Software Architecture & Design of Large Scale Systems

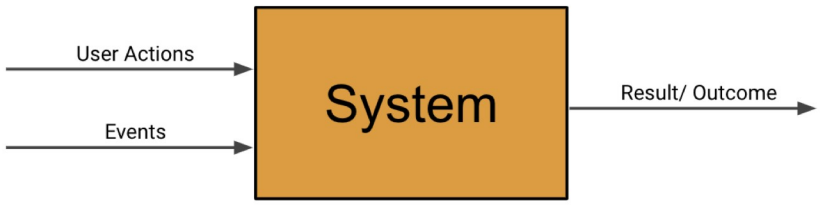
By Michael Pogrebinsky

Copyright Notice

Introduction to System Requirements & Architectural Drivers

Introduction to System Design & Architectural Drivers

- **Requirements** - Formal description of what we need to build
- **Types of Requirements** - Architectural Drivers
    - Features of the System
        - Functional requirements



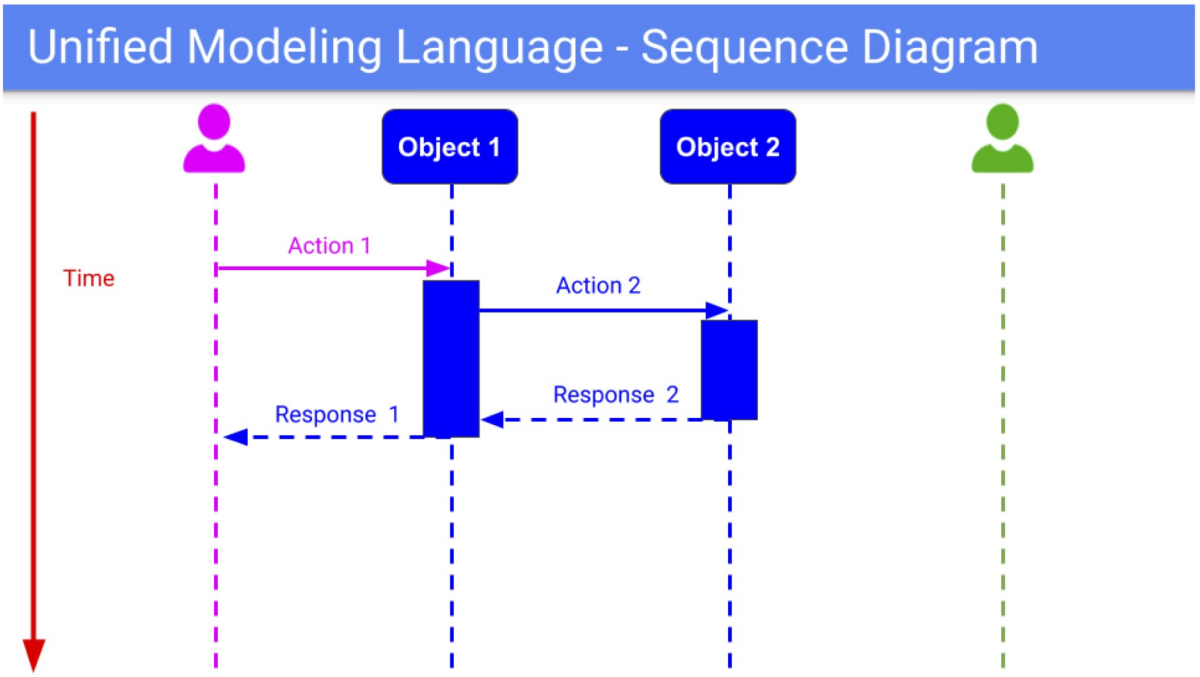
    - Quality Attributes
        - Non-Functional requirements
            - Examples:
                - Scalability
                - Availability
                - Reliability
                - Security
                - Performance
        - Dictate the software architecture of our system

    - System Constraints
        - Limitations and boundaries

Notes:

Feature Requirements - Step by Step Process

- Methods of Gathering Requirements
  - Use Cases
    - Situation / Scenario in which our system is used
  - User Flows
    - A Step-By-Step / Graphical representation of each use case
- Requirement Gathering Steps
  - Identify all the actors/users in our system
  - Capture and describe all the possible use-cases/ scenarios
  - User Flow - Expand each use case through flow of events.
    - Each event contains
      - Action
      - Data
- Sequence Diagram
  - Diagram that represents interactions between actors and objects.



Notes:

1) My personal note 1
2) My personal note 2

System Quality Attributes Requirements

- System Quality Attributes
  - Provide a quality measure on how well our system performs on a particular dimension
  - Have direct correlation with the architecture of our system
- Important Considerations
  - Testability and Measurability
  - Trade Offs
    - No single software architecture can provide all the quality attributes.
    - Certain quality attributes contradict one another
    - Some combinations of quality attributes are very hard / impossible to achieve
  - Feasibility
    - We need to make sure that the system is capable of delivering with the client asking for

Notes:

System Constraints in Software Architecture

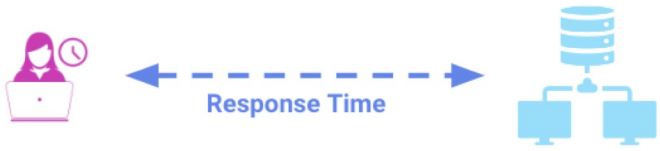
- Definition:
  - “A system constraint is essentially a decision that was already either fully or partially made for us, restricting our degrees of freedom.”
- Types of Constraints:
  - Technical constraints
  - Business constraints
    - Forces us to make sacrifices in:
      - Architecture
      - Implementation
  - Regulatory/legal constraints
    - Global
    - Specific to a region
- Considerations:
- We shouldn't take any given constraint lightly
- Use loosely coupled architecture

Notes:

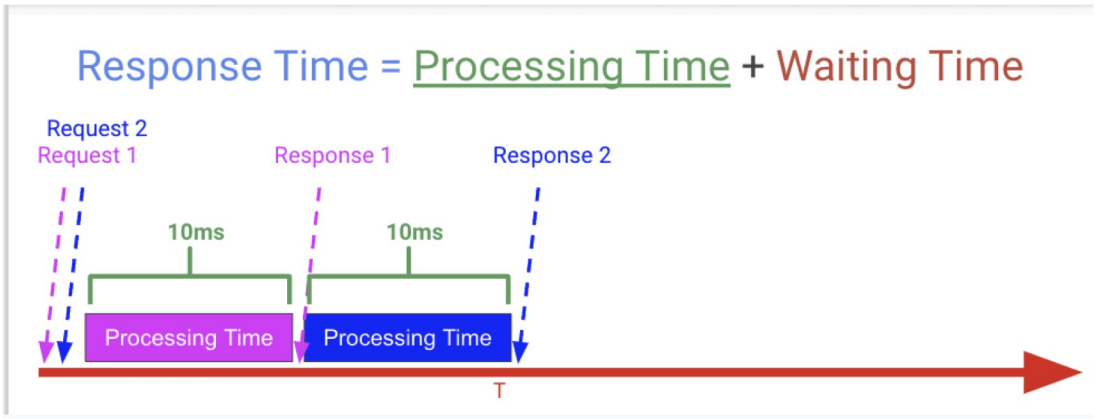
Most Important Quality Attributes in Large Scale Systems

Performance

- Definitions
  - **Response Time**:
    - Time between a client sending a request and receiving a response
    - Response Time = Processing Time + Waiting Time
    - Waiting Time - Duration of time request/response spends inactively in our system

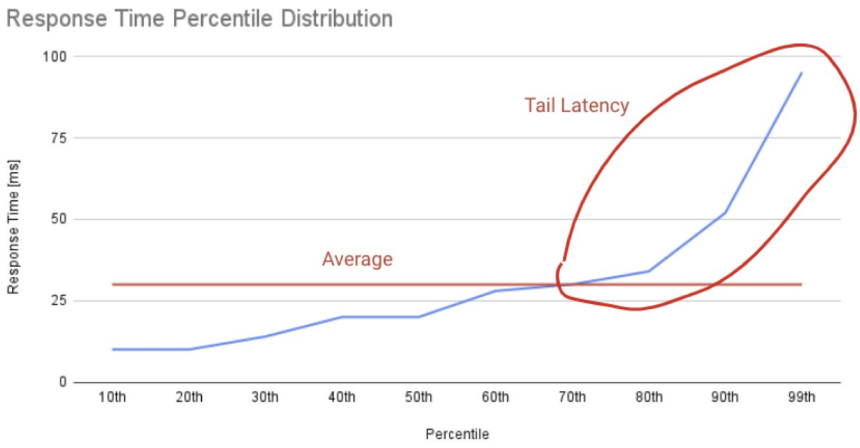


  - **Throughput**
    - Amount of work performed by our system time
      - Measured in tasks/second
    - Amount of data processed by our system per unit of time
      - Measured in bits/second, Bytes/second, MBytes/second
- Important Considerations:
  - Measuring Response Time Correctly



