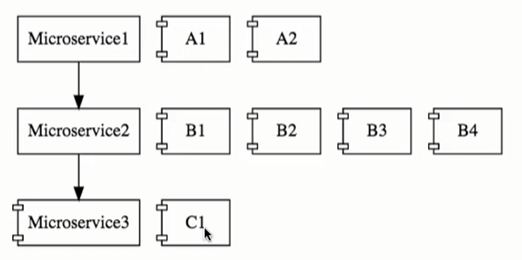
**Microservices Design Pattern Notes**

**Q1. Microservice, Microservice Architecture definition**

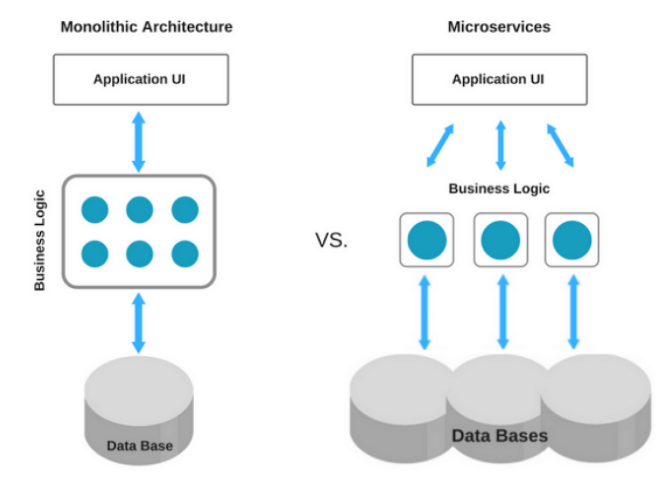
* Microservice Architecture is an approach to develop a single application as a suite of small autonomous services called microservices, each running its own process, loosely coupled & communicate with each other over standard protocols.
* These services are built around business capabilities & independently deployable by fully automated deployment machinery.
* Microservices are RESTful web services, well-chosen small deployable units & cloud enabled. Cloud enabled means if there is more load on one microservice, then we can easily bring up another instance of that microservice. Same if there is less load one microservice.



* The communication b/w microservices is a stateless communication where each pair of request & response is an independent transaction.
* In MSA, data is federated i.e., each microservice is responsible for its own data model & data.

**Q2. Monolithic Architecture**

* Monolithic Architecture is an approach to develop the entire application as a single piece that is designed to be self – contained i.e., all the components of the application are interconnected & tightly coupled i.e., each component along with its associated components must be present in order to execute the functionality.



|  |  |
| --- | --- |
| **Advantages of Monolith**  1. Easy to develop, test & deploy for small apps  2. Easy to scale by having multiple copy of same app (horizontal scaling)  3. Less technicality  4. Better for small scale apps | **Challenges with Monolith Architecture**  1. Limitation in size & Complexity grows with time.  2. Slow development & deployment for large apps  3. Blocks continuous development & delays new feature release.  4. More time to spend the fix for production bug.  5. Even small change in one module/component needs redeployment of whole app.  6. Unscalable in terms of each component, Unreliable because of Single point of failure, tightly coupled components, Inflexible.  7. High Dependency on few team members & hiring new team & making them understand whole app is tough.  8. Stuck in one technology. |
| **Advantages of Microservice Architecture**  1. Domain expertise i.e., we can outsource additional microservices.  2. Easy & Quick to scale – on demand in terms of individual microservices.  3. Isolated decision making in terms of individual microservices.  4. Self-Organization because of small team size.  5. Quick response to change  6. Increase uptime or Availability  7. Can experiment with any Tech  8. Loose coupling  9. Service reusability  10. Agile, SCRUM  11. Best for large scale apps | **Challenges to Microservice Architecture**  1. Additional complexity with distributed systems.  2. Deployment complexity  3. Visibility & Monitoring complexity  4. Increased resource consumption  5. Communication among services is challenging  6. Testing each service is also a challenge  7. Bounded context: how to identify boundary for each microservices.  8. Configuration management  9. Dynamic scale up & down  10. Fault tolerance  We have N number of solutions to these challenges considering performance & resource expense. |

**Q3. Monolithic Vs Microservice Architecture**

|  |  |  |
| --- | --- | --- |
| **No.** | **Monolithic Architecture** | **Microservice Architecture** |
| 1. | Monolithic Architecture is like a big container where all the components of an application are assembled together. | Microservice Architecture is composed of small autonomous services where each service delivers a specific business goal. |
| 2. | Components are tightly coupled. | Components/Services are loosely coupled. |
| 3. | Fault tolerance is difficult i.e., If any specific feature is not working, the complete system goes down. | Fault tolerance is easy i.e., if even one service goes down, others can continue to work. |
| 4. | Service startup takes more time. | Service startup is relatively quick. |
| 5. | Application scaling is challenging & wasteful  i.e., since service is not isolated, individual resource allocation not possible. | Individual resource allocation is possible i.e., More H/w resources can be allocated to the service that is frequently used. |
| 6. | Low availability | High availability |
| 7. | Data is centralized. | Data is federated. This allows individual microservice to adopt a data model best suited for its needs. |
|  | Change in data model affects the entire DB. | Change in data model of one microservice doesn’t affect other microservice. |
| 8. | Large team & considerable team management effort is required. | Parallel & faster development small focused team. |
| 9. | NA | Interacts with other microservices by using well – defined interface. |
| 10. | Put emphasis on the entire project. | Microservice works on principle that focus on products not project. |

**Q4. Principle being Microservices**

* For developing an optimal microservice architecture, we need to follow below design principles:

1. Independent & Autonomous/ Self-governing services

* Small team size, Parallel development, individually deployable
* Clear contracts

1. Domain Driven

* Focus on business, Core Domain & Domain logic

1. Discoverable

* All services should be registered at one place.
* It makes client’s life easy when looking for specific service.

1. High Cohesion (Doing one thing only)

* Doing one thing only i.e., Single Responsibility Principle
* Helps in Scalability & Availability

1. Single Source of Truth

* There should be only one source to get the complete information (using Ids)
* This helps in avoiding the duplicity.

1. Resilient services / Fault isolation / Fault Tolerant / Design for Failure

* Avoid Single Point of Failure, Avoid cascading failure
* Consider failure as events & analyze it properly.

1. Real time load balancing
2. Seamless API integration & continuous monitoring / observation

* Centralized monitoring, logging, Health check system.

1. Decentralization

* Database for each service
* Choice of DB depends on the nature of particular service

1. Continuous delivery through Devops integration like Versioning, Cloud support etc.

**Q5. Challenges with building microservices**

1. Bounded context: how to identify boundary for each microservices.
2. Configuration management
3. Dynamic scale up & down
4. Visibility & monitoring
5. Fault tolerance

* To the typical problems which are present for distributed systems in the cloud, Spring Cloud provides a range of solutions.
* Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems. (e.g., Configuration management, service discovery, circuit breakers, intelligent routing, micro proxy, control bus, one time token, global locks, leadership election, distributed sessions, cluster state)

**Q6. What is design Pattern & why do we need design patterns**

* Software design pattern is defined as software template or a description to solve a problem that occurs in multiple instances while designing a software application or a software framework.
* To ensure that all the teams follow the same process or may be same pattern we can use the concept of design pattern.
* By using the design patterns, the team working on various projects use the same pattern to build the similar application.

**Q7. Microservice Design Pattern**

* Design patterns help in solving the specific microservice architecture challenges.
* It reduces the risk of failure in microservices.
* As per the software/application lifecycle phases, there are different architecture patterns to be followed:

1. Decomposition patterns

* By Business Capabilities
* By Subdomain
* Strangler Pattern
* Sidecar pattern / Service mesh

1. Database patterns

* Database per service
* Shared Database
* CQRS
* SAGA
* Event Sourcing

1. Communication among services

* Synchronous
* Asynchronous event / messaging based
* Communication Medium like HTTP REST – XML/JSON, GraphQL, gRPC

1. Integration Patterns

* API Gateway
* Aggregator Pattern
* Client-Side UI composition pattern

1. Deployment Patterns

* Multiple service instances per host
* Service instance per host
* Service instance per VM
* Service instance per Container
* Server less deployment
* Blue Green deployment
* Canary deployment

1. Observability Patterns

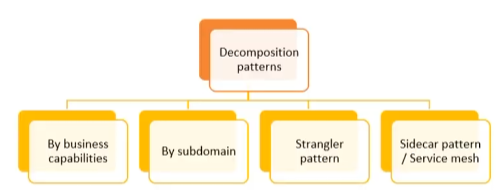
* Log Aggregation
* Performance metrics
* Distributed Tracing
* Health Check

1. Cross Cutting Concern Patterns

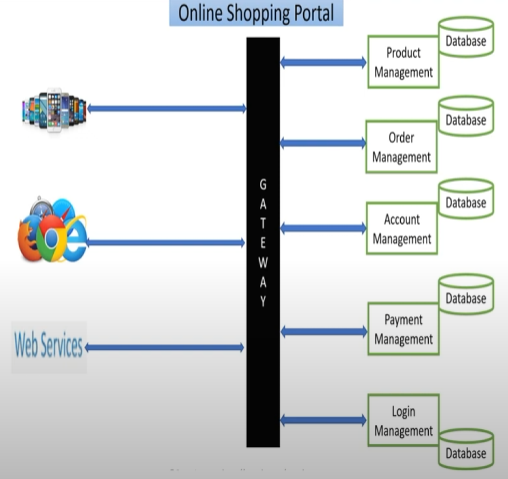
* External Configuration
* Service discovery pattern
* Circuit breaker pattern

**1. Decomposition**

* It comes when we’re designing microservice architecture.



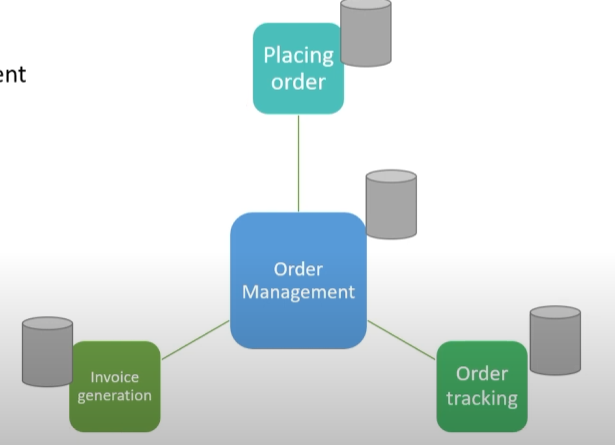
There are 2 kinds of project under microservices

1. Monolithic to Microservices – **Brown Field Projects**
2. Microservices in nature from scratch – **Green Field Projects**

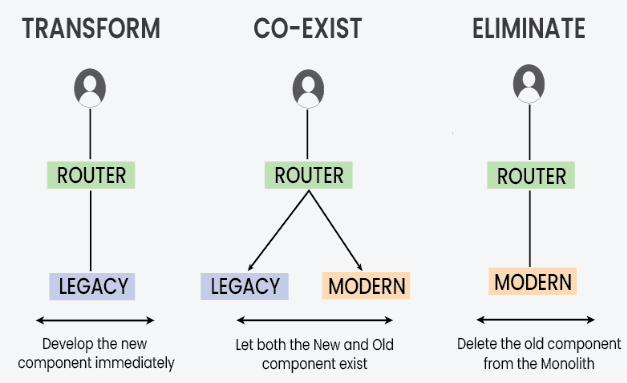
**Decomposition by Business Capabilities**

* The size of microservice is decided on the basic of business functionality (E.g., Product Management, Order Management, Account Management, Payment Management, Login Management)

**Decomposition by Subdomain**

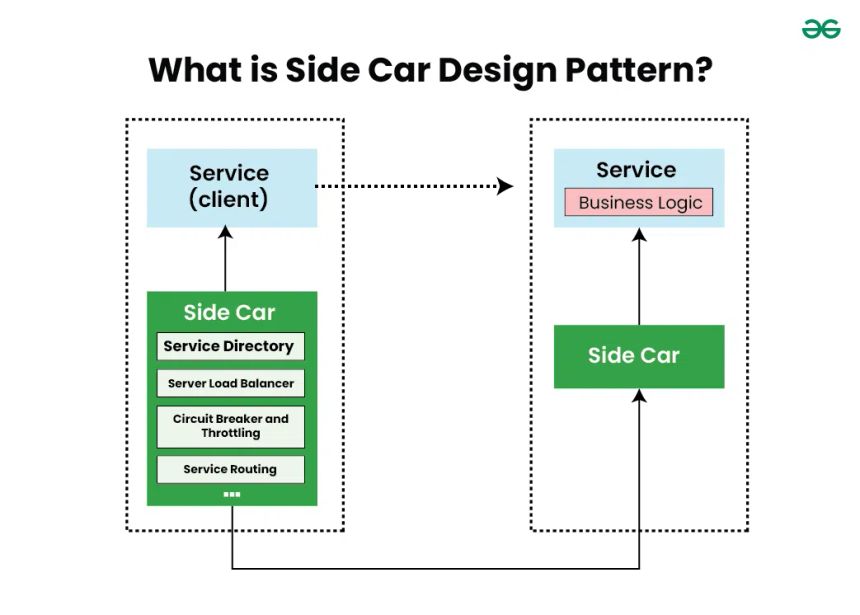
* God Classes
* ****When we break monolithic apps to microservice, there are God classes (classes that are shared/common among modules) & dividing these god classes is really difficult.
* In order to resolve the problem of God classes, decomposition by Subdomain comes into picture.
* Decomposition by Subdomain comprise of Domain Driven Design & Bounded context. For each bounded context, there is one microservice. (E.g., Order Management domain can have subdomains as Placing Order, Invoice generation, Order tracking etc.)

**Strangler Pattern / Strangler Fig**

* The Strangler pattern is an architectural approach employed during the migration from a monolithic application to a microservice – based architecture.
* It involves replacing parts of a monolithic application with microservices over time.
* In order to implement Strangler pattern, we need to follow 3 steps i.e., **Transform, Co-exists, Eliminate.**
* It proves beneficial in scenarios where complete system rewrites pose significant risks & disruption. This pattern is particularly suitable for legacy systems with complex codebases that’re challenging to refactor entirely.

**Sidecar Pattern / Sidekick Pattern**

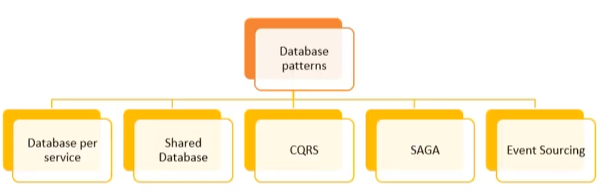
* In microservice architecture, it’s very common to have multiple services/apps often require common functionalities like logging, configuration, monitoring & networking services. These functionalities can be implemented & run as a separate service within the same container or in a separate container.



* When these functionalities are implemented in the same application, they are tightly linked & run within the same process by making efficient use of the shared resources. In this case, these components are not well segregated & they are interdependent which may lead to failure in one component can in-turn impacts another component or the entire application.
* In this pattern, a sidecar container or process is deployed alongside a primary application container to extend or enhance its functionality. The sidecar container runs within the same execution environment as the primary application & typically provides supporting services such as logging, monitoring, security or communication with other services.
* This pattern enables the primary application container to focus on its core functionality while offloading secondary tasks (logging, monitoring, security, service discovery, or communication proxies) to the sidecar, promoting modularity, scalability, and maintainability in distributed systems.
* Communication mechanism b/w microservices & sidecar instances: It occurs through inter-container communication mechanism provided by the container runtime or orchestration platform. Some mechanism includes:
* **Local Networking:** Microservices & Sidecar instances deployed within the same container network can communicate with each other using local networking. They can exchange messages over TCP/IP or UDP protocols using localhost or container – local IP addresses.
* **Shared Volumes:** Microservices & Sidecar instances ca share data or files through shared volumes mounted into their respective containers. This allows them to read from & write to common directories or files, enabling data exchange or synchronization.
* **Inter-Process-Communication:** Microservices & sidebar instances running within the same container environment can communicate through inter-process communication mechanisms provided by the OS, such as UNIX sockets or named pipes. This allows for efficient & low-latency communication between processes.
* **Message Brokers:** In some cases, microservices & sidecar instances may communicate indirectly through a message broker or event bus. Microservices can publish messages or events to a broker & sidecar instances can subscribe to these messages for processing or forwarding to other services.
* **Service Mesh:** In Service mesh architecture, communication between microservices & sidecar instances is often facilitated by a service mesh infrastructure. Sidecar proxies intercept & route traffic between microservices, providing features such as load balancing, circuit breaker & observability.
* For more refer: <https://www.geeksforgeeks.org/sidecar-design-pattern-for-microservices/>

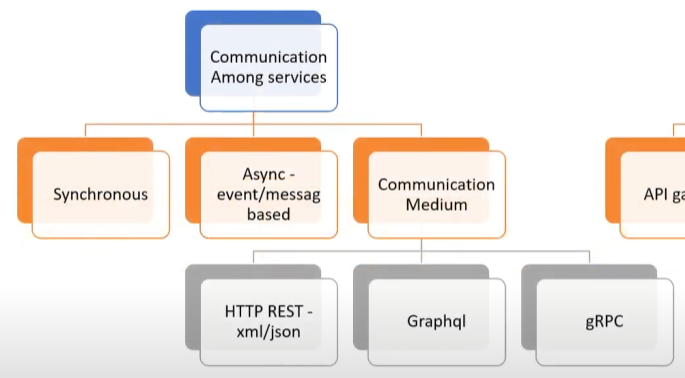
2. Database

* It comes when we’re required to communicate with DB for data storage & management.



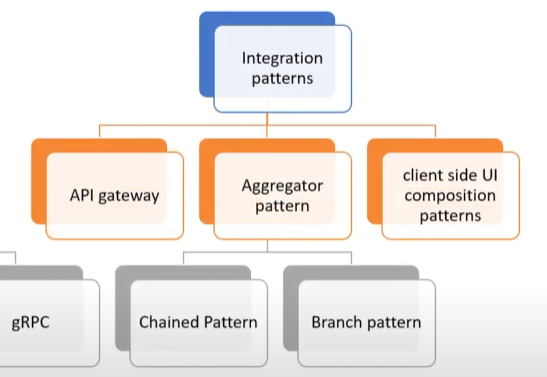
3. Communication among services

* It comes when microservices are required to communicate with each other.



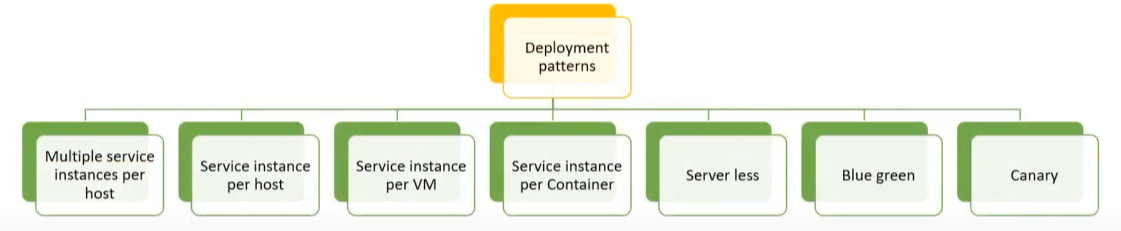
4. Integration

* It comes integrate request & response from multiple microservices & then response to the User.



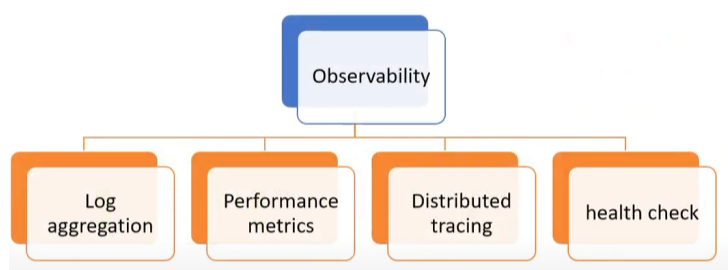
5. Deployment

* It comes when we’re required to deploy microservices.



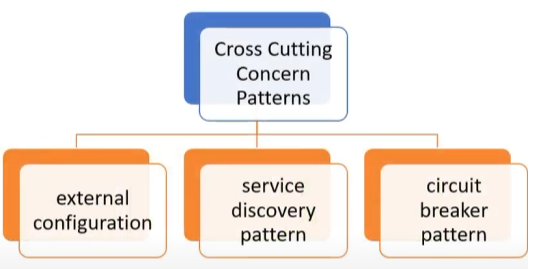
6. Observability / Monitoring

* It comes in production phase where we’re required to monitor each microservice health.



7. Cross – Cutting concern

* Whatever left comes under Cross – Cutting concerns.



1. Aggregator Design Pattern (based on DRY principle)

2. API Gateway Design Pattern

3. Chained or Chain of Responsibility

4. Asynchronous messaging Design Pattern

5. Database or Shared DB design Pattern

6. Event Sourcing Design Pattern

7. Command Query Responsibility Segregator (CQRS)

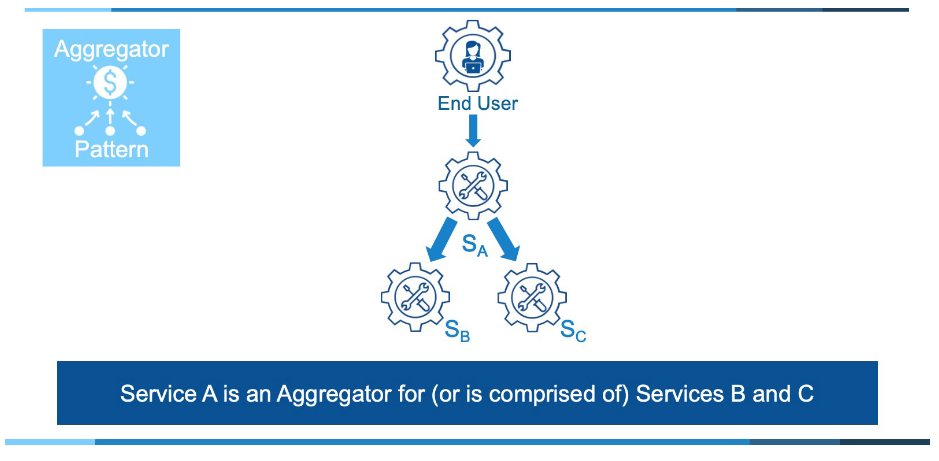
8. Circuit breaker design pattern

9. Branch Design pattern

10. Decomposition design pattern

**1. Aggregator Design Pattern**

* In Aggregator Design Pattern, when a service receives a request, it subsequently makes requests to multiple microservices, combines/aggregates the result & responds to the initiated request.
* Aggregator design pattern is based on DRY principle that states we can abstract the logic into composite microservices and aggregate that particular business logic into one service.



Drawbacks of the Aggregator Design Pattern

* We should avoid this pattern for a majority of our critical functionality. The pattern is an implementation of the AFK (Anti-Pattern Fan out) Microservice.
* Reduces availability as result of the multiplicative effect of failure.
* Likely increases latency of responses as a result of the anti-pattern.
* Increases complexity (cost) of troubleshooting which results in additional time to restore service & repair faults for many incidents.

When to use Aggregator pattern

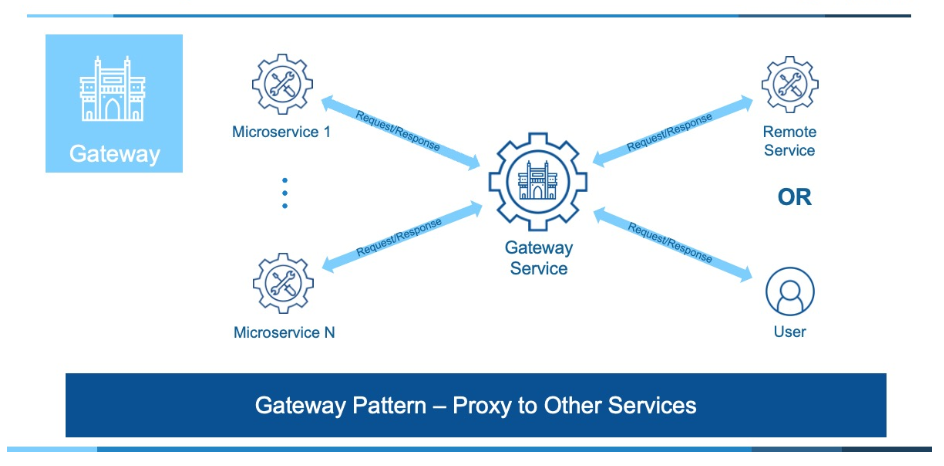
* Low Scale Needs
* Low Business Value
* Team Velocity
* Third Party calls
* This is a special exception to our advice not to use the aggregator pattern (or composite service).
* When service B & C are remote, 3rd party calls & the calls to service B & C are asynchronous to offset the high failure rate of distant 3rd party calls, the Aggregator pattern can be useful.
* Service A is built with circuit breakers & becomes the 3rd party proxy for all communications.

When to avoid the Aggregator pattern

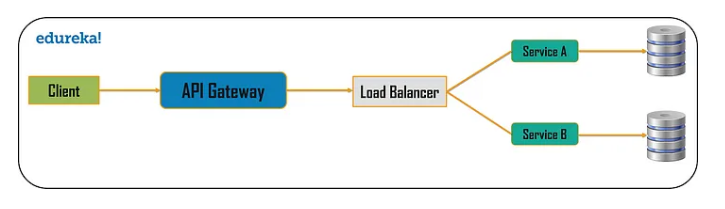
* High Scalability Needs
* High Availability Needs
* Solutions that comprise significant business value

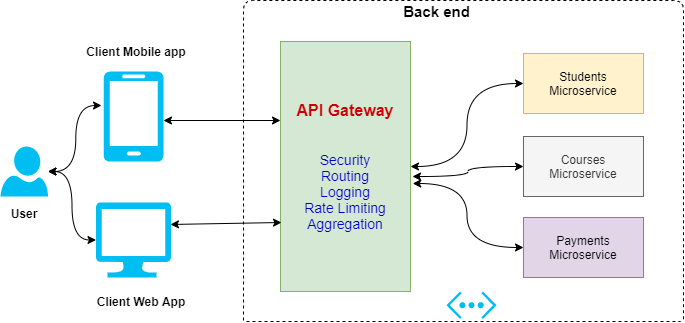
**2. API Gateway Design Pattern**

* API Gateway design pattern can be considered as the reverse proxy service to route a request coming from different kinds of client using different communication protocols to the concerned microservice.



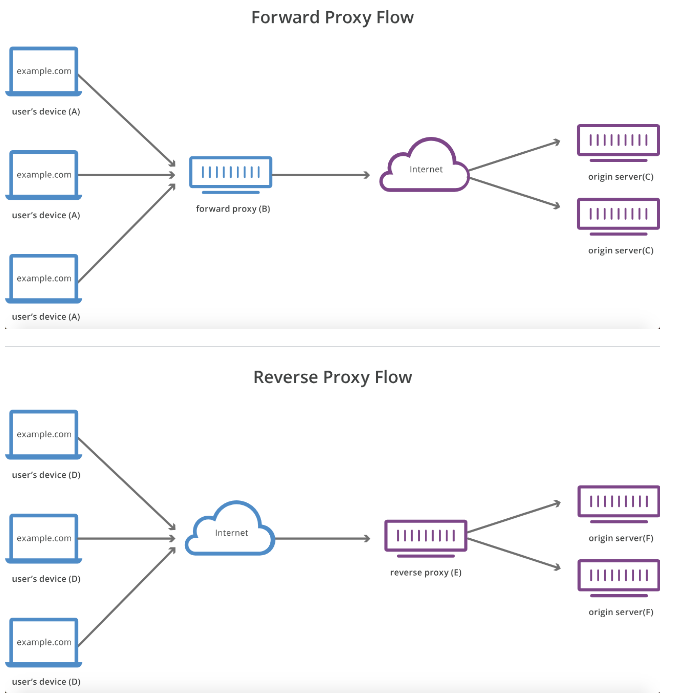
* The purpose of the Gateway is to offload common shared services needed for all traffic. These services include:
* Authentication & Authorization (AuthN & AuthZ)
* Security/access monitoring
* SSL termination & offloading
* Common certificate management (SSL & others)
* Throttling (as in rate-limiting activity or validating usage is consistent with contract/pricing)
* i.e.., Anything that can be shared for the purpose of simplifying solutions & reducing overall costs becomes a candidate for the Gateway pattern.
* Flow:
* So, once the client sends a request, these requests are passed to the API Gateway which acts as an entry point to forward the client’s request to the appropriate microservices.
* Then with the help of the load balancer, the load of the request is handled & the request is sent to the respective services.
* Microservices use Service Discovery which acts as a guide to find the route of communication b/w each of them.
* Microservices then communicate with each other via a stateless server i.e., either by HTTP request/Message Bus.





**Proxy Server Vs Reverse Proxy Server**

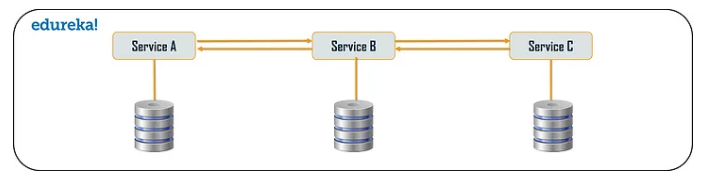
|  |  |  |
| --- | --- | --- |
| **No.** | **Proxy / Forward Proxy** | **Reverse Proxy** |
| 1. | It makes the request on the behalf of the client. | It receives the request on the behalf of the server. |
| 2. | The server will return the response to the proxy, & the proxy will forward the response to the client. | It forwards the request to the server, receives the response & then returns the response to the client. |
| 3. | The server will never learn who the client was (the client’s IP address); it will only know the proxy.  However, the client definitely knows the server, since it essentially formats the HTTP request destined for the server but it just hands it to the proxy. | In this case, the client will never learn who was the actual server (the server’s IP address); it will only know the proxy.  The server will or won’t know the actual client, depending on the configurations of the reverse proxy. |
| 4. | Forward proxies grant the client anonymity. | Reverse proxies grant back-end servers’ anonymity. |
| 5. | It hides identities of clients. | It hides identities of servers. |



Refer: <https://stackoverflow.com/questions/224664/whats-the-difference-between-a-proxy-server-and-a-reverse-proxy-server>

**3. Chained or Chain of Responsibility Pattern**

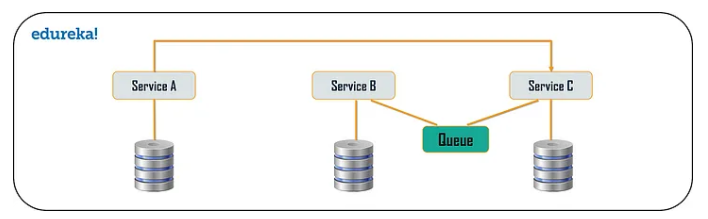
* It produces single output which is a combination of multiple chained outputs.
* All the services in this design pattern use synchronous HTTP request or response for messaging. Also, until the request passes through all the services & the respective responses are generated, the client doesn’t get any output.
* So, it’s always recommended to not make a long chain, as the client will wait until the chain is completed.



* One more important aspect that we need to understand is that the request from Service A to Service B may look different from Service B to Service C. Similarly, the response from Service C to Service B may look completely different from Service B to Service A.

**4. Asynchronous Messaging Design Pattern**

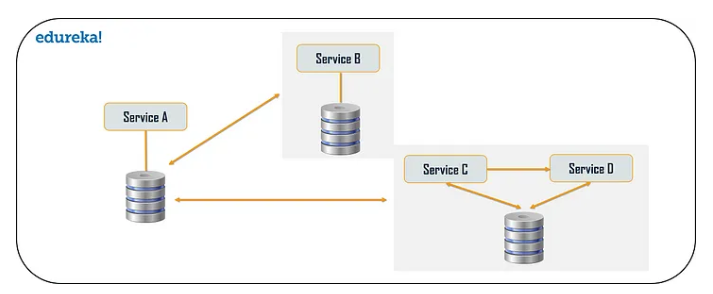
* From the Chain of Responsibility pattern, we know that the client gets blocked or has to wait for a long time in synchronous messaging.
* Instead of Synchronous, we can opt for Asynchronous messaging design pattern.
* In Asynchronous Messaging design pattern, all the services can communicate with each other but not sequentially.



* So, let’s say we have 3 services. The request from the client can be directly sent to the service C & service B simultaneously. The requests will be in a Queue.

**5. Database or Shared Database Design Pattern**

* For every application, there is a vast amount of data present. So, when we break down an application from its monolithic architecture to microservices, it’s very important to note that each microservice has a sufficient amount of data to process a request.



* So, either the system can have a database per service or it can have shared database per service to solve various problems:
* Duplication of data & inconsistency

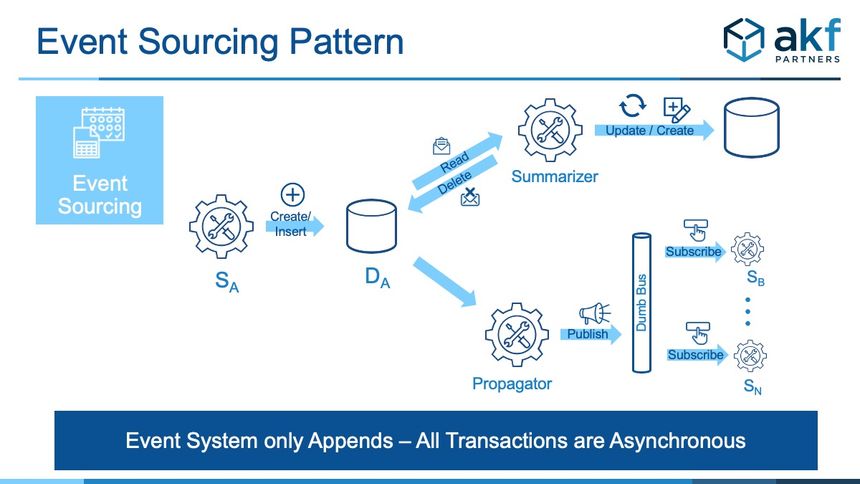
For these 3 problems, we can go with Shared DB

* Different services have different kinds of storage requirements
* Few business transactions can query the data, with multiple services
* De-normalization of data

We can choose shared databases per service, to align more than one DB for each microservice. This will help us gather data, for the monolithic applications which are broken down into microservices. But, we have to limit these DB to 2 – 3 microservice; else scaling these services will be a problem.

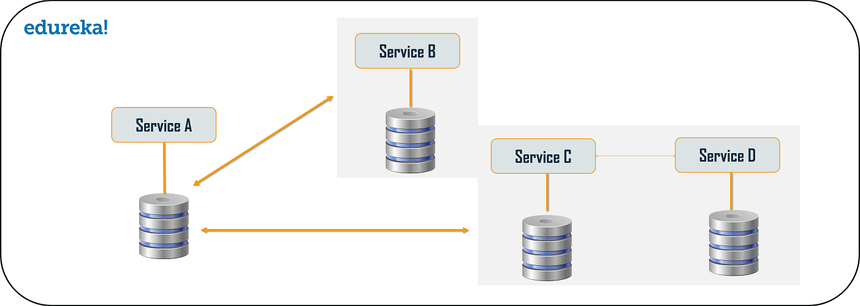
**6. Event Sourcing Design Pattern**

* The event sourcing design pattern creates events regarding the changes in the application state.
* Also, these events are stored as a sequence of events to help the developers track which change was made when. So, with the help of this, you can always adjust the application state to cope up with the past changes.
* We can also query these events, for any data change and simultaneously publish these events from the event store. Once the events are published, we can see the changes in the application state on the presentation layer.



