Micro services

Two types of architecture

1. Monolithic architecture
2. SOA, service oriented architecture

1. Monolithic Architecture:

1. The original architecture
2. All software components are executed in a single process
3. No distribution of any kind
4. Strong coupling b/w all classes
5. Usually implemented as silo

↓

means monolithic application is standalone and would not share anything with others.

↓

do not expose or share data & functionality

- Easier to design
- Performance – no serialization & deserialization

Service oriented architecture SOA:-

. service expose metadata to declare their functionaly

. Usually implemented using SOAP & WSDL

. usually implemented with ESB.

↓

Enterprise service Base

SOA pros:

1. sharing Data & Functionality.

2. Polyglot Betwee Services.

↓

avoid platform dependency

problems with monolith and SOA:

problems relevant to technology, deployment, cost and mode

Single Technology platforms:

1. with monolith all the components must be developed u
the same development platform.

2. Not always the best for the task.

3. = can't use specific platform for specific feature

4. = Future upgrade is a problem - need to upgrade

Inflexible deployment:

- With monolith, new deployment is always for the whole app
- No way to deploy only part of the app.
- Even when updating only one component - the whole codebase is deployed.
- Forces rigorous testing for every deployment
- Forces long development cycles

Inefficient compute resources:

- With monolith, compute resources (CPU and RAM) are divided across all components.
- If a specific component needs more sources - no way to do that
- Very inefficient

Large and Complex:

- With monolith, the codebase is large and complex.
- Every little change can affect other components
- Testing not always detects all the bugs.
- difficult to maintain

Complicated and Expensive ESB :-

With SOA, the ESB is one of the main components
Can quickly become bloated and expensive
Tries to do everything.
Very difficult to maintain.

Lack of tooling :-

For SOA to be effective, short development cycles were needed.
Allow for quick testing and deployment.
No tooling existed to support this.
No time saving was achieved.

Microservice architecture :-

problems with monolith and SOA led to a new paradigm

Characteristics of microservices :-

1. Componentization via services
2. organized Around Business Capability
3. Products not projects

5. Decentralized Governance.

6. Decentralized Data management.

7. Infrastructure automation.

8. Design for failure.

9. Evolutionary design.

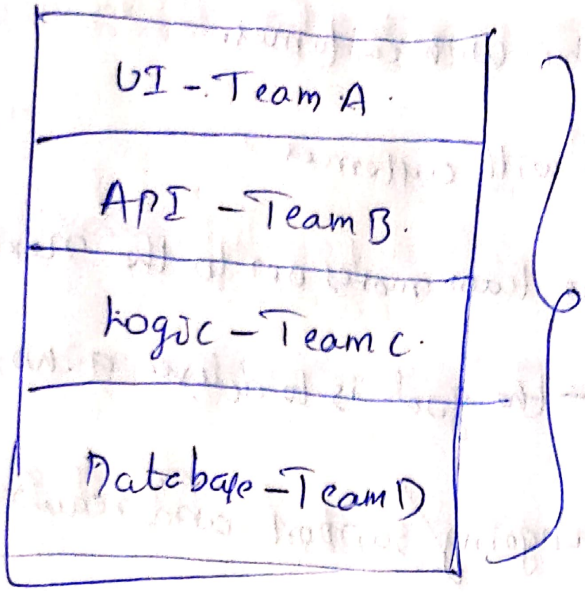
Componentization via Services:

1. Modularity can be achieved using:
   - Libraries - called directly within the process.
   - Services - called by out-of-process mechanism (web, API, RPC)
   - In microservices we prefer using services for the componentization.
   - Libraries can be used inside the service.

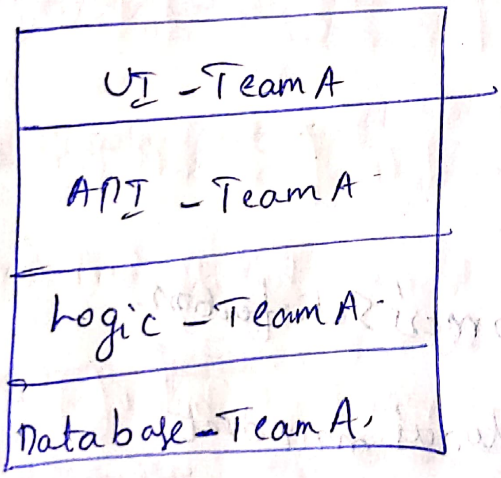
2. Motivation:
   - Independent deployment
   - Well defined Interface

Organized around business Capabilities:

Traditional projects have teams with horizontal responsibilities - UI, API, Logic, DB etc.



with microservice every service handles a well-defined business capability.



Motivation:

- Quick development
- well-defined boundaries

Products not projects:

With traditional projects, the goal is to deliver a working

- No lasting relationship with customers
- Often no acquaintance with customer.
- After delivering - the team moves on to the next project
- with microservices - the goal is to deliver a working product
- A product needs ongoing support and requires close relationship with customer.
- The team is responsible for the microservice after the delivery too.

- Motivation:
  - Increase customer's satisfaction
  - change developer's mindset.

Smart Endpoints and Dumb pipes:

- Traditional SOA projects used two complicated mechanisms
  - ESB
  - WS-* protocol.
    ↓
    extension of original soap protocol

OS - * help to locate services in internet

OS security.

messages delivered

, made inter-service communication complicated and difficult to maintain.

, microservices systems use "dumb pipes" - simple protocols

. strive to use what the web already offers

. Usually - REST API, the simplest API in existence

Each service expose Rest API.

Note:

. direct connection between services is not a good idea

. Better use discovery service or a gateway

Motivation:

. Accelerate development

. Make the app easier to maintain-

Decentralized governance:

With microservices each team makes its own decisions.

which dev platform to use

- Each team is fully responsible for its service
- You build it run it
- Multi env platform called polyglot

- Motivation:

- Enables making the optimal technological decisions for the specific service.

Decentralized Data management:

- Traditional systems have a single database.
- Stores all the system's data from all the components
- With microservices each service has its own database.

note: most controversial attribute of microservices.

- Raises problems such as distributed transactions, data duplication.

Motivations:

- Right tool for the right task - having the right database is important.
- Encourages isolation.

Infrastructure automation:-

- The SOA paradigm suffered from lack of tooling
- Tooling greatly helps in deployment using:
  - Automated Testing
  - Automated Deployment.
- For microservices automation is essential.
- Short deployment cycles are a must.
- Can't be done manually.

Motivation:

- Short deployment cycles.

Design for failure:

- With microservices there are a lot of processes and a lot of network traffic.
- A lot can go wrong.
- The code must assume failure can happen and handle it gracefully
- Extensive logging and monitoring should be in

- Catch the Exception.
- Retry
- log the Exception

- Motivation:

  - Increase systems reliability

Evolutionary design:

- The move to microservices should be gradual
- No need to break everything apart.
- Start small and upgrade each part separately

Problems solved by microservices:

- With monolith all components must be developed using the same development platform.
- Not always the best for task
- Can't use specific platform for specific features

- With microservices, the decentralized Governance attribute solved it.

Inflexible Deployment:

With microservices, the Componentization via services attribute solves it.

Inefficient compute Resources,

the componentization via Services attribute solves it

Large and complex:

with microservices, the componentization via services attribute Solves it

Complicated and Expensive ESB:

With microservices, the Smart Endpoint and Dumb pipes attribute Solves it.

- Application gateway & Discovery
- other APIs : Graphql, grpc

Lack of Tooling :

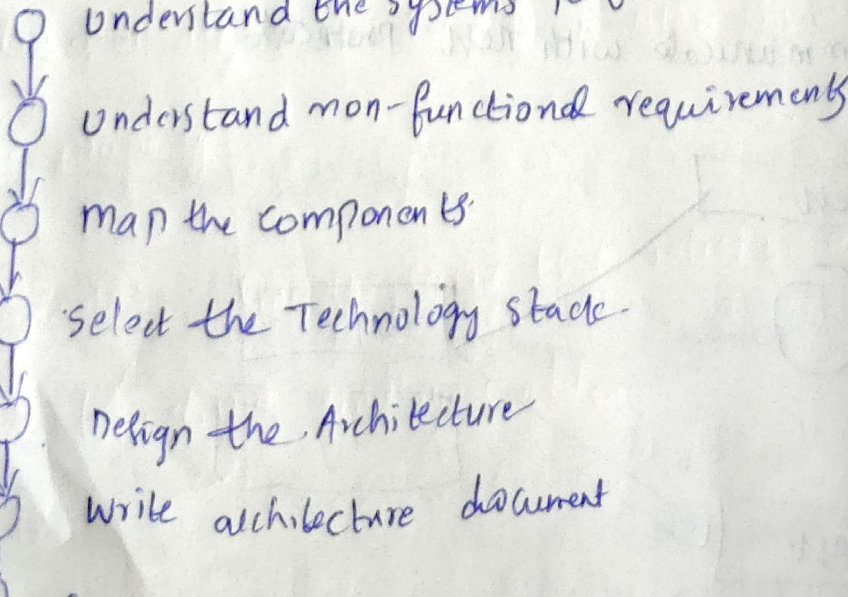
With microservices, the infrastructure automation attribute Solves it.

- Automates testing and deployment
- provides short deployment cycles.

Designing microservices architecture :

1. Designing microservices architecture should be methodical
2. Do not rush in to development
3. "plan more, code less".
4. Critical to the success of the System.

The architecture process :



Map the components.

Split to

- mapping
- communication patterns

Mapping the components:

- The single most important step in the whole process.
- determines how the system will look like - in the long run.
- once set - not easy to change.

- defining the various components of the system

  Components = services.

- mapping should be based on :
  - Business requirements
  - Functional autonomy
  - Data entities
  - Data autonomy

Business requirements:

- The collection of requirements around a specific business capability.
- For example : orders management.

• Functional autonomy:

The maximum functionality that does not involve other business requirement.

For example:

- Retrieve the orders made in the last week ✓
- Get all the orders made by users aged 34–45 ✗

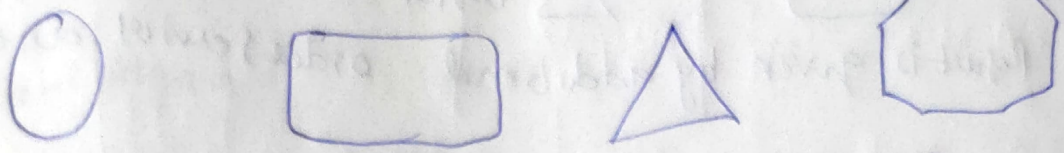
• Data Entities:

- Service is designed around well-specified data entities.
- For example: orders, items.
- Data can be related to other entities but just by ID
  - Example: order stores the customer ID.

• Data autonomy:

- Underlying data is an atomic unit
- Service does not depend on data from other Service to function properly
- For example: Employee Servie that relies on addresses Service to return employees data

ecommerce system

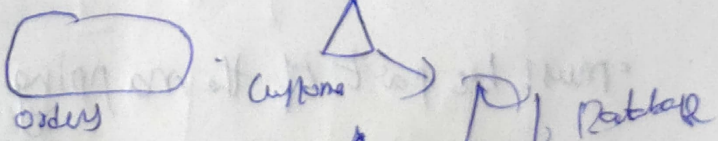


Inventory Orders Customers Payments

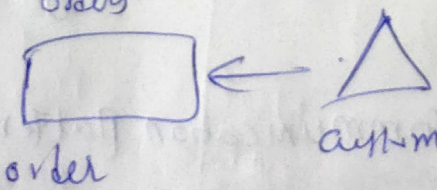
Business requirements	manage Inventory	manage orders	manage customers	Perform Payments
Functional	Add, remove update, quantity	Add, Cancel, Calculate sum	Add, update remove, get account details	Perform Payments
Data Entities	Items	orders, shipping address	customer, address, Cont act details	Payment history
Data Autonomy	None	Related to Items By Id, Related to Customer by ID	Related to orders by ID	None

Three approaches:

1. Data duplication

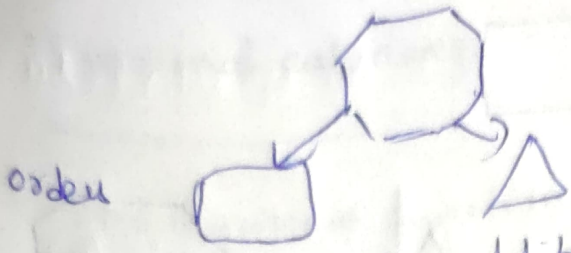


2. service Query



Communicate with each other

aggregate the result of the query



Result is given by additional order services and custom ers

no contact with each other

Edge case: #2

- Retrieve a list of all the orders in the system
- services are not designed for this scenario
- ~~Find out~~ to what the purpose of this query
- Report Engine is the preferred mechanism for this

cross-cutting services:

Services that provide system-wide utility

Common Examples:

- Logging
- Caching
- User management

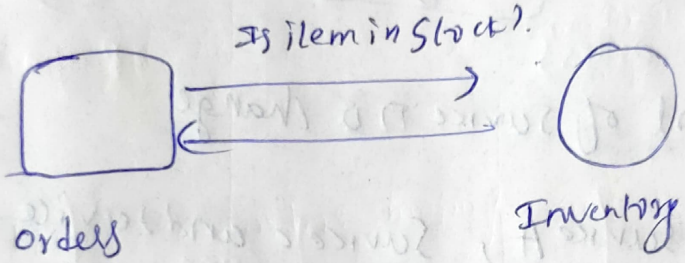
- must be part of the mapping

Defining Communication Patterns:

- Its important to choose the correct Communication

- Main patterns:
  - 1-to-1 Sync
  - 1-to-1 Async
  - pub-sub / Event Driven.

- A service calls another service and waits for the response.
- Used mainly when the first service needs the response to continue processing



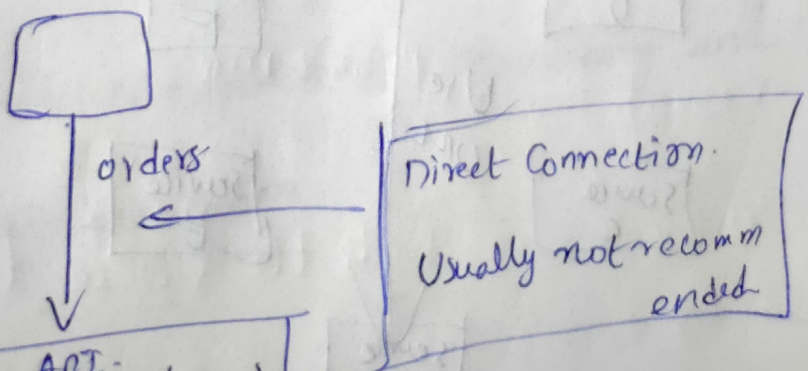
Pros

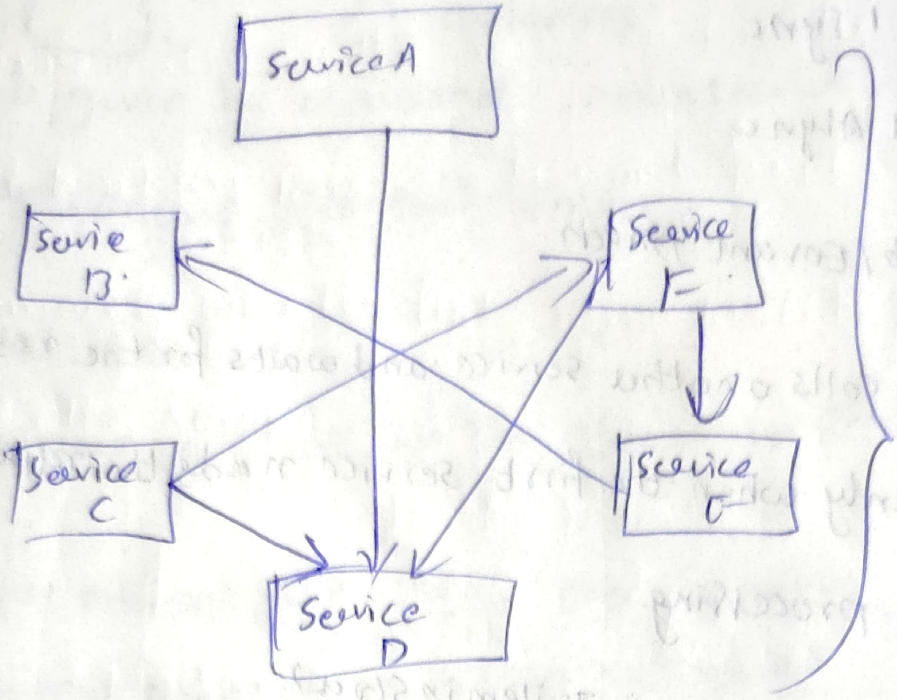
- Immediate response
- Error Handling
- Easy to implement

Cons

- performance.

1-to-1-Sync implementation:



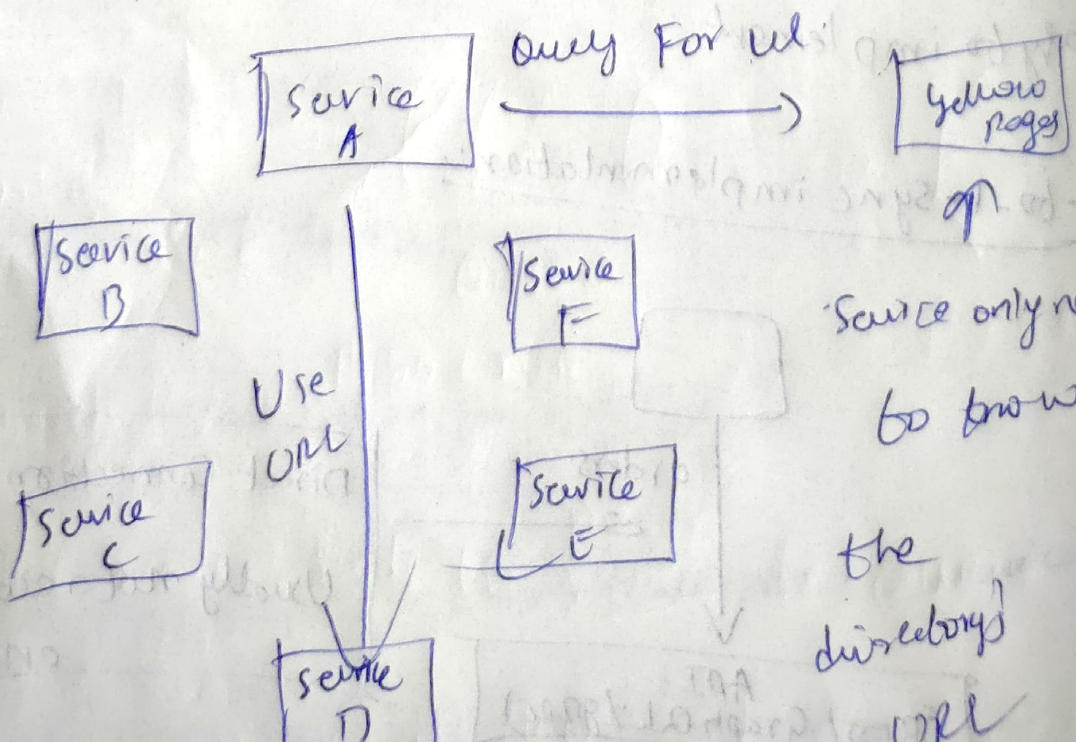


If the url of service 'D' is change the service A, service c and service F got affected, so, direct connection is not recommended.

Using two approaches:-

①

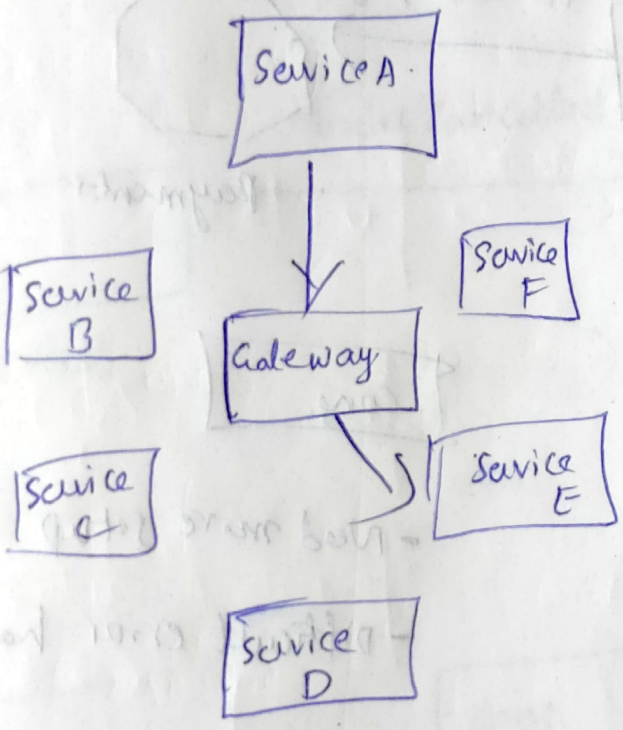
Service discovery



when a service wants to access another service it will goes to yellow pages and ask for url of other service.

Consul tool, configuration & monitoring

Gateway:



Services only need to know the Gateway's URL.

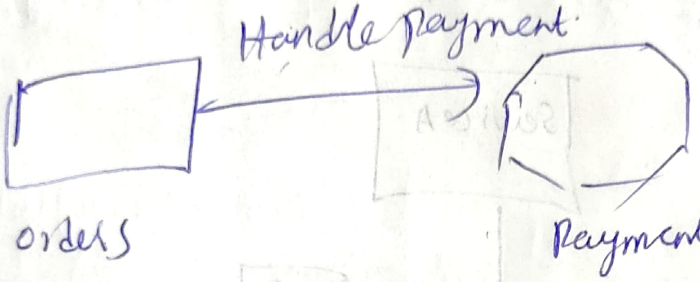
to access a service,

services goes to gateway and gateway goes to required service.

gateway provide configuration, monitoring and authentication

1-to-1 Async:

- A service calls another service and continues working
- Doesn't wait for Response - Fire and forget
- Used mainly when the first service wants to pass a message to the other service.



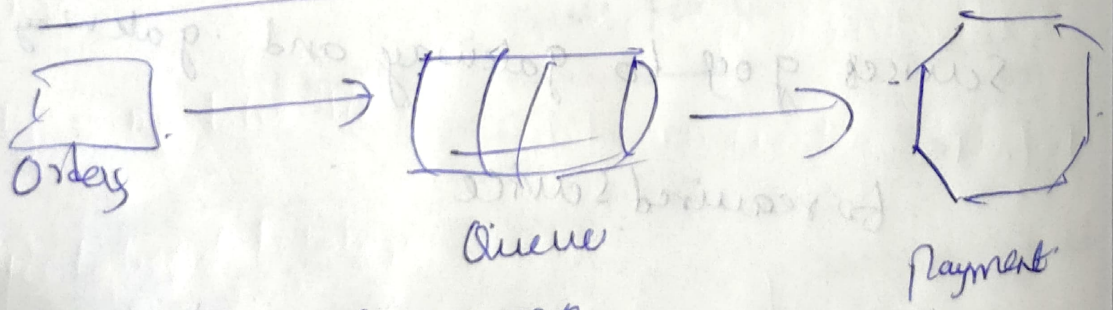
Pros

- Performance.

Cons.

- Need more setup
- Difficult error handling

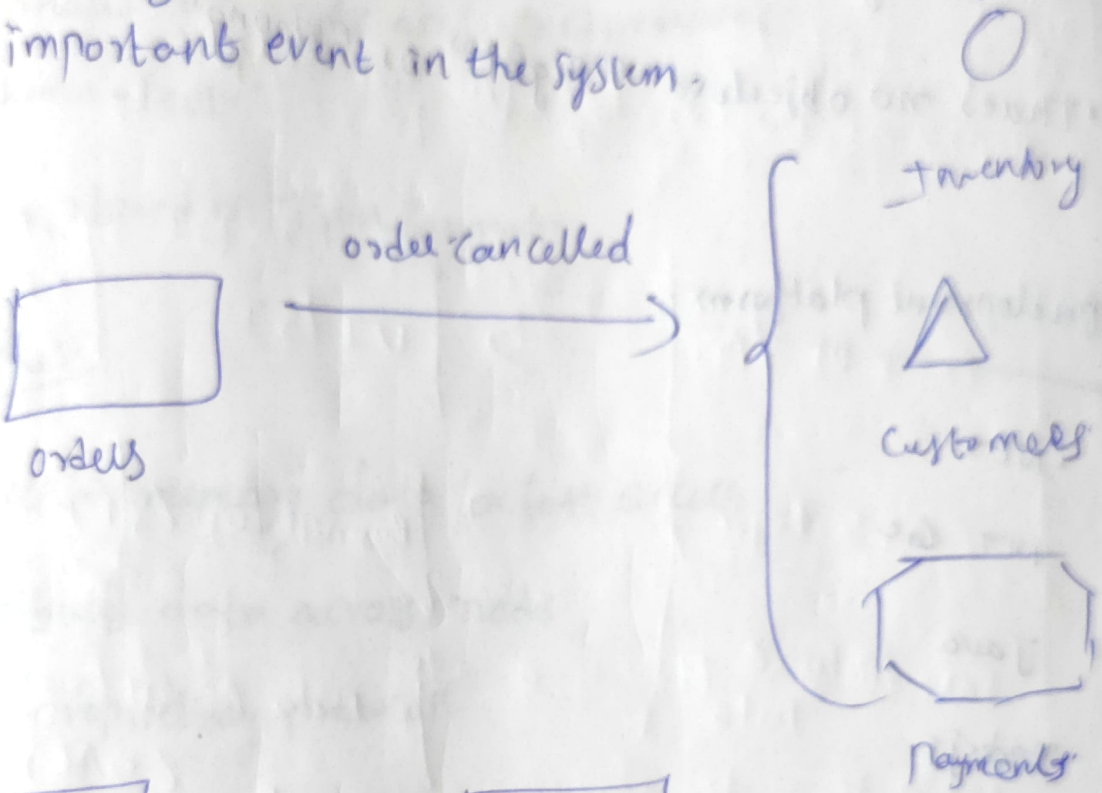
1-to-1 Async implementation:-



Rabbit MQ.

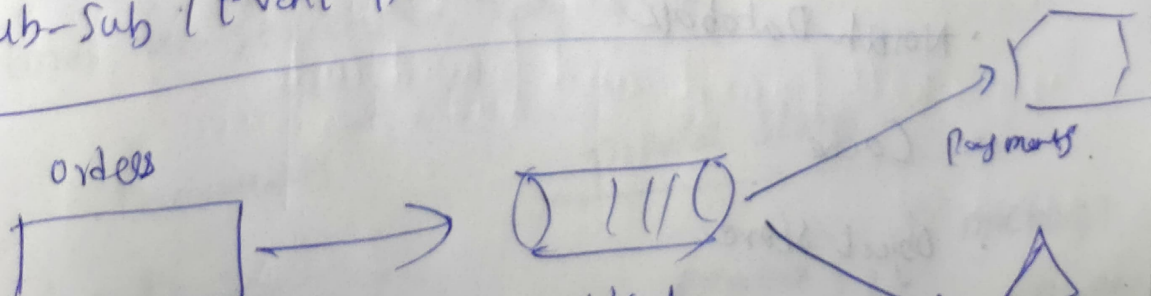
Orders Service sends the message to Queue

A savice wants to notify other savices about something
The service has no idea how many services listen
Doesn't wait for response - Fire and forget
used mainly when the first service wants to notify about
an important event in the system.



Pros	Cons
- performance	- Needs more setup
- Notify multiple savices at once	- Difficult error handling
	- might cause load

Pub-Sub / Event Driven Implementation.



- choosing correct communication pattern is crucial.

Selecting Technology Stacks

- The Decentralized governance allows selecting different technology stack for each service.
- There's no objective "Right" or "Wrong".

Development platforms:

- .NET
- .NET Core
- Java
- node.js
- PHP
- Python

Data store:

4 types of data store

- Relational Database
- NoSQL Database
- Cache
- store

Relational Database:

- Stores data in tables.
- Tables have concrete set of columns.

NoSQL Database:

- Emphasis on scale and performance.
- Schema-less.
- Data stored in JSON format

Caches:

- Stores in-memory data for fast access.
- Distributes data across nodes.
- Uses proprietary protocol.
- Stores serializable objects.

Popular Cache:

redis.

Object store:

- Stores un-structured, large data

- Documents
- Photos

Popular Store:

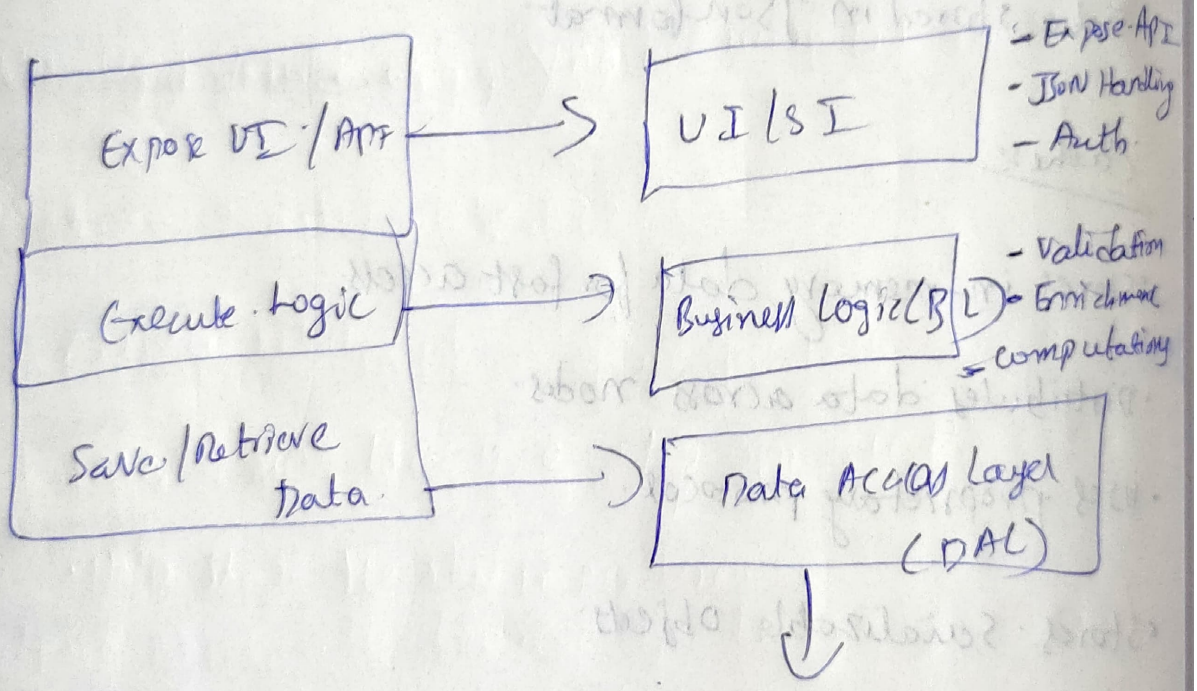
Amazon S3, microsoft azure

Design the architecture

- Service architecture is no different from regular s/w
- Based on layers paradigm

Layers:

Represent horizontal functionality

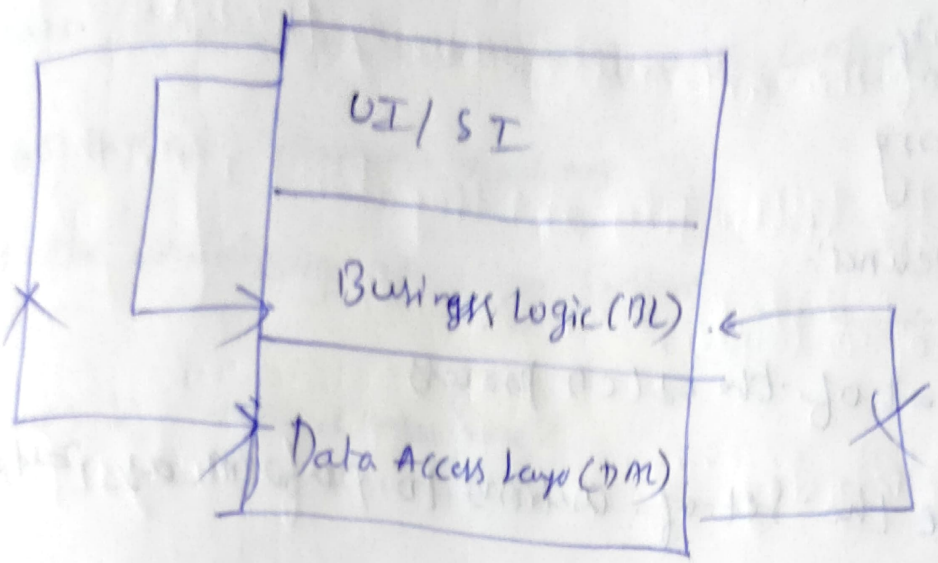


Purpose of Layers:

- Forces well formed and focused code
- Modular

Concepts of Layers:

.Code flow.

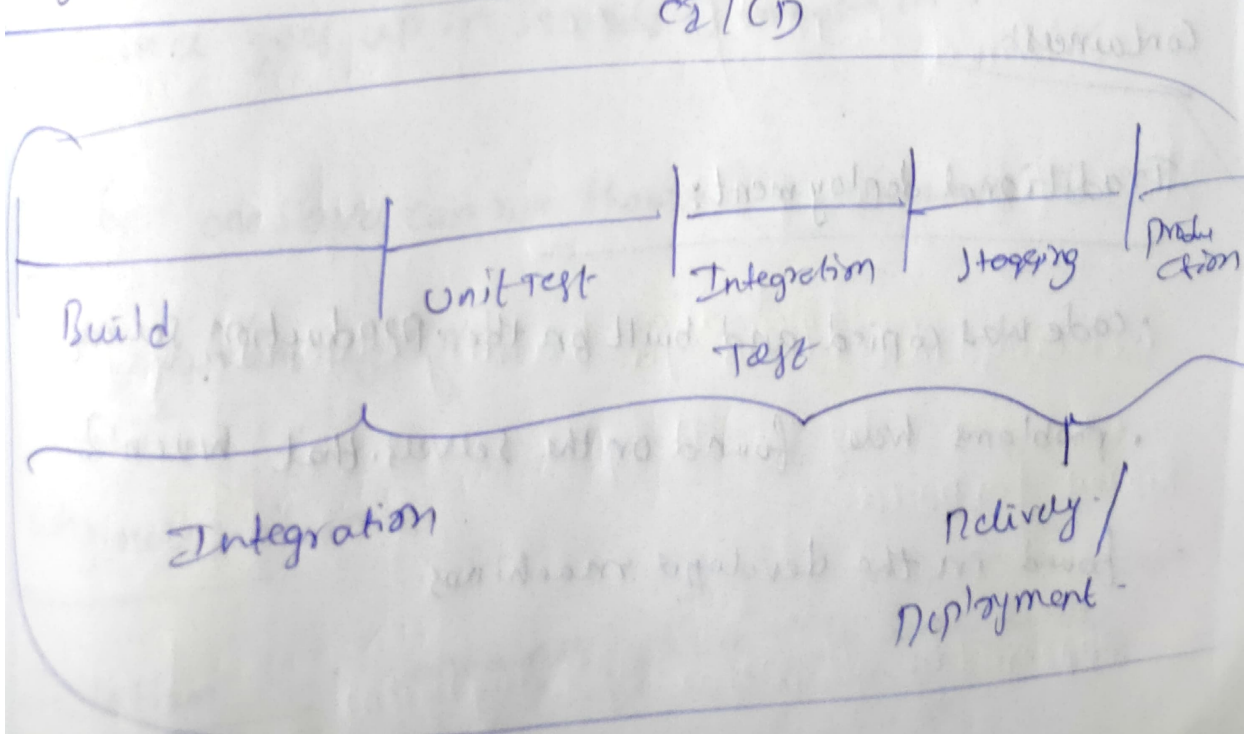


CI/CD:

Stands for Continuous Integration / Continuous Delivery

.The full automation of integration and delivery stages.

Integration & Delivery:



Why we CI/CD

- Faster release cycle
- Reliability
- Reporting

CI/CD pipelines:

The heart of the CI/CD process.

- Define the set of actions to perform as part of CI/CD
- Usually defined using YAML, with UI representation

CI/CD:

As the architect:

- make sure there is a CI/CD engine in place
- Help shape the steps in pipeline

Containers:

Traditional deployment:-

- code was copied and built on the production server
- problems were found on the servers that weren't found in the developer machines.

Containers

- Thin packaging model.
- packages software, its dependencies, and configuration files
- can be copied between machines
- uses the underlying operating system

Containers Vs Virtual machine:

Hypervisor is responsible for running VM.

Why containers:

The same package is deployed from the dev machine to the test to production is called predictability.

Performances:

Container goes up in seconds Vs minutes in VM.

Density: one server can run thousands of containers Vs dozens of VMs

Why not containers:

Isolation: Containers share the same OS so isolation is lighter than VM

Docker:

The most popular container environment

Images of the containers, containing the software to run

Images are static files and they don't run. They are basis of containers.

Container Registry: → A collection of images from where the image we can pull.

Container registry can be public.

Containers are instances of images that can be build and run. Container managed by docker daemon.

client is a CLI to manage the images and Containers.

dockerfile:

containing instructions for building custom images.

Support for Docker:

Supported by all major operating system

Containers Management.

- Containers are great deployment mechanism when there are many containers.

1. Deployment
2. Scalability: Load more effectively.
3. monitoring.
4. Routing: similar to load balancer.
5. High-availability

The most popular container management platform

Kubernetes.

It is De-facto standard for Container management

- kurbenetes provides all aspects of management.
  - Routing
  - Scaling
  - High availability
  - Automated Deployment
  - Configuration Deployment

Testing microservices:-

Test types:-

- Unit test
- Integration tests
- End-to-End tests

challenges with microsevices -testing:-

- Testing and tracing all the services is not trivial
  - Testing state across services
  - Non-functional dependent services

Unit Tests:-

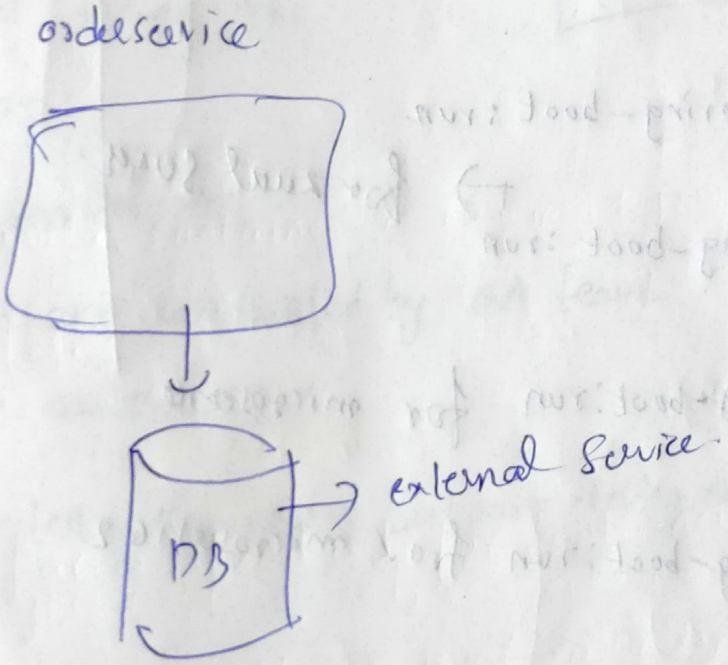
- Test individual code units
  - Method, interface etc
- In process only
- Usually automated
- Developed by the developers

Unit Tests in the microservices:-

- No different
- Test only in-process mode.
- Use the same framework and methodologies

Integration Tests:-

- Test the service's functionality
- covers all code paths in the service



Test Double:-

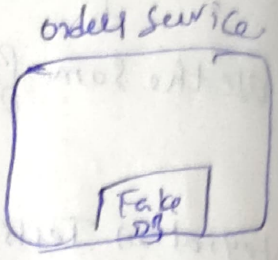
- pretends to be the real object / service to allow testing

Three types:

1. Fake
2. Stub

Fakes:-

- Implements a shortcut to the external service
- For Example - Stores data in-memory
- many times implemented in-process
- Requires code change in the code



Instead of having external DB have DB - in memory

DB

In cmd:

mvn spring-boot:run

↳ for zuul server

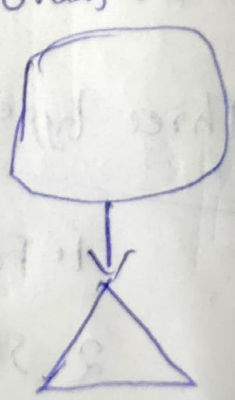
mvn spring-boot:run

mvn spring-boot:run for microservice

mvn-spring-boot:run for microservice2

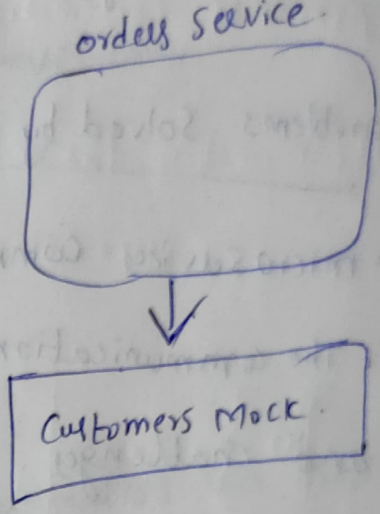
Stubs:

- Is a piece of code holds hard-coded data
- usually replaces data stored in a DB.
- Allows simulating data services quickly.
- No code change required.



Mock

- we dont provide fake data using fake & Stub.
- with mock we verify access was made
- But out test check service calls external service.
- Holds no data.
- No code change required.



Integration testing are implemented using the service's API.

- Use the service's API.
- Developed and conducted by QA team
- Should be automated.
- Most unit testing framework support integration test

End-to-End tests:-

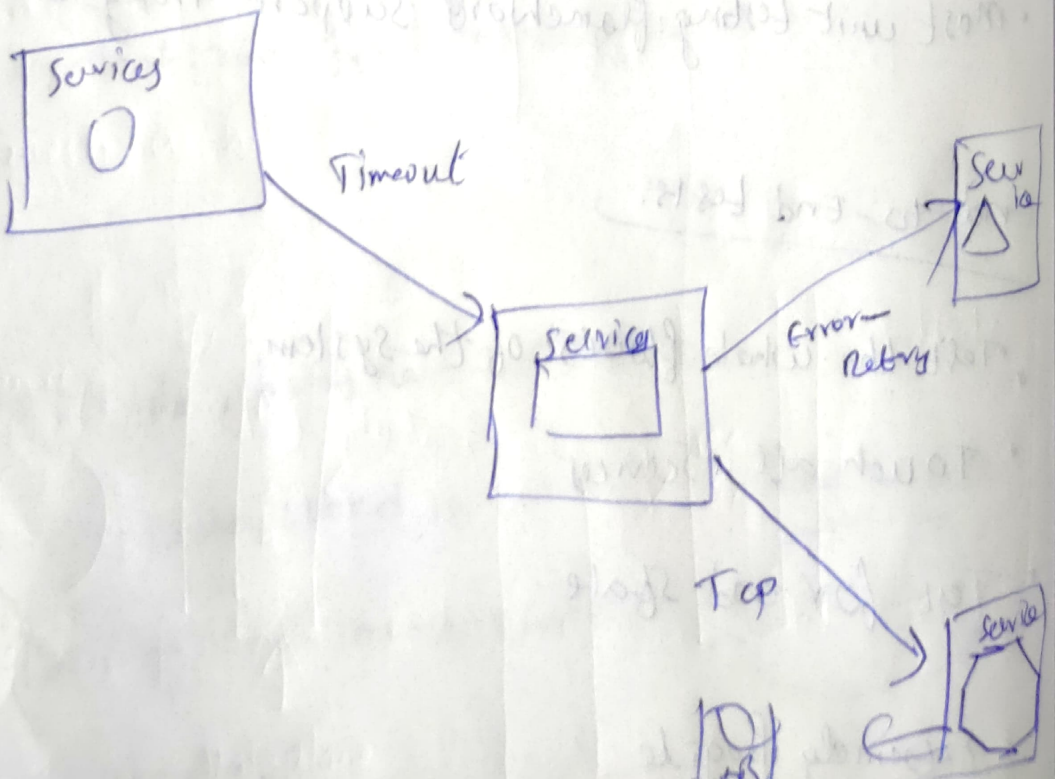
- Test the whole flow(s) of the system.
- Touch all services
- Test for end state.
- Extremely fragile
- Require code. usually used for main scenarios only

Service mesh:-

- Manages all service-to-service communication
- provides additional services.
- Platform agnostic (usually).

Problems solved by service mesh:

- microservices communicate between them a lot
- The communication might cause a lot of problems and challenges
  - Timeouts
  - Security
  - Retries
  - monitoring.



In-process vs sidecar:

In-process

- performance

Sidecar

- platform agnostic
- code agnostic

↓

Code is not changed

↑

more popular

Products and implementations:

- Some inprocess, most Sidecar
- most free, some don't.
- No not develop your own

Sidecar:

Istio, Linkerd, nginx → ~~tools~~ mesh

In-process:

DDS foundation inprocess mesh.

when should you use service mesh?

only if

- you have a lot of services
- which communicate a lot with each other
- or you have a complex communication requirement with various protocols or brittle networks

Logging & monitoring:-

- Important in microservices
- Flow goes through multiple processes.
- Hard to get wholistic view.

Logging vs monitoring:

1. Logging is used for analyzing the systems' behaviour and analysing roles. Recording the system's activity
2. Using logging we can trace user behaviour: Audit
3. Logging is used for documenting error.

timestamp, error exception.

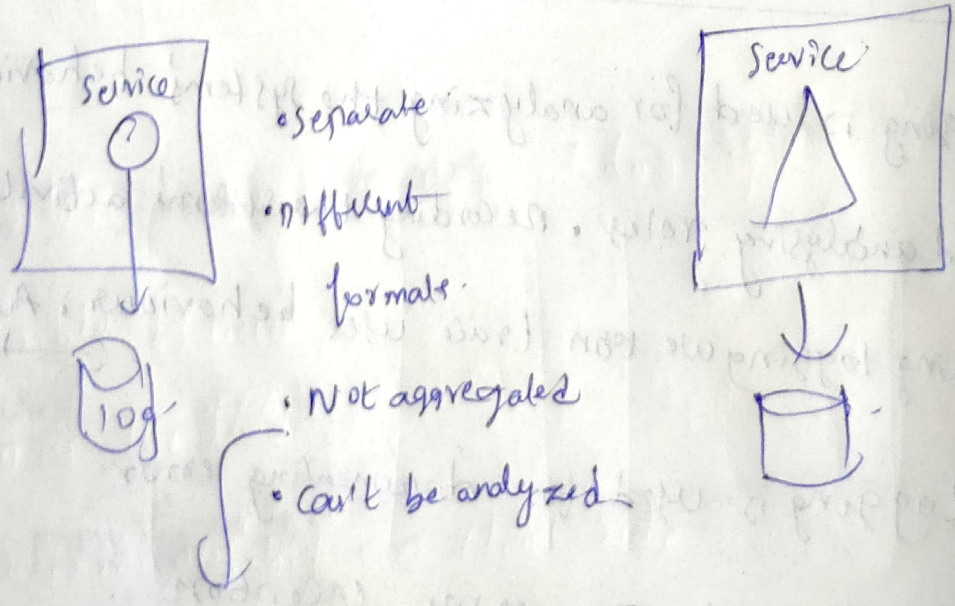
Monitoring:

- Based on system's metrics → Infrastructure related metric
  - such as CPU, RAM etc
- application related metrics, request per minute orders per day
- Altering when need

Implement Logging:

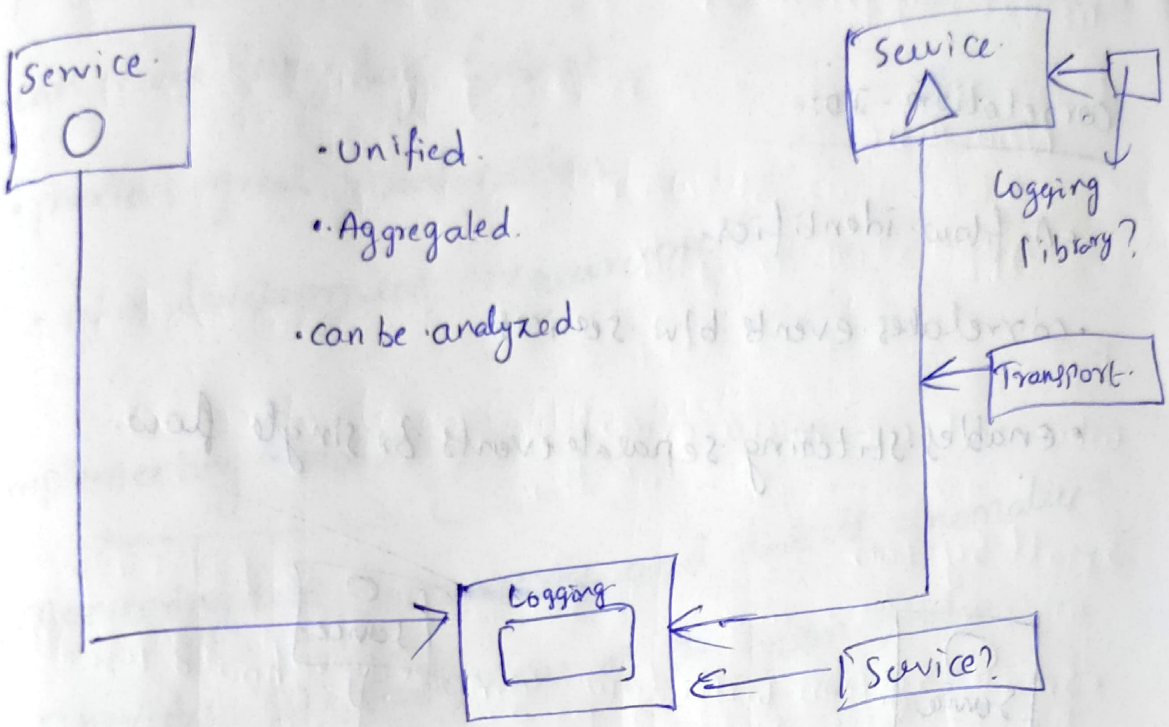
- logging should provide wholistic view on the system
- should allow tracing end-to-end flow.
- should contain as much information as possible.
- can be filtered using severity, module, time etc

Traditional logging:



calculations cannot run

microsavices



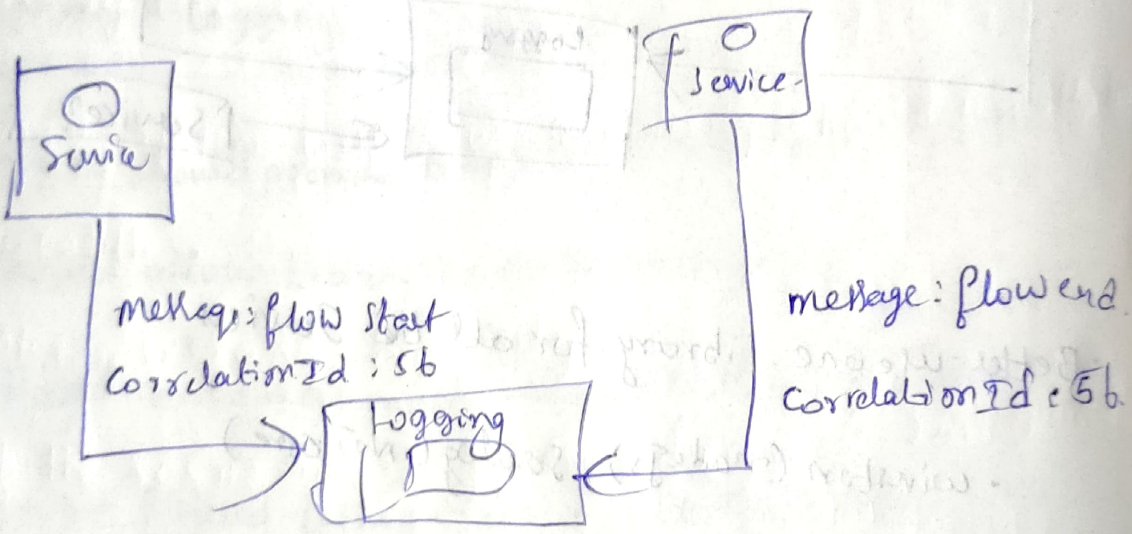
- Unified.
- Aggregated.
- can be analyzed.

- Better use one library for all the services
  - winston (nodeJs), Serilog (.Net core)
- If using various platforms - one library for each platform.
- Use severity wisely
- log as much info as possible, at least
  - Time stamp - exact time happened
  - User - performing action
  - severity - of event
  - service - event happend.

- Stack Trace (if error)
- correlation ID

Correlation-ID:-

- A flow identifier
- correlates events b/w services
- Enables stitching separate events to single flow.



Same flow { 5b, flow start → Central logging.
5b flow end.

Transport to service to Central logging

- preferably Queue
- Balances the load.
- No performance hit on client side

Logging service:

- preferably based on indexing / digesting / search product
- can index any log format
- provides great visualization.
- No development requirement

Implementing monitoring:

- monitoring looks at metrics and detects anomalies
- provides simplied view of the system status
- Alerts when there is a problem.

Types of monitoring:

- Infrastructure
  - monitors the server
  - CPU / RAM / Disk / Network
  - Alerts when infrastructure problem is detected
  - Data source: agent on the machine

- Application
  - monitors the application
  - Looks at metric published by app
  - Alerts when application is detected
  - Data source: application logs

Most monitoring products provide both

Monitoring products:

New Relic

Application Insights

When not use microservices

Small systems:

- Small systems with low complexity should usually be a monolith.
- microservices add complexity
- If the service mapping results in 2-3 services - microservices not right option.

Interminged functionality:

- one of the most important microservices attributes is its autonomy.
- when there is no way to separate logic or data - micro services will not help
- If almost all requests for data span more than

performance Sensitive Systems):-

- microservices systems have performance results of the network hops.
- If the system is very performance sensitive - think twice.

Quicks and dirty - systems:-

- microservices design and implementation takes time.
- Usually has a short lifespan.

No planned updates:-

- Some systems have no planned updates.

No updates = No microservices.

Microservices and the organization:-

- microservices require different mindset.
- Traditional organizations will have hard time with microservices.