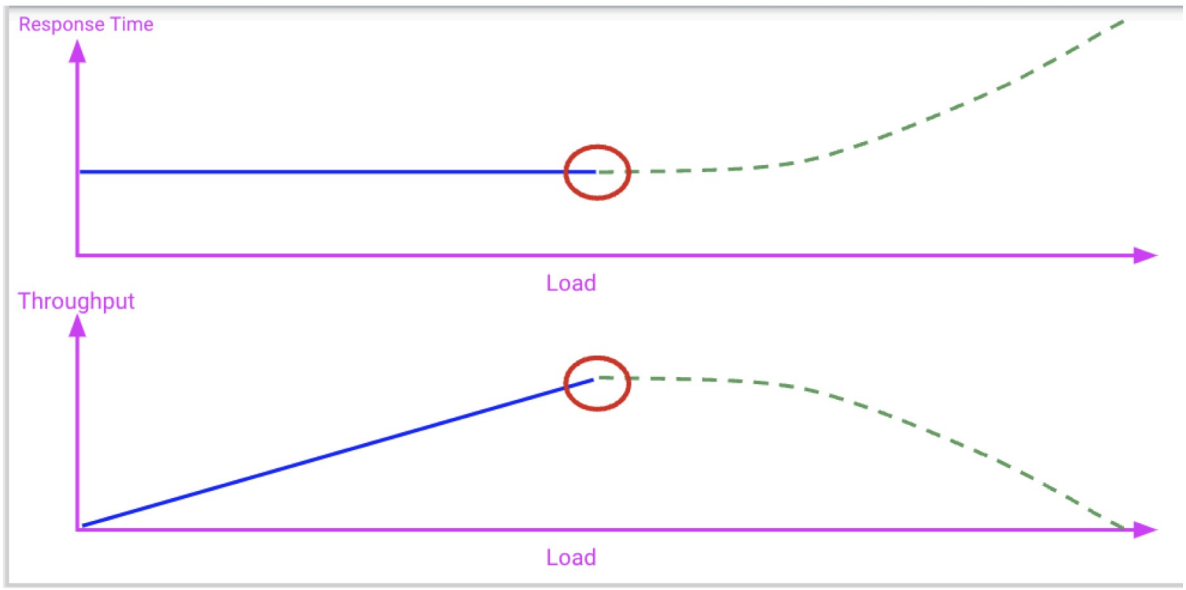
  - Response Time Distribution

- **Percentile**: The “xth percentile” is the value below which x% of the values can be found



- **Tail Latency**: The small percentage of response times from a system, that take the longest in comparison to the rest of values

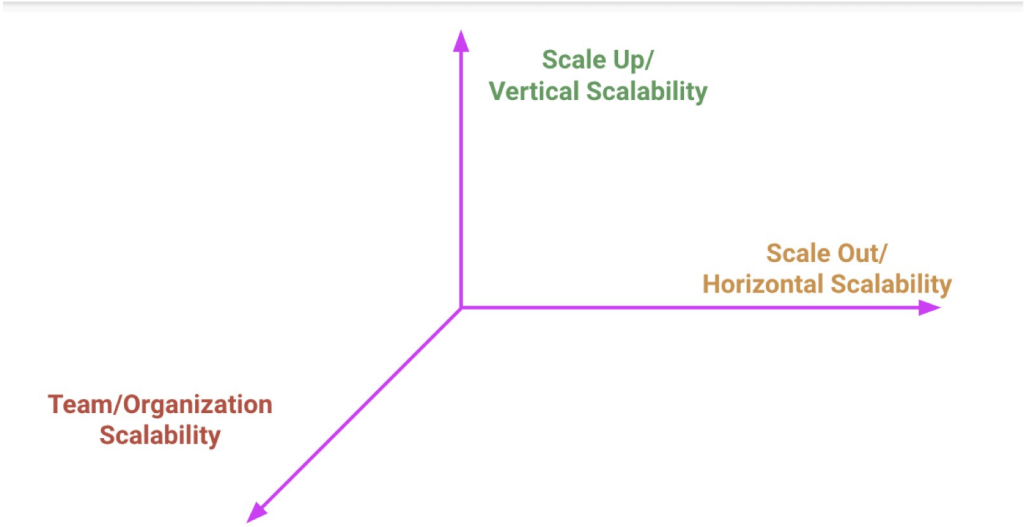
- Performance Degradation



Notes:

Scalability

- **Scalability Definition:**
  - "The measure of a systems ability to handle a growing amount of work, in an easy and cost effective way, by adding resources to the system"
- **Types of Scalability**



  - Vertical Scalability
    - Adding resources or upgrading the existing resources on a single computer
  - Horizontal Scalability
    - Adding more resources in a form of new instances running on different machines
  - Team/Organizational Scalability
    - Software Architecture impacts engineering velocity (team productivity)

Notes:

Availability - Introduction & Measurement

- **Availability**:
  - *"The fraction of time/probability that our service is operationally functional and accessible to the user."*

$$Availability = Uptime / (Uptime + Downtime)$$

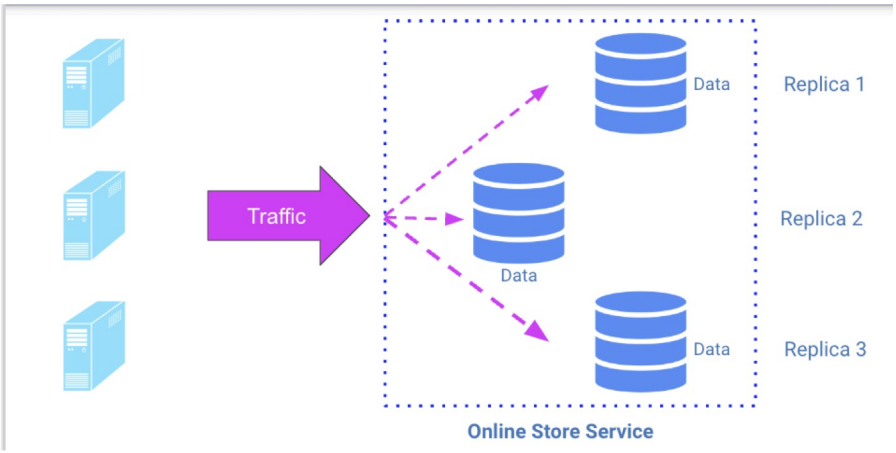
- **Uptime**:
  - Time that our system is operationally functional and accessible to the user
- **Downtime:**
  - Time that our system is unavailable to the user
- **MTTR**
  - Mean Time to Recovery

$$Availability = MTBF / (MTBF + MTTR)$$

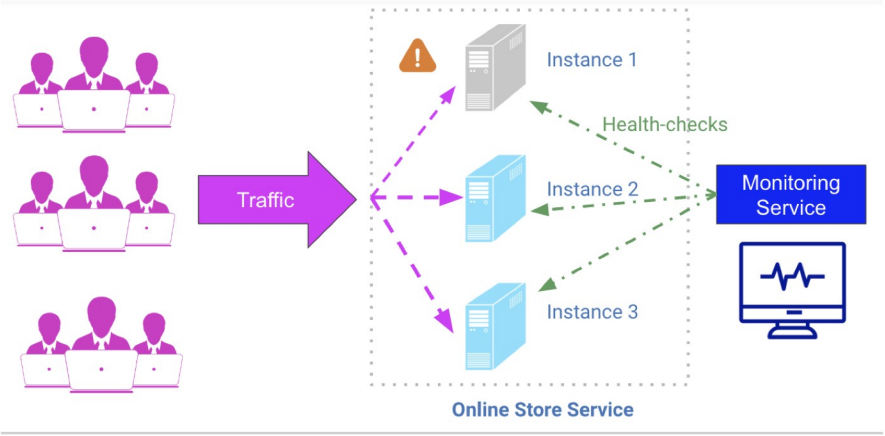
Notes:

Fault Tolerance & High Availability

- **Sources of Failure:**
  - Human Error
  - Software Errors
  - Hardware Failures
- **Fault Tolerance:**
  - *"Enables our system to remain operational and available to the users despite failures within one or multiple of its components".*
- **Tactics for achieving Fault Tolerance**
  - Failure Prevention
    - Redundancy and Replication



  - Failure Detection and Isolation:
    - Monitoring



  - Recovery
    - Stop sending traffic
    - Restart the host
    - Rollback

Notes:

SLA, SLO, SLI

- **SLA - Service Level Agreement**
  - It is a legal contract that represents our quality service
- **SLOs - Service Level Objectives**
  - Each SLO represents a target value/range that our service needs to meet
- **SLIs - Service Level Indicators**
  - Quantitative measure of our compliance with a service-level objective
- **Important Considerations:**
  - We shouldn't take every SLI that we can measure in our system and define an objective associated with it
  - Promising fewer SLOs is better
  - Set realistic goals with a budget for error
  - Create a recovery plan for when the SLIs show that we are not meeting our SLOs

Notes:

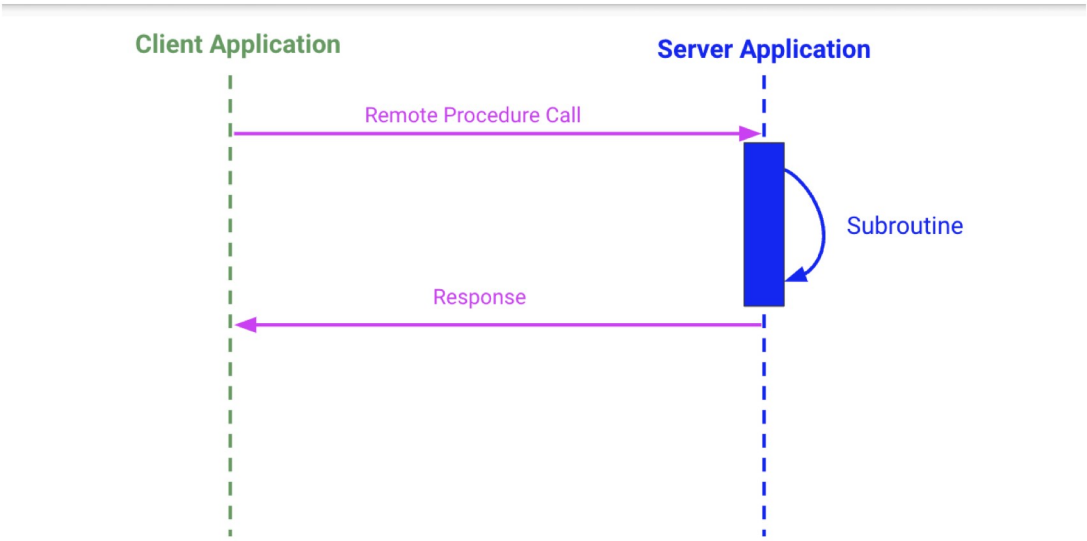
API Design

Introduction to API Design for Software Architects

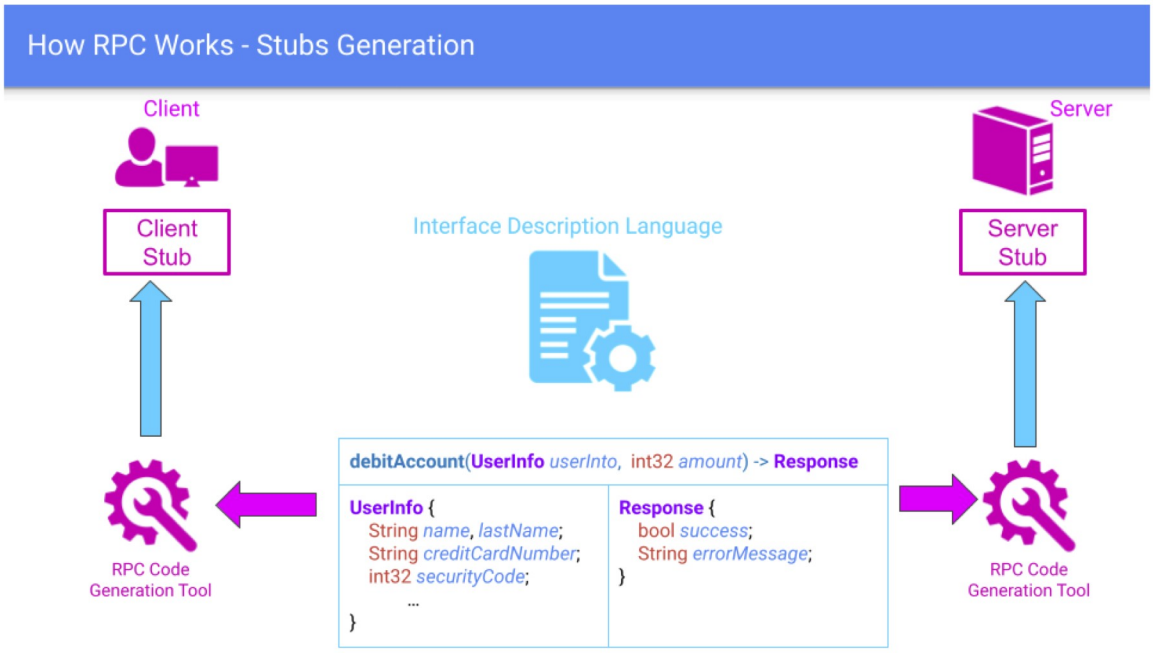
- **An API is a contract between:**
  - Engineers who implement the system
  - Client applications who use the system
- **Categories of API**
  - Public APIs
  - Private/Internal APIs
  - Partner APIs
- **API best practices and patterns:**
  - Complete Encapsulation of the internal design and implementation
  - Easy to Use
  - Keeping the Operations Idempotent
    - *"An operation doesn't have any additional effect on the result if it is performed more than once"*
  - API Pagination
  - Asynchronous Operations
  - Versioning our API

Notes:

RPC



- **Features of RPC:**
  - Looks like calling a normal local method
  - RPC frameworks support multiple programming languages



- **Benefits of RPC:**
  - Convenience to the developers
  - The details of communication establishment/data transfer between client to server are abstracted
  - Failures in communication with server result in an error or exception depending on the programming language

- **Drawbacks of RPC over local method invocation:**
  - Slower
  - Less reliable

Notes:

REST API

- **REST - Representational State Transfer**
  - Set of architectural constraints and best practices for defining APIs for the web
- **Important Concepts:**
  - HATEOAS -
    - The interface is dynamic through Hypermedia as the Engine of the Application State (HATEOAS)
  - Statelessness
  - Cacheability
  - Named Resources - Each resource is either:
    - Simple resource
    - Collection resource
- **Resources - Best Practices:**
  - Naming our resources using nouns
  - Making a distinction between collection resources and simple resources
  - Giving the resources clear and meaningful names
  - The resource identifiers should be unique and URL friendly
- **REST API Operations Mapping to HTTP Methods**
  - REST operations are mapped to HTTP methods as follows:
    - ***Create*** *a new resource* -> **POST**
    - ***Update*** *an existing resource* -> **PUT**
    - ***Delete*** *an existing resource* -> **DELETE**
    - ***Get*** *the state of a resource* } **GET**
    - ***List*** *the sub-resources of a collection* } **GET**
  - In some situations, we define additional custom methods

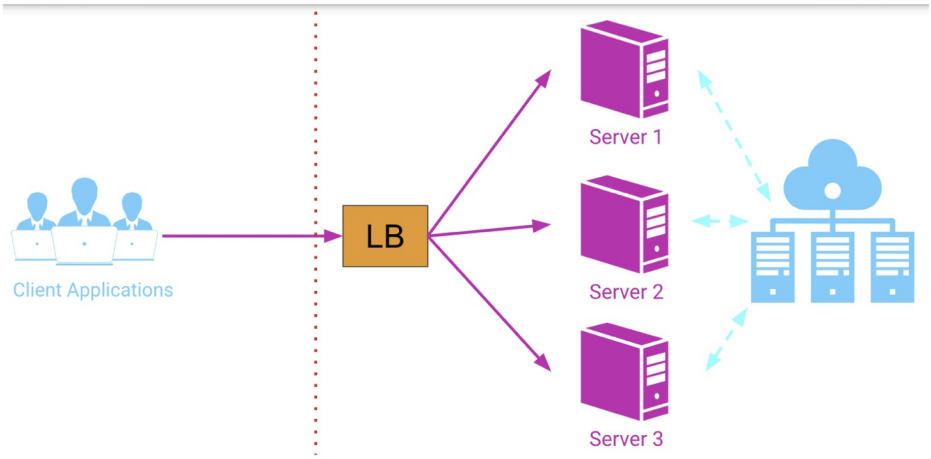
- **REST API - Step by Step Process**
  - Identifying Entities
  - Mapping Entities to URIs
  - Defining Resources' Representations
  - Assigning HTTP Methods To Operations on Resources

Notes:

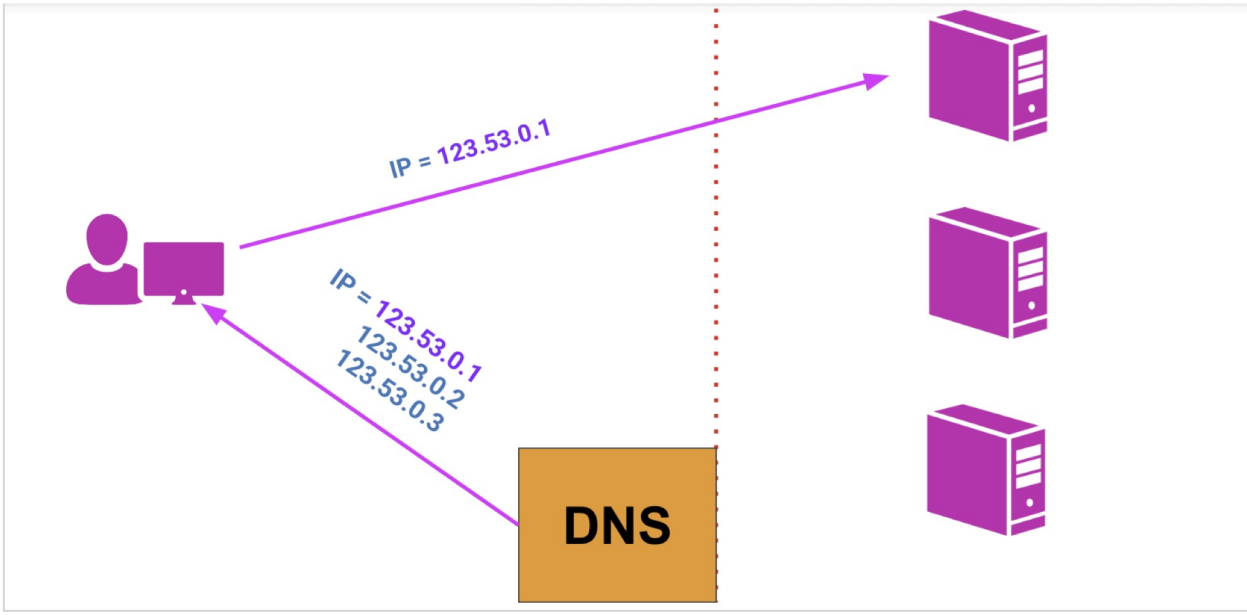
Large Scale Systems Architectural Building Blocks

