# Microservices Concepts

## Monolith and Microservices Architectures

### Monolithic Architecture

The monolithic architecture couples and runs all the application’s components (UI, business and DB calls) as a single, unified system. Therefore, any modifications made to one part of the application can potentially impact the entire system.

Advantages: easy to develop, debug, test. scale vertically, easy to deploy as a single jar/war file

Disadvantages: becomes difficult over time to manage and understand, hard to implement new features, a small change means new deployment of whole project, new technologies are problematic to implement, highly tight coupling

#### Patterns & Principles

**DRY** – Don’t Repeat Yourself

Try to maintain the behavior of a functionality of the system in a single piece of code, it should not have duplicated code or design item.

**KISS** – Keep It Simple, Stupid – a simple code is easier to understand, avoid unnecessary complexity

**YAGNI** – You Ain’t Gonna Need It – should not create new features, if it’s not really necessary

### N-Layer Architecture

It is known as n-tier architecture style or multi-layered architecture style. Components of an application is divided into horizontal logical layers. It is still monolithic but components are interconnected but **don’t depend** on each other. **Layers of isolation** that layers can be modified won’t affect other layers.

Components of a Layered Architecture:

* Presentation Layer – responsible for user interactions for example a web app
* Application/Business Layer – handles functional requirements
* Database Layer – responsible for handling data

### Clean Architecture

Separates the elements of a design into circle levels. Keep the business logic and application domain at the center of the solution structure that independent with presentation and data access layers. Clean architecture divided into two main elements:

* **The policies** – the business rules and procedures
* **The details** – the implementation code to carry out the policies

### Microservices Architecture

In the microservices architecture, the application is broken down into small, independent, loosely coupled services that communicate with each other using well-defined APIs. Furthermore, each microservice is responsible for a specific feature or functionality of the application, and each service can be developed, deployed, and scaled independently of the other services. Therefore, allowing for greater flexibility and agility in developing and maintaining the application.

Advantages: scale and deploy independently, handle and process millions of requests, different technology stacks, adding new features immediately without affecting other parts.

## Microservices Communication and Types

### Synchronous Communication

* client directly sends the request to MS and waits for response, using **HTTP** and **gRPC**

**gRPC –** High performance Remote Procedure Call. Uses HTTP/2 protocol to transport binary messages.It works same as FeignClient but not using by http protocol. It uses protobuf (byte array [byte protocol]).

### Asynchronous Communication

* used queues, client sends request and does not wait for response, **AMQP** (Advanced Message Queueing Protocol).The message producer does not wait for a response.

## Patterns

### Integration Patterns

#### API Gateway Pattern

#### Aggregator Pattern

It is a website or program where it collects related data from different MSs and combined the data then return it. The pattern is a service that receives a request, then makes requests to multiple services, combines the results and responses the initial request.

#### Proxy Pattern

A1 service work with other services and it should be updated (new-version) and deployed. Consumers will fail because they don’t get info or updates to work with new version. To create Proxy service and all consumers connect through Proxy service. The service will direct which consumers are working with new and old versions.

#### Gateway Routing Pattern

### Database Patterns

#### Database per Service Pattern

Because of loose coupling services, we can choose the technology stack per service. For instance, we can decide to use a relational database for service A and a NoSQL database for service B. This model lets the service manage domain data independently on a data store that best suites its data types and schema. Further, it also lets the service scale its data stores on demand and insulates it from the failures of other services. Unlike single monolithic applications (where ACID easily applied), distributed applications are dealing with multiple services where handling transactions could be a challenge.

