

Dendrobium

# K8S를 사용한 Web Service 배포

김명현, 박 출, 박태우, 이민수, 이원우

# CONTENT

---

01  
목표

02  
프로젝트 계획

03  
팀원 역할

04  
개발 환경

05  
구성도

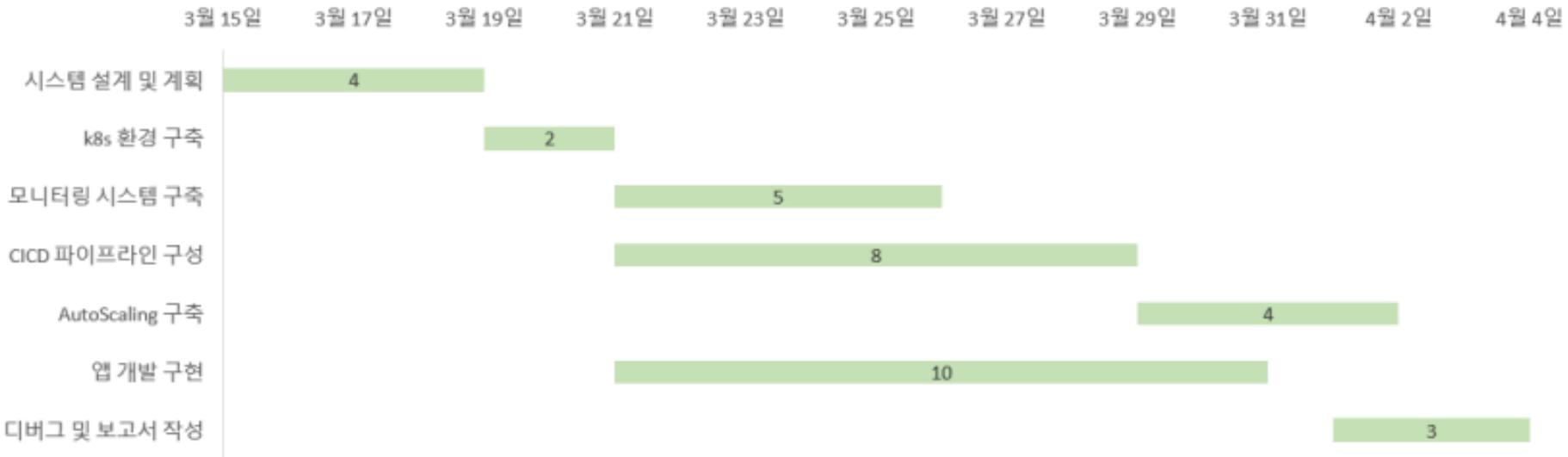
06  
상세 내용

07  
향후 계획

## 목표 하는 바

K8S를 이용해 제공되는 환경에서 제공되는  
클레이저스터드 기반 환경에서 CICD 구현으로  
제작된 배포 및 배포 관리하고 한다.

## 02 프로젝트 계획



INTRO

# 03 역할 분담

이원우 (팀장)

일정 수립 및 역할 분당  
어려움을 겪는 팀원 지원  
보고서 작성

김명현

K8S Cluster 구성  
Dockerfile 작성

박태우

서비스 구현  
back-end 담당

박 솔

모니터링  
(Grafana/Prometheus)

이민수

CI/CD 파이프라인  
(Jenkins/ArgoCD)

## 04 개발 환경

---

- Infra:
  - on-premise Kubernetes
  - CI service: Jenkins
  - CD Service: ArgoCD
  - Git Service: github
  - Monitoring Service: Grafana, Prometheus
  - Docker repo: Docker-Hub
- Server Application:
  - Java-Spring
  - Spring-Boot
- DataBase: Mysql

# 04 기술 스택



## Metallb

베어메탈로 구성된 쿠버네티스 환경에서도  
로드밸런서를 사용할 수 있게 해준다



## Git

github를 이용한 형상관리



## Jenkins & ArgoCD

지속적 통합(CI)과 지속적 배포(CD)를 위한 도구 조합



MySQL

## Java Spring & MySQL

웹 서비스 구현을 위한 도구 조합



+



## Prometheus & Grafana

모니터링 및 시각화 도구, 시스템 상태와 성능을 모니터링하고 시각화함

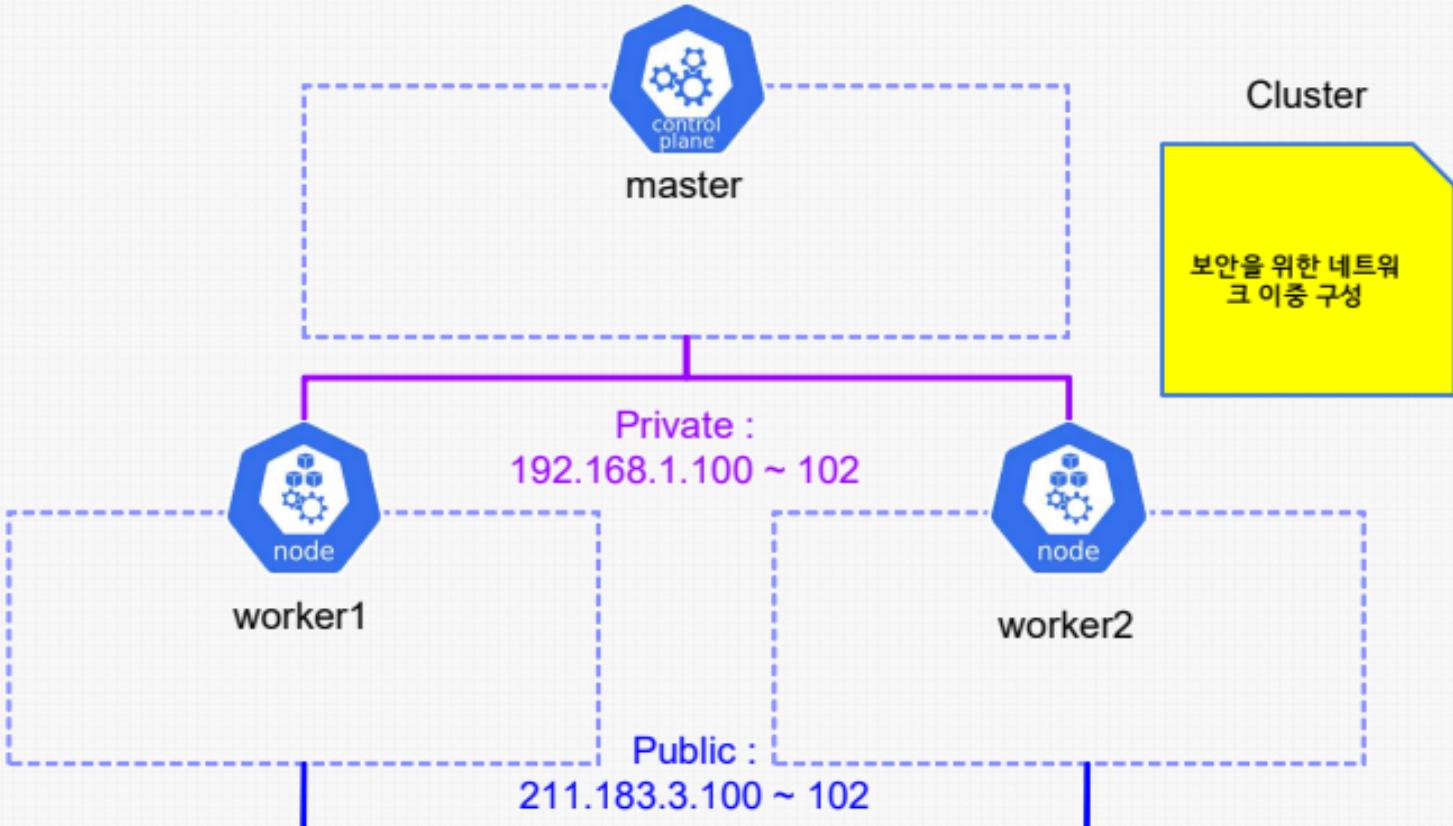
# 05 플랫폼 구성 및 환경

## 컴퓨팅 환경 및 세팅

	master	worker1	worker2
CPU/Memory	4/4	4/4	4/4
Storage	40	20	20
OS	Ubuntu	Ubuntu	Ubuntu
IP	211.183.3.100	211.183.3.101	211.183.3.102
ETC	Grafana + Prometheus Jenkins Argocd HPA	Web svc/db	Web svc/db

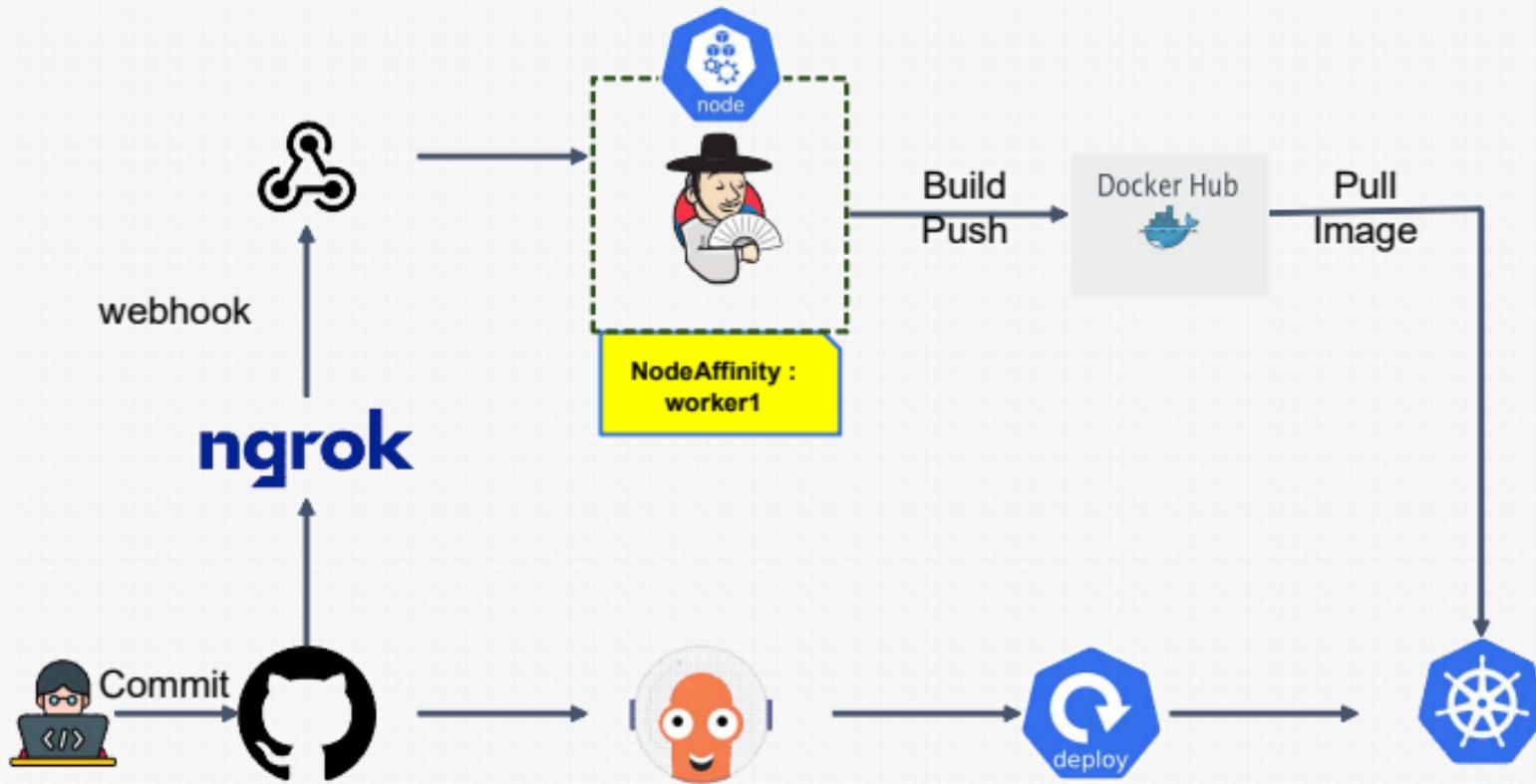
DG : 211.183.3.2

## 05 플랫폼 구성 및 환경

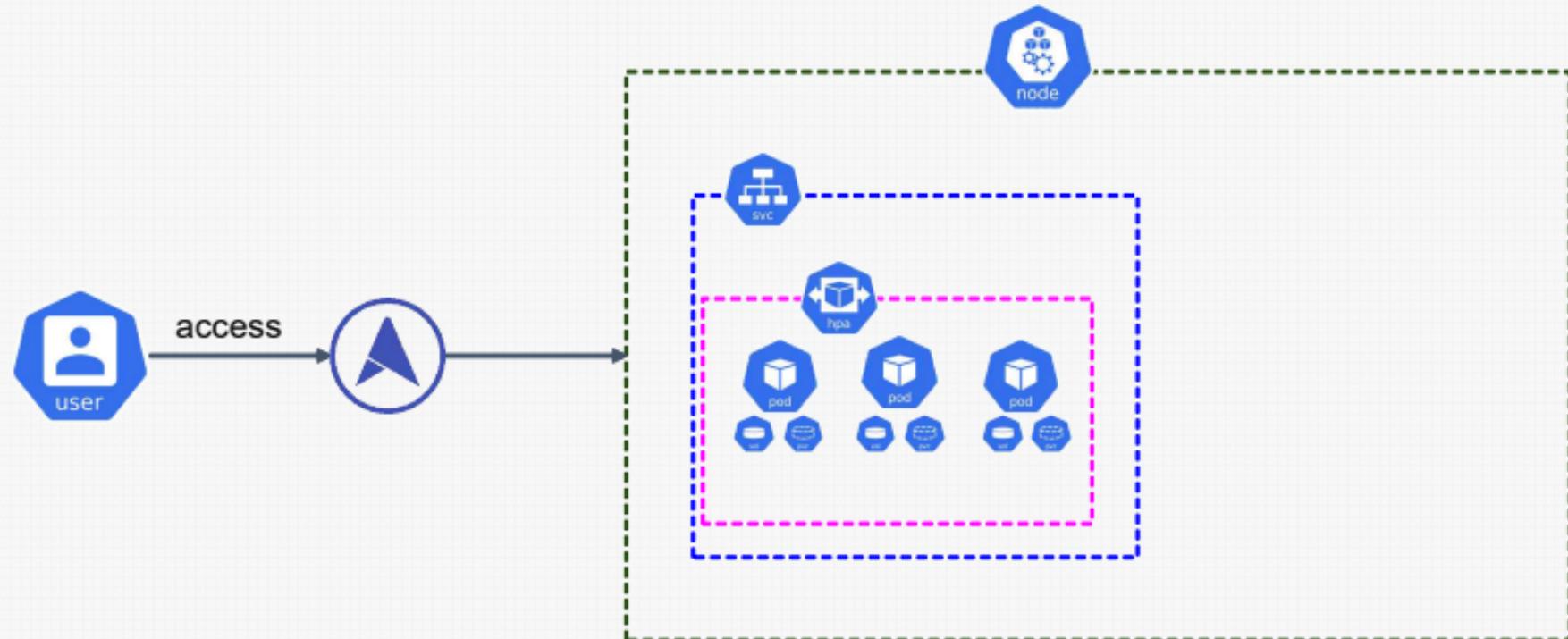


## 05 플랫폼 구성 및 환경

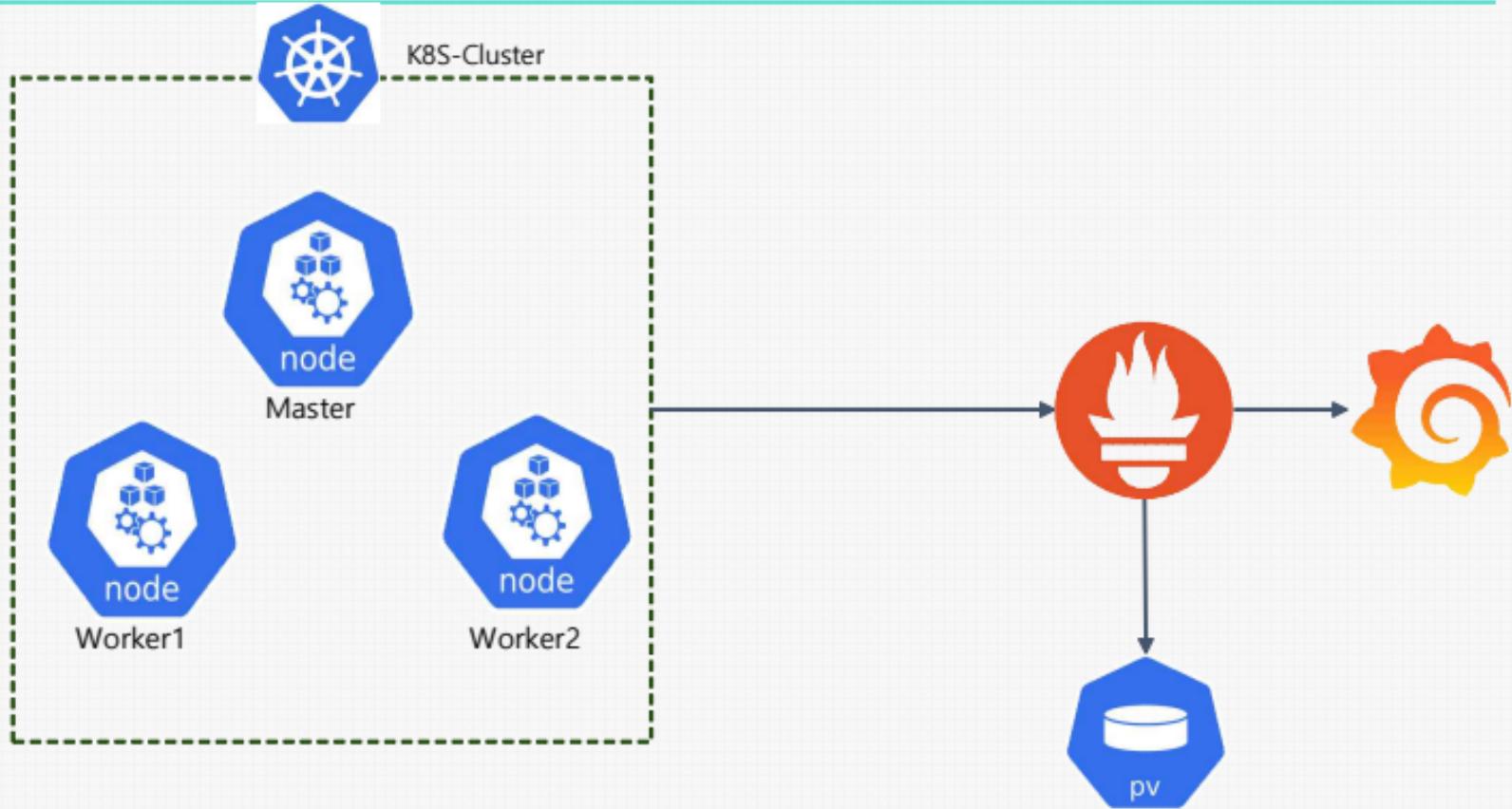
### CICD 플로우차트



## 05 플랫폼 구성 및 환경



## 05 플랫폼 구성 및 환경



# 06 상세 및 특징

Monitoring - prometheus

## 메모리 사용량에 대한 알림

### container CPU alert

#### Rule

```
alert: container CPU usage rate is very high( > 10%)
expr: sum(rate(container_cpu_usage_seconds_total[pod!=""])[1m]) / sum(machine_cpu_cores) * 100 > 10
for: 1m
labels:
  severity: fatal
annotations:
  summary: High Cpu Usage
```

## CPU 사용량에 대한 알림

### container memory alert

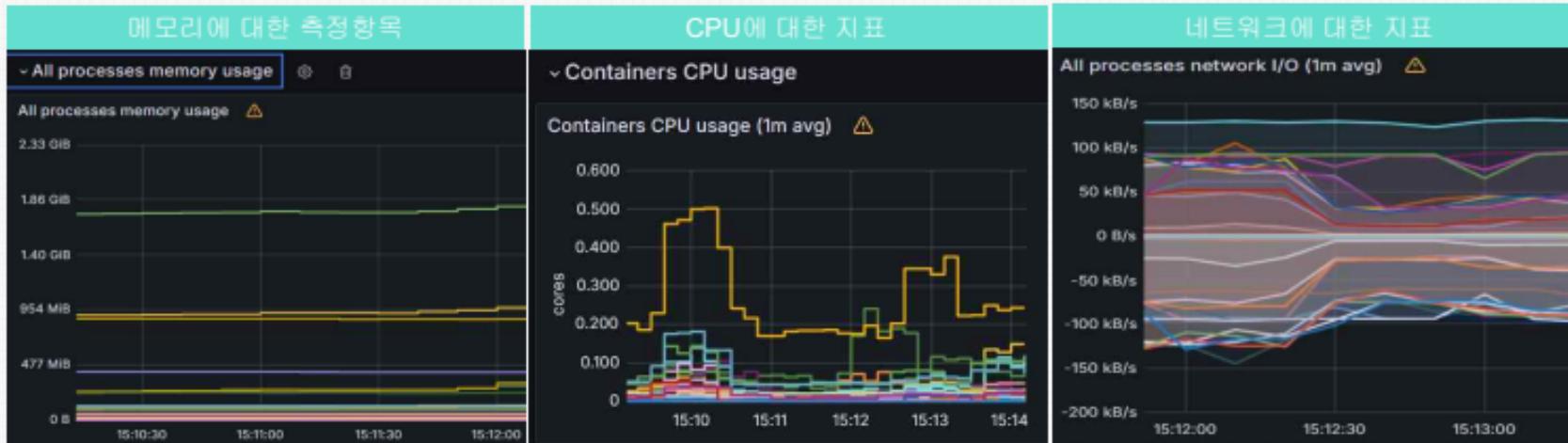
#### Rule

```
alert: container memory usage rate is very high( > 55%)
expr: sum(container_memory_working_set_bytes{name=""+pod+""}) / sum(kube_node_status_allocatable_memory_bytes) * 100 > 55
for: 1m
labels:
  severity: fatal
annotations:
  description: Memory Usage:
  identifier:
  summary: High Memory Usage on
```

경고 규칙을 사용하면 Prometheus 표현 언어 표현을 기반으로  
경고 조건을 정의하고 외부 서비스에 경고 실행에 대한 알림을 보낼 수 있습니다.

# 06 상세 및 특징

Monitoring - grafana



전체 메모리 사용량 확인

서비스의 평균적인  
CPU 사용량을 확인

서비스들의 네트워크  
통신량 확인

# 06 상세 및 특징

Monitoring - 장애 대응

## 장애 발생 원인

사용자의 증가로 인하여  
트래픽의 증가로 이어지고  
서비스의 품질이 떨어지며  
전체적인 서버의 하향이 발생 가능.

## 장애 발생 대비

- 서비스의 평균적인 CPU 사용량을 확인하여  
장애 발생 상황에 대비
- 미리 서비스를 제공할 파드의 개수를 늘려  
문제 상황의 발생을 예방

## 장애 발생 시 대처

원인 파악 또한 중요하지만,  
서비스의 정상화를 위하여  
롤백을 우선적으로 시행하고  
장애 보고서를 제작하여  
이후에 동일한 문제를 방지.

# 06 상세 및 특징

Web.yaml에 Auto-Scaling 적용

Web Server Service	Horizontal Pod Autoscaling(HPA)	Stress Code
<pre> apiVersion: apps/v1 kind: Deployment metadata:   name: my-nginx-deployment3 spec:   replicas: 1   selector:     matchLabels:       app: my-nginx-app3   template:     metadata:       labels:         app: my-nginx-app3     spec:       containers:         - name: my-nginx-container           image: rlaasudus/my-test3:52           ports:             - containerPort: 80       resources:         requests:           cpu: 200m # CPU 요청을 200m (0.2 CPU)로 줄가시킵니다.           memory: 256Mi # 예모리 요청을 256Mi로 증가시킵니다.         limits:           cpu: 500m # CPU 저정을 500m (0.5 CPU)로 즐가시킵니다.           memory: 512Mi # 예모리 저정을 512Mi로 즐가시킵니다. </pre>	<pre> apiVersion: autoscaling/v1 kind: HorizontalPodAutoscaler metadata:   name: my-nginx-hpa spec:   scaleTargetRef:     apiVersion: apps/v1     kind: Deployment     name: my-nginx-deployment3   minReplicas: 1   maxReplicas: 5   targetCPUUtilizationPercentage: 20 </pre>	<pre> #!/bin/bash ab -c 1000 -n 200 -t 60 http://211.183.3.204:80/ </pre>

web.yaml

web.yaml

test.sh

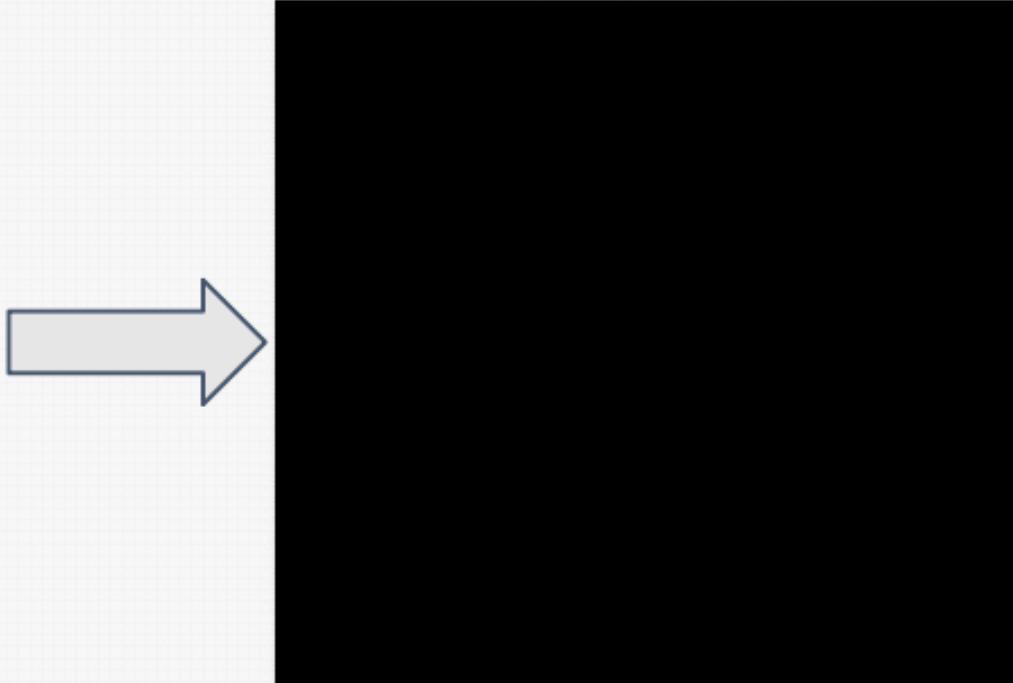
# 06 Web Server Stress 적용

## Auto-Scaling 확인

```
/211.163.1.100 [211.163.1.10] [211.163.1.10]
Percentage of the requests served within a certain time (ms)
50% 223
66% 265
75% 337
90% 555
95% 1219
99% 1306
99.9% 3130
99.99% 3500
100% 16570 (longest request)
root@master:~#
```

```
/211.163.1.100
my-apinx-deployment3-54f449dddf-ptzlv 0/1 ContainerCreating 0 0s
my-apinx-deployment3-54f449dddf-f2fsv 0/1 ContainerCreating 0 0s
my-apinx-deployment3-54f449dddf-ptzlv 1/1 Running 0 4s
my-apinx-deployment3-54f449dddf-f2fsv 1/1 Running 0 11s
my-apinx-deployment3-54f449dddf-fw4s 0/1 Pending 0 0s
my-apinx-deployment3-54f449dddf-fw4s 0/1 Pending 0 0s
my-apinx-deployment3-54f449dddf-fw4s 0/1 ContainerCreating 0 0s
my-apinx-deployment3-54f449dddf-fw4s 1/1 Running 0 4s
```

```
/211.163.1.100
root@master:~# k get hpa -w
NAME      REFERENCE          TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
my-apinx-hpa Deployment/my-apinx-deployment3 0%/30% 1 5 1 4d5h
my-apinx-hpa Deployment/my-apinx-deployment3 40%/30% 1 5 1 4d5h
my-apinx-hpa Deployment/my-apinx-deployment3 204%/30% 1 5 2 4d5h
my-apinx-hpa Deployment/my-apinx-deployment3 476%/30% 1 5 4 4d5h
my-apinx-hpa Deployment/my-apinx-deployment3 0%/30% 1 5 5 4d5h
```



Auto Scaling 시연 영상

# 06 Jenkins pipeline 구성

```
pipeline {  
    agent any  
    environment {  
        DOCKER_IMAGE_NAME = "claendys/my-test"  
        DOCKERFILE_PATH = "web/Dockerfile"  
        YAML_FILE_PATH = 'root/my-app3.yaml' // Correct path to my-app3.yaml  
    }  
  
    stages {  
        stage('Checkout') {  
            steps {  
                script {  
                    git branch: 'master', credentialsId: 'github_access_token', url: 'https://github.com/meycophyun-2011/toyproject123.git'  
                }  
            }  
        }  
  
        stage("Build Docker Image") {  
            steps {  
                script {  
                    def imageTag = env.BUILD_NUMBER  
                    docker.build("${DOCKER_IMAGE_NAME}:${imageTag}", "--file ${DOCKERFILE_PATH} .")  
                }  
            }  
        }  
  
        stage('Update YAML with New Image Tag') {  
            steps {  
                script {  
                    def imageTag = env.BUILD_NUMBER  
                    sh "sed -i 's/image: ${DOCKER_IMAGE_NAME}/#image: ${DOCKER_IMAGE_NAME}:$imageTag/' ${YAML_FILE_PATH}"  
                }  
            }  
        }  
    }  
}
```

1. 도커허브로부터 web이미지 가져오기

2. webhook이 작동해 github에 Dockerfile 경로를 찾는다.

3. 가상머신 경로에 my-app3.yaml 찾아낸다.

1. checkout: GitHub에 소스 코드를 체크아웃 합니다.

2. credentials Id 생성한 이름 넣어줌.

1. 도커 이미지 빌드 정의

1. 환경 변수 사용하여 image Tag 할당

2. Build\_Number는 현재 빌드 번호

3. 쉘 스크립트를 사용하여 빌드 번호 부분이 새로운 번호로 업데이트

# 06 Jenkins pipeline 구성

```
pipeline {  
    agent any  
    environment {  
        DOCKER_IMAGE_NAME = "claendys/my-test"  
        DOCKERFILE_PATH = "web/Dockerfile"  
        YAML_FILE_PATH = 'root/my-app3.yaml' // Correct path to my-app3.yaml  
    }  
  
    stages {  
        stage('Checkout') {  
            steps {  
                script {  
                    git branch: 'master', credentialsId: 'github_access_token', url: 'https://github.com/meycophyun-2011/toyproject123.git'  
                }  
            }  
        }  
  
        stage("Build Docker Image") {  
            steps {  
                script {  
                    def imageTag = env.BUILD_NUMBER  
                    docker.build("${DOCKER_IMAGE_NAME}:${imageTag}", "--file ${DOCKERFILE_PATH} .")  
                }  
            }  
        }  
  
        stage('Update YAML with New Image Tag') {  
            steps {  
                script {  
                    def imageTag = env.BUILD_NUMBER  
                    sh "sed -i 's/image: ${DOCKER_IMAGE_NAME}/#image: ${DOCKER_IMAGE_NAME}:$imageTag/' ${YAML_FILE_PATH}"  
                }  
            }  
        }  
    }  
}
```

1. 도커허브로부터 web이미지 가져오기

2. webhook이 작동해 github에 Dockerfile 경로를 찾는다.

3. 가상머신 경로에 my-app3.yaml 찾아낸다.

1. checkout: GitHub에 소스 코드를 체크아웃 합니다.

2. credentials Id 생성한 이름 넣어줌.

1. 도커 이미지 빌드 정의

1. 환경 변수 사용하여 image Tag 할당

2. Build\_Number는 현재 빌드 번호

3. 쉘 스크립트를 사용하여 빌드 번호 부분이 새로운 번호로 업데이트

# 06 Jenkins pipeline 구성

```
stage('Deploy to Kubernetes') {  
    steps {  
        script {  
            sh "kubectl apply -f ${YAML_FILE_PATH}"  
        }  
    }  
}  
  
stage('Test') {  
    steps {  
        echo 'Running tests...'  
        // Add your test commands here  
    }  
}  
  
stage('Deploy to Docker Hub') {  
    steps {  
        script {  
            def imageTag = env.BUILD_NUMBER  
            docker.withRegistry('https://index.docker.io/v1/', 'hub-token') {  
                docker.image("${DOCKER_IMAGE_NAME}:${imageTag}").push()  
            }  
        }  
    }  
}  
  
post {  
    always {  
        echo 'Pipeline finished.'  
    }  
}
```

1. yaml파일을 실행시키는 블럭

1. Docker Hub에 최신 버전을 Push

# 06 deploy된 내용 describe로 확인

```
root@master:~# cat my-app3.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx-service3
spec:
  selector:
    app: my-nginx-app3
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: LoadBalancer
```

```
---
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx-deployment3
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-nginx-app3
  template:
    metadata:
      labels:
        app: my-nginx-app3
```

```
spec:
  containers:
    - name: my-nginx-container
      image: riaaudgus/my-test3:latest
    ports:
      - containerPort: 80
    resources:
      requests:
        cpu: 200m # CPU 요청을 200m (0.2 CPU)로 증가시킵니다.
```

memory: 512Mi # 메모리 제한을 512Mi로 증가시킵니다.

tag를 docker hub에 있는 ---  
태그로 변경 가능

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: my-nginx-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-nginx-deployment3
  minReplicas: 1
  maxReplicas: 5
  targetCPUUtilizationPercentage: 30
```

root@master:~#

## 06 Deploy.yaml 파일 적용 (k describe Deploy.yaml)

```
MinReadySeconds:          0
RollingUpdateStrategy:   25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=my-nginx-app3
  Containers:
    my-nginx-container:
      Image:      rlaaudgus/my-test3:latest
      Port:       80/TCP
      Host Port:  0/TCP
      Limits:
        cpu:      500m
        memory:   512Mi
      Requests:
        cpu:      200m
        memory:   256Mi
      Environment: <none>
      Mounts:     <none>
      Volumes:    <none>
  Conditions:
    Type      Status  Reason
    ----      ----   -----
    Available  True    MinimumReplicasAvailable
    Progressing True    NewReplicaSetAvailable
  OldReplicaSets: <none>
  NewReplicaSet:  my-nginx-deployment3-7fdd4977c (1/1 replicas created)
Events:
  Type      Reason           Age   From            Message
  ----      ----   ----   ----
  Normal   ScalingReplicaSet 69s   deployment-controller  Scaled up replica set my-nginx-deployment3-7fdd4977c to 1
root@master:~#
```

## 06 Deploy.yaml 파일 적용

```
root@master:~# k apply -f my-app3.yaml
service/my-nginx-service3 unchanged
deployment.apps/my-nginx-deployment3 configured
horizontalpodautoscaler.autoscaling/my-nginx-hpa unchanged
root@master:~#
```

Image: rlaaudgus/my-test3:38로 수정 한 후 yaml파일 적용

## 06 deploy된 내용 describe로 확인

```
Selector:           app=my-nginx-app3
Replicas:          1 desired | 1 updated | 1 total | 1 available | 0 unavailable
StrategyType:      RollingUpdate
MinReadySeconds:   0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=my-nginx-app3
  Containers:
    my-nginx-container:
      Image:  rlaaudgus/my-test3:38
      Port:   80/TCP
      Host Port:  0/TCP
      Limits:
        cpu:  500m
        memory:  512Mi
      Requests:
        cpu:  200m
        memory:  256Mi
      Environment:  <none>
      Mounts:  <none>
    Volumes:  <none>
  Conditions:
    Type        Status  Reason
    ----        ----  -----
    Available   True    MinimumReplicasAvailable
    Progressing True    NewReplicaSetAvailable
OldReplicaSets:  my-nginx-deployment3-77548647f6 (0/0 replicas created), my-nginx-deployment3-5577fb9787 (0/0 replicas created)
NewReplicaSet:   my-nginx-deployment3-c9bcc77d7 (1/1 replicas created)
Events:          <none>
root@master:~#
```

rlaaudgus/my-test3:latest에서  
rlaaudgus/my-test3:38로 수정됨

# 06 jenkins에서 수정한 내용 적용 확인

```
root@master:~# k apply -f my-app3.yaml
service/my-nginx-service3 created
deployment.apps/my-nginx-deployment3 created
horizontalpodautoscaler.autoscaling/my-nginx-hpa unchanged
root@master:~#
```

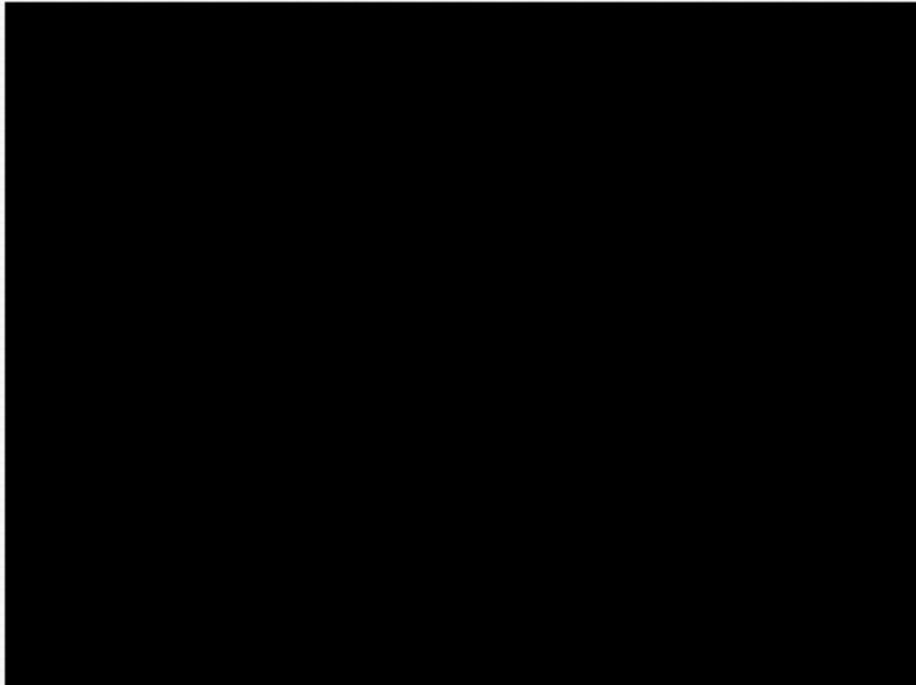
The screenshot shows a Jenkins pipeline interface for a project named 'toy-project'. On the left, there's a sidebar with navigation links like 'Dashboard', 'Status', 'Changes', 'Pipeline', 'GitHub', 'Resumable', 'Pipeline Syntax', and 'GitHub Hook Log'. The main area is titled 'Stage View' and displays a timeline of pipeline stages. The stages and their average run times are:

Stage	Average Stage Time (ms)
Checkout	3s
Build Docker Image	7s
Update YAML with New Image Tag	583ms
Deploy to Kubernetes	3s
Test	227ms
Deploy to Docker Hub	5s
Declarative Post Actions	140ms

Below the table, a summary bar indicates the total average run time is 32s, with 7.2s spent on GitHub changes.

## 06 jenkins에서 수정한 내용 적용 확인

---



# 06 ArgoCD

우리 회장 https://211.103.3.203/applications

카카오기 캐릭터 github GitHub API 정보 myproject 웹사이트 Frontend 코드문서 쟁그 캐카오

### Applications

+ NEW APP    SYNC APPS    RERESH APPS    search applications...    -

Icon	Name	Project	Status	Last Sync
git	my-project	default	<span>Yellow</span> Missing <span>Green</span> OutOfSync	2024-05-28 11:38:42 (4 days ago)
git	my-project1	default	<span>Yellow</span> Missing <span>Green</span> Synced	2024-05-28 11:38:42 (4 days ago)
git	web	default	<span>Green</span> Healthy <span>Yellow</span> OutOfSync	2024-05-28 11:38:42 (4 days ago)

**my-project**

Project: default  
Label:  
Status: Yellow Missing Green OutOfSync  
Repository: <https://github.com/myproject/my-project>  
Target Ref: HEAD  
Path: /  
Default: myroute  
Namespace: default  
Created: 2024-05-28 11:38:42 (4 days ago)  
Last Sync: 2024-05-28 11:38:42 (4 days ago)

**my-project1**

Project: default  
Label:  
Status: Yellow Missing Green Synced  
Repository: <https://github.com/myproject/my-project1>  
Target Ref: HEAD  
Path: /gitops  
Default: myroute  
Namespace: default  
Created: 2024-05-28 11:38:42 (4 days ago)  
Last Sync: 2024-05-28 11:38:42 (4 days ago)

**web**

Project: default  
Label:  
Status: Green Healthy Yellow OutOfSync  
Repository: <https://github.com/myproject/web>  
Target Ref: HEAD  
Path: /web  
Default: myroute  
Namespace: default  
Created: 2024-05-28 11:38:42 (4 days ago)  
Last Sync: 2024-05-28 11:38:42 (4 days ago)

**Actions**    **Checklist**    **Details**



# 06 ArgoCD

The screenshot shows a dual-pane interface for managing a Kubernetes application. On the left, a terminal window displays the YAML configuration for a Deployment named 'my-nginx-deployment3'. The configuration includes a single replica, a selector, and matching labels. On the right, the 'Application Details Tree' pane visualizes the deployment's components and their relationships:

- Deployment:** my-nginx-deployment3 (1/1 healthy, rev-10) - This is the central component.
- Service:** my-nginx-service3 (1/1 healthy) - Associated with the deployment.
- Endpoint:** my-nginx-service3-ws8hx (1/1 healthy) - Associated with the service.
- End-to-end Service:** my-nginx-service3 (1/1 healthy) - Associated with the endpoint.
- HPA:** my-nginx-hpa (1/1 healthy) - Associated with the deployment.
- Pod:** my-nginx-deployment3-54f44... (1/1 healthy, running/5/5) - The final execution unit.

The tree structure shows the flow from the deployment through the service and endpoint to the final pod. Each component has a status indicator (green for healthy) and a revision number (rev-10 for the deployment).

# 06 ArgoCD

The screenshot shows a dual-pane interface for managing a Kubernetes application. On the left, a terminal window titled "SuperPutTY - 211.183.3.100" displays the YAML configuration for a Deployment named "my-nginx-deployment3". The configuration includes fields for metadata, spec (replicas: 3, selector: app: my-nginx-app3), and template (metadata, labels). On the right, the "APPLICATION DETAILS TREE" pane provides a hierarchical view of the application's components:

- Deployment:** my-nginx-deployment3 (1.4.0.9) - Last updated 18 hours ago. Contains 3 replicas.
- Service:** my-nginx-service3 (1.4.0.9) - Last updated 18 hours ago. Associated with my-nginx-deployment3.
- Pods:**
  - my-nginx-deployment3-7fdd4... (11) - Last updated a few seconds ago. Status: running.
  - my-nginx-deployment3-7fdd4... (11) - Last updated a minute ago. Status: running.
  - my-nginx-deployment3-54f4d... (11) - Last updated 17 hours ago. Status: running.

The tree view is currently collapsed, indicated by a "Tree" icon in the top right corner of the pane.

# 07 TroubleShooting

## Problem

GCP환경에서 처음 올린 서비스는 External IP를 할당받았으나 어느 순간부터 서비스에 대한 External IP가 할당되지 않는 문제가 발생했습니다.

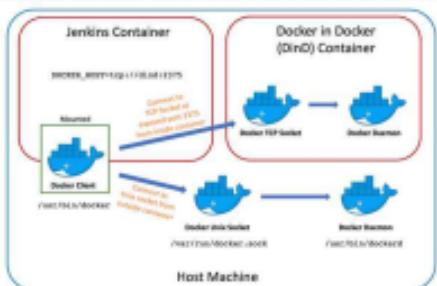
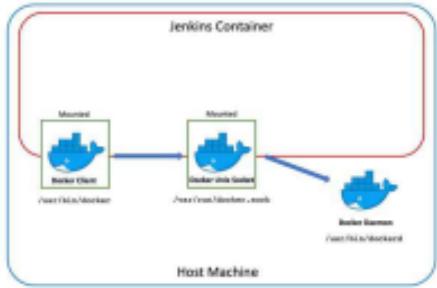
## Analyze

1. 보안 정책을 다시 확인해 봤습니다  
→ 문제X
2. 서비스 배포 시 사용했던 yaml파일에 대해 코드 인스펙션을 진행하였습니다  
→ 문제X
3. GKE 자체에 문제가 있는지 검토해봤습니다

## Solution

GKE를 free tier로 사용하다보니 External IP의 할당 갯수 제한이 있었습니다.  
→ GKE에 on-premiss로 개발 환경을 변경하였습니다.

# 07 TroubleShooting



## Problem

Jenkins를 Pod로 배포하게되면 docker명령어를 사용 불가!  
이를 어떻게 해결할 것인가!

## Analyze

- 원인은 컨테이너는 독립되어 있어서 도커 명령어를 사용 할 수 없기 때문!
- 그렇다면 도커 명령어 사용법은?  
컨테이너에 도커 설치  
도커 소켓 연결  
도커 컨테이너를 띄워서 도커 컨테이너와 통신하여 사용 (DinD)

## Solution

일반적으로 알려진 방식인 docker.sock 마운트를 통해 host vm의 docker를 사용한다!

# 07 TroubleShooting

## Problem

On-premises 환경에서 Jenkins를 설치한 후 GitHub에 커밋이 되었음에도 불구하고 Jenkins가 자동으로 작업을 실행하지 않습니다.

## Analyze

Github에서 보내는 event 신호를 받으려면 24시간 대기 중인 Jenkins서버가 필요합니다.  
따라서 GitHub의 웹훅이 Jenkins로의 메시지를 전달할 수 없었습니다.

## Solution

1. 로컬 네트워크 설정에서 포트 포워딩 시도  
-> 외부에서 Jenkins에 액세스할 수 없었습니다.
2. ngrok을 사용하여 로컬 Jenkins 서버를 외부에 노출시키는 방법을 시도  
-> 해결!

## ngrok이란?

Github와 Jenkins를 연동하기 위해 url을 입력할 때 localhost url은 사용할 수가 없기 때문에 ngrok은 localhost에서도 외부 ip을 사용하는 것처럼 해주는 역할을 합니다

## 07 After work 시간이 더 있었다면...

---

1

local에서는 실행되는 서비스가 이미지로 만들어 pod로 배포하려고 했으나 Dockerfile 작성에 오류가 있어 이미지 만드는데 문제가 있었습니다. 시간이 좀 더 있었으면 이미지를 만들고 웹도 띄어 볼 수 있었을 것 같다는 아쉬움이 있습니다.

2

맨 처음에는 GCP상에서 서비스를 운영하려 하였으나 팀원 모두 GCP에 대해 공부 할 시간이 부족했습니다.  
GCP에 대해 공부를 하고 현재 프로젝트를 GKE환경에서 운영해보고 싶습니다.

## 06 플랫폼의 장점, 한계점 및 개선 방안

---



파일의 무결성 및 안정성 보장



배포의 용이성 및  
유지 보수의 용이성 보장

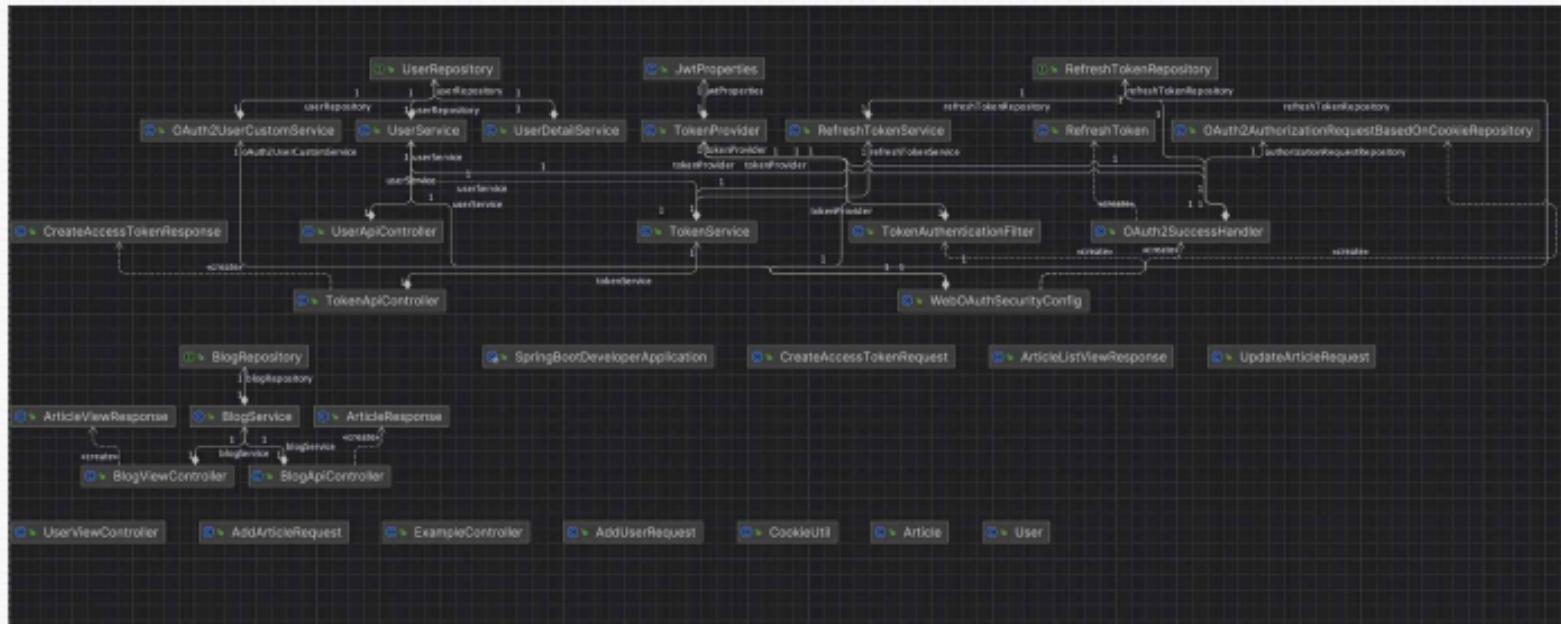


Manager노드의 이중화로  
1개의 manager가 다운 되도 다른  
manager가 역할을 할 수 있음



내부 ip와 외부 ip를 따로 두어  
보안성 강화

# 04 클래스 다이어그램 상속관계



## 04 Prior Knowledge

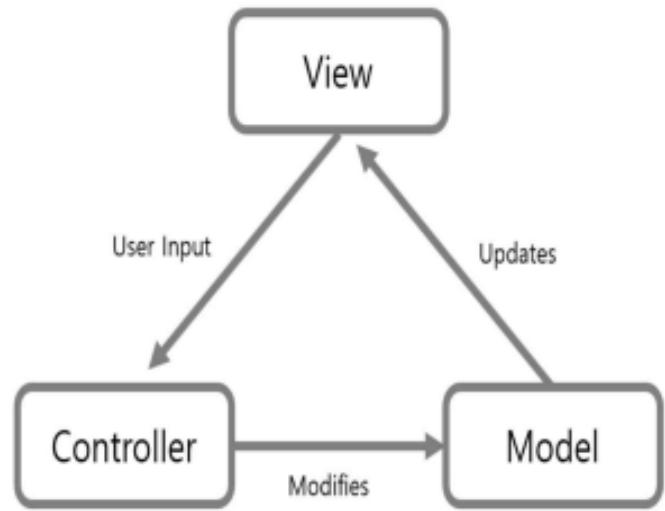
---

Spring MVC (Design)

Spring Data JPA (Hibernate)

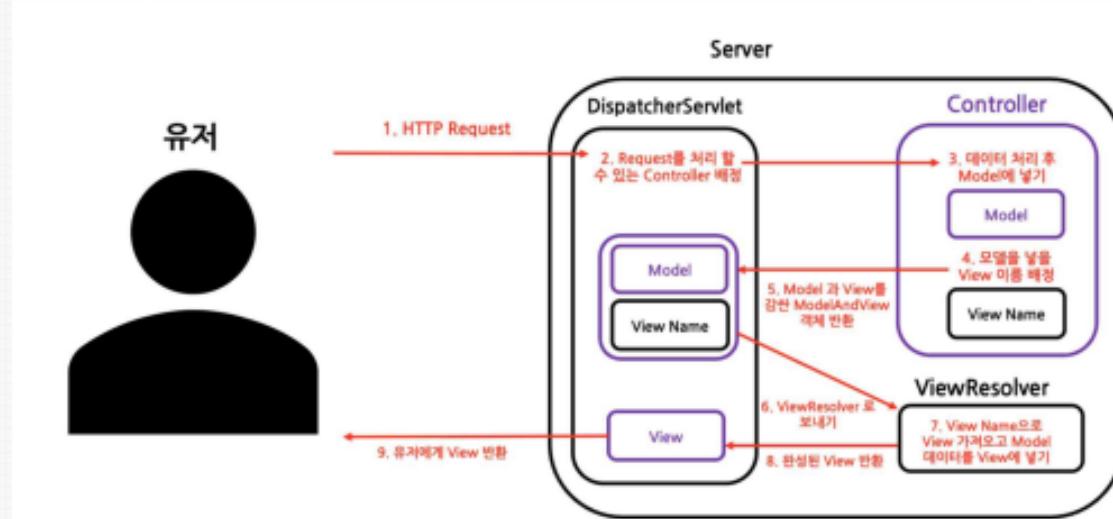
Spring security (OAUTH)

# 04 MVC



Model –View – Controller로 하나의 구성요소로 세 가지의 역할로 나눠지는 패턴

# 04 MVC



모델은 서비스 로직을 소화하는 역할  
뷰는 컨트롤러를 통해 미리 넘겨 받은 인자만을 사용해서 모델 코드를 사용  
컨트롤러는 사용자가 접근한 URL에 따라서 데이터 모델을 뷰에 반영

## 04 MVC 예시 – 서비스 메서드 코드

```
@NoArgsConstructor  
@AllArgsConstructor  
@Getter  
public class AddArticleRequest {  
    private String title;  
  
    private String content;  
  
    public Article toEntity(String author) {  
        return Article.builder()  
            .title(title)  
            .content(content)  
            .author(author)  
            .build();  
    }  
}
```

서비스 계층에 블로그에 글을 추가하는 코드

AddArticleRequest라는 객체를 생성한다.

패키지 분류는 dto이고 데이터를 전달하는 객체들의 집합.

데이터를 전달하는 목적으로 설계되었으므로  
별도의 비즈니스 로직을 고려하지 않는다.

# 04 MVC 예시 – 서비스 메서드 코드

```
@Service
public class BlogService {

    private final BlogRepository blogRepository;

    public Article save(AddArticleRequest request, String userName) {
        return blogRepository.save(request.toEntity(userName));
    }

    public List<Article> findAll() {
        return blogRepository.findAll();
    }

    public Article findById(long id) {
        return blogRepository.findById(id)
            .orElseThrow(() -> new IllegalArgumentException("not found : " + id));
    }

    public void delete(long id) {
        Article article = blogRepository.findById(id)
            .orElseThrow(() -> new IllegalArgumentException("not found : " + id));

        authorizeArticleAuthor(article);
        blogRepository.delete(article);
    }
}
```

블로그 서비스 객체는 블로그와 관련된 기능을 제공하는 서비스 클래스

게시물 추가, 게시물 조회, 삭제, 업데이트 등이 이뤄지는 비즈니스 로직 클래스

Ex) save()로  
AddarticleRequest 클래스에  
저장된 값들을 Article 데이터  
베이스에 저장

## 04 MVC 예시 – 컨트롤러

```
@RestController
public class BlogApiController {
    private final BlogService blogService;

    @PostMapping("/api/articles")
    public ResponseEntity<Article> addArticle(@RequestBody AddArticleRequest request, Principal principal) {
        Article savedArticle = blogService.save(request, principal.getName());
        return ResponseEntity.status(HttpStatus.CREATED)
            .body(savedArticle);
    }

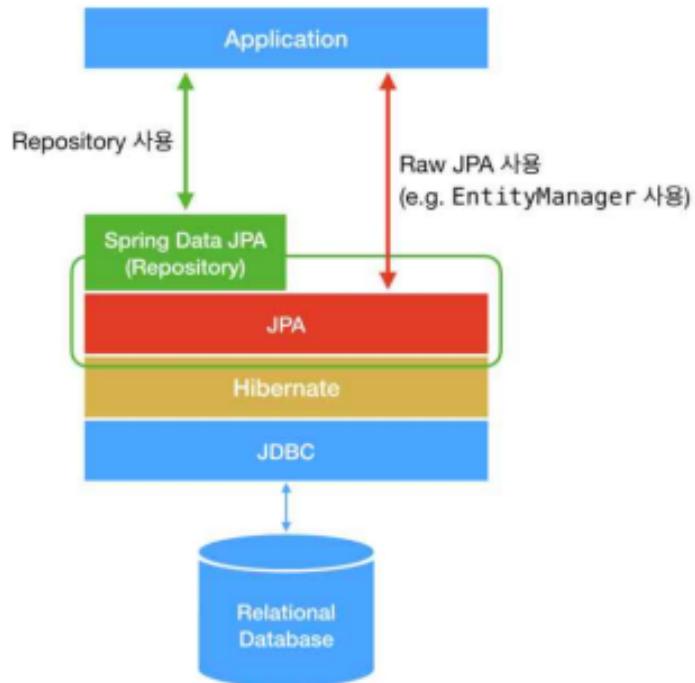
    @GetMapping("/api/articles")
    public ResponseEntity<List<ArticleResponse>> findAllArticles() {
        List<ArticleResponse> articles = blogService.findAll() List<Article>
            .stream() Stream<Article>
            .map(ArticleResponse::new) Stream<ArticleResponse>
            .toList();
        return ResponseEntity.ok()
            .body(articles);
    }
}
```

컨트롤러의 역할은 요청이 오면 url을 통해 요청을 매핑한뒤 요청된 작업을 반환해주는 역할

BlogService 의존성 주입으로  
BlogApiController는 BlogService를 의존하고 정의된 서비스 로직을 사용

Ex) /api/articles에 post 요청이 오면  
BlogService의 save() 메서드를 호출하고  
생성된 글을 반환

# 04 JPA? Hibernate?

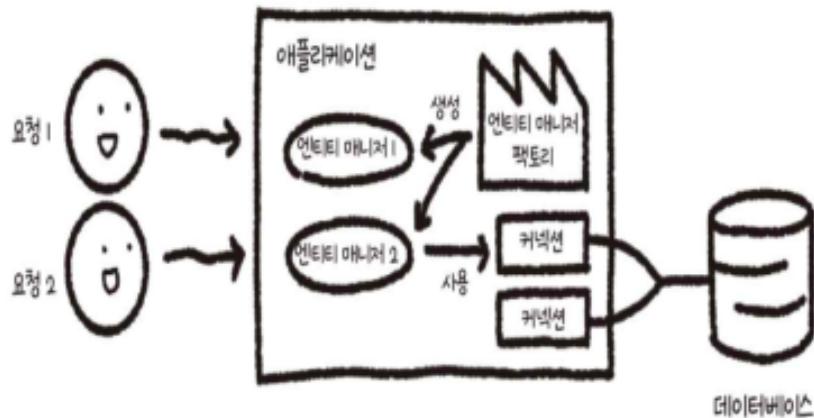


JPA는 자바에서 관계형 데이터베이스를 사용하는 방식의 인터페이스

Hibernate는 JPA인터페이스를 구현한 구현체이자 자바용 ORM 프레임워크

하이버네이트를 사용하는 이유는 모든 것을 객체로 바라보기 때문에 데이터베이스 종류에 상관없이 데이터베이스를 자유자제로 사용함

## 04 Prior Knowledge



엔티티는 데이터베이스의 테이블과 매핑되는 객체

엔티티 매니저는 엔티티를 관리하는 데이터베이스  
와 애플리케이션 사이에서 객체를 생성, 수정한다.

# 04 엔티티 상태

```
public class EntityManagerSampleTest {  
  
    @Autowired  
    EntityManager em;  
  
    public void example() {  
  
        //1. 관리되지 않은 상태  
        Member member = new Member( 1L, "길동" );  
  
        //2. 영속상태(관리상태)  
        em.persist( member );  
  
        //3. 분리상태  
        em.detach( member );  
  
        //4. 삭제상태  
        em.remove( member );  
  
    }  
}
```

엔티티(db객체)는 기본적으로 4가지 상태를 가짐

특정 메서드를 호출해 엔티티의 상태를 조절하여 데이터를 올바르게 유지하는 목적으로 사용

Spring Data JDBC가 객체라는 개념으로 Data와 table을 Mapping시켜 놓았다.

## 04 엔티티를 어떻게 다룰까?

```
public interface RefreshTokenRepository extends JpaRepository<RefreshToken, Long> {  
    Optional<RefreshToken> findByUserId(Long userId);  
    Optional<RefreshToken> findByRefreshToken(String refreshToken);  
}
```

JPA는 인터페이스고 사용하려는 Repository에서 JPA를 사용받아 JPA에서 제공하는 기능들을 사용하여 객체를 관리