DNS, Load Balancing & GSLB

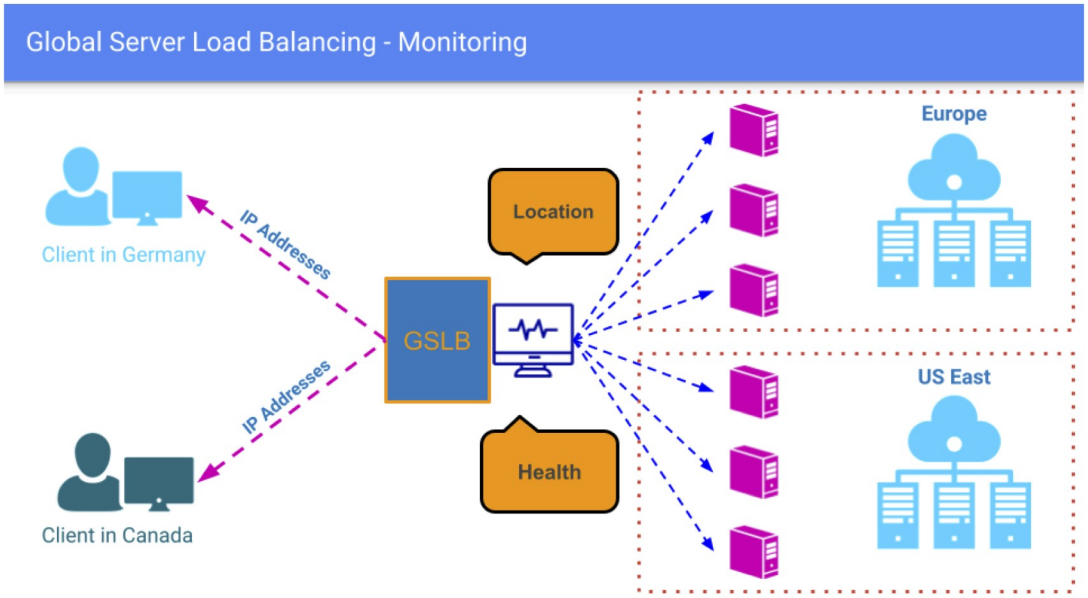
- **Role of Load Balancer:**
  - Balance load among a group of servers



- **Types of load balancers**
  - DNS load balancing



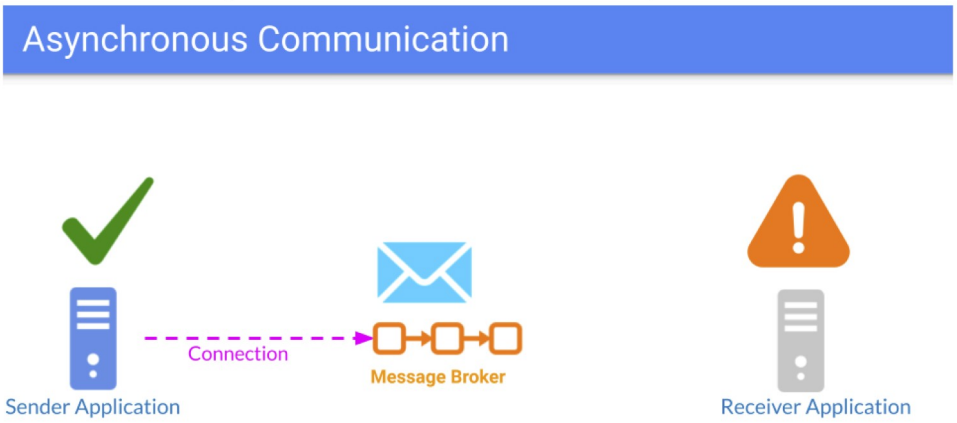
  - Hardware load balancing
    - Run on dedicated devices designed and optimized specifically for load balancing
  - Software load balancing
    - Programs that can run on a general-purpose computer and perform a load balancing function
  - Global Server Load Balancing



Notes:

Message Brokers

- **Defination:**
  - A software architectural building block that uses the queue data structure to store messages between senders and receivers
  - Used inside our system and not exposed externally

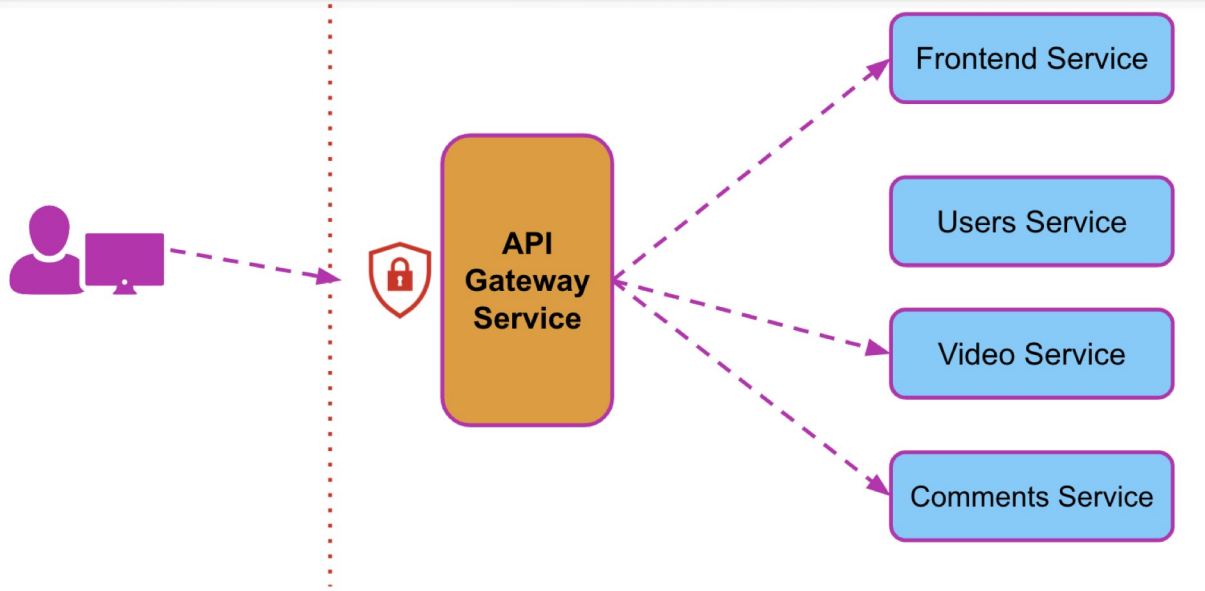


- **Benefits:**
  - Services can
    - Publish messages to a particular channel
    - Subscribe to that channel
    - Get notified when a new event is published

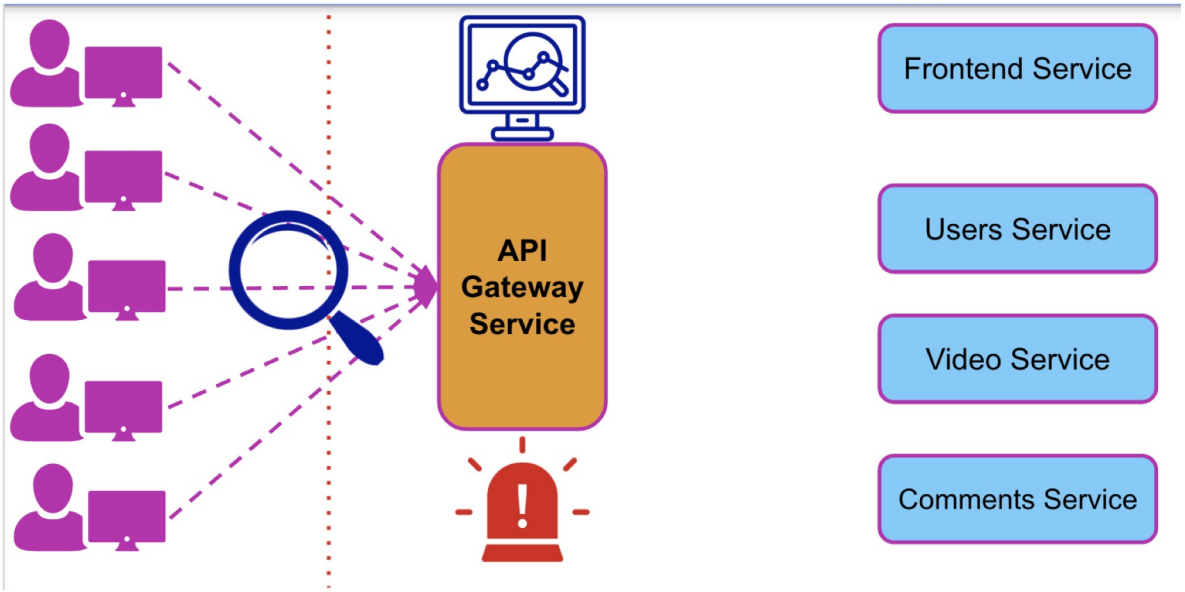
Notes:

API Gateway

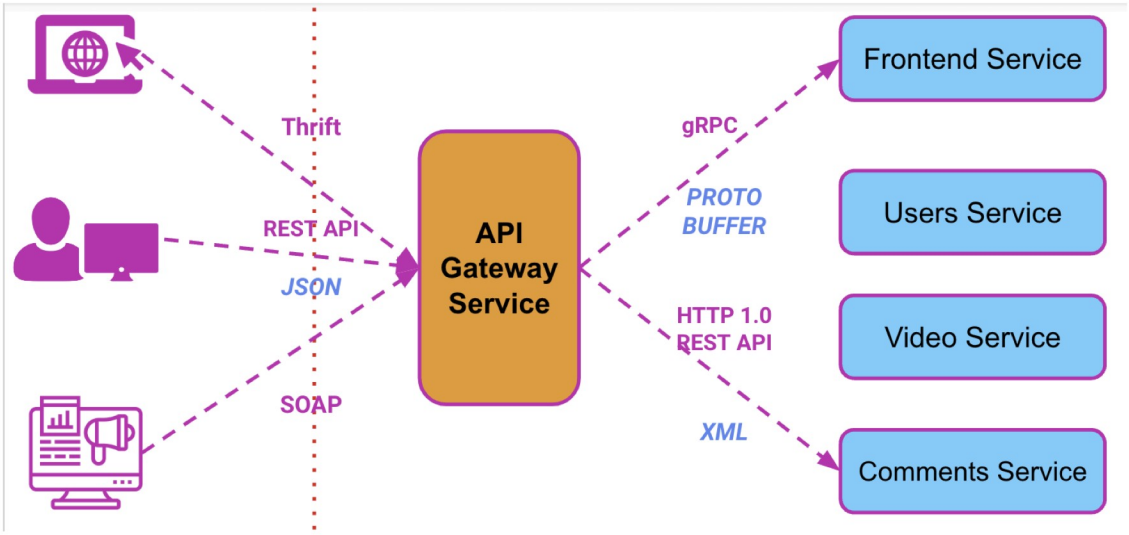
- **Definition**:
  - Follows a software architecture pattern called “API composition”
  - The client applications can call one single service



- **Benefits**
  - Seamless internal modifications/Refactoring
  - Consolidating all security, authorization, and authentication in a single place
    Request Routing
  - Static content and response caching
  - Monitoring and alerting



  - Protocol Translation

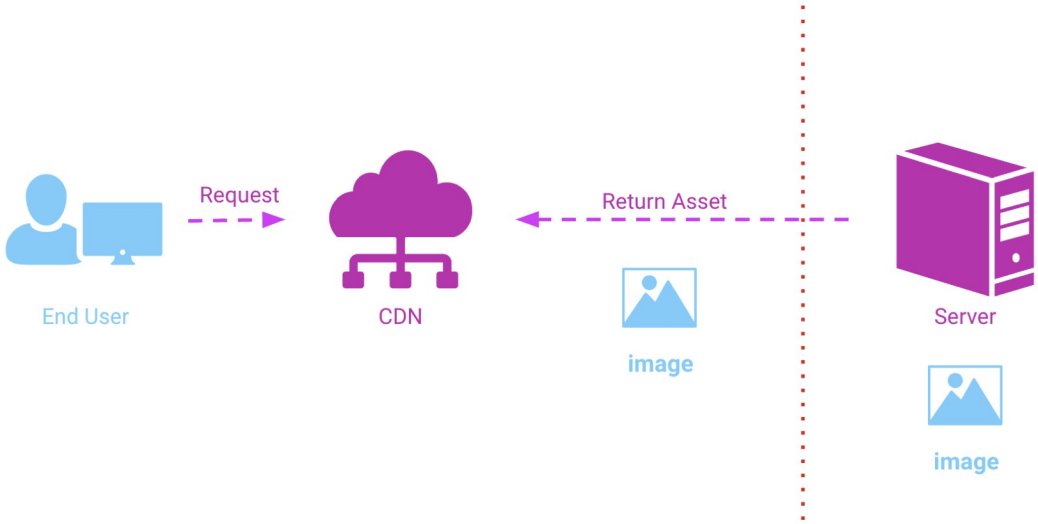


- **Considerations**:
  - API Gateway shouldn't contain any business logic
  - API Gateway may become a Single Point of Failure
  - Avoid bypassing API Gateway from external services

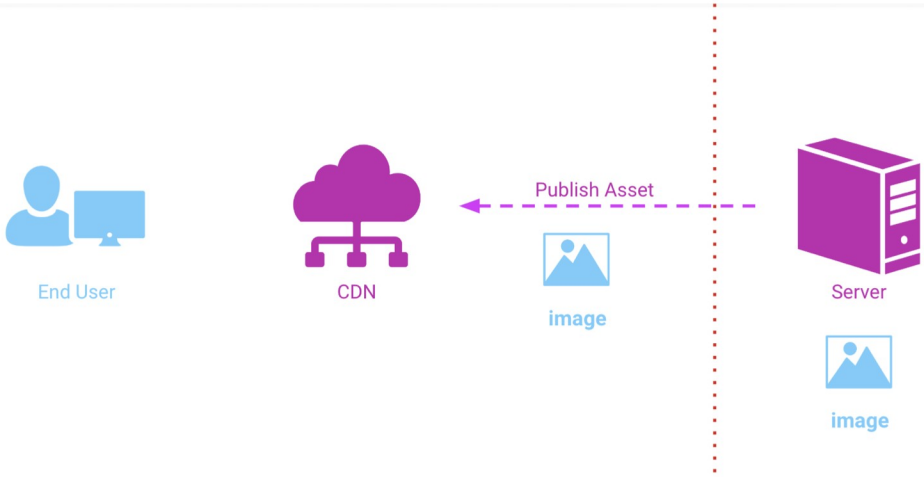
Notes:

Content Delivery Network - CDN

- **Definition:**
  - Globally distributed network of servers located in strategic places
- **Main purpose:**
  - Speeding up the delivery of content to end-users
- **Content Publishing Strategies**
  - *Pull Strategy*



  - *Push Strategy*



Notes:

Data Storage at Global Scale

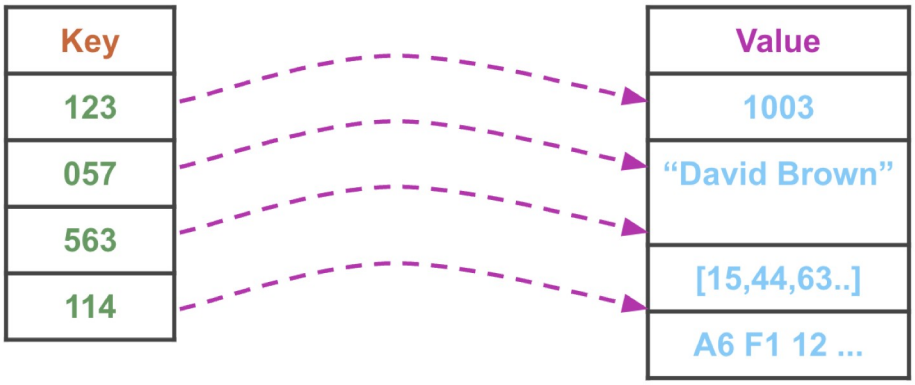
Relational Databases & ACID Transactions

- **Properties:**
  - The structure (schema) of each table is defined ahead of time
  - Gives us the knowledge of each what each record must have
- **Advantages:**
  - Ability to form complex and flexible queries
  - Efficient storage
  - Natural structure of data for humans
  - ACID transactions
    - *Atomicity* - Each set of operations that are part of one transaction either:
      - Appear all at once
      - Don't appear at all
    - *Consistency* -
      - A transaction that was already committed is seen by all future queries/transactions
      - A transaction doesn't violate any constraints that we set for our data
    - *Isolation*
      - Related to Atomicity in the context of concurrent operations performed on our database
    - *Durability*
      - Once a transaction is complete, its final state will persist and remain permanently inside the database

Notes:

Non-Relational Databases

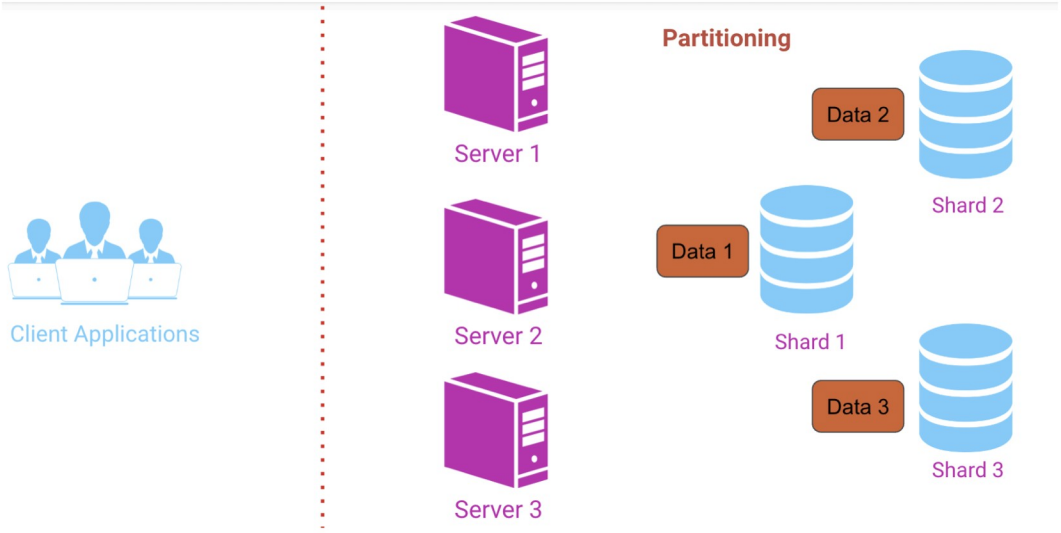
- Categories:
  - **Key/Value Store**



  - **Document Store**
    - We can store collections of documents, with more structure inside each document
    - Each document is an object with different attributes
  - **Graph Database**
    - Optimized for navigating and analyzing relationships between different records

Notes:

- **Database Partitioning/Sharding**



Notes:

Techniques to Improve Performance, Availability & Scalability Of Databases

- **Database Indexing**

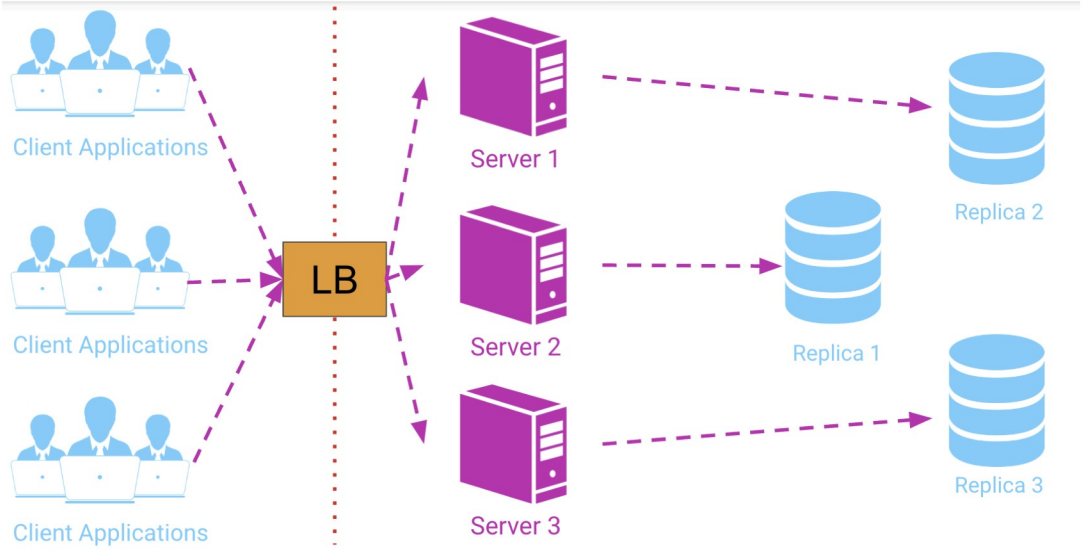
Index Table

City	Row
Los Angeles	1
Boston	2
Los Angeles	3
New York	4
Amsterdam	5
...	...

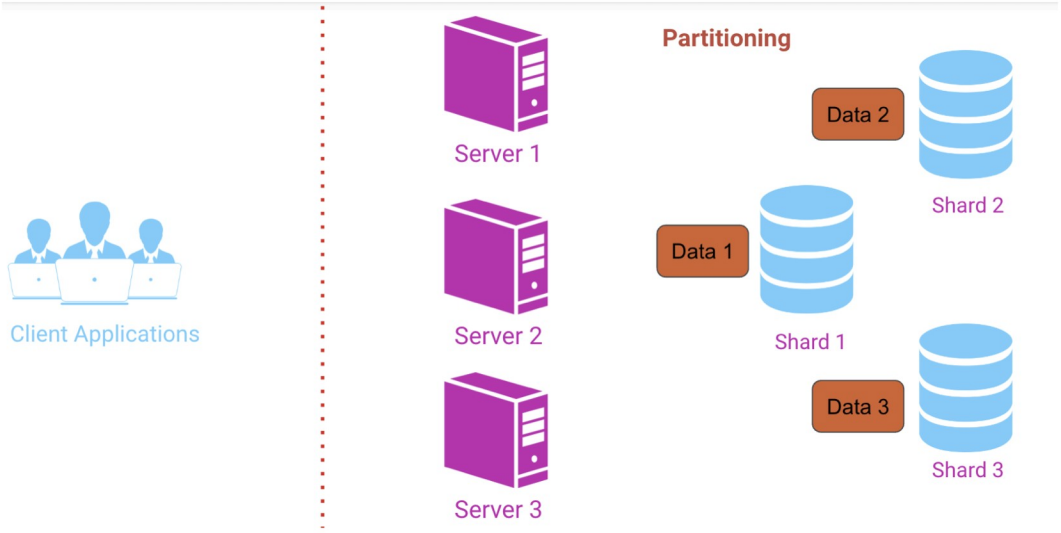
Users Table

Id	City	FirstName	LastName	Age
1	Los Angeles	Alexander	Ivanov	35
2	Boston	John	Smith	44
3	Los Angeles	Mary	Johnson	15
4	New York	Michael	Bird	86
5	Amsterdam	Joseph	Young	51
...	...	...	...	...

- **Database Replication**



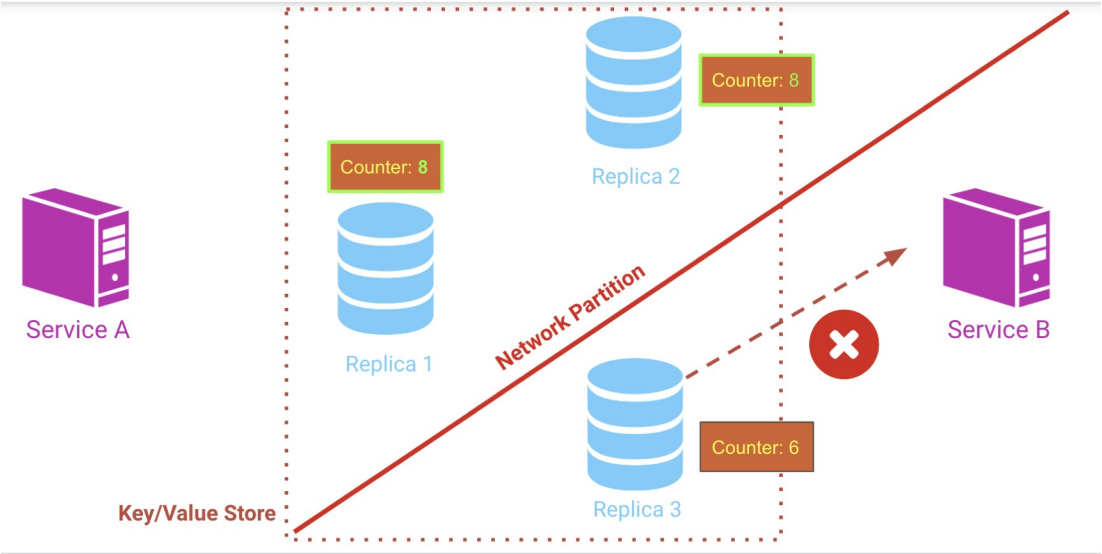
- **Database Partitioning/Sharding**



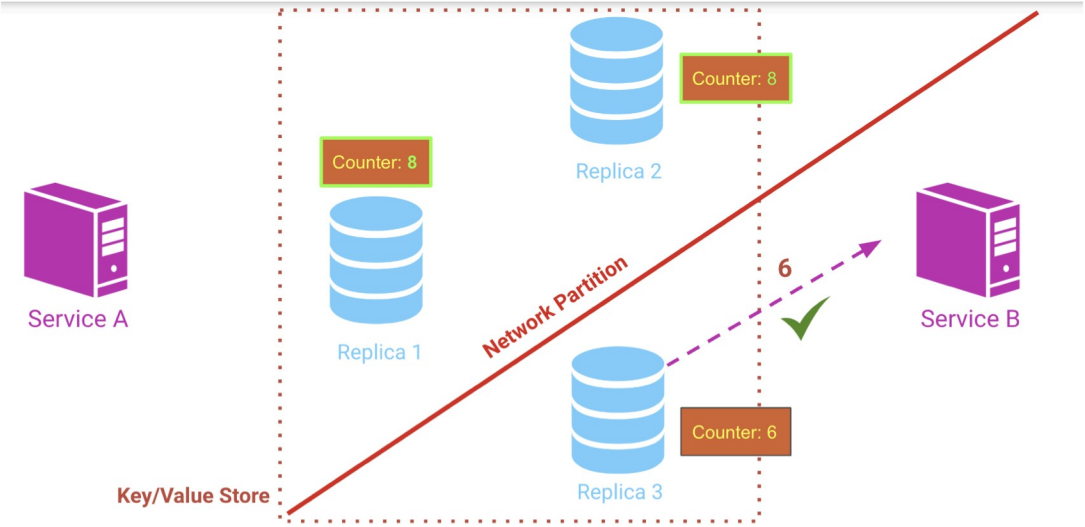
Notes:

Brewer's (CAP) Theorem

- Definition:
  - *"In the presence of a Network Partition, a distributed database cannot guarantee both Consistency and Availability and has to choose only one of them."*
- CAP
  - **Consistency**
    - "Every read request receives either the most recent write or an error"



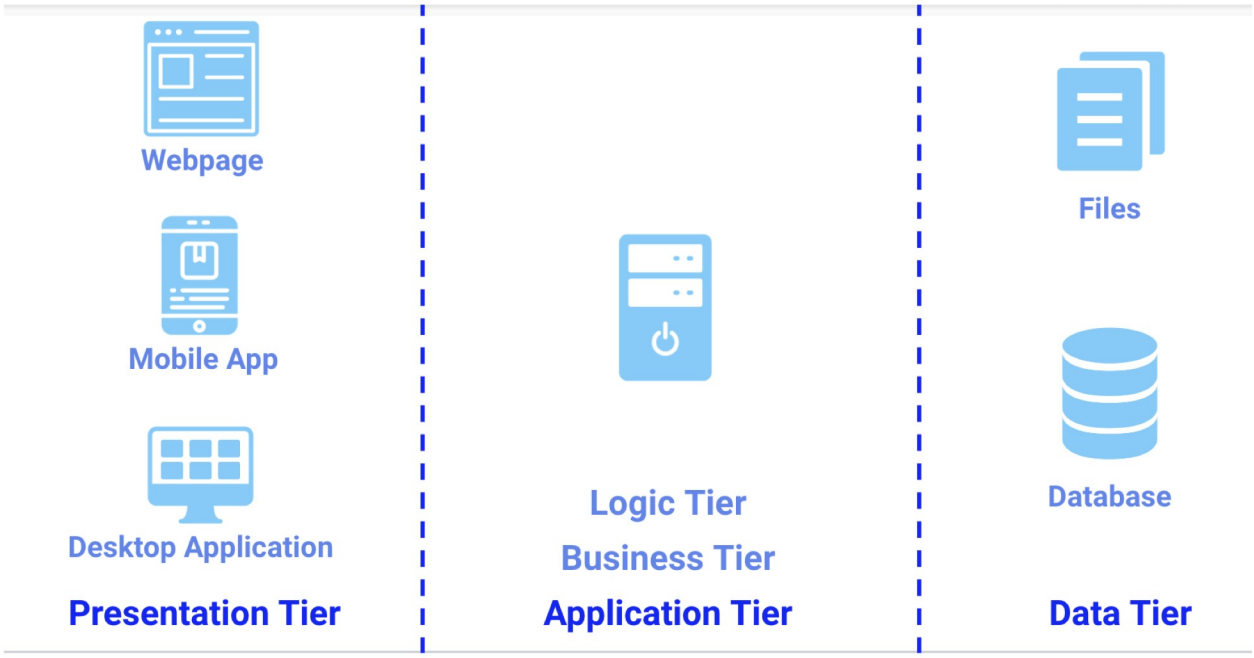
  - **Availability**
    - "Every request receives a non-error response, without the guarantee that it contains the most recent write"



Notes:

Software Architecture Patterns

Multi-Tier Architecture

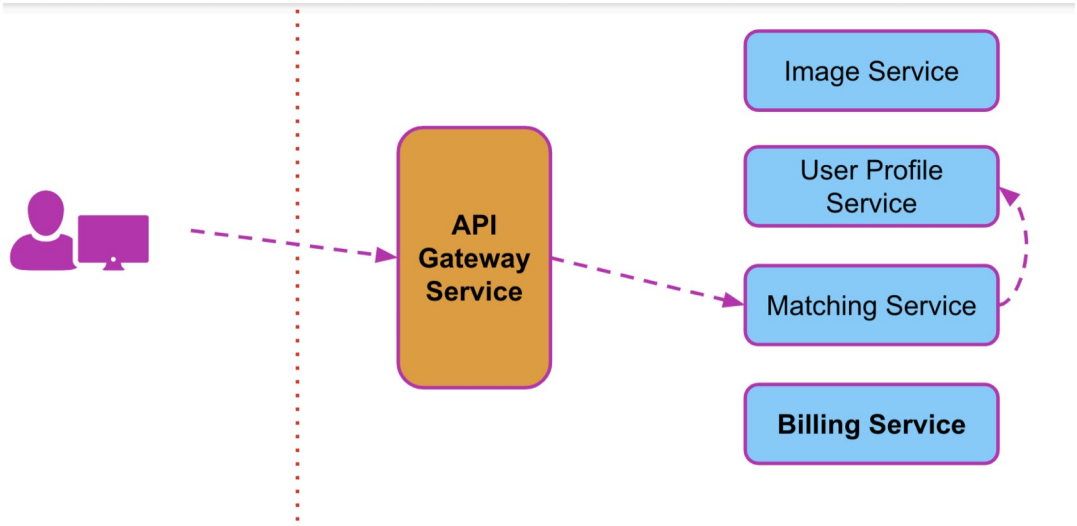


- **Advantages:**
  - Fits a large variety of use cases
  - Easy to scale horizontally
- **Drawbacks:**
  - Monolithic structure of our logic tier

Notes:

Microservices Architecture

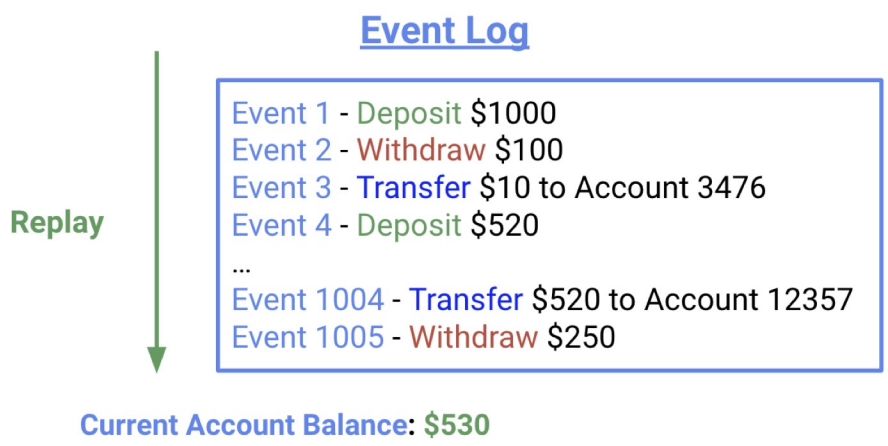
- Definition:
  - *"Microservices Architecture organizes our business logic as a collection of loosely coupled and independently deployed services"*
- Best Practices:
  - Single Responsibility Principle
  - Separate Database Per Service

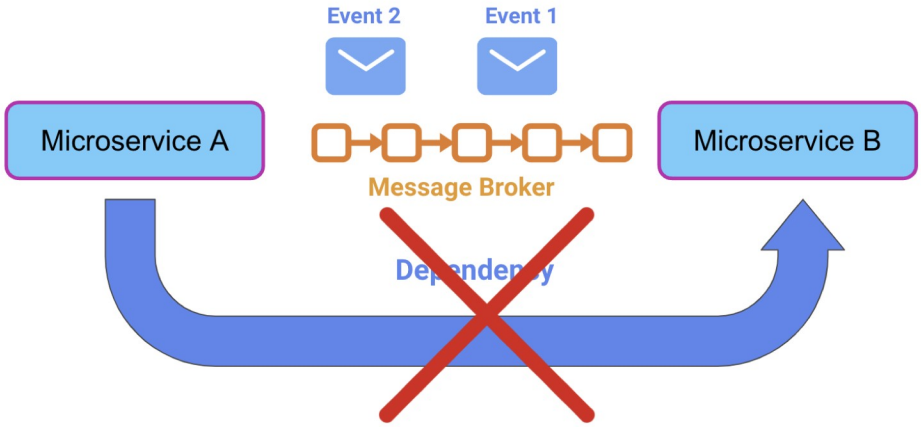


Notes:

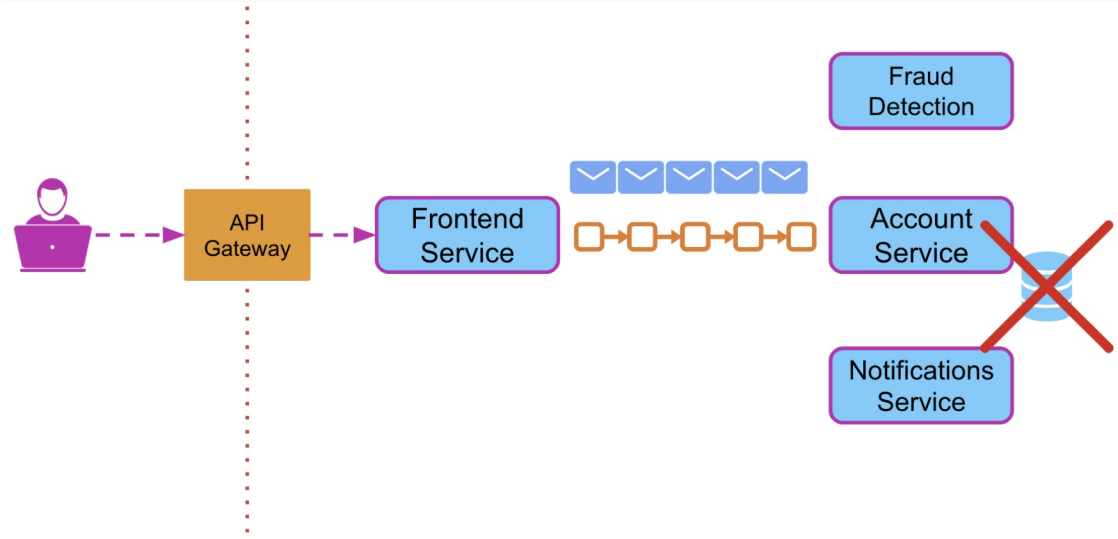
Event Driven Architecture

- Definition:
  - An event is an immutable statement of a fact or a change

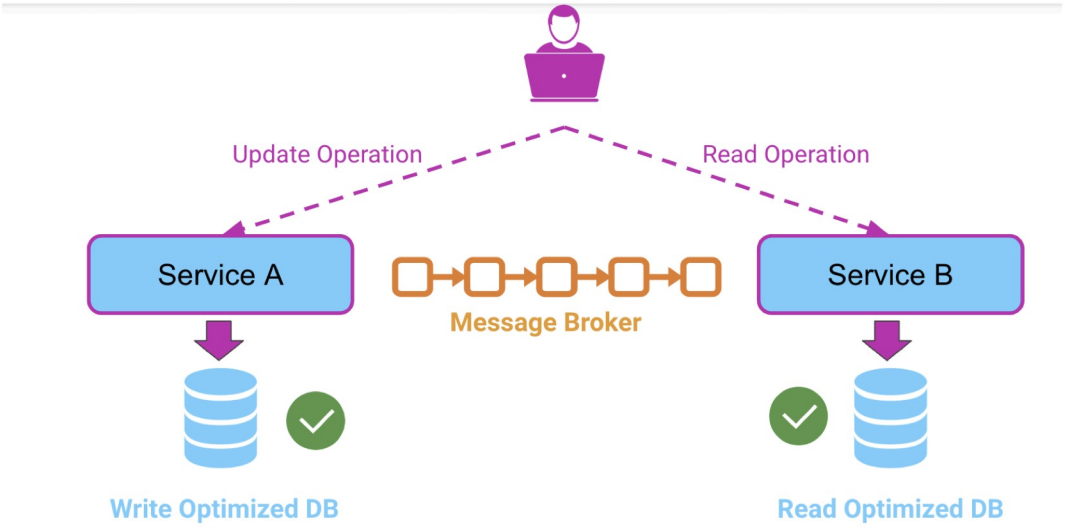


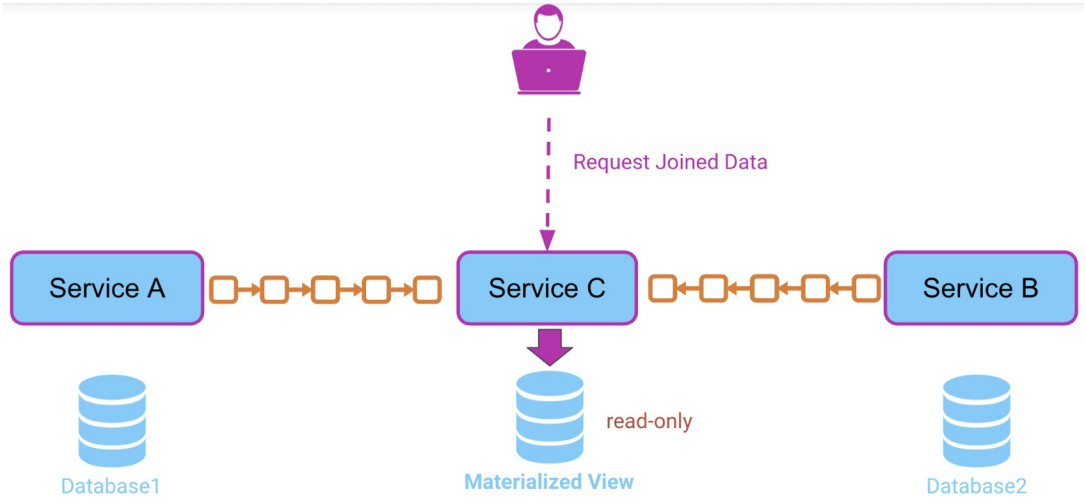


- Event Sourcing Pattern



- CQRS
  - C = Command
  - Q = Query
  - R = Responsibility
  - S = Segregation



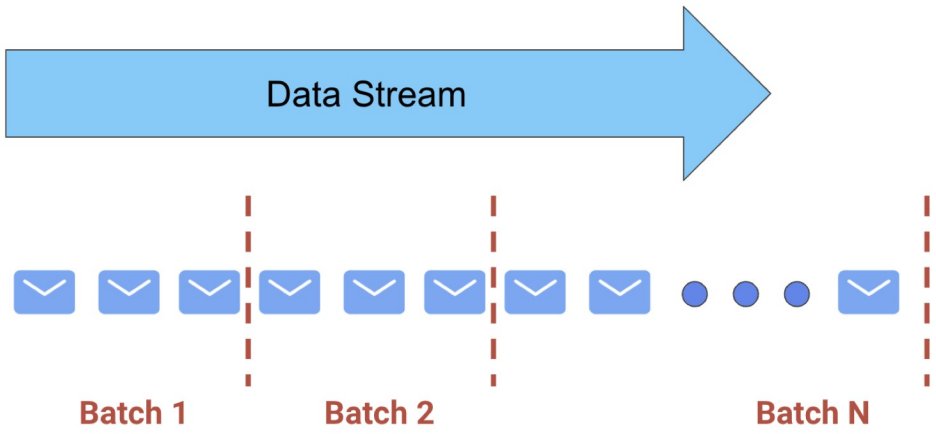


Notes:

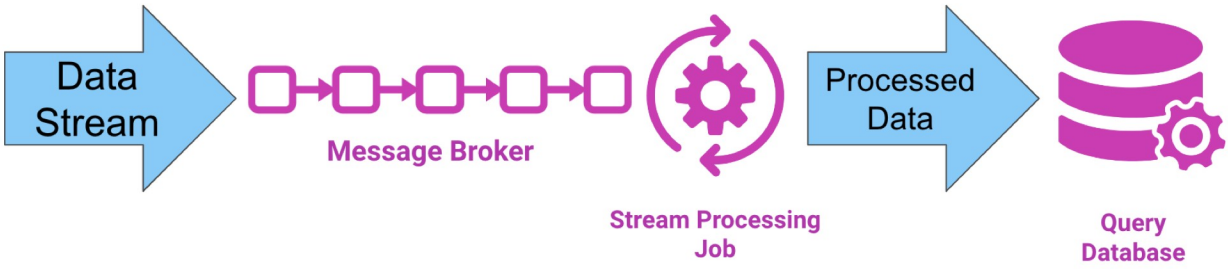
Big Data Architecture Patterns

Big Data Processing Strategies

- **Batch Processing**



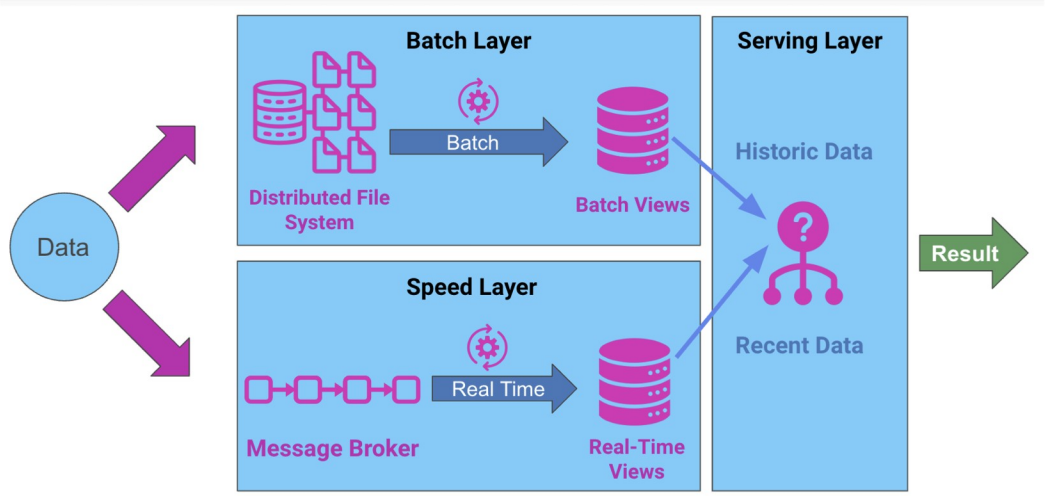
- **Real Time Processing**



Notes:

Lambda Architecture

- Layers:
  - Batch Layer
  - Speed Layer
  - Serving Layer



Notes: