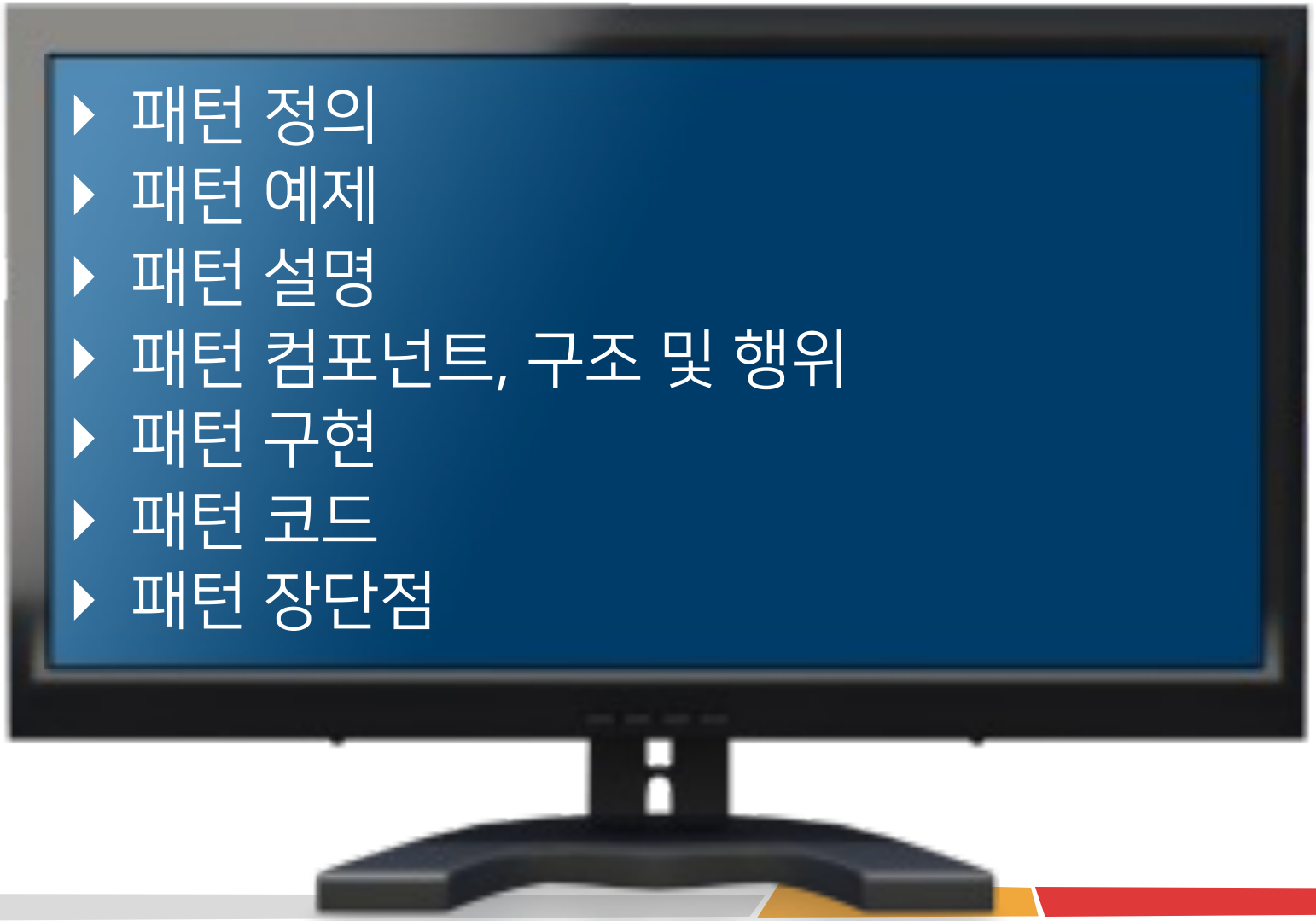
Spring 2023

소프트웨어 아키텍처 패턴: Microservice

Seonah Lee

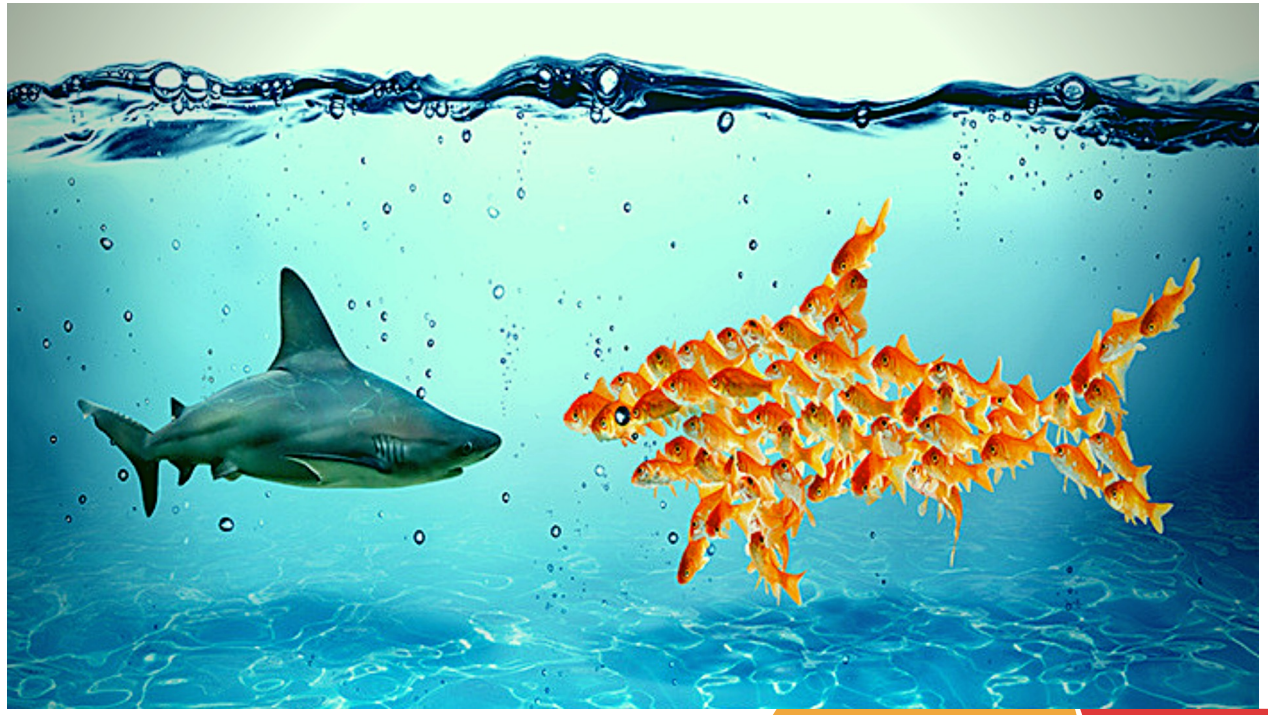
Gyeongsang National University

Microservice 패턴



Microservice Pattern: Definition

- 정의
  - 대형 소프트웨어 프로젝트의 기능들을 작고 독립적이며 느슨하게 결합된 모듈로 분해하여 서비스를 제공하는 아키텍처

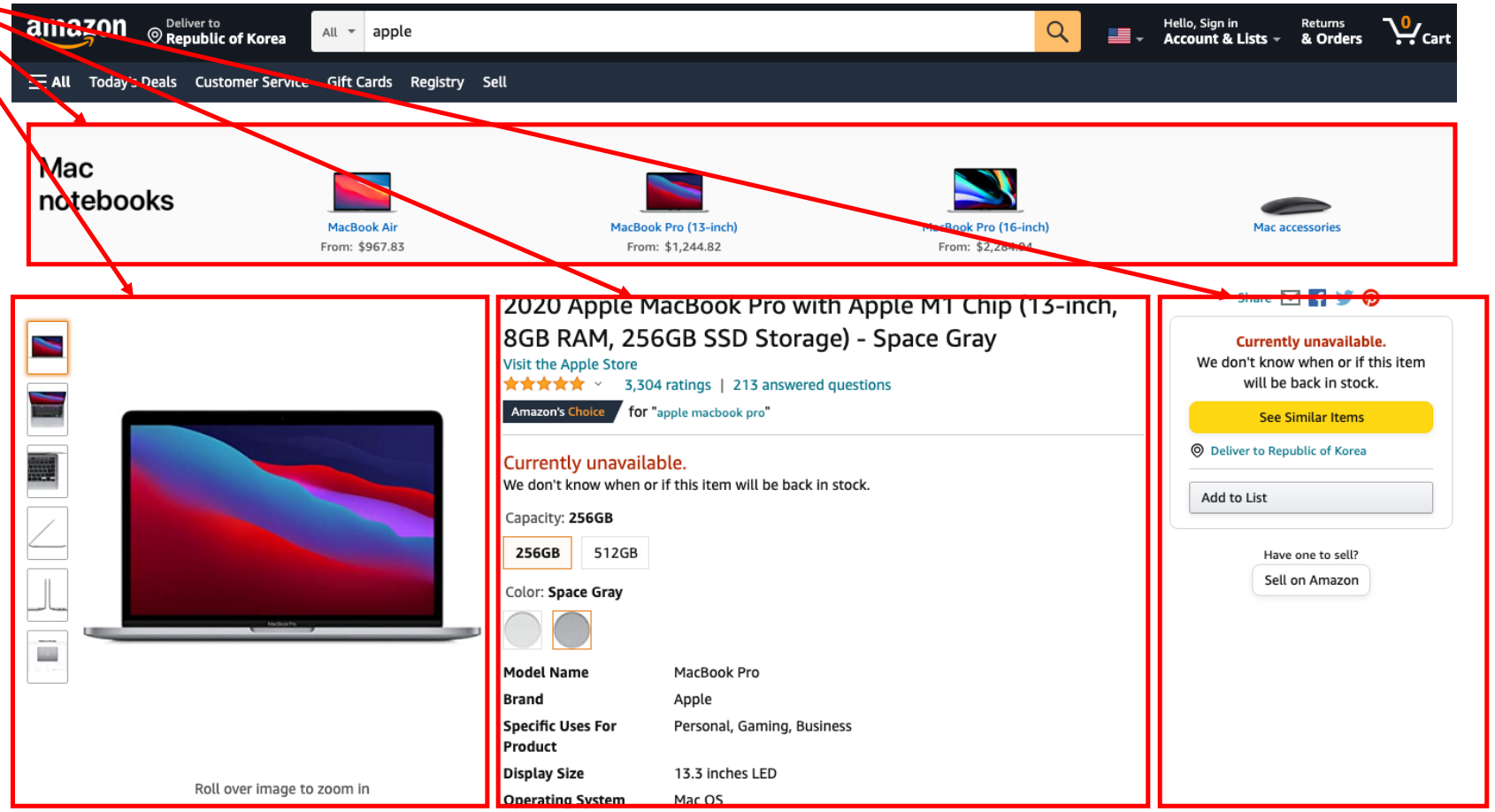


Microservice Pattern: Background Information

- 마이크로 서비스는 분산 환경의 아키텍처의 일종임
- 도메인 주도 설계(**Domain-Driven Design**) 영향을 받음
  - 특히, **Bounded Context** 개념이 결정적인 영향을 미침
    - 예: **Catalog Checkout** 도메인은 카탈로그 항목, 고객, 결제 등의 개념 가짐
    - **Context** 내부: 코드, 데이터 스키마 등 함께 연동하여 작동
    - **Context** 외부: 외부의 데이터베이스 및 클래스와는 전혀 결부되지 않음
  - **Bounded Context**의 논리적 개념을 기반으로 고도의 디커플링 추구
    - 재사용 **<** 디커플링 ➔ 중복을 우선함
    - 도메인이나 워크플로를 캡쳐하는 것이 목표

Microservice Pattern: Example

Microservices

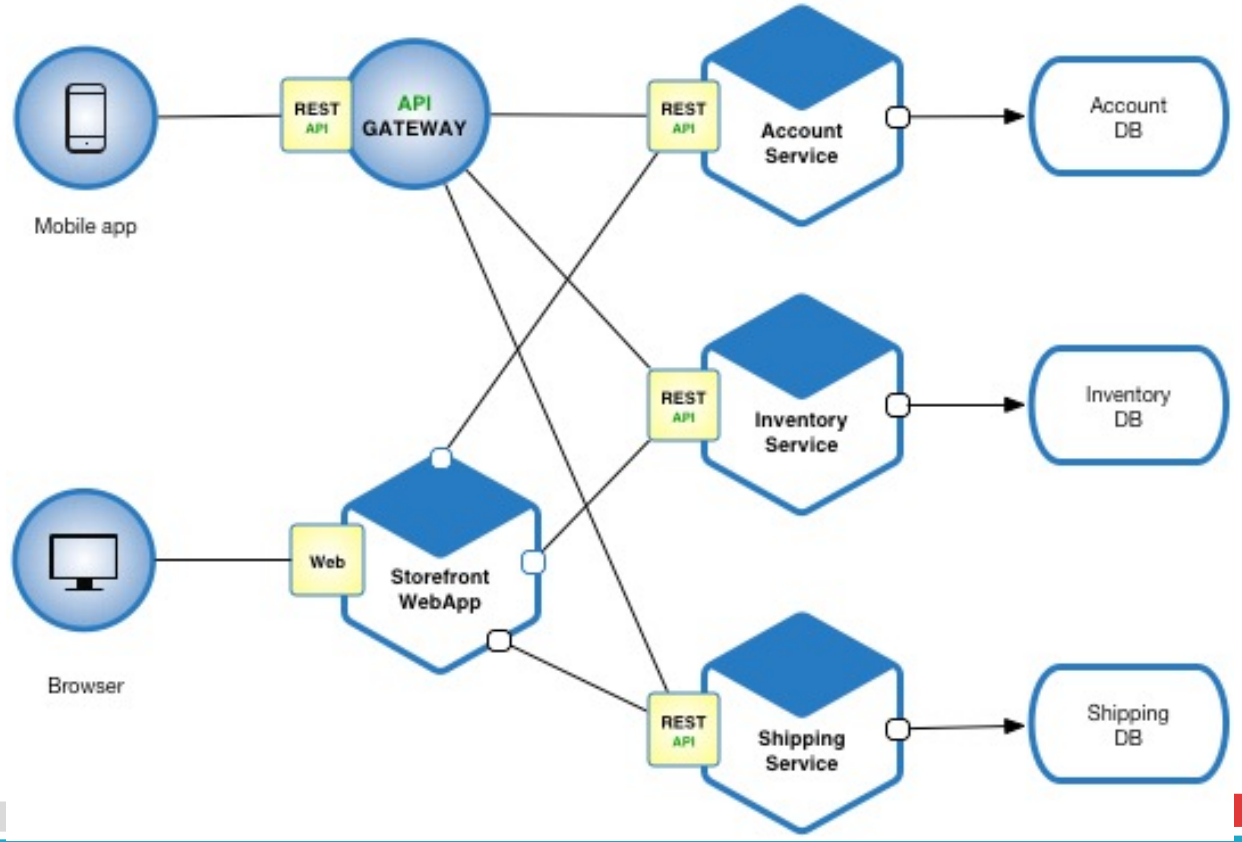


Microservice Pattern: Description

- 정황(Context)
  - 서버 측면의 엔터프라이즈 애플리케이션을 개발하며, 클라이언트가 웹 브라우저, 모바일 앱 등 다양한 형태로 존재, 다른 앱과 연동
  - 클라이언트이 요청을 처리, 데이터베이스 접근과 다른 시스템과의 메시지 교환, 응답을 **HTML/JSON/XML**형태로 돌려줄 필요가 있음
- 문제(Problem)
  - 애플리케이션의 배치(**Deployment**)의 구조를 어떻게 가져갈 것인가**?**
    - 팀들 각각이 배포할 수 있어야 하며, 애플리케이션을 수정하기 용이해야 함
    - 지속적인 배치(**Deployment**)를 수행할 수 있어야 함
    - 여러 개의 인스턴스를 수행할 수 있어야 함
    - 다양한 기술(프레임워크, 프로그래밍 언어) 등을 편하게 사용하기를 원함

Microservice Pattern: Description

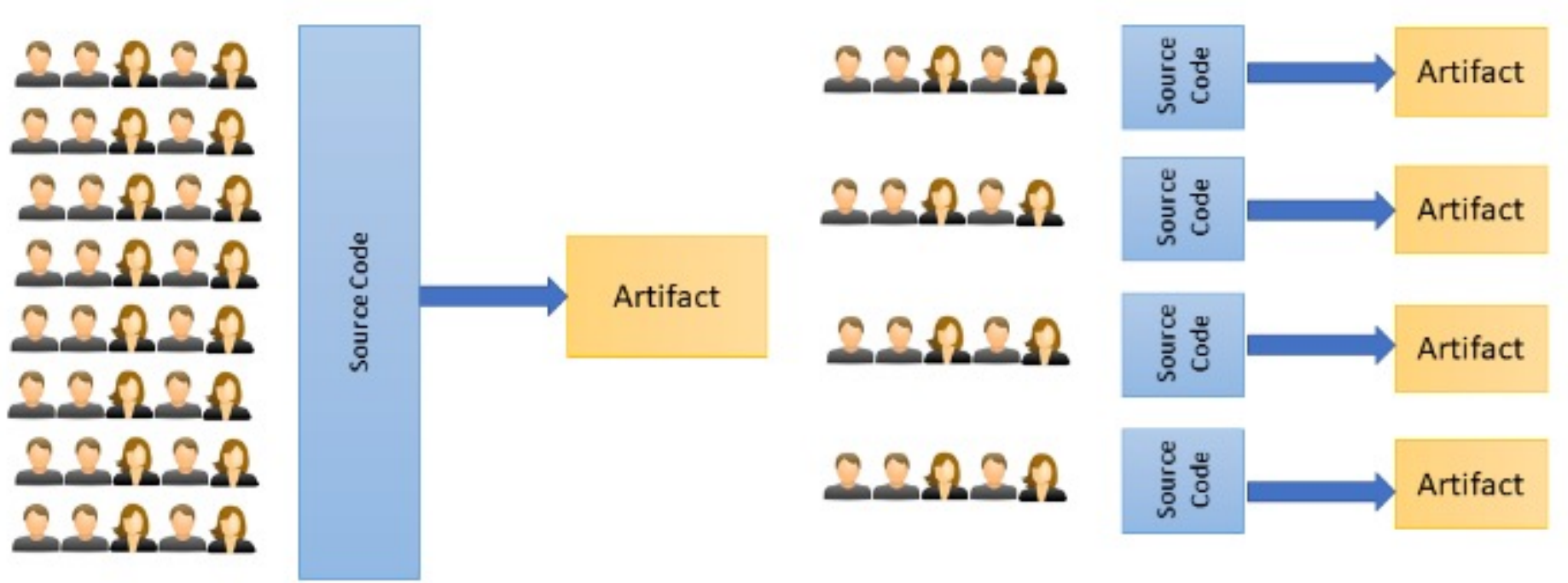
- 해법(Solution)
  - 하나의 애플리케이션이 서비스 집합으로 구성



→ 코드 커플링 해소
→ 병렬 개발 가능
→ 아웃소싱이 쉬움
→ 확장성 강화

Microservice Pattern: Description

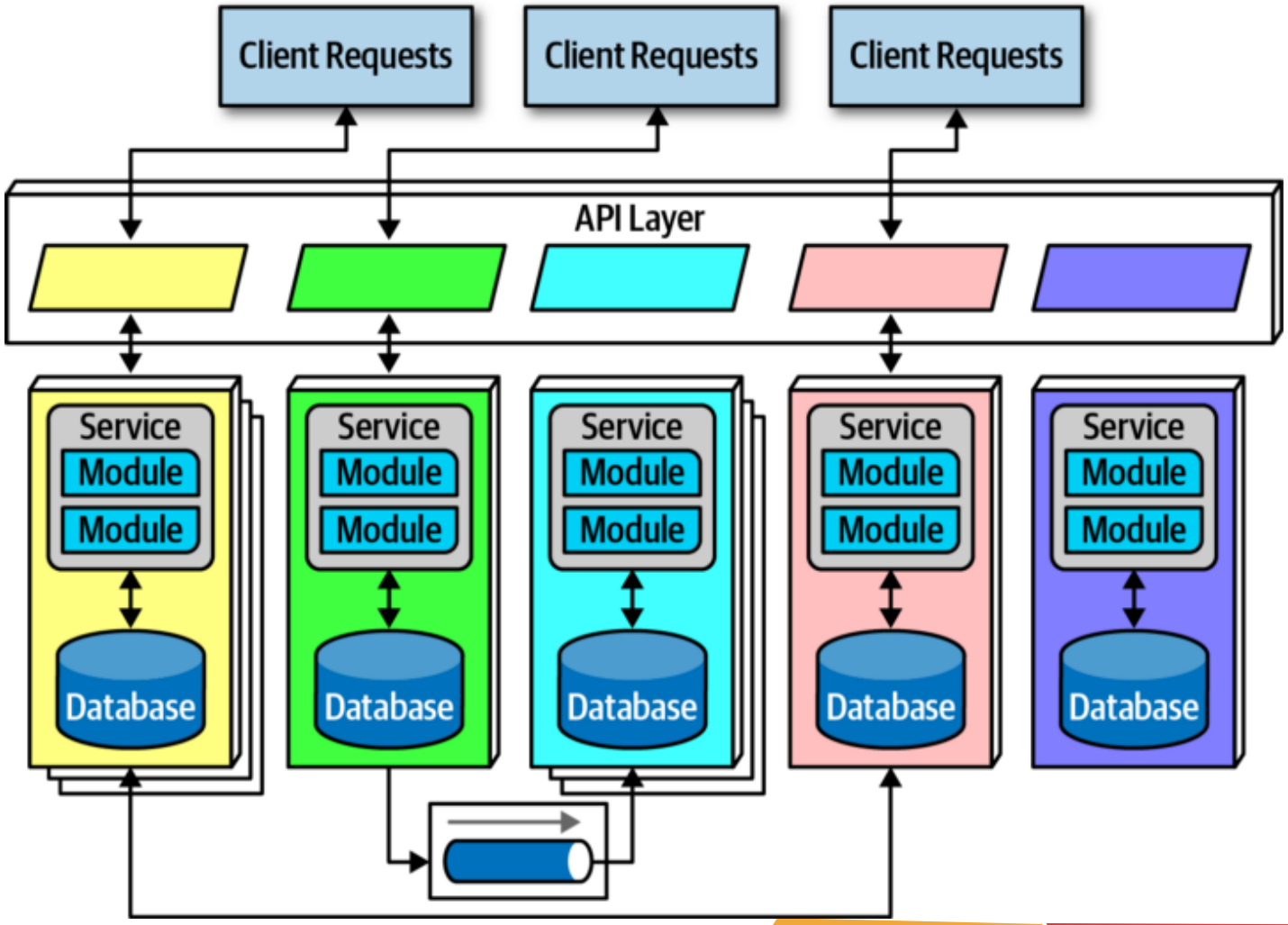
- **Microservices**
  - 작고 독립적이며 느슨하게 결합되어 있는 서비스
    - 소규모의 개발팀이 서비스를 관리; 개발팀 별 서비스마다 별도의 코드 베이스로 구성



Microservice Pattern: Description

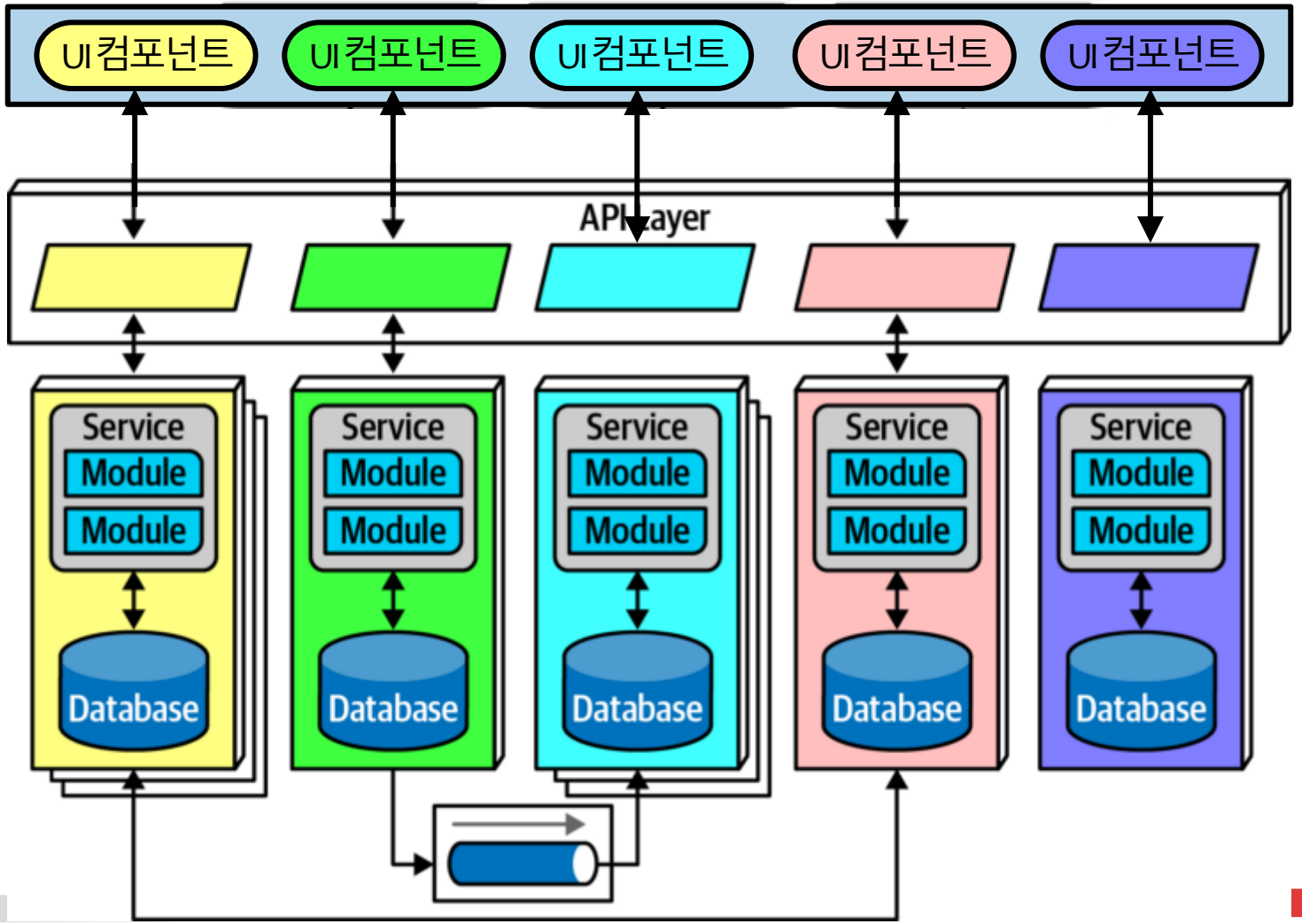
- **Microservices**
  - 서비스는 독립적으로 배포할 수 있음
    - 팀은 전체 애플리케이션을 다시 빌드하고 재배포할 필요없이 기존 서비스를 업데이트함
  - 서비스는 자체 데이터 또는 외부 상태를 유지할 책임이 있음.
    - 이는 기존 모델에서 데이터를 관리하는 별도의 티어가 있었던 것과 비교할 때 다른 부분임
  - 서비스는 잘 정의된 **API**를 사용하여 서로 통신할 수 있음.
    - 각 서비스의 구현 세부 정보는 다른 서비스에서는 보이지 않음
  - 서비스는 동일한 기술 스택, 라이브러리 그리고 프레임워크를 공유할 필요가 없음

Microservice Pattern: Description



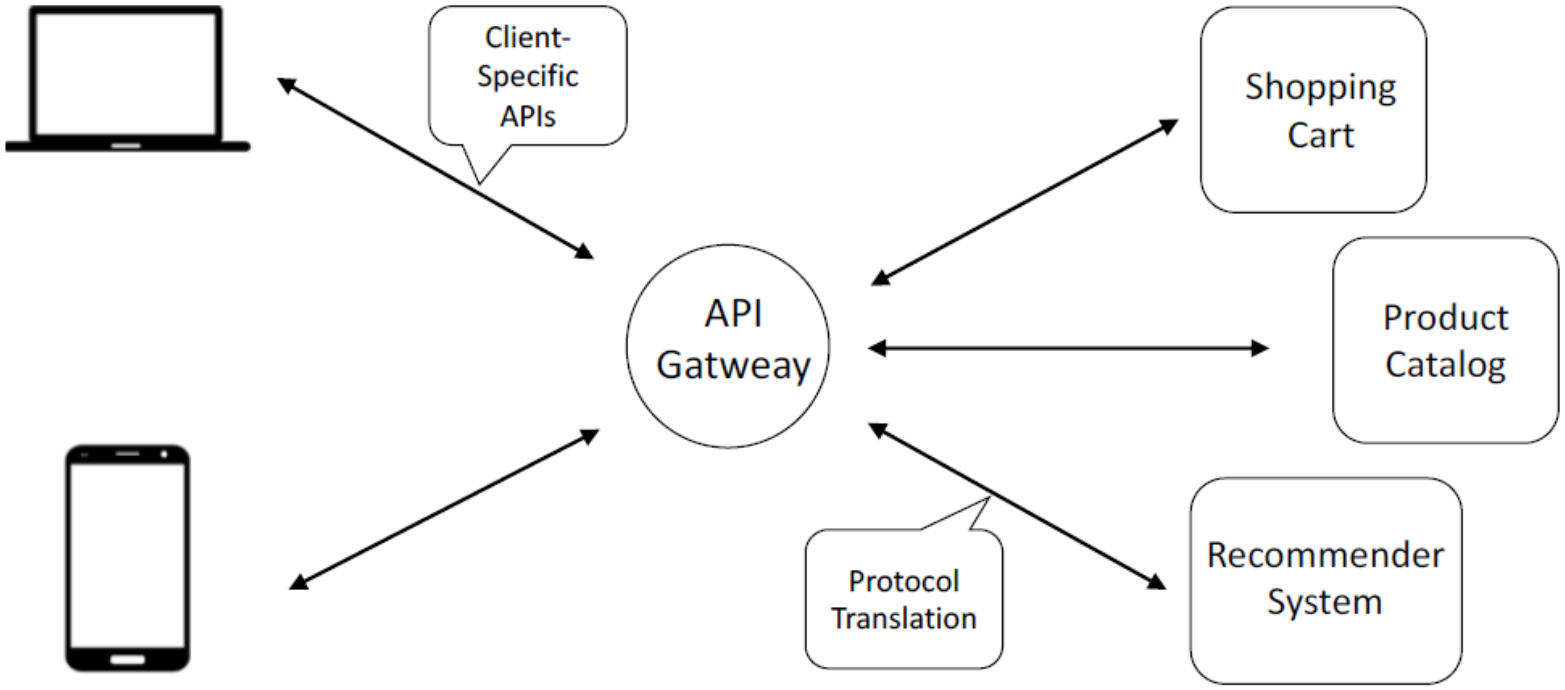
Microservice Pattern: Description

- Microfontend UI



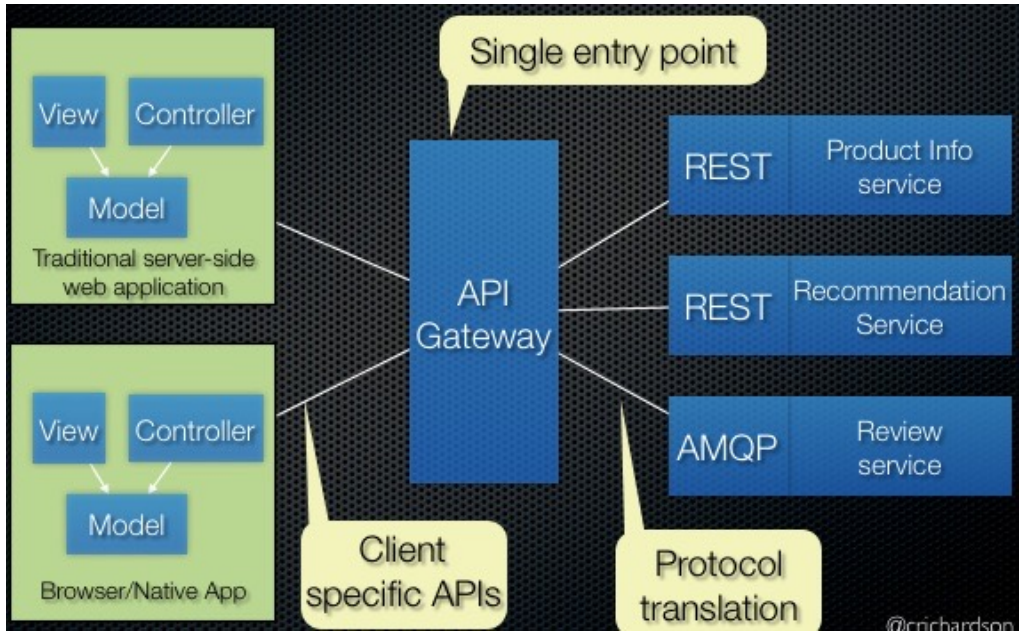
Microservice Pattern: Description

- **API-Gateway Architecture Pattern**
  - **API Gateway**는 시스템의 진입점으로의 역할을 수행
  - 요청을 적절한 마이크로 서비스로 라우팅하고 여러 마이크로 서비스를 호출



Microservice Pattern: Description

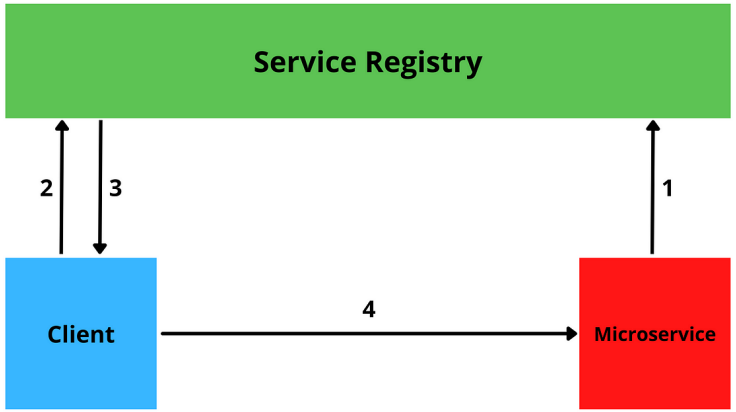
- API-Gateway Architecture Pattern
  - Context: the code that displays the product details needs to fetch information from all of multiple services.
  - Problem: How do the clients of a Microservices-based application access the individual services?
  - Solution: Implement an API gateway that is the single entry point for all clients
    - It handles requests by fanning out requests to multiple services



Microservice Pattern: Description

- Service Discovery Pattern
  - Context: In a modern microservice-based application, the number of instances of a service and their locations changes dynamically
  - Problem: How does the client of a service - the API gateway or another service - discover the location of a service instance?
  - Solution: The router queries a service registry, which might be built into the router, and forwards the request to an available service instance.

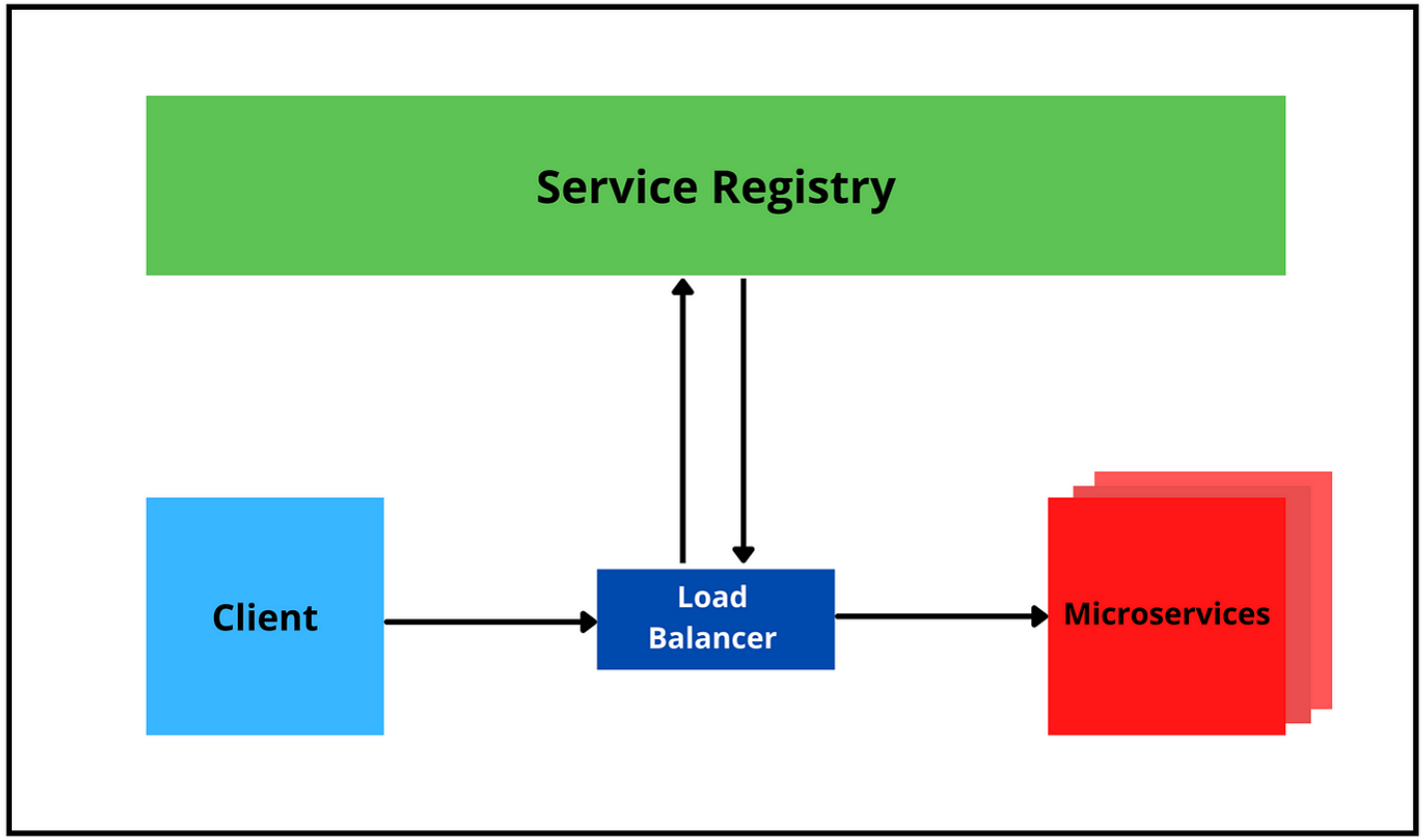
1.The service provider registers its location in the service registry.
2.The client looks for relevant service locations in the service registry.
3.The service registry returns the location of the required microservice.
4.The client directly calls the microservice.



https://microservices.io/patterns/server-side-discovery.html; https://blog.bitsrc.io/service-discovery-pattern-in-microservices-55d314fac509

Microservice Pattern: Description

- Server-side Discovery Pattern

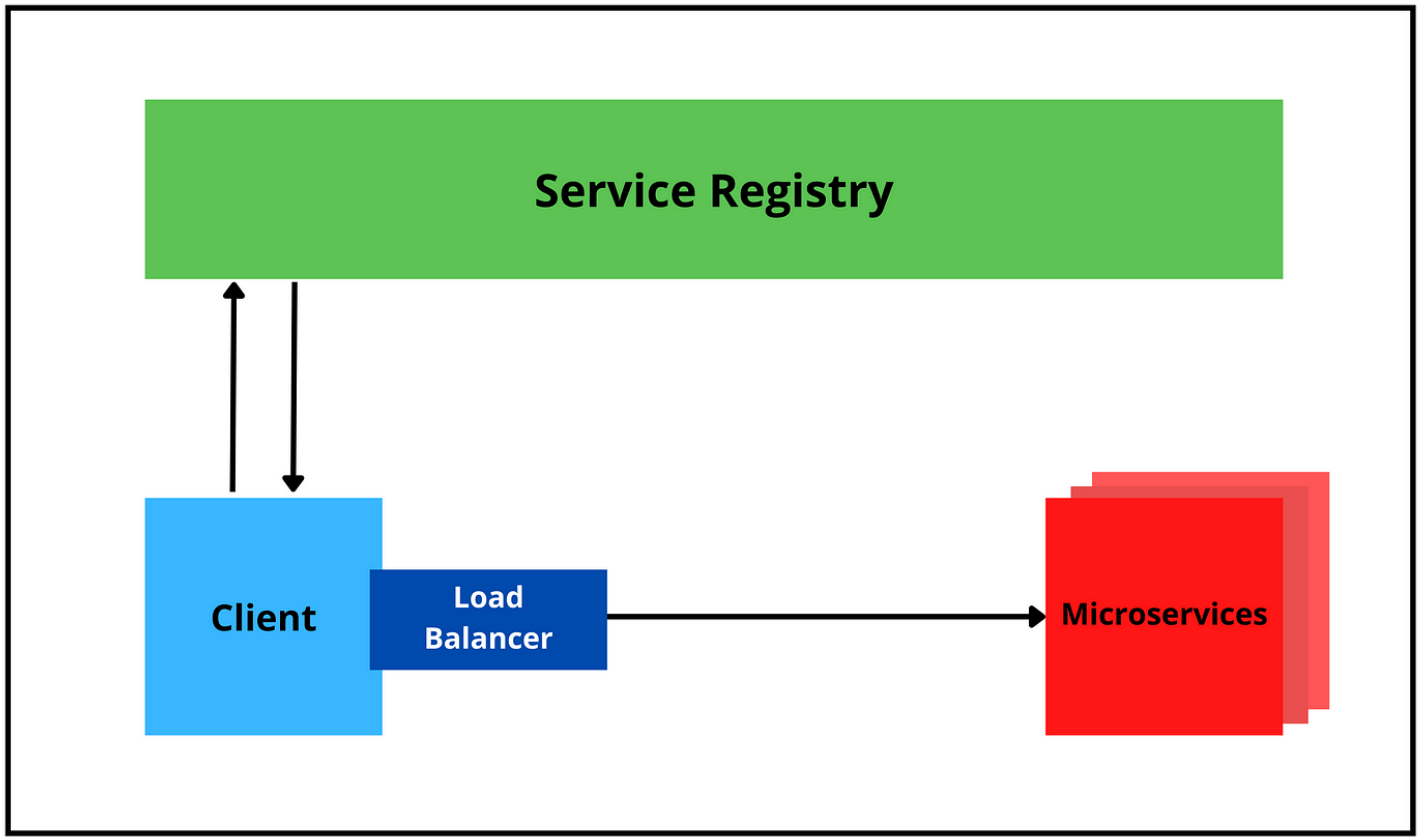


1.First, the client requests a microservice through the load balancer.
2.Then, the load balancer queries the service registry and finds the location of the relevant microservice.
3.Finally, the load balancer routes the request to the microservice.

https://blog.bitsrc.io/service-discovery-pattern-in-microservices-55d314fac509

Microservice Pattern: Description

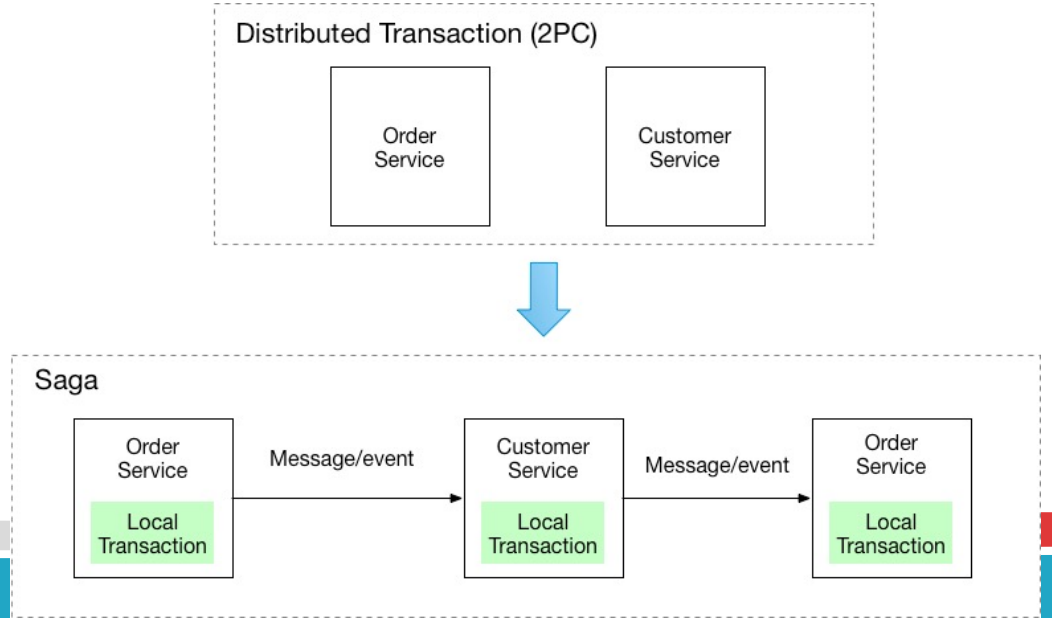
- Client-side Discovery Pattern



1.First, the client queries the service registry and retrieves the locations.
2.Then, the client uses its dedicated load balancer to select a microservice and sends the request.

Microservice Pattern: Description

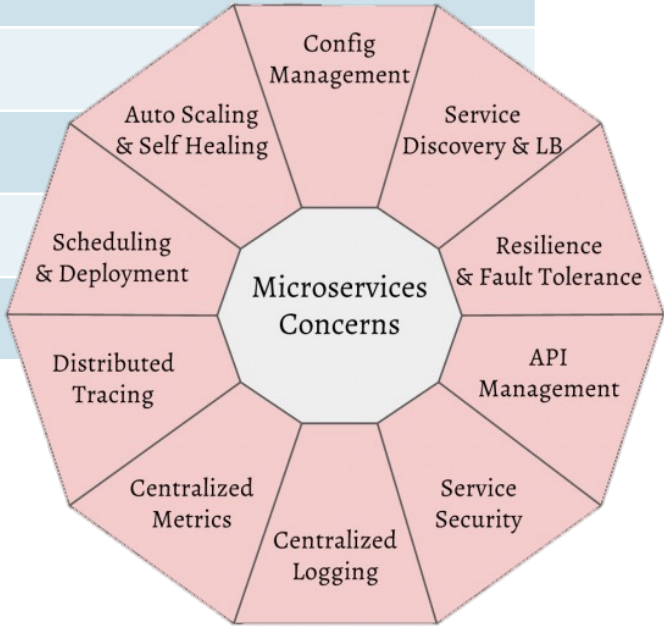
- **Saga Pattern**
  - 정황: 서비스마다 서로 다른 데이터 베이스를 가진다
  - 문제: 어떻게 **Transaction**을 여러 서비스에 적용할 것인가**?**
  - 해법: **Saga**는 일련의 지역적 **Transaction**들이다. 여러 개의 서비스에 적용하는 각 비즈니스 트랜잭션을 구현한다.



→ 마이크로서비스는 클라우드 환경에서 Kubernetes와 Kafka 위에 구현되는 경우가 대부분

Microservice Pattern: Realization

Concern	Technologies
Configuration Management	Netflix Archaius, Kubernetes Configmap
Service Discovery & Load Balancing	Netflix Eureka, Kubernetes Service, Istio
API Gateway	Netflix Zuul, Kubernetes Ingress
Centralized Logging	ELK Stack
Centralized Metrics	Netflix Spectator, Heapster
Distributed Tracing	Spring Cloud Sleuth, Zipkin
Resilience & Fault Tolerance	Resilience4j, Kubernetes Health check

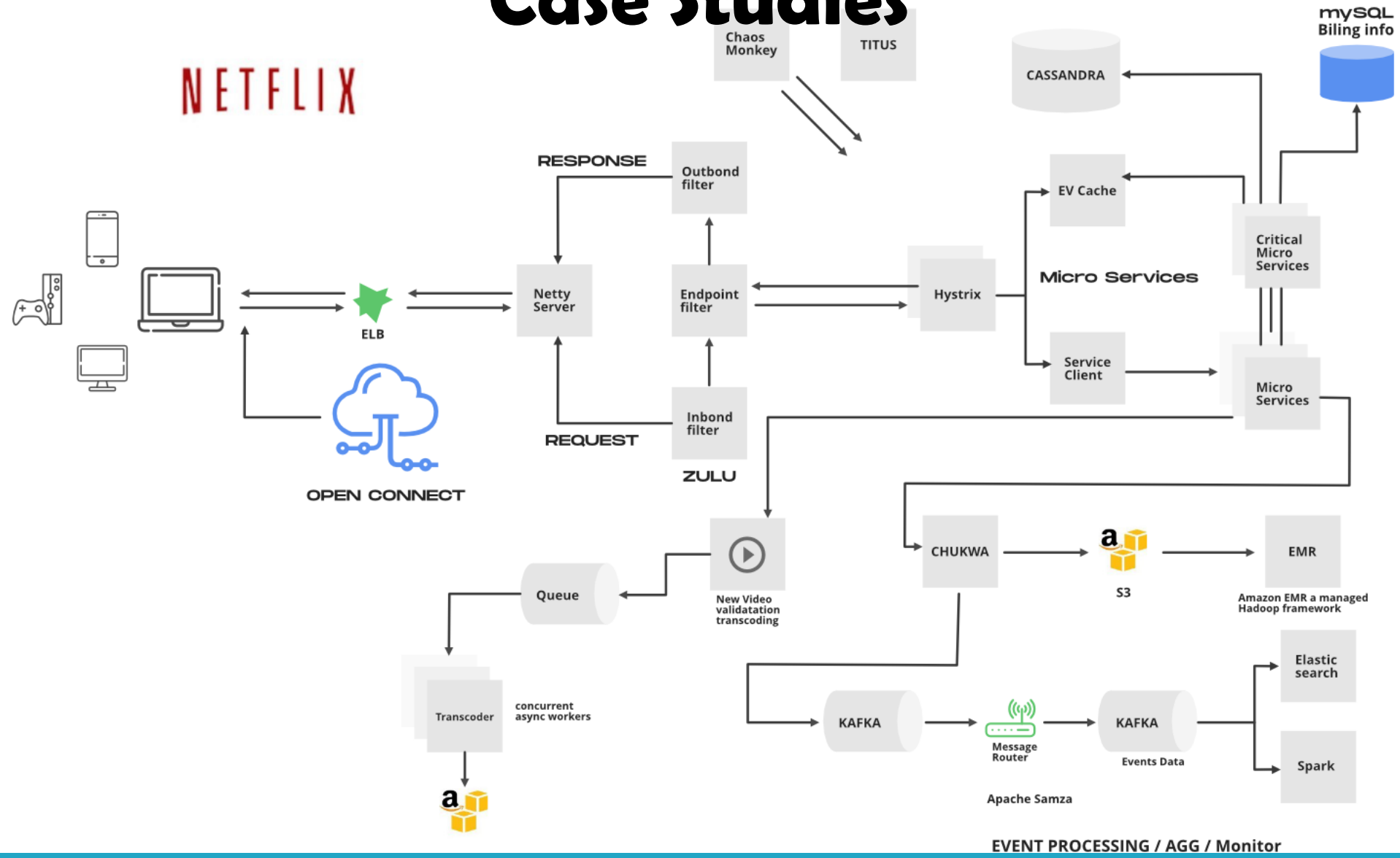


https://developers.redhat.com/blog/2016/12/09/spring-cloud-for-microservices-compared-to-kubernetes

Microservice Pattern: Realization

1. 개념 설계
   - 마이크로 서비스 식별: 서비스 대상의 경계 기준으로 나눔
   - 업무 별 관계 분석 및 화면 설계
   - 마이크로 서비스 설계
   - 마이크로 서비스를 위한 데이터 설계
   - 메시지 전송 방식 설계
2. 구조 설계
   - 마이크로서비스 프로젝트 구조 설계
   - 마이크로서비스 패키지 구조 설계
3. 마이크로서비스 아키텍처 구성

Microservice Pattern: Case Studies



Cloud native architecture case study - Neflix

- The journey
  - 2009: Migrating video master content system logs into AWS S3
  - 2010: DRM, CDN Routing, Web Signup, Search, Moving Choosing, Metadata, Device Management, and more were migrated into AWS
  - 2011: Customer Service, International Lookup, Call Logs, and Customer Service analytics
  - 2012: Search Pages, E-C, and Your Account
  - 2013: Big Data and Analytics
  - 2016: Billing and Payments

Cloud native architecture case study - Neflix

- The benefits
  - The growth Netflix was enjoying around 2010 and beyond made it difficult for them to logistically keep up with the demand for additional hardware and the capacity to run and scale their systems.
  - Elasticity of the cloud is possibly the key benefit for Netflix, as it allows them to add thousands of instances and petabytes of storage, on demand, as their customer base grows and usage increases.
  - These benefits have enabled Netflix to have a laser focus on their customer and business requirements, and not spend resources on areas that do not directly impact that business mission.

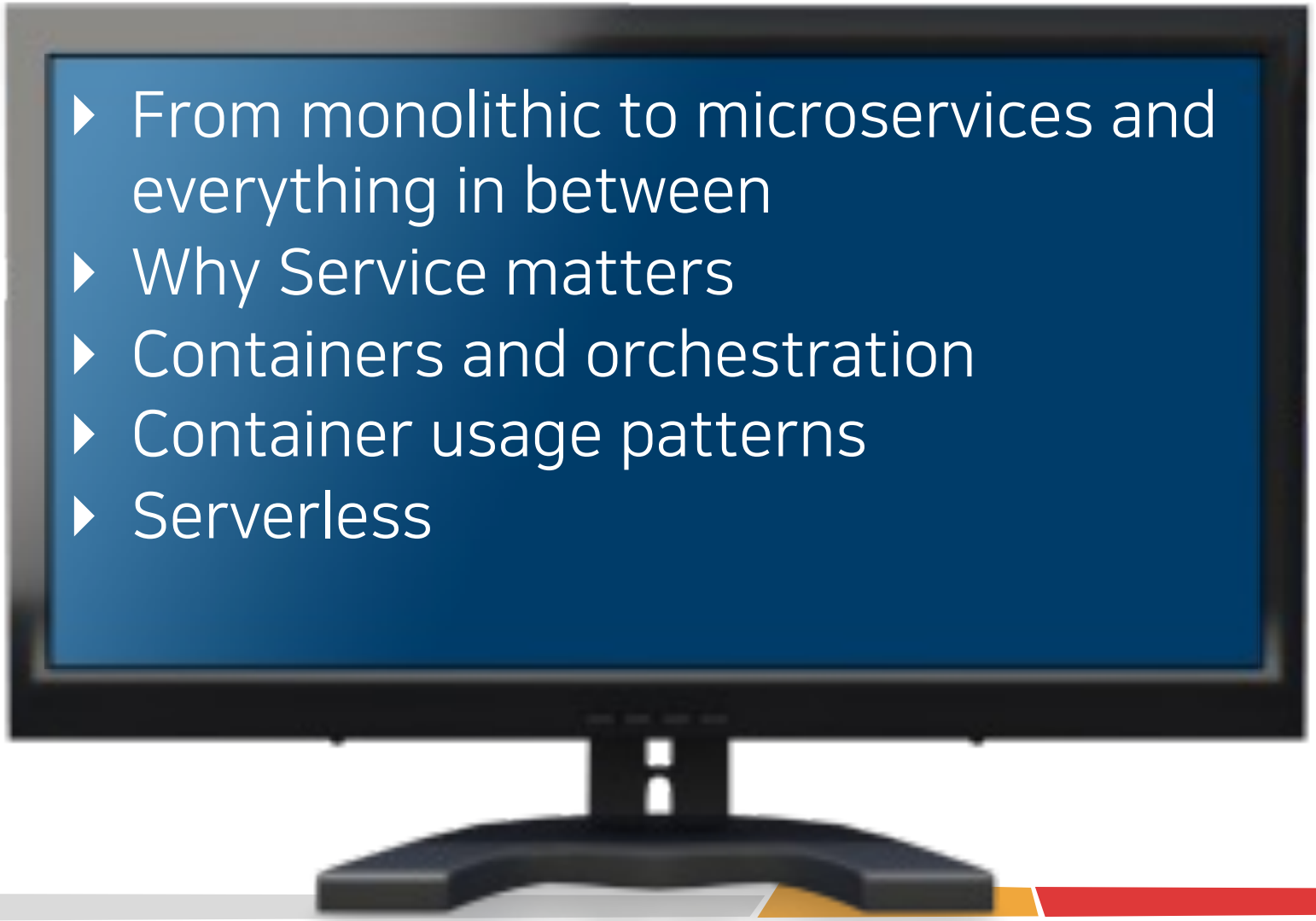
Microservice Pattern: Benefits

- 장점– 확장성, 탄력성, 진화성
  - 복잡한 애플리케이션의 지속적 배포, 배치가 가능하다.
  - 마이크로서비스는 상대적으로 작아, 이해하기 쉽고 생산적일 수 있다
  - 오류의 고립(**Fault Isolation**)을 돕는다.
    - 만약 메모리 릭이 있다면 해당 문제의 서비스만 진행을 멈추고 다른 서비스는 지속된다.
  - 하나의 기술에 대한 지속적인 유지를 할 필요가 없다.
    - 새로운 기술을 도입, 적용하기가 용이하다.

Microservice Pattern: Liabilities

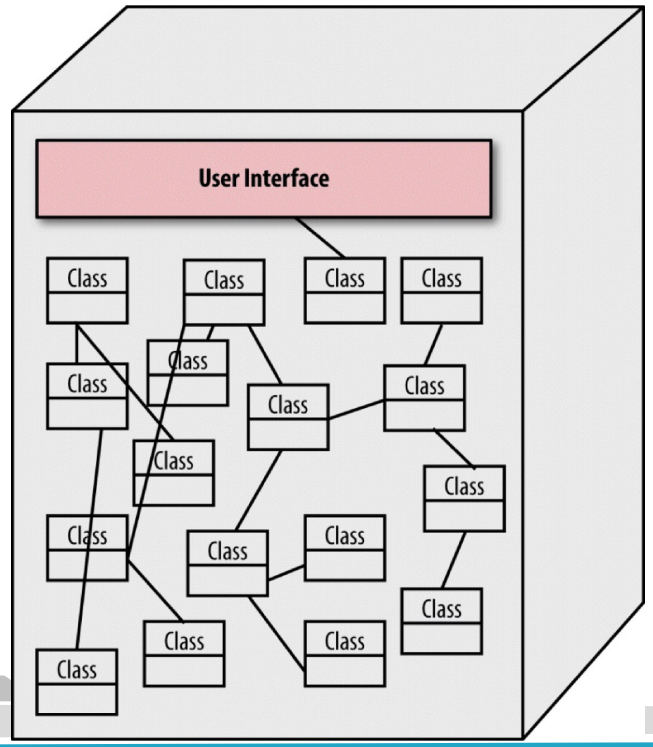
- 단점 – 성능, 단순성, 비용
  - 개발자는 분산 시스템에서 나오는 추가적인 문제들을 다루어야 한다.
  - 여러 서비스를 배치를 하는 것이 좀 더 복잡해 진다.
  - 메모리 사용량이 증가한다 (예를 들어 각 서비스가 각자 **VM**을 사용할 경우).

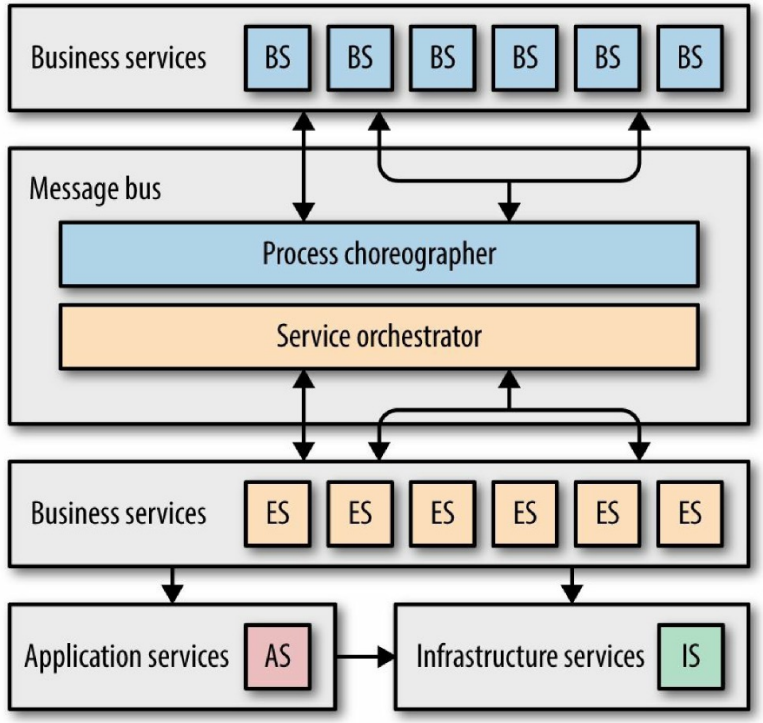
Cloud Native Application Design



From monolithic to microservices and everything in between

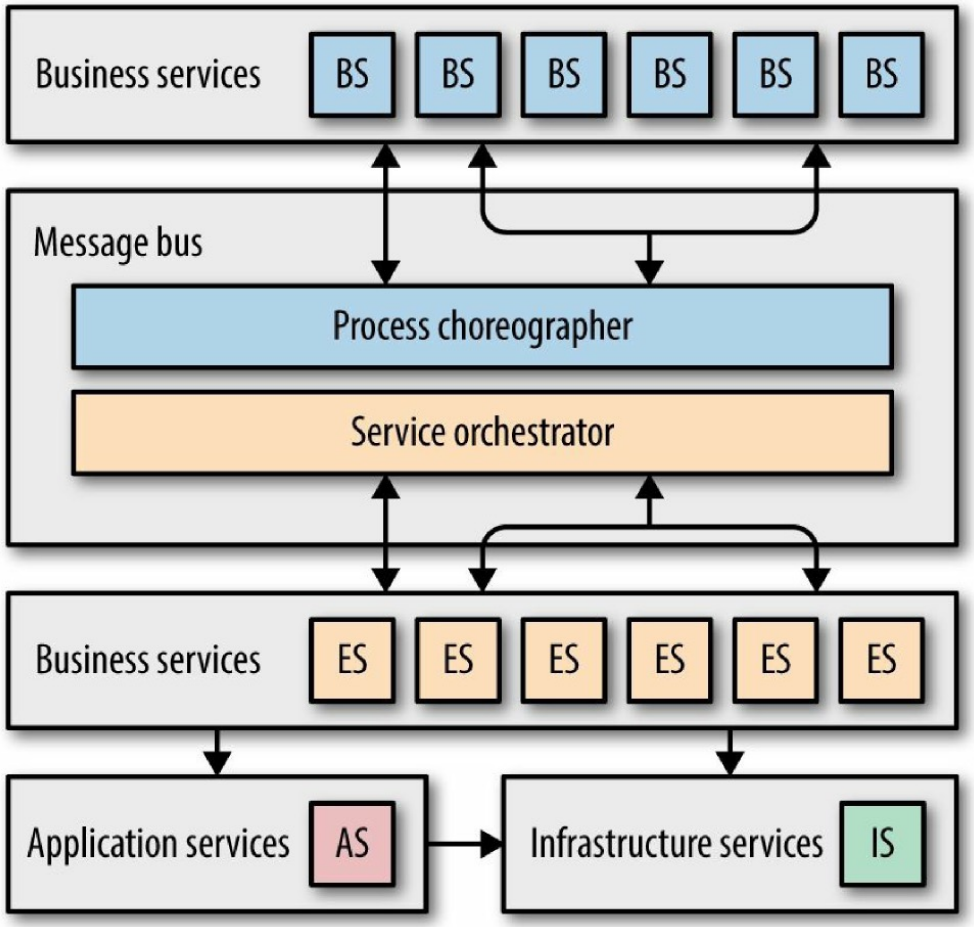
- Service-Oriented Architectures (SOAs)
  - The concept of breaking down the components that make up a monolith or server into more discrete business services





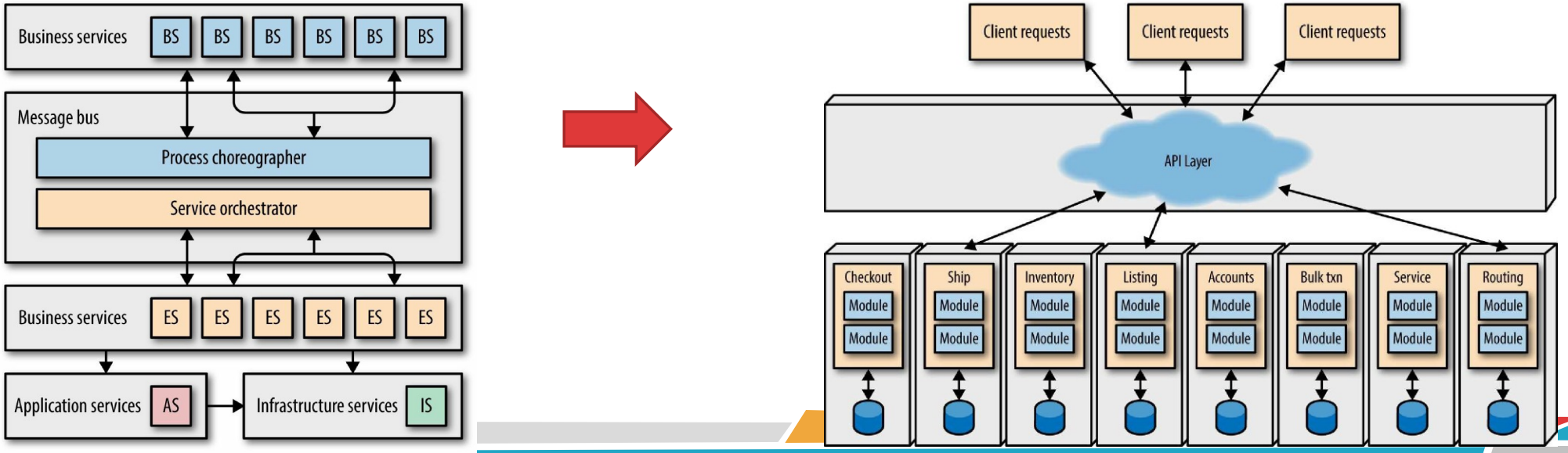
Service-oriented architectures, SOAs

- **Enterprise Service Bus (ESB)**
  - It is to mediate service requests
  - **It** allows for heterogeneous communication protocols and further service decoupling
    - often through the exchange of messaging
  - The ESB allows for message translation, routing, and other forms of mediation to ensure
    - The consumer service is receiving the messages in the right form at the right time



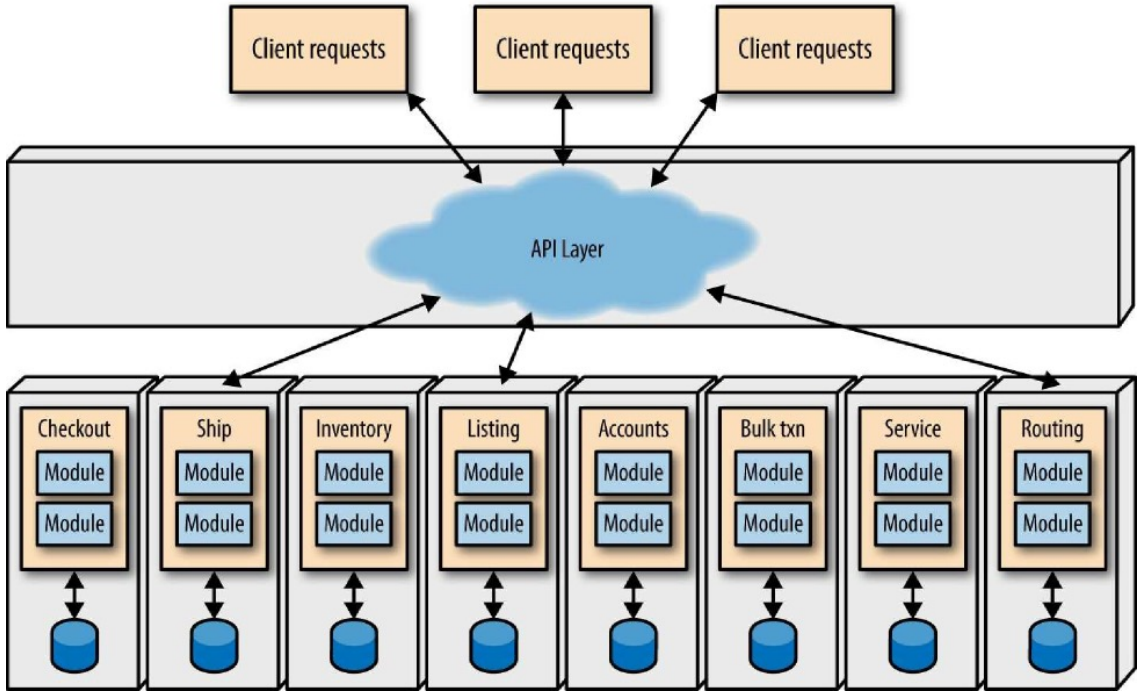
From monolithic to microservices and everything in between

- Microservices
  - that allows for a specific service to be broken down into discrete components
    - A service in an SOA is a black box that performs a full business function
    - Microservices are made up of that function, which is broken down to perform subsets of the larger business function



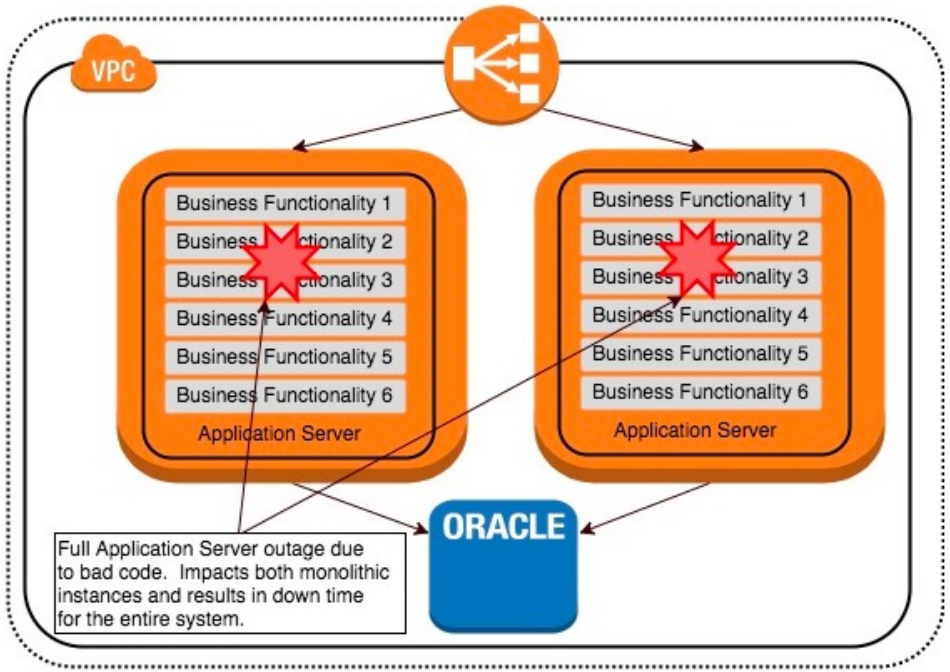
Microservices

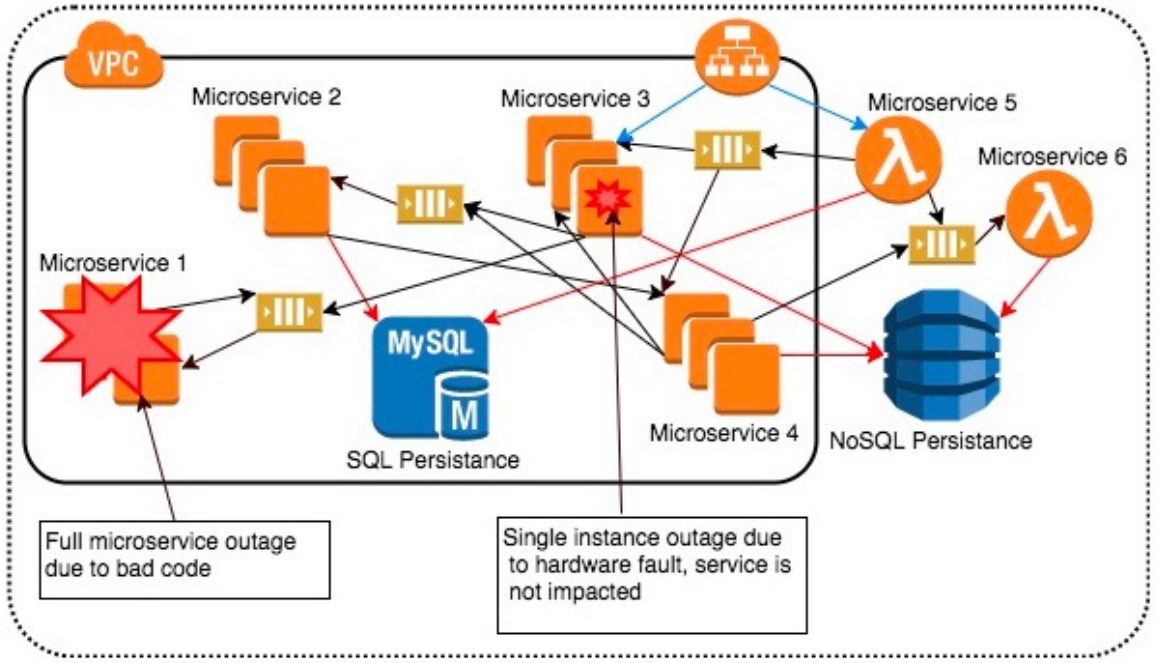
- **Two key differentiators when compared to SOA**
  - the scope of service
    - primitives that perform a subset of the main service
      - business logic or a workflow task
      - a database transaction, auditing, or logging
  - communication methods
    - microservices expose an API that can be called from any consumer service



Why service matter

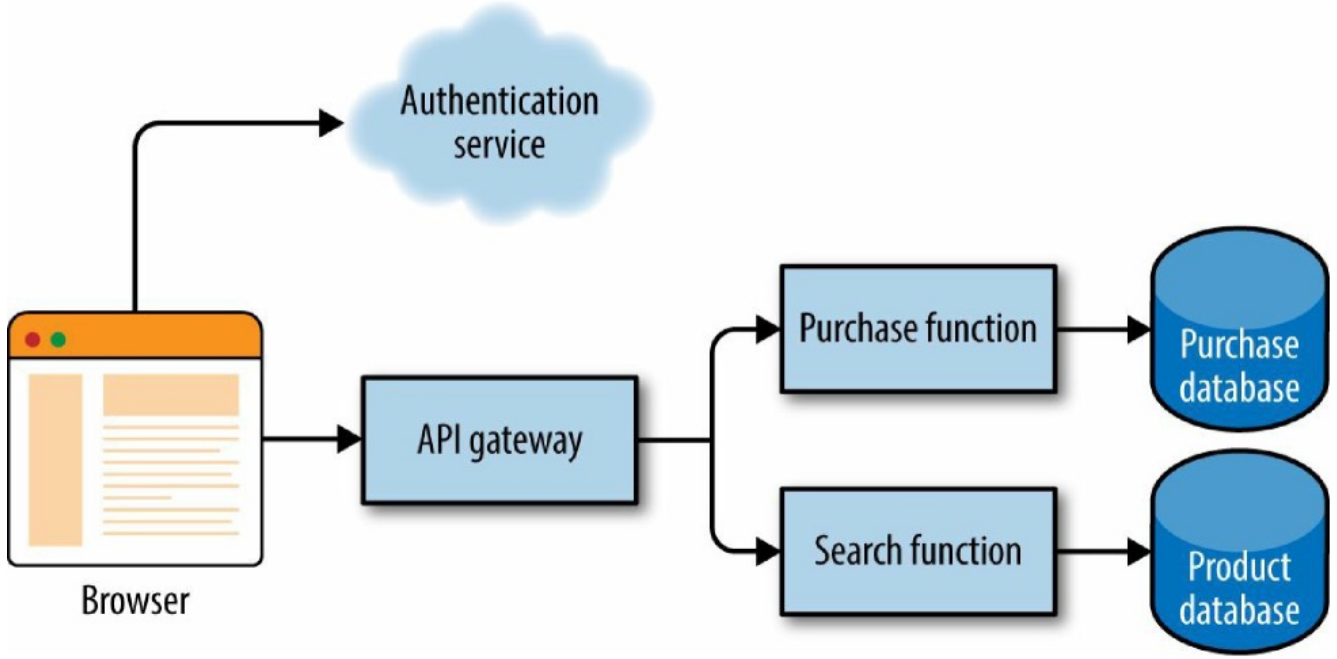
- Services matter
  - they allow for applications to be designed into ever smaller components





Containers and serverless

- There are multiple ways to construct a microservice with various technologies and approaches
  - containers and serverless approaches are the most common



Containers and orchestration

- Containers
  - a natural progression of additional isolation
  - virtualization of hardware
    - a virtual instance allows for a bare metal server to have multiple hosts
    - containers behave similarly but are much more lightweight, portable, and scalable
  - This is then treated as the deployment artifact
  - This is published to a registry to be deployed or updated across the fleet of containers already running in the cluster via the orchestration service

Containers and orchestration

- Registries
- Orchestration
- Container usage patterns
  - Microservices with containers
  - Hybrid and migration of application deployment
- Container anti-patterns

Registries

- A container registry
  - a public or private area for the storage, and serving, of pre-built containers
- many public hosted registries
  - the Docker Hub registry
  - that holds publicly available pre-built container files that have lots of common configurations ready to use
- it's recommended that hosted private registries are used
  - Why? Security and Performance
- Registries play a critical role in the CI/CD pipelines that will be used for deployment of applications.

Orchestration

- Deployment of new containers to the cluster is critical
  - It usually needs to be done with little or no downtime
  - making a service that performs those updates very valuable
- Kubernetes

Concept	Description
Kubernetes master	클러스터의 원하는 상태를 유지 관리
Kubernetes node	애플리케이션, VM, 물리서버 등, Master가 제어함
Pod/Service	모델에서 생성 배포하는 가장 작은 기본 빌딩 블록, 프로세스
Replica controllers and ReplicaSets	특정 수의 Pod 복제본을 한번에 실행되도록 함
Deployment	Pod, ReplicaSets 등의 선언적 업데이트 제공
DaemonSet	Node가 Pod의 복사본을 실행하도록 함, 삭제 시 생성 Pod 정리

Container usage patterns

- Microservices with containers
- Hybrid and migration of application deployment
- Container anti-patterns

Microservices with containers

- Containers are often synonymous with microservices
  - This is due to their lightweight properties
    - containing only a minimal operating system
    - containing the exact libraries and components needed to execute a specific piece of business functionality
- The use of containers for microservices works well for keeping small groups agile and moving fast
  - a logging framework, security approach for secrets, orchestration layer, and the CICD technology could be mandated
  - the programming languages would be open to what the team wants to use

Hybrid and migration of application deployment

- Containers are a great way
  - to run a hybrid architecture or to support the migration of a workload to the cloud faster and easier than with most other methods
- A container is a discrete unit
  - it's deployed on premises (such as a private cloud)
  - it's deployed in the public cloud (for example, AWS)
- It is recommended to use only a single cloud platform for cloud native applications
  - online transaction processing (OLTP) was properly handled would be successful

Container anti-patterns

- Putting multiple components or concerns on the same container is not the right way.
  - containers are not virtual machines
- Allowing SSH daemons (or other types of console access) on a container
  - It defeats the purpose of the container in the first place
- If the application requires multiple concerns or SSH access
  - consider using cloud-instance virtual machines instead of containers

Serverless

- Serverless does not mean the absence of servers
  - It does mean that resources can be used without having to consider server capacity
  - Serverless applications do not require the design or operations team to provision, scale, or maintain servers
- Function as a Service
  - The big three cloud providers each have their version
    - AWS Lambda
    - Azure Functions
    - Google Cloud Functions

Serverless

- Services
  - Compute
  - API proxy
  - Storage
  - Data stores
  - Inter process messaging
  - Orchestration
  - Analytics
  - Developer tools

Serverless

- Serverless usage patterns
  - Web and backend application processing
  - Data and batch processing
  - System automation
- Scaling
- Serverless anti patterns and concerns

Scaling

- Serverless applications are able to scale by design
  - Developers do not need to understand individual server capacity units
- Consumption units are the important factor for these services
  - Typically, throughput or memory which can be automatically updated to have more or less capability as the application requires
- Thread or function will only execute when called
  - Charging the company per execution at a 100 millisecond amount

Serverless anti patterns and concerns

- Long-running requests are often not valid
  - given the short-lived time for a function (less than 5 minutes)
- An event source executes, triggering the function more than expected
  - This scenario would saturate the CPU quickly
  - bog down the instance until it crashed
  - stop the service

Question?

Seonah Lee
saleese@gmail.com