#### Shared Database per Service

#### CQRS Pattern

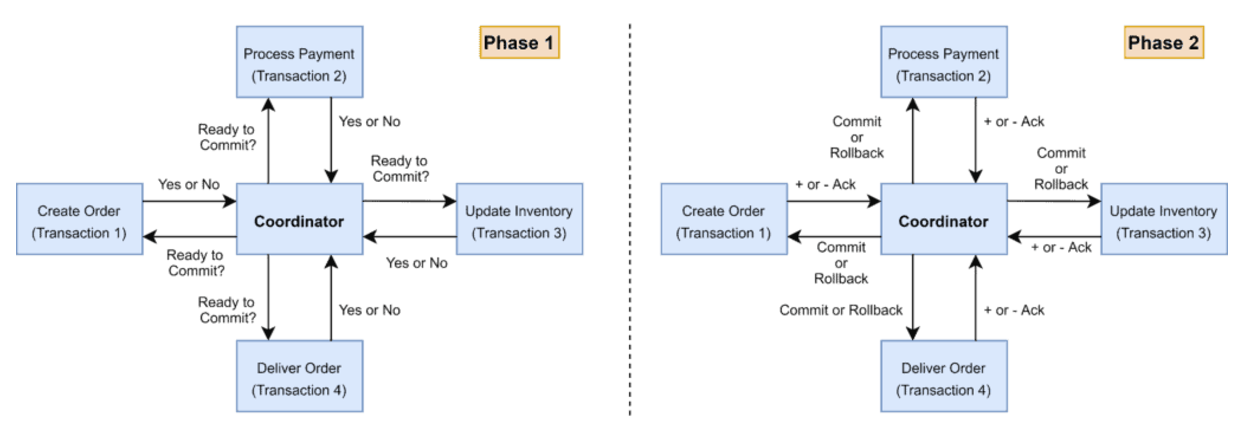
#### Event Sourcing Pattern

#### Saga Pattern

Handling rollback operation in microservice architecture.

##### 2PC

Two phase commit protocol is a widely used pattern to implement distributed transactions.



2PC protocol runs a distributed transaction in two phases:

1. **Prepare Phase** – The coordinator asks the participating nodes whether they are ready to commit the transaction. The participants returned with a yes or no.
2. **Commit Phase** – If all the participating nodes respond yes in phase 1, the coordinator asks all of them to commit. If at least one node returns negative, the coordinator asks all participants to roll back their local transactions.

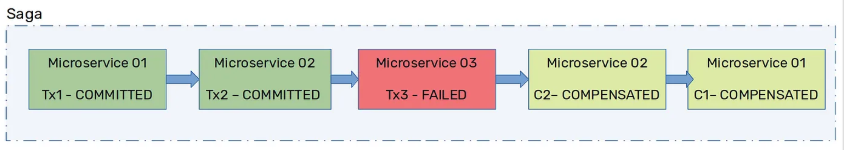
Problems with 2PC:

* Coordinator is the single point of failure
* All other services need to wait until the slowest service finishes its confirmation. So, the overall performance of the transaction is bound by the slowest service.
* Not supported in NoSQL databases and message brokers such as Apache Kafka, RabbitMQ

Not recommended because of its synchronous blocking.

##### Saga Architecture

It provides transaction management using a sequence of local transactions. Every operation that is a part of the Saga can be rolled back by a compensating transaction.



##### SEC

Saga Execution Coordinator is the central component to implement a Saga flow. It contains a Saga log that captures the sequence of events of a distributed transaction. For any failure, SEC component inspects Saga log to identify the impacted components and compensating transactions should run.

##### Implementations

In a choreography pattern, a local transaction publishes event that trigger other participants to execute local transactions. In an orchestrated-saga, a centralized saga orchestrator sends command messages to saga participants telling them to execute local transactions.

###### Choreography Pattern

There is no central coordinator to tell saga participants what to do. Saga participants subscribe to each other’s events and respond accordingly. Each saga participant updates its local database and publishes an event that triggers the next participant.

Interaction type: Publisher / Subscriber Messaging

###### Orchestration Pattern

A single orchestration is responsible for managing the overall transaction status. Camunda is the one of frameworks that this pattern can be applied.

Interaction type: Asynchronous Request / Response

### Observability Patterns

#### Log Aggregation

#### Performance Metrics

#### Distributed Tracing

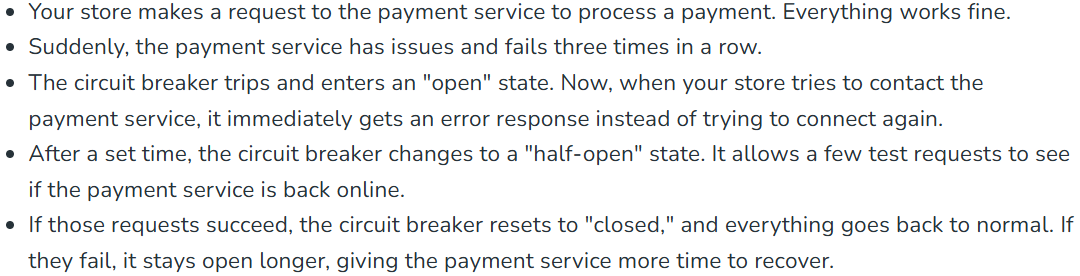
#### Health Check

### Cross-Cutting Concerns Patterns

#### External Configuration

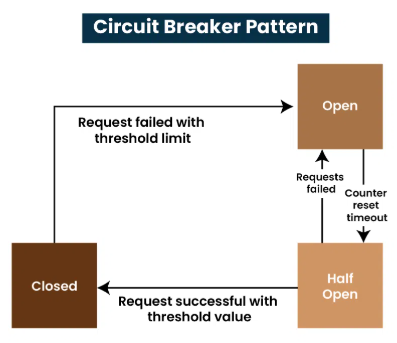
#### Service Discovery Pattern

#### Circuit Breaker Pattern

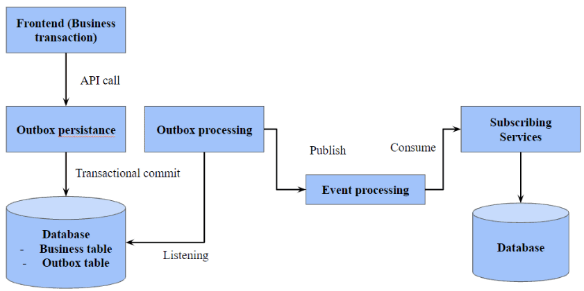


Used to enhance system resilience and fault tolerance. It prevents an app from repeatedly trying to execute an operation that is likely to fail, which can lead to cascading failures across the system. It operates mainly in 3 states:

* **Closed State** – circuit breaker routes requests to MS normally (healthy or up) and counts the number of failures in each period of time. If number of failures exceeds the threshold, a circuit will stop and move to Open state.
* **Open** **State** – signaling potential issues with the downstream service, the circuit breaker transitions to the Open state. Stops forwarding requests to the failing service and returning exception.
* **Half-Open State** – After a specified timeout period (30 s) in the Open state, transitions to Half-Open state. For example, 5 request icaze verilir, eger success dirse hamisi veya 50%i, kecsin Close state’e, else to Open state.

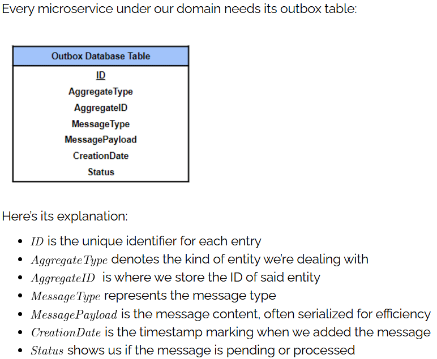


### Transactional OutBox Pattern



When API publishes event messages, it doesn’t directly send them. Instead, the messages are persisted in a database table. After that, A1 job publish events to message broker system in predefined time intervals. Basically the pattern provides to publish events **reliably**. Domain events are not written directly to event bus, instead it is written to a **outbox** table where microservices log their outgoing data changes. This pattern serializes the data, preparing it for transmission to other services.

**When to use**: when working with critical data, and need to catch all requests, making sure that system not losing any event messages. It is a method of maintaining data consistency, useful as the system grows.

**Implementation**: instead immediately sending events, they are serialized first and recorded in primary db with Pending status, then with the help of periodic task, pending tasks are fetched and send to message brokers.

## Extra Notes

### Load Balancer

Round Robin Algorithm - requests are distributed across the group of servers sequentially. Health Checks - microservice tells LB whether it is in good shape to handle request or not

------------------------------------------------------------------------------------------------------------------------

### Rate Limiter

It is a strategy to limit access to APIs. It restricts the number of API calls that a client can make within a certain time frame (4 **RPS** -> means 4 **R**equest **P**er **S**econd). Security meqsedile istifade olunur.

**Replenish rate** – 1 saniyede ms nece requets qebul ede biler, onu user’e gosterir (replenish rate = 5 olsa, it means saniye’de api 5 request handle ede bilir)

**Burst capacity** – maximum number of request that api can handle per second; replenish rate’i kecse meselen max 10 request ata biler user

### Resilience4J

For retry cases.

### CDN

### Sticky session

Load balancer randomly routes each request to one of the servers. However, it can be possible to route request to a specific server, which keeps the data for the requets, using sticky session. When a server knows how a user interacts with a site, it can reuse that session repeatedly, not having to create a brand-new session all over again.

For instance, request102 comes to LB, and with the help of sticky session, LB knows that request102 went to server1 2 minutes ago, and it should go to server1. The problem occurs when the server1 is down, and request102 that should be routed to server1 will fail.

------------------------------------------------------------------------------------------------------------------------

### Service Discovery

services register on Eureka Server and become Eureka clients; services that call each other does not need to remember the ip and ports, they just need to call Eureka server indicating opposite server name in order to contact each other.

Eureka (Spring Cloud using) provided by Neftlix OSS, Ribbon, Hysterix, Zuul

Why we use **@LoadBalanced** over RestTemplate Bean? - it does discovering service. it tells RestTemplate do not go directly whatever the url I gave you, it is not actual url, it is a hint about what service you need to discover. Because we use the name of services in url paths (as ip) not the url itself. Additionally, it also balances the load if there are more than one instance of the same service.

Discovery service -> **euroku server** : stores ip and ports to make communication easy between microservices

### Scalability

Scalability is the number of requests an application can handle. If no longer handle any more requests, it has reached its scalability limit. To prevent downtime and reduce latency, application must scale.

**Vertical scaling** by adding more power, hardware (RAM, CPU, DISK). It has scalability limits because of the capacity of the single machine.

**Horizontal scaling** by adding more machines to split the load.

### Caching

Stores frequently accessed data to reduce load on database and speed up response times.

Client-side

Server-side

Startegies: write-through, write-back, write-around

Distributed cache: redis, memcached

Cache eviction policies: LRU, LFU

Drawbacks: cache invalidation – ensuring cache remains up-to-date with the most recent data.

### Zipkin

Used in implementing distributed tracing pattern. Microservice’e gelen requets details trace ede bilirik.

# Database Concepts

## Reliable, Scalable and Maintainable Applicatoins

**Reliability** – the system should work correctly even there are software, hardware or human faults.

**Scalability** – as the system grows (in data, traffic volume or complexity), continuing to work correctly.

**Maintainability** –

## ACID

**A (**Atomicity**) –** All or nothing rule.A transaction must be Atomic, meaning all changes made by the transaction are completed as a single unit, or none of the changes are made. If a partial transaction were committed, the atomic property is violated, and the database is left in an inconsistent state. All the transactions(insert, update, delete) inside a transaction are either completed or rolled back. The ability to commit or roll back transactions is required for Atomicity. For example, in an application that transfers funds from one account to another, the atomicity property ensures that, if a debit is made successfully from one account, the corresponding credit is made to the other account.

**C (**Consistency**) –** the transaction must preserve database Consistency, which means that the database must begin the transaction in a state of consistency and return to a state of consistency when the transaction is complete. Consider the following example. Assume a foreign key constraint exists between a table named Orders and a table named OrderDetails based on a field named *OrderID*. This constraint ensures that before an OrderID exists in the OrderDetails table, the OrderID must first exist in the Orders table. If a transaction were to fail to write a record to the Orders table but successfully writes the record to the OrderDetails table, the database would no longer be in a consistent state. For example, in an application that transfers funds from one account to another, the consistency property ensures that the total value of funds in both the accounts is the same at the start and end of each transaction.

**I (**Isolation**) –** transactions are individual.Every transaction is individual, multiple transactions occur independently without interference. When multiple users are reading and writing from the same table all at once, isolation of their transactions ensures that the concurrent transactions do not interfere with or affect one another. Each request can occur as though they were occurring one by one, even though they are actually occurring simultaneously.

**D (**Durability**) –** Once the transaction completed, then the changes it has made to the database will be permanent. The changes of a successful transaction will be saved even if the system failure occurs.

To assure the ACID properties of a transaction, any changes made to data in the course of a transaction must be *committed* or *rolled back*.

When a transaction completes normally, a transaction processing system *commits* the changes made to the data; that is, it makes them permanent and visible to other transactions.

When a transaction does not complete normally, the system rolls back the changes; that is, it restores the data to its last consistent state.

Resources that can be rolled back to their state at the start of a transaction are known as recoverable resources: resources that cannot be rolled back are non-recoverable.

## Indexing

Main idea is narrow the search. Database indexing is a data structure technique used to locate and quickly access data in db. But it also takes a storage in the disk and there is also cost in indexing.

Improves read data, but slowes down write operations since the index needs to be updated whenever data is modified.

Indexing methods:

Ordered indices:

Primary Index:

Clustering Index:

Secondary index:

## Partitioning

Server daxilinde meselen high load order table’imiz var ve onu 5 hisseye boluruk key’e gore, meselen ID’ye gore. Order1, …, Order 5. Meselen, 1 – 500 ID’li orderler yerlesir Order1 table’da. Hemcinin 5 index table’imiz olacaq.

Partitions are defined in a such way that each piece of data (each record, row or document) belongs to exactly one partition. The main reason using partition is scalability. Partitioning is usually combined with replication, so that copies of each partition are stored on multiple nodes. This means each record belongs to exactly one partition, it may still be stored on several different nodes for fault tolerance.

How to partition data into nodes – the goal with partitioning is to spread the data and the query load evenly across nodes. If partitioning is unfair, so that some partitions have more data or queries than others, it is called *skewed*. In extreme case, 90% load end up on one node. Partition with extremely high load is called a hot spot.

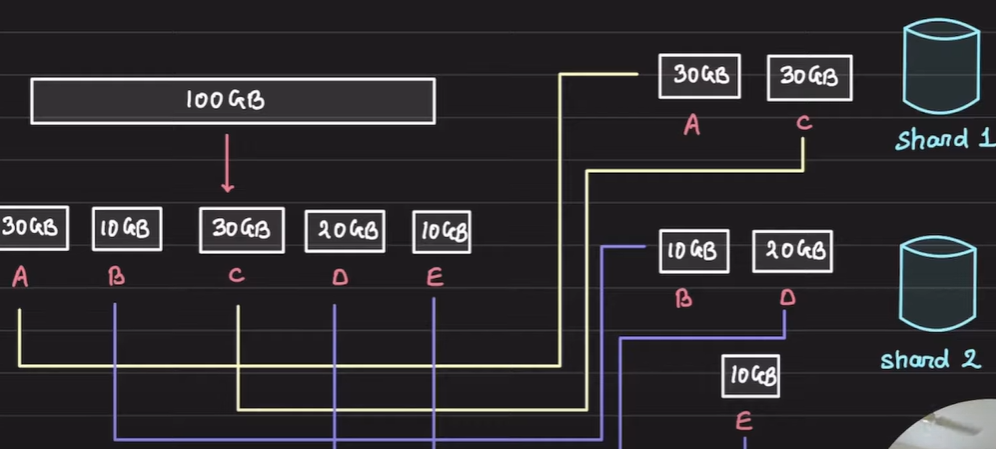
Vertical partitioning

Horizontal partitioning

## Sharding

Sharding is one specific type of partitioning, part of what is called **horizontal partitioning**. Sharding database’in server’ler arasinda more manageable and smaller hisselere bolunmesidi. Each shard is separate database (server) that contains subset of the data.

Partitioning is about splitting data, sharding is about splitting database. Partitions A and C is in Shard 1.



Read Replica Case: there is sharding but not partitioning. 2 database servers, the same data in each server.

Higher write input case: sharding and partitioning, both. 2 servers and data is splitted to balance load.

Advantages: handle large reads and writes; higher availability;

Disadvantages: cross-shards queries expensive; **JOINS** -> because query needs to go different shards, pull out and join the data across the network

## Replication

Multiple replicas of the same database on different servers for scaling the reads and distributing load.

Reasons to replicate data:

* To keep data geographically close to users (reduce latency)
* A part of system fails, continue to work (increase availability)
* To scale out number of machines that can serve read queries (increase read throughput)

**Leaders and Followers**

Master-Slave approach: Esas Master databazamiz (Leader) olur ve n sayda slave (back-up or followers)’lari olur. Read’ler backup’lara gedir, write’lar master’e. Leader send write operations through replication\_ log to the followers.

Follower fail olsa, each follower stores log of data changes, in its local disk, it has receieved from the leader. and it knows where it interrupted before failure. It can easily recover itself form that point.

If leader fails, one of the follower promoted to be the new leader (the most up-to-date one).

**Replication Log** – the leader logs every write request (insert, update, delete) that it executes, and sends that statement log to its followers.

**Synchronous Replication**: data is copied to replica servers simultaneously ensuring consistency. In case of leader fails, we ensure that data is still available on the follower. But, there is a latency as primary server waits for all replicas to confirm write operations. If synchronous follower does not respond, it means whole system fails. In practise, enabling synchronous replication on database, it usually means one of followers is synchronous, others are asynchronous. If synchronous follower becomes unavailable, one of asynchronous follower becomes synchronous which guarantees we have up-to-date 2 nodes: leader and synchronous nodes. It is called semi–synchronous.

**Asynchronous Replication**: primary server does not wait for replicas to conform the write, which improves performance but may lead to temporary inconsistencies.

## Consistency

Models:

Casual (Weak) consistency

Strong consistency

Eventual Consistency

## NoSQL vs RDBMS

Know when to use which database:

when we need ACID, choose SQL. when we don’t know data type, go with NoSQL.

CAP Theorem

Writes in no-sql is faster (look LSM tree).

Why reading is faster than writing in an indexed table? - Any kind of index usually slows down writes, because the index also needs to be updated every time data is written.

**SQL –** primary & foreign keys, having constraints, indexing which results in faster searching and sorting. Scalability is high but expensive.

**NoSQL –** key & value pairs.

Scaling is horizontal in which the load are distributed on all nodes. Partition key should be unique and well-distributed. For instance, it cannot be male or female. It goes to hashing function and the result will be one of the machines (selected randomly) where data stores.

All data are inserted in one insert. All relevant data is contained together as one block. So, no need for joining tables which is expensive operation.

Database types: key-value pair, column-oriented, graph-based and document-oriented (JSON).

**------------------------------------------------------------------------------------------------------------------------**

**Primary key** ensures unique row identification. This results in faster sorting, searching, and querying operations. It must obey the Unique and NOT NULL database constraints. **Foreign key** creates a link between two tables. It maintains referential integrity between the referencing column(s) and the referenced column(s).

**------------------------------------------------------------------------------------------------------------------------**

# Microservices Patterns [Book Notes]

## Distributed Systems

# Kubernetes

## Virtualization 1 Technology:

VMWare and VirtualBox – host əməliyyat sistemində (kompüterimizdə) guest əməliyyat sistemlərini qaldirmaq üçün istifadə olunur. Guest əməliyyat sistemlərini Hypervisor vasitəsilə host əməliyyat sistemində qaldiririq.

## **Virtualization 2 Technology:**

when container comes up, no need for Hypervisor.

Docker – container management tool.



## Kubernetes

Container management problemini həll edən Container Orchestartor’dur. With the help of Kubernetes, we can easily deploy, manage, monitor and scale our applications. Kubernetes Go dilində yazilmişdir.

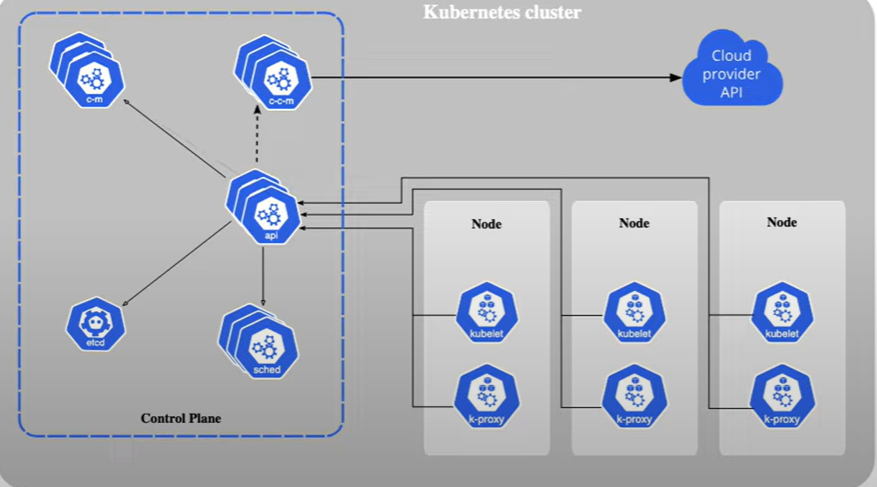
Kubernetes (K8S) Architecture: **Cluster** ən böyük component’di. Cluster tərkibində **master** (master slave or Control Plane) and **worker nodes**. Əsas məntiq və konfiqurasiyalar Control Plane’in içindədir. Application’lar node’larin içində run olur.

Control Plane’de 4 component var:

* **etcd** – persistence storage; cluster’in butun informasiyasin saxliyir (pods states)
* **controller manager** – cluster’in up (healthy) olmasindan cavabdehdir
* **api** – API server is like a REST api. Butun componentler bir-birinnen kubeApi vasitesile elaqe yaradir. Butun command lar yazilanda kubeApi server e gedir ilk olaraq.
* **sched** – kube scheduler application deploy etmey istedikde etcd’den info’sun cekir, meselen app deploy etmek ucun ne qeder resource ayrilir veya nece replicasi olacaq, bu datalara uygun olaraq hansi node’a deploy olacaq onu teyin edir.

Worker Node components:

* **kubelet** – node agent; app deploy olmasindan sorumlu; api’ye sorgu atir, etcd’en app’in info’sun cekir ve cri desteyile app’I deploy edir;
* **k-proxy** – assigns unique IP address to each pod; also acts like load balancer; routes traffic to appropriate pods
* **cri** – container runtime interface (docker);



## Pod

Kubernetes de en kicik component; container pod’un icinde olur; containerleri manage etmek ucun poddan istifade olunur. Kubernetes does not manage containers, it manages pods. 1 pod’un icinde 1 veya 100 container ola biler. Eger 1’den artiq container varsa bir pod’da, Kubernetes says there should be dependency between containers. Or containers have dependencies on each other, they should be on the same pod. Each pod has an unique IP address.

**Pod Lifecycles in K8S**

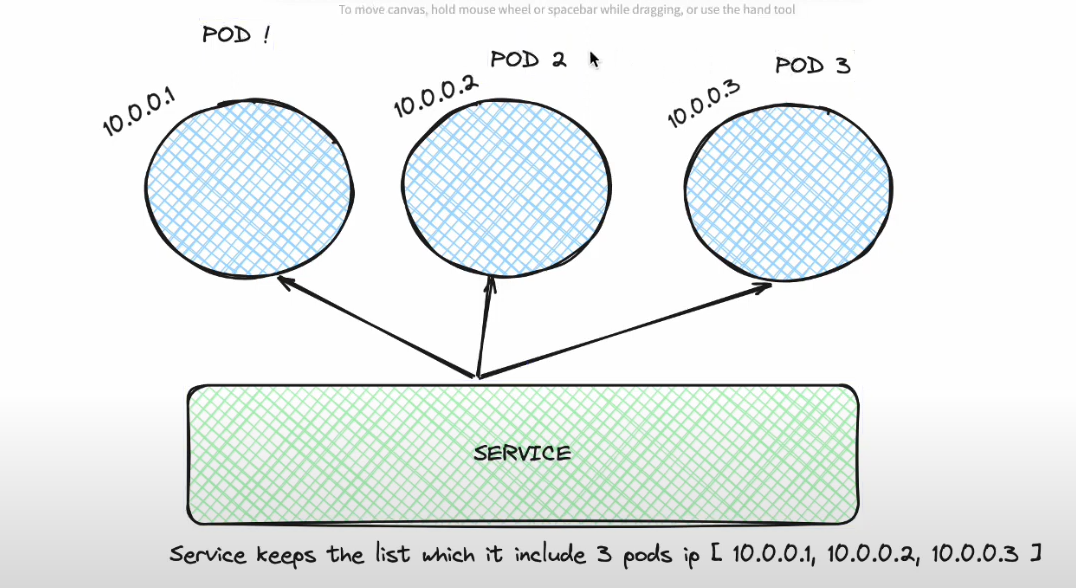
**Pod has 5 states:**

* **Pending** – its containers are starting
* **Running** – at least one of the containers defined in the pod is successfully running
* **Succeeded** – all containers in the pod have terminated successfully
* **Failed** – one or more containers in the pod has terminated unsuccessfully
* **Unknown** – Kubelet stop reporting to API server; when kublet api server uzerinden etcd’den melumat ala bilmir pod state info’sun

**Pod Probes:**

* **Liveness Probe** – pod’un veziyyeti healthy’dirmi?, bir k8s podunun application icerisindeki xususi bir endpoint’ine (springboot’da meselen /health) muntezem request atib app healthy’dirmi (success veya running state’dedirmi) deye yoxluyur, deyilse healthy veziyetine getirmeye calisir (kills pod and restart regarding on restart policy);
* **Readiness Probe** – pod’un request trafikini qebul edib, ede bilmediyini yoxluyur;
* **Startup Prove** – pod’un start olunduqdan bir nece saniye muddetinde aktiv olur. Diger probe’lardan ferqi limited muddet isleyir, yalniz pod successfully up olduqdan bir nece saniye ve ya deqiqe sora.

Service (Load Balancer) high load olan (exp. card ms) ms’e gelen requestleri beraber sayda (Round Robin) podlar (replicas) arasinda bolur. Service ozunde list of healthy ips saxliyir Readiness Probe sayesinde. Eger pod3 fail olsa, listden pod3’un IP’sin cixarir ve neticede requestler pod1 ve pod2 arasinda bolunur till pod3 restart olana kimi. [10.0.0.1, 10.0.0.2, 10.0.03] ---> [10.0.0.1, 10.0.0.2]



## Deployment

Deployment k8s seviyesinde bir obyektdir (like pod, service). Deploy etmek ucun Deployment object’inden istifade olunur. Birbasa olaraq REST obyektleridir. Configurasiya file yazilir ve k8s’e gonderilir ki bu file’i oxu ve deploy ele. YAML veya JSON fayllarinda manifest fayli olaraq movcud olur. Yaradilmasi kube-api-server vasitesile heyata kecirilir.

## Service

Service – K8S objectidir; xaricden gelen traffic’i podlara yonlendirir. Standart IP and DNS’i olur. Load Balancing funksiyasini icra edir. TCP protocla isleyir. Hemise traffic healhty podlara yonlendirilir.

Service Types: ClusterIP, NodePort, LoadBalancer

## ReplicaSet

Hansisa pod’un mueyyen sayda islemesine teyin eden kubernetes object’idir. Teyin olunmus sayda kopya yaradaraq, pod’larin duzgun islemesinden emin olur. Bu sayede, pod’larda yaranacaq xetalar ve ya nasazliqlar zamani, ReplicaSet yeni bir pod yaradaraq application’in duzgun islemesini temin edir.

Automatic Scaling: ReplicaSet traffic’e gore pod sayini artiraraq performance problemlerinin qarisini alir.

Update Management: yeni versiya release zamani, versiyalari podlara yuklenmesini temin edir.

## Rollout / Rollback

Roollout – bir app’in yeni versiyasini release edende bu mexanizmden istifade olunur. Yeni versiyani tedricen pod’lara yayir ve kohne versiyalari merheleli silir. App’in yeni versiya release’inde min kesintilere endirir.

Rollback – release zamani problem olsa, app’in bir evvelki versiyasina qayida bilmeliyik.