

# [실습] 프로젝트 import

1. 아직 Clone 안했다면 Clone!

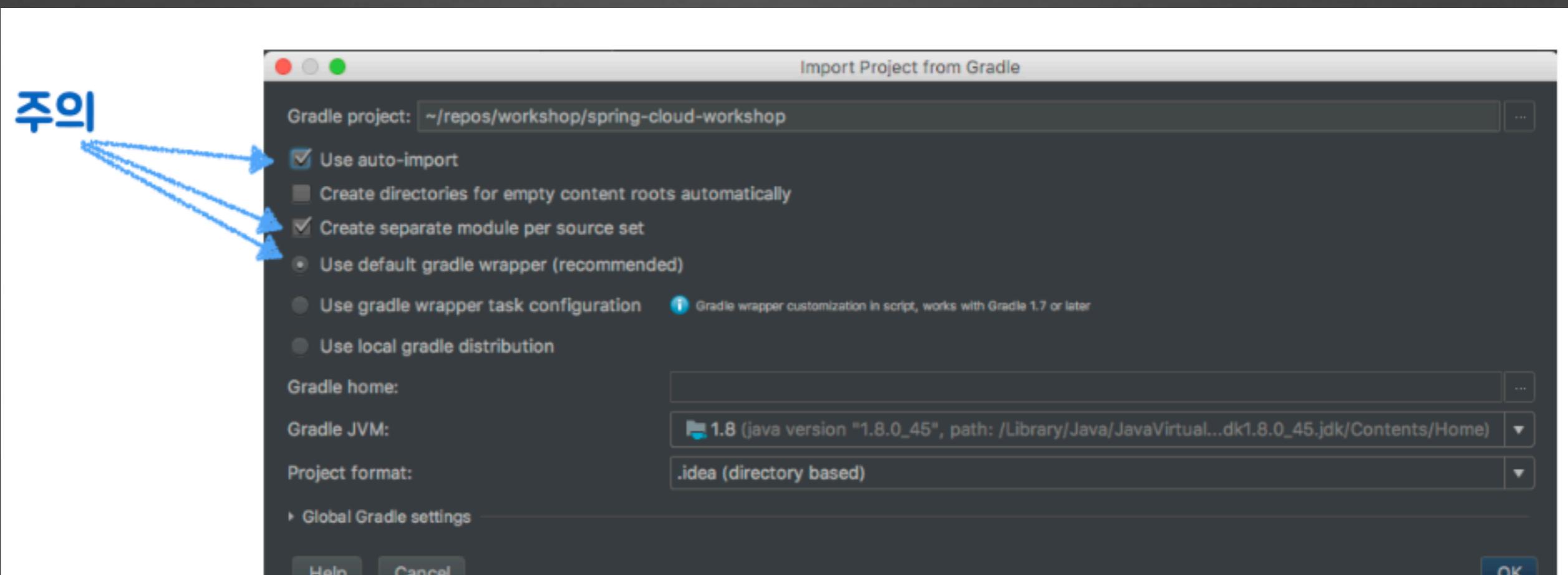
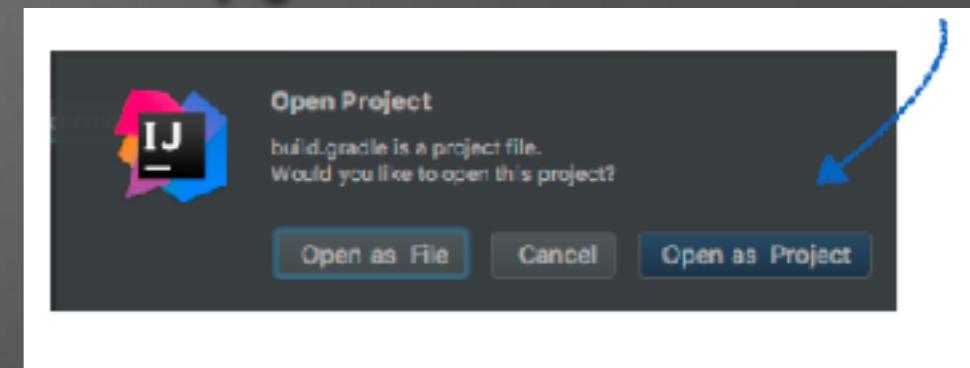
> git clone https://github.com/Gunju-Ko/spring-cloud-workshop.git

2. clone 한 directory 로 이동

> cd spring-cloud-workshop

3. IntelliJ에서 build.gradle import

File > Open - root 폴더 build.gradle 선택 - Open as Project



# 시작하기에 앞서...

- 세미나 도중 언제라도 손 들고 질문해주세요
- 실습이 제대로 동작하지 않을 경우 손을 들고 계시면 발표자나 뒤에 있는 TA 가 자리로 가겠습니다
- 실습에서 사용되는 소스 코드는 아래의 주소에 있습니다.
  - <https://github.com/Gunju-Ko/spring-cloud-workshop>
- 실습의 완성된 코드는 master branch 에 있습니다. 실제 실습은 별도 Branch 에서 진행합니다
- 특정 단계에서 실습에 못 따라 오시더라도 각 단계별로 'tag' 를 부여해 놨으니 막바로 Checkout 해서 시작하시면 됩니다
- 모든 자료의 출처는 오른쪽 하단에 있습니다. 실습 부분은 윤용성님 Workshop 자료와 99% 동일합니다

## 현재 작업 하던 폴더 초기화

```
git reset HEAD --hard # Dirty 파일들 초기화
```

## 실습을 건너뛰고 막바로 특정 단계에서 시작하기

```
git checkout tags/<tag 이름>
```

혹은

```
git checkout tags/<tag 이름> -b <새로운 브랜치 이름>
```

# Netflix OSS, Spring Cloud 를 활용한 MSA 구축 및 개발

11번가  
플랫폼 엔지니어링팀

임성묵

w/ TA 고건주

# 발표자 소개

- 이름 : 임성묵
- 이메일 : ipes4579@gmail.com
- 개발하고, 운영하고, 경험해 본 것들
  - (전) 스타트업
    - 소셜 네트워크 서비스
  - (전) SK Planet
    - 빌드 배포 시스템
    - Software Quality Engineering
  - (현) 11번가
    - Vine
    - MSA Platform
    - Event Driven Architecture Platform



# 세미나 목표

- 모놀리틱이 나온 배경과 아키텍쳐를 이해한다
- Cloud Native 한 MSA 를 이해한다
- Netflix OSS, Spring Cloud 를 통해 MSA 플랫폼을 구축한다

- 시작하기에 앞서
- 모놀리틱 아키텍쳐 <- 여기 할 차례예요
- Microservices Architecture(MSA)
- Cloud Native
- Netflix OSS, Spring Cloud
  - Hystrix - Circuit Breaker
  - Eureka - Service Discovery
  - Zuul - API Gateway
- Configuration Server, Tracing, Monitoring, 그리고 남은 이야기..

- 10년 전으로 거슬러 올라가 보겠습니다
  - Amazon Web Services 가 없던 시절
  - E-commerce 스타트업 시작
    - 이름 : 12번가
    - 업종 : 오픈마켓
    - 개발자 3명

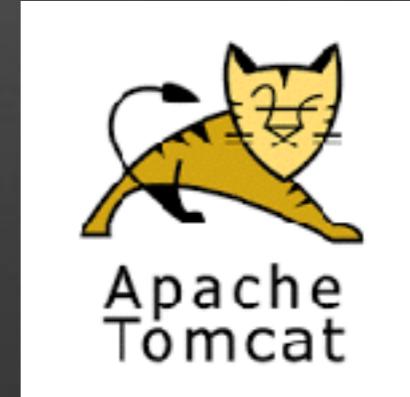
개발자가 개발한 코드는  
Tomcat(WAS)에 의해 실행되고  
프로그램의 상태(State)는  
데이터베이스에 저장



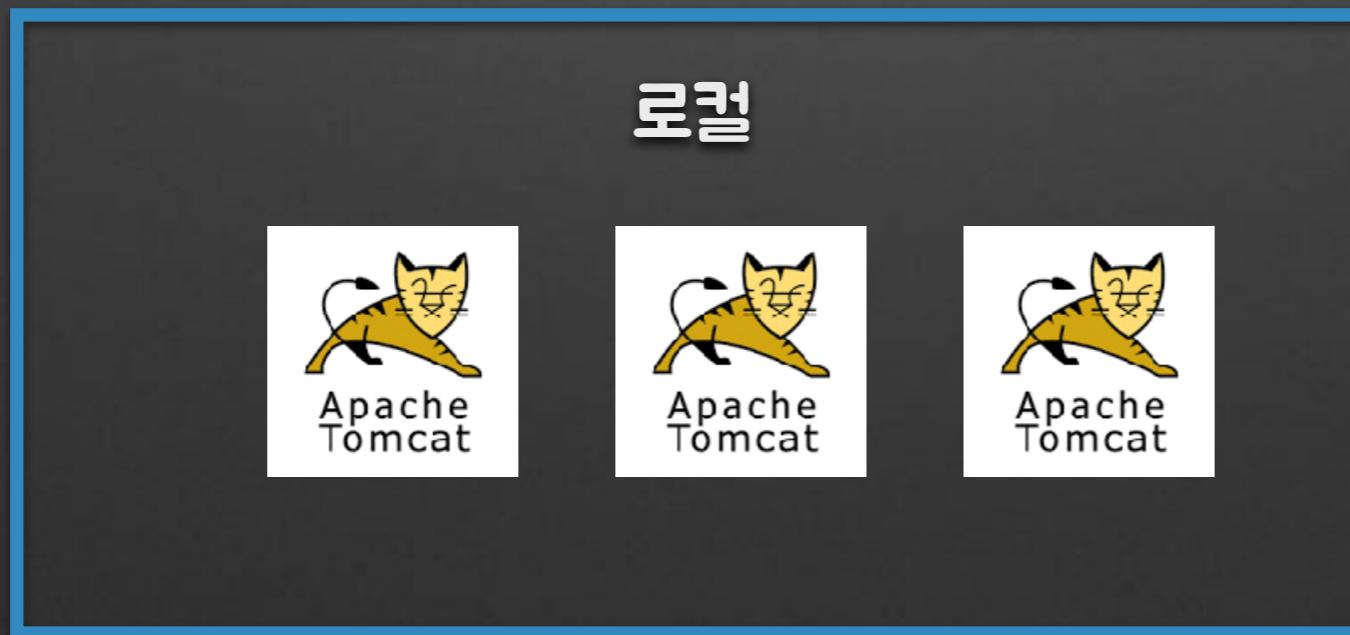
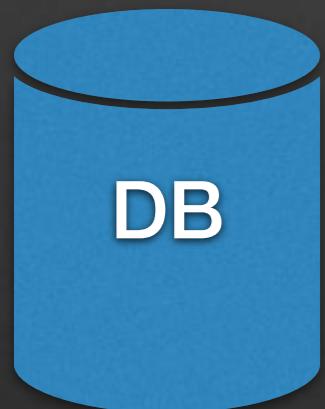
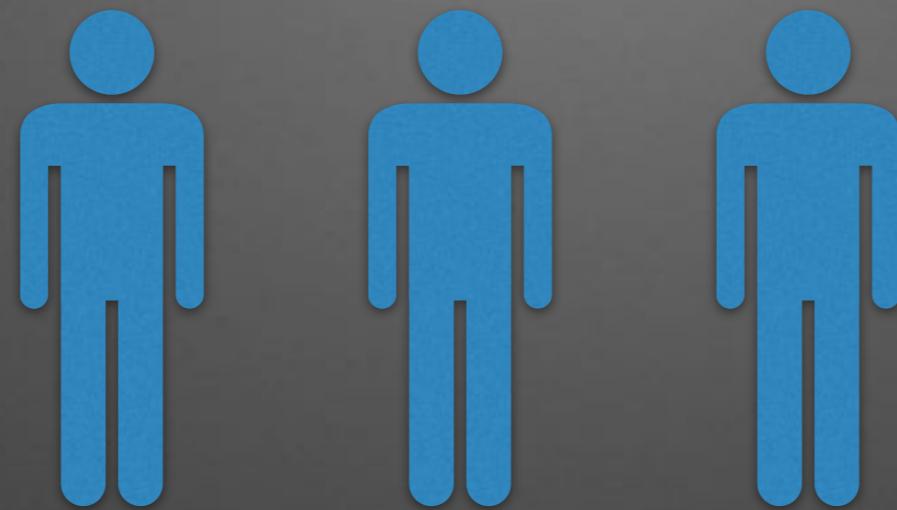
개발자



로컬



개발자가 여러명이 되면 코드를  
SCM(Source Code Management) 으로 관리  
DB 는 공유해서 사용

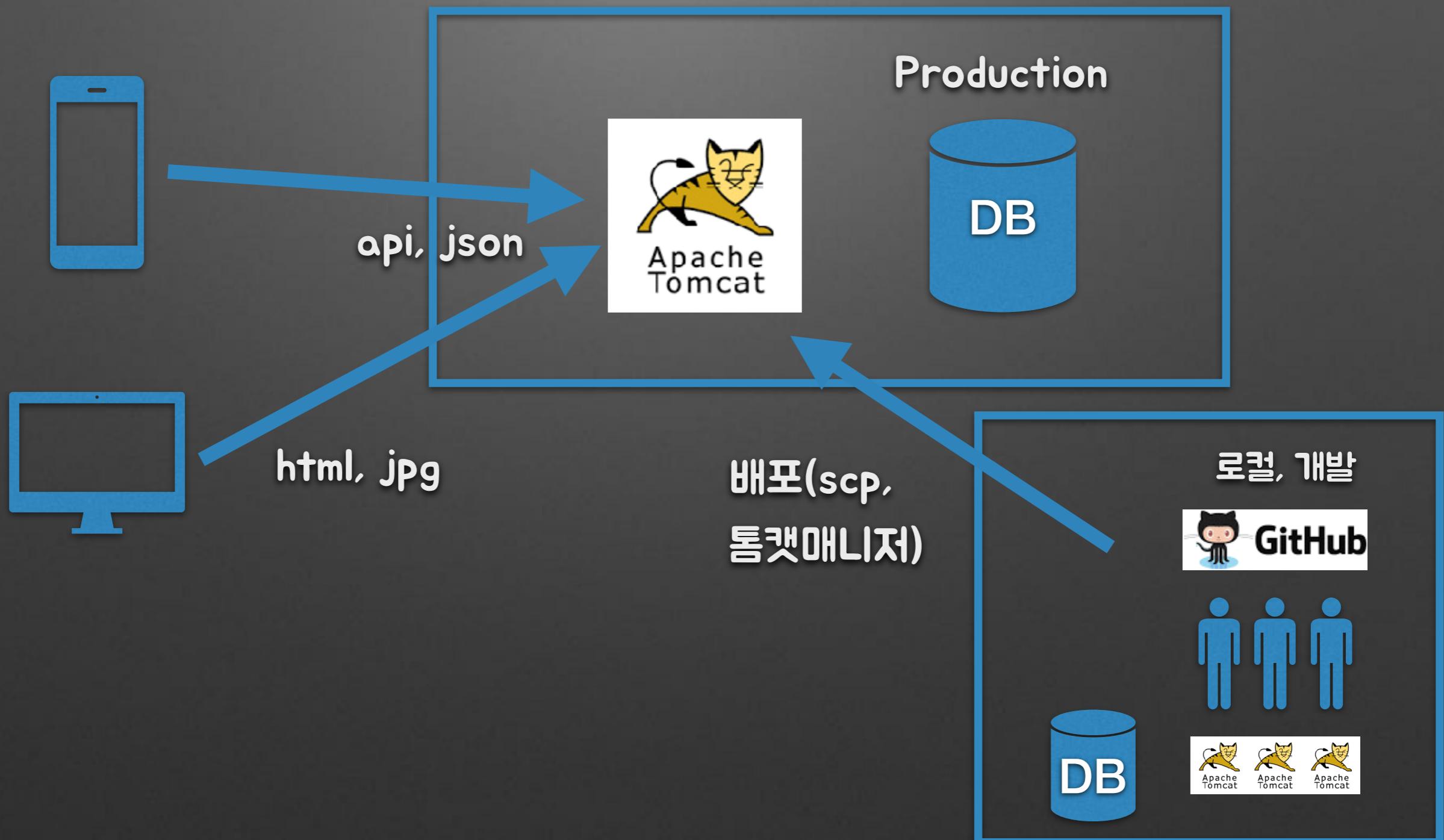


최초 상용 배포  
톰캣 1대, DB 1대 구입

무중단 배포 불가능!

SCP 를 통해 Clean 배포(stop -> delivery -> start)

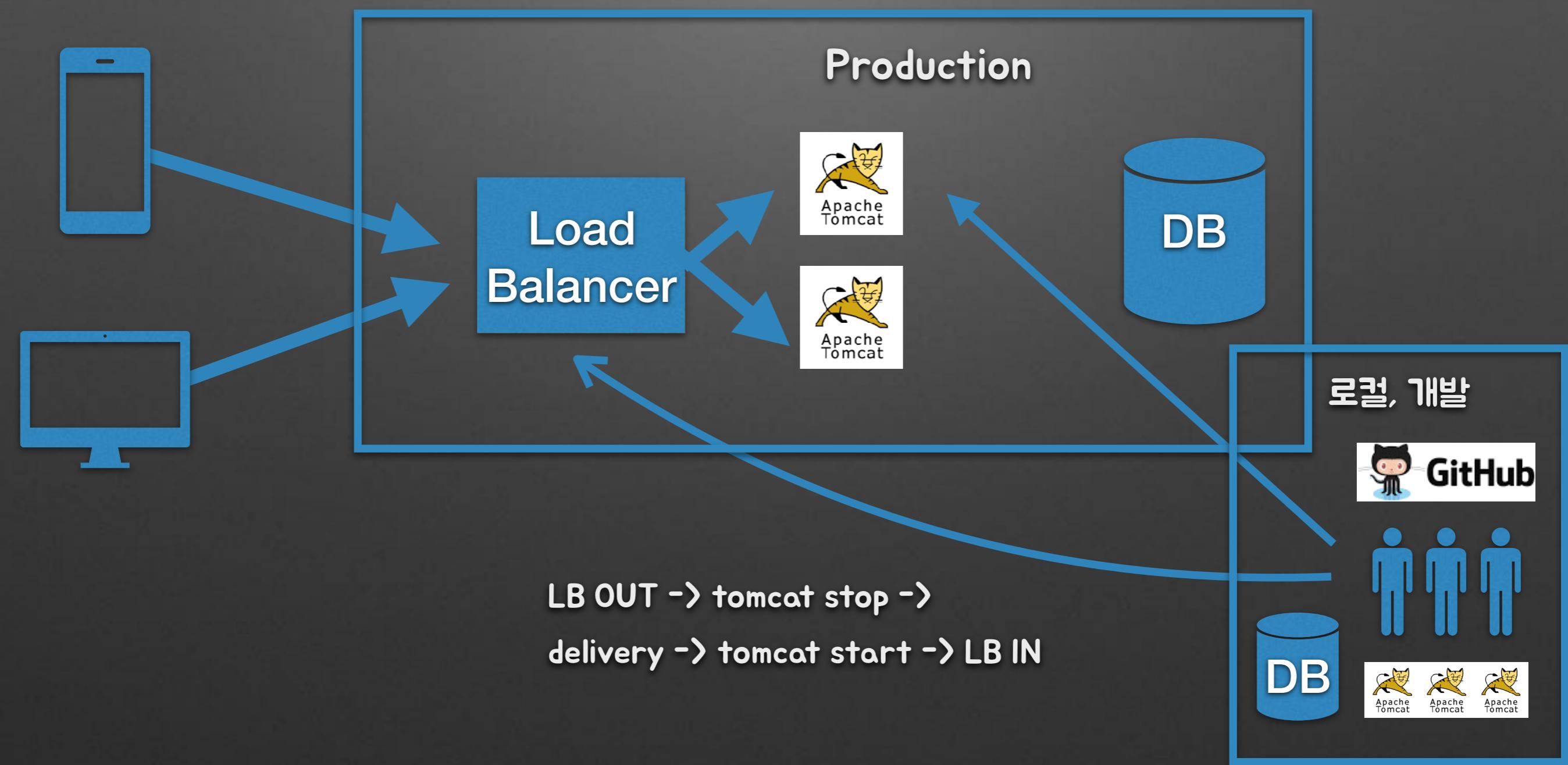
DNS(12st.com) -> Tomcat



# 서버 1대 추가 구입

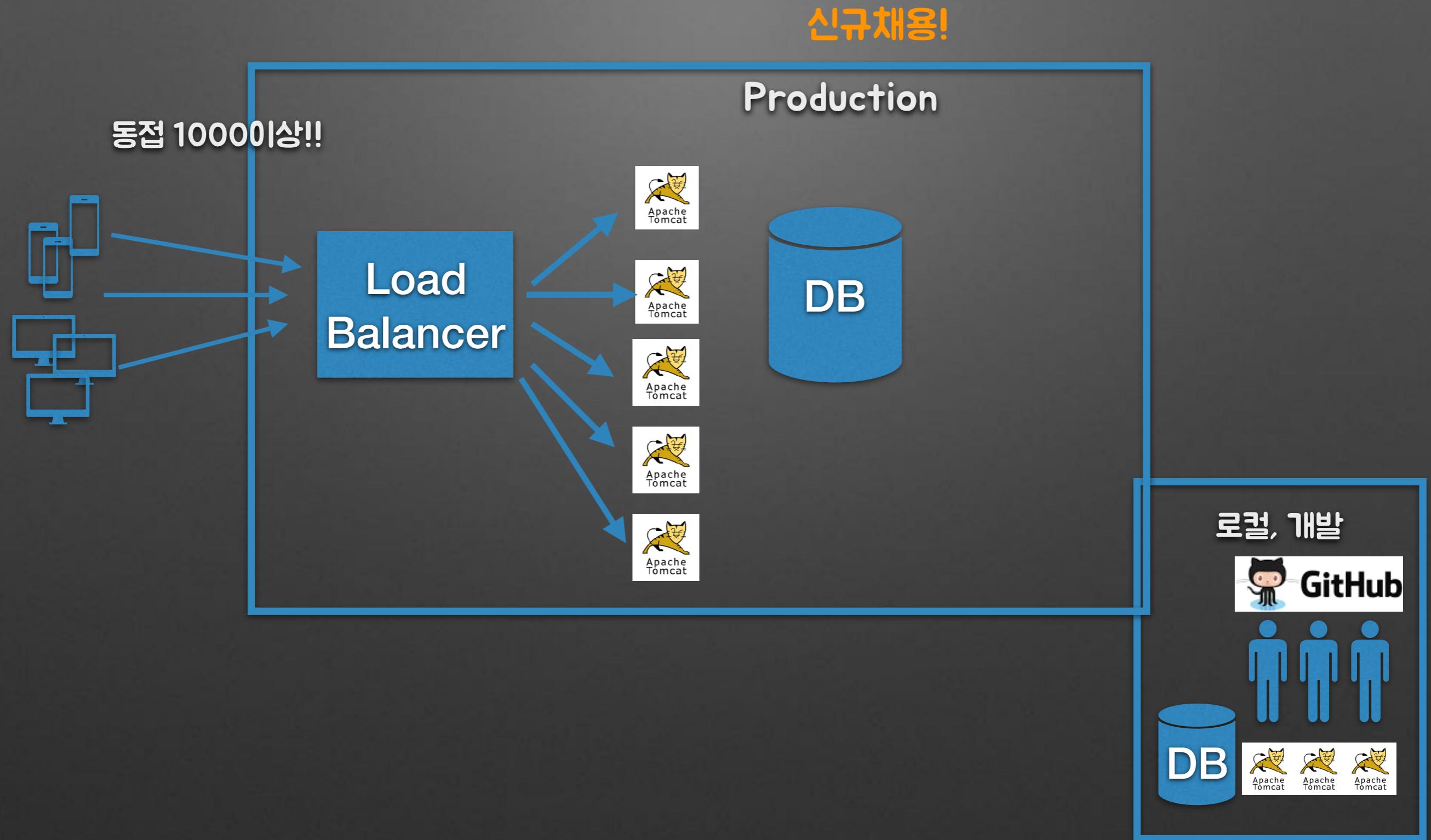
HA(High Availability) 구성 - 지속적으로 구동되는(uptime) 시스템

Load Balancer - L4 switch(hardware), L7(Nginx, HAProxy)



# 서버 3대 추가 구입

LB 설정 변경, 배포 방식 변경(Jenkins, Ansible, Chef, etc..)



# 조직개편. 상품팀, 주문팀으로 분리

## 단일 Repository

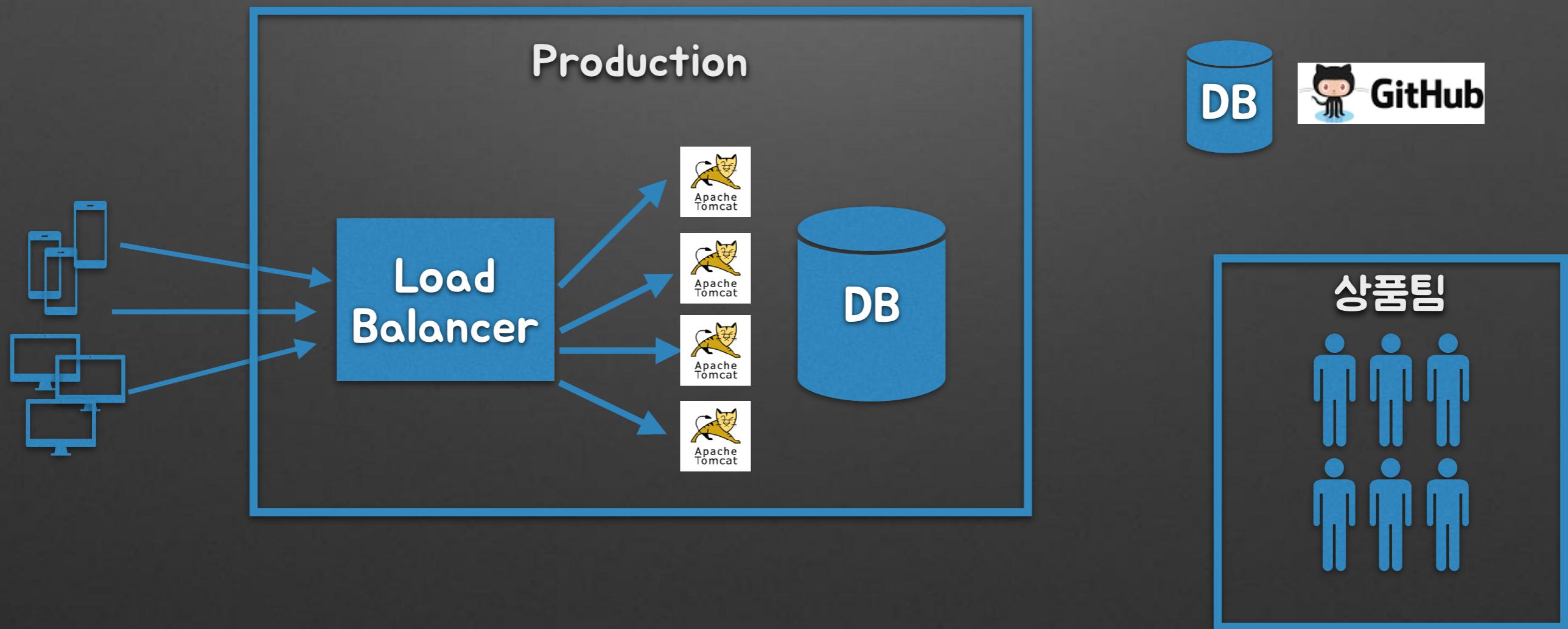
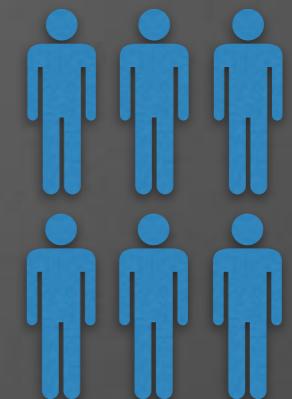
Issue 1. Branch Merge 시 Conflict

Issue 2. QA 를 어디까지? - 정기배포일

Issue 3. 각자 다른 일정, 배포 이슈

Issue 4. etc,.....

주문팀

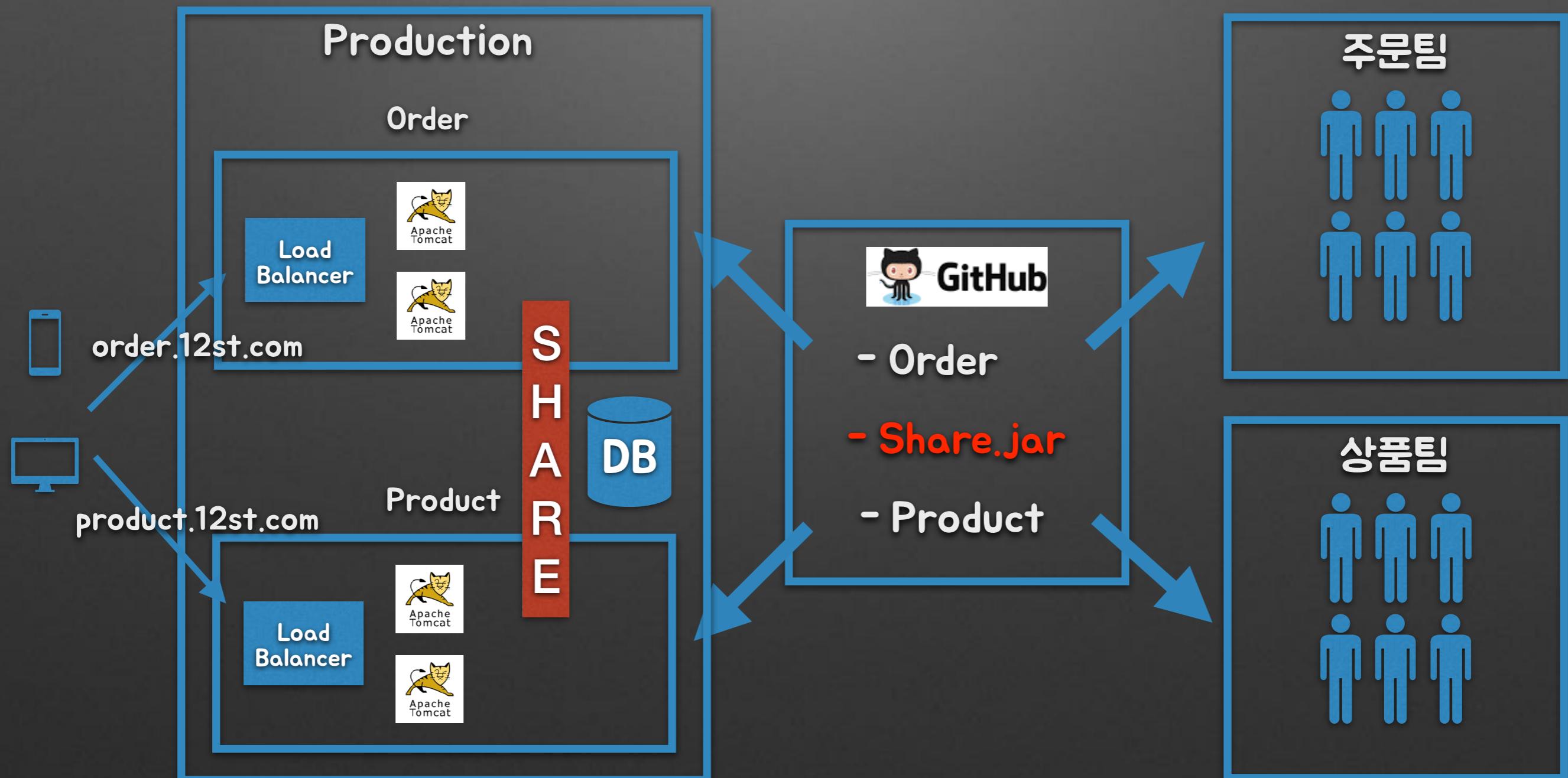


팀 별 Repository 분리, 각 코드 commit 시 팀별 배포 가능

공통적인 부분들을 Share Repository 에 넣음

각 프로젝트에서 compile 'kr.co.12st:share:1.0.0-SNAPSHOT'

Issues : 여전히 정기배포 필요(sync)



# 회사는 더더 성장하는데…

- 개발자 150명+
- 회원팀, 상품팀, 주문팀, 정산팀, 셀러팀, 등등.. 그리고 지원 조직(DB팀,...)
- 하나의 DB. (share.jar에서 접근해야 하고.. 그게 쉬우니까)
- 수백만 라인의 공통 코드(소스 코드만 100MB 이상)
- copy & paste에서 오는 중복과 복잡도

# 콘웨이의 법칙(Conway's Law)

- (광범위하게 정의하면) 모든 조직은 조직의 의사소통 구조(communication structure)와 똑같은 구조를 갖는 시스템을 설계한다. - Melvin Edward Conway, 컴퓨터 과학자
- 콘웨이 법칙은 조직도에 초점을 두지 않고, 실질적인 소통 관계에 관심을 둔다. 너무 많은 통신 관계를 갖는 것은 프로젝트에 대한 진정한 위험이 된다

모놀리틱 Application

인프라 팀, 배포 팀

주문팀

상품팀

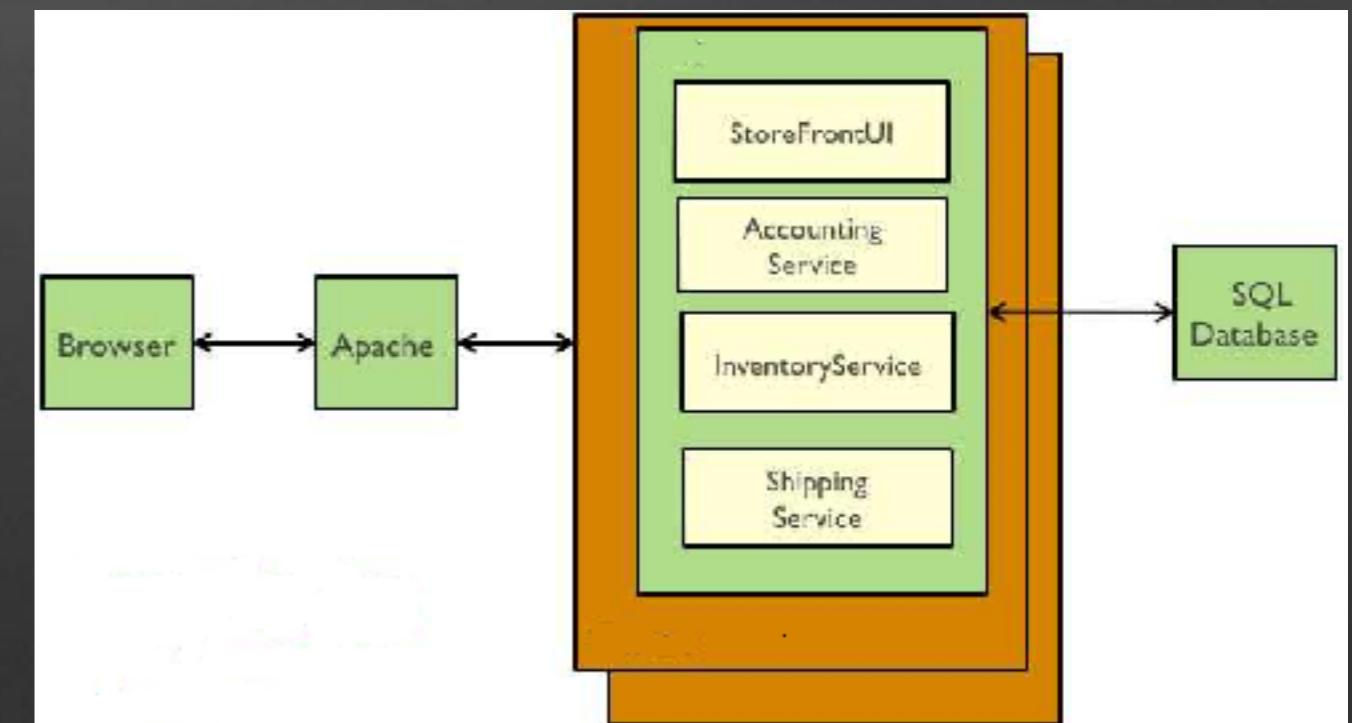
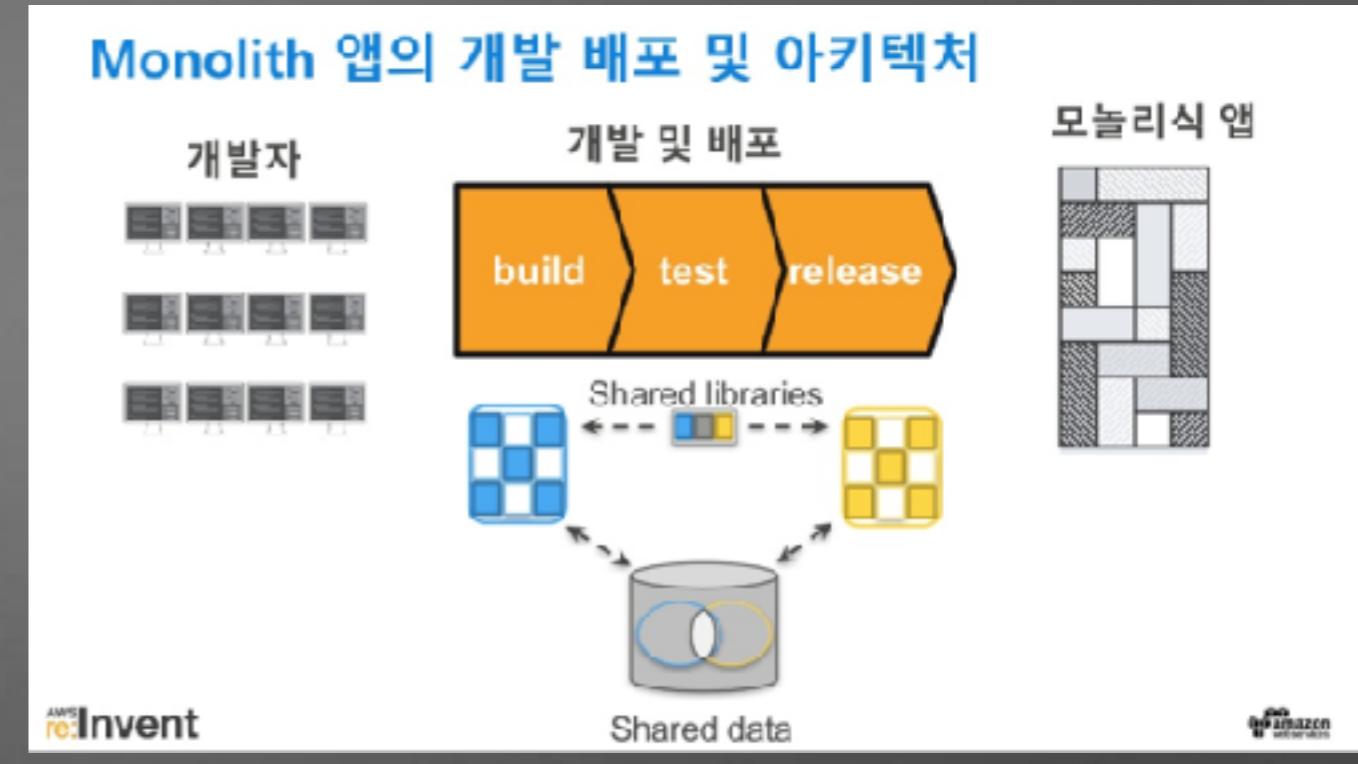
정산팀

UI팀

DB팀

# 모놀리틱 아키텍쳐 정리

- 대부분 IT 회사의 시작은 모놀리틱(Amazon, Netflix, 11번가, etc...)
- 장점
  - 개발이 단순하다(repository 하나 체크아웃 받아서 띄우면 되니까..)
  - 배포가 단순하다(war 하나만 배포하면 되니까..)
  - Scale-out 이 단순하다(서버 하나 복사하면 되니까..)
    - 하지만, DB 성능으로 인한 한계가 있다
- 단점
  - 무겁다 - IDE 가 못 받쳐줌
  - 어플리케이션 시작이 오래 걸린다
  - 기술 스택 바꾸기가 어렵다
  - 높은 결합도
  - 코드베이스의 책임 한계와 소유권이 불투명



# 이쯤에서 회사는 결정을 하게 됩니다

- 새로운 언어, 팔끔한 코드로 전면 재개편하거나
  - 차세대 프로젝트, 빅뱅
  - Netscape
- MSA 플랫폼을 구축하여 기존 Legacy 를 고사(strangle)시킵니다
  - Amazon, Netflix, 11번가, etc...

- 시작하기에 앞서
- 모놀리틱 아키텍쳐
- **Microservices Architecture(MSA) <- 예기 할 차례예요**
- Cloud Native
- Netflix OSS, Spring Cloud
  - **Hystrix - Circuit Breaker**
  - **Eureka - Service Discovery**
  - **Zuul - API Gateway**
- Configuration Server, Tracing, Monitoring, 그리고 남은 이야기..

“MSA란,  
시스템을 여러개의 독립된 서비스로 나눠서,  
이 서비스를 조합함으로서 기능을 제공하는  
아키텍쳐 디자인 패턴”

-조대협

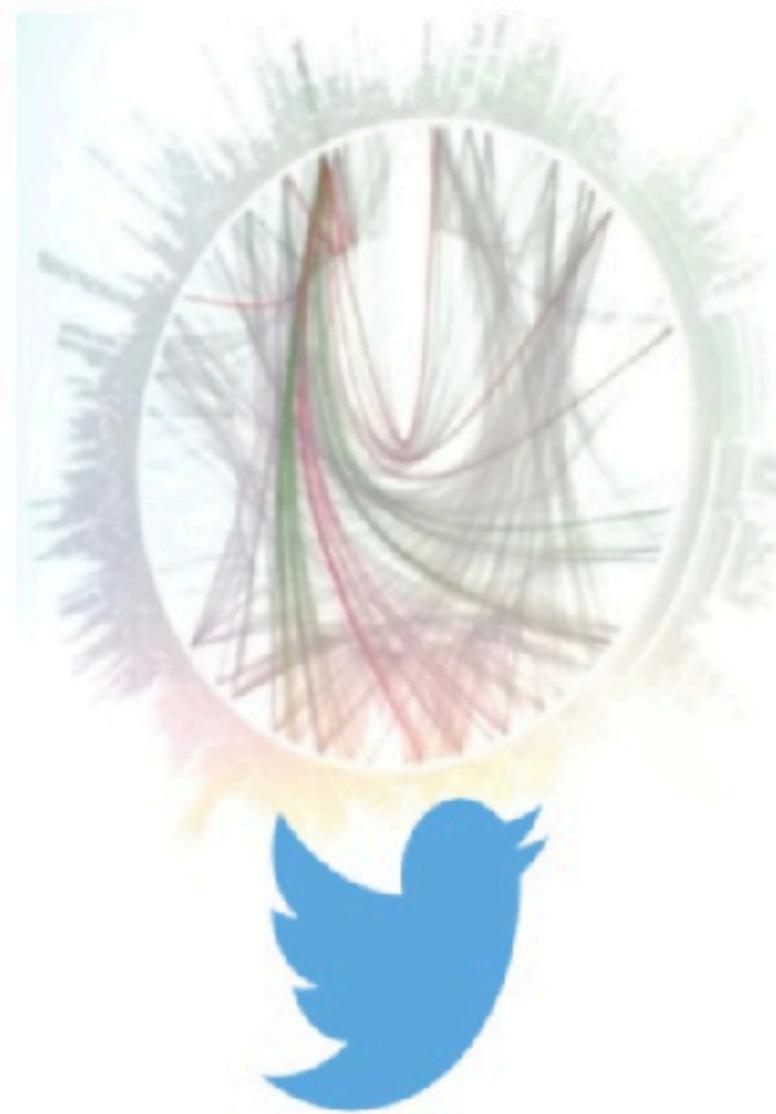
450 microservices



500+ microservices



500+ microservices



# 아마존의 선택(2002년경 제프 베조스 메일)

1. 모든 팀들은 데이터와 기능들을 서비스 인터페이스로 연결시켜라.
2. 팀들은 이 인터페이스를 통해서만 연락해야 한다.
3. 다른 어떤 커뮤니케이션 방법도 허용되지 않는다. 직접 링크를 보내거나 다른 팀의 스토리지에 직접 억세스 해서도 안 되며, 공유 메모리나 백도어 같은 것도 안 된다. 모든 커뮤니케이션은 네트워크를 통한 서비스 인터페이스로 이루어져야 한다.
4. 어떤 기술을 쓰든 상관없다. HTTP, Cobra, Pubsub, 독자 프로토콜...그건 상관없다. 베조스는 그런데 관심 없다.
5. 모든 서비스 인터페이스는 예외 없이 외부에서 이용 가능하게 만들어져야 한다. 그 말은 팀들은 외부 개발자들이 인터페이스를 이용할 수 있게 해야한다는 것이다. 예외는 없다.
6. 이를 실천하지 않는 사람은 누구든 해고될 것이다.

MSA 의 가장 본질적인 부분(해고 빼고;)

- 2006년 아마존 웹 서비스(AWS) 릴리즈
- 내부적으로 사용한 것과 똑같은 플랫폼

# 넷플릭스의 선택

- 2008년 데이터베이스에 심각한 문제가 발생해 고객에게 DVD를 보낼 수 없는 사태 발생(Single points of failure)
  - Scale-up 확장만 가능한 인프라 스트럭처와 단일 장애 지점(SPOF)이라는 한계에서 벗어나길 선언
  - 그 시작은 아파치 카산드라
- 2009년 아마존 웹 서비스(AWS)로 이관 시작
  - 세 가지 목표에 집중. 확장성(Scalability), 성능(Performance), 가용성(Availability)
  - '자체 보유 인프라와 솔루션으로는 이렇게 급증한 트래픽을 감당할 수 있는 확장성을 확보할 수 없었을 것'  
- 유리 이즈라일레프스키(Yuri Izrailevsky), Netflix 클라우드 및 플랫폼 엔지니어링 부문 부사장



# MSA란

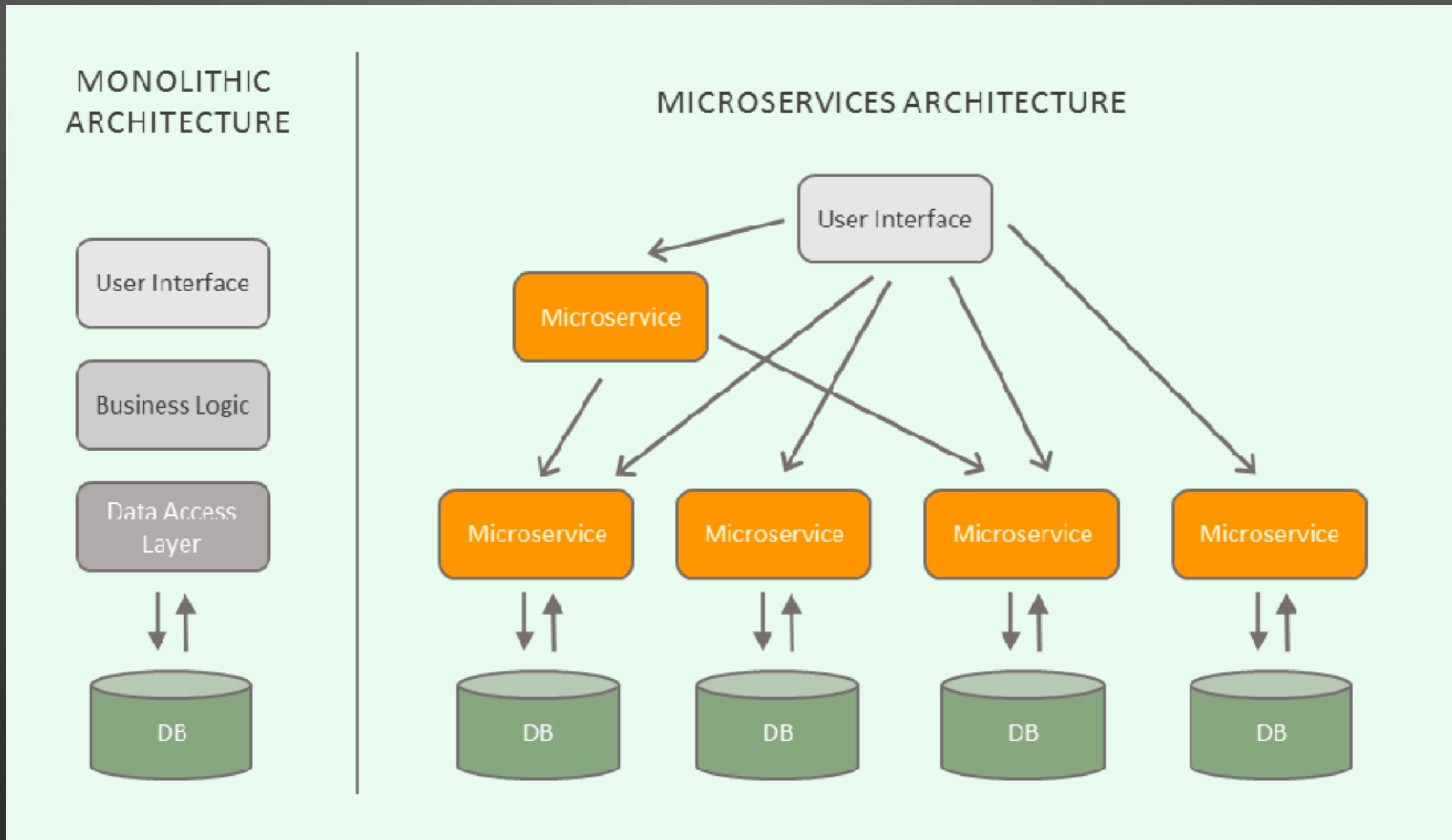
- 공식적인 정의는 없다. 단지, 다음 공감대가 있을 뿐(Microservices in 위키피디아)
  - 각 서비스 간 Network를 통해, 보통 HTTP를 통해
  - 독립된 배포 단위
  - 각 서비스는 쉽게 교체 가능
  - 각 서비스는 기능 중심으로 구성됨. e.g. 프론트 엔드, 추천, 정산, 상품, 등
  - 각 서비스에 적합한 프로그래밍 언어, 데이터베이스, 환경으로 만들어진다
  - 서비스는 크기가 작고, 상황에 따라 경계를 정하고, 자율적으로 개발되고, 독립적으로 배포되고, 분산되고, 자동화 된 프로세스로 구축되고 배포된다
- 마이크로서비스는 한 팀에 의해 개발할 수 있는 크기가 상한선이다. 절대로 3~9명의 사람들이 스스로 더 많은 개발을 할 수 없을 정도로 커지면 안된다
- '마이크로서비스는 아직까지 아이디어 수준에서 크게 벗어나 있지 않다. 다양한 산업 분야에서 폭넓게 적용되고 있지만 그것이 좋은지 나쁜지는 시간이 더 지나봐야 알 수 있다' - 조쉬 룽, 케니 바스타니, 피보탈

출처 : (책) 마이크로서비스 : 유연하고 확장 가능한 소프트웨어 아키텍처

출처 : <https://en.wikipedia.org/wiki/Microservices>

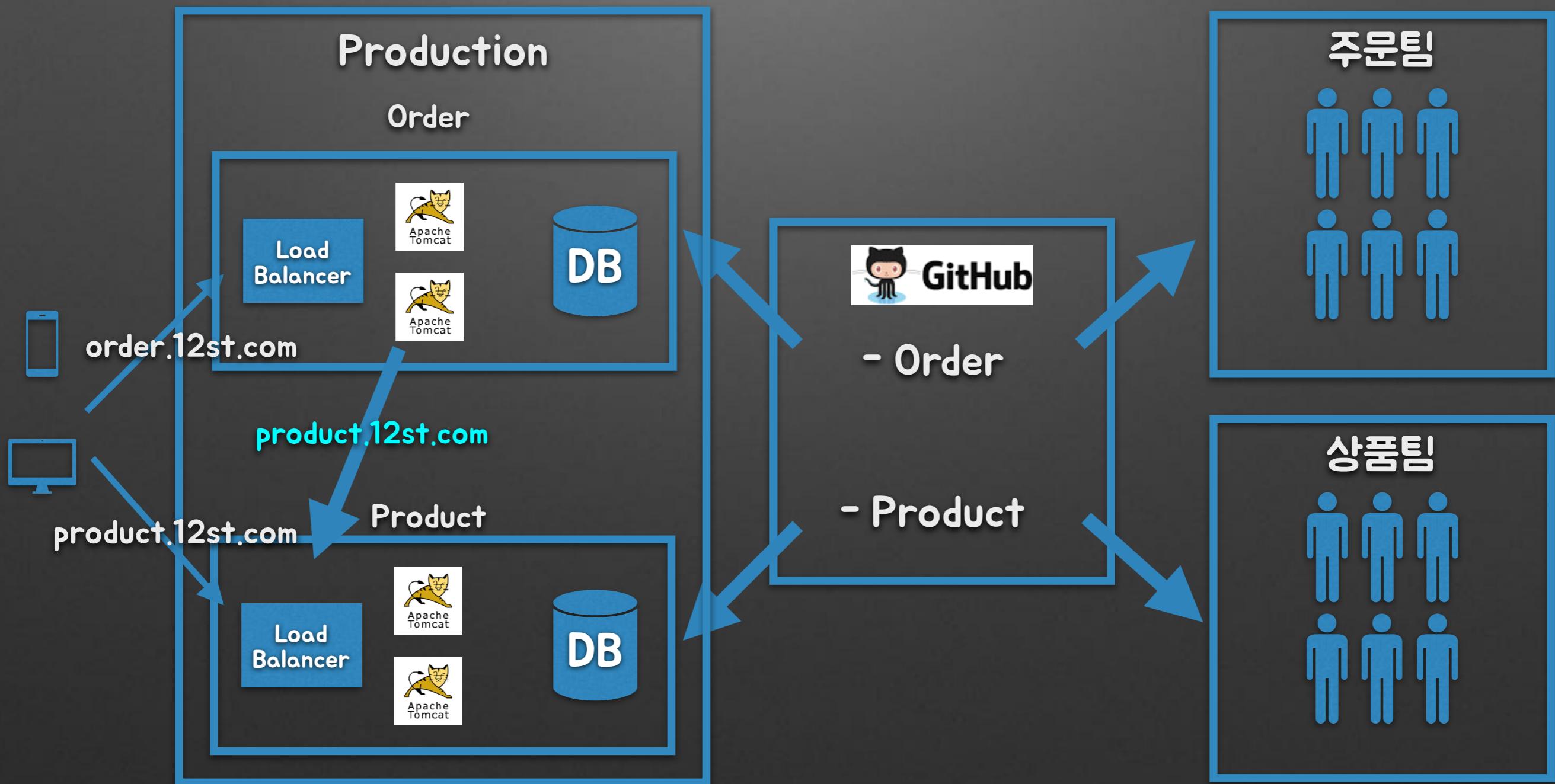
출처 : (책) 클라우드 네이티브 자바

# Monolithic VS Microservices



각 Microservice 별로 DB 를 가짐(공유되는 상태가 없음)

REST API 인터페이스 외엔 호출할 방법이 없음



Tag : step-1

## [실습] 소스 살펴 보기

### 1. 전체 구조

request : /products/{productId}

```
response : "[product id = " + productId +  
" at " + System.currentTimeMillis() + "]";
```



request : /displays/{displayId}

response : "[display id = %s at %s %s ]"

,displayId

,System.currentTimeMillis()

,/products/12345의 응답

# [실습] Rest API로 Display → Product 연동

## 1. Display → Product 호출

```
@RestController
@RequestMapping(path = "/displays")
public class DisplayController {

    @Autowired
    ProductRemoteService productRemoteService;

    @RequestMapping(path = "/{displayId}", method = RequestMethod.GET)
    public String getDisplayDetail(@PathVariable String displayId) {

        //String productInfo = "[unknown]";
        String productInfo = productRemoteService.getProductInfo( productId: "12345" );

        return String.format("[display id = %s at %s ms ]", displayId, System.currentTimeMillis(), productInfo);
    }
}
```

## 2. Display / Product 실행

- Product 동작 확인 : <http://localhost:8082/products/22222>
- Display 동작 확인 : <http://localhost:8081/displays/11111>

- 시작하기에 앞서
- 모놀리틱 아키텍쳐
- Microservices Architecture(MSA)
- Cloud Native <- 여기 할 차례예요
- Netflix OSS, Spring Cloud
  - Hystrix - Circuit Breaker
  - Eureka - Service Discovery
  - Zuul - API Gateway
- Configuration Server, Tracing, Monitoring, 그리고 남은 이야기..

# Cloud Native 란

- '클라우드 네이티브'의 핵심은 애플리케이션을 어떻게 만들고 배포하는지에 있으며 위치는 중요하지 않다.
- 클라우드 서비스를 활용한다는 것은 컨테이너와 같이 **민첩하고 확장 가능한** 구성 요소를 사용해서 재사용 가능한 개별적인 기능을 제공하는 것을 의미한다. 이러한 기능은 멀티 클라우드와 같은 여러 기술 경계 간에 매끄럽게 통합되므로 제공 팀이 반복 가능한 자동화와 오케스트레이션을 사용해서 빠르게 작업 과정을 반복할 수 있다 - 앤디 맨, Chief Technology Advocate at Splunk

신축성(Resiliency)

민첩성(Agility)

확장 가능성(Scalable)

자동화(Automation)

무상태(State-less)

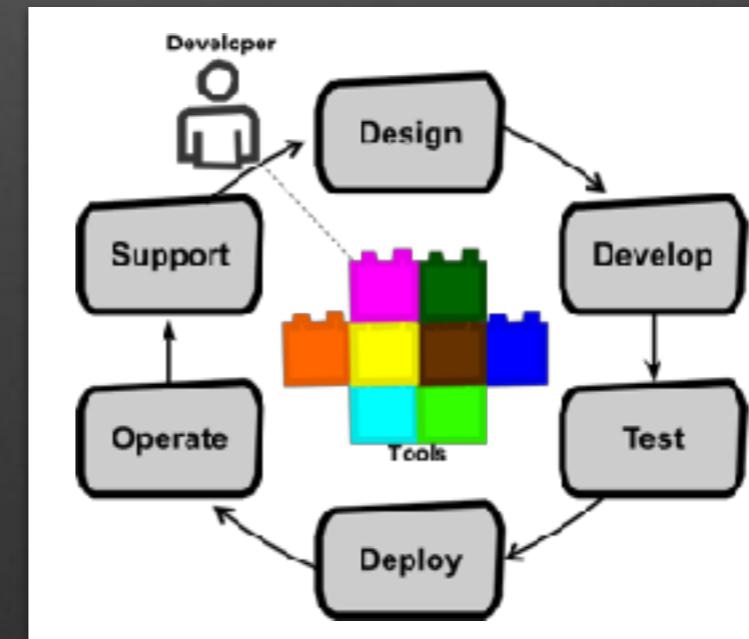


출처 :<https://pivotal.io/cloud-native>

출처 :<http://www.itworld.co.kr/news/109679>

# DevOps

- 전통적 모델
  - 개발과 운영 조직의 분리
  - 다른 쪽으로 일을 던진 후에 알아서 처리하라며 잊어버리는 방식
- DevOps, 더 나아가 Full Cycle Developers 로
  - You run it, you build it. 만들면 운영까지 - 베르너 보겔스, 아마존 CTO
  - 개별 팀은 프로젝트 그룹이 아닌 제품(Product) 그룹에 소속
  - 운영과 제품 관리 모두가 포함되는 조직적 구조, 제품 팀은 소프트웨어를 만들고 운영하는 데 필요한 모든 것을 보유



# Twelve-Factors

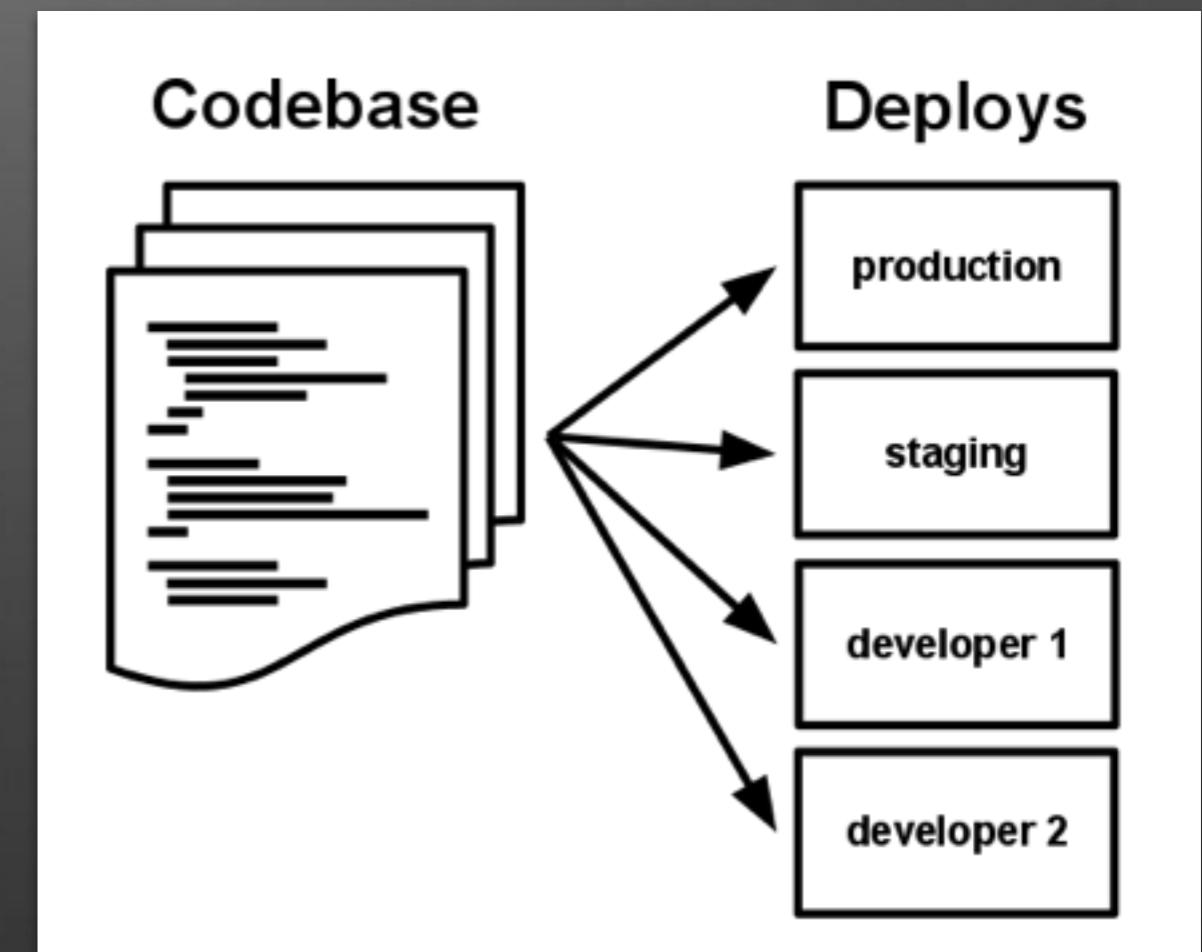
- 12 Factors
  - Heroku 클라우드 플랫폼 창시자들이 정립한 애플리케이션 개발 원칙 중 유익한 것을 모아서 정리한 것
  - 탄력적(elastic)이고 이식성 있는(portability) 배포를 위한 베스트 프랙티스(Best Practices)
- 핵심 사상
  - 선언적 형식으로 설정을 자동화해서 프로젝트에 새로 참여하는 동료가 적응하는 데 필요한 시간과 비용을 최소화한다.
  - 운영체제에 구애받지 않는 투명한 계약을 통해 다양한 실행 환경에서 작동할 수 있도록 이식성을 극대화 한다.
  - 현대적인 클라우드 플랫폼 기반 개발을 통해 서버와 시스템 관리에 대한 부담을 줄인다
  - 개발과 운영의 간극을 최소화해서 지속적 배포(continuous deployment)를 가능하게 하고 애자일성을 최대화한다.
  - 도구, 아키텍처, 개발 관행을 크게 바꾸지 않아도 서비스 규모 수직적 확장이 가능하다

# The Twelve Factors - 12가지 제약 조건

1	Codebase	코드베이스	단일 코드베이스. 버전 관리되는 하나의 코드베이스와 다양한 배포. 개발/테스트/운영서버(인스턴스)는 동일한 코드 기반으로 있어야 함
2	Dependencies	의존성	명시적으로 선언되고 분리된 의존성. 필요한 의존성을 애플리케이션과 함께 담음
3	Config	설정	환경설정은 분리하여 외부에 보관. 소스코드(코드베이스)는 하나, 환경(개발/테스트/운영)에 따라 설정만 바꿔야 함
4	Backing Services	백엔드 서비스	백엔드 서비스를 연결된 리소스로 취급. URL을 통해 접근(바인딩)되어야 함
5	Build, release, run	빌드, 릴리즈, 실행	분리된 빌드와 실행 단계를 가져야 함
6	Stateless process	무상태 프로세스	애플리케이션을 하나 혹은 여러개의 무상태 프로세스로 실행. 상태는 외부저장소에 보관
7	Port binding	포트 바인딩	포트 바인딩을 사용해서 서비스 노출. 별도의 웹서버를 두지 않고 자기완결적으로 서비스 제공
8	Concurrency	동시성	프로세스 모델을 사용한 확장(scale out). 프로세스가 복제를 통해 확장될 수 있게 설계해야 함
9	Disposability	폐기 가능	빠른 시작과 그레이스풀 셧다운(graceful shutdown)을 통한 안정성 극대화
10	Dev/prod parity	dev/prod 일치	development, staging, production 환경을 최대한 동일하게 유지
11	Logs	로그	로그를 이벤트 스트림으로 취급. 로컬서버에 저장하지 말고 중앙저장소로 수집
12	Admin processes	Admin 프로세스	admin/maintenance 작업을 일회성 프로세스로 실행

# Twelve-Factors - 코드 베이스

- 버전 관리되는 하나의 코드베이스가 여러 번 배포된다
- 코드베이스와 앱 사이에는 항상 1대1 관계가 성립된다



# Twelve-Factors - 의존성

- 애플리케이션의 의존관계(dependencies)는 명시적으로 선언되어야 한다.
- 모든 의존 라이브러리는 아파치 메이븐, 그레이디틀 등의 의존관계 관리 도구를 써서 라이브러리 저장소에서 내려받을 수 있어야 한다

```
dependencies {  
    implementation('org.springframework.boot:spring-boot-starter-web')  
    implementation('org.springframework.boot:spring-boot-starter-actuator')  
    implementation('org.springframework.cloud:spring-cloud-starter-netflix-eureka-client')  
    implementation('io.micrometer:micrometer-registry-prometheus')  
    compileOnly('org.projectlombok:lombok')  
    testImplementation('org.springframework.boot:spring-boot-starter-test')  
}
```

# Twelve-Factors - 설정

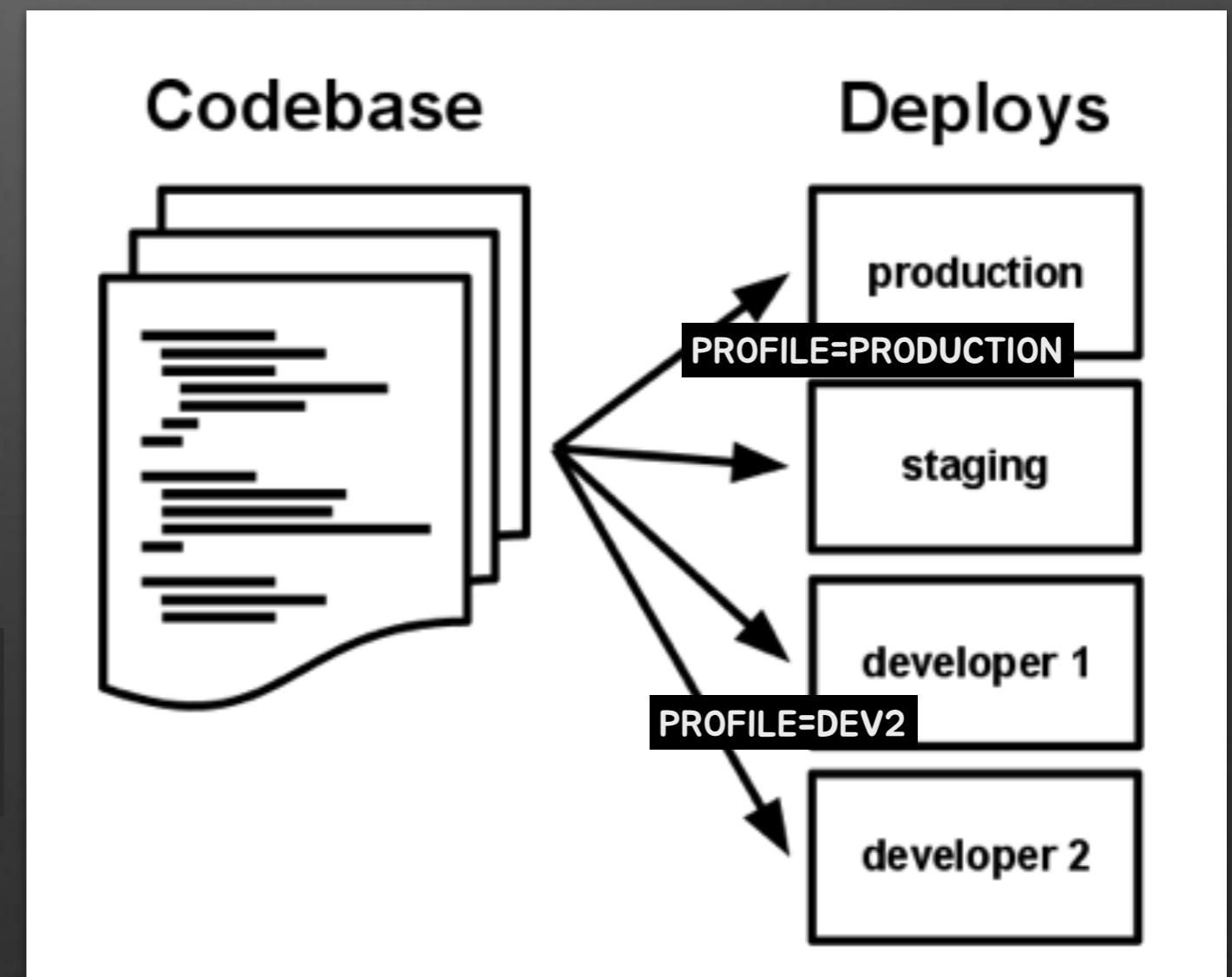
- 설정 정보는 실행 환경에 저장한다
- 설정 정보(configuration)는 애플리케이션 코드와 엄격하게 분리
- 설정은 배포(스테이징, 프로덕션, 개발 환경 등)마다 달라질 수 있는 모든 것(DB정보, 외부 서비스 인증, 호스트 이름 등)
- 설정을 환경 변수(env)에 저장한다

application-local.yml

```
datasource.main:  
  url: jdbc:h2:mem:main;DB_CLOSE_ON_EXIT=FALSE  
  driver-class-name: org.h2.Driver  
  username: sa  
  password:
```

application-dev.yml

```
datasource.main:  
  url: jdbc:log4jdbc:oracle:thin:@XXXXX:TMALL  
  driver-class-name: net.sf.log4jdbc.sql.jdbcapi.DriverSpy  
  username: user  
  password: password
```



# Twelve-Factors - 백엔드(지원) 서비스

- 지원 서비스(backing service)는 필요에 따라 추가되는 자원으로 취급한다
- 지원 서비스는 데이터베이스, API 기반 RESTful 웹 서비스, SMTP 서버, FTP 서버 등
- 지원 서비스는 애플리케이션의 자원으로 간주한다
- 테스트 환경에서 사용하던 임베디드 SQL 을 스테이징 환경에서 MySQL 로 교체할 수 있어야 한다

## application-local.yml

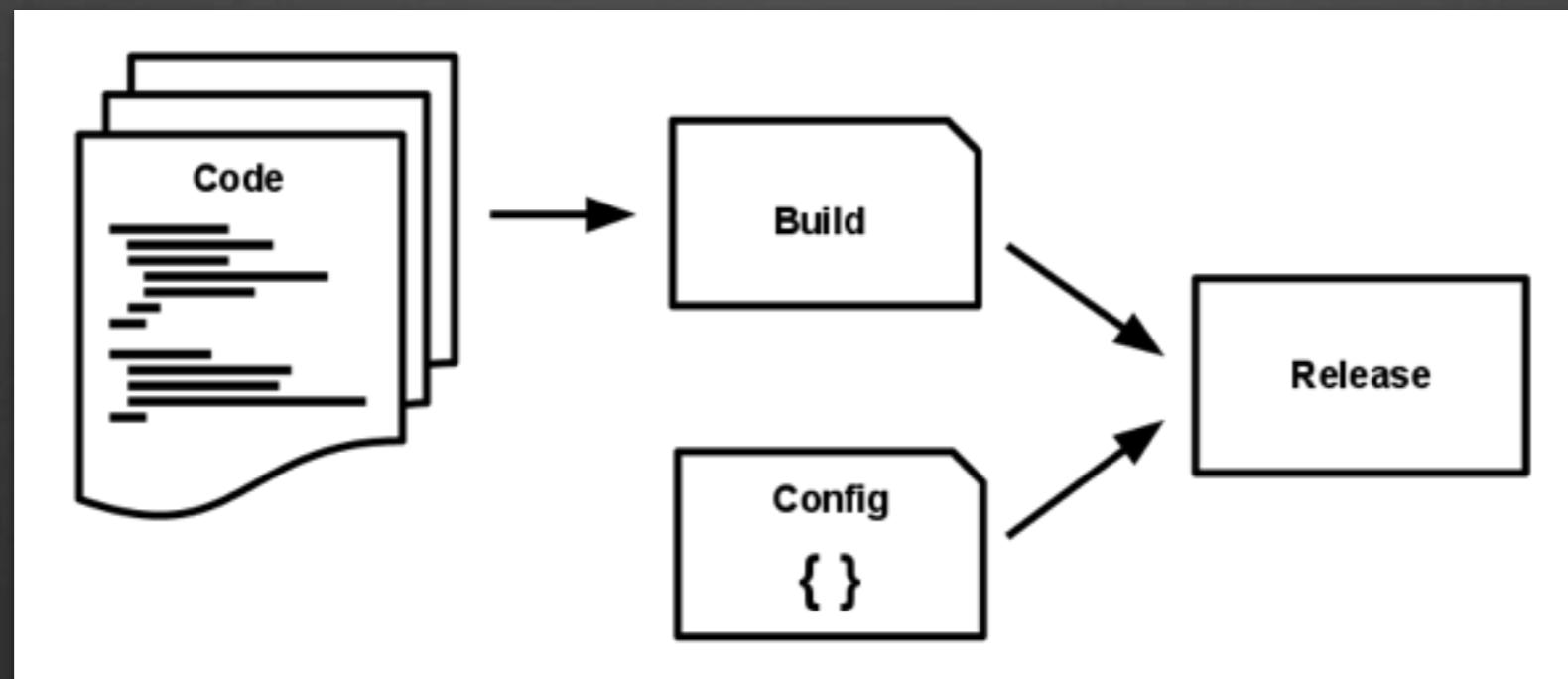
```
datasource.main:  
  url: jdbc:h2:mem:main;DB_CLOSE_ON_EXIT=FALSE  
  driver-class-name: org.h2.Driver  
  username: sa  
  password:
```

## application-dev.yml

```
datasource.main:  
  url: jdbc:log4jdbc:oracle:thin:@XXXXX:TMALL  
  driver-class-name: net.sf.log4jdbc.sql.jdbcapi.DriverSpy  
  username: user  
  password: password
```

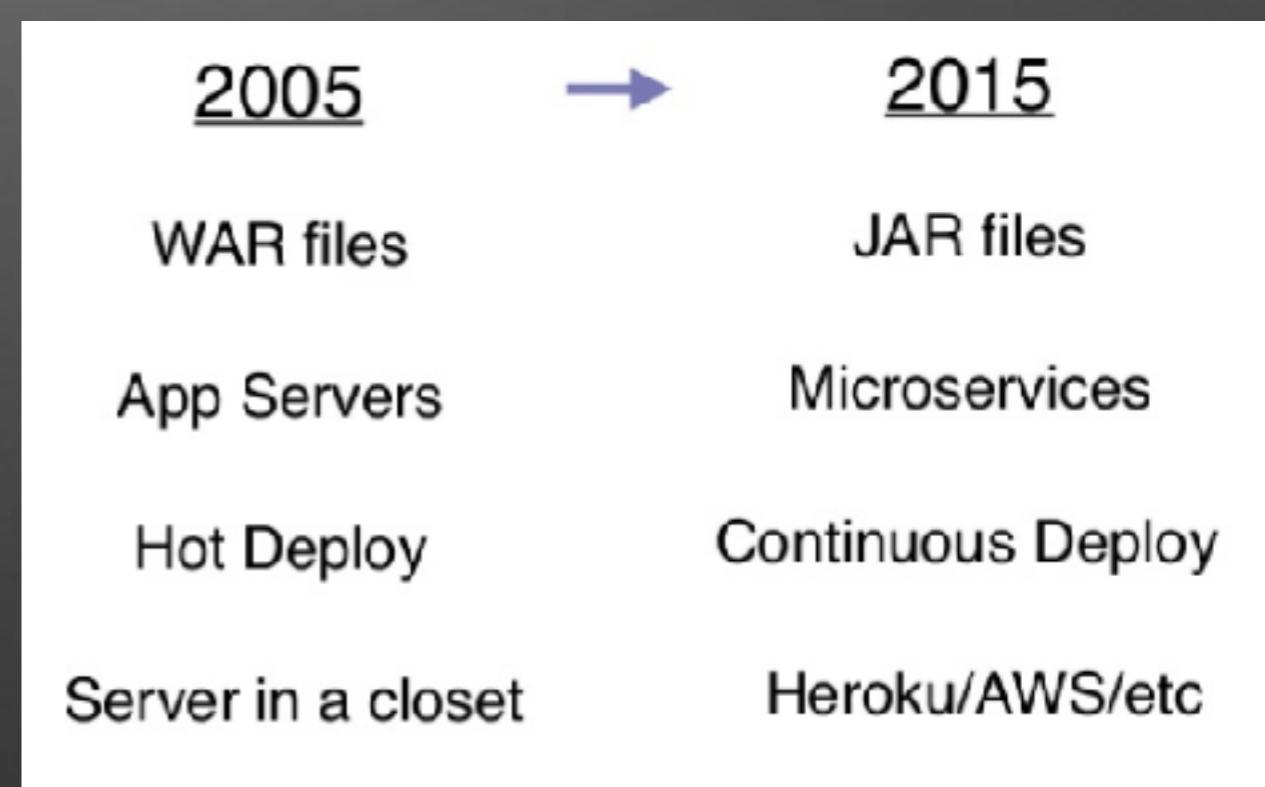
# Twelve-Factors - 빌드, 릴리즈, 실행

- 철저하게 분리된 빌드와 실행 단계
- 코드베이스는 3단계를 거쳐 (개발용이 아닌) 배포로 변환된다
  - **빌드 단계** : 소스 코드를 가져와 컴파일 후 하나의 패키지를 만든다
  - **릴리스 단계** : 빌드에 환경설정 정보를 조합한다. 릴리스 버전은 실행 환경에서 운영될 수 있는 준비가 완료되어 있다. 시맨틱 버저닝 등 식별자가 부여됨. 이 버전은 롤백하는 데 사용
  - **실행 단계** : 보통 런타임이라 불림. 릴리스 버전 중 하나를 선택해 실행 환경 위에 애플리케이션 실행



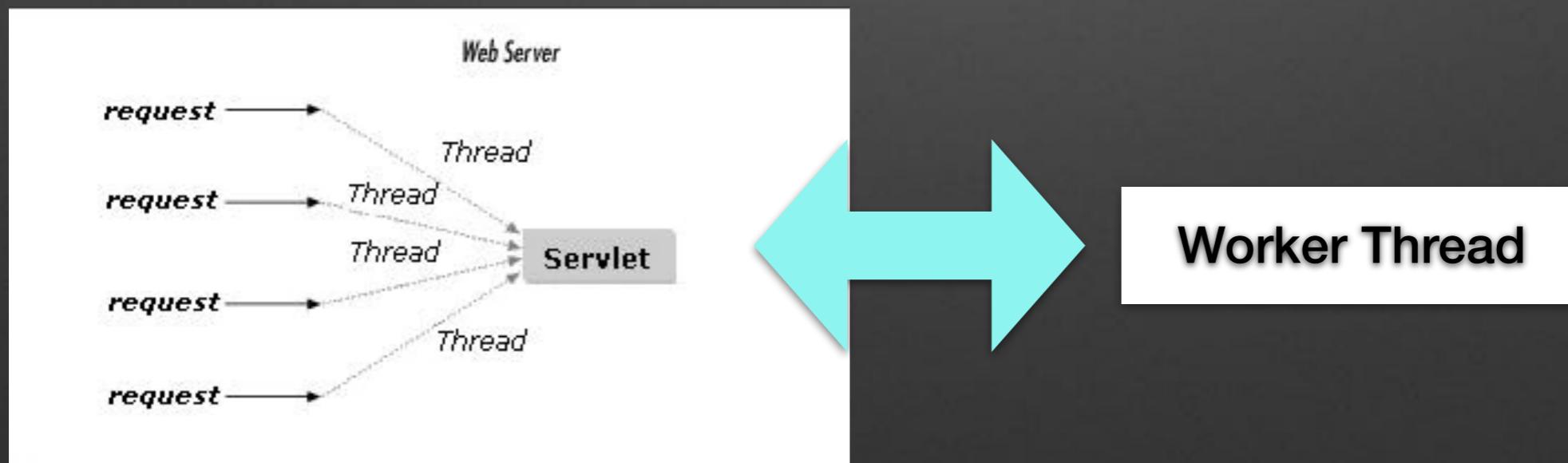
# Twelve-Factors - 포트 바인딩

- 서비스는 포트에 연결해서 외부에 공개한다
- 실행 환경에 웹 서버를 따로 추가해줄 필요 없이 스스로 웹 서버를 포함하고 있어서 완전히 자기 완비적 (self-contained) 이다.



# Twelve-Factors - 동시성(concurrency)

- 프로세스 모델을 통해 수평적으로 확장한다
- 애플리케이션은 필요할 때마다 프로세스나 스레드를 수평적으로 확장해서 병렬로 실행할 수 있어야 한다
- 장시간 소요되는 데이터 프로세싱 작업은 스레드풀에 할당해서 스레드 실행기(executor)를 통해 수행되어야 한다
- 예를 들어, HTTP 요청은 서블릿 쓰레드가 처리하고, 시간이 오래 걸리는 작업은 워커 쓰레드가 처리해야 한다



# Twelve-Factors - 처분성(Disposability)

- 빠른 시작과 그레이스풀 셧다운(graceful shutdown)을 통한 안정성 극대화
- 애플리케이션은 프로세스 실행 중에 언제든지 중지될 수 있고, 중지될 때 처리되어야 하는 작업을 모두 수행한 다음 깔끔하게 종료될 수 있다
- 가능한 한 짧은 시간 내에 시작되어야 한다

Resilience to failure



# Twelve-Factors - dev/prod 일치

- development, staging, production 환경을 최대한 비슷하게 유지
- 개발 환경과 운영 환경을 가능한 한 동일하게 유지하는 짹맞춤(parity)을 통해 분기(divergence)를 예방할 수 있어야 한다
- 유념해야 할 세 가지 차이
  - 시간 차이 : 개발자는 변경 사항을 운영 환경에 빨리 배포해야 한다
  - 개인 차이 : 코드 변경을 맡은 개발자는 운영 환경으로의 배포 작업까지 할 수 있어야 하고, 모니터링도 할 수 있어야 한다
  - 도구 차이 : 각 실행 환경에 사용된 기술이나 프레임워크는 동일하게 구성되어야 한다



# Twelve-Factors - 로그

- 로그는 이벤트 스트림으로 취급한다
- 로그를 `stdout`에 남긴다
- 애플리케이션은 로그 파일 저장에 관여하지 말아야 한다
- 로그 집계와 저장은 애플리케이션이 아니라 실행 환경에 의해 처리되어야 한다

# Twelve-Factors - Admin 프로세스

- admin/maintenance 작업을 일회성 프로세스로 실행
- 실행되는 프로세스와 동일한 환경에서 실행
- admin 코드는 애플리케이션 코드와 함께 배포되어야 한다

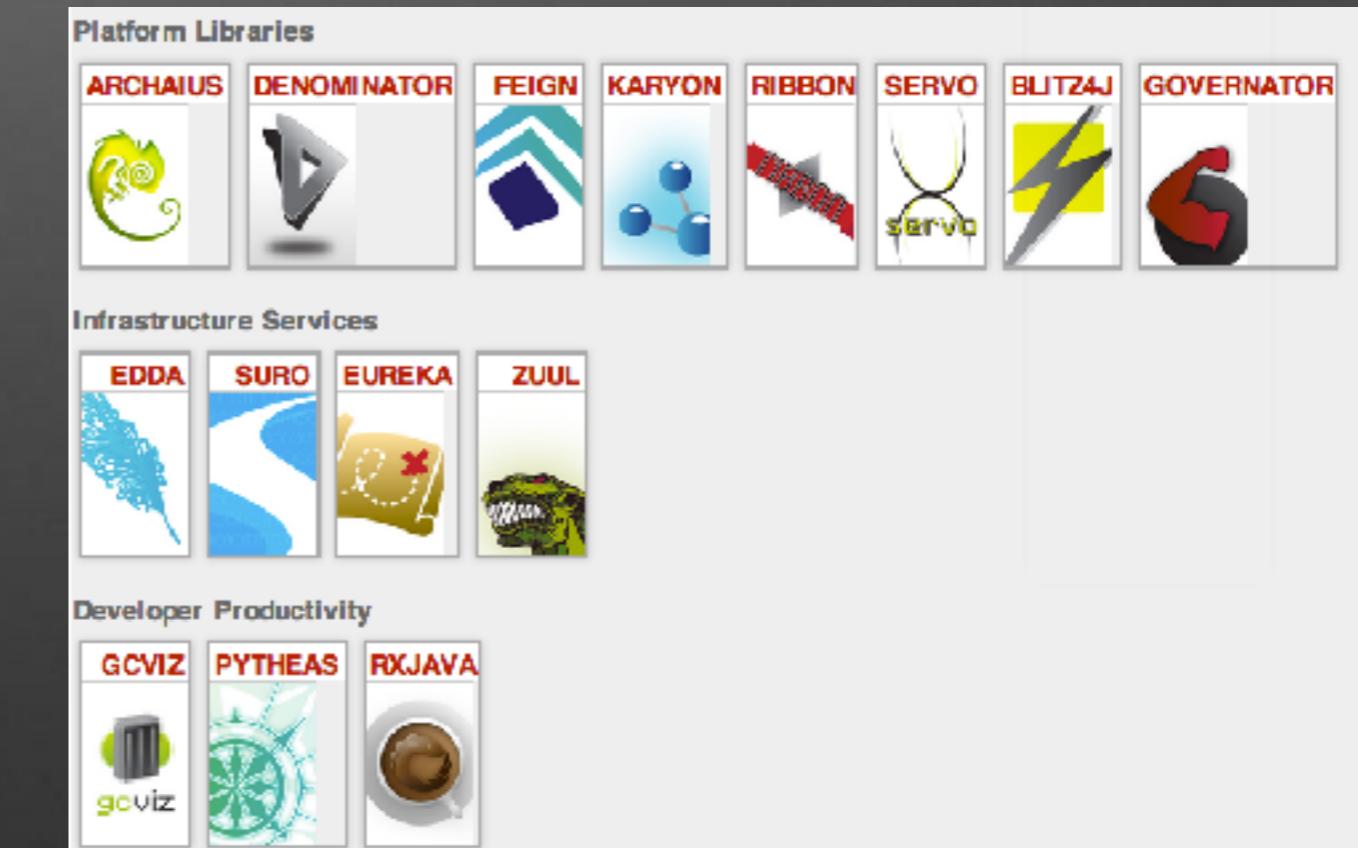
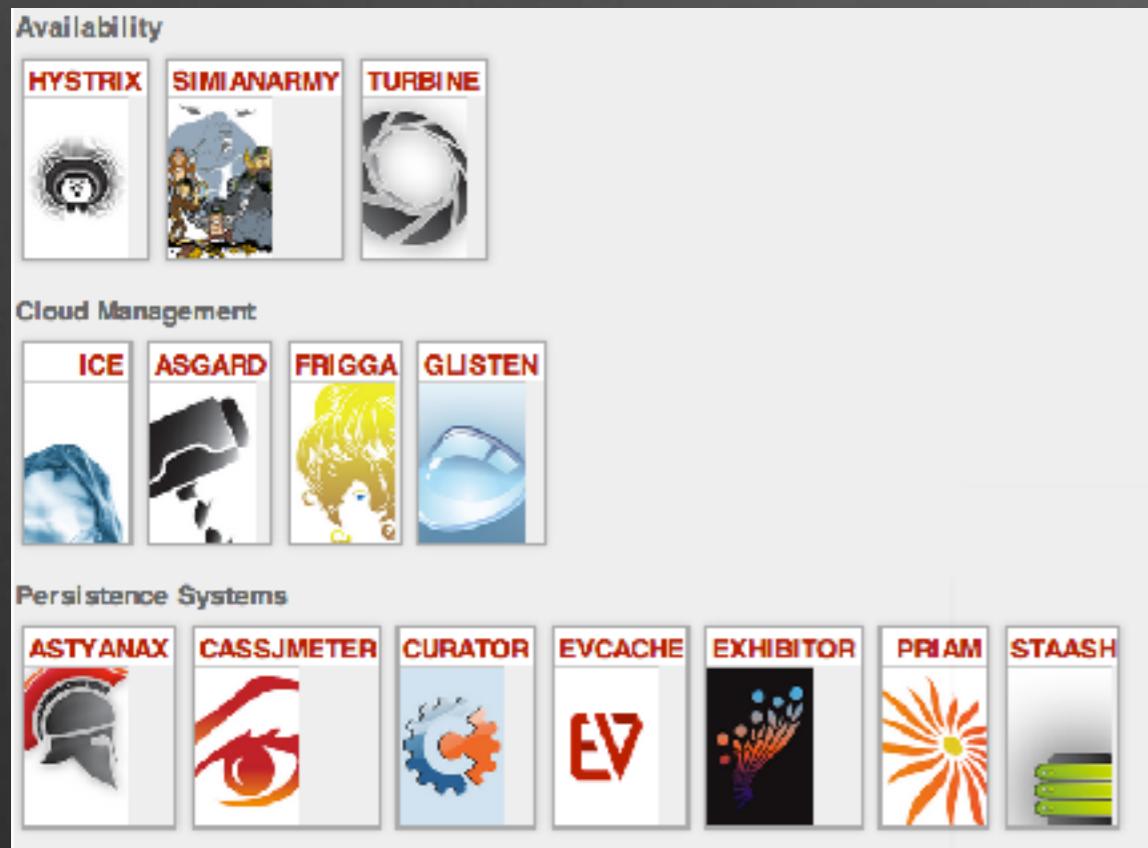
# HTTP, REST API

- HTTP
  - 클라이언트의 상태를 갖지 않음(stateless)
  - 각 요청은 자기 완비적(self-contained)
- REST vs 그 외(EJB, SOAP, etc..)
- REST API
  - 2000년 로이 필딩(Roy Fielding) 박사가 소개(HTTP 명세 writer)
  - 원격 자원(resource) 와 엔티티(Entity) 를 다루는 데 초점
  - 동사 대신 명사를, 행위 대신 엔티티에 집중
  - REST 는 기술 표준이 아닌 아키텍처 제약사항
  - 상태가 없고 요청이 자기 완비적이기 때문에 서비스도 수평적으로 쉽게 확장될 수 있다

- 시작하기에 앞서
- 모놀리틱 아키텍쳐
- Microservices Architecture(MSA)
- Cloud Native
- Netflix OSS, Spring Cloud <- 여기 할 차례예요
  - Hystrix - Circuit Breaker
  - Eureka - Service Discovery
  - Zuul - API Gateway
- Configuration Server, Tracing, Monitoring, 그리고 남은 이야기..

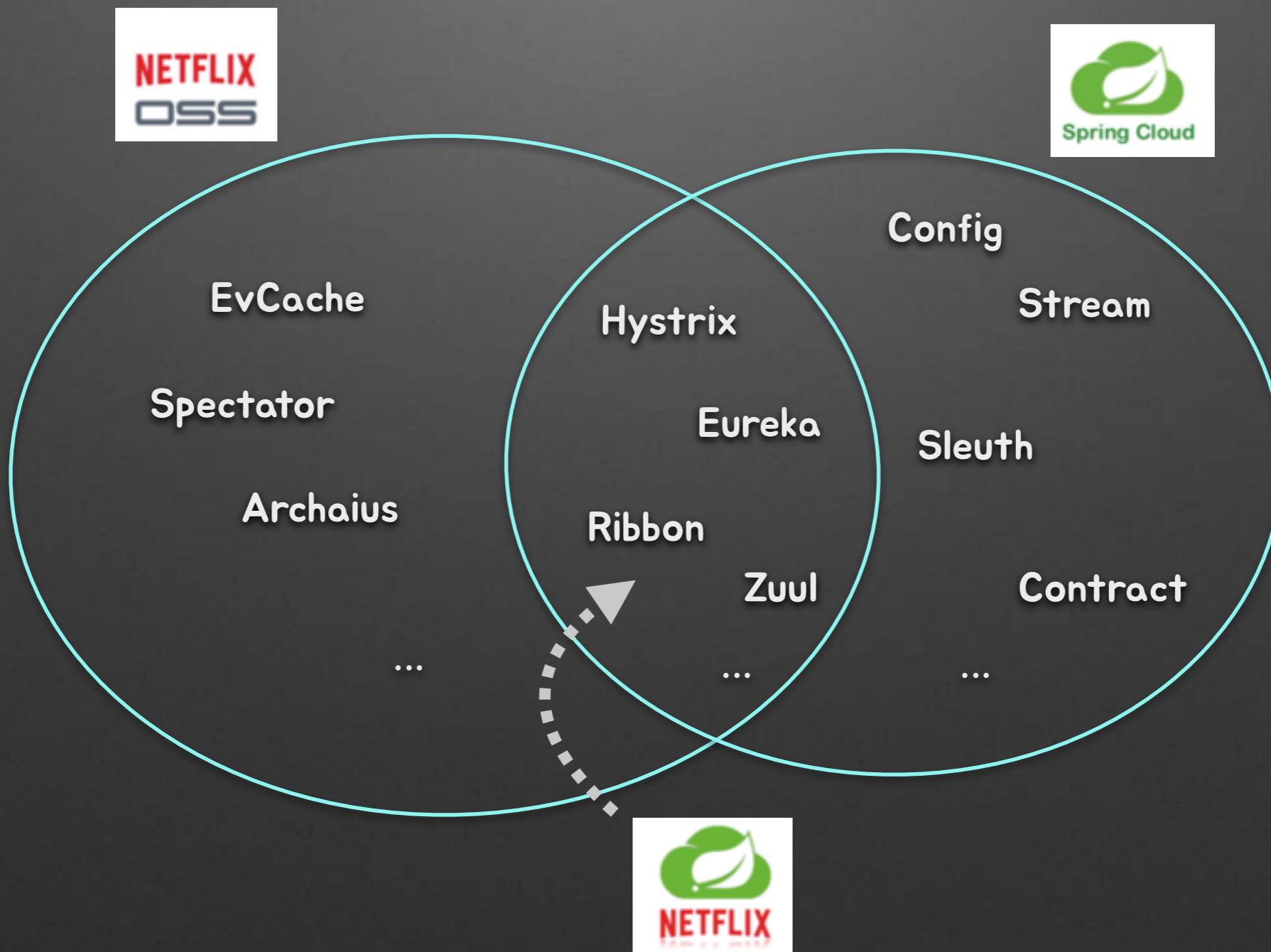
# Netflix OSS

- 50 개 이상의 사내 프로젝트를 오픈소스로 공개
- 플랫폼(AWS) 안의 여러 컴포넌트와 자동화 도구를 사용하면서 파악한 패턴과 해결 방법을 블로그, 오픈 소스로 공개



# Spring Cloud

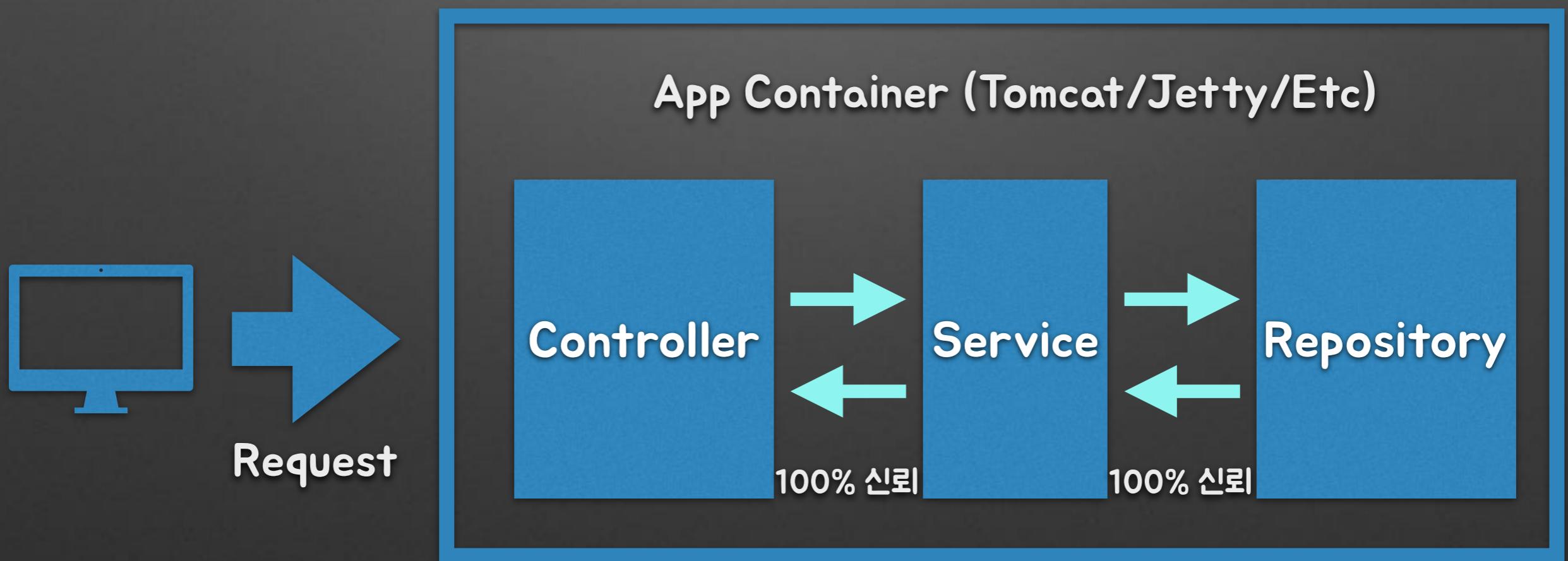
- Spring Cloud 란
- 교집합이 spring-cloud-netflix



- 시작하기에 앞서
- 모놀리틱 아키텍쳐
- Microservices Architecture(MSA)
- Cloud Native
- Netflix OSS, Spring Cloud
  - **Hystrix - Circuit Breaker <- 여기 할 차례예요**
  - Eureka - Service Discovery
  - Zuul - API Gateway
- Configuration Server, Tracing, Monitoring, 그리고 남은 이야기..

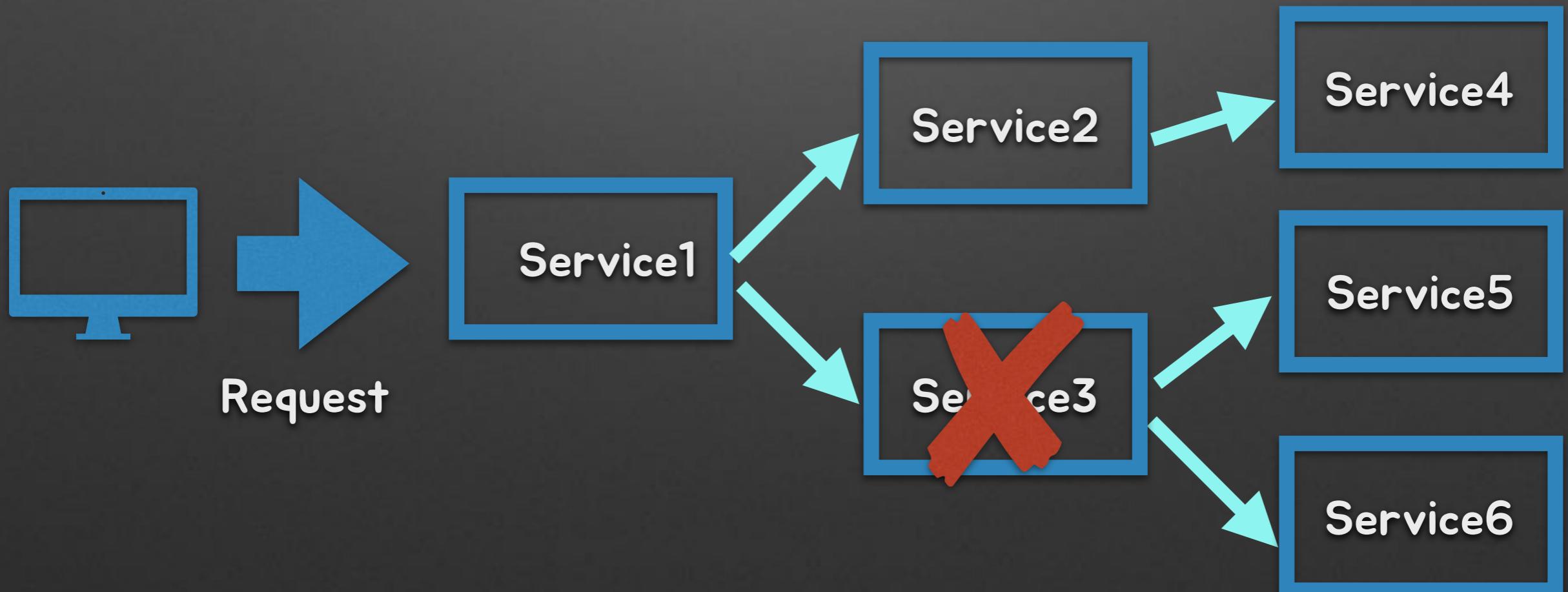
# 모놀리틱에서의 의존성 호출

- 모놀리틱에서의 의존성 호출은 100% 신뢰



# Failure as a First Class Citizen

- 분산 시스템, 특히 클라우드 환경에선 실패(Failure)는 일반적인 표준이다.
- 모놀리틱엔 없던(별로 신경 안썼던..) 장애 유형
- 한 서비스의 가동율(uptime) 최대 99.99%
  - $99.99^{30} = 99.7\%$  uptime
  - 10 억 요청 중 0.3% 실패 = 300만 요청이 실패
- 모든 서비스들이 이상적인 uptime 을 갖고 있어도 매 달마다 2시간 이상의 downtime 이 발생



# Circuit Breaker - 회로 차단기

과부하가 걸리거나 단락으로 인한 피해를 막기 위해 자동으로 회로를 정지시키는 장치

퓨즈와 다른 점은, 어느 정도 시간이 지난 뒤, 원래의 기능이 동작하도록  
복귀된다

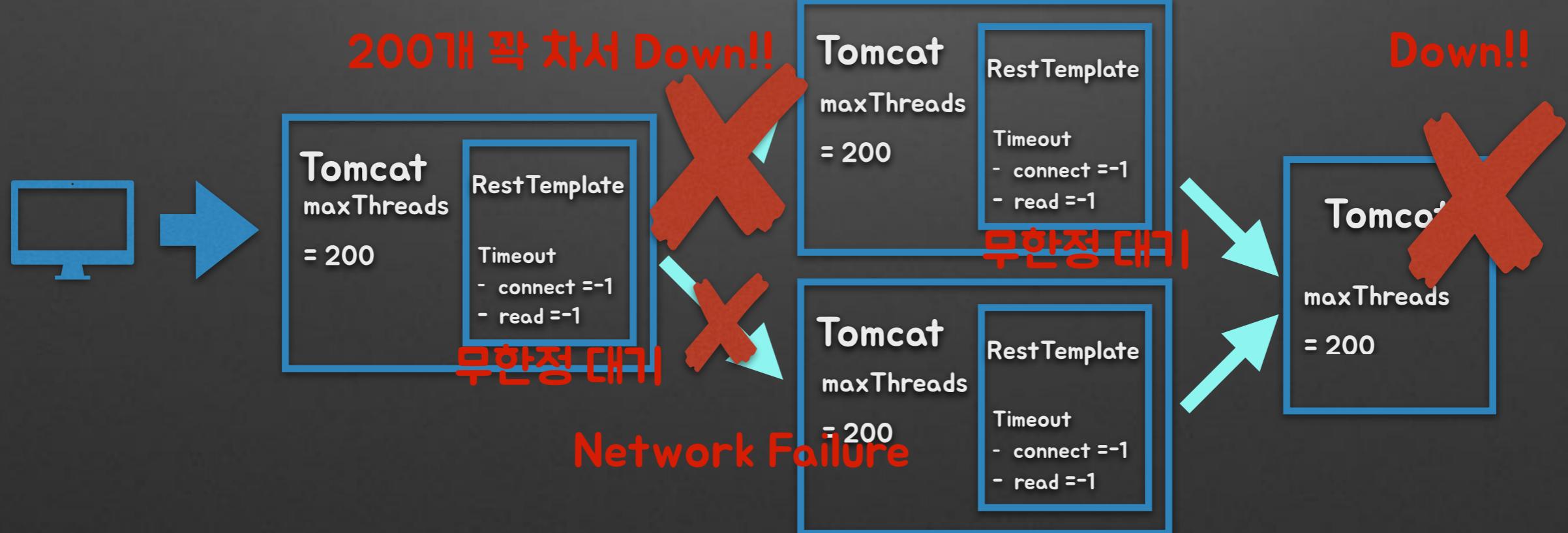


이미지 출처 : <http://11st.kr/QR/P/2024768362>

# Circuit Breaker - Hystrix

- Latency Tolerance and Fault Tolerance for Distributed Systems

모든 설정이 기본이라고 가정해보겠습니다



# Hystrix 적용하기

## @HystrixCommand

```
public String anyMethodWithExternalDependency() {  
    URI uri = URI.create("http://172.32.1.22:8090/recommended");  
    String result = this restTemplate.getForObject(uri, String.class);  
    return result;  
}
```

위의 메소드를 호출할 때 벌어지는 일

- 이 메소드 호출을 'Intercept' 하여 '대신' 실행
- 실행된 결과의 성공 / 실패 (Exception) 여부를 기록하고 '통계'를 낸다. - 통계 ??
- 실행 결과 '통계'에 따라 Circuit Open 여부를 판단하고 필요한 '조치'를 취한다.

# Hystrix - Circuit Breaker

- Circuit Open ??
  - Circuit이 오픈된 Method는 (주어진 시간동안) 호출이 '제한'되며, '즉시' 에러를 반환한다.
- Why ?
  - 특정 메소드에서의 지연 (주로 외부 연동에서의 지연)이 시스템 전체의 Resource 를 (Thread, Memory등)를 모두 소모하여 시스템 전체의 장애를 유발한다.
  - 특정 외부 시스템에서 계속 에러를 발생 시킨다면, 지속적인 호출이 에러 상황을 더욱 악화 시킨다.
- So !
  - 장애를 유발하는 (외부) 시스템에 대한 연동을 조기에 차단 (Fail Fast) 시킴으로서 나의 시스템을 보호한다.
- 기본 설정
  - 10초동안 20개 이상의 호출이 발생 했을때 50% 이상의 호출에서 에러가 발생하면 Circuit Open

# Hystrix - Circuit Breaker

- Circuit이 오픈된 경우의 에러 처리는 ? - Fallback
  - Fallback method는 Circuit이 오픈된 경우 혹은 Exception이 발생한 경우 대신 호출될 Method. 장애 발생시 Exception 대신 응답할 Default 구현을 넣는다.

```
@HystrixCommand(commandKey = 'ExtDep1', fallbackMethod='recommendFallback')

public String anyMethodWithExternalDependency1() {
    URI uri = URI.create("http://172.32.1.22:8090/recommended");
    String result = this restTemplate.getForObject(uri, String.class);
    return result;
}

public String recommendFallback() {
    return "No recommend available";
}
```

# Hystrix - Circuit Breaker

- 오랫동안 응답이 없는 메소드에 대한 처리 방법은 ? - Timeout

```
@HystrixCommand(commandKey = 'ExtDep1', fallbackMethod='recommendFallback',
    commandProperties = {
        @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "500")
    })
```

```
public String anyMethodWithExternalDependency1() {
    URI uri = URI.create("http://172.32.1.22:8090/recommended");

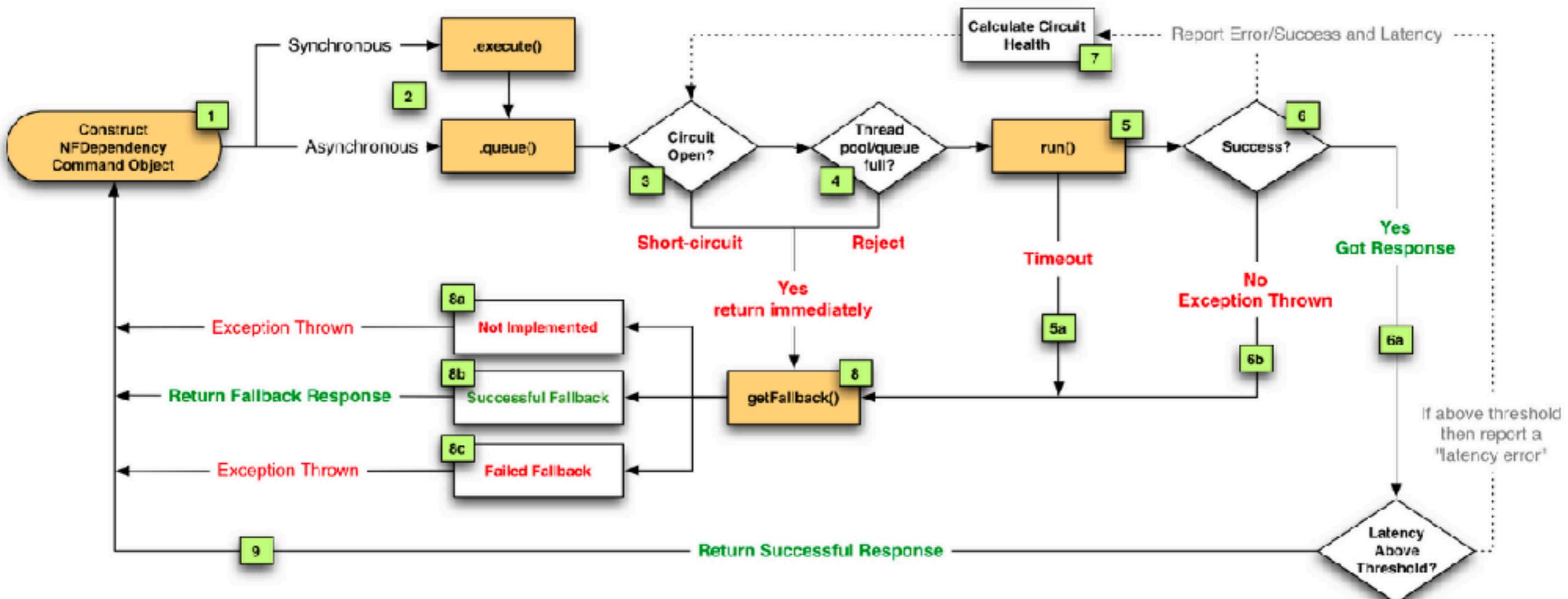
    String result = this restTemplate.getForObject(uri, String.class);

    return result;
}

public String recommendFallback() {
    return "No recommend available";
}
```

- 설정하지 않으면 default 1,000ms
- 설정 시간동안 메소드가 끝나지 않으면 (return / exception) Hystrix는 메소드를 실제 실행중인 Thread에 interrupt()를 호출하고, 자신은 즉시 HystrixException 을 발생시킨다.
- 물론 이경우에도 Fallback이 있다면 Fallback을 수행

# Hystrix - Circuit Breaker



Tag : step-2-hystrix-basic

## [실습 Step-2] Hystrix 사용하기

- 배경
  - Display 서비스는 외부 Server인 Product API와 연동되어있음
  - Product API에 장애가 나더라도 Display의 다른 서비스는 이상없이 동작하였으면 합니다
  - Product API에 응답 오류인경우, Default값을 넣어 주고 싶습니다
- 결정 내용
  - Display -> Product 연동 구간에 Circuit Breaker를 적용 !

Tag : step-2-hystrix-basic

## [실습 Step-2] Hystrix 사용하기

1. [display] build.gradle에 hystrix dependency추가

```
compile('org.springframework.cloud:spring-cloud-starter-netflix-hystrix')
```

2. [display] DisplayApplication에 @EnableCircuitBreaker추가

```
@EnableCircuitBreaker
```

3. [display] ProductRemoteServiceImp에 @HystrixCommand 추가

```
@HystrixCommand
```

Tag : step-2-hystrix-fallback

## [실습 Step-2] Hystrix Fallback 적용하기

4. [product] ProductController에서 항상 Exception 던지게 수정하기 (장애 상황 흉내)

```
throw new RuntimeException('I/O Error')
```

5. [display] ProductRemoteServiceImp에 Fallback Method 작성하기

```
@Override  
@HystrixCommand(fallbackMethod = "getProductInfoFallback")  
public String getProductInfo(String productId) {  
    return this.restTemplate.getForObject(url: url + productId, String.class);  
}  
  
public String getProductInfoFallback(String productId) {  
    return "[This Product is sold out]";  
}
```

Tag : step-2-hystrix-fallback

## [실습 Step-2] Hystrix Fallback 적용하기

### 6. 확인

<http://localhost:8082/products/22222>

<http://localhost:8081/displays/11111>

Hystrix가 발생한 Exception을 잡아서 Fallback을 실행

Fallback 정의 여부와 상관없이 Circuit 오픈 여부 판단을 위한 '에러통계'는 계산하고 있음  
아직, Circuit이 오픈된 상태 X

Tag : step-2-hystrix-fallback2

## [실습 Step-2] Hystrix Fallback 적용하기

### 7. Fallback 원인 출력하기

Fallback 메소드의 마지막 파라매터를 'Throwable'로 추가하면 Fallback 일으킨 Exception을 전달 해줌

```
public String getProductInfoFallback(String productId, Throwable t) {  
    System.out.println("t = " + t);  
    return "[This Product is sold out]";  
}
```

Tag : step-2-hystrix-fallback2

## [실습 Step-2] Hystrix Fallback 적용하기

### 8. 정리

Hystrix에서 Fallback의 실행 여부는 Exception이 발생 했는가 여부

Fallback의 정의 여부는 Circuit Breaker Open과 무관.

Throwable 파라매터의 추가로, Fallback 원인을 알 수 있다.

Tag : step-2-hystrix-timeout

## [실습 Step-2] Hystrix로 Timeout 처리하기

Hystrix로 할 수 있는 또 한가지! **Timeout** 처리

@HystrixCommand로 표시된 메소드는 지정된 시간안에 반환되지 않으면  
자동으로 **Exception** 발생 (기존 설정 : 1,000ms)

1. [product] ProductController의 throw Exception을 Thread.sleep(2000)로 수정

```
@RequestMapping(path = "{productId}", method = RequestMethod.GET)
public String getProductInfo(@PathVariable String productId) {
    try {
        Thread.sleep( millis: 2000 );
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    //throw new RuntimeException("I/O Error");
    return "[product id = " + productId + " at " + System.currentTimeMillis() + "]";
}
```

2. 확인

Product API를 직접 호출하는 경우는 동작하나, Display는 동작 안함

t = com.netflix.hystrix.exception.HystrixTimeoutException

h

## 3. [display] application.yml 수정하여 Hystrix Timeout 시간 조정하기

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=3000

```
hystrix:  
  command:  
    default:  # command key. use 'default' for global setting.  
      execution:  
        isolation:  
          thread:  
            timeoutInMilliseconds: 3000
```

## 4. 다시 확인 -> Product/Display 모두 동작

Tag : step-2-hystrix-timeout

## [실습 Step-2] Hystrix로 Timeout 처리하기

### 5. 정리

- Hystrix를 통해 실행 되는 모든 메소드는 정해진 응답시간 내에 반환 되어야 한다.
- 그렇지 못한 경우, Exception이 발생하며, Fallback이 정의된 경우 수행된다.
- Timeout 시간은 조절할 수 있다. (Circuit 별로 세부 수정 가능하며 뒷 부분에 설명)
- 언제 유용한가 ?

항상 !!

모든 외부 연동은 최대 응답 시간을 가정할 수 있어야 한다.

여러 연동을 사용하는 경우 최대 응답시간을 직접 Control하는 것은 불가능하다 (다양한 timeout, 다양한 자연등..)

Tag : step-2-hystrix-circuit-open

## [실습 Step-2] Hystrix Circuit Open 테스트

1. [display] application.yml 수정하여 Hystrix 프로퍼티 추가.

기본 설정 (테스트 하기 힘들다).

- 10초동안 20개 이상의 호출이 발생 했을때 50% 이상의 호출에서 에러가 발생하면 Circuit Open

```
hystrix:  
  command:  
    default:    # command key. use 'default' for global setting.  
      execution:  
        isolation:  
          thread:  
            timeoutInMilliseconds: 3000    # default 1,000ms  
  circuitBreaker:  
    requestVolumeThreshold: 1    # Minimum number of request to calculate circuit breaker's health. default 20  
    errorThresholdPercentage: 50 # Error percentage to open circuit. default 50
```

변경설정

- h.c.d.circuitBreaker.requestVolumeThreshold : 1
- h.c.d.circuitBreaker.errorThresholdPercentage: 50

Tag : step-2-hystrix-circuit-open

## [실습 Step-2] Hystrix Circuit Open 테스트

### 2. [product] ProductController 다시 수정하여 Exception 던지도록 수정

```
@RestController
@RequestMapping("/products")
public class ProductController {

    @RequestMapping(path = "{productId}", method = RequestMethod.GET)
    public String getProductInfo(@PathVariable String productId) {

        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        //return "[product id = " + productId + " at " + System.currentTimeMillis() + "]";
        throw new RuntimeException("I/O Exception");
    }
}
```

Thread.sleep(2000) 제거 - 테스트 편의

Throw new RuntimeException 복원 - Exception 발생으로 Circuit Open 유도하기 위해

Tag : step-2-hystrix-circuit-open

## [실습 Step-2] Hystrix Circuit Open 테스트

### 3. 확인

<http://localhost:8082/products/22222>

› Exception 발생 확인

<http://localhost:8081/displays/11111>

› 10초 이상 호출. Exception 메세지 변경되었나 확인

```
t = org.springframework.web.client.HttpServerErrorException: 500 null  
t = java.lang.RuntimeException: Hystrix circuit short-circuited and is OPEN  
t = java.lang.RuntimeException: Hystrix circuit short-circuited and is OPEN
```

› Product Console 지운 후 Display 호출 및 확인

'Hystrix Circuit Short-Circuited and is OPEN'이 Display에 출력되는 경우는  
Product 콘솔에는 아무것도 찍히지 않는다.

-> Circuit Open (호출 차단)

Tag : step-2-hystrix-circuit-open

## [실습 Step-2] Hystrix Circuit Open 테스트

### 4. 부가 설명

- Circuit Open 여부는 통계를 기반으로 한다.
- 최근 10초간 호출 통계 (metrics.rollingStats.timeInMilliseconds : 10000)
- 최소 요청 갯수(20) 넘는 경우만 (circuitBreaker.requestVolumeThreshold : 20)
- 에러 비율 넘는 경우(50%) (circuitBreaker.errorThresholdPercentage : 50)
- 한번 Circuit이 오픈되면 5초간 호출이 차단되며, 5초 경과후 단 '1개'의 호출을 허용하며 (Half-Open), 이것이 성공하면 Circuit을 다시 CLOSE하고, 여전히 실패하면 Open이 5초 연장된다.  
(circuitBreaker.sleepWindowInMilliseconds : 5000)

## [실습 Step-2] Hystrix Circuit Breaker의 단위는 ??

- Circuit Breaker의 단위 ?
  - 에러 비율을 통계의 단위
  - Circuit Open / Close가 함께 적용되는 단위
  - 즉, A 메소드와 B 메소드가 같은 Circuit Breaker를 사용한다면, A와 B의 에러 비율이 함께 통계내어지고, Circuit이 오픈되면 함께 차단된다.
- Circuit의 지정은 ?
  - `commandKey`라는 프로퍼티로 지정 가능.
  - @HystrixCommand에서는 지정하지 않는 경우 메소드 이름 사용
    - 이렇게 사용하지 말것 !
    - 메소드 이름은 겹칠 수 있으며, 나중에 나오는 Feign의 경우 또 다르기 때문에 헷갈릴 수 있다.
    - 항상 직접 지정 해서 사용하기

Tag : step-2-hystrix-command-key

## [실습 Step-2] Hystrix Circuit Breaker의 단위는 ??

### 1. [display] `commandKey` 부여하기

```
@Override  
@HystrixCommand(commandKey = "productInfo", fallbackMethod = "getProductInfoFallback")  
public String getProductInfo(String productId) {  
    return this.restTemplate.getForObject(url + productId, String.class);  
}
```

### 2. [display] application.yml에 commandKey로 속성 지정해보기

```
hystrix:  
  command:  
    productInfo: # command key. use 'default' for global setting.  
    execution:  
      isolation:  
        thread:  
          timeoutInMilliseconds: 3000 # default 1,000ms  
    circuitBreaker:  
      requestVolumeThreshold: 1 # Minimum number of request to calculate circuit breaker's health. default 20  
      errorThresholdPercentage: 50 # Error percentage to open circuit. default 50
```

'default' : global 설정. 이 위치에 'commandKey'값을 넣으면, 해당 Circuit Breaker에만 해당 속성 적용된다.

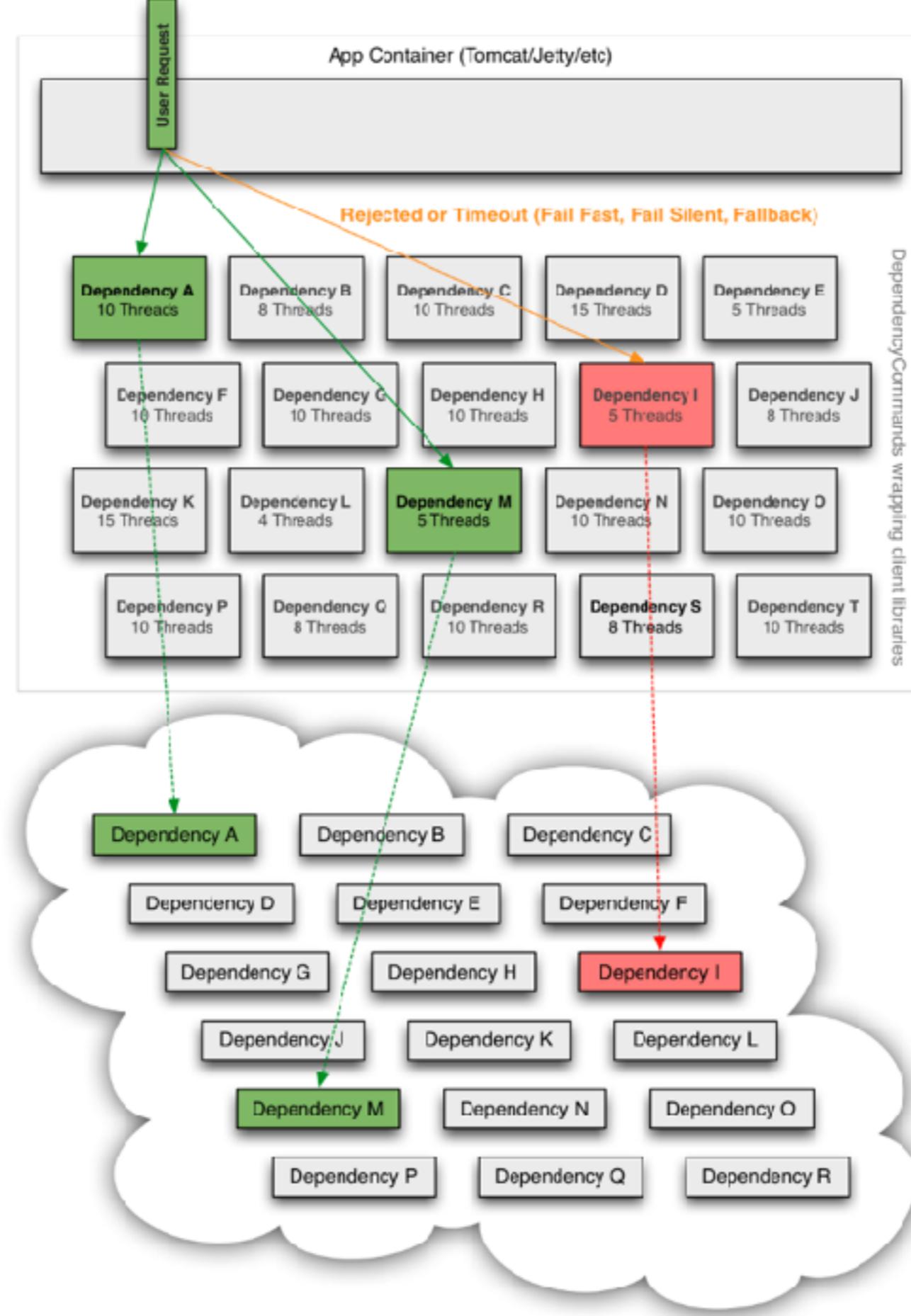
# Hystrix - Circuit Breaker

Hystrix를 쓴다는 것의 또다른 의미

- Hystrix를 통해 실행한다는 것은 별도의 Thread Pool에서 내 Code를 실행한다는 것
- ThreadPool을 여러개의 Circuit Breaker가 공유할 수 있다.

ThreadPool 사용을 통해서 얻을 수 있는 것

- 내 시스템의 Resource가 다양한 기능(서비스)들을 위해 골고루 분배 될 수 있도록



# Hystrix - Circuit Breaker

동시에 실행될 수 있는 Command의 갯수 제한 - ThreadPool 설정

- HystrixCommand로 선언된 Method는 default로 호출한 Thread가 아닌 별도의 ThreadPool에서 '대신' 실행된다.
- ThreadPool에 대한 설정을 하지 않으면 Default로는 @HystrixCommand가 선언된 클래스 이름이 ThreadPool 이름으로 사용된다.

```
public class MyService {  
    @HystrixCommand(commandKey = 'serviceA')  
    public String anyMethodWithExternalDependency1() {  
  
        URI uri = URI.create("http://172.32.1.22:8090/recommended");  
  
        String result = this restTemplate.getForObject(uri, String.class);  
  
        return result;  
    }  
  
    @HystrixCommand(commandKey = 'serviceB')  
    public String anyMethodWithExternalDependency2() {  
  
        URI uri = URI.create('http://172.32.2.33:8090/search');  
        String result = this restTemplate.getForObject(uri, String.class);  
  
        return result;  
    } }
```

- 위의 같은 경우 두개의 메소드는 각각 Circuit Breaker를 갖지만, 두개의 Command가 'MyService'라는 하나의 ThreadPool에서 수행됨.

# Hystrix - Circuit Breaker

## ThreadPool의 세부 옵션 설정

```
@HystrixCommand(commandKey = 'ExtDep1', fallbackMethod='recommendFallback',
commandGroupKey = 'Myservice1', commandProperties = {
    @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds",
value = "500")
},
threadPoolProperties = {
    @HystrixProperty(name = "coreSize", value = "30"),
    @HystrixProperty(name = "maxQueueSize", value = '101'),
    @HystrixProperty(name = "keepAliveTimeMinutes", value = "2"),
    @HystrixProperty(name = "queueSizeRejectionThreshold", value = "15"),
    @HystrixProperty(name = "metrics.rollingStats.numBuckets", value = "12"),
    @HystrixProperty(name = "metrics.rollingStats.timeInMilliseconds", value =
"value = "1440")
}
```

- Hystrix에 관한 모든 세부 옵션 : <https://github.com/Netflix/Hystrix/wiki/Configuration>
- ThreadPool의 기본 사이즈는 100이므로 적용시 필히 상향 여부 검토 필요

# Hystrix - Circuit Breaker

Project에 적용하기

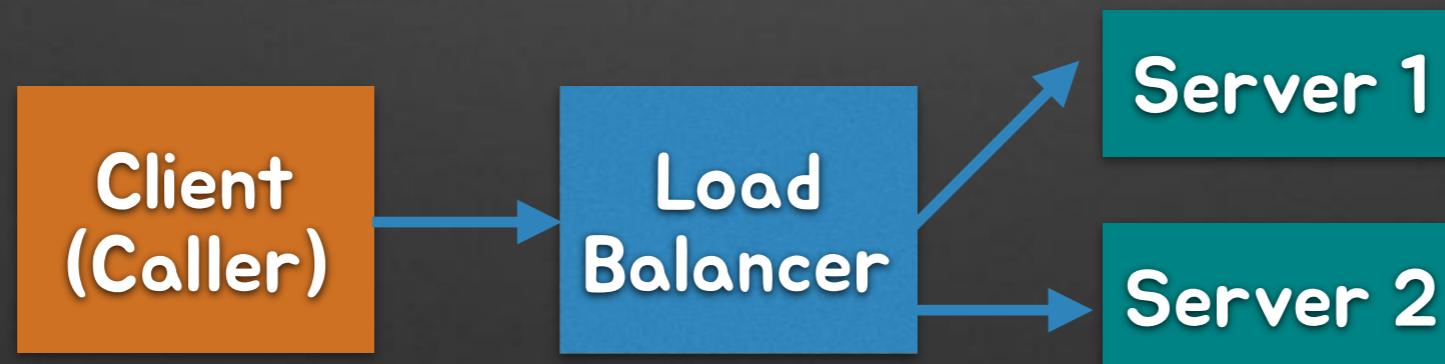
- Spring Boot 프로젝트인 경우
  - Spring Cloud Dependency 추가 와 @EnableHystrix 주석
- Spring 프로젝트인 경우 혹은 AOP 사용 가능한 경우
  - Netflix Hystrix-Javanica
  - <https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib/hystrix-javanica>
- 기타 Java Project인 경우
  - (Pure) Netflix Hystrix
  - 주석 없이 Coding으로 가능
  - <https://github.com/Netflix/Hystrix>

```
public class CommandHelloWorld extends HystrixCommand<String> {  
  
    private final String name;  
  
    public CommandHelloWorld(String name) {  
        super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));  
        this.name = name;  
    }  
  
    @Override  
    protected String run() {  
        return "Hello " + name + "!";  
    }  
}
```

- 시작하기에 앞서
- 모놀리틱 아키텍쳐
- Microservices Architecture(MSA)
- Cloud Native
- Netflix OSS, Spring Cloud
  - Hystrix - Circuit Breaker
  - Eureka - Service Discovery <- 여기 할 차례예요
  - Zuul - API Gateway
- Configuration Server, Tracing, Monitoring, 그리고 남은 이야기..

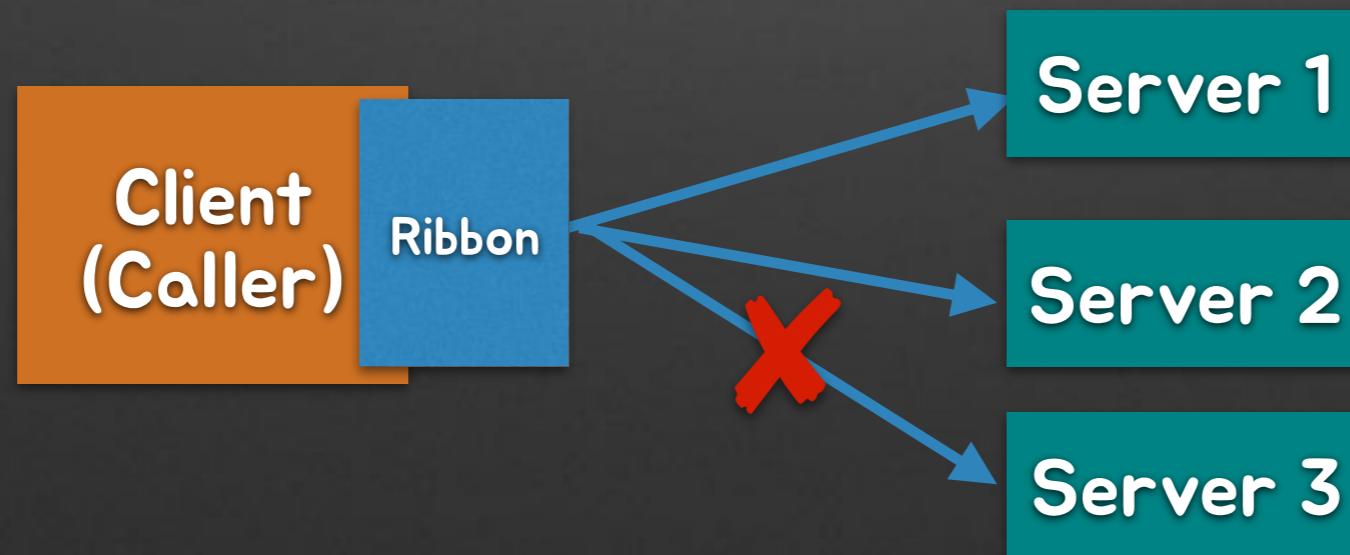
# Server Side LoadBalancer

- 일반적인 L4 Switch 기반의 Load Balancing
- Client 는 L4 의 주소만 알고 있음
- L4 Switch 는 Server 의 목록을 알고 있음(Server Side Load Balancing)
- H/W Server Side Load Balancer 단점 (장점도 있지만..)
  - H/W 가 필요 (비용 up, 유연성 down)
  - 서버 목록의 추가를 위해서는 설정 필요 (자동화 어려움)
  - Load Balancing Schema 0| 한정적 (Round Robin, Sticky)
- 12 factors 의 dev/prod 를 만족하기 어려움



# Client LoadBalancer - Ribbon

- Client (API Caller) 에 탑재되는 S/W 모듈
- 주어진 서버 목록에 대해서 Load Balancing 을 수행함
- Ribbon 의 장점 (단점도 있지만... )
  - H/W 가 필요 없이 S/W 로만 가능 (비용 down, 유연성 up)
  - 서버 목록의 동적 변경이 자유로움 (단 Coding 필요)
  - Load Balancing Schema 이 마음대로 구성 가능 (단 Coding 필요)



Tag : step-3-baseline

## [실습 Step-3] RestTemplate에 Ribbon 적용하기

### 1. [product] ProductController 정상 코드로 복원

- 준비작업 (Application 정상화)
- Sleep 제거 / Throw Exception 제거

```
@RequestMapping(path = "{productId}", method = RequestMethod.GET)
public String getProductInfo(@PathVariable String productId) {

    // try {
    //     Thread.sleep(2000);
    // } catch (InterruptedException e) {
    //     e.printStackTrace();
    // }

    return "[product id = " + productId + " at " + System.currentTimeMillis() + "]";
    // throw new RuntimeException("I/O Exception");
}
```

Tag : step-3-baseline

## [실습 Step-3] RestTemplate에 Ribbon 적용하기

2. [display] build.gradle에 dependency 추가

- compile('org.springframework.cloud:spring-cloud-starter-netflix-ribbon')

잠시 대기 후 외부 라이브러리에 아래 확인

- ▶  Gradle: com.netflix.ribbon:ribbon:2.2.5
- ▶  Gradle: com.netflix.ribbon:ribbon-core:2.2.5
- ▶  Gradle: com.netflix.ribbon:ribbon-httpclient:2.2.5
- ▶  Gradle: com.netflix.ribbon:ribbon-loadbalancer:2.2.5
- ▶  Gradle: com.netflix.ribbon:ribbon-transport:2.2.5

Tag : step-3-baseline

## [실습 Step-3] RestTemplate에 Ribbon 적용하기

### 3. [display] DisplayApplication의 RestTemplate 빈에 @LoadBalanced 추가

```
@SpringBootApplication
@EnableCircuitBreaker
public class DisplayApplication {

    @LoadBalanced
    @Bean
    RestTemplate restTemplate(){
        return new RestTemplate();
    }

    public static void main(String[] args) { SpringApplication.run(DisplayApplication.class, args); }
}
```

Tag : step-3-ribbon-loadbalanced

## [실습 Step-3] RestTemplate에 Ribbon 적용하기

### 4. [display] ProductRemoteServiceImpl에서 주소 제거하고 'product'로 변경

```
// private final String url = "http://localhost:8082/products/";  
private final String url = "http://product/products/";
```

### 5. [display] application.yml에 ribbon 설정 넣기

```
product:  
  ribbon:  
    listOfServers: localhost:8082
```

- listOfServers에 서버 목록 추가

Tag : step-3-ribbon-loadbalanced

## [실습 Step-3] RestTemplate에 Ribbon 적용하기

### 7. 확인

<http://localhost:8081/displays/11111>

동작 !!

RestTemplate에 주소가 아닌 단지 이름 'product'을 넣었는데 동작

Tag : step-3-ribbon-loadbalanced

## [실습 Step-3] RestTemplate에 Ribbon 적용하기

### 8. 정리

- Ribbon 디펜던시 추가 후 RestTemplate에 '@LoadBalanced'
- 설정에 특정 서비스(product)의 주소 설정
- RestTemplate 사용 시 주소 넣지 않고 서비스 이름(product) 사용
- 로드 밸런스는 ???

Tag : step-3-ribbon-retry

## [실습 Step-3] Ribbon의 Retry 기능

3. [display] application.yml에 서버 주소 추가 및 Retry 관련 속성 조정

```
product:  
  ribbon:  
    listOfServers: localhost:8082,localhost:7777  
    MaxAutoRetries: 0  
    MaxAutoRetriesNextServer: 1
```

4. [display] build.gradle에 retry dependency 추가  
compile('org.springframework.retry:spring-retry:1.2.2.RELEASE')

### 5. 확인

<http://localhost:8081/displays/11111>

localhost:7777은 없는 주소 이므로 Exception 발생. 그러나 Ribbon Retry로 항상 성공

→ Round Robin Client Load Balancing & Retry.

Tag : step-3-ribbon-retry

## [실습 Step-3] Ribbon의 Retry 기능

### 6. 주의

- Retry를 시도하다가도 HystrixTimeout이 발생하면, 즉시 에러 반환 리턴할 것이다. (Hystrix로 Ribbon을 감싸서 호출한 상태이기 때문에)
- Retry를 끄거나, 재시도 횟수를 0으로 하여도 해당 서버로의 호출이 항상 동일한 비율로 실패하지는 않는다. (실패한 서버로의 호출은 특정 시간동안 Skip 되고 그 간격은 조정된다 - BackOff)
- classpath에 retry 가 존재해야 한다는 점 주의

### 6. 정리

- Ribbon은 여러 Component에 내장되어있으며, 이를 통해 Client Load Balancing이 수행 가능하다.
- Ribbon에는 매우 다양한 설정이 가능하다 (서버선택, 실패시 Skip 시간, Ping 체크)
- Ribbon에는 Retry 기능이 내장 되어있다.
- Eureka와 함께 사용될 때 강력하다 (뒤에 실습)

Ribbon 예제에서 서버 목록을 yaml에 직접 넣었는데 자동화 할 방법은 ?

'서버가 새롭게 시작되면 그것을 감지하여 목록에 자동으로 추가되고,  
서버가 종료되면 자동으로 목록에서 삭제하기 위한 방법은 없을까 ?'

# Dynamic Service Discovery - Eureka

- **Service Registry**

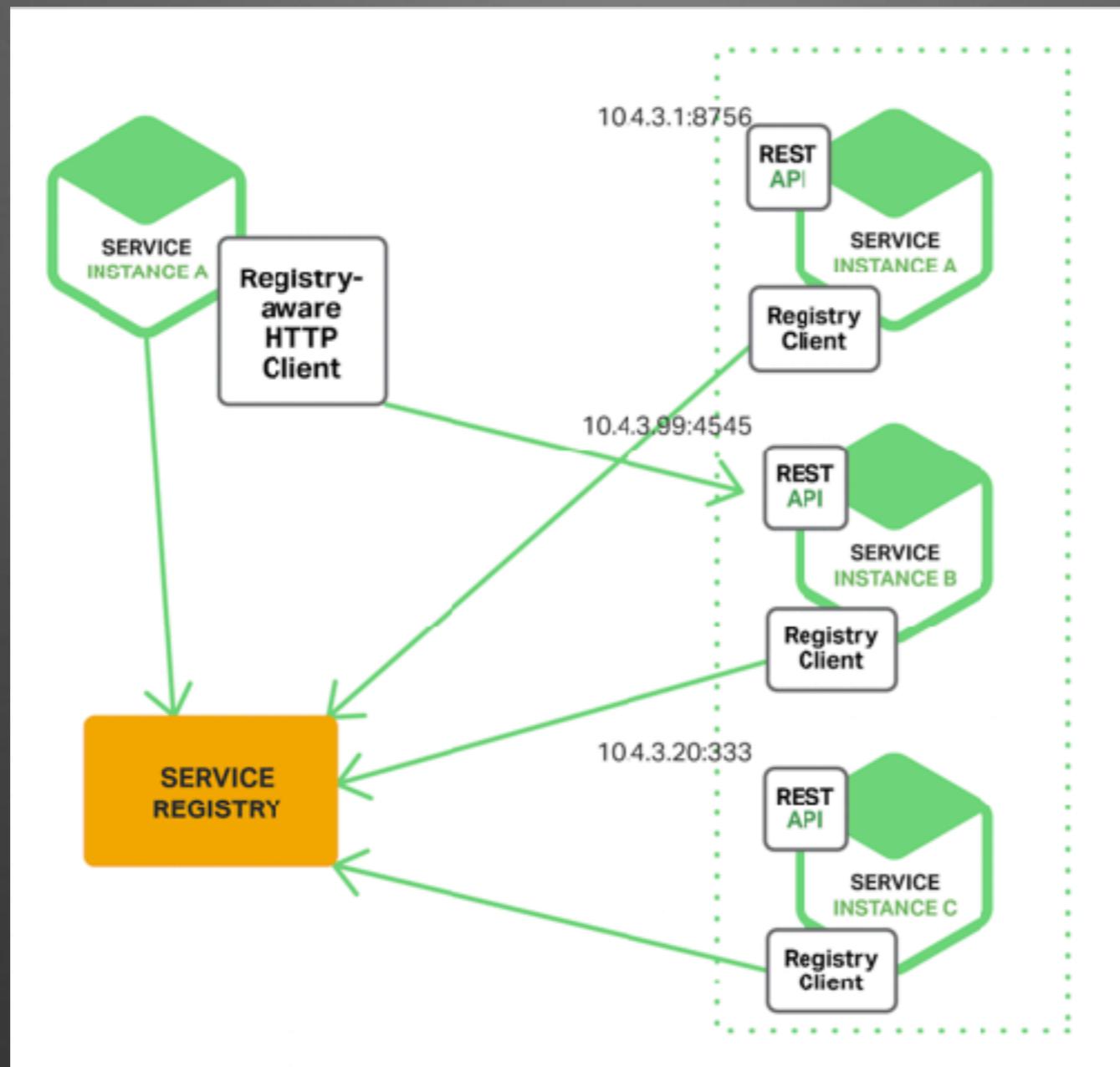
- 서비스 탐색, 등록
- 클라우드의 전화번호부
- (단점) 침투적 방식 코드 변경



- **DiscoveryClient**

- spring-cloud에서 서비스 레지스트리 사용 부분을 추상화(Interface)
- Eureka, Consul, Zookeeper, etcd 등의 구현체가 존재

# Dynamic Service Discovery - Eureka



Ribbon은 Eureka와 결합하여 사용 할 수 있으며 서버 목록을 자동으로 관리.

# Eureka Server(Registry) 만들기

```
@EnableEurekaServer  
@SpringBootApplication  
public class EurekaServiceApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(EurekaServiceApplication.class);  
    }  
}
```

Spring Boot Application 작성 후 `@EnableEurekaServer` 만 주석을 달면 끝

# Eureka Client 만들기 - 내 서버 정보 등록 하기

## - 내가 호출의 대상이 되고 싶을 때

```
@EnableEurekaClient
```

```
@SpringBootApplication
```

```
public class EurekaServiceApplication {
```

```
    public static void main(String[] args) {
```

```
        SpringApplication.run(EurekaServiceApplication.class);
```

```
}
```

```
}
```

```
application.yml
```

```
eureka:
```

```
    client:
```

```
        serviceUrl:
```

```
            defaultZone: http://localhost:8761/eureka/
```

Spring Boot Application 작성 후 `@EnableEurekaServer` 만 주석을 달면 끝

## Eureka Client 만들기 - 다른 서버의 목록을 가져오고 싶을 때

- Eureka에 등록된 서버 주소 목록을 알고 싶을 때

```
@Service  
class ServiceInstanceRestController {  
    @Autowired  
    private DiscoveryClient discoveryClient;  
    public void doSomething() {  
        List<ServiceInstance> = this.discoveryClient.getInstances(applicationName);  
    }  
}  
  
application.yml  
spring:  
    cloud:  
        service-registry:  
            auto-registration:  
                enabled: false      # 현재 인스턴스를 Registry에 등록하지 않고 싶을 때 false
```

- 특별한 시나리오 없이 DiscoveryClient 직접 호출 필요 없음
- (Ribbon이 DiscoveryClient를 이용해서 서버 목록을 가져옴)

# Eureka in Spring Cloud

- 서버 시작 시 Eureka Server(Registry)에 자동으로 자신의 상태를 등록(UP)  
  > eureka.client.register-with-eureka : true(default)
- 주기적 HeartBeat 으로 Eureka Server에 자신이 살아 있음을 알림  
  > eureka.instance.lease-renewal-interval-in-seconds: 30(default)
- 서버 종료 시 Eureka Server에 자신의 상태 변경(DOWN) 혹은 자신의 목록 삭제
- Eureka 상에 등록된 이름은 'spring.application.name'

Tag : step-4-baseline

## [실습 Step-4] Eureka Server 실행

### 1. [준비작업] Eureka Server 띄우기

시간 절약을 위해 미리 준비된 Eureka Server 사용

- > git reset HEAD --hard
- > git checkout tags/step-4-baseline -b my-step-4

### 2. [eurekaserver] 소스 보기 build.gradle

```
compile('org.springframework.cloud:spring-cloud-starter-netflix-eureka-server')
```

application.yml : 데모용 각종 설정들

Main Class : @EnableEurekaServer

# [실습 Step-4] Eureka Server 실행

## 3. [eurekaserver] 서버 실행

EurekaServer Application 실행  
[http://localhost:8761/ 확인](http://localhost:8761/)

The screenshot shows the Spring Eureka web interface. At the top, it displays "spring Eureka" and "HOME LAST 1000 SINCE STARTUP". Below this is a "System Status" section with two tables:

Environment	test
Data center	default

Current time	2017-06-12T03:09:49 +0900
Uptime	00:00
Lease expiration enabled	true
Renewal threshold	1
Renewal (est min)	0

A red warning message below the tables states: "THE SELF PRESERVATION MODE IS TURNED OFF. THIS MAY NOT PROTECT INSTANCE EXPIRY IN CASE OF NETWORK/OTHER PROBLEMS."

The next section is "DS Replicas", which lists "localhost" as a replica.

Finally, there is a section titled "Instances currently registered with Eureka" with a table:

Application	AMIs	Availability Zones	Status
No instances available			

Tag : step-4-eureka-client

## [실습 Step-4] Eureka Client 적용하기 - Product 서버

### 1. [product] build.gradle

```
compile('org.springframework.cloud:spring-cloud-starter-netflix-eureka-client')
```

### 2. [product] ProductApplication : @EnableEurekaClient

```
@SpringBootApplication  
@EnableEurekaClient  
public class ProductApplication {  
    public static void main(String[] args) { SpringApplication.run(ProductApplication.class); }  
}
```

### 3. [product] application.yml

```
eureka:  
  instance:  
    prefer-ip-address: true
```

OS에서 제공하는 hostname 대신 자신의 ip address를 사용하여 등록

Tag : step-4-eureka-client

## [실습 Step-4] Eureka Client 적용하기 - Product 서버

### 4. [display] build.gradle

```
compile('org.springframework.cloud:spring-cloud-starter-netflix-eureka-client')
```

### 5. [display] DisplayApplication : @EnableEurekaClient

```
@SpringBootApplication
@EnableCircuitBreaker
@EnableEurekaClient
public class DisplayApplication {

    @LoadBalanced
    @Bean
    RestTemplate restTemplate() { return new RestTemplate(); }

    public static void main(String[] args) { SpringApplication.run(DisplayApplication.class, args); }
}
```

### 6. [display] application.yml

```
eureka:
  instance:
    prefer-ip-address: true
```

Tag : step-4-eureka-client

## [실습 Step-4] Eureka Client 적용하기 - 서버 확인

7. Display / Product 서버 둘 다 재시작 후 Eureka 서버 확인

<http://localhost:8761/> 확인

Application	AMIs	Availability Zones	Status
DISPLAY	n/a (1)	(1)	UP (1) - <a href="http://10.202.32.233:display:8081">10.202.32.233:display:8081</a>
PRODUCT	n/a (1)	(1)	UP (1) - <a href="http://10.202.32.233:product:8082">10.202.32.233:product:8082</a>

Tag : step-4-eureka-client

## [실습 Step-4] Eureka 서버 주소 직접 명시

### 8. [product/display] application.yml에 Eureka Server 주소 명시

```
eureka:  
  instance:  
    prefer-ip-address: true  
  client:  
    service-url:  
      defaultZone: http://127.0.0.1:8761/eureka # default address
```

default 값이 http://127.0.0.1:8761/eureka 이므로 로컬 테스트 시 설정 불필요

Tag : step-4-eureka-client

## [실습 Step-4] Eureka 서버 주소 직접 명시

### 9. 정리

@EnableEurekaServer / @EnableEurekaClient를 통해 서버 구축, 클라이언트 Enable 가능하다.

@EnableEurekaClient를 붙인 Application은 Eureka 서버로 부터 남의 주소를 가져오는 역할과 자신의 주소를 등록하는 역할을 둘다 수행 가능하다.

Eureka Client가 Eureka Server에 자신을 등록할 때 `spring.application.name`이 이름으로 사용된다.

Tag : step-4-eureka-listOfServers

## [실습 Step-4] RestTemplate에 Eureka 적용하기

### 1. 목적

Display -> Product 출시시에 Eureka를 적용하여 ip주소를 코드나 설정  
‘모두’에서 제거하려고 함.

### 2. [display] application.yml 변경 (주석 처리)

```
product:  
  ribbon:  
    #      listOfServers: localhost:8082,localhost:7777  
    MaxAutoRetries: 0  
    MaxAutoRetriesNextServer: 1
```

product.ribbon.listOfServers 제거

-> 서버 주소는 Eureka Server에서 가져와라 !

Tag : step-4-eureka-listOfServers

## [실습 Step-4] RestTemplate에 Eureka 적용하기

### 3. [확인] display 서버 재시작

<http://localhost:8081/displays/11111>

- Product의 주소가 설정/코드에서 모두 제거
- 코드에서는 <http://product/> 으로 접근

### 4. 정리

- [@LoadBalanced RestTemplate](#)에는 Ribbon + Eureka 연동
- Eureka Client가 설정되면, 코드상에서 서버 주소 대신 Application 이름을 명시해서 호출 가능
- Ribbon의 Load Balancing과 Retry가 함께 동작

# Declarative Http Client - Feign

- Interface 선언을 통해 자동으로 Http Client 를 생성
- RestTemplate 은 concrete 클래스라 테스트하기 어렵다
- 관심사의 분리
  - 서비스의 관심 - 다른 리소스, 외부 서비스 호출과 리턴값
  - 관심 X - 어떤 URL, 어떻게 파싱할 것인가
- Spring Cloud 에서 Open-Feign 기반으로 Wrapping 한 것이 Spring Cloud Feign

# Declarative Http Client - Spring Cloud Feign

인터페이스 선언 만으로 Http Client 구현물을 만들어 줌

```
@FeignClient(name="dp", url = "http://localhost:8080/")

public interface ProductResource {
    @RequestMapping(value = "/query/{itemId}", method=RequestMethod.GET)
    String getItemDetail(
        @PathVariable(value = "itemId") String itemId);
}
```

- `@FeignClient`을 Interface에 명시
- 각 API를 Spring MVC Annotation을 이용해서 정의

How to Use

```
@Autowired
ProductResource productResouce;
```

- `@Autowired`을 통해 DI 받아 사용

Tag : step-5-feign-url

## [실습 Step-5] Feign 클라이언트 사용하기

### 1. 목적

Display -> Product 호출시에 Feign을 사용해서 RestTemplate을 대체해 보려고 함.

### 2. [display] build.gradle에 dependency 추가

- compile('org.springframework.cloud:spring-cloud-starter-openfeign')

# [실습 Step-5] Feign 클라이언트 사용하기

## 3. [display] DisplayApplication에 @EnableFeignClients 추가

```
@SpringBootApplication  
@EnableCircuitBreaker  
@EnableEurekaClient  
@EnableFeignClients  
public class DisplayApplication {  
  
    @LoadBalanced  
    @Bean  
    RestTemplate restTemplate() { return new RestTemplate(); }  
  
    public static void main(String[] args) { SpringApplication.run(DisplayApplication.class, args); }  
}
```

## 4. [display] Feign용 Interface 추가

- 이전 'ProductRemoteService' 사용하지 않고 데모를 위해 별도 정의

```
package com.elevenst.service;  
  
public interface FeignProductRemoteService {  
    String getProductInfo(String productId);  
}
```

# [실습 Step-5] Feign 클라이언트 사용하기

## 5. [display] FeignProductRemoteService에 Annotation 넣기

```
package com.elevenst.service;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;

@FeignClient(name = "product", url = "http://localhost:8082/")
public interface FeignProductRemoteService {
    @RequestMapping(path = "/products/{productId}")
    String getProductInfo(@PathVariable("productId") String productId);
}
```

Client 구현 완성. 서버 가동시

‘FeignProductRemoteService’용 Spring Bean 자동 생성

‘FeignProductRemoteService’를 주입 받아서 사용하면 됨.

# [실습 Step-5] Feign 클라이언트 사용하기

## 6. [display] DisplayController에서 호출 부분 변경

```
@RestController
@RequestMapping(path = "/displays")
public class DisplayController {

    private final ProductRemoteService productRemoteService;
    private final FeignProductRemoteService feignProductRemoteService;

    public DisplayController(ProductRemoteService productRemoteService,
                            FeignProductRemoteService feignProductRemoteService) {
        this.productRemoteService = productRemoteService;
        this.feignProductRemoteService = feignProductRemoteService;
    }

    @GetMapping(path = "/{displayId}")
    public String getDisplayDetail(@PathVariable String displayId) {
        String productInfo = getProductInfo();
        return String.format("[display id = %s at %s %s ]", displayId, System.currentTimeMillis(), productInfo);
    }

    private String getProductInfo() { return feignProductRemoteService.getProductInfo(productId: "12345"); }
}
```

기존 RestTemplate 사용했던 서비스 대신 Feign이 자동으로  
생성해준 서비스 빈을 사용

Tag : step-5-feign-url

## [실습 Step-5] Feign 클라이언트 사용하기

### 7. 동작 확인

<http://localhost:8081/displays/11111>

### 8. 정리

Feign은 Interface 선언을 통해 자동으로 HTTP Client를 만들어주는  
**Declarative Http Client**

Open-Feign 기반으로 Spring Cloud가 Wrapping한것이 오늘 실습한  
**Spring Cloud Feign**

방금 한 실습 (Feign + URL 명시)은

- No Ribbon
- No Eureka
- No Hystrix

Tag : step-5-feign-eureka

## [실습 Step-5] Feign + Hystrix,Ribbon,Eureka

### 1. 배경

Feign의 또 다른 강점은 Ribbon + Eureka + Hystrix와 통합되어 있다는 점.

### 2. [display] Feign에 Eureka + Ribbon 적용하기

```
@FeignClient(name = "product")
public interface FeignProductRemoteService {
    @RequestMapping(path = "/products/{productId}")
    String getProductInfo(@PathVariable("productId") String productId);
}
```

@FeignClient에서 url만 제거  
완성 !!!

Tag : step-5-feign-eureka

## [실습 Step-5] Feign + Hystrix,Ribbon,Eureka

### 3. Feign의 동작

@FeignClient에 URL 명시 시

-> 순수 Feign Client로서만 동작

@FeignClient에 URL 명시 하지 않으면 ?

-> Feign + Ribbon + Eureka 모드로 동작

-> 어떤 서버 호출하나 ? @FeignClient(name='product')

-> 즉, eureka에서 product 서버 목록을 조회해서 ribbon을 통해  
load-balancing하면서 HTTP 호출을 수행

Tag : step-5-feign-hystrix

## [실습 Step-5] Feign + Hystrix,Ribbon,Eureka

### 4. [display] application.yml : Feign + Hystrix

```
feign:  
  hystrix:  
    enabled: true
```

위의 설정이 들어가면 메소드 하나 하나가 Hystrix Command 로서 호출됨

### 5. Feign에서 Fallback은 ? Hystrix 설정은 ?

# Declarative Http Client - Spring Cloud Feign

그럼 Hystrix Fallback은 ??

- Feign으로 정의한 Interface를 직접 구현하고 Spring Bean으로 선언

```
@Component
public class ProductResourceFallback implements ProductResource {

    @Override
    public String getItemDetail(String itemId) {
        return 'default value';
    }
}
```

- Fallback 클래스를 @Feign선언시 명시

```
@FeignClient(fallback= ProductResourceFallback.class, name="dp", url = "http://
localhost:8080/")
public interface ProductResource {

    @RequestMapping(value = "/query/{itemId}", method=RequestMethod.GET)
    String getItemDetail(
        @PathVariable(value = "itemId") String itemId);
}
```

Tag : step-5-feign-hystrix

## [실습 Step-5] Feign + Hystrix,Ribbon,Eureka

### 6. [display] Feign용 Hystrix Fallback 선언 - Spring Bean으로 정의

```
package com.elevenst.service;

import org.springframework.stereotype.Component;

@Component
public class FeignProductRemoteServiceFallbackImpl implements FeignProductRemoteService {
    @Override
    public String getProductInfo(String productId) {
        return "[ this product is sold out ]";
}
```

### 7. [display] Feign용 Hystrix Fallback 명시

```
@FeignClient(name = "product", fallback = FeignProductRemoteServiceFallbackImpl.class)
public interface FeignProductRemoteService {

    @RequestMapping(path = "/products/{productId}")
    String getProductInfo(@PathVariable("productId") String productId);
}
```

@FeignClient에 'fallback' 속성으로 클래스 명시

Tag : step-5-fallbackfactory

## [실습 Step-5] Feign + Hystrix,Ribbon,Eureka

### 8. [display] 기본 Fallback은 에러 원인(Exception)을 알 수 없음 - FallbackFactory 사용

```
package com.elevenst.service;

import feign.hystrix.FallbackFactory;
import org.springframework.stereotype.Component;

@Component
public class FeignProductRemoteServiceFallbackFactory
    implements FallbackFactory<FeignProductRemoteService> {

    @Override
    public FeignProductRemoteService create(Throwable cause) {
        System.out.println("t = " + cause);
        return productId -> "[ this product is sold out ]";
    }
}
```

### 9. [display] Feign용 Hystrix Fallback 명시

```
@FeignClient(name = "product", fallbackFactory = FeignProductRemoteServiceFallbackFactory.class)
public interface FeignProductRemoteService {
    @RequestMapping(path = "/products/{productId}")
    String getProductInfo(@PathVariable("productId") String productId);
}
```

@FeignClient에 'fallback' 속성으로 클래스 명시

Tag : step-5-feign-hystrix-properties

## [실습 Step-5] Feign + Hystrix,Ribbon,Eureka

### 10. [display] Feign용 Hystrix 프로퍼티 정의하는 법

```
hystrix:  
  command:  
    productInfo: # command key. use 'default' for global setting.  
      execution:  
        isolation:  
          thread:  
            timeoutInMilliseconds: 3000 # default 1,000ms  
    circuitBreaker:  
      requestVolumeThreshold: 1 # Minimum number of request to calculate circuit breaker's health. default 20  
      errorThresholdPercentage: 50 # Error percentage to open circuit. default 50  
FeignProductRemoteService#getProductInfo(String):  
  execution:  
    isolation:  
      thread:  
        timeoutInMilliseconds: 3000 # default 1,000ms  
  circuitBreaker:  
    requestVolumeThreshold: 1 # Minimum number of request to calculate circuit breaker's health. default 20  
    errorThresholdPercentage: 50 # Error percentage to open circuit. default 50
```

Feign 사용하는 경우 commandKey 이름 주의

ex) FeignProductRemoteService#getProductInfo(String)

# [실습 Step-5] Feign + Hystrix,Ribbon,Eureka

정리

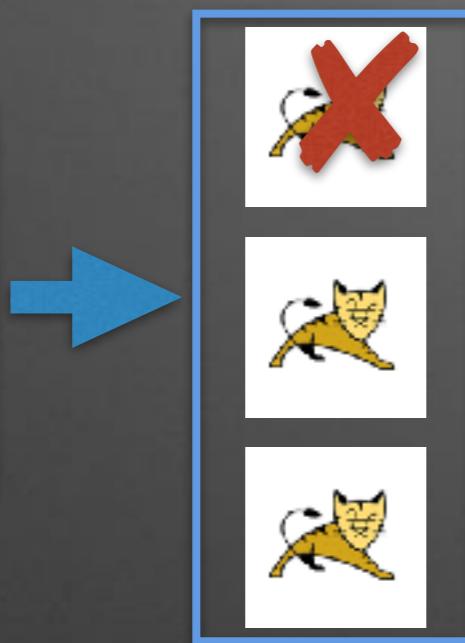
Feign은 인터페이스 선언 + 설정으로 다음과 같은 것들이 가능하다

- Http Client
- Eureka 타겟 서버 주소 획득
- Ribbon을 통한 Client-Side Load Balancing
- Hystrix를 통한 메소드별 Circuit Breaker

# Feign + Hystrix, Ribbon, Eureka

## 장애 유형 별 동작 예

Hystrix  
+  
Ribbon  
+  
Eureka



특정 API 서버의 인스턴스가 한개 Down 된 경우

EUREKA - Heartbeat 송신이 중단 됨으로 일정 시간 후

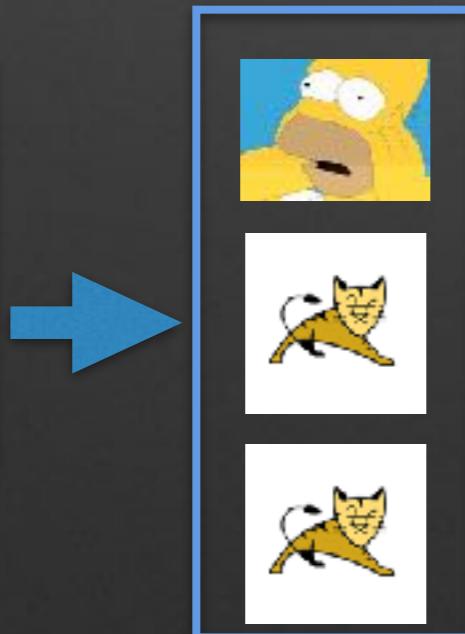
목록에서 사라짐

RIBBON - IOException 이 발생한 경우 다른 인스턴스로  
Retry

Hystrix - Circuit 은 오픈되지 않음( ERROR = 33% )

Fallback, Timeout 은 동작

Hystrix  
+  
Ribbon  
+  
Eureka



특정 API 가 비정상 동작하는 경우 (지연, 에러)

Hystrix - 해당 API 를 호출하는 Circuit Breaker 오픈

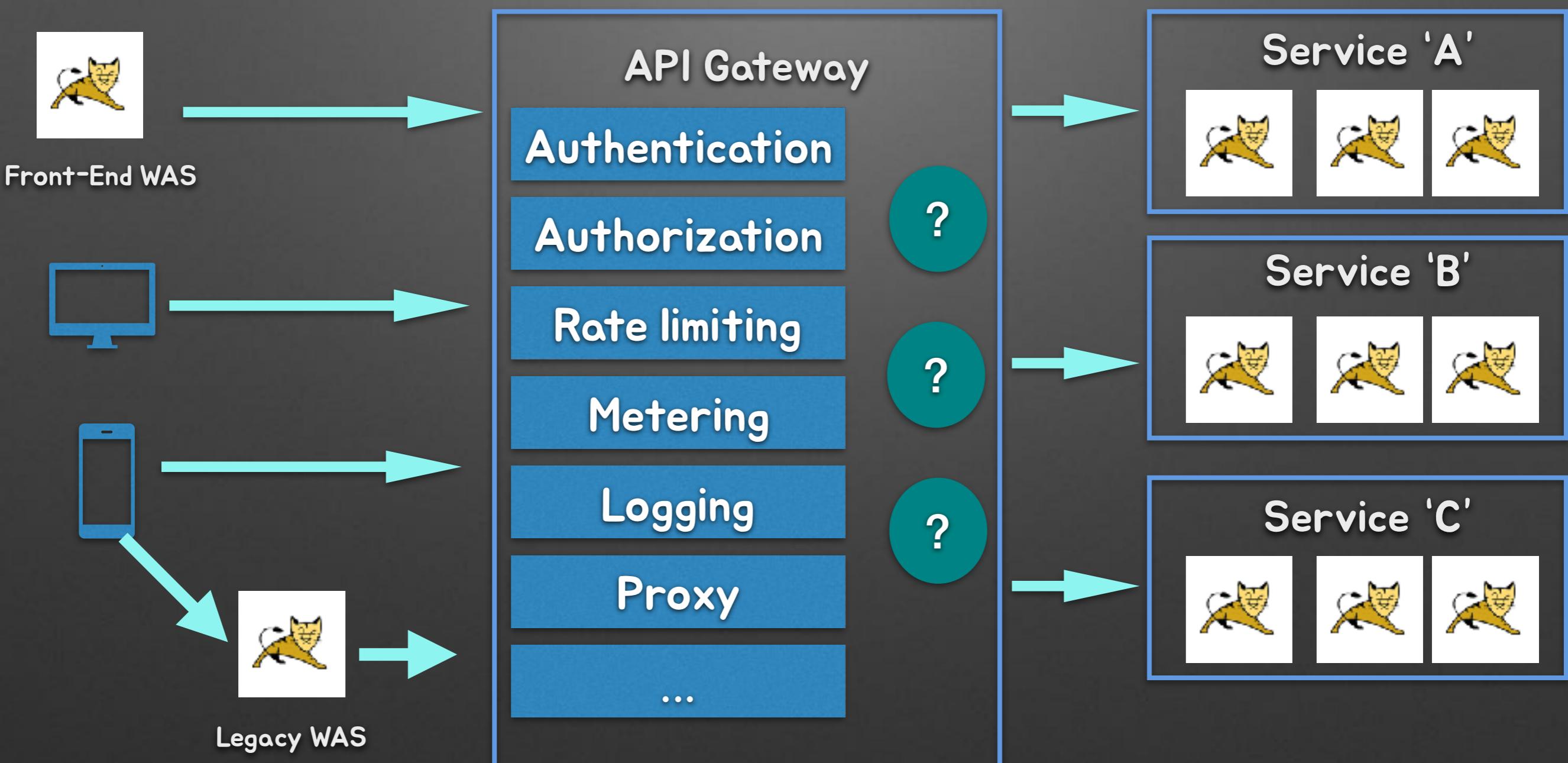
Fallback, Timeout 도 동작

- 시작하기에 앞서
- 모놀리틱 아키텍쳐
- Microservices Architecture(MSA)
- Cloud Native
- Netflix OSS, Spring Cloud
  - Hystrix - Circuit Breaker
  - Eureka - Service Discovery
  - Zuul - API Gateway <- 여기 할 차례예요
- Configuration Server, Tracing, Monitoring, 그리고 남은 이야기..

# API Gateway

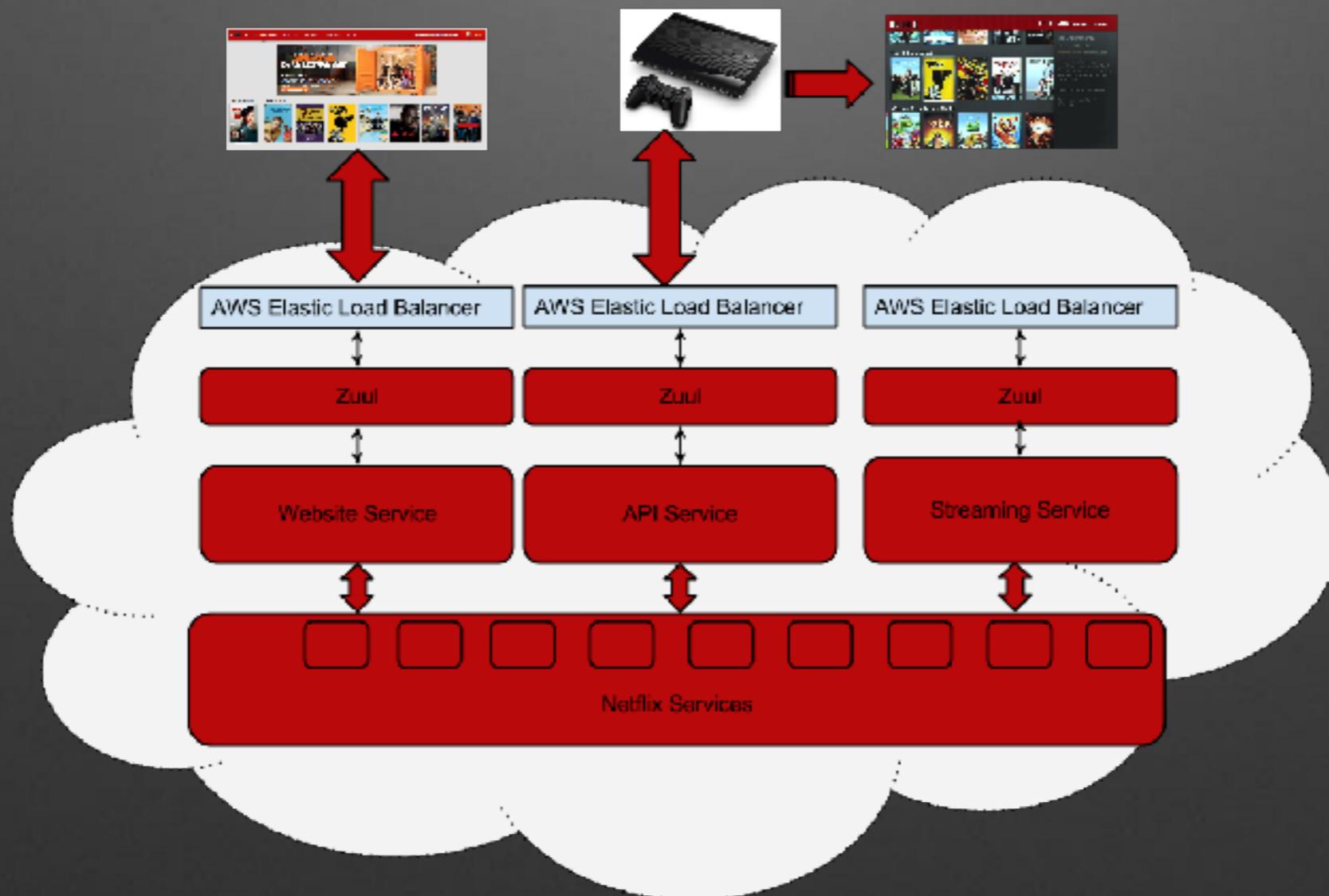
- 클라이언트와 백엔드 서버 사이의 출입문(front door)
- 라우팅(라우팅, 필터링, API변환, 클라이언트 어댑터 API, 서비스 프록시)
- 횡단 관심사 cross-service concerns
  - 보안, 인증(authentication), 인가(authorization)
  - 일정량 이상의 요청 제한(rate limiting)
  - 계측(metering)

# API Gateway



# Netflix Zuul

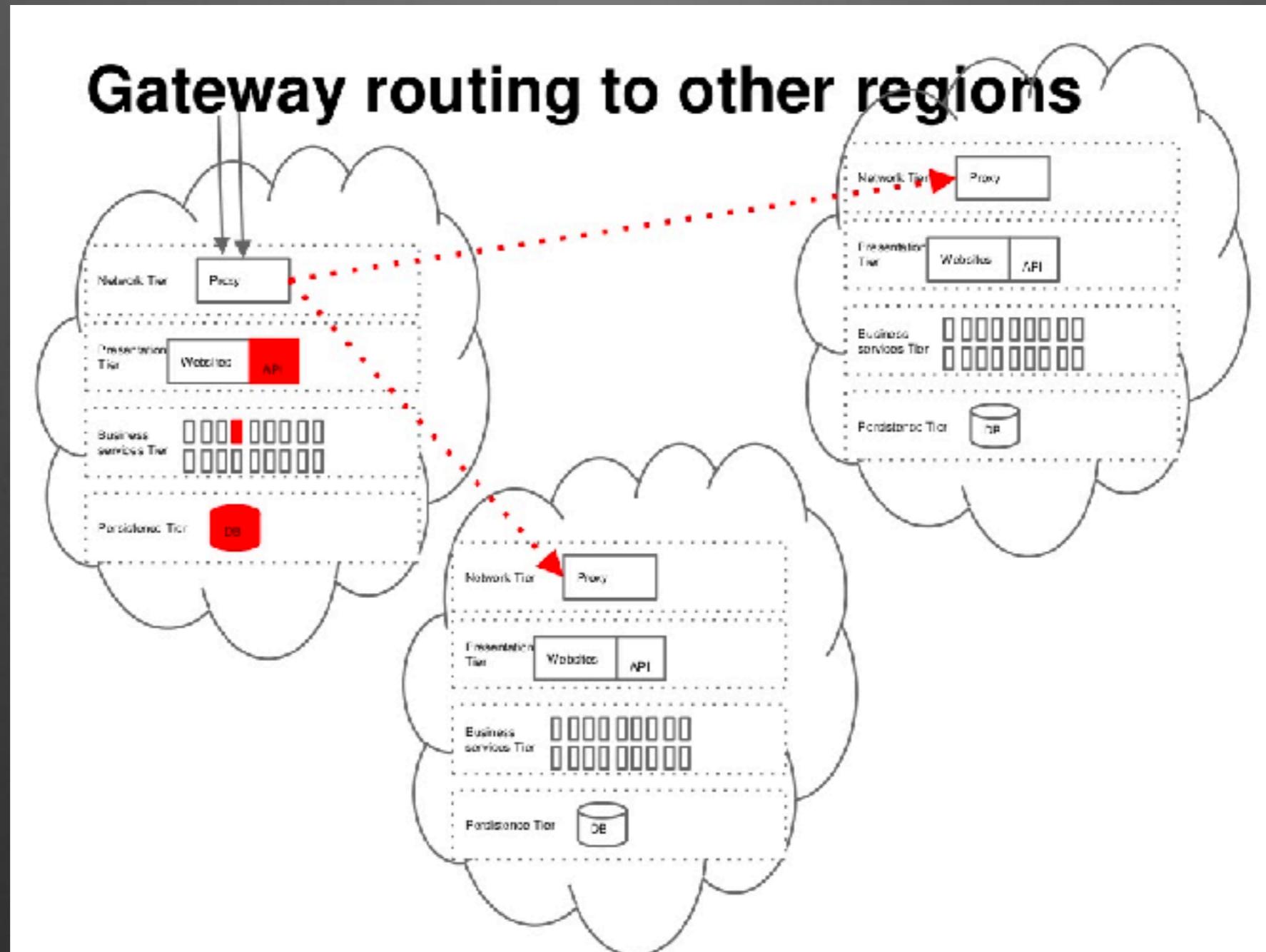
- 마이크로 프록시
- 50개 이상의 AWS ELB 의 앞단에 위치해 3개의 AWS 리전에 걸쳐 하루 백억 이상의 요청을 처리(2015년 기준)



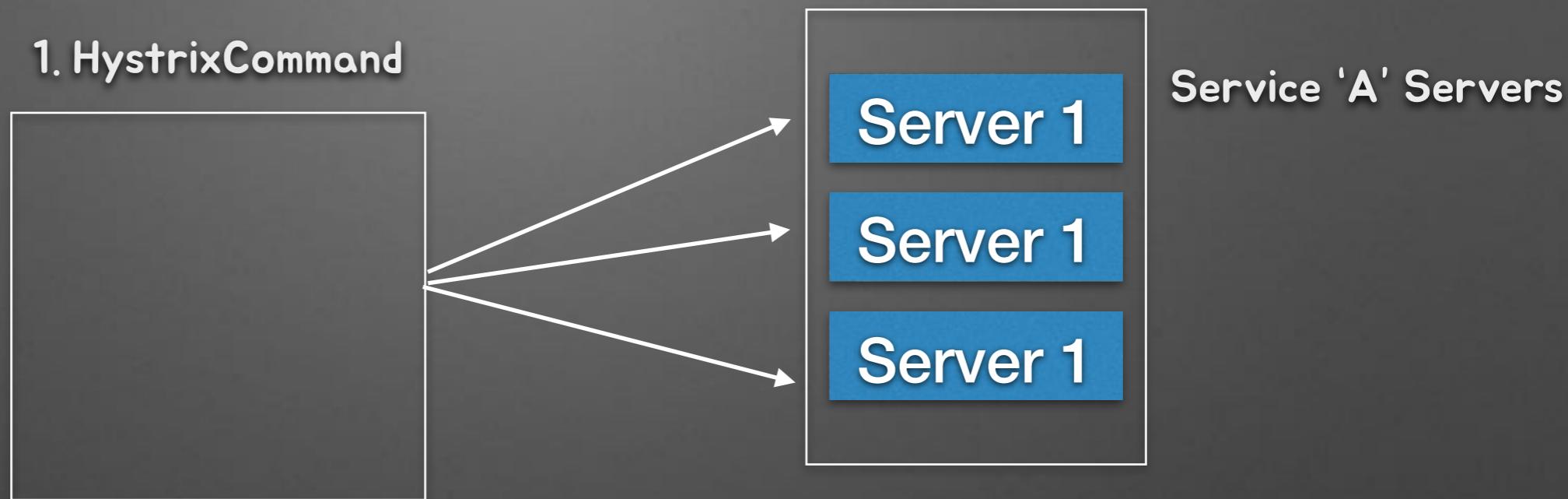
출처 : <https://github.com/Netflix/zuul/wiki/How-We-Use-Zuul-At-Netflix>

출처 : <https://www.slideshare.net/MikeyCohen1/rethinking-cloud-proxies-54923218>

# Netflix Zuul



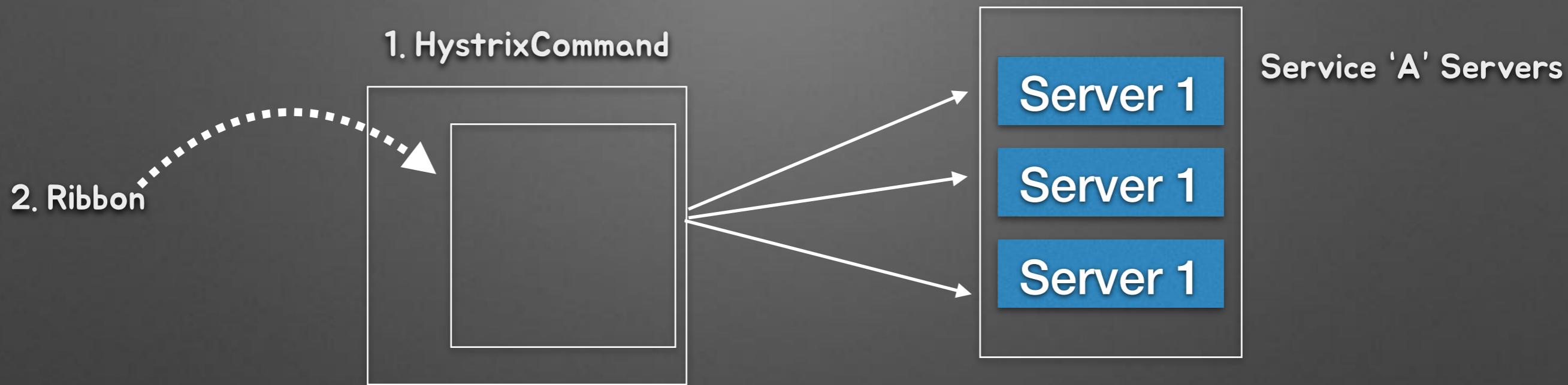
# API Gateway - Zuul



Zuul의 모든 API 요청은 HystrixCommand로 구성되어 전달된다.

- 각 API 경로 (서버군) 별로 Circuit Breaker 생성
- 하나의 서버군이 장애를 일으켜도 다른 서버군의 서비스에는 영향이 없다.
- CircuitBreaker / ThreadPool의 다양한 속성을 통해 서비스 별 속성에 맞는 설정 가능

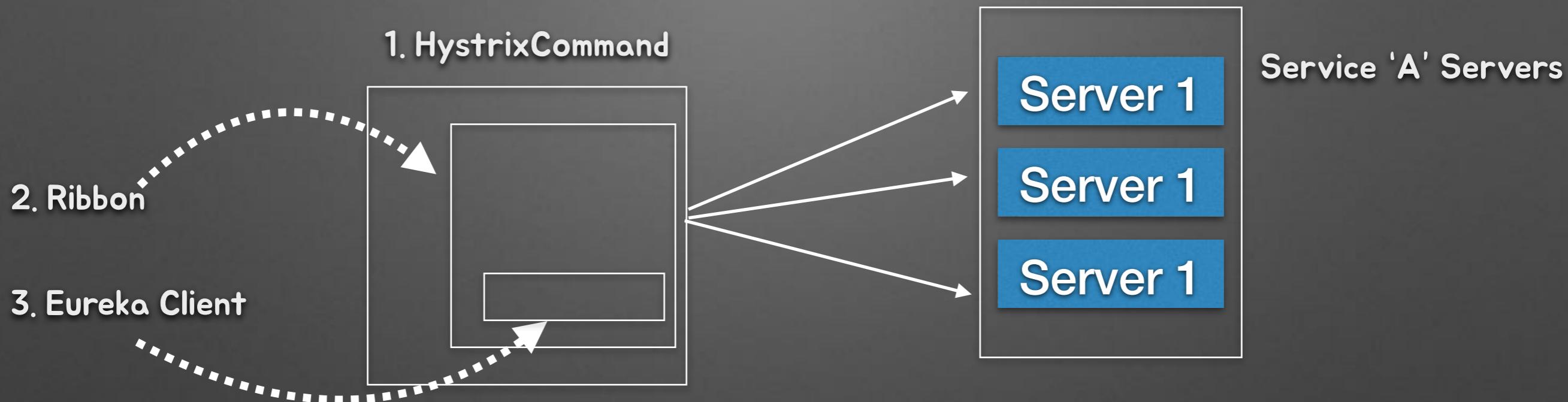
# API Gateway - Zuul



2. API를 전달할 서버의 목록을 갖고 Ribbon을 통해 Load-Balancing을 수행한다.

- 주어진 서버 목록들을 Round-Robin으로 호출
- Coding을 통해 Load Balancing 방식 Customize 가능

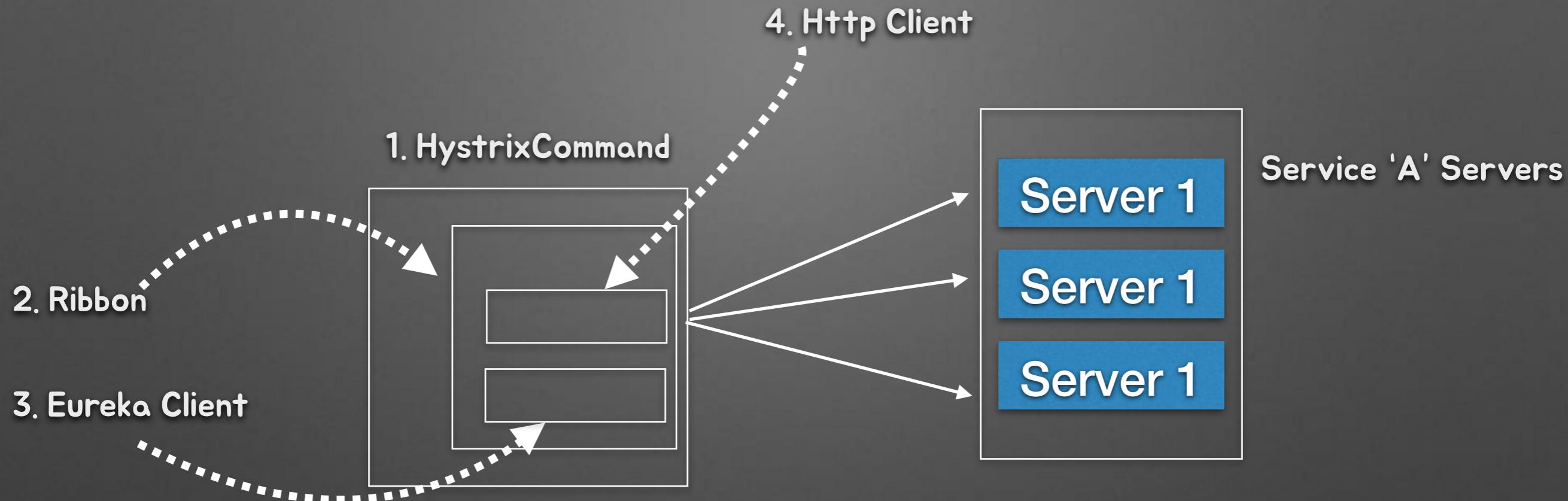
# API Gateway - Zuul



3. Eureka Client를 사용하여 주어진 URL의 호출을 전달할 '서버 리스트'를 찾는다.

- Zuul에는 Eureka Client가 내장
- 각 URL에 Mapping된 서비스 명을 찾아서 Eureka Server를 통해 목록을 조회 한다.
- 조회된 서버 목록을 'Ribbon' 클라이언트에게 전달한다.

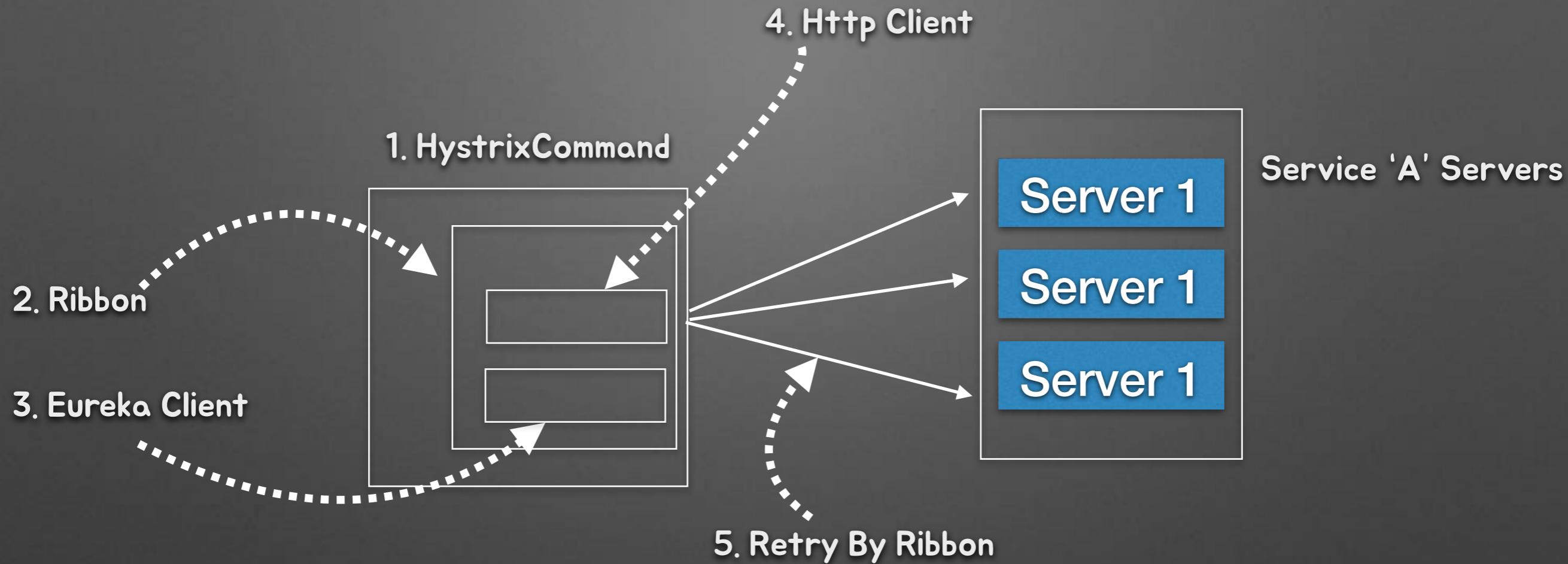
# API Gateway - Zuul



4. Eureka + Ribbon에 의해서 결정된 Server 주소로 HTTP 요청

- Apache Http Client가 기본 사용
- OKHttp Client 사용 가능

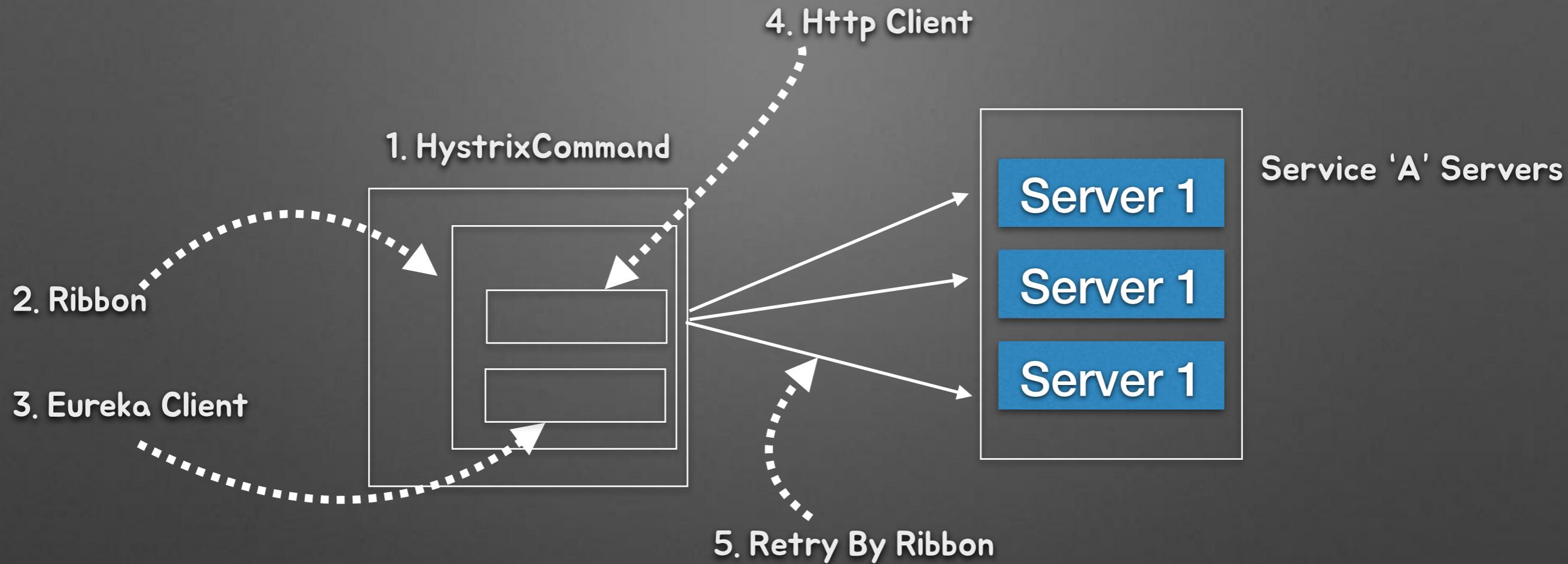
# API Gateway - Zuul



5. 선택된 첫 서버로의 호출이 실패할 경우 Ribbon에 의해서 자동으로 Retry 수행

- Retry 수행 조건
  - Http Client에서 Exception 발생 (IOException)
  - 설정된 HTTP 응답코드 반환 (ex 503)

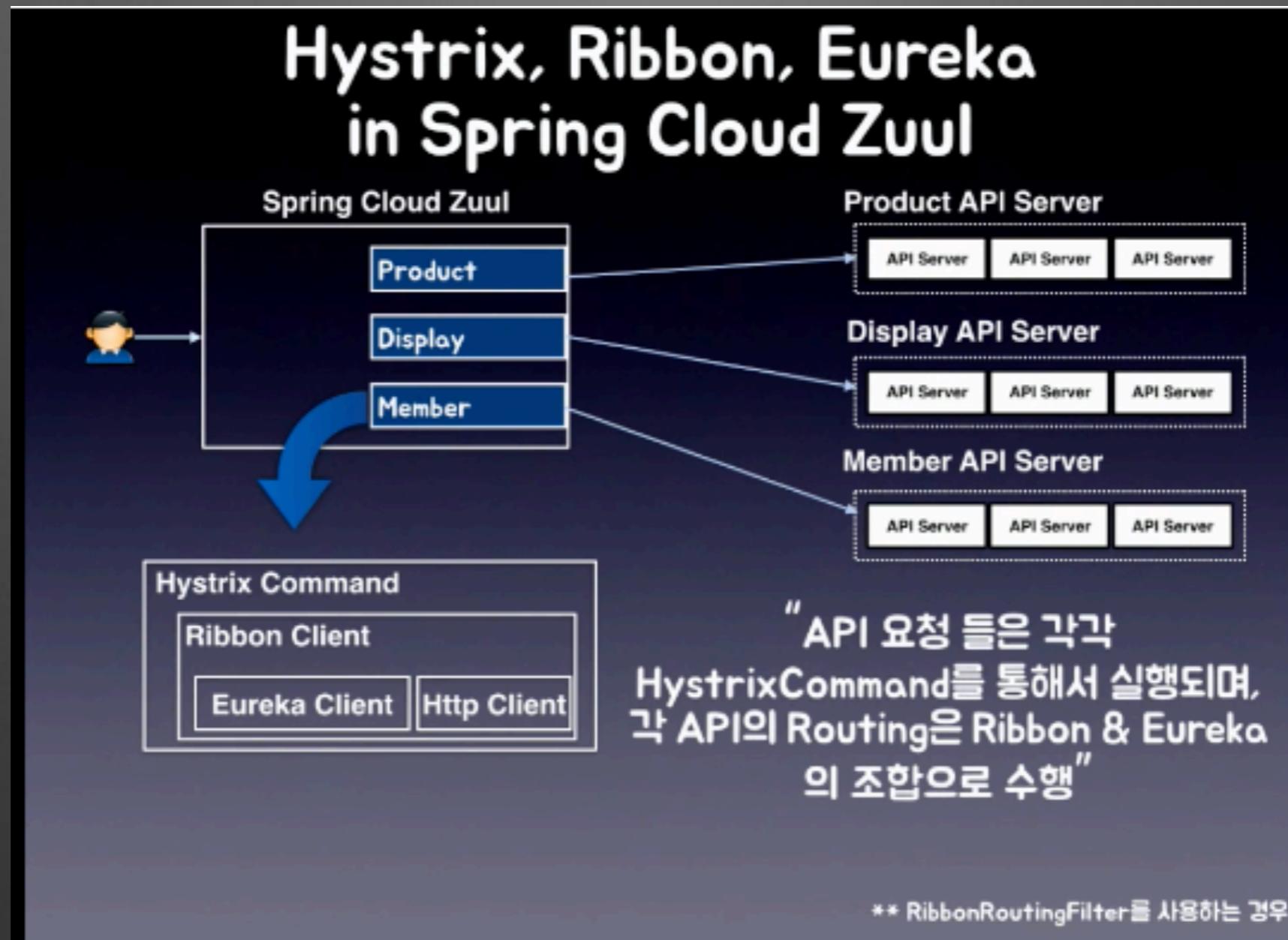
# API Gateway - Zuul



Zuul의 모든 호출은....

- **HystricCommand**로 실행되므로 **Circuit Breaker**를 통해
  - 장애시 **Fail Fast** 및 **Fallback** 수행 가능
- **Ribbon + Eureka** 조합을 통해
  - 현재 서비스가 가능한 서버의 목록을 자동으로 수신
- **Ribbon**의 **Retry** 기능을 통해
  - 동일한 종류의 서버들로의 자동 재시도가 가능

# API Gateway - Zuul



Tag : step-6-zuul-baseline

# [실습 Step-6] Spring Cloud Zuul

## 1. [준비작업] Checkout

시간 절약을 위해 미리 준비된 모듈 Checkout('zuul')

> git reset HEAD --hard

> git checkout tags/step-6-zuul-baseline -b my-step-6

## 2. [zuul] Zuul 과 Eureka 디펜던시 추가 (build.gradle)

```
compile('org.springframework.cloud:spring-cloud-starter-netflix-zuul')
```

```
compile('org.springframework.cloud:spring-cloud-starter-netflix-eureka-client')
```

```
compile('org.springframework.retry:spring-retry:1.2.2.RELEASE')
```

application.yml : Enable Zuul, Eureka

Main Class :

```
@EnableZuulProxy
```

```
@EnableDiscoveryClient
```

Tag : step-6-zuul-proxy-with-eureka

## [실습 Step-6] Spring Cloud Zuul

### 3. [zuul] application.yml 를 다음과 같이 설정

```
spring:  
  application:  
    name: zuul  
  
server:  
  port: 8765  
  
zuul:  
  routes:  
    product:  
      path: /products/**  
      serviceId: product  
      stripPrefix: false  
    display:  
      path: /displays/**  
      serviceId: display  
      stripPrefix: false  
  
eureka:  
  instance:  
    non-secure-port: ${server.port}  
    prefer-ip-address: true  
  client:  
    service-url:  
      defaultZone: http://localhost:8761/eureka/
```

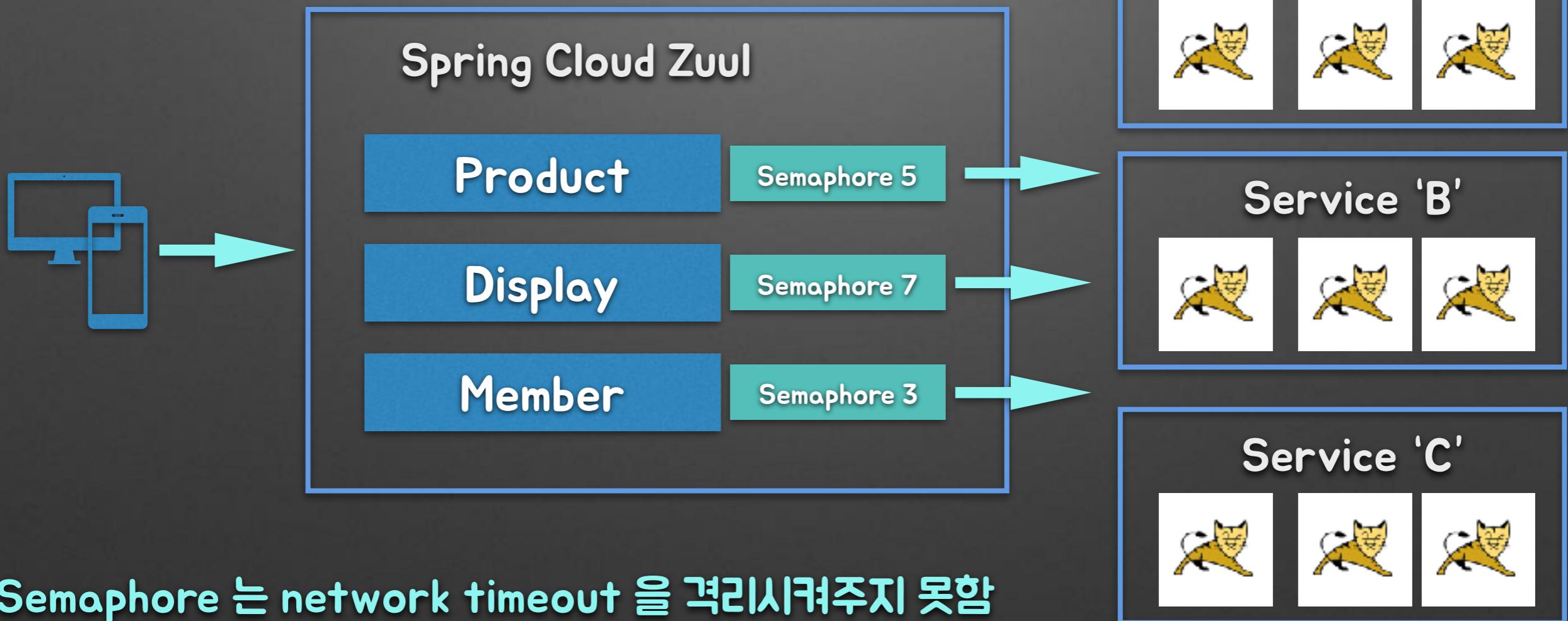
### 4. 테스트

<http://localhost:8765/product/products/11111>

<http://localhost:8765/display/displays/11111>

# Spring Cloud Zuul - Isolation

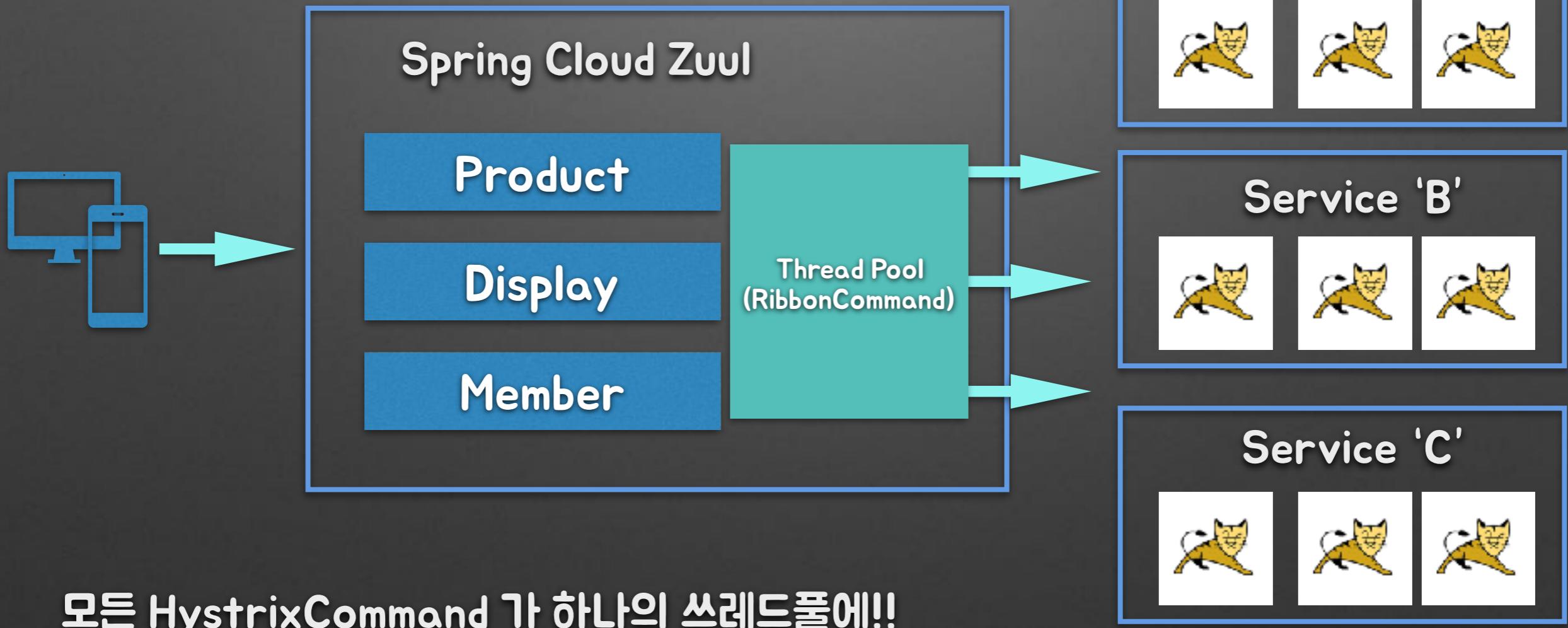
spring-cloud-zuul 의 기본 Isolation 은 SEMAPHORE  
(Netflix Zuul 은 threadpool)



# Spring Cloud Zuul - Isolation

spring-cloud-zuul 의 Isolation 을 THREAD 로 변경

zuul.ribbon-isolation-strategy : thread

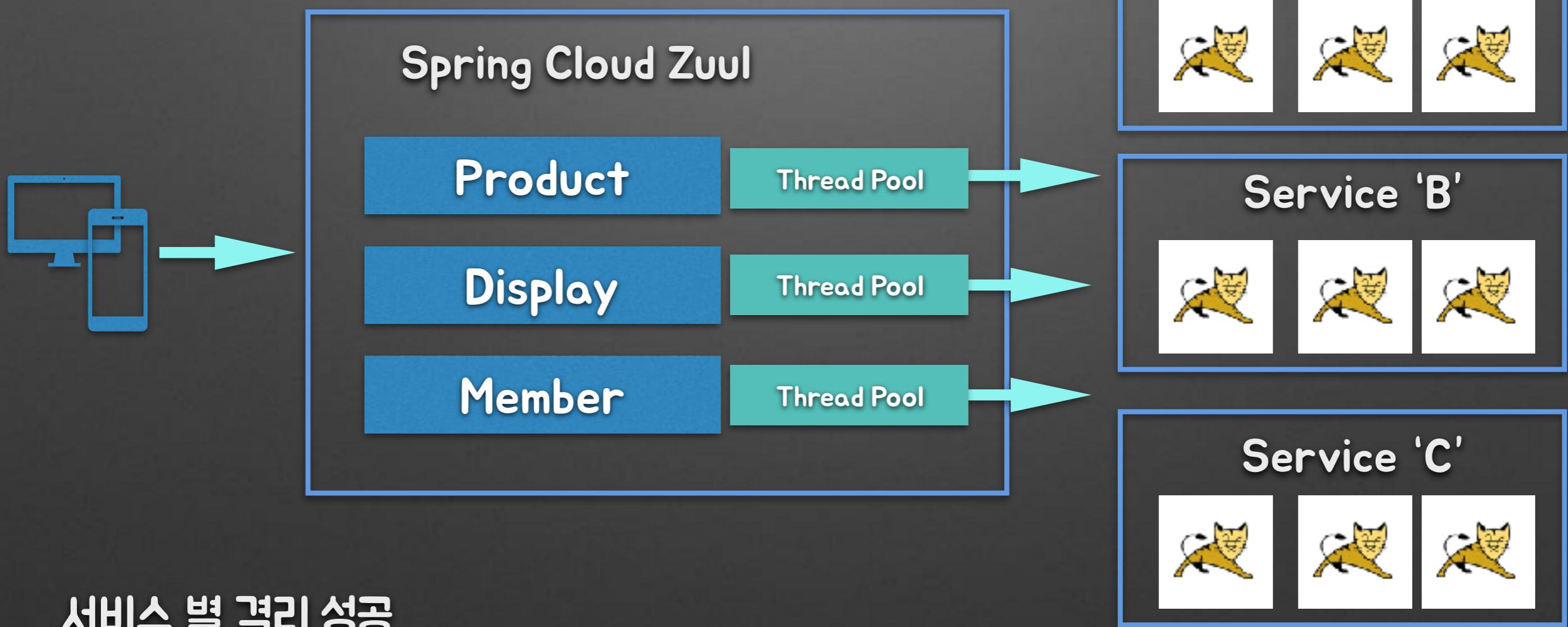


# Spring Cloud Zuul - Isolation

유레카 등록된 서비스 별로 THREAD 생성

zuul.thread-pool.useSeparateThreadPools : true

zuul.thread-pool.threadPoolKeyPrefix : zuulgw



Tag : step-6-zuul-isolation-thread-pool

## [실습 Step-6] Spring Cloud Zuul

### 4. [zuul] application.yml 를 다음과 같이 설정

```
zuul:  
  routes:  
    product:  
      path: /products/**  
      serviceId: product  
      stripPrefix: false  
    display:  
      path: /displays/**  
      serviceId: display  
      stripPrefix: false  
  ribbon-isolation-strategy: thread  
  thread-pool:  
    use-separate-thread-pools: true  
    thread-pool-key-prefix: zuul-
```

```
hystrix:  
  command:  
    default:  
      execution:  
        isolation:  
          thread:  
            timeoutInMilliseconds: 1000  
    product:  
      execution:  
        isolation:  
          thread:  
            timeoutInMilliseconds: 10000  
  threadpool:  
    zuul-product:  
      coreSize: 30  
      maximumSize: 100  
      allowMaximumSizeToDivergeFromCoreSize: true  
    zuul-display:  
      coreSize: 30  
      maximumSize: 100  
      allowMaximumSizeToDivergeFromCoreSize: true
```

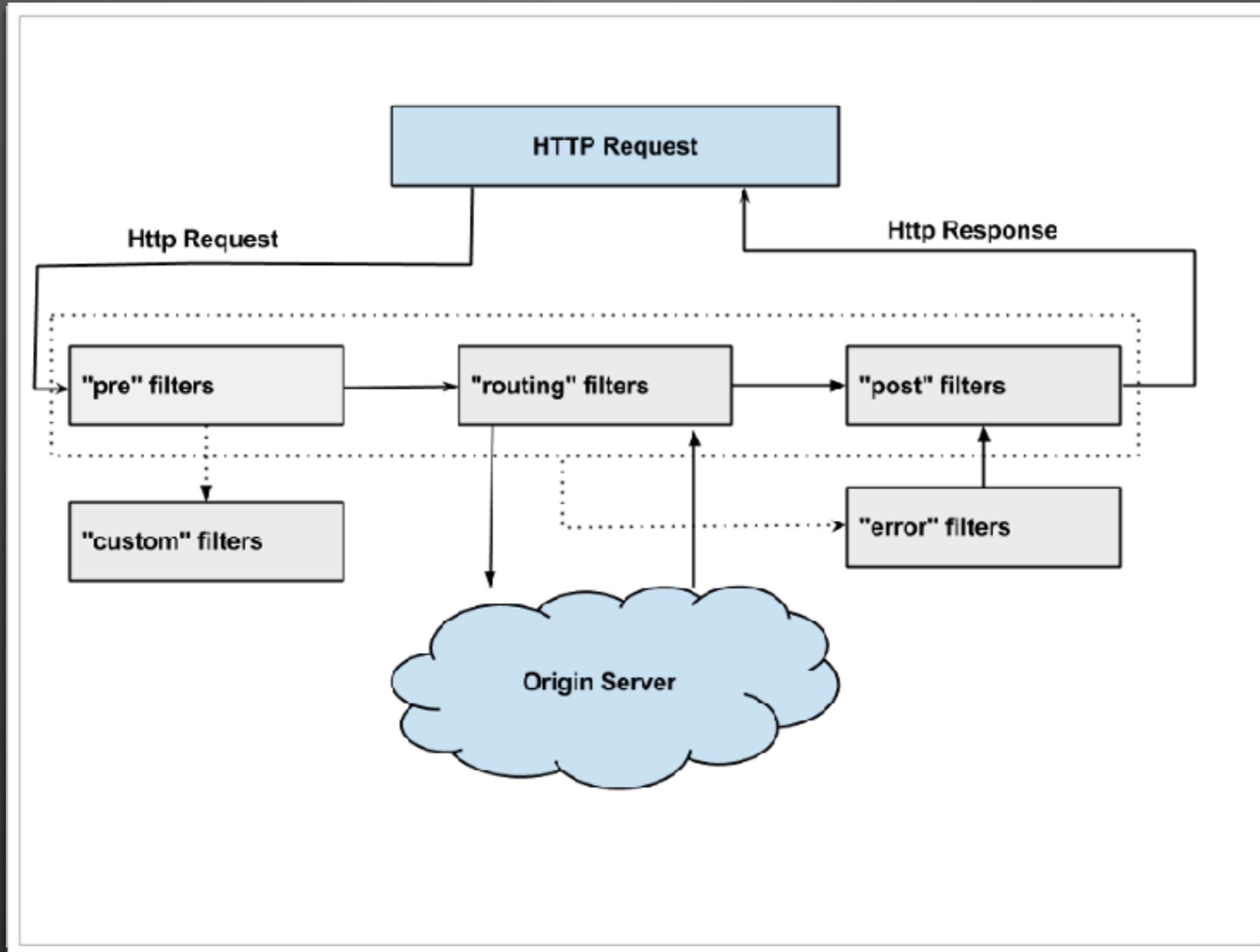
Tag : step-6-zuul-ribbon-config

## [실습 Step-6] Spring Cloud Zuul

### 4. [zuul] application.yml 를 다음과 같이 설정

```
product:  
  ribbon:  
    MaxAutoRetriesNextServer: 1  
    ReadTimeout: 3000  
    ConnectTimeout: 1000  
    MaxTotalConnections: 300  
    MaxConnectionsPerHost: 100  
  
display:  
  ribbon:  
    MaxAutoRetriesNextServer: 1  
    ReadTimeout: 3000  
    ConnectTimeout: 1000  
    MaxTotalConnections: 300  
    MaxConnectionsPerHost: 100
```

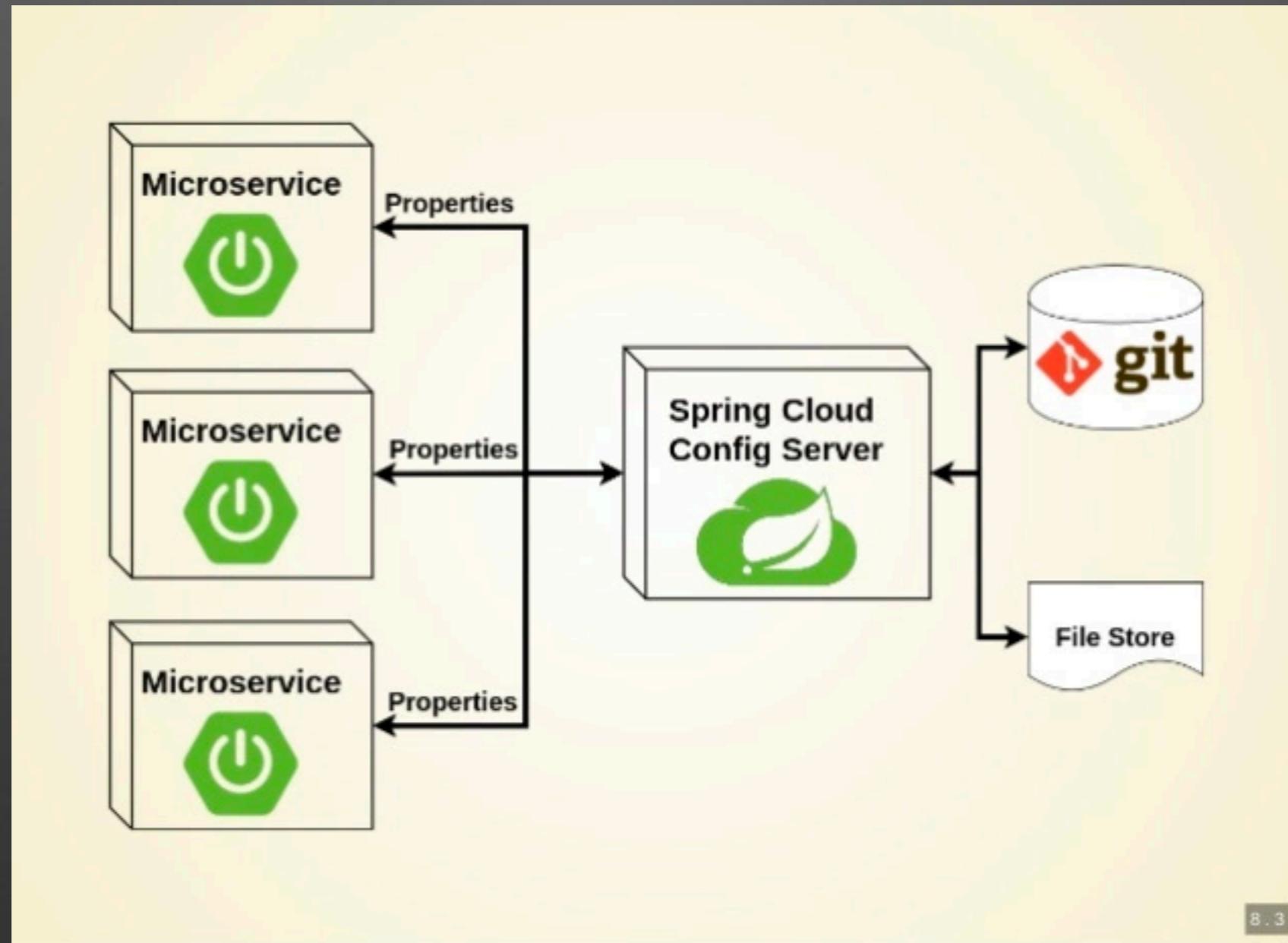
# Spring Cloud Zuul



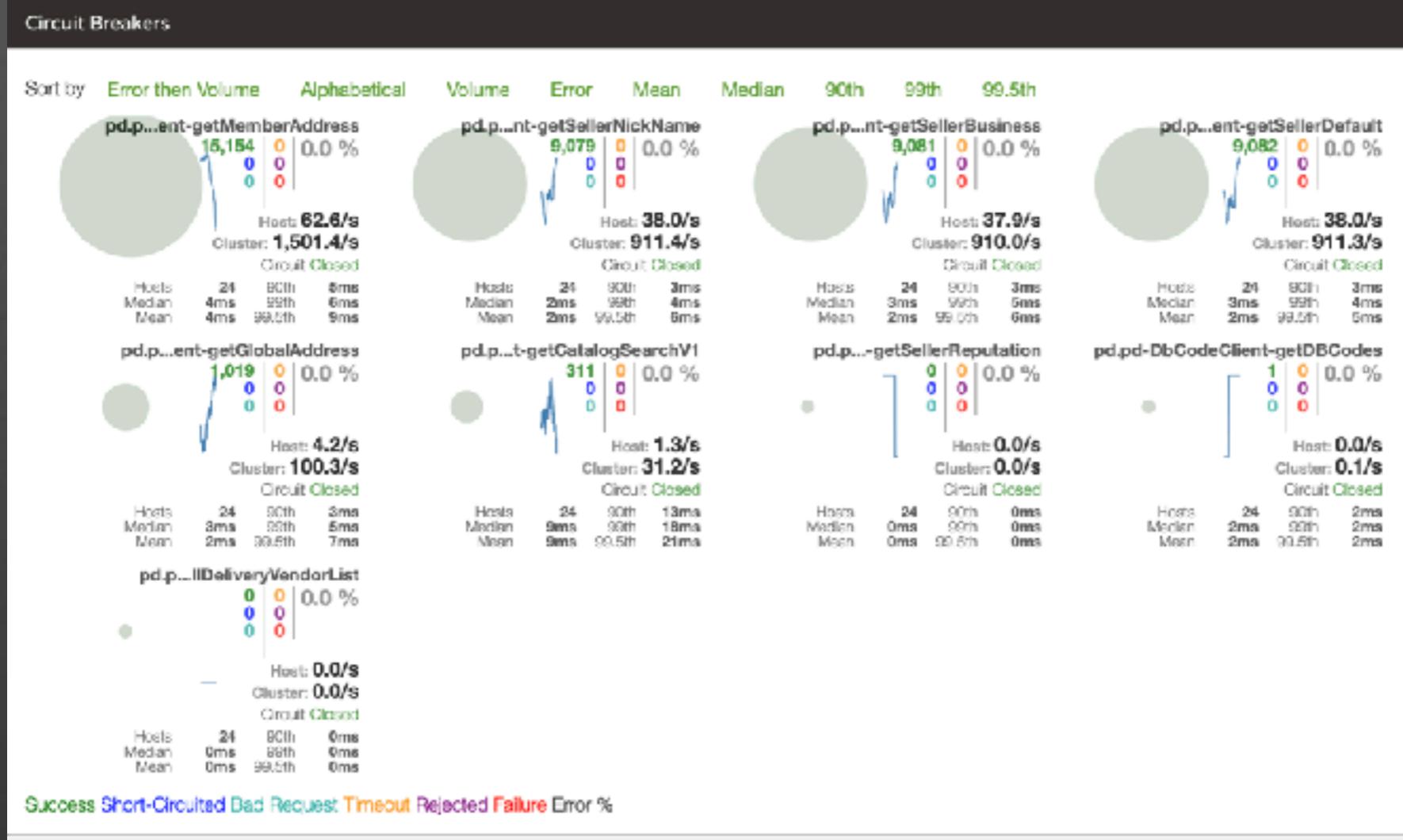
- 시작하기에 앞서
- 모놀리틱 아키텍쳐
- Microservices Architecture(MSA)
- Cloud Native
- Netflix OSS, Spring Cloud
  - **Hystrix - Circuit Breaker**
  - **Eureka - Service Discovery**
  - **Zuul - API Gateway <- 여기 할 차례예요**
- Configuration Server, Tracing, Monitoring, 그리고 남은 이야기..

# Spring Cloud Config

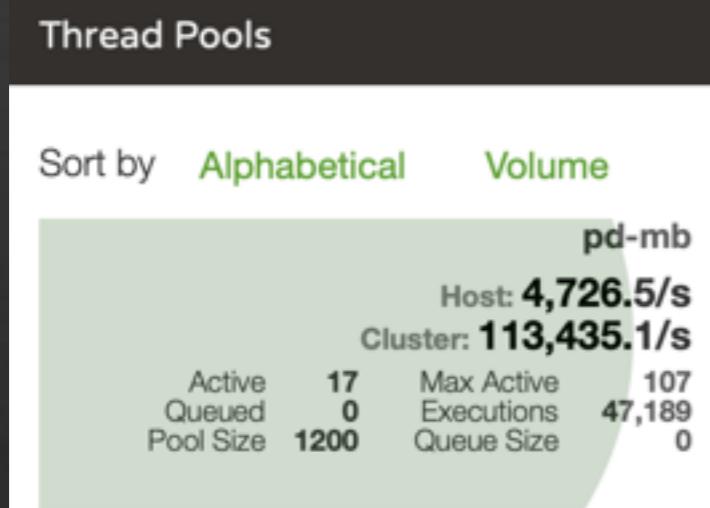
- 중앙화 된 설정 서버



# Spring Boot Admin, Turbine

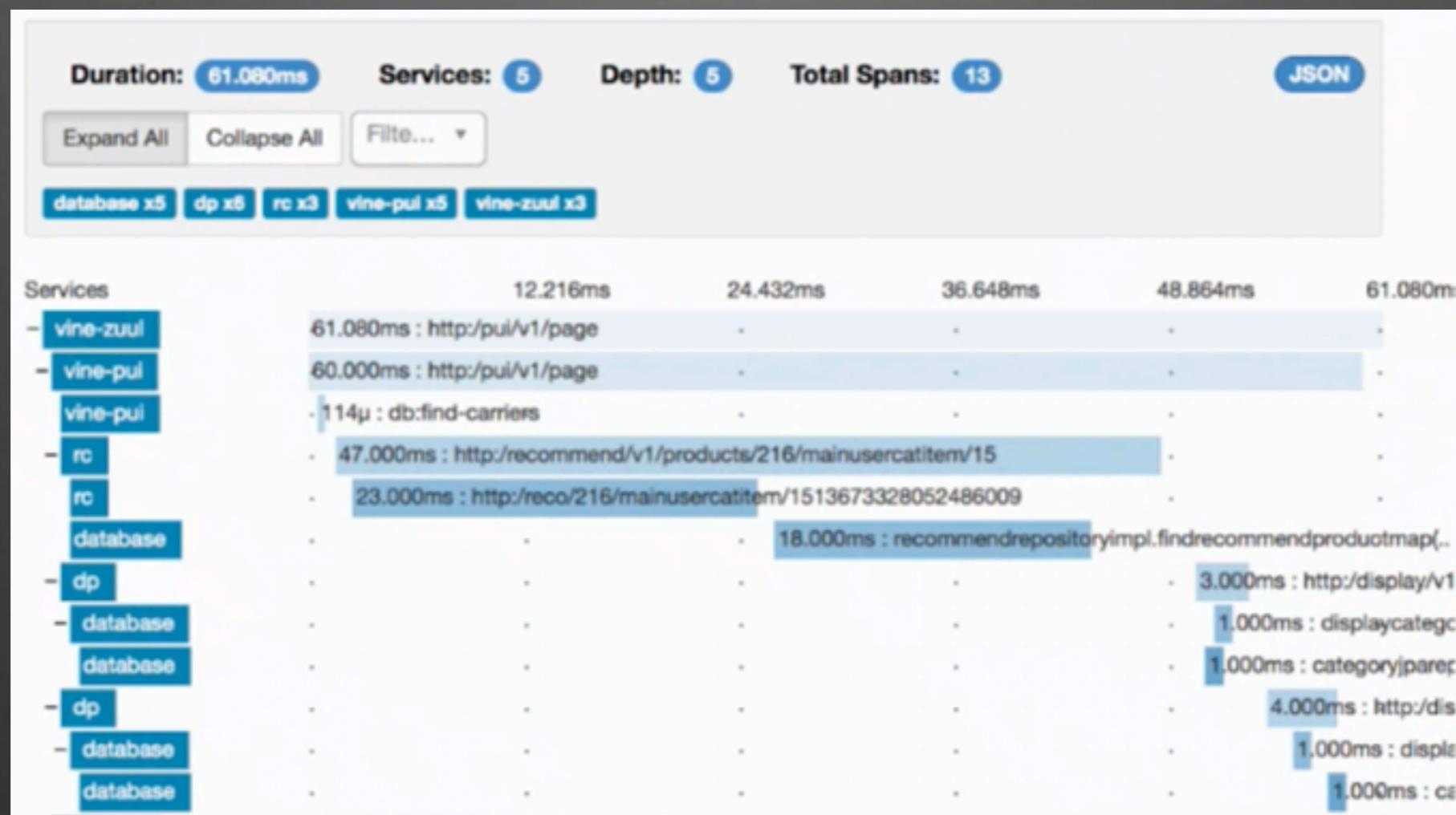


- Success - 성공 횟수
- Short-Circuited - 차단 횟수
- Bad Request - 사용자 입력 오류
- Timeout - 시간 초과
- Rejected - Isolation 초과
- Failure - 일반 오류
- Error % - 전체 에러 비율



# Zipkin, Spring Cloud Sleuth

- Distributed Tracing, Twitter



# 세미나에서 다루지 못한 얘기들.. FAQ

- 코드 리뷰
- TDD
- Database 분리 방법
- 분산 트랜잭션
- Event Driven Architecture
- ...