interfaces



Service to Service Communications Patterns

Service-to-Service Communications between Backend Internal Microservices

- Sync request comes from the clients
- Client requests are required to visit more than one internal microservices
- Reducing inter-service communication
- Query request to internal microservices



- Dependent and coupling service

Service-to-Service Communications – Chain Queries

- Much chain calls
- Create order use case
- Request/Response sync Messaging pattern
- Increase latency and negatively impact the performance

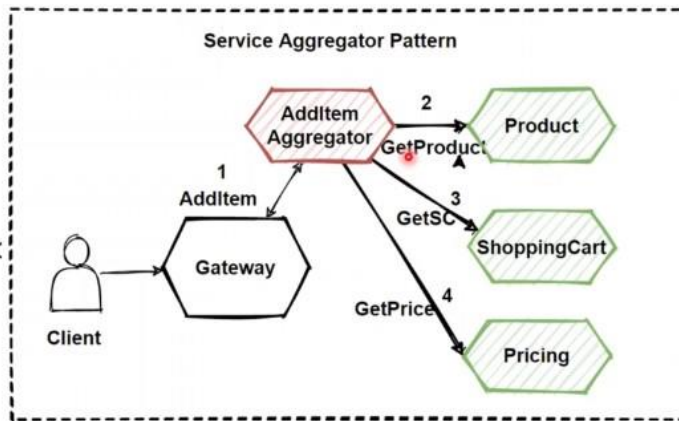


If middle microservices is down its effect entire system . To avoid such condition :-

Service Aggregation Pattern

Service Aggregator Pattern

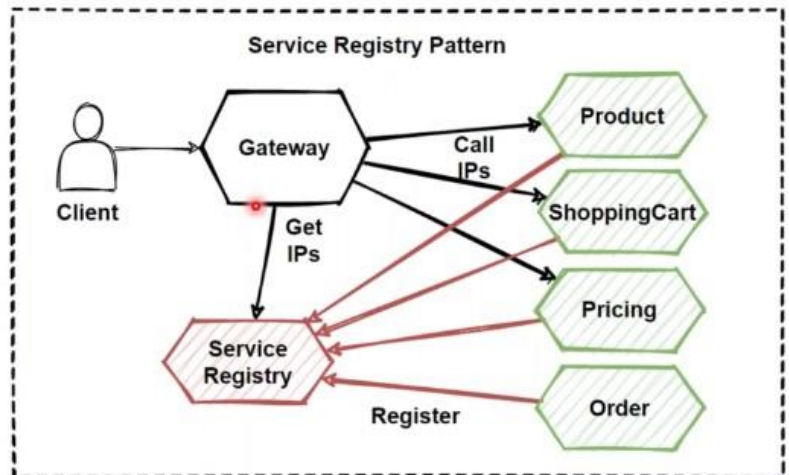
- Minimize service-to-service communications
- Receives a request from api gw
- Dispatchs requests of multiple internal backend microservices
- Combines the results



Service Registry Pattern

Service Registry Pattern

- Microservice Discovery Patterns
- Register and discover microservices
- Microservices should register to Service Registry
- Finds the location of microservice



Microservices Asynchronous Message-Based Communication(async AMQP)

<https://medium.com/design-microservices-architecture-with-patterns/microservices-asynchronous-message-based-communication-6643bee06123>

Synchronous communication

microservices need to call each other and wait some **long operations** until finished

that **dependency** and **coupling** of microservices will create **bottleneck** and create serious problems of the architecture.

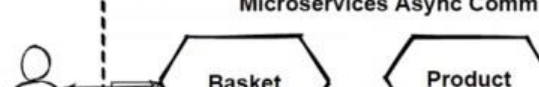
chain of request and **highly coupled depended**

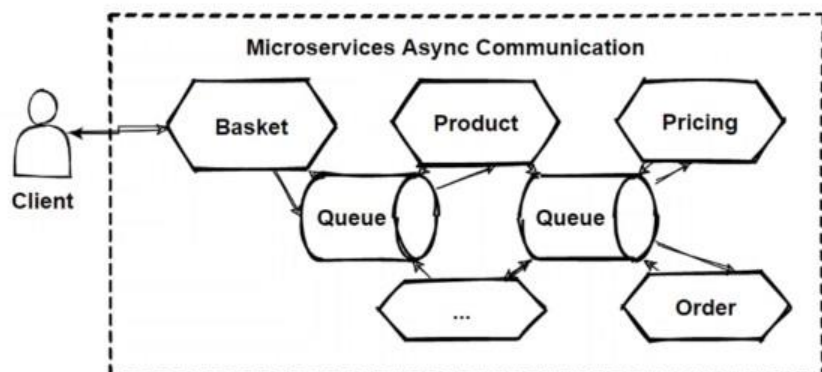
If we have a few interaction with querying microservices then we should use **HTTP request/response** with resource APIs. But when it comes to busy interactions in communication across multiple microservices, then we should use **asynchronous messaging platforms** like **message broker** systems.

Asynchronous

multiple microservices are required to **interact** each other and if you want to interact them **without** any **dependency** or make **loosely coupled**, then we should use **Asynchronous message-based communication** in Microservices Architecture. Because **Asynchronous** message-based communication is providing works with **events**. So events can place the communication between microservices. We called this communication is a **event-driven communication**.

Asynchronous Message-Based Communication

- Multiple microservices are required to interact each other
 - Without any dependency or make loosely coupled
 - Message-based communication
 - Eventual consistency
 - Message broker systems
 - AMQP protocol
 - Kafka and Rabbitmq
- 
- The diagram illustrates asynchronous communication between microservices. A **Client** (represented by a person icon) interacts with a **Basket** microservice (hexagon). The **Basket** microservice sends a message to a **Queue** (cylinder). The **Queue** then distributes the message to multiple microservices, including **Product** (hexagon) and an unnamed microservice (hexagon), which are part of the **Microservices Async Communication** system. The **Queue** acts as a message broker, ensuring that the message is received by all intended recipients.

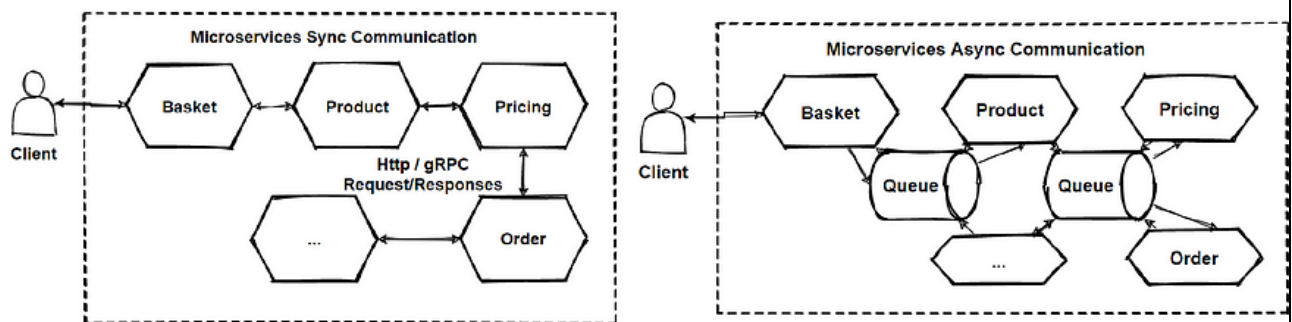


That means if any changes happens in the **Domains** of microservices, it is propagating changes across multiple microservices as an event after that these events consumed by **subscriber microservices**. This **event-driven communication** and **asynchronous messaging** brings us

“**eventual consistency**” that we will discuss this in the upcoming articles.

AMQP protocol, the producer send a message and doesn't wait a response. It only send message and expects that it will consume by **subscriber services** via to message broker systems.

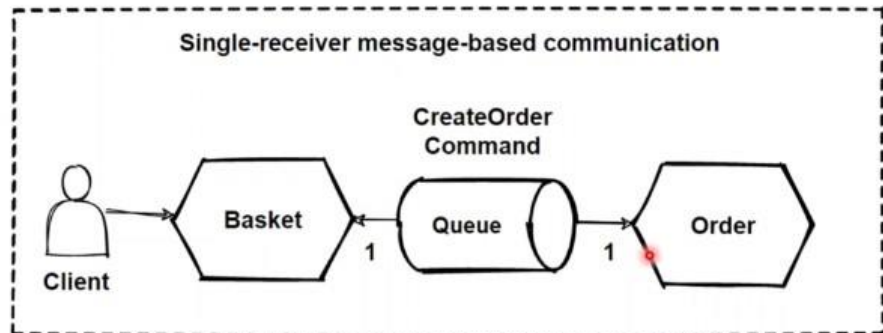
The client microservice sending a message or event to the message broker systems and no need to wait reply. Because it aware of this is **message-based communication**, and it won't be **respond immediately**. A message or event can includes some data. And these messages are sent through asynchronous protocols like AMQP over the message broker systems like **Kafka** and **Rabbitmq**.



2 Type of Asynchronous Messaging Communication

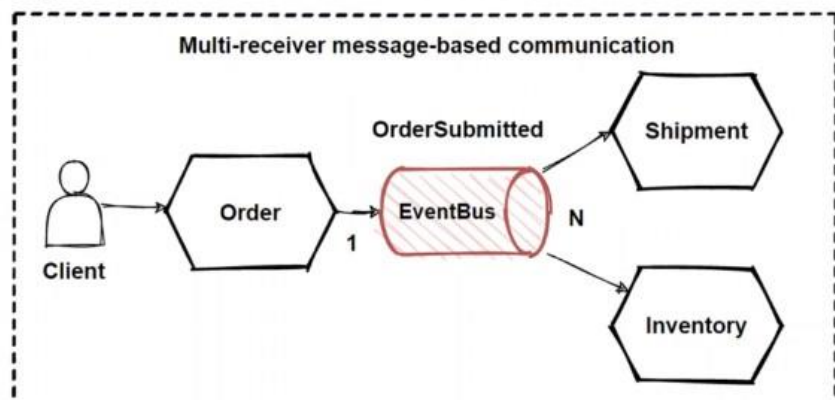
Single-receiver message-based communication

- One-to-one or point-to-point communications
- Without any dependency or make loosely coupled
- Message-based Communication
- CreateOrder event



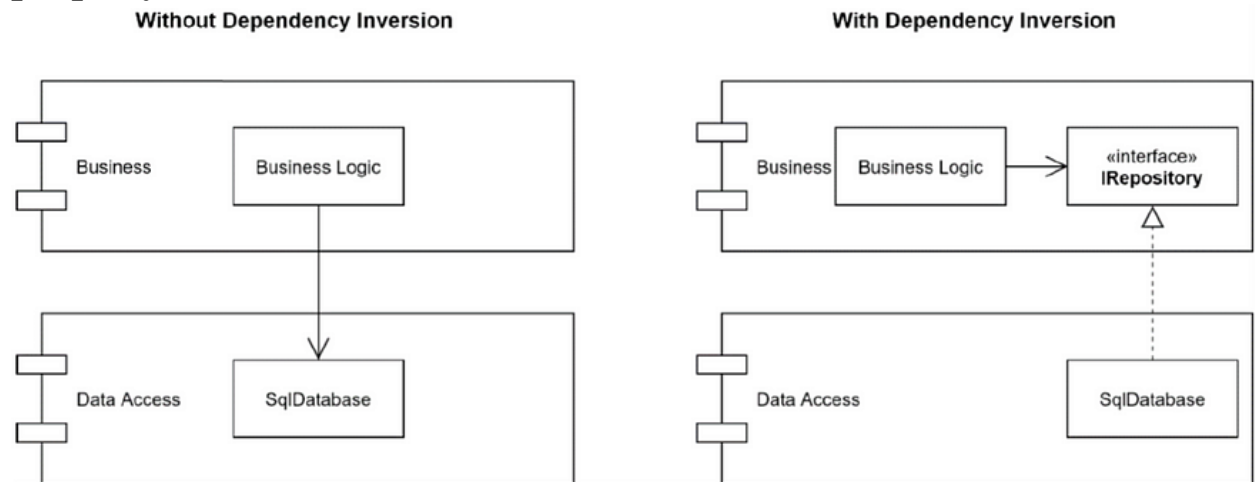
Multiple-receiver message-based communication

- One-to-many or publish/subscribe communications
- Service publish a message and it consumes from several microservices
- Event bus interface
- OrderSubmitted Event
- Event-driven Architecture, CQRS pattern, event storing, eventual consistency principles



Design Principles — Dependency Inversion Principles (DIP)

this design principles for software development and it provide to **broke dependency of classes** by **inverting dependencies** and inject dependent classes via **constructor** or property of main class.



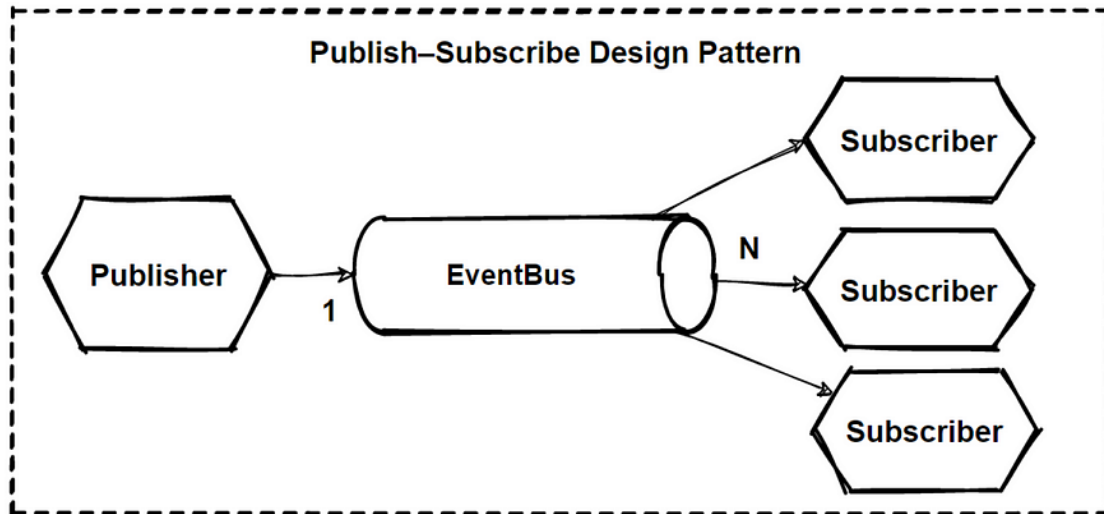
Briefly explain this method; **Upper-level modules** or classes and **lower-level classes** must not be dependent on modules. Lower-level modules must be dependent on higher-level modules (interfaces of modules). For short, we call it **Dependency Inversion**.

See the above picture again, On the **left side** we found an **Layered Application** where the Business Logic depends on the SqlDatabase implementation. It is a coupled way to write code.

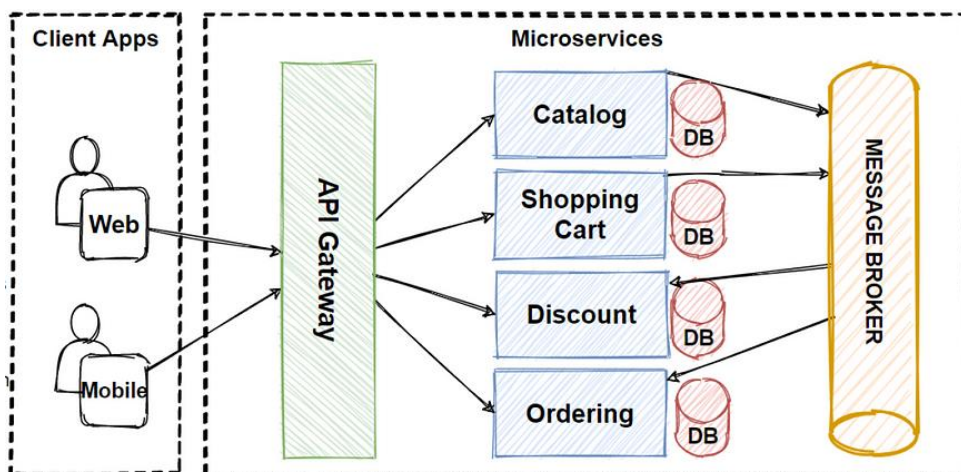
On the **right side**, by adding an **IRepository** and applying **DIP** then the **SqlDatabase** has its dependency pointing inwards. So its basically broke the dependency with layers and inject them with using DIP principle.

Publish-Subscribe Design Pattern

Publish-subscribe is a messaging pattern, that has sender of messages which's are called **publishers**, and has specific receivers which's are called **subscribers**.



publishers don't send the **messages** directly to the subscribers. Instead categorize **published messages** and send them into message broker systems without knowledge of which **subscribers** are there. Similarly, subscribers express **interest** and only receive messages that are of interest, without knowledge of which **publishers** send to them.



Kafka and RabbitMQ Architecture

Message Brokers ?

1- **Kafka**

2- **RabbitMQ**

Message broker

Message brokers play a crucial role in microservices architectures by facilitating communication and coordination between loosely coupled, independently deployable services. Here's how message brokers are typically used in microservices

Some features:

- **Decoupling**
- **Async Communication**
- **Event-Driven Architecture**
- **Scalability**
- **Fault Tolerance**
- **Load Balancing**
- **Ordering and Sequencing**
- **Integration with Legacy System**
- **Handling Bursty Loads**
- **Protocol Translanton**

What is Apache Kafka ?

- Open-source event streaming platforms
- Horizontally scalable, distributed, and fault-tolerant
- Distributed publish-subscribe
- Event-driven Architecture
- Topic Messages
- ZooKeeper sync
- Topics, Partitions, Brokers, Producer, Consumer, Zookeeper

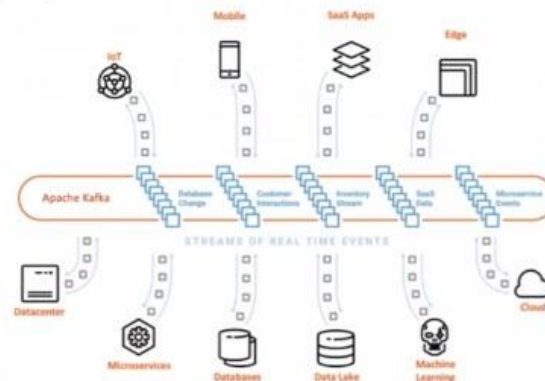


Apache Kafka Benefits and Use Cases

- Reliability, Scalability, Durability, Performance
- Built-in partitioning, replication and fault -tolerance

Use Cases

- Messaging
- Metrics
- Log Aggregation
- Stream Processing
- Activity Tracking
- Event Sourcing



- Global-scale
- Real-time
- Persistent Storage
- Stream Processing



Kafka Components - Topic, Partitions, Offset and Replication Factor

