3일차: 확장하기

Kubernetes for 금융결제원

- [1,2교시] Kubernetes 그리고 인터페이스
  - OCI: Docker? Containerd? CRI-O?
  - CRI: Container Runtime Interface
  - CNI: Cilium? Calico? Flannel?
  - CSI: Persistent Volume?
  - CPI: Loadbalancer?
- [3교시] CI/CD
  - 책임의 분리: CI 와 CD
  - Kubernetes 에서 CD 란?
  - Jenkins, ArgoCD, Helm
- [4,5교시] Logging, Metrics
  - Log 파이프라인 아키텍처
  - OpenMetrics
- [6교시] Recap

DevOps, CI 그리고 CD

DevOps 방법론

- 개발팀과 운영팀이 분리되어있다면 CI 는 몰라도 CD 는 상당히 힘듦
- 무엇보다 코드를 짜는사람이 이것이 어떻게 돌아가는지에 대해서 가장 잘 알고있음
- 그럼, 이 코드를 짜는 사람이 운영도 해주는것이 제일이 아닐까...?
- 개발자가 운영도 해라 => DevOps

DevOps 방법론

- 제시된지는 20년 넘은 개념이지만,
  - 개발하기도 바쁜데
  - Linux / SSH / LB / Syslog / Kerberos / 네트워킹 / 등등등...
  - 무중단 배포 / 모니터링 / DB 운영 / 등등등...
  - 이런것도 알아야한다고...?
- 등등의 이유로 상당히 오랜기간동안 주목받지 못한 개념

DevOps 방법론 ♥ Kubernetes

- DevOps 를 작은규모의 회사에서도 도입하게 된것은 K8s 의 시기와 겹침
- DevOps 를 채택하기에 알아야할것이 너무 많다면?
  - 진짜 알아야할것들만 남기고
  - 나머지는 전부 추상화해버리자
- 2010년 후반에 Kubernetes 는 이런 역할이 가능하다는것을 세상에 보여줌
- 개발자들은 Kubernetes 를 이해하기에 그렇게 어렵지 않음
- 운영자들은 Kubernetes 를 운영하기에 그렇게 어렵지 않음
- Kubernetes 를 매개로 각자는 자신이 잘하는 분야에 더 신경쓴다.

DevOps How?

- DevOps 를 어떻게 해야한다는 사실 회사와 상황마다 사정이 다른부분이 존재
- 아쉽지만, 최적의 DevOps 경험적으로 찾아야함
- 개인적으로 선호하는 방식은
  - 운영자는 K8s 를 유지하는데 초점
  - 개발자는 K8s 를 사용하는데 초점
- 그럼에도 대부분의 경우 공통적으로 신경써주는 부분이
  - CI: 어떻게 코드를 관리(통합)할 것인가
  - CD: 어떻게 코드를 배포할 것인가

Continuous Integration (CI)

- 안정적인 변경사항이 지속적으로 main branch 에 통합되는 개발 방식
- 개발자의 코드 변경사항은
  - 빌드를 수행한 후
  - 유닛 / 통합 테스트를 수행한 후
  - 조직에서 정한 여러 Quality Control (QC) 프로세스를 거친 후
- 지속적으로 모든 개발자가 공유하는 main stream 에 코드를 통합한다.
- 목표: 검증된 코드가 지속적으로 main stream 에 합류하여
  릴리즈 기간을 단축하고, 소프트웨어 품질을 높이기 위해

Before: Continuous Integration (CI)

- 개발자는 특정 기능을 자기 로컬에서 수주 ~ 수개월동안 혼자서 개발하다가
- 나중에 엄청난양의 코드 변경을 main stream 에 통합하려고 하지만
- 그 사이 다른개발자들의 변경사항때문에 이 통합 과정에 상당한 노력이 필요하다.
  - 코드 리뷰
  - 테스트코드 동작 여부
  - 타 기능들과의 충돌 여부
  - 등등등...

After: Continuous Integration (CI)

- 개발자는 자신이 개발하고 있는 기능을 "일" 단위로 main stream 에 통합한다.
- 코드의 변경사항은 조금씩 지속적으로 main stream 에 통합되고
- 기능 개발이 완전히 끝났을때는 별도의 통합노력이 필요하지 않다.

CI - 자동화를 전제로

- CI 에 자동화는 필수는 아니지만, 없으면 안하는게 사람이다.
- CI 과정은 개발중인 프로그램의 성격에 따라 접근방식이 많이 달라진다.
  - 많은 트래픽을 감당하는것이 중요한가? => 부하 테스트 병행
  - 빠른 응답속도가 중요한가? => 벤치마크 테스트 병행
  - 정확한 처리가 중요한가? => 수많은 unit, intergration, e2e 테스트들
  - 어느정도 장애상황을 허용하는가? => 생산성에 초점
- 하지만 Kubernetes 위에서 동작을 할것이라면
  - CI의 최종 산출물은 Container Image 를 포함하는것이 좋다.
  - (API / 기술 문서, 릴리즈노트, 실행파일등도 포함될 수 있다.)

CI - 제품들

- Github Actions
- Gitlab Runners
- Travis CI
- Circle CI
- Drone CI
- Spinaker
- Tekton
- Jenkins

CI - 제품들

- 다양한 종류들이 존재하지만,
- 전부 Container Image 를 생성하는 기능들은 필수로 추가되어있다.
- 개발 성격과 환경에 맞춰서 골라서 사용하면 된다.

Continuous delivery (CD)

- CI 가 검증된 코드로부터 Container Image 를 만들어준다면
- CD 는 위 코드를 실제 테스트해볼 수 있는 Staging 환경으로 배포도 자동화
- 아무리 테스트코드를 돌리고 QC 과정을 패스했다고 해도 결국 버그는 나는법
- 하지만 이 코드를 매번 실행하는데
  - SSH 로 접속해서
  - 이전에 실행했던 프로세스를 종료하고
  - 새로 빌드된 파일을 복사한다음에
  - 등등등...
- 결국 자주 안함 (하루에 수번 ~ 수십번은 이런일을 해야하므로)

CD

- CI 가 Container Image 를 어떻게 만들것인가에 대해서 집중했다면
- CD 는 주어진 Container Image 를 어떻게 배포할 것인가에 대해서 집중하면 된다.
- 흔히들 하는 실수
  - CI 와 CD 를 하나의 파이프라인에 정의하려고 한다.

CD

- 흔히들 하는 실수
  - CI 와 CD 를 하나의 파이프라인에 정의하려고 한다.
- 만약
  - 이전 특정 버전을 다시 테스트 해보고 싶다면?
  - DB Migration 과 같이 비가역적인 영구적 변화가 동반된다면?
  - Stage / Prod 등 두개 이상의 배포환경이 존재한다면?

CD

- 정답은 없지만, CI / CD 를 하나의 파이프라인으로 구축하는것은 대개 고통을 초래함
- CI 파이프라인이 CD 파이프라인을 Trigger 할수는 있을지언정 별개의 프로세스로 두는것이 대체로 덜 고통스러움

파이프라인 예시

- [CI] 유닛테스트
- [CI] 통합테스트
- [CI] E2E 테스트
- [CI] Docker build
- [CI] 코드리뷰 (n명 이상의 approve)
- [코드 병합]

파이프라인 예시

- [Stage CD] 배포계획 시각화
- [Stage CD] 관리자 승인 대기 (보통 Auto Approval)
- [Stage CD] Stage K8s 클러스터에 배포
- [QA 테스트]
- [신규 릴리즈]
- [Prod CD] 배포계획 시각화
- [Prod CD] 관리자 승인 대기
- [Prod CD] Prod K8s 클러스터에 배포

CD - 제품들

- CI 는 상당히 다양한 제품이 있는 반면
- CD 는 제품 다양성이 크게 떨어짐
- 크게 두가지 방식
  1. `ArgoCD` 방식
     - Git 에 정의된 K8s 리소스를 Cluster 와 동기화
     - Github 과 통합하며 GitOps 방식에 어울리는 배포가 가능
  2. `이벤트` 방식 (임의 명명)
     - 기존 CI 툴에 CD 의 기능을 구현한것으로 별도의 시스템 구축 필요 X
     - 기본적으로 `kubectl apply -f` 명령어를 대신 입력

CD - 보조 도구들

- Helm
  - 오픈소스를 다양한 환경에 일부 설정만 변경한채로 배포하는데 많이 사용
  - 같은 프로그램을 서울, 일본, 미국 등 다양한 리전에 배포하는 시나리오에 상당한 이점을 가지는 yaml 템플릿팅 툴
- Jsonnet
  - Helm 보다 더 복잡한 구성을 가지고 더 많은 재사용성이 요구될때 많이 사용
  - 주로 인프라 레벨에서 많이 사용
- Kustomize
  - Helm 보다 간단하며 Stage, Prod 등의 환경만 운영할경우 적절

CD - 보조 도구들

- 결국 목표는 Kubernetes 에 입력할 manifest 파일을 어떻게 편하게 정의할까에 대한 고민들
- Kustomize 로 시작해서 한계를 느낄때쯤 helm 으로 변경하는 시나리오가 대체로 무난함
- 사용환경에 따라서 다르겠지만,
  대게 보안적이유로 Kubernetes 리소스에 라벨을 붙여서 자동화된 관리를 하는 경우가 많음
- 이러한 템플릿팅 도구들은 이러한 라벨을 부착하고 관리하는 업무를 상당히 편하게 해줌

Q&A

Observability

애플리케이션을 모니터링하는 방법은?

Observability

애플리케이션을 모니터링하는 방법은?

1. Logging
2. Metric
3. Tracing
4. Continuous Profiling

Observability

애플리케이션을 모니터링하는 방법은?
(위에서 아래로 순차적으로 가게됩니다.)

1. Logging
2. Metric
3. Tracing
4. Continuous Profiling

Observability

애플리케이션을 모니터링하는 방법은?
(위에서 아래로 순차적으로 가게됩니다.) ~~(가고싶지 않아도)~~

1. Logging
2. Metric
3. Tracing
4. Continuous Profiling

Observability

애플리케이션을 모니터링하는 방법은?
(위에서 아래로 순차적으로 가게됩니다.) ~~(가고싶지 않아도)~~

1. Logging
2. Metric
3. ~~Tracing~~
4. ~~Continuous Profiling~~

Logging

Logging

정의: 애플리케이션의 STDOUT 혹은 STDERR 로 출력된 문자열
고민: 수십 개의 Pod에서 나온 로그를 어떻게 보면 좋을까?

Logging: 답변

답변 - 1 : Kubernetes API 를 잘 사용하자.

```
# 랜덤으로 Pod 하나 골라서 로그 보기
kubectl logs deployments/blue-app

# 라벨 일치하는거 전부 보기
kubectl logs -l app=blue

# 최근 로그만 보기
kubectl logs some-pod --since 1s -f

# 왜 죽었는지 보기
kubectl logs some-pod --previous
```

Logging: 답변

답변 - 2 : Kubernetes API 를 잘 사용하는 툴을 쓰자

```
stern somepod
stern somepod -o raw | jq
```

한계점

부하 큼. 실시간으로 밖에 못 봄.

```
client -http-> kube-apiserver -http-> kubelet -> file
```

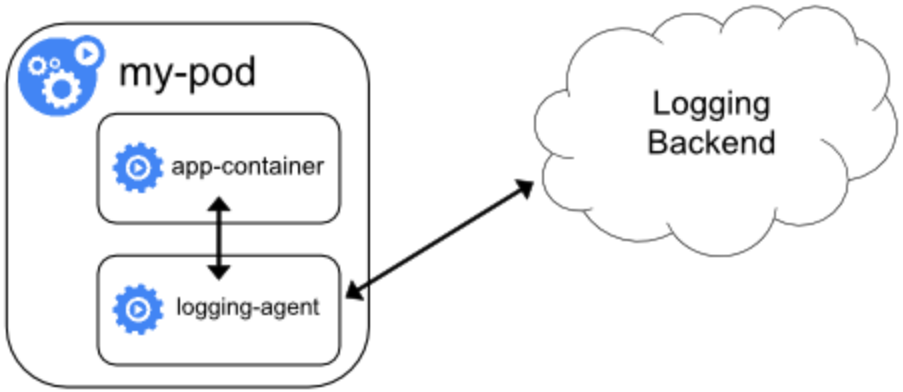
Cluster-Level Logging

- 클러스터 관점으로 로그를 로그 저장소에 저장

Cluster-Level Logging

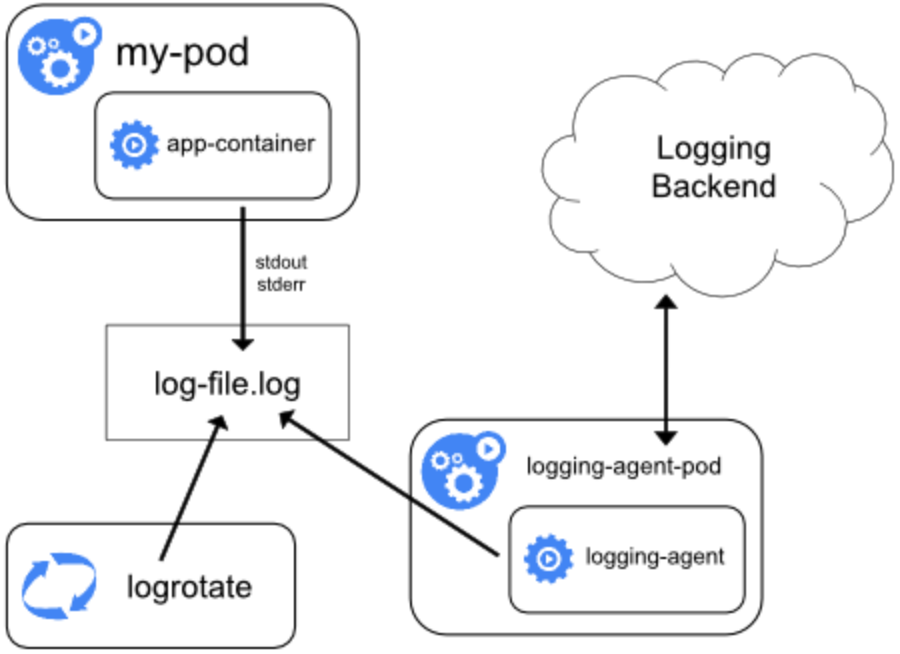
- 클러스터 관점으로 로그를 로그 저장소에 저장
- 두 가지 접근법
  - Side Car
  - Node Agent

Cluster-Level Logging - Sidecar



- 장점: 특이한 요구사항 맞추기 좋다.
- 단점: 그 이외의 모든 것
  [설정이 귀찮다. {예외 상황에 대한 고민을 해줘야 한다. (강제종료시, 메모리관리, 디스크 관리 등등)}]

