ASP.NET CORE MICROSERVICES

BY:

MAHESH SABNIS

HTTPS://WWW.WEBNETHELPER.COM

WWW.DEVCURRY.COM

WWW.DOTNETCURRY.COM

WHAT ARE MICROSERVICES?

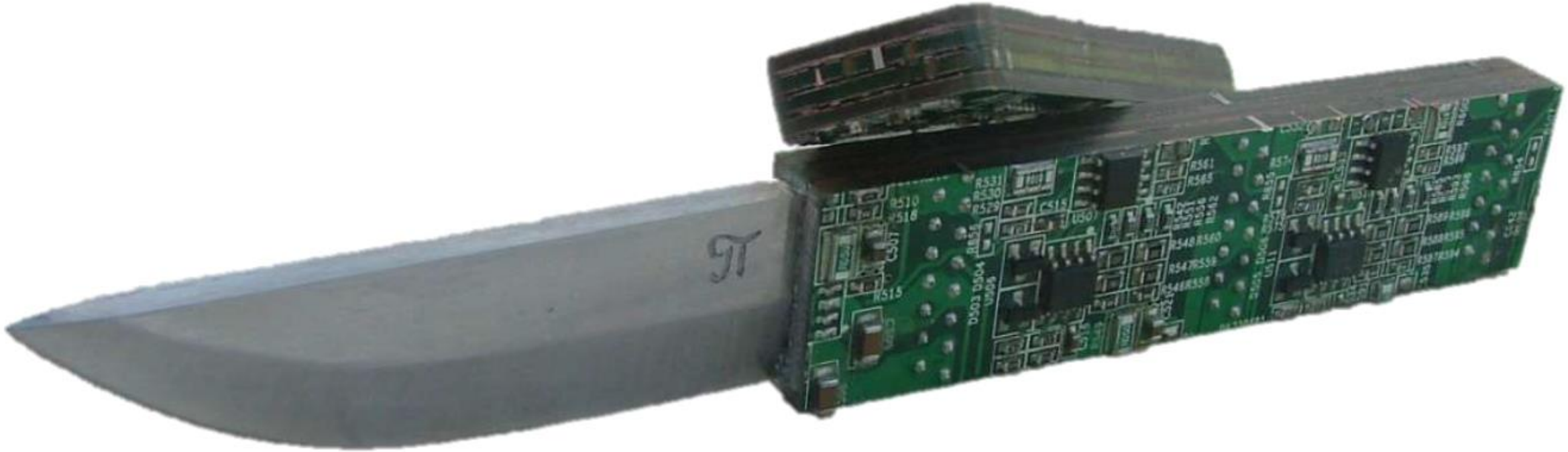
The design requirement from the software system

WHAT ARE MICROSERVICES?

….and what we design and deliver

WHAT ARE MICROSERVICES?

How each tool looks like from inside



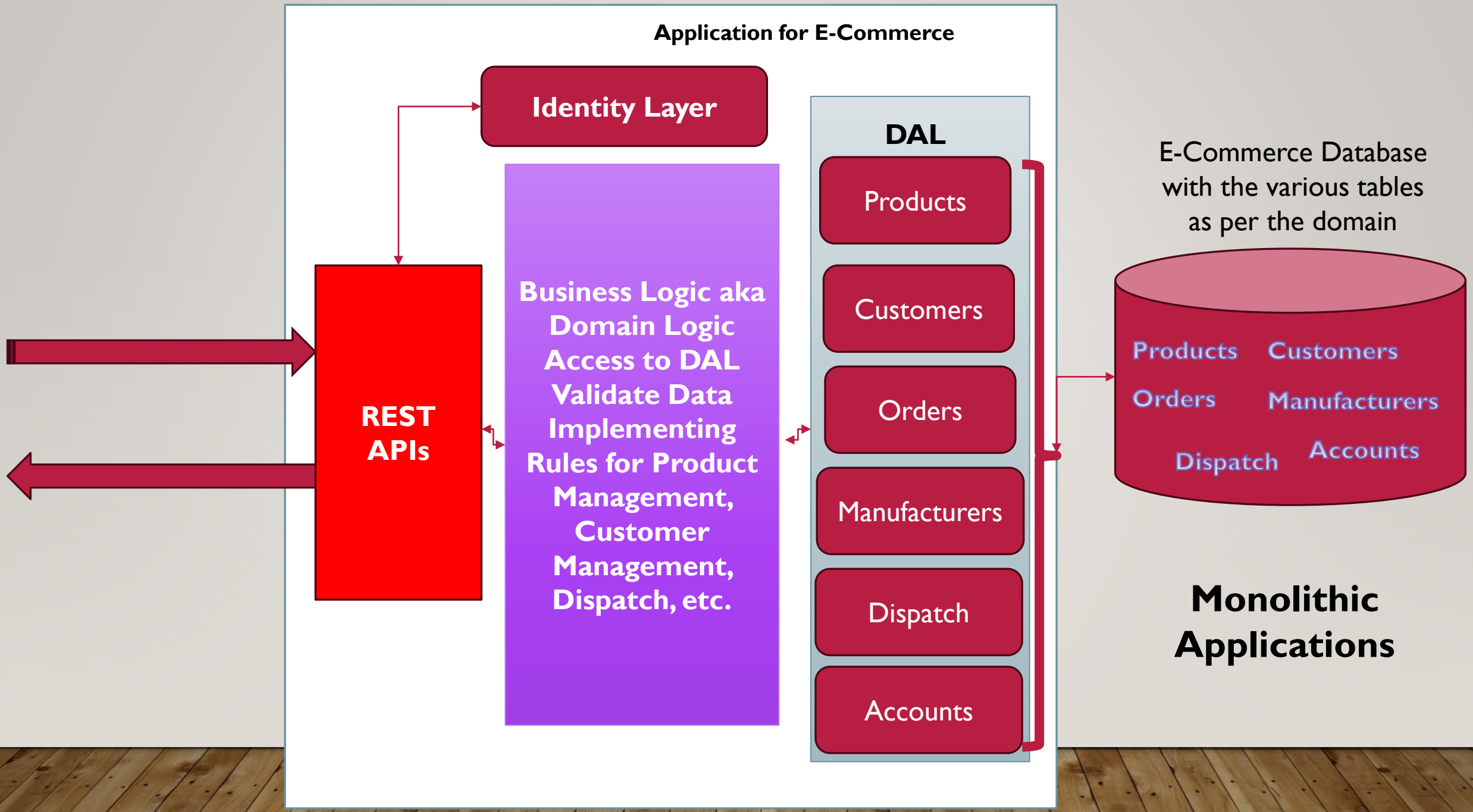
What is Microservice?

Isn't this just another service-oriented architecture (SOA) or domain-driven design (DDD) approach?

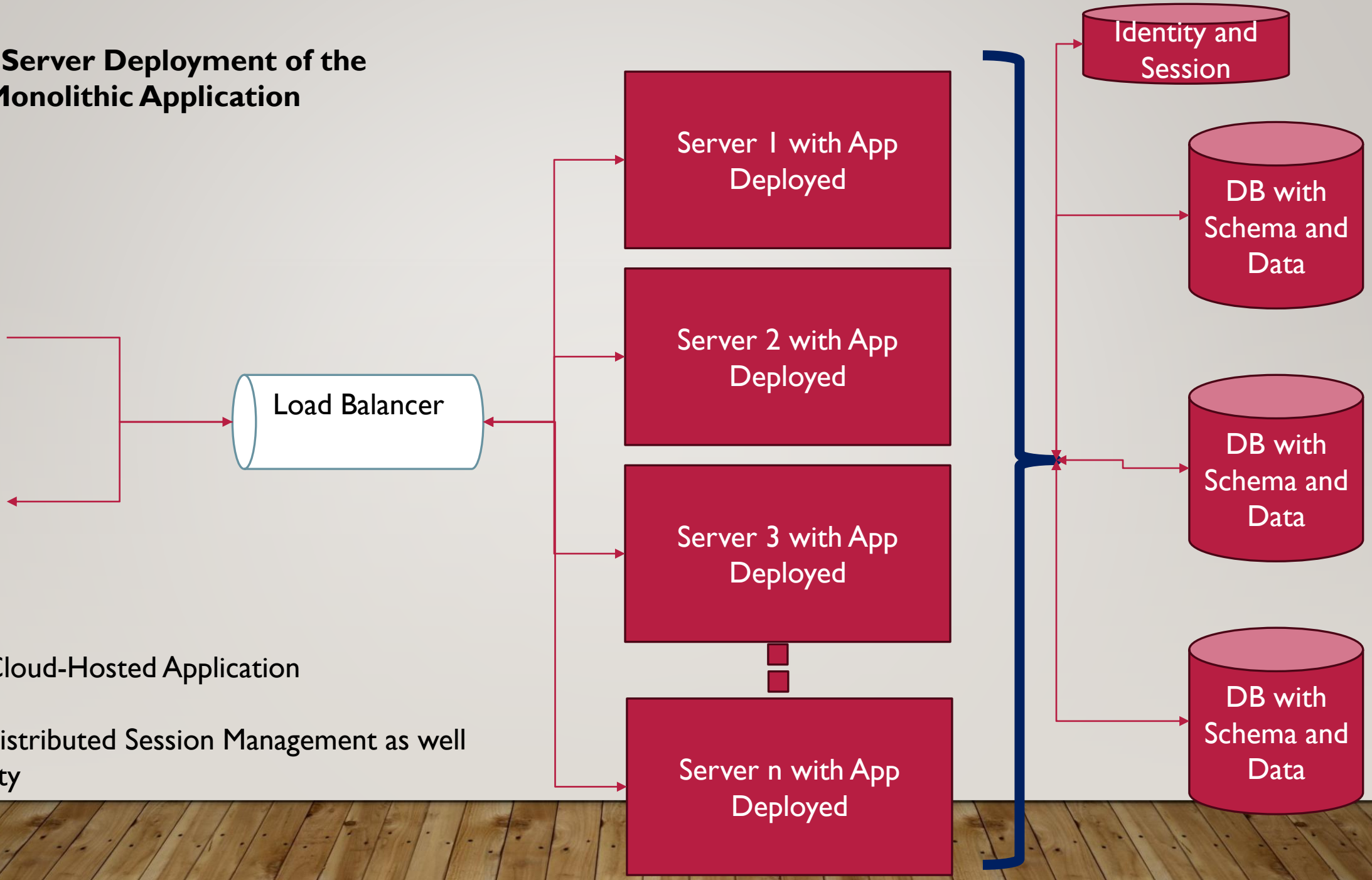
You can think of microservices as "SOA done right," with principles and patterns like autonomous services, Bounded-Context pattern and event-driven all having their roots in SOA and DDD.

WHAT ARE MICROSERVICES?

- The microservice architectural style is an approach to developing a single application as a **suite of small services**, each **running in its own process** and communicating with lightweight mechanisms, often an HTTP resource API. These services are **built around business capabilities** and **independently deployable** by fully automated deployment machinery. There is a **bare minimum of centralized management** of these services, which may be written in different programming languages and use different data storage technologies.
- -- James Lewis and Martin Fowler (2014)


Application for E-Commerce
Identity Layer
REST APIs
Business Logic aka Domain Logic Access to DAL Validate Data Implementing Rules for Product Management, Customer Management, Dispatch, etc.
DAL
Products
Customers
Orders
Manufacturers
Dispatch
Accounts
E-Commerce Database with the various tables as per the domain
Products
Customers
Orders
Manufacturers
Dispatch
Accounts
Monolithic Applications

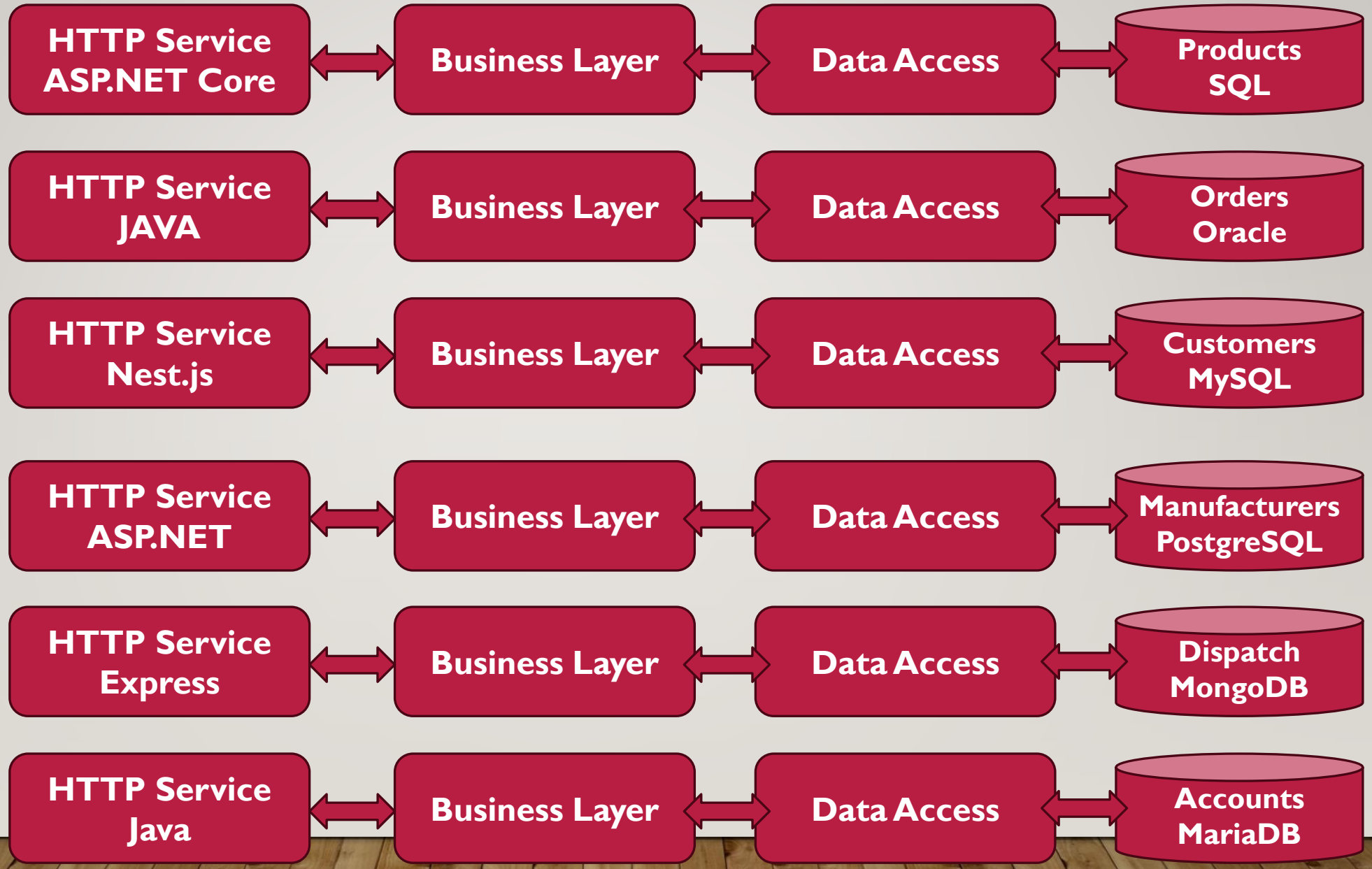
Multi-Server Deployment of the Monolithic Application



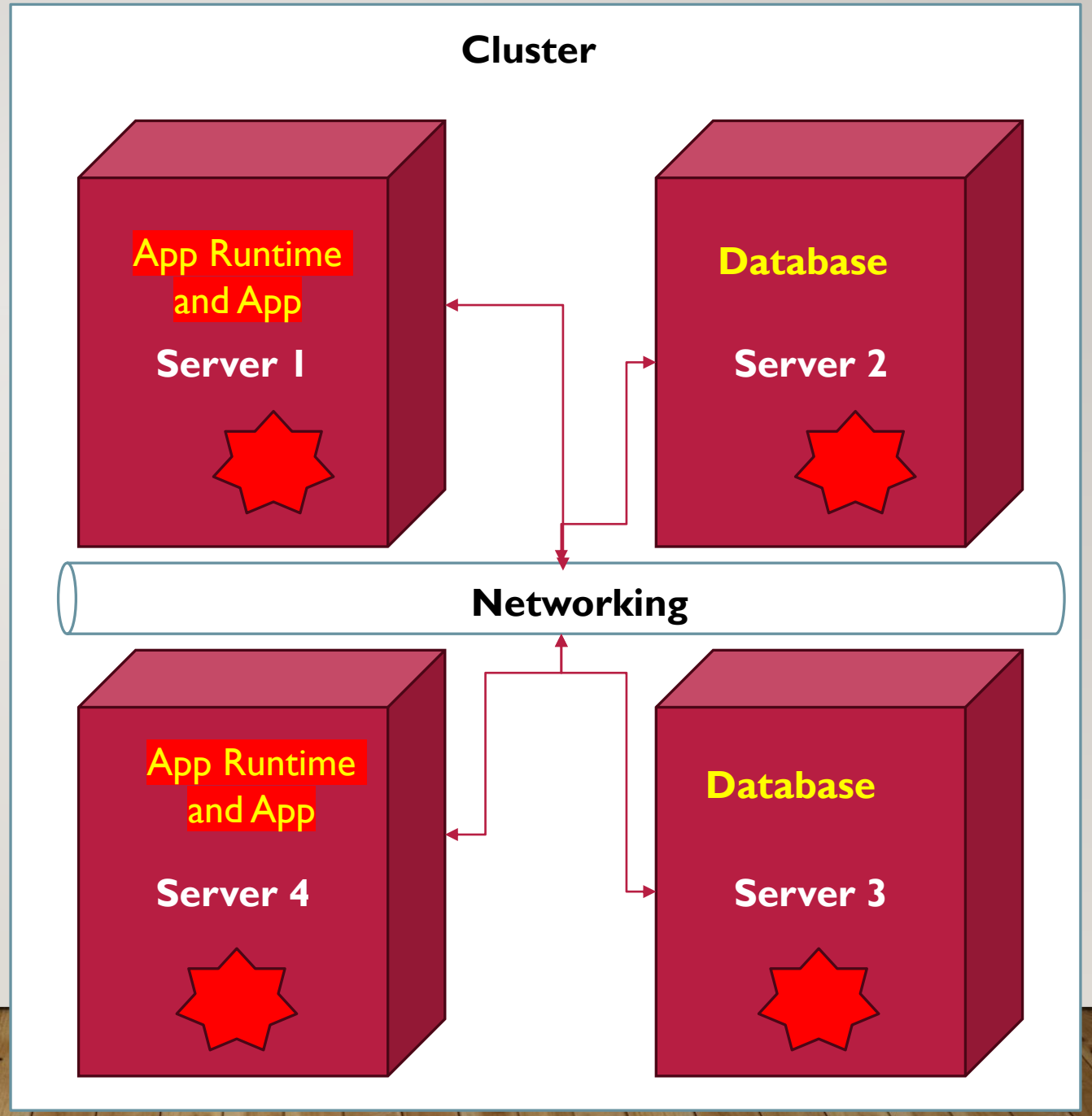
Private Cloud-Hosted Application

Needs Distributed Session Management as well as Security

Separate Each Workflow as independent Application



1. Multi-Server Deployment
    1. Each Server MUST be configured separately
    2. Each server MUST have an App deployed on it
    3. App Dependencies MUST be configured on each Server
2. State of the user (if needed) MSUT be saved in distributed storage

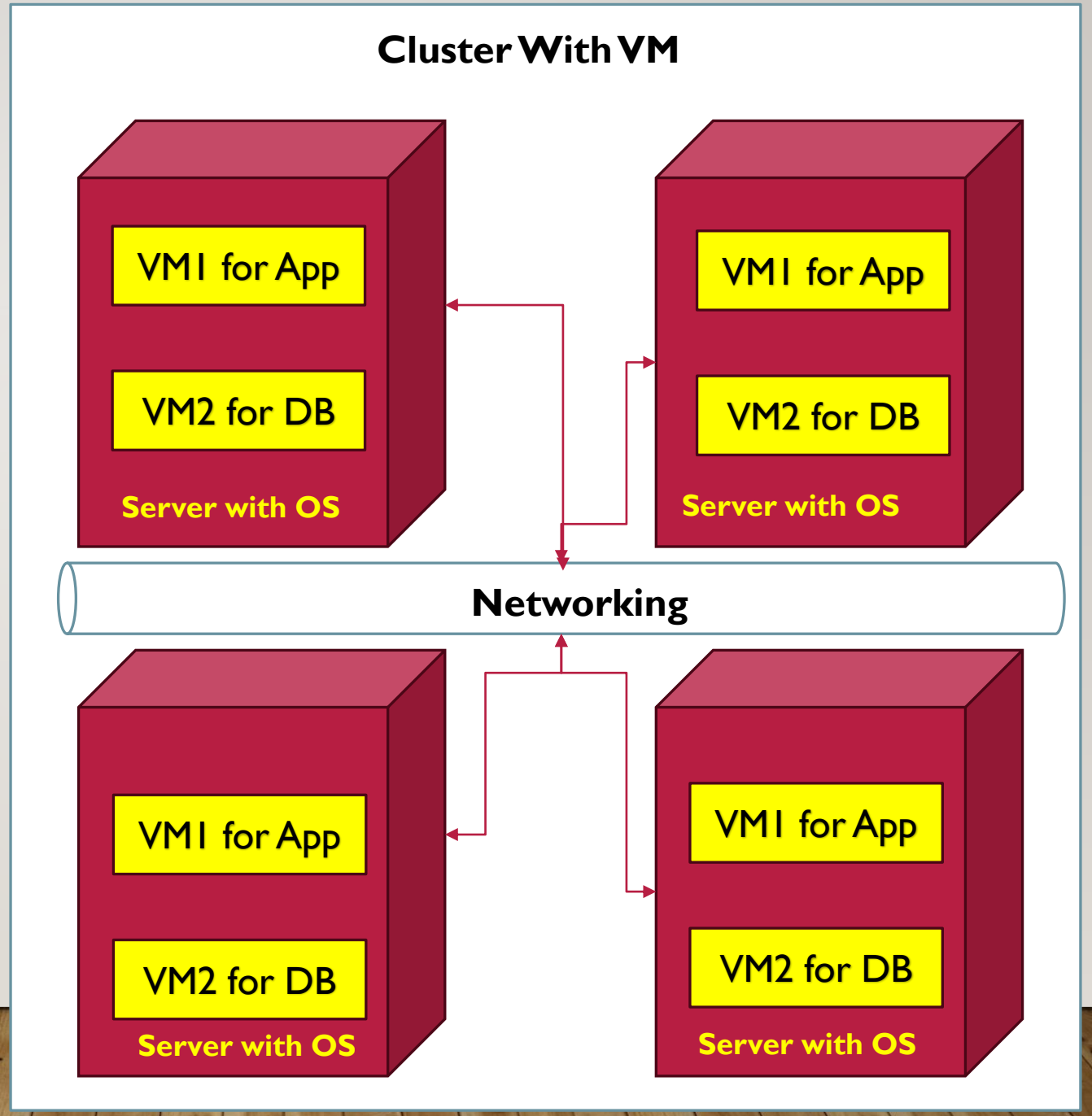


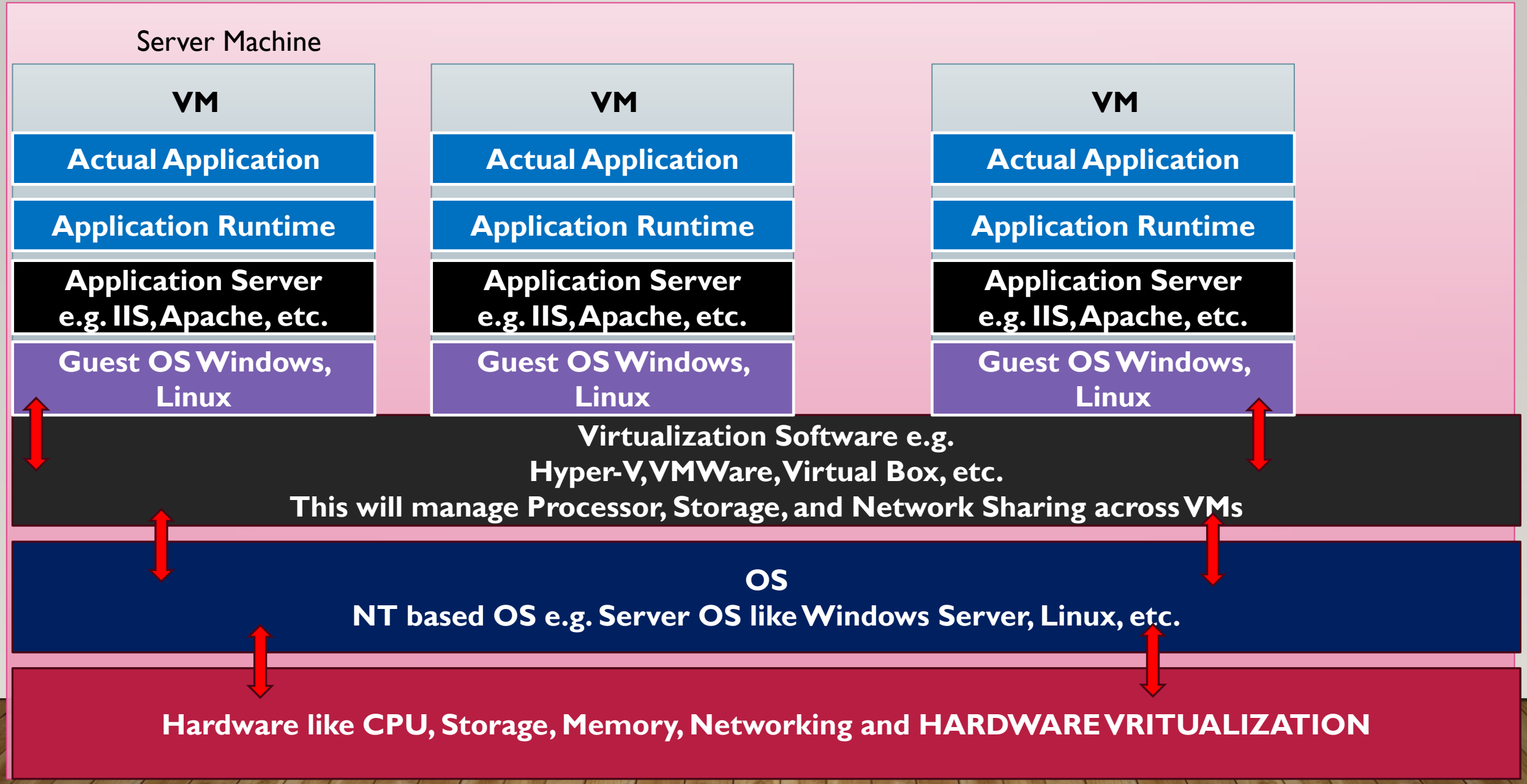
Generally, this is used by Cloud Enables Applications

Each Server MUST have Host OS and It MUST have Virtualization Software to run VM

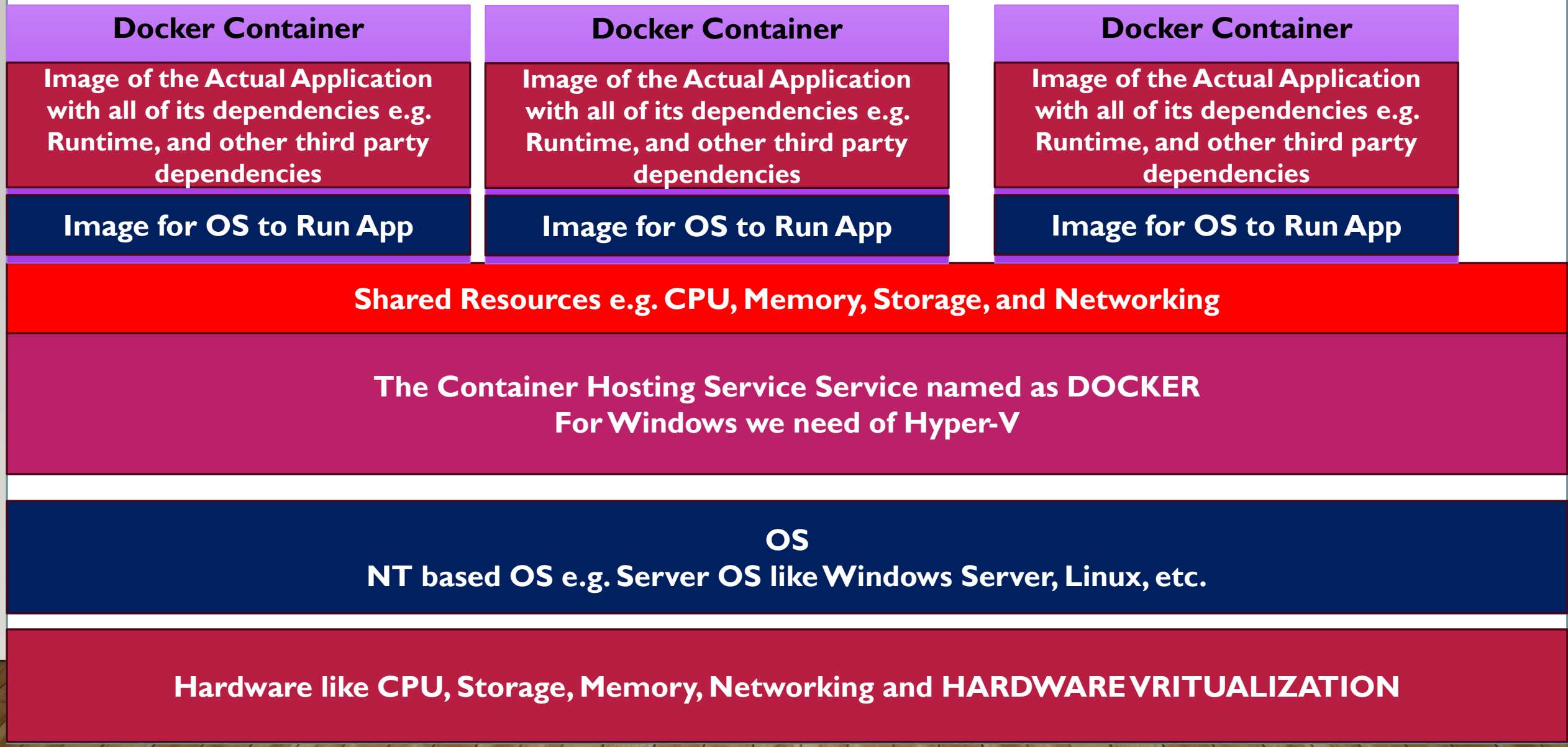
VM MUST have

- Guest OS
- Application Runtime (If VM for App)
  - Actual App
- Database (if VM for DB)




Server Machine
VM
Actual Application
Application Runtime
Application Server
e.g. IIS, Apache, etc.
Guest OS Windows,
Linux
VM
Actual Application
Application Runtime
Application Server
e.g. IIS, Apache, etc.
Guest OS Windows,
Linux
VM
Actual Application
Application Runtime
Application Server
e.g. IIS, Apache, etc.
Guest OS Windows,
Linux
Virtualization Software e.g.
Hyper-V, VMWare, Virtual Box, etc.
This will manage Processor, Storage, and Network Sharing across VMs
OS
NT based OS e.g. Server OS like Windows Server, Linux, etc.
Hardware like CPU, Storage, Memory, Networking and HARDWARE VRITUALIZATION

The Containerization on the Physical machine (Windows, Linux, macOS)



WHAT ARE MICROSERVICES?
…….SOME TECHNICAL SPECIFICATIONS

Autonomous	Isolated	Elastic	Resilient
Responsive	Automated	Programmable	Message Oriented
Intelligent	Configurable	Discoverable	Domain driven

WHAT ARE MICROSERVICES?
…….SOME BENEFITS

Evolutionary	Owned	Small	Safe
Versioned	Replaceable	Scale Governance	Deployment Governance
Reuse	Speed of Development	Resilient	Open

WHAT ARE MICROSERVICES?
…….SOME CHALLENGES

Evolutionary	Discoverability	Testing	Domain Modeling
Versioning must be Supported	Platform Matters	Automation is not an Option	Communication is Key

MICROSERVICES DESIGN SOME THOUGHTS

THINKING FOR DOMAIN MODELING IN EXISTING SYSTEM

Bounded Context	• Find the capability within the system - Where does the language change?
Rate of Change	• How fast or slow areas of the system change can identify a seam
Team Structure	• Service boundaries defined by how the org is structured
Pain	• Whatever hurts most is a good candidate
Namespace Modeling	• Partition by existing coupling

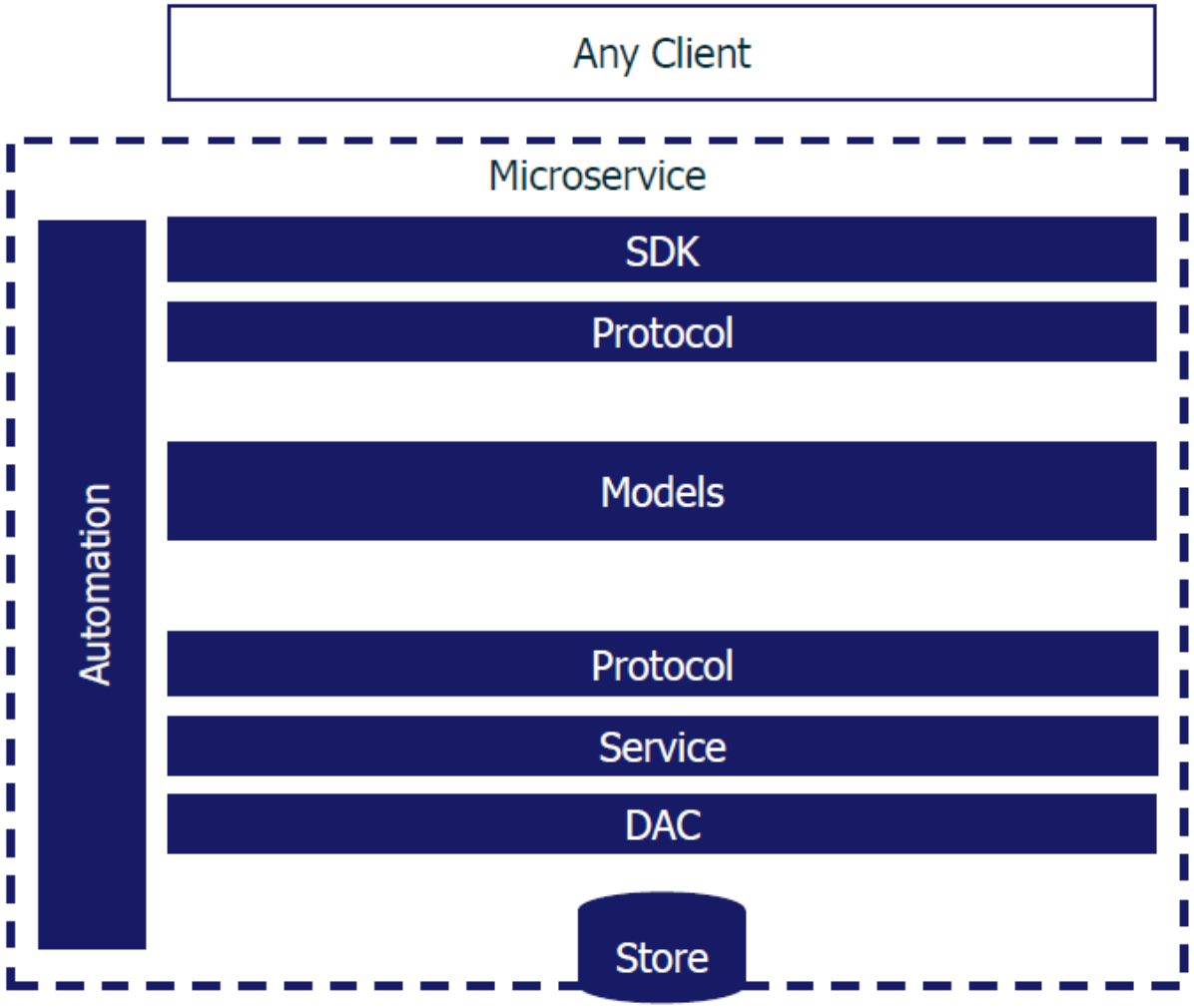
GOING FOR NEW DESIGN -- DOMAIN MODELING – GREEN FIELD

User Stories	• Leverage User Journey to find the capabilities
Evolutionary	• One service at a time
Single Responsibility	• High Cohesion
Loose Coupling	• Minimize dependencies

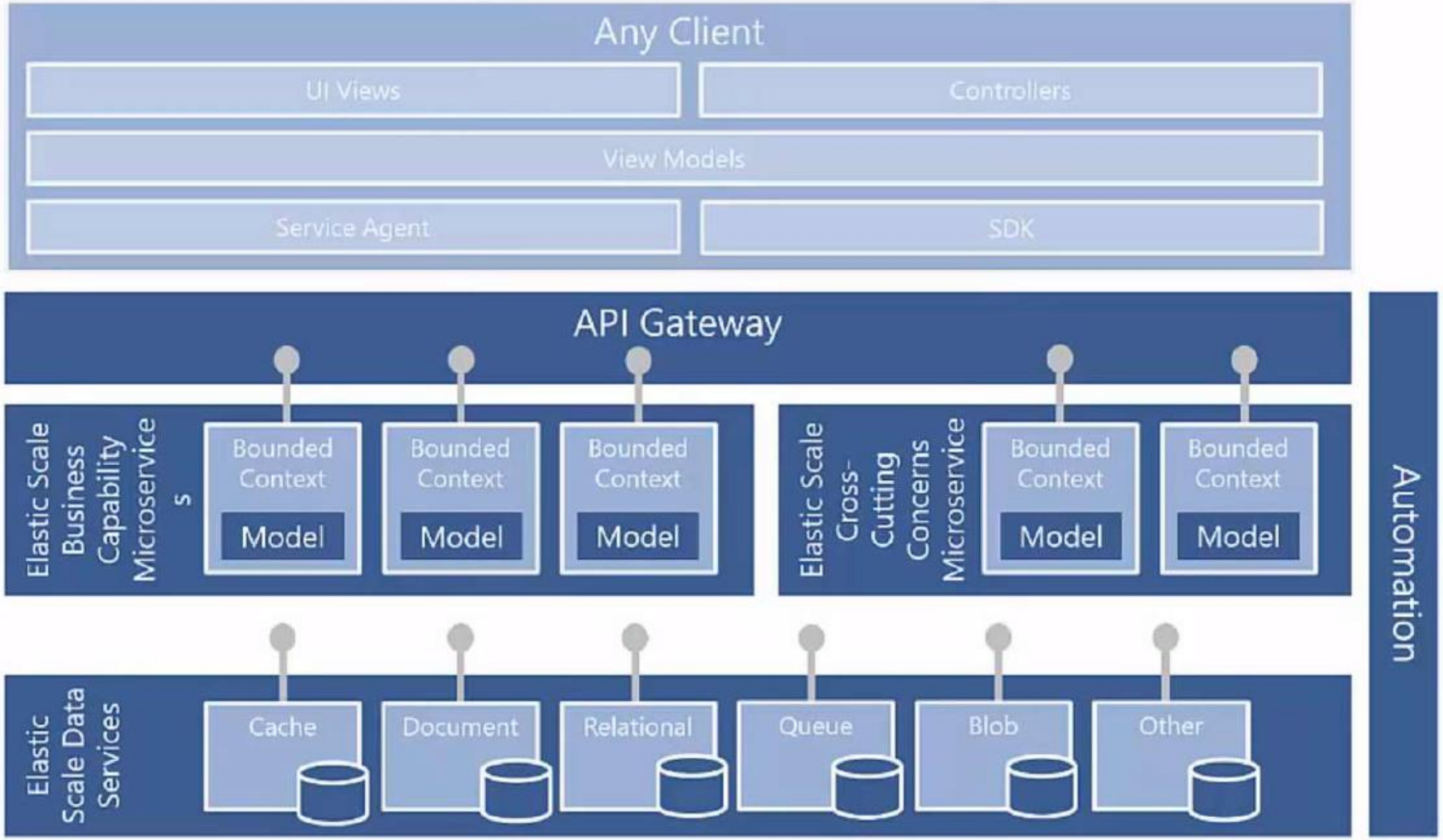
MICROSERVICES LOGICAL ARCHITECTURE FOR DOMAIN DRIVEN DEVELOPMENT

LOGICAL ARCHITECTURE

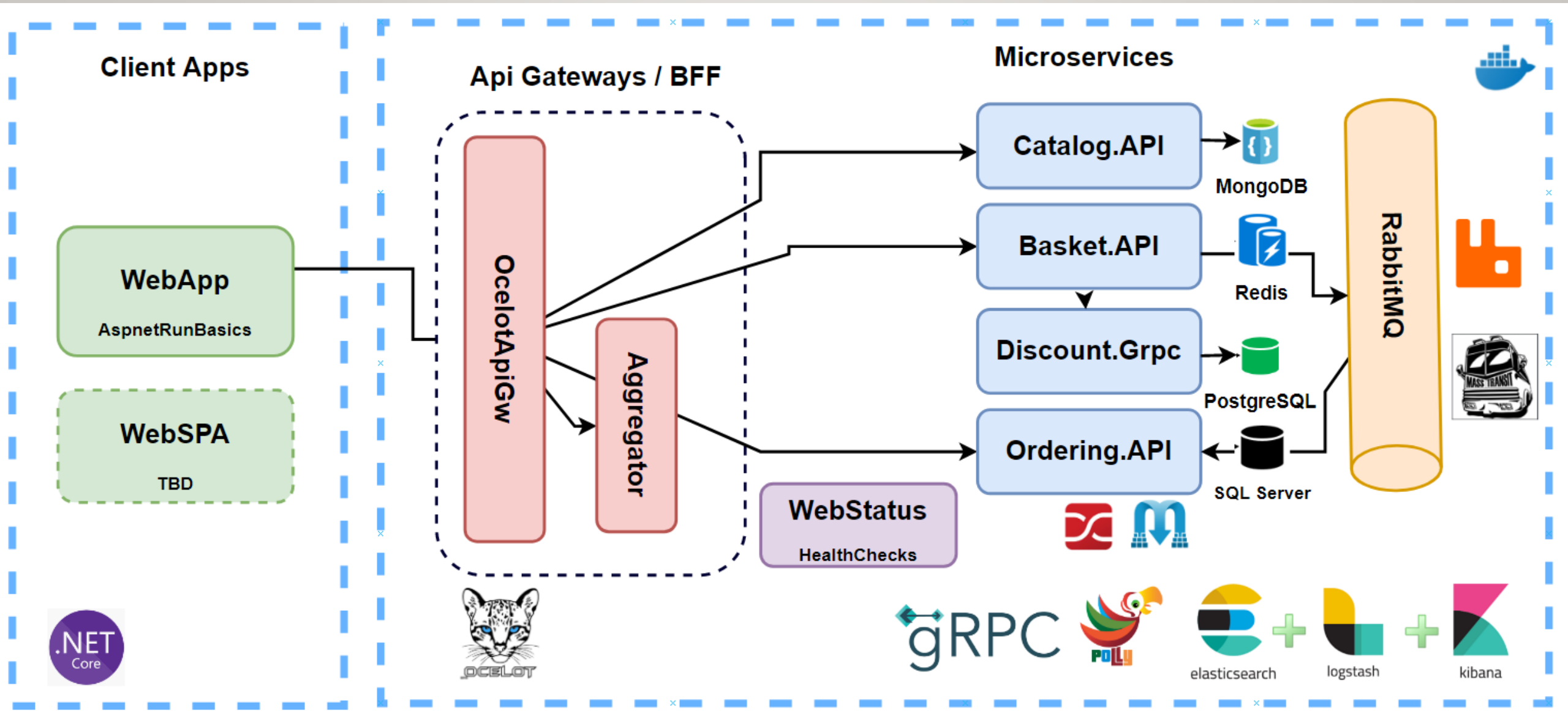
Experience	• Any client
SDK	• SDK for accessing service
Protocol	• Communication protocol such as REST/HTTP
Models	• On the wire data models
Service	• Business Rules/Logic
DAC	• Data Access
Store	• Model persistent store
Automation	• Continuous Delivery



SERVER-SIDE DISCOVERY PATTERN



ASP.NET CORE MICROSERVICES



https://github.com/aspnetrun/run-aspnetcore-microservices

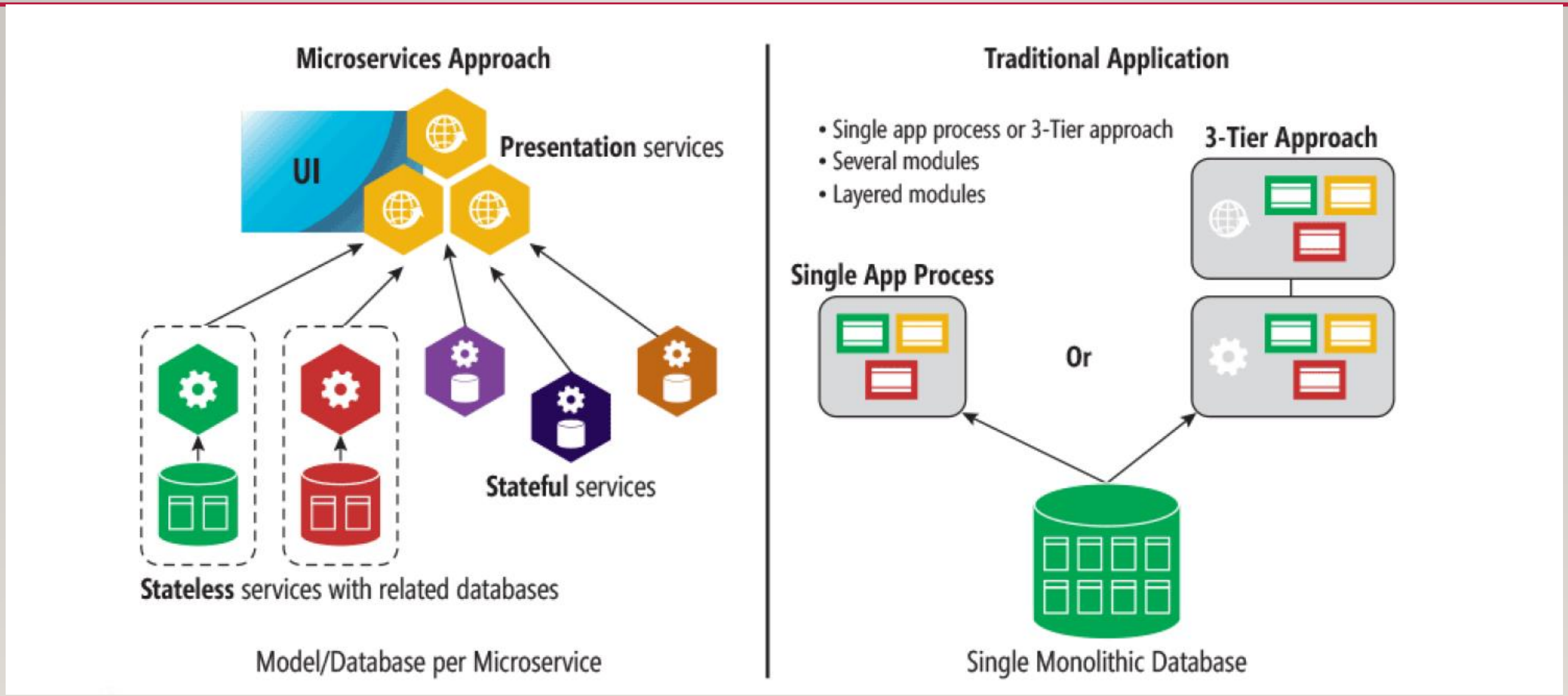
THE 'STATE' IN MICROSERVICE

- What is State means in case of the Microservices?
- The data
  - Send to Microservices
  - Data processed and Generated by Microservices during processing
- Case to maintain the state
  - Microservices needs to maintain the data for high data availability
  - Data processing performance improvements
  - Data Consistency
  - Transaction Management
  - Data Reliability

THE 'STATE' IN MICROSERVICE

- Stateless Microservices
  - They need an external data repository to maintain the state of the data
    - E.g. Cache, Message Queues, File System, Database etc.
  - The data is stored out of the process of the microservices
  - Need to arrange and configure external store explicitely
  - Challenges in reliability because of the external dependency
- Stateful Microservices
  - Data is maintained in the local process of the Microservice
  - The cluster manager is responsible to transfer the data state
  - Reliability and availability is controlled by the cluster manager
  - Cluster manager comes with additional cost

STATEFUL VS STATELESS VS MONOLITHIC APPLICATIONS



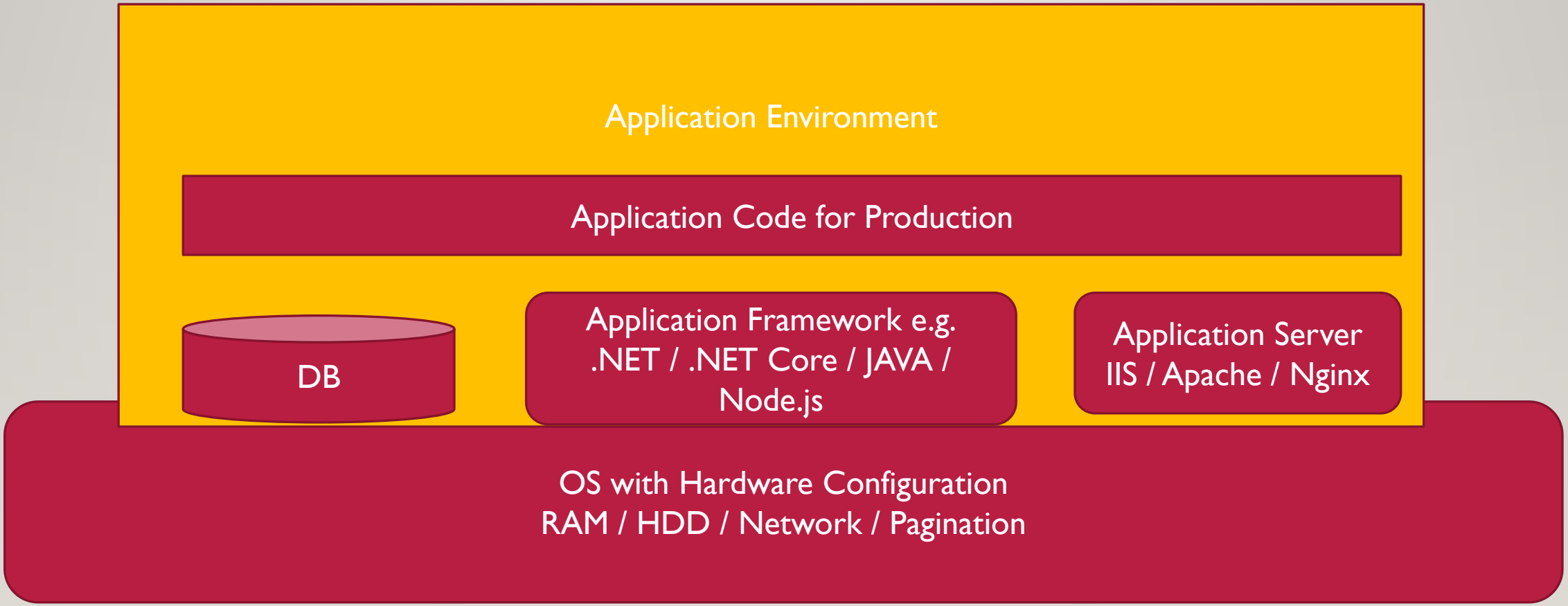
PROGRAMMING WITH MICROSERVICES

PROGRAMMING WITH MICROSERVICES

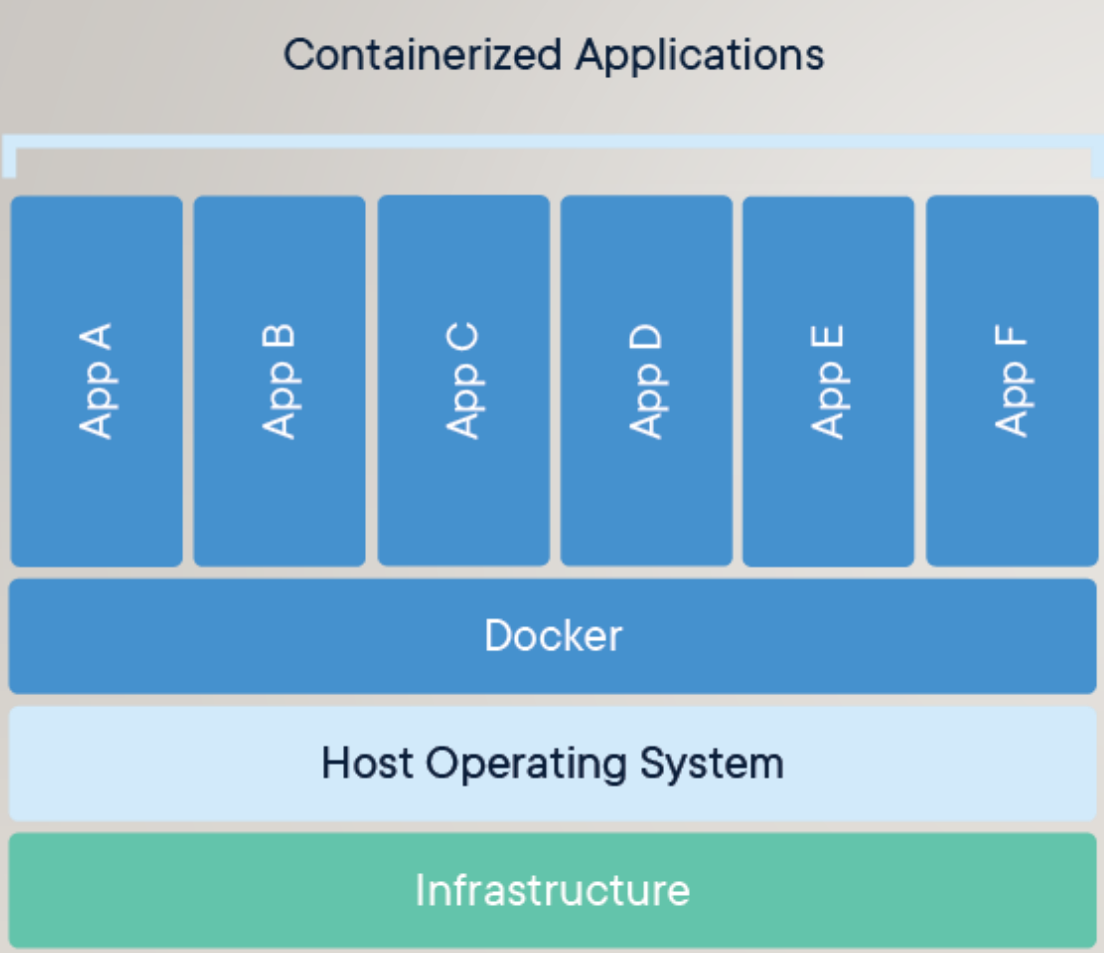
- Suitable Technologies for Modern Web Apps
  - ASP.NET Core
  - Node.js
  - Java
- Infrastructure Services
  - Rabbit MQ
  - MassTransit
  - Azure Service Bus
  - Apache Kafka
  - Other messaging Technologies
- Hosting and Management
  - Docker
  - Kubernetes
  - Service Fabric

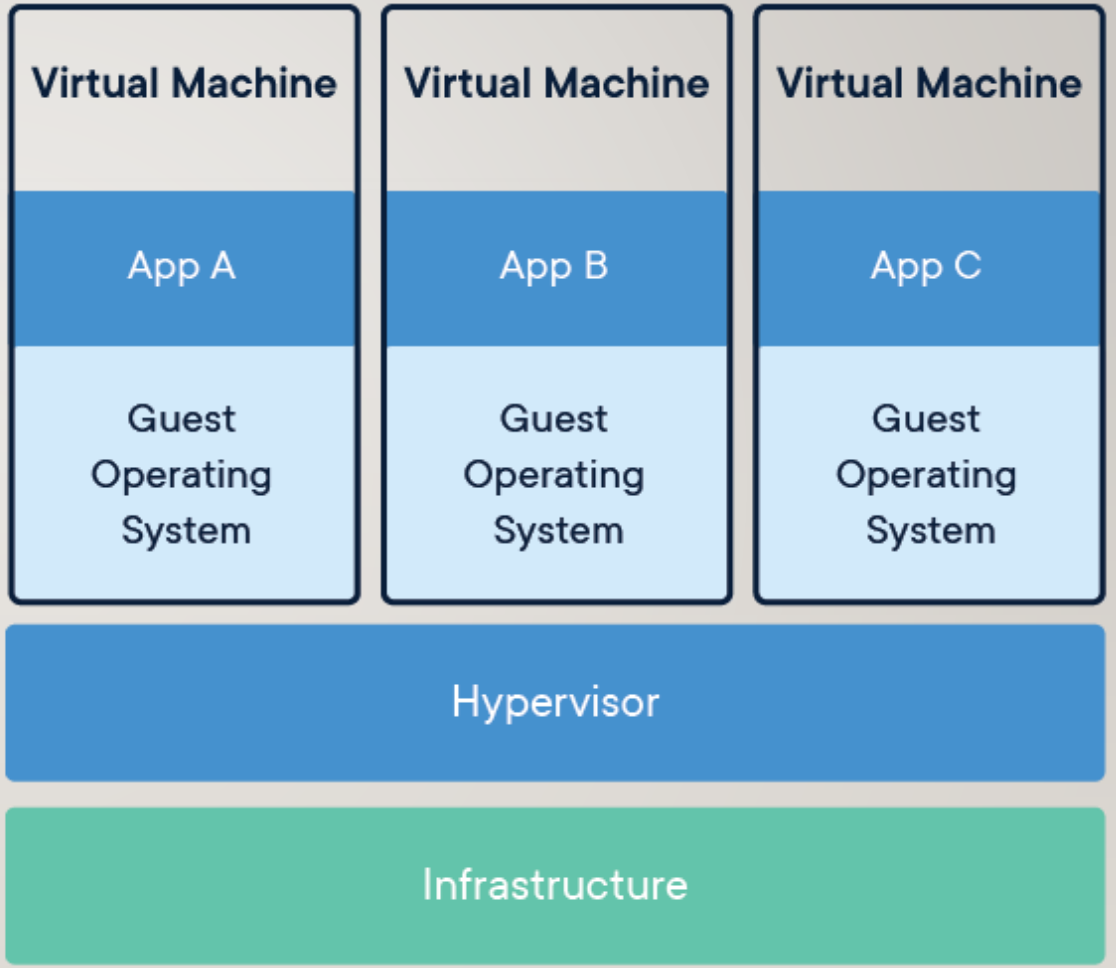
DOCKER

Virtual Env. For Customers



DOCKER


Containerized Applications
App A
App B
App C
App D
App E
App F
Docker
Host Operating System
Infrastructure


Virtual Machine
Virtual Machine
Virtual Machine
App A
App B
App C
Guest Operating System
Guest Operating System
Guest Operating System
Hypervisor
Infrastructure

```
FROM mcr.microsoft.com/dotnet/core/aspnet:3.1-buster-slim AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/core/sdk:3.1-buster AS build
WORKDIR /src
COPY Core_NewServie/Core_NewServie.csproj Core_NewServie/
RUN dotnet restore "Core_NewServie/Core_NewServie.csproj"
COPY . .
WORKDIR "/src/Core_NewServie"
RUN dotnet build "Core_NewServie.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "Core_NewServie.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "Core_NewServie.dll"]
```

Image from Docker Hub

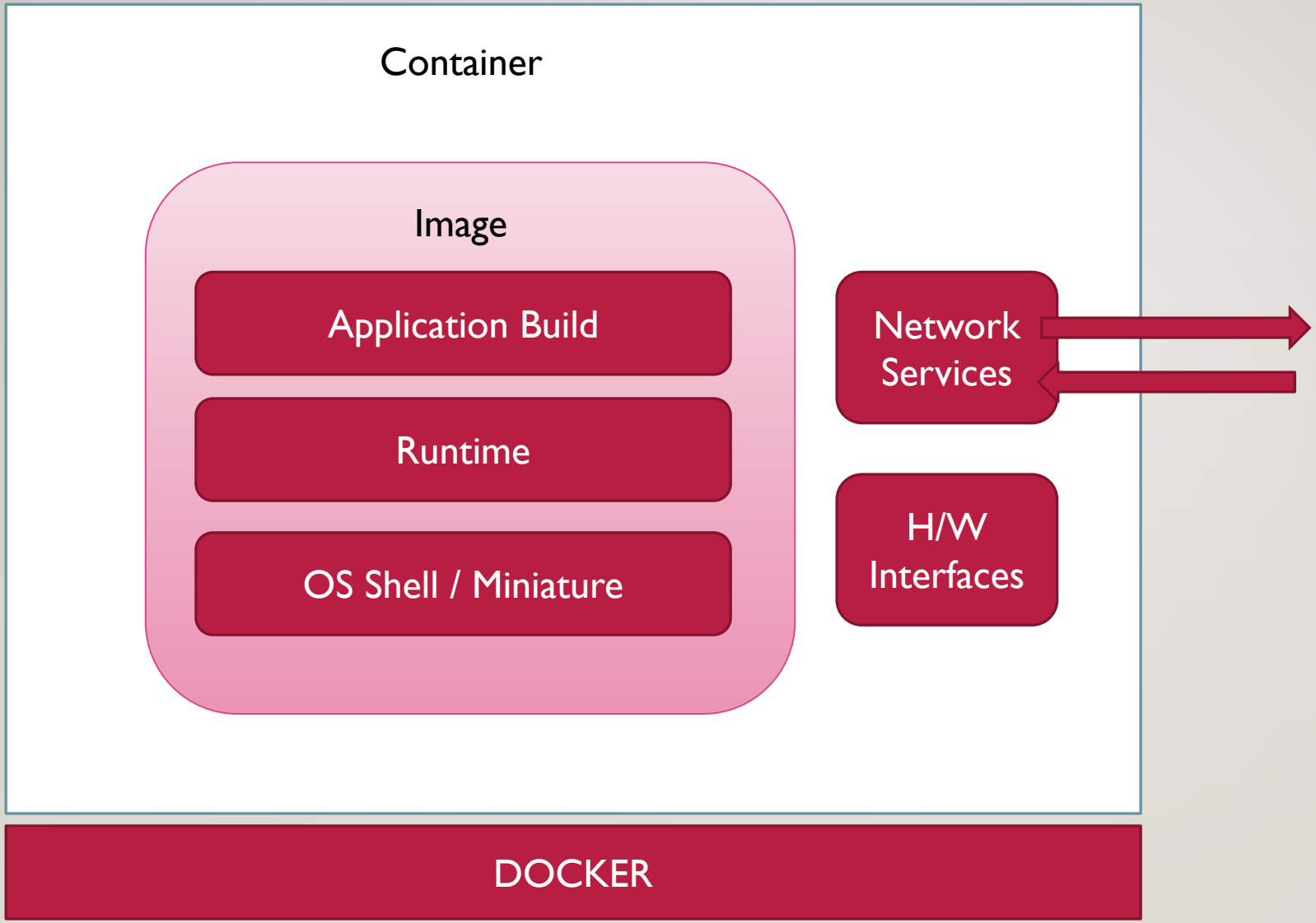
Folder on image for app files

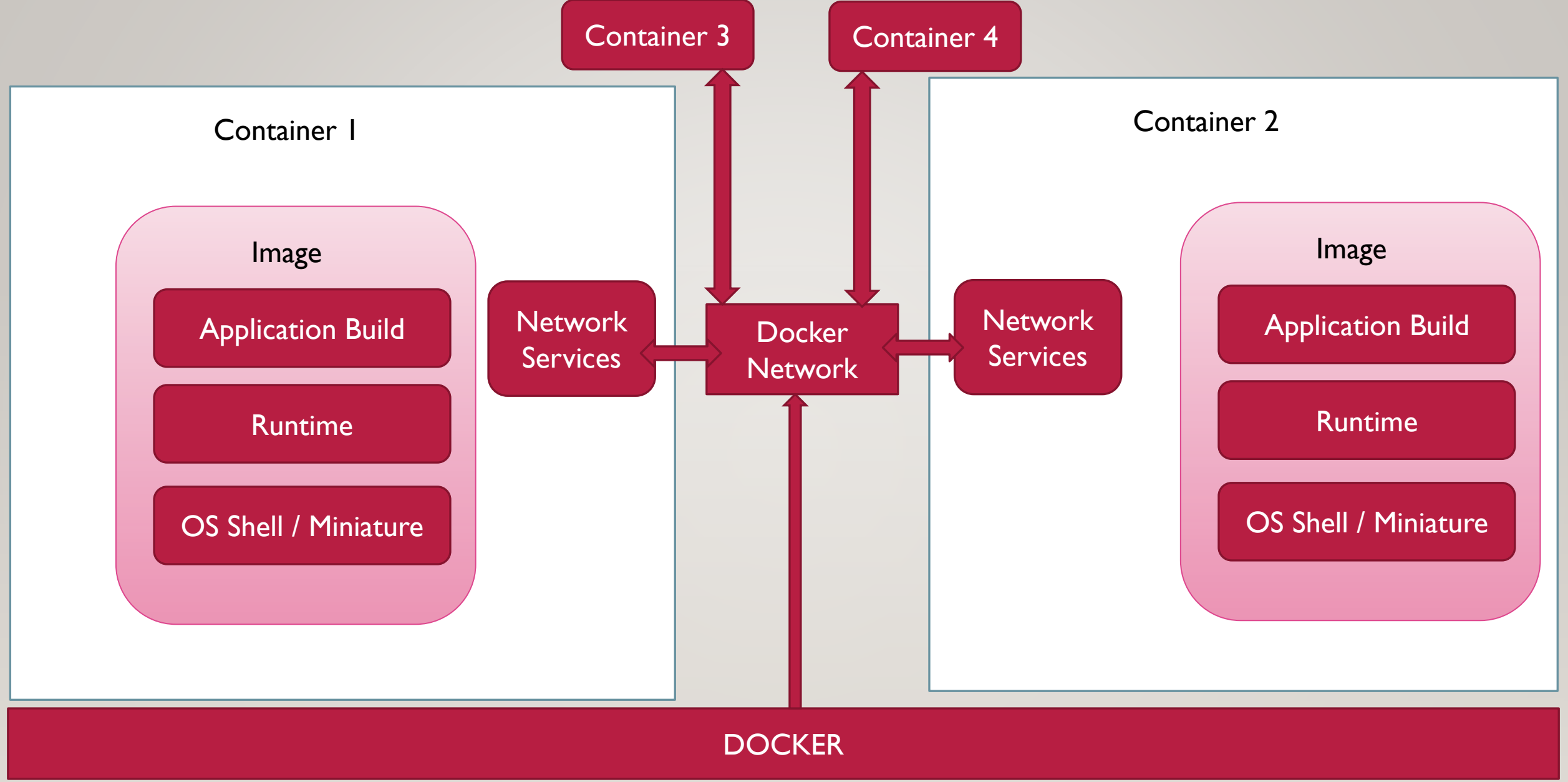
Port of container to access the app.

ASP.NET Core App build is created, create src folder for copying build file from dependencies specified in csproj file. Create a release build

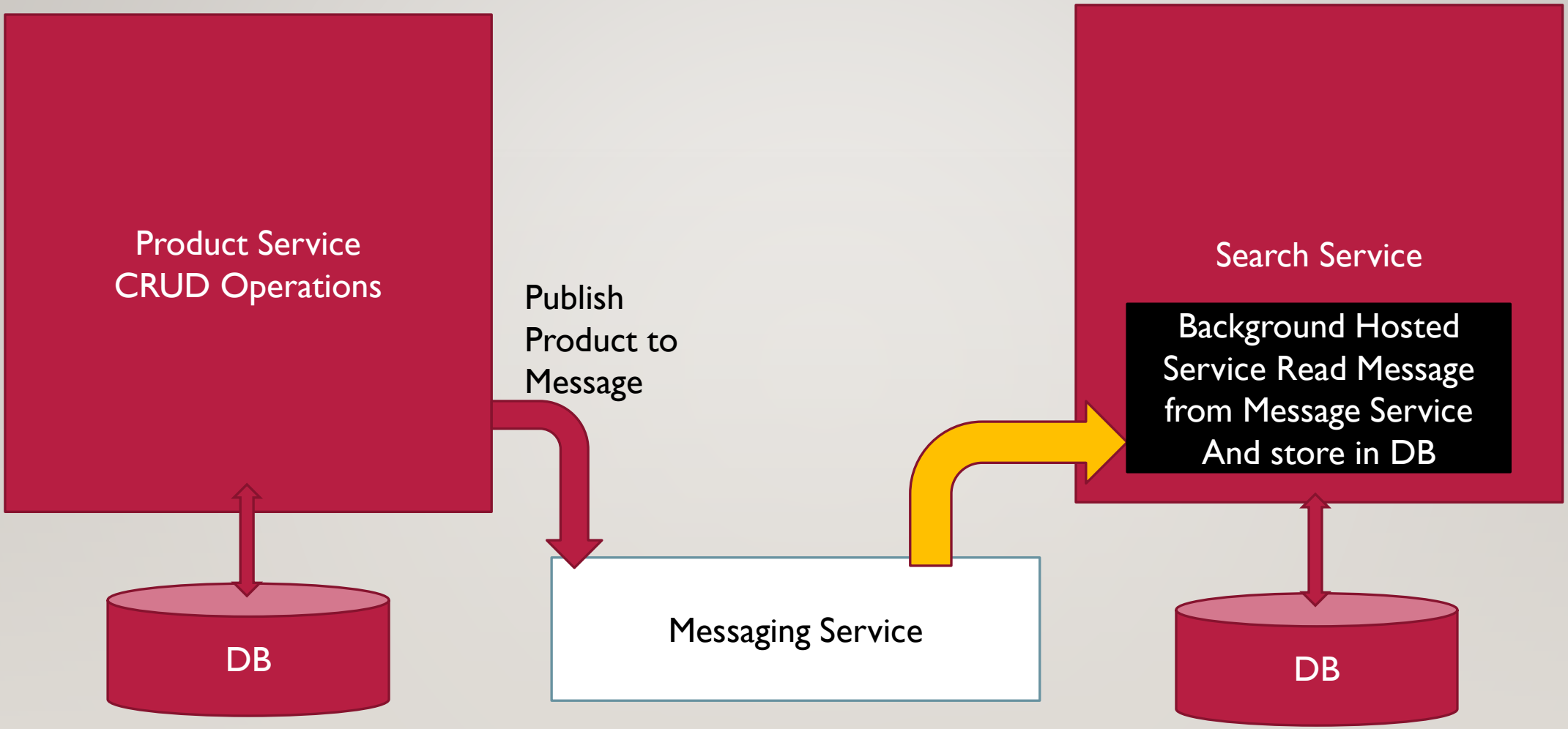
Create a public profile in Docker image, means download all nuget packages (interna; and external depednencies) and public application image

Image is successfully built and ASP.NET Core Runtime is hosting the application on Port (80/443) and entry-point is set so that users can access this application.


Container
Image
Application Build
Runtime
OS Shell / Miniature
Network Services
H/W Interfaces
DOCKER


Container 3
Container 4
Container 1
Container 2
Image
Application Build
Runtime
OS Shell / Miniature
Network Services
Docker Network
Network Services
Image
Application Build
Runtime
OS Shell / Miniature
DOCKER

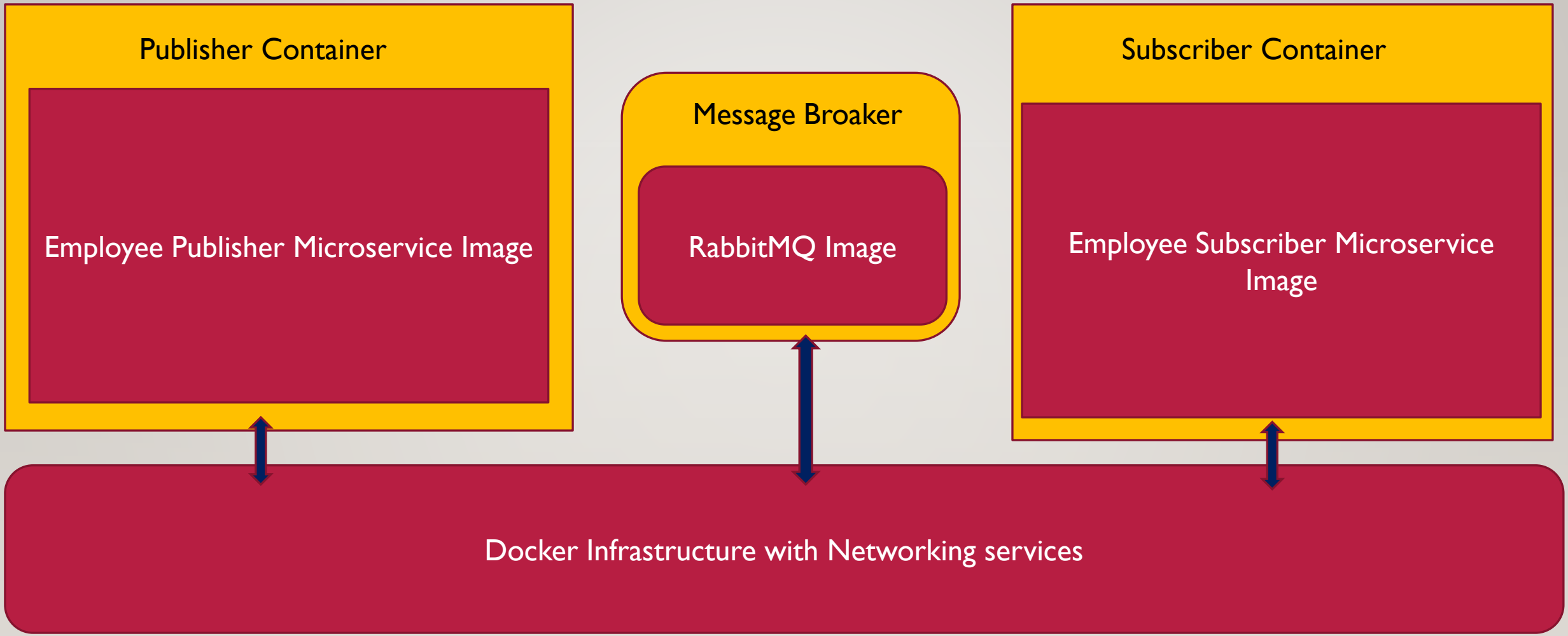
eShopping Microservices App



http://publisher:6001/api/EmployeeAPI

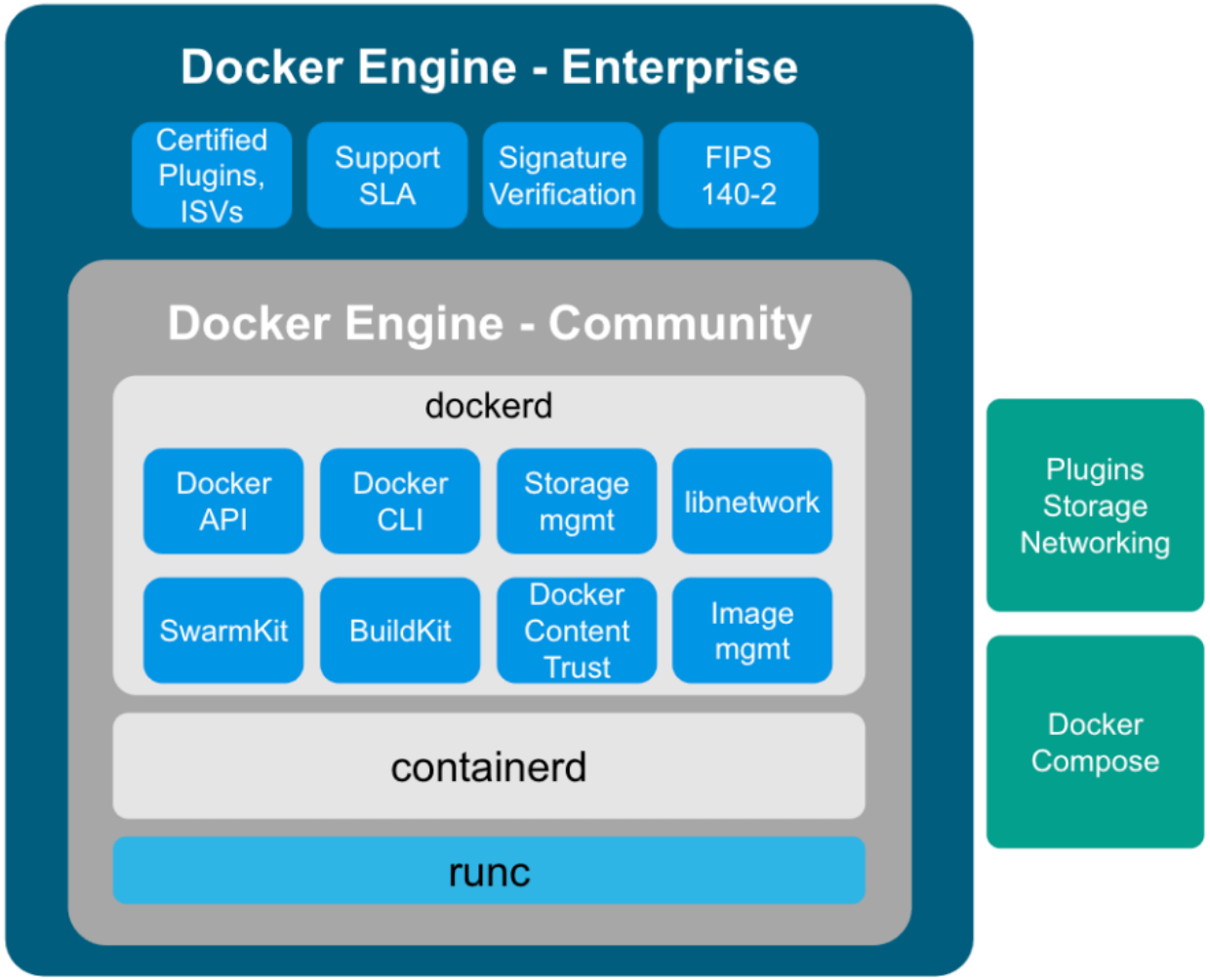
http://U:P@mb:5672

http://subscriber:6002/api/EmployeeAPI



```yaml
version: "3.6"
services:
  rabbitmq:
    container_name: emessage-rabbit
    ports:
      - 5672:5672
      - 15672:15672
    environment:
      - RABBITMQ_DEFAULT_USER=guest
      - RABBITMQ_DEFAULT_PASS=guest
    image: rabbitmq:3-management
  publisherapi:
    container_name: publisherapi
    ports:
      - 9001:80
    build: ASP_Net_CorePublisher
    image: netcorepublisher
    restart: on-failure
    depends_on:
      - rabbitmq
  subscriberapi:
    container_name: subscriberapi
    ports:
      - 9002:80
    build: ASP_Net_CoreSubscriber
    image: netcoresubscriber
    restart: on-failure
    depends_on:
      - rabbitmq
```

DOCKER


Docker Engine - Enterprise
Certified Plugins, ISVs
Support SLA
Signature Verification
FIPS 140-2
Docker Engine - Community
dockerd
Docker API
Docker CLI
Storage mgmt
libnetwork
SwarmKit
BuildKit
Docker Content Trust
Image mgmt
containerd
runc
Plugins Storage Networking
Docker Compose

CQRS PATTERN

CQRS

- CQRS stands for Command and Query Responsibility Segregation, a pattern that separates read and update operations for a data store. Implementing CQRS in your application can maximize its performance, scalability, and security. The flexibility created by migrating to CQRS allows a system to better evolve over time and prevents update commands from causing merge conflicts at the domain level.

OCELOT GATEWAY

THE GATEWAY ROUTING PATTERN

- Gateway Routing pattern main objective is Route requests to multiple services using a single endpoint. This pattern is useful when you wish to expose multiple services on a single endpoint and route to the appropriate service based on the request.

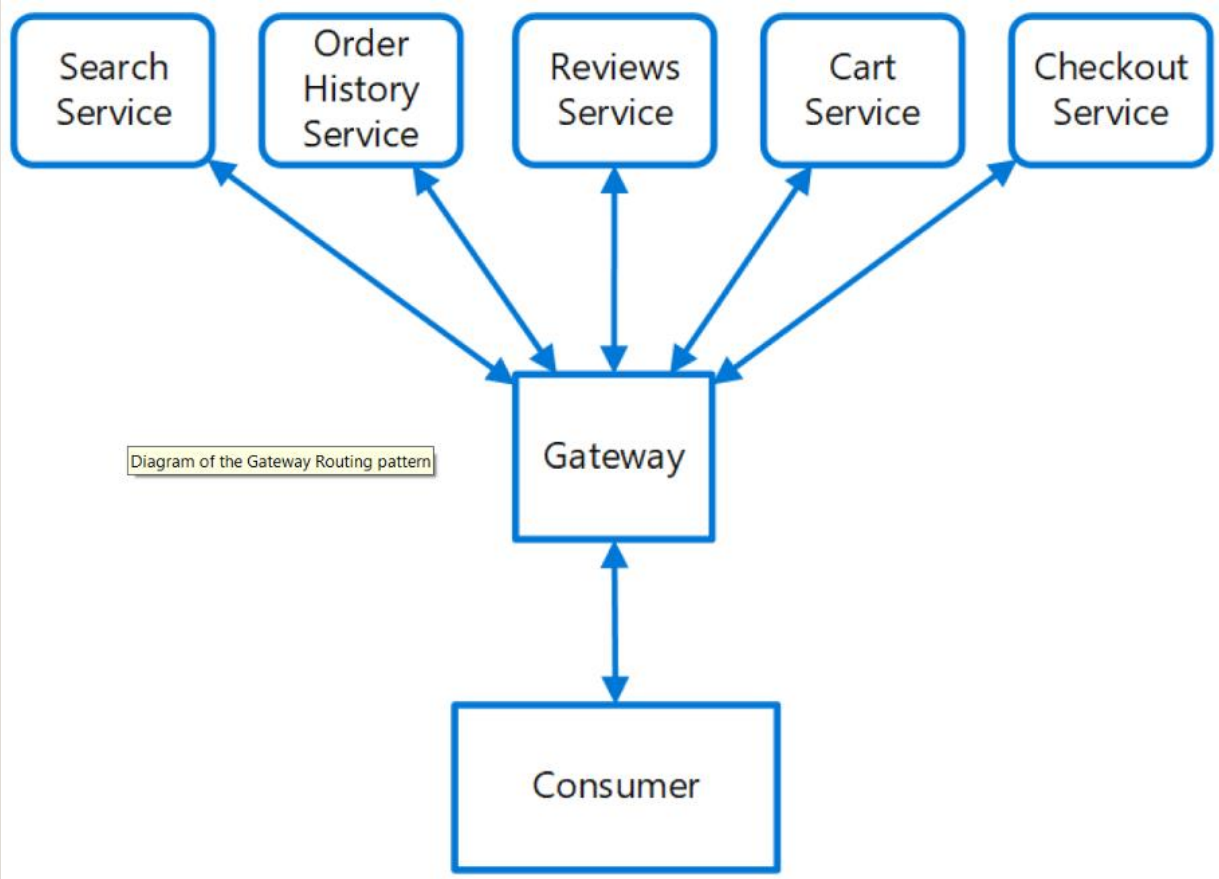
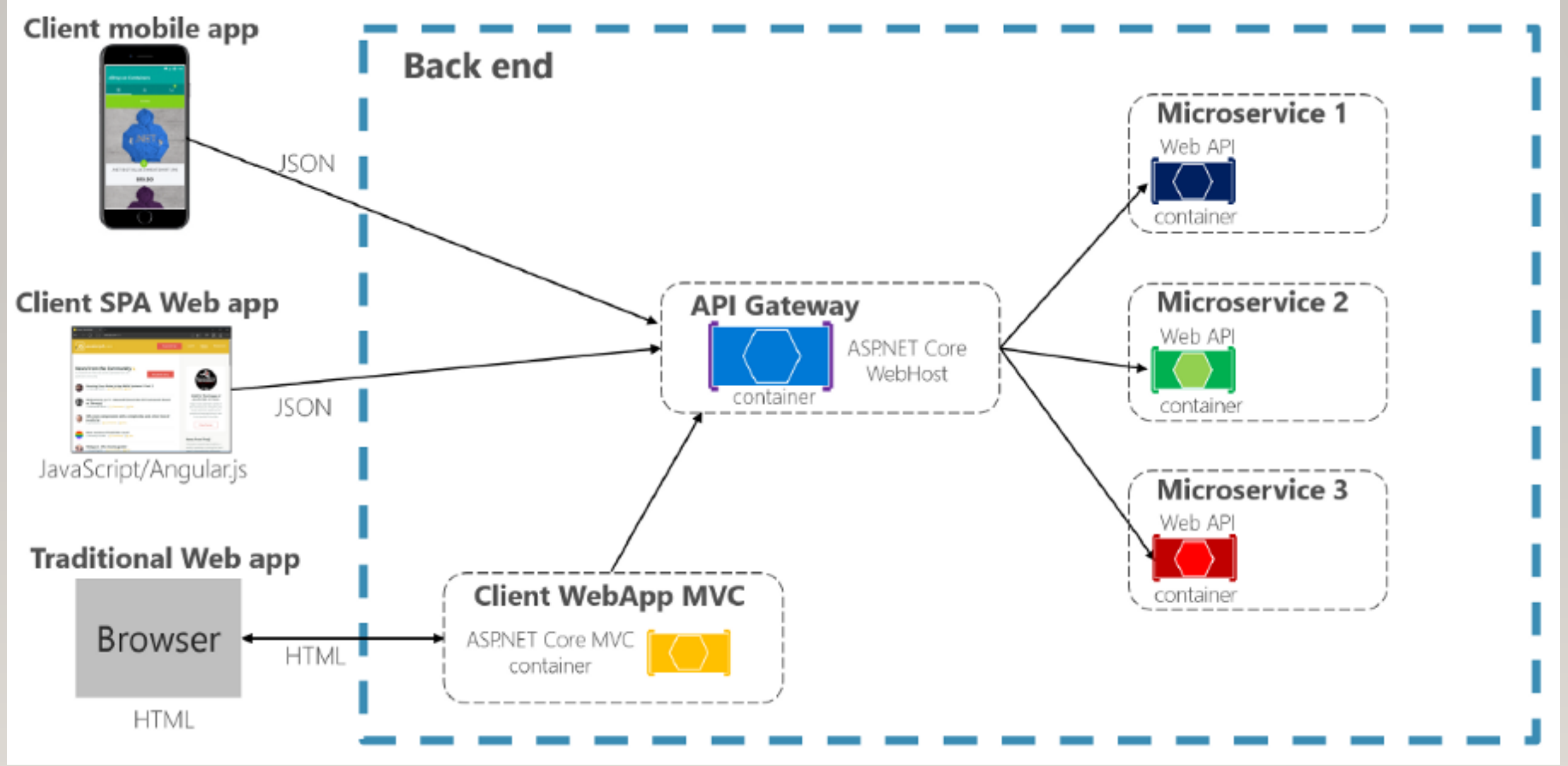
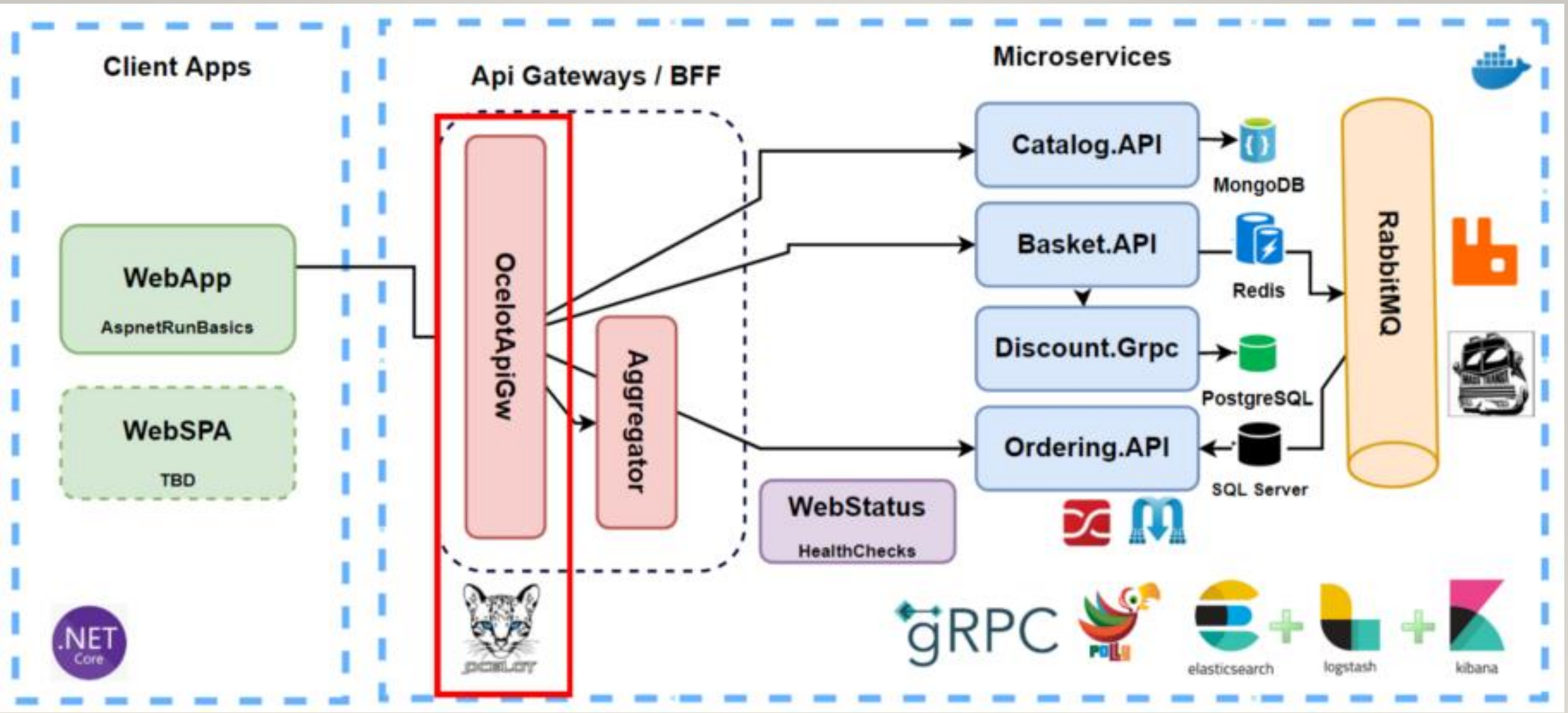


Diagram of the Gateway Routing pattern

Building Ocelot API Gateway Microservices with ASP.NET Core and Docker Container | aspnetrun (medium.com)



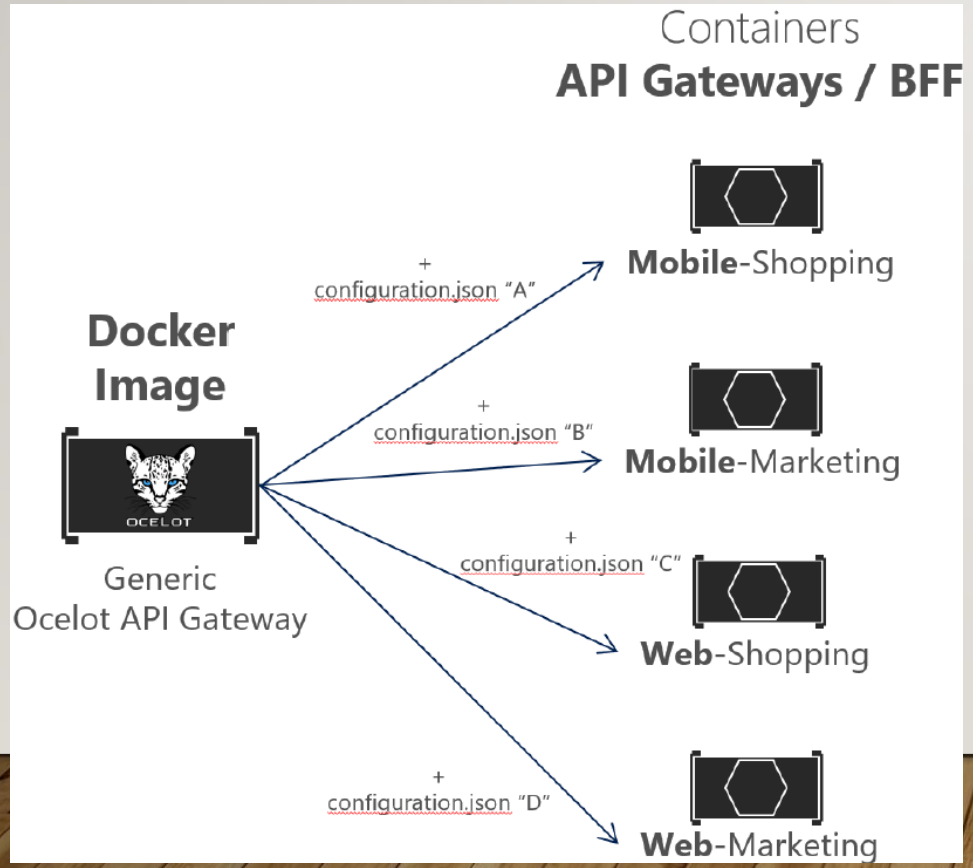
Building Ocelot API Gateway Microservices with ASP.NET Core and Docker Container | aspnetrun (medium.com)



Building Ocelot API Gateway Microservices with ASP.NET Core and Docker Container | aspnetrun (medium.com)

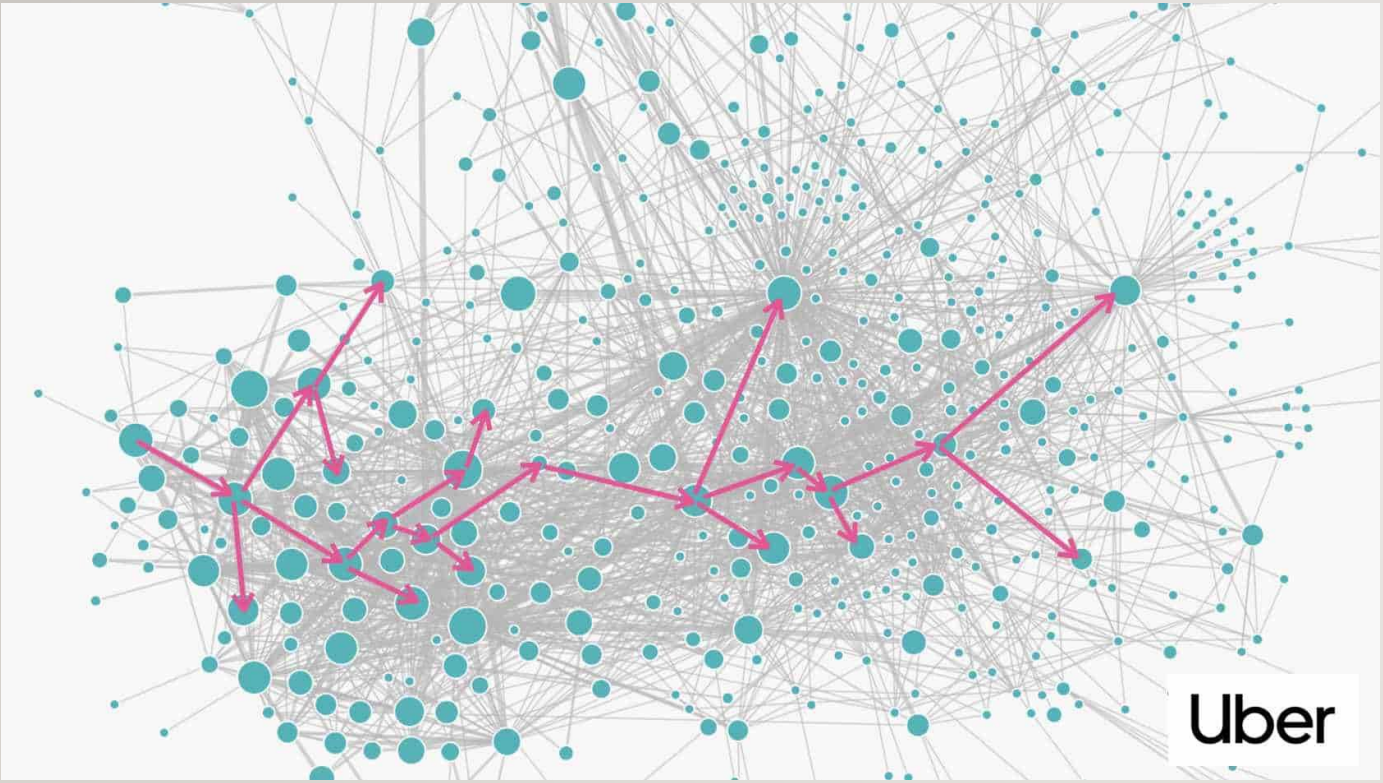
OCELOT API GATEWAY

- Ocelot is basically a set of middlewares that you can apply in a specific order.
- Ocelot is a lightweight API Gateway, recommended for simpler approaches. Ocelot is an Open Source .NET Core-based API Gateway especially made for microservices architectures that need unified points of entry into their systems. It's lightweight, fast, and scalable and provides routing and authentication among many other features.



MICROSERVICE RESILIENCY

RESILIENCY



Identifying Failure Points

If you have been troubleshooting your application and individual service performance, you have already likely identified a few services that either receive or send a lot of requests.

Optimizing those requests is important and can help to prolong availability. But, given a high enough load, the services sending or receiving those requests are likely failure points for your application.

For enterprise microservices applications like Uber, where engineers are using thousands upon thousands of microservices, tracing requests across these services can be hopelessly complex – with traces that have hundreds of thousands of spans.

https://www.jrebel.com/blog/microservices-resilience-patterns

CIRCUIT BREAKER PATTERN

- Microservices applications often rely on remote resources, like third-party services, as a key component of their program.
- But what happens when one of those remote resources times out upon request? Does your microservice continue calling that resource in an endless loop until it fulfills that request?
- What happens when multiple services are requesting that same remote resource?

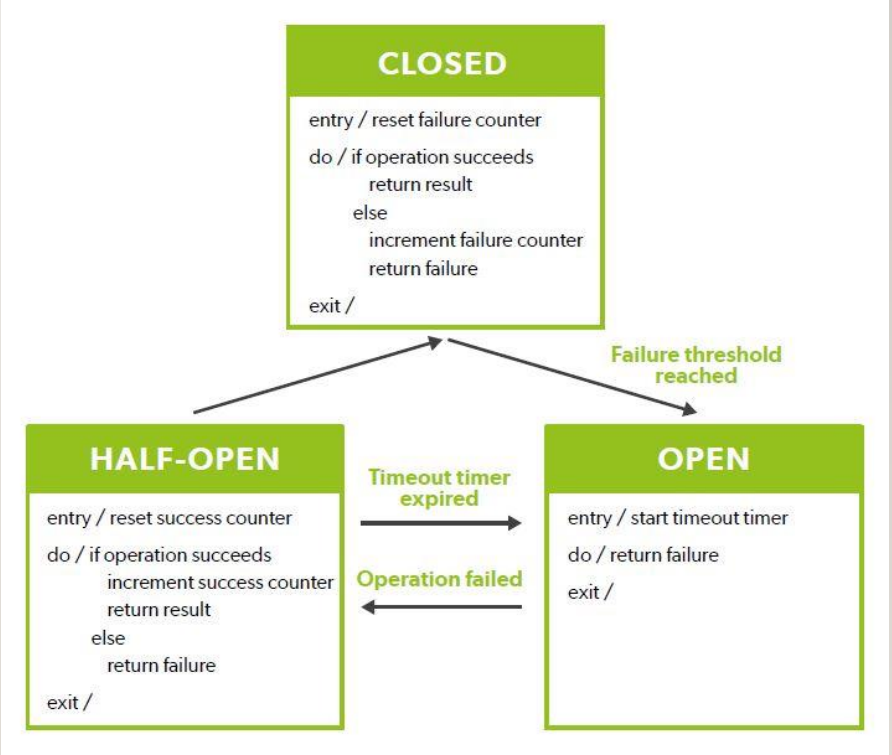
What Is the Circuit Breaker Pattern?

- **The circuit breaker pattern is an application resiliency pattern used to limit the number of requests to a service based on configured thresholds — helping to prevent the service from being overloaded.**

https://www.jrebel.com/blog/microservices-resilience-patterns

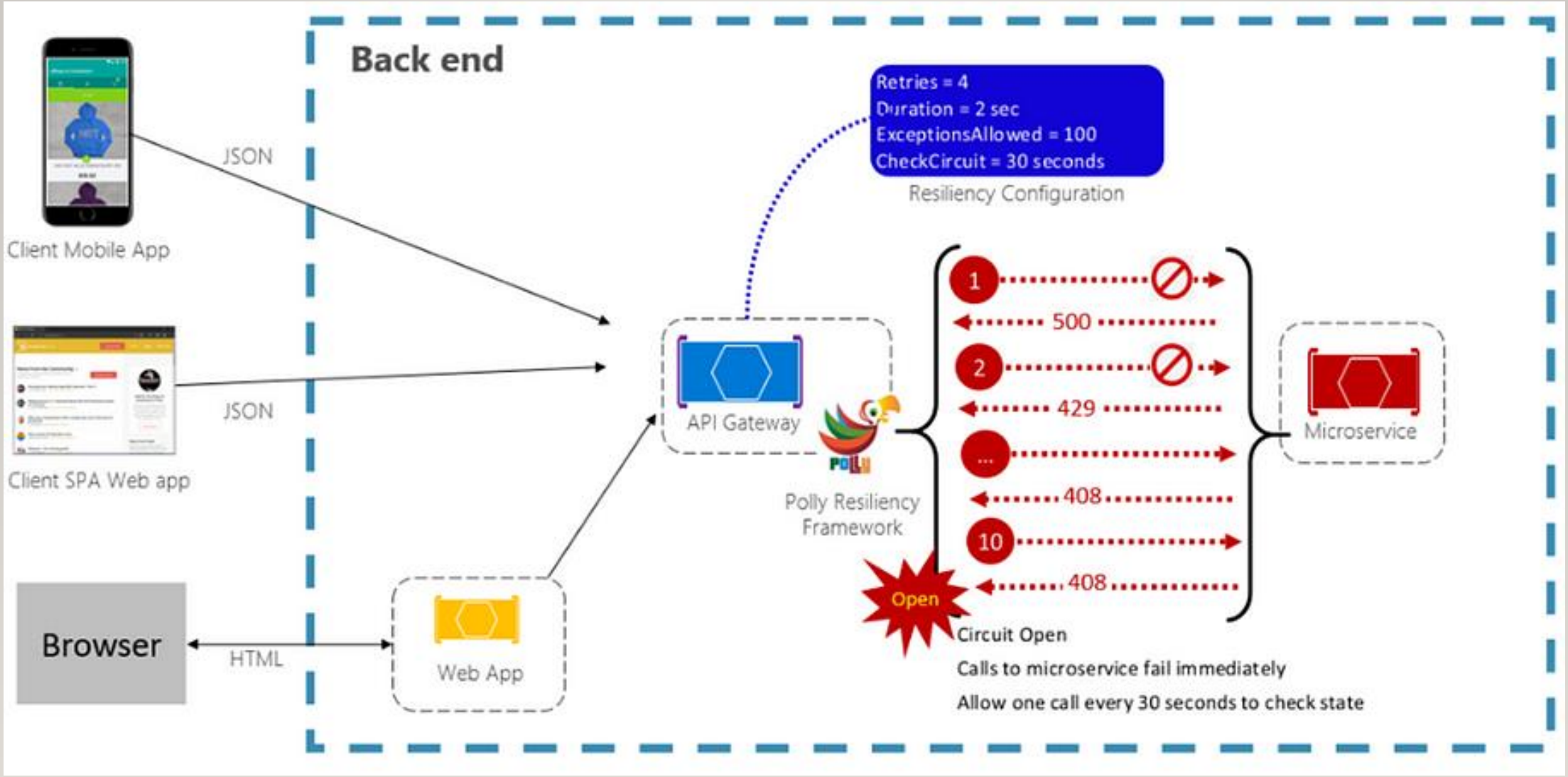
WHEN TO USE THE CIRCUIT BREAKER MICROSERVICES PATTERN

- The circuit breaker pattern can add significant overhead to your application, so it's best to use sparingly, and only in cases where you're accessing a remote service or shared resource prone to failure.
- If you user a service mesh like Istio, it's easy to experiment with various resiliency patterns, including Circuit Breaker, to see which techniques work best for each service.



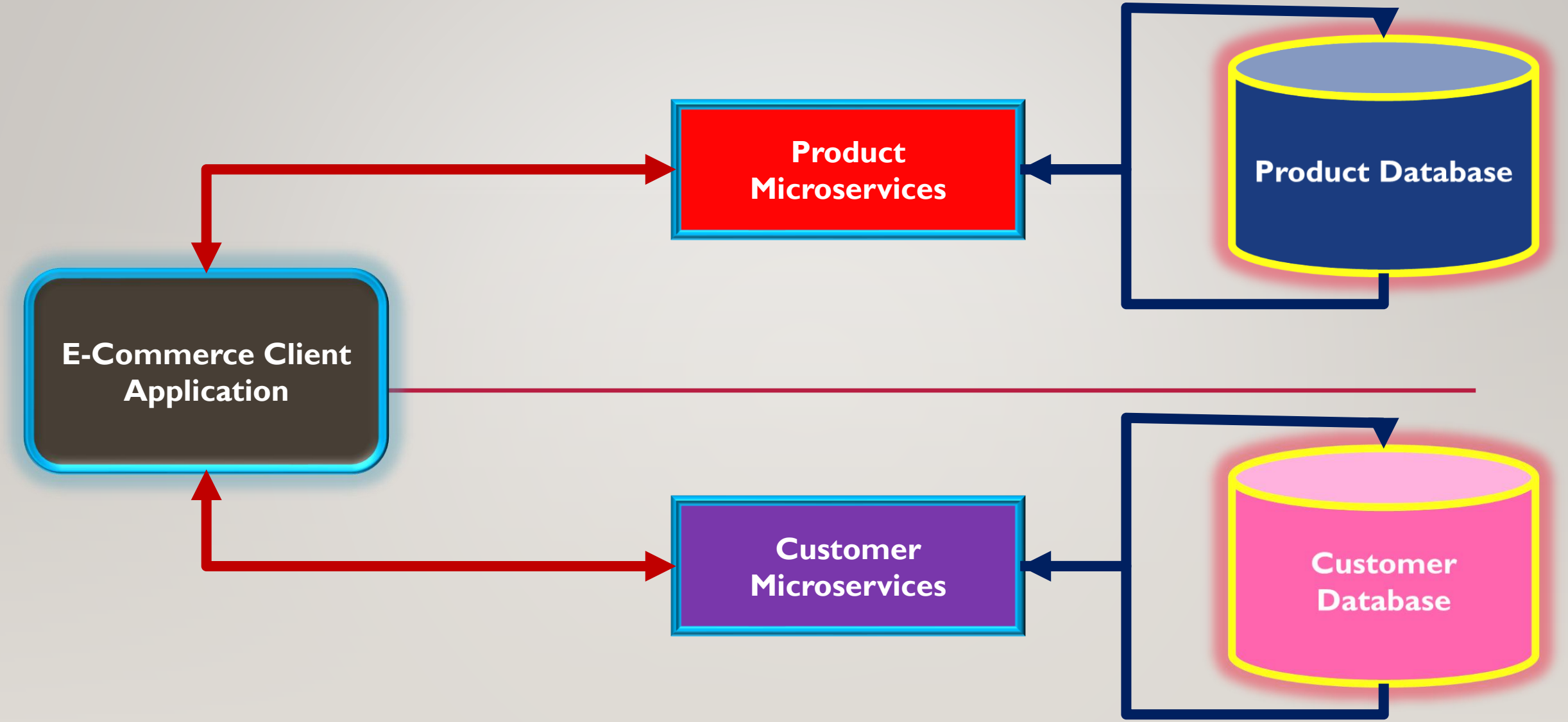
https://www.jrebel.com/blog/microservices-resilience-patterns

RESILIENC

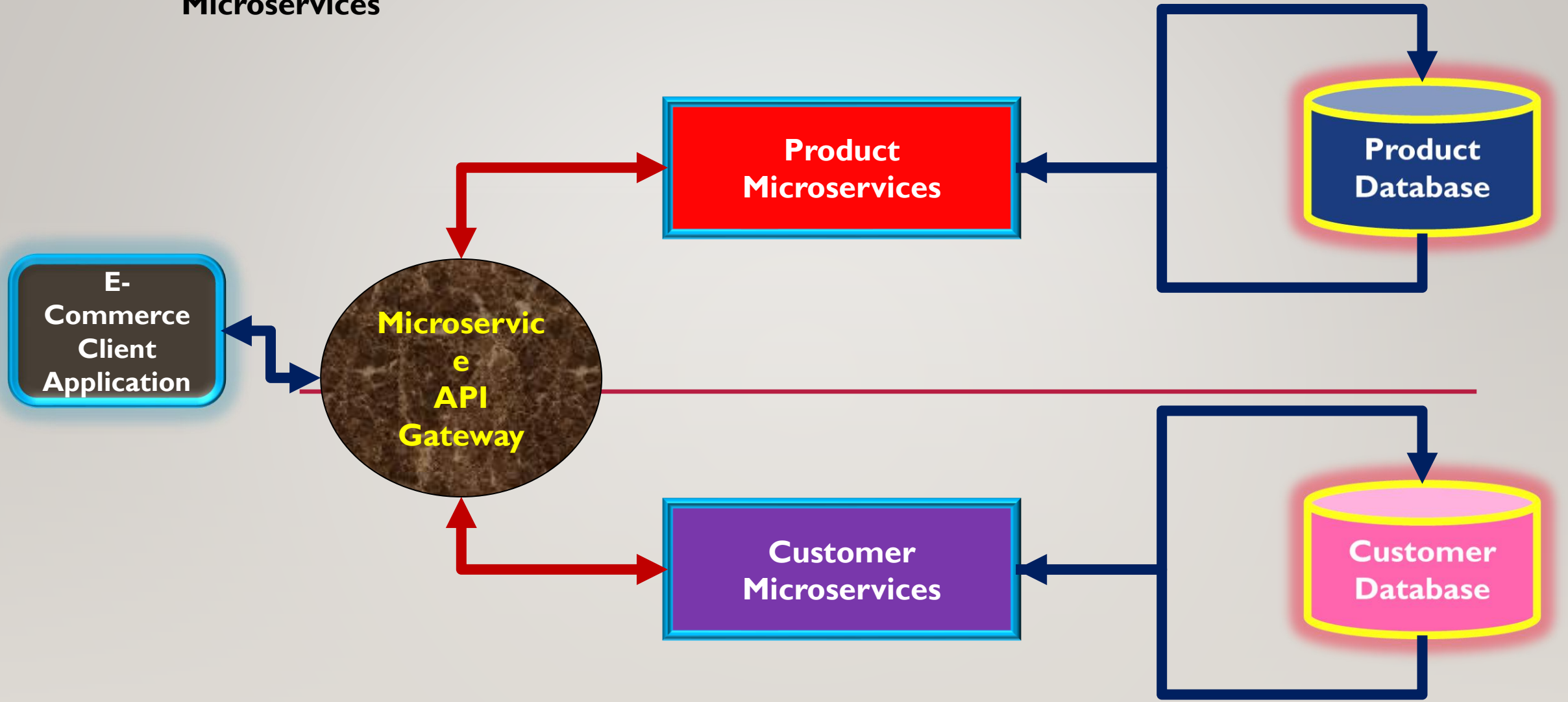


https://medium.com/aspnetrun/microservices-resilience-and-fault-tolerance-with-applying-retry-and-circuit-breaker-patterns-c32e518db990

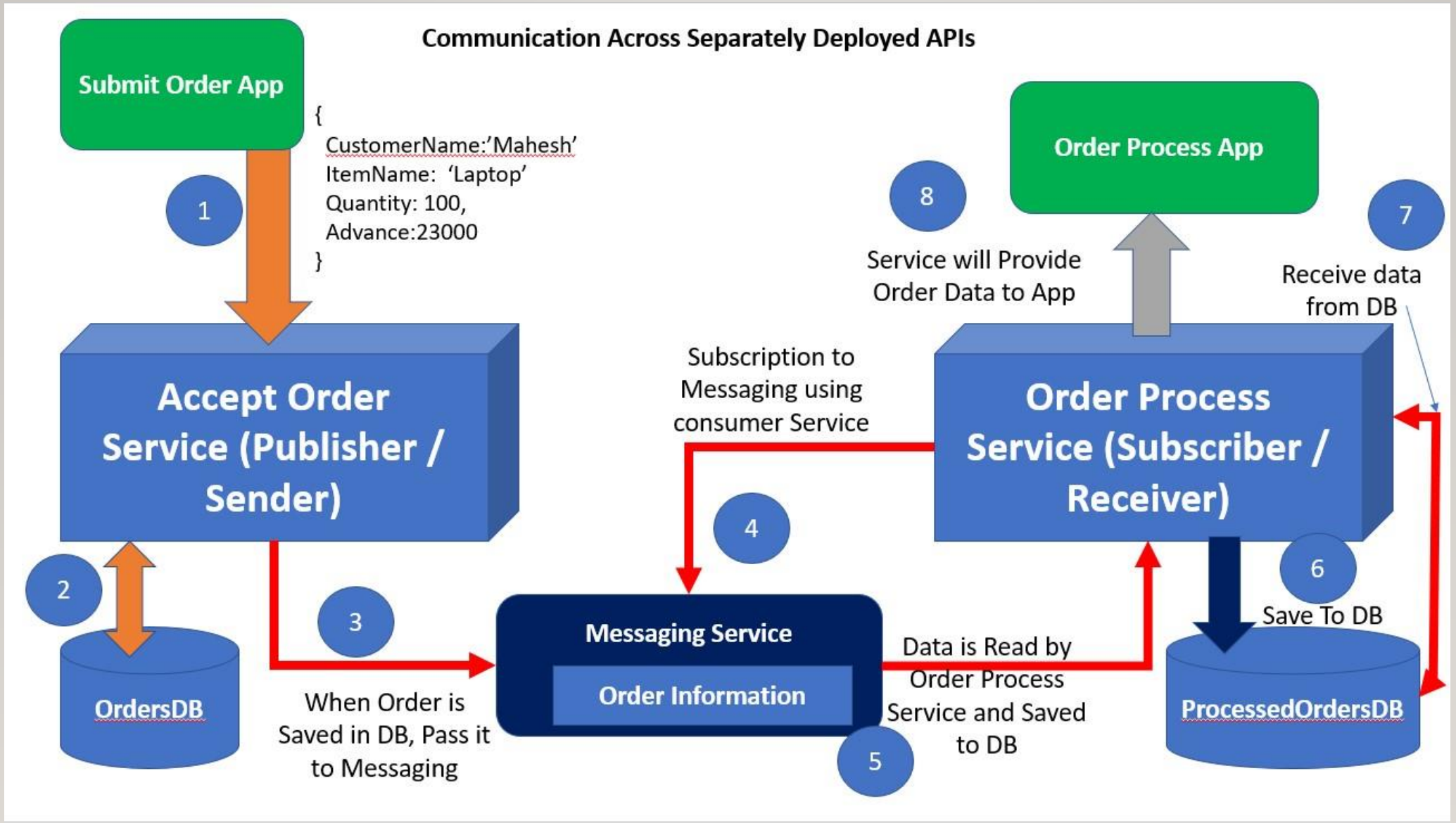
Direct Access of the Microservices



Resilience Design of the Microservices



MICROSERVICES COMMUNICATION

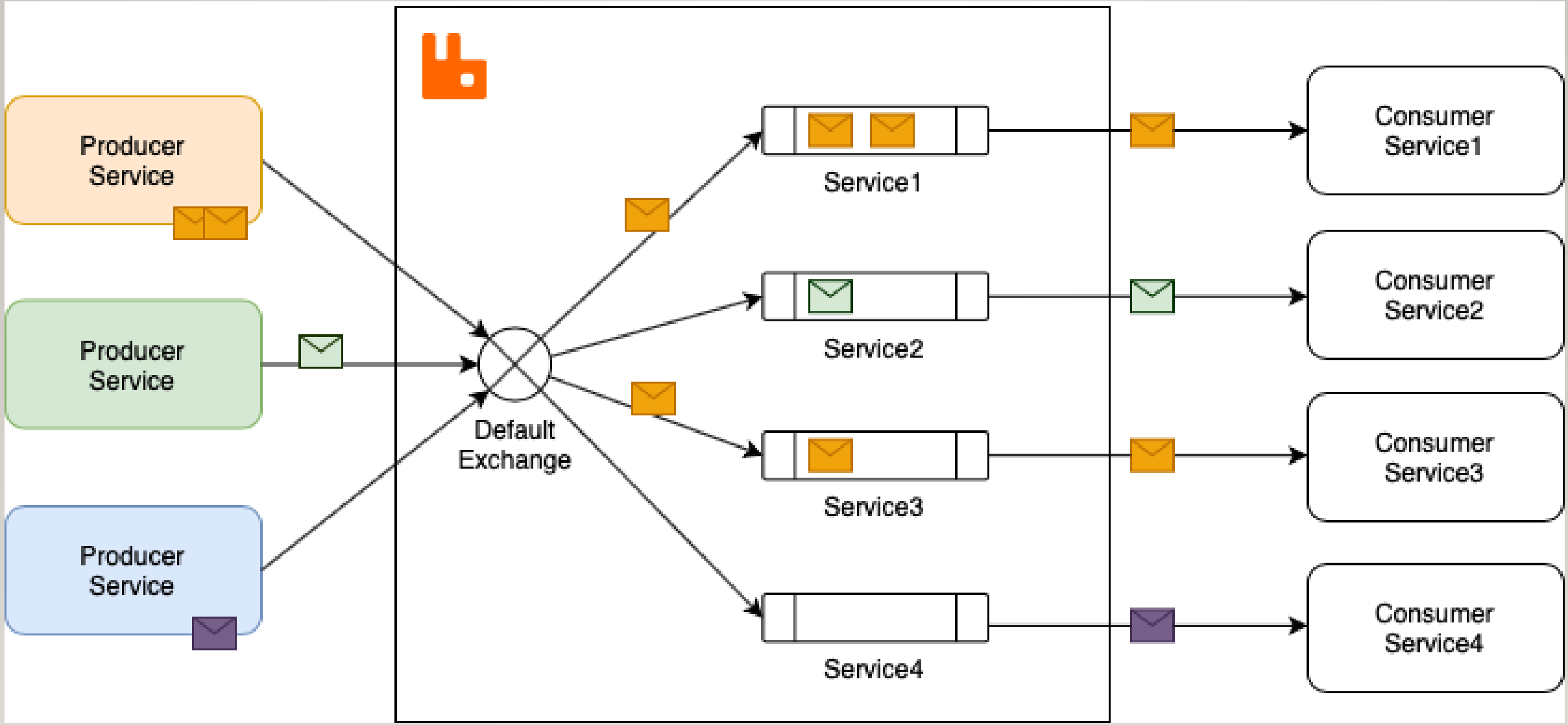

Communication Across Separately Deployed APIs
Submit Order App
{
CustomerName:'Mahesh'
ItemName: 'Laptop'
Quantity: 100,
Advance:23000
}
1
Accept Order Service (Publisher / Sender)
2
OrdersDB
3
When Order is Saved in DB, Pass it to Messaging
Messaging Service
Order Information
4
Subscription to Messaging using consumer Service
5
Data is Read by Order Process Service and Saved to DB
Order Process Service (Subscriber / Receiver)
6
Save To DB
ProcessedOrdersDB
7
Receive data from DB
8
Service will Provide Order Data to App
Order Process App

RABBITMQ

- RabbitMQ is the most widely deployed open source message broker.
- RabbitMQ is lightweight and easy to deploy on premises and in the cloud. It supports multiple messaging protocols. RabbitMQ can be deployed in distributed and federated configurations to meet high-scale, high-availability requirements.
- It handles the accepting, storing, and sending of messages between our applications. Using a message broker allows us to build decoupling, performant applications, relying on asynchronous communication between our applications.
- The durable queue will keep message persisted even after broker restarts

INSTALLING RABBITMQ

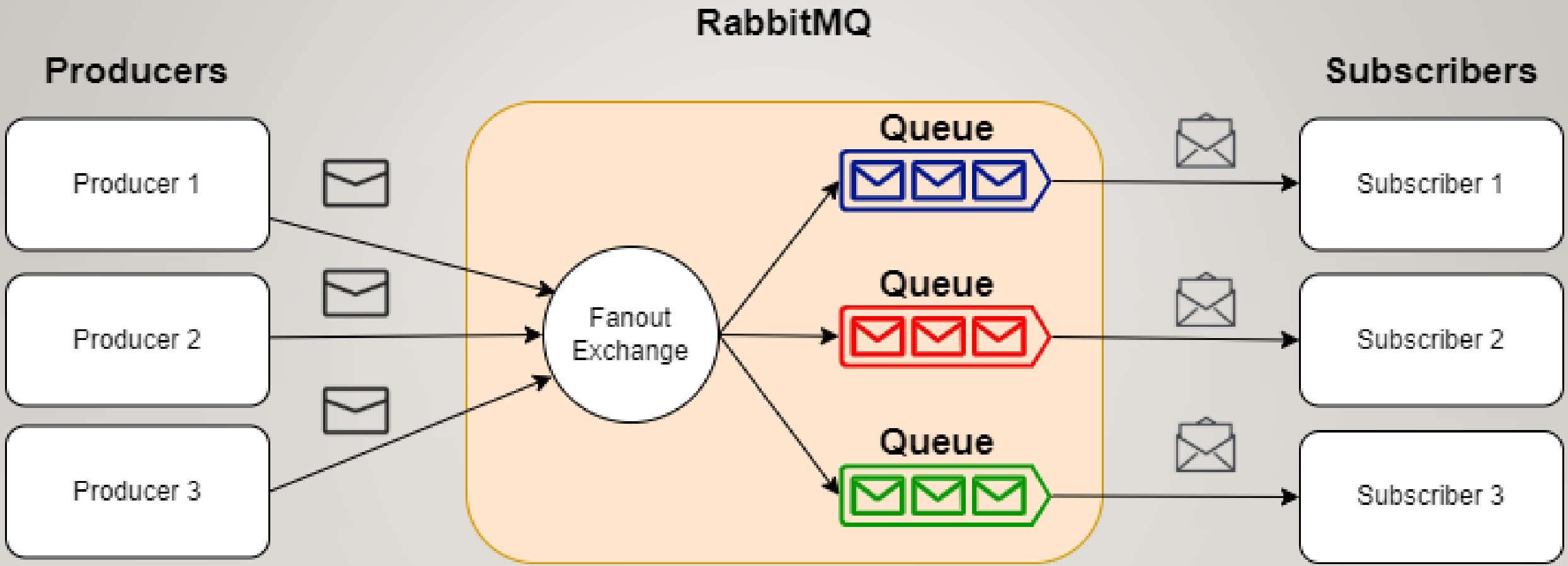
- docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3.11-management



What Is RabbitMQ? An Introduction (erlang-solutions.com)

WHAT ARE RABBITMQ EXCHANGES?

- When working with RabbitMQ, producers can send messages to a couple of different endpoints:
  - Queues
  - Exchanges
- When a producer sends directly to a queue, this message will be received by all consumers of that queue. But what if we want to selectively send messages to different queues based on metadata found in the message? This is where exchanges come into play.
- An exchange receives messages from producers, and depending on its configuration, will send the message to one or many queues. We must create a **binding**, which will ensure our messages get sent from our exchange to one or many queues.
- We can define exchanges from one of the following types:
  - Direct
  - Topic
  - Headers
  - Fanout



Using MassTransit with RabbitMQ in ASP.NET Core - Code Maze (code-maze.com)

MASS TRANSIT

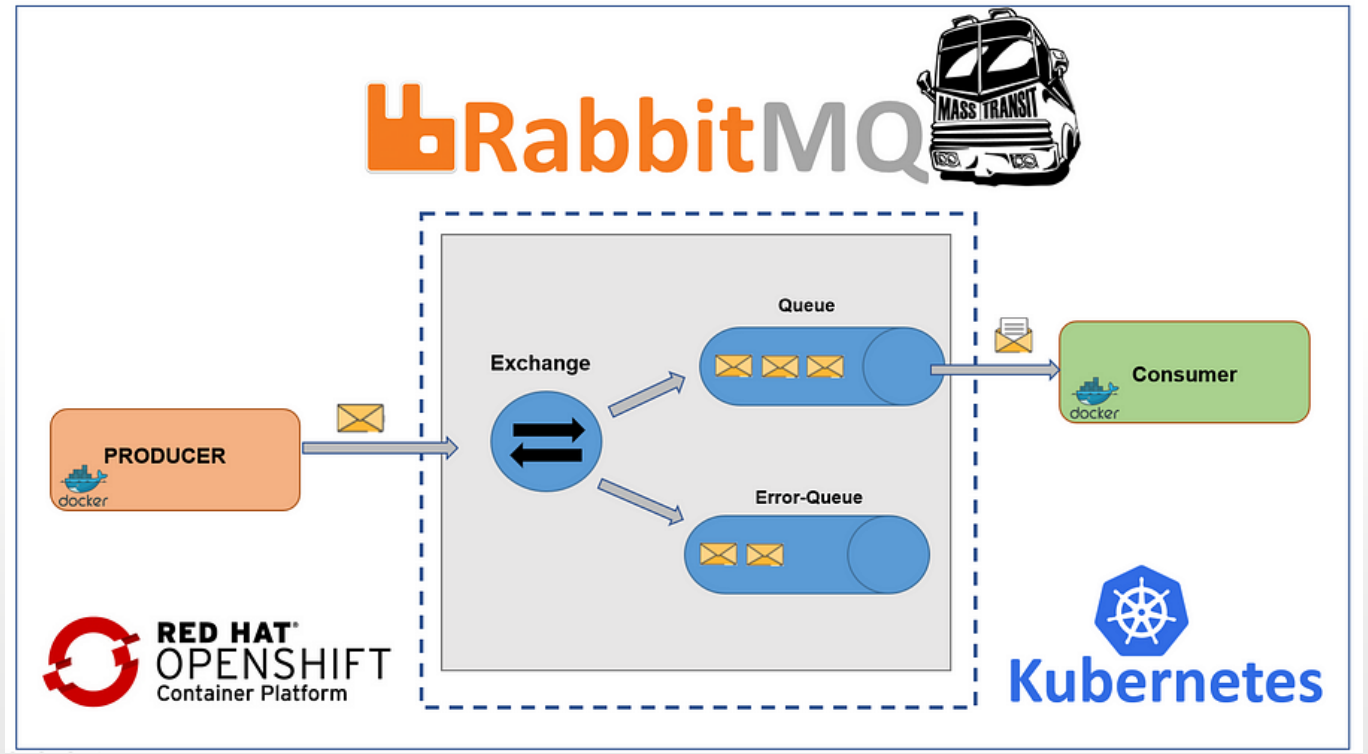
MASSTRANSIT

- MassTransit is a free, open-source, distributed application framework for .NET applications.
- **It abstracts away the underlying logic required to work with message brokers, such as RabbitMQ, making it easier to create message-based, loosely coupled applications**.

- **Service Bus,** usually shortened to **Bus**, is the term given to the type of application that handles the movement of messages.
- **Transports** are the different types of message brokers MassTransit works with, including RabbitMQ, InMemory, Azure Service Bus, and more.
- **Message** is a contract, defined *code first* by creating a .NET class or interface.
- **Command** is a type of message, specifically used to tell a service to do something. These message types are **sent** to an endpoint (queue) and will be expressed using a verb-noun sequence.
- **Events** are another message type, signifying that something has happened. Events are **published** to one or multiple consumers and will be expressed using noun-verb (past tense) sequence.

MASSTRANSIT

- **Why Use MassTransit?**
- There are a few benefits to choosing to use a library such as MassTransit, instead of working with the native message broker library.
- Firstly, **by abstracting the underlying message broker logic, we can work with multiple message brokers**, without having to completely rewrite our code.
  - This allows us to work with something such as the InMemory transport when working locally, then when deploying our code, use another transport such as Azure Service Bus or Amazon Simple Queue Service.
- Additionally, when we work with a message-based architecture, **there are a lot of specific patterns we need to be aware of and implement, such as *retry*, *circuit breaker*, *outbox* to name a few.**
- **MassTransit handles all of this for application**, along with many other features such as *exception handling*, *distributed transactions*, and *monitoring*.



https://darshanadinushal.medium.com/implement-an-event-driven-microservice-architecture-on-kubernetes-with-rabbitmq-using-masstransit-c24a400e5e00

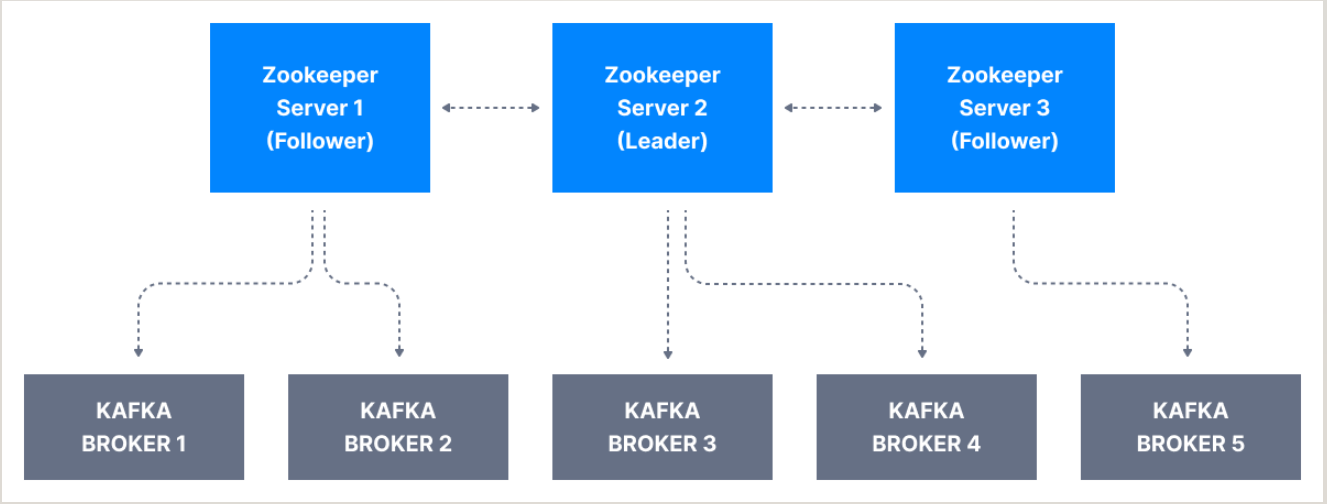
KAFKA

KAFKA

- Apache Kafka is an open-source distributed streaming system used for stream processing
- Apache Kafka is a distributed event store and stream-processing platform. It is an open-source system developed by the Apache Software Foundation written in Java and Scala.
- The project aims to provide a unified, high-throughput, low-latency platform for handling real-time data feeds.

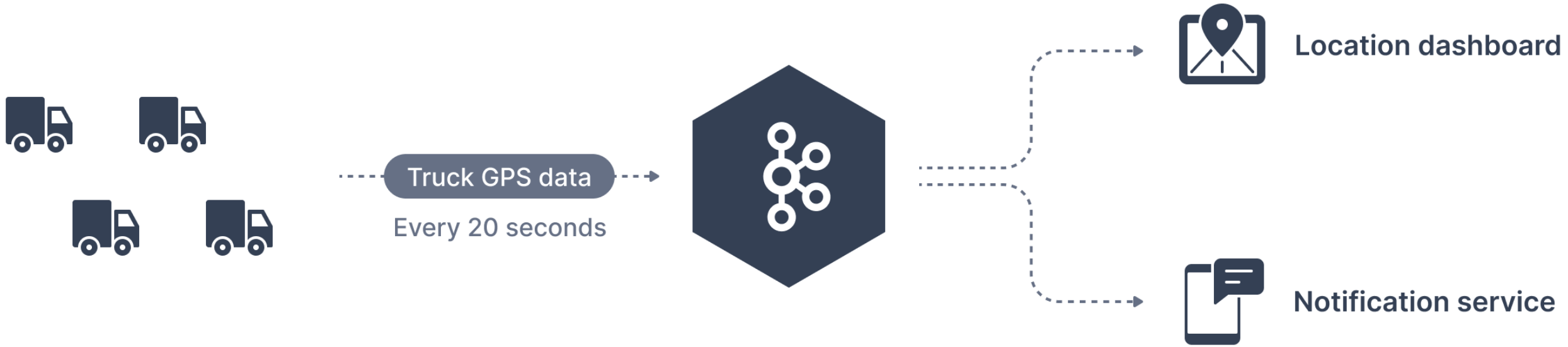
ZOOKEEPER

- Zookeeper is used for metadata management in the Kafka world. For example:
- Zookeeper keeps track of which brokers are part of the Kafka cluster
- Zookeeper is used by Kafka brokers to determine which broker is the leader of a given partition and topic and perform leader elections
- Zookeeper stores configurations for topics and permissions
- Zookeeper sends notifications to Kafka in case of changes (e.g. new topic, broker dies, broker comes up, delete topics, etc.…)



KAFKA TOPICS

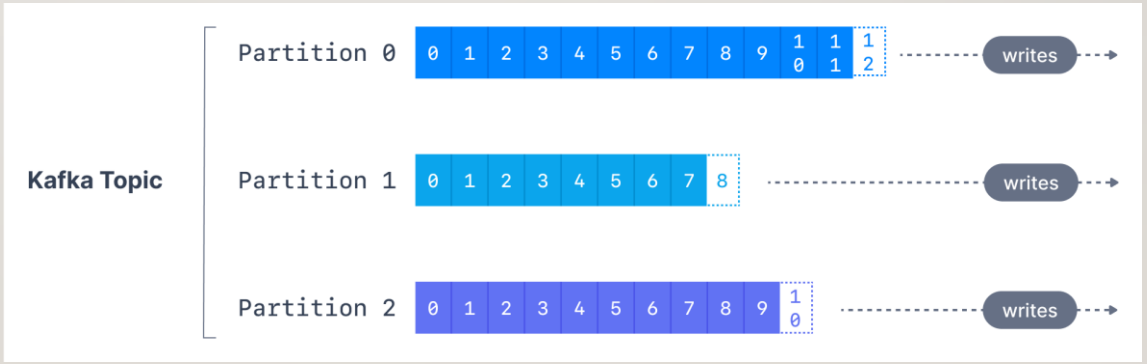
- **What is a Kafka Topic?**
- Similar to how databases have tables to organize and segment datasets, Kafka uses the concept of topics to organize related messages.
- A topic is identified by its name. For example, we may have a topic called **logs** that may contain log messages from our application, and another topic called **purchases** that may contain purchase data from our application as it happens.
- Kafka topics can contain any kind of message in any format, and the sequence of all these messages is called a data stream.
- Data in Kafka topics is deleted after one week by default (also called the default message retention period), and this value is configurable. This mechanism of deleting old data ensures a Kafka cluster does not run out of disk space by recycling topics over time.


Truck GPS data
Every 20 seconds
Location dashboard
Notification service

KAFKA TOPICS

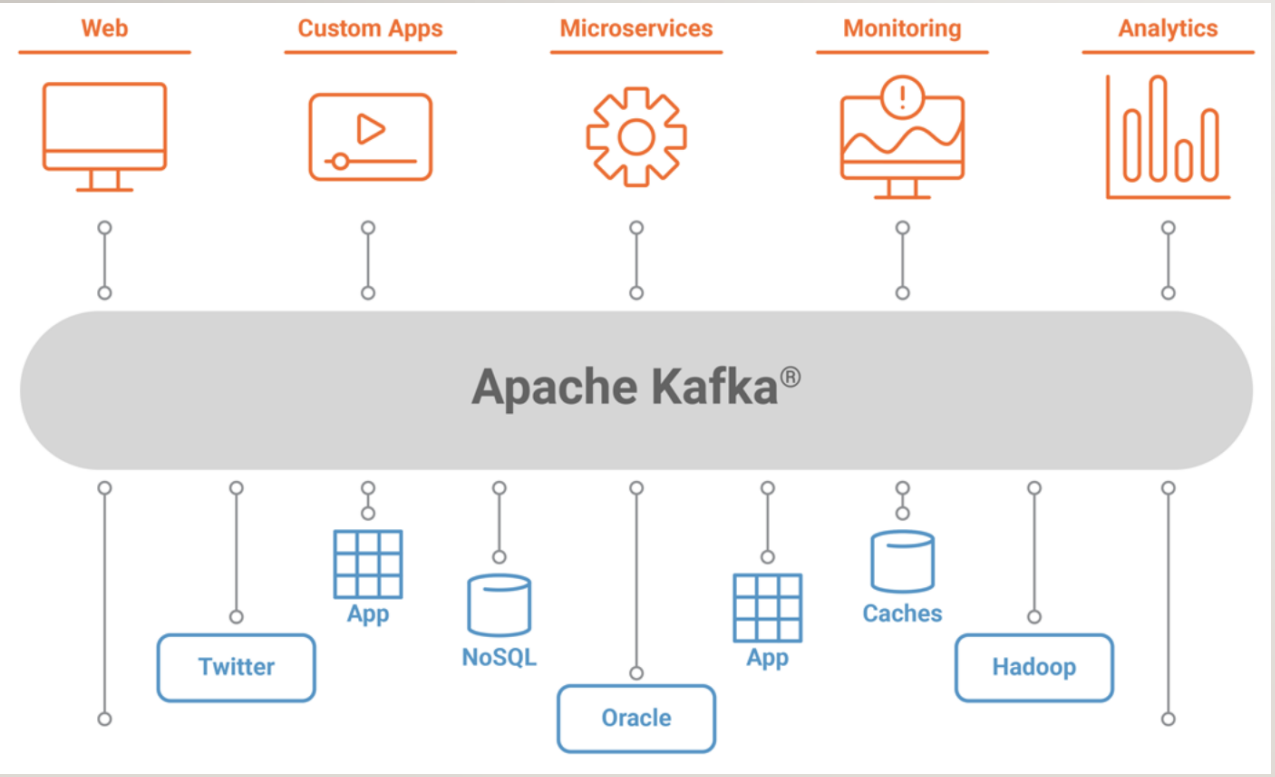
- What are Kafka Partitions?
- Topics are broken down into a number of partitions. A single topic may have more than one partition, it is common to see topics with 100 partitions.

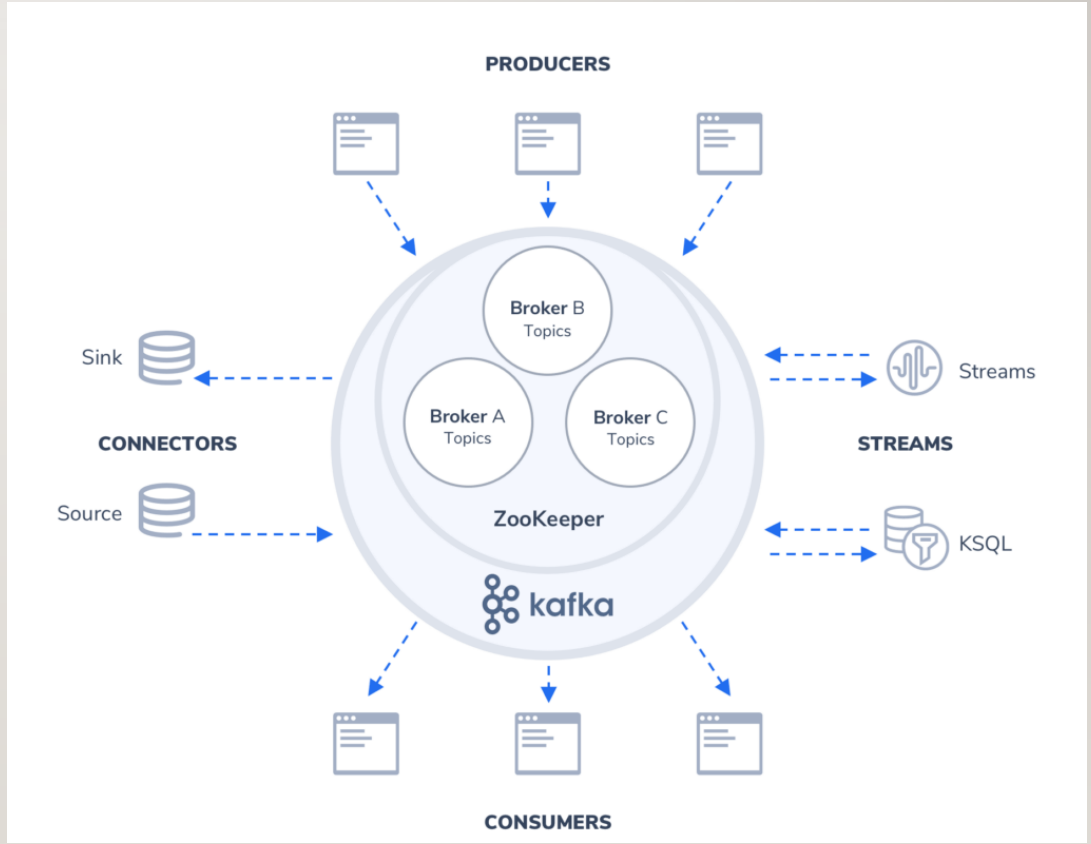
- The number of partitions of a topic is specified at the time of topic creation. Partitions are numbered starting from 0 to N-1, where N is the number of partitions. The figure below shows a topic with three partitions, with messages being appended to the end of each one



KAFKA

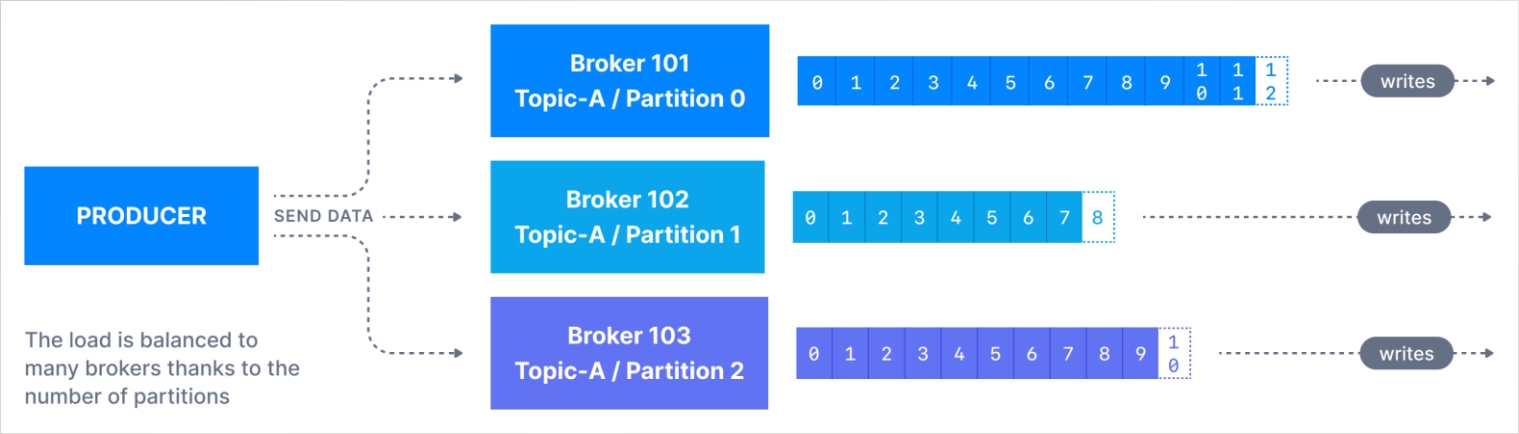
- Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications





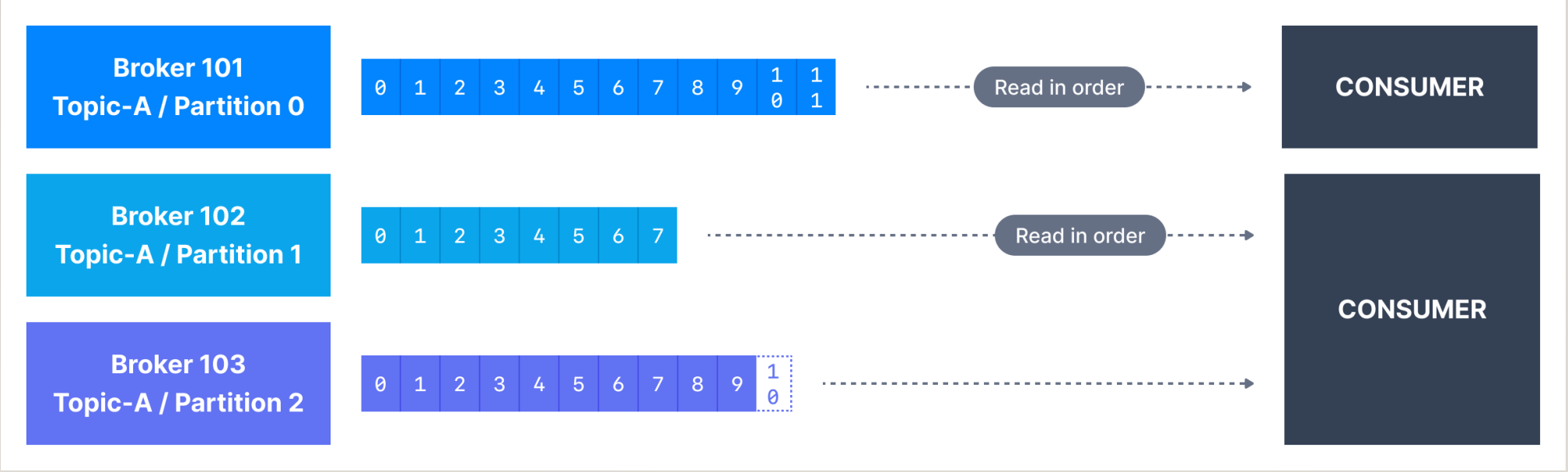
KAFKA PRODUCERS

- Applications that send data into topics are known as Kafka producers. Applications typically integrate a Kafka client library to write to Apache Kafka. Excellent client libraries exist for almost all programming languages that are popular today including Python, Java, Go, and others.
- A Kafka producer sends messages to a topic, and messages are distributed to partitions according to a mechanism such as key hashing



KAFKA CONSUMER

- Applications that read data from Kafka topics are known as consumers. Applications integrate a Kafka client library to read from Apache Kafka. Excellent client libraries exist for almost all programming languages that are popular today including Python, Java, Go, and others.
- Consumers can read from one or more partitions at a time in Apache Kafka, and data is read in order **within each partition** as shown below.



MICROSERVICES ARCHITECTURE PATTERN - SAGA

WHAT IS A DISTRIBUTED TRANSACTION?

- In the world of microservices, each service has its own local database and some operations we do in microservices may involve multiple database operations for multiple services.
- So, a distributed transaction may contain local transactions for multiple databases.
- What we often encounter is the need to guarantee data consistency, updated in parallel by different actors.

WHAT IS SAGA?

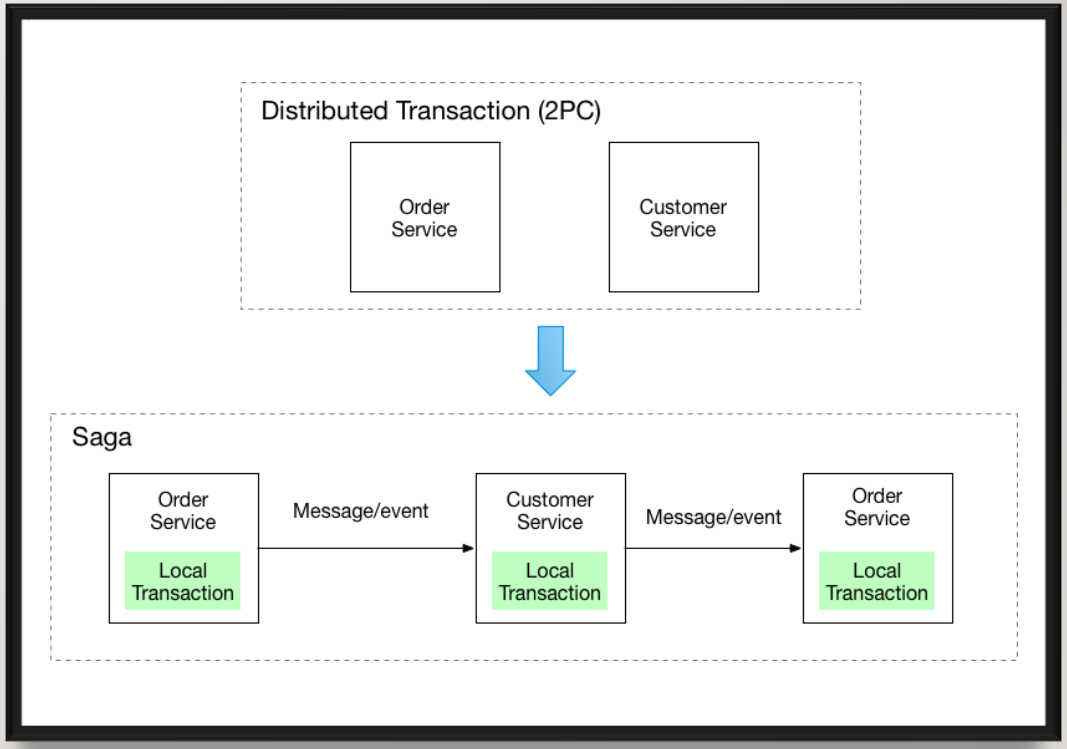
- Saga first appeared in a paper published by Hector Garcaa-Molrna & Kenneth Salem in 1987.
- The core idea is to split a long transaction into multiple short transactions, coordinated by the Saga transaction coordinator, with the global transaction completing normally if each short transaction completes successfully, and invoking the compensating operations one at a time according to the reverse order if a step fails.
- The Saga pattern is a widely used pattern for distributed transactions. It is asynchronous and reactive.

PATTERN: SAGA

- Context
  - You have applied the Database per Service pattern. Each service has its own database. Some business transactions, however, span multiple service so you need a mechanism to implement transactions that span services.
  - For example, let's imagine that you are building an e-commerce store where customers have a credit limit. The application must ensure that a new order will not exceed the customer's credit limit.
  - Since Orders and Customers are in different databases owned by different services the application cannot simply use a local ACID transaction.

PATTERN: SAGA

- Implement each business transaction that spans multiple services as a saga.
- A saga is a sequence of local transactions.
- Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga.
- If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.

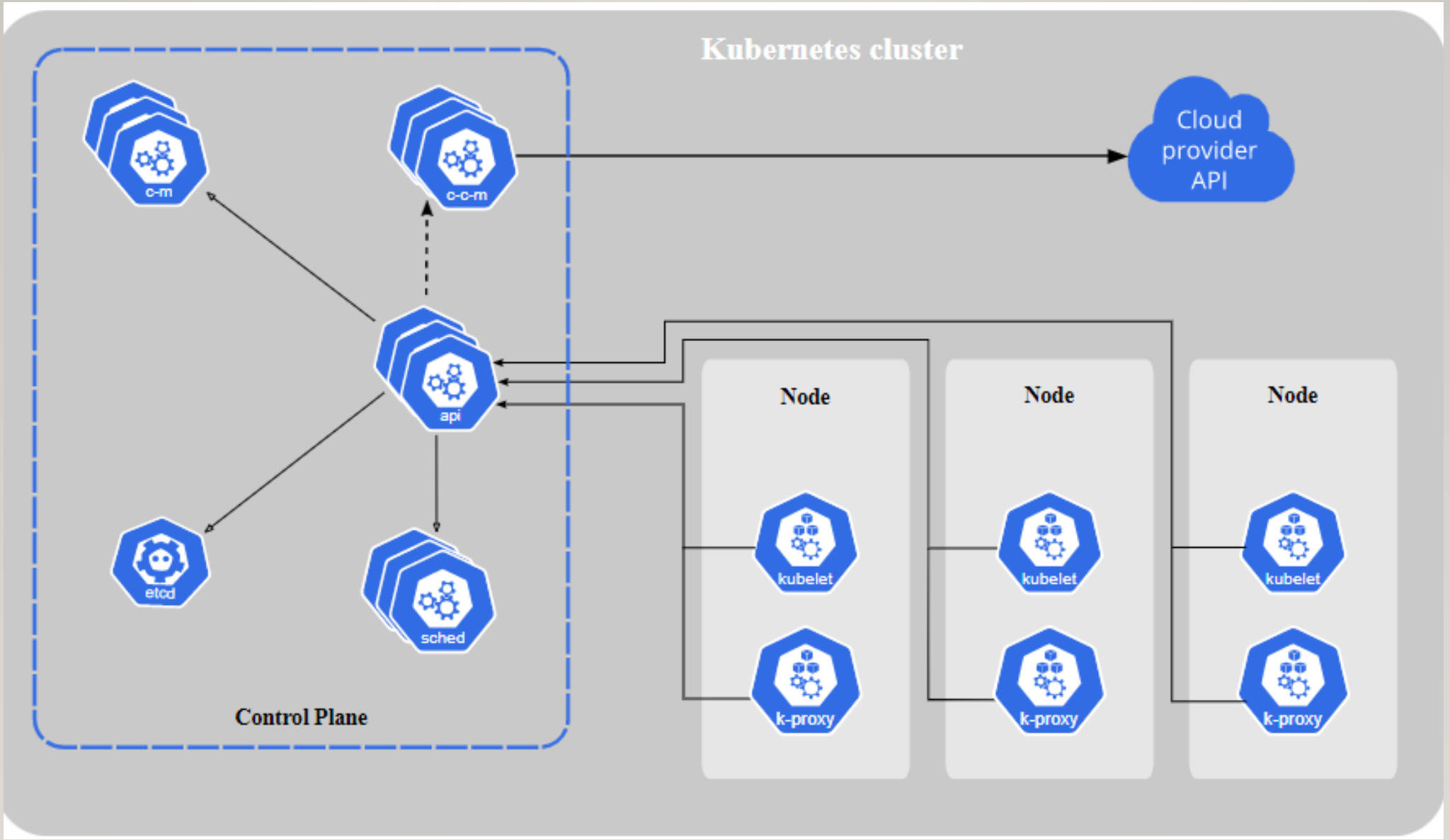


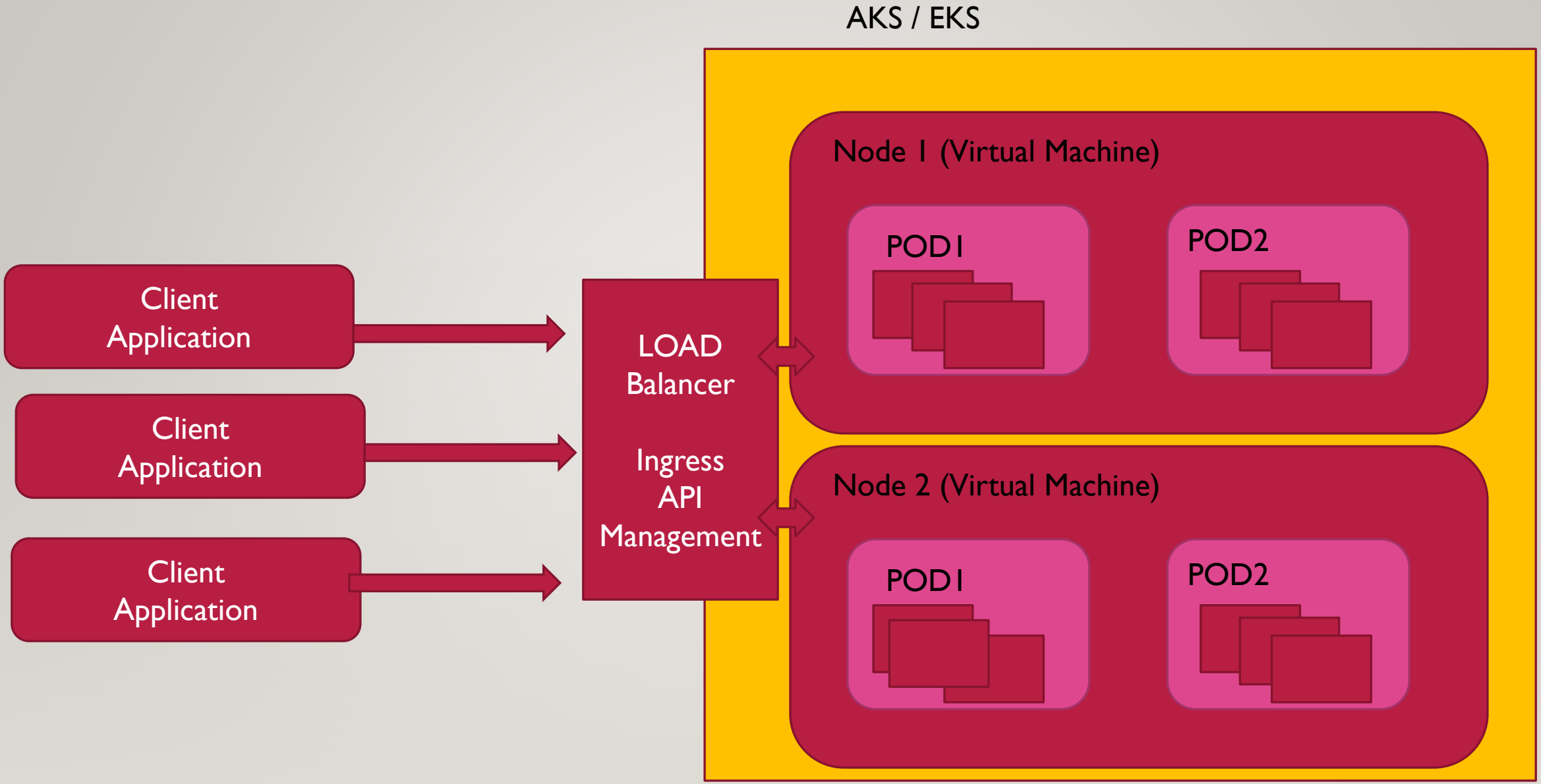
https://microservices.io/patterns/data/saga.html

WHAT IS DTM?

- DTM is a distributed transaction framework which provides cross-service eventual data consistency.
- It provides saga, tcc, xa, 2-phase message strategies for a variety of application scenarios.
- It also supports multiple languages and multiple store engine to form up a transaction.
- There are some features of DTM:
  - Extremely easy to adopt
  - Easy to use
  - Multi-language support
  - Easy to deploy, easy to extend
  - Multiple distributed transaction protocols
- `docker run -d -p 36789:36789 -p 36790:36790 --name=dtm-svc yedf/dtm:1.13`

KUBERNETES


Kubernetes cluster
c-m
c-c-m
Cloud provider API
api
etcd
sched
Control Plane
Node
Node
Node
kubelet
kubelet
kubelet
k-proxy
k-proxy
k-proxy


AKS / EKS
Node 1 (Virtual Machine)
POD1
POD2
Node 2 (Virtual Machine)
POD1
POD2
LOAD Balancer
Ingress API Management
Client Application
Client Application
Client Application