**7. Branch Pattern**

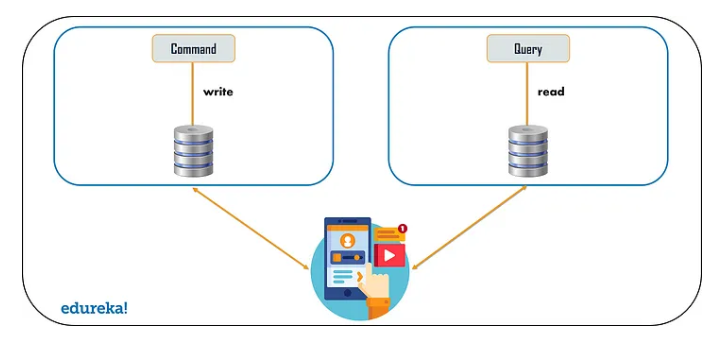
* Branch microservice design pattern is a design pattern in which we can simultaneously process the requests & responses from 2 or more independent microservices.



* So, unlike the chained design pattern, the request is not passed in a sequence, but the request is passed to 2 or more mutually exclusive microservices chains.
* This design pattern extends the Aggregator design pattern & provides the flexibility to produce responses from multiple chains or single chain.
* For e.g., If we consider an e-commerce application, then we may need to retrieve data from multiple sources & this data could be a collaborated output of data from various services. So, we can use the branching pattern, to retrieve data from multiple sources.

**8. Command Query Responsibility Segregator (CQRS) design pattern**

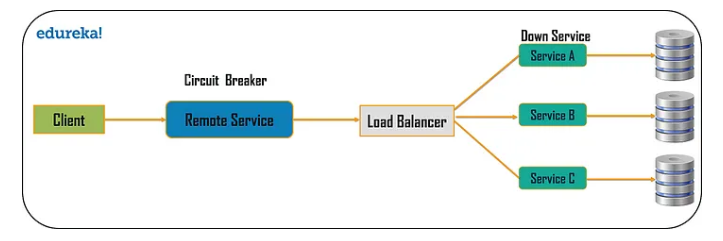
* Every microservice design has either the database per service model or the shared database per service. But in the database per service model, we can’t implement a query as the data access is only limited to one single database. So, in such scenario, we can use the CQRS pattern.



* According to CQRS design pattern, the application will be divided into 2 parts: Command & Query.
* The Command part will handle all the request related to CREATE, UPDATE & DELETE.
* The Query part will take care of the materialized views.
* The materialized views are updated through a sequence of events which are created using the Event Sourcing Design pattern.

**9. Circuit Breaker Pattern**

* The Circuit breaker design pattern is used to stop the process of request & response if a service is not working.
* For e.g., Let’s say a client is sending a request to retrieve data from multiple services but due to some issues, one of the services is down. Now, there are mainly 2 problems we will face:
* Since the client will not have any knowledge about a particular service being down, the request will be continuously sent to that service.
* The network resources will be exhausted with low performance & bad user experience.
* To avoid such problems, we use the Circuit breaker design pattern.



* With this pattern,
* The client will invoke a remote service via a proxy. This proxy will basically behave as a circuit barrier. So, when the no. of failures crosses the threshold number, the circuit breaker trips for a particular time period. All the attempts to invoke the remote service will fail for this timeout period.
* Once the time period is finished, the circuit breaker will allow a limited number of test to pass through & if those requests succeed, the circuit breaker resumes back to the normal operation else if there is a failure, then the time out period begins again.

**10. Decomposition Design pattern**

* Microservices are developed with an idea to create small services, with each having their own functionality. But breaking an application into small autonomous units has to be done logically. So, to decompose a small or big application into small services, we can use the Decomposition patterns.
* With the help of this pattern, either we can decompose an application based on business capability or on based on the sub-domains. For e.g., if we consider an e-commerce application, then we can have separate services for order, payment, customers, products if we decompose by business capability.
* But in the same scenario, if we design the application by decomposing the sub-domains, then we can have services for each & every class. Here, if we consider the customer as a class, then this class will be used in customer management, customer support etc.
* So, To decompose, we can use the Domain-Driven Design through which the whole domain model is broken down into sub-domains. Then each of these sub-domains will have their own specific model & scope (bounded context).
* Though these patterns may sound feasible to us, but they are not feasible for big monolithic applications. This is because of the fact that identifying sub – domains & business capabilities is not an easy task for big applications. So, the only way to decompose Big monolithic applications is by following the vine pattern or the Strangler pattern.

**11. Strangler Pattern or Vine Pattern**

* The Strangler pattern or the Vine pattern is based on the analogy to a vine which basically strangles a tree that it’s wrapped around.
* So, when this pattern is applied on the web applications, a call goes back & forth for each URI call & the services are broken down into different domains. These domains will be hosted as separate services.
* According to the Strangler pattern, 2 separate applications will live side by side in the same URI space & one domain will be taken into account at an instance of time. So, eventually, the new refactored application wraps around or strangles / replaces the original application until we can shut down the monolithic application.