Cluster-Level Logging - Node Agent

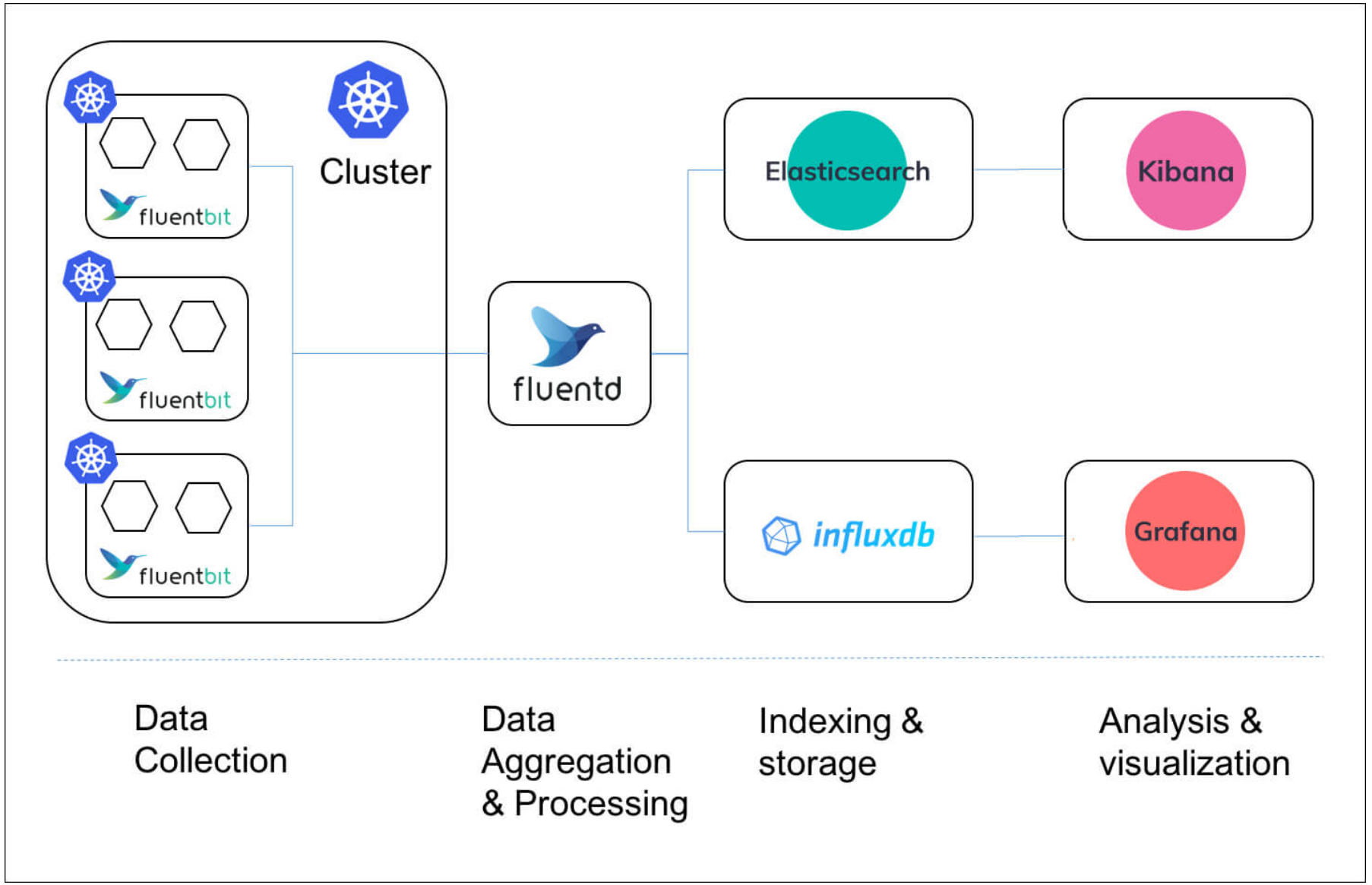


- 장점: 설정이 편하다.
- 단점: 특이 케이스 핸들링이 좀 까다롭다.

Cluster-Level Logging - 원리

- kubelet 이 CRI 를 통해서 컨테이너를 실행할때 로그 경로를 미리 지정한다.
- 이 경로는 `/var/log/pod` 로 이 값은 하드코딩 되어있다.
- `fluentd`, `filebeat`, `vector`, `promtail` 등의 로그 수집기들은 이 경로상에 존재하는 로그를 긁어간다.
- `Elasticsearch`, `Loki`, `Datadog` 등의 방법으로 바로 인덱싱하던가
- or `Kafka`, `fluentd` 등의 버퍼를 두고 주기성을 가지고 인덱싱한다.
- 부르는 명칭은 다양하지만 보통 다음과 같이 아키텍처를 구현

ref
plus


Cluster
fluentbit
fluentbit
fluentbit
fluentd
Elasticsearch
Kibana
influxdb
Grafana
Data Collection
Data Aggregation & Processing
Indexing & storage
Analysis & visualization

Cluster-Level Logging - 유명한 것

1. Elasticsearch w/ fluentd or filebeat

- 전통적인 강자 / 빠르고 무거움 / 사용하기 편함 / 운영하기는 전문적인 지식이 필요하고 까다로움

2. Splunk

- 무겁고 / 어렵고 / 비쌈

3. Loki

- 신흥 강자 / 제약사항이 있지만 가벼움 / S3 를 스토리지로 활용 / Kubernetes 친화적

Cluster-Level Logging - 유명한 것

4. Datadog

- 운용 인력이 없으면 돈으로 해결 가능한 모니터링 솔루션 (상당히 비쌈)

5. New Relic

- 운용 인력이 없으면 돈으로 해결 가능한 모니터링 솔루션 (Datadog 보다 상당히 저렴)

6. Dynatrace

- ??

Metric

Metric

- 로그가 이벤트성 데이터라면
- 메트릭은 어느 시점에 현재 상태를 표시하는 지표

Metric

- 로그는 이미 발생한 일을 수습하는 것이 목표라면
- 메트릭은 이것에 더해서 문제가 터지기 전에 예방하는 목적이 강함
- 메트릭을 시각화할 때는 이상 조건이 무엇인지를 항상 명심해야 함.

  e.g.,
  - conntrack 이 30만이니깐 TCP 커넥션 개수가 30만을 넘기지 않는지 모니터링 (임계값 존재)
  - CPU 사용량이 평소에는 50% 수준인데 갑자기 90%를 찍지는 않는지 모니터링 (평소와 다른 패턴)
  - 특정 Pod 만 과도한 메모리 혹은 CPU 를 사용하고 있지 않는지 모니터링 (Abnormal 탐지)

Metrics

여러 프로젝트들이 존재하지만 Prometheus 가 제일 기본
Prometheus로 시작해서 부족하면 다른 것으로 옮겨가는 것이 좋음

1. 고가용성이 필요 > link
2. 운영부담 줄이기 > Datadog, Elasticsearch

Metric

보통은 Prometheus 를 기반으로 만들어진 OpenMetrics 포맷에 맞춰서 특정 http 기반 endpoint 를 통해서 수집

https://github.com/prometheus/docs/blob/main/content/docs/instrumenting/exposition_formats.md#text-based-format

```
metric_name [
  "{" label_name "=" `"` label_value `"` { "," label_name "=" `"` label_value `"` } [ "," ] "}"
] value [ timestamp ]
```

http://127.0.0.1:8001/metrics

Metric

흔히 다음으로부터 메트릭을 가져옴

1. kubelet
2. kube-apiserver
3. kube-state-metrics
4. node-exporter
5. etcd

다만 처음부터 모든 메트릭을 전부 보려고 노력하는 것은 리소스 낭비
어떤 게 존재하는지만 알아두었다가 실제 문제가 발생하는 것만 시각화하는 것이 좋음

다음은 node-exporter에서 추출되는 메트릭 중 볼만한 것들

DEMO

쿼리를 생성하는법

DEMO

메트릭 리뷰하는 법

더 자세하게는

- Kubernetes 만큼이나 별도의 공부가 필요

Q&A

Recap