```
public interface UserRepository extends JpaRepository<User, Long> {  
    Optional<User> findByEmail(String email);  
}
```

Spring Data JDBC가 객체라는 개념으로 Data와 table을 Mapping시켜 놓았다.

# 04 Article Table

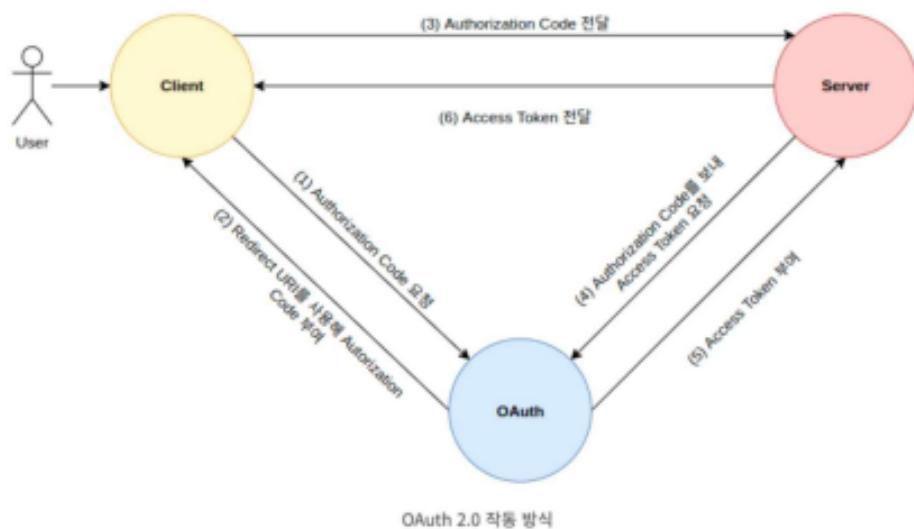
```
17 @Entity  
18 public class Article {  
19  
20     @Id  
21     @GeneratedValue(strategy = GenerationType.IDENTITY)  
22     @Column(name = "id", updatable = false)  
23     private Long id;  
24  
25     @Column(name = "title", nullable = false)  
26     private String title;  
27  
28     @Column(name = "content", nullable = false)  
29     private String content;  
30  
31     @Column(name = "author", nullable = false)  
32     private String author;  
33  
34     @CreatedDate  
35     @Column(name = "created_at")  
36     private LocalDateTime createdAt;  
37  
38     @LastModifiedDate  
39     @Column(name = "updated_at")  
40     private LocalDateTime updatedAt;  
41  
42     @Builder  
43     public Article(String author, String title, String content) {  
44         this.author = author;  
45         this.title = title;  
46         this.content = content;  
47     }  
}
```

## O/R Mapping

Spring Data JDBC가 객체라는 개념으로 Data와 table을 Mapping시켜 놓았다.

@Entity 속성에서 name을 사용하면 name의 값을 가진 테이블 이름과 매핑된다.

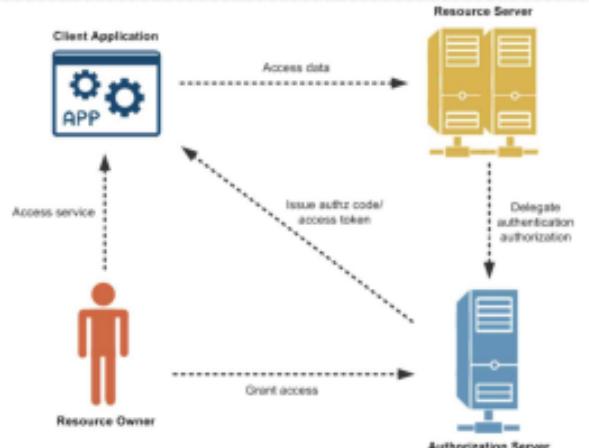
## Security OAuth2



우선 Oauth란 제 3의서비스에 계정 관리를 맡기는 방식이다.

Oauth를 사용하면 인증 서버에서 토큰을 사용해서 리소스 서버에 리소스 오너의 정보를 요청하고 응답받아 사용한다.

## Oauth2 작동 방식



Oauth는 권한부여 타입을 사용한다.

리소스 오너가 애플리케이션에게 권한을 요청하면 서버는 데이터 접근 권한을 부여하고 애플리케이션은 인증 코드를 발급해주고 엑세스 토큰을 발급해 준다.

```
implementation 'org.springframework.boot:spring-boot-starter-oauth2-client'  
implementation 'io.jsonwebtoken:jjwt:0.9.1'
```

Oauth 기능을 사용하기 위한  
의존성 추가

## Oauth2 토큰 발급

### OAuth 클라이언트 생성됨

API 및 서비스의 사용자 인증 정보에서 언제든지 클라이언트 ID와 보안 비밀에 액세스할 수 있습니다.

- ❶ OAuth 액세스는 [OAuth 동의 화면](#)에 나열된 [테스트 사용자](#) [x]로 제한됩니다.

클라이언트 ID	450518053586-d4mmc1gvc6o28ct056rpt6q9i7qd1rre.apps.googleusercontent.com	
----------	--	--

클라이언트 보안 비밀번호	QOCSPX-QWv6yq3btZl8s92a1a0by3KL3oJW	
생성일	2024년 3월 31일 PM 5시 43분 54초 GMT+9	
상태	사용 설정됨	

JSON 다운로드

#### OAuth 2.0 클라이언트 ID

이름	생성일	유형	클라이언트 ID
springboot-developer	2024. 3. 31.	일반 클라이언트	450518053586-d4mc...

```
security:  
  oauth2:  
    client:  
      registration:  
        google:  
          client-id: 45051~  
          client-secret: ~  
          scope:
```

발급받은 토큰은 프로젝트 Application.yaml에 인증정보를 등록한다.

등록한 내용의 바탕으로 로그인 기능을 구현

## 쿠키 관리 클래스 구현

```
public class CookieUtil {  
  
    public static void addCookie(HttpServletRequest response, String name, String value, int maxAge) {  
        Cookie cookie = new Cookie(name, value);  
        cookie.setPath("/");  
        cookie.setMaxAge(maxAge);  
  
        response.addCookie(cookie);  
    }  
  
    public static void deleteCookie(HttpServletRequest request, HttpServletResponse response, String name) {  
        Cookie[] cookies = request.getCookies();  
  
        if (cookies == null) {  
            return;  
        }  
  
        for (Cookie cookie : cookies) {  
            if (name.equals(cookie.getName())) {  
                cookie.setValue("");  
                cookie.setPath("*");  
                cookie.setMaxAge(0);  
                response.addCookie(cookie);  
            }  
        }  
    }  
}
```

Oauth 인증 플로우 구현하면 쿠키를 사용할 일이 생기는데 그때마다 쿠키를 생성하고 삭제하는 로직을 추가하면 불편하기 때문에 쿠키를 관리할 객체를 구현

## 인증 요청 관련 저장소

```
public class OAuth2AuthorizationRequestBasedOnCookieRepository implements AuthorizationRequestReposito  
y  
  
    public final static String OAUTH2_AUTHORIZATION_REQUEST_COOKIE_NAME = "oauth2_auth_request";  
    private final static int COOKIE_EXPIRE_SECONDS = 18000;  
  
    @Override  
    public OAuth2AuthorizationRequest removeAuthorizationRequest(HttpServletRequest request, HttpServletRespon  
se response) {  
        return this.loadAuthorizationRequest(request);  
    }  
  
    @Override  
    public OAuth2AuthorizationRequest loadAuthorizationRequest(HttpServletRequest request) {  
        Cookie cookie = WebUtils.getCookie(request, OAUTH2_AUTHORIZATION_REQUEST_COOKIE_NAME);  
        return CookieUtil.deserialize(cookie, OAuth2AuthorizationRequest.class);  
    }  
  
    @Override  
    public void saveAuthorizationRequest(OAuth2AuthorizationRequest authorizationRequest, HttpServletRequest  
request, HttpServletResponse response) {  
        if (authorizationRequest == null) {  
            removeAuthorizationRequestCookies(request, response);  
            return;  
        }  
  
        CookieUtil.addCookie(response, OAUTH2_AUTHORIZATION_REQUEST_COOKIE_NAME, CookieUtil.serialize(au  
thorizationRequest));  
    }  
}
```

Oauth에 필요한 정보를 세션이 아닌 쿠  
키에 저장소 역할 클래스

권한 인증 흐름에서 클라이언트의 요청  
을 유지하는 데 사용하는 클래스를 구현  
해 Oauth의 정보를 가져오고 저장하는  
로직을 작성하였다.

## 인증 성공 반환 클래스

```
@RequiredArgsConstructor  
@Component  
public class OAuth2SuccessHandler extends SimpleUrlAuthenticationSuccessHandler {  
  
    public static final String REFRESH_TOKEN_COOKIE_NAME = "refresh_token";  
    public static final Duration REFRESH_TOKEN_DURATION = Duration.ofDays(14);  
    public static final Duration ACCESS_TOKEN_DURATION = Duration.ofDays(1);  
    public static final String REDIRECT_PATH = "/articles";  
  
    private final TokenProvider tokenProvider;  
    private final RefreshTokenRepository refreshTokenRepository;  
    private final OAuth2AuthorizationRequestBasedOnCookieRepository authorizationRequestRepository;  
    private final UserService userService;  
  
    @Override  
    public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response, Authentication authentication) {  
        OAuth2User oAuth2User = (OAuth2User) authentication.getPrincipal();  
        User user = userService.findByEmail((String) oAuth2User.getAttributes().get("email"));  
  
        String refreshToken = tokenProvider.generateToken(user, REFRESH_TOKEN_DURATION);  
        saveRefreshToken(user.getId(), refreshToken);  
        addRefreshTokenToCookie(request, response, refreshToken);  
  
        String accessToken = tokenProvider.generateToken(user, ACCESS_TOKEN_DURATION);  
        String targetUrl = getTargetUrl(accessToken);  
  
        clearAuthenticationAttributes(request, response);  
  
        net.RedirectStrategy[] sendRedirect(request, response, targetUrl);  
    }  
}
```

인증 성공 시 실행할 핸들러를 구현한다.

토큰 교환 또는 다른 인증 프로세스가

완료되면 다른 인증 핸들러에서 권한을  
로드하고 반환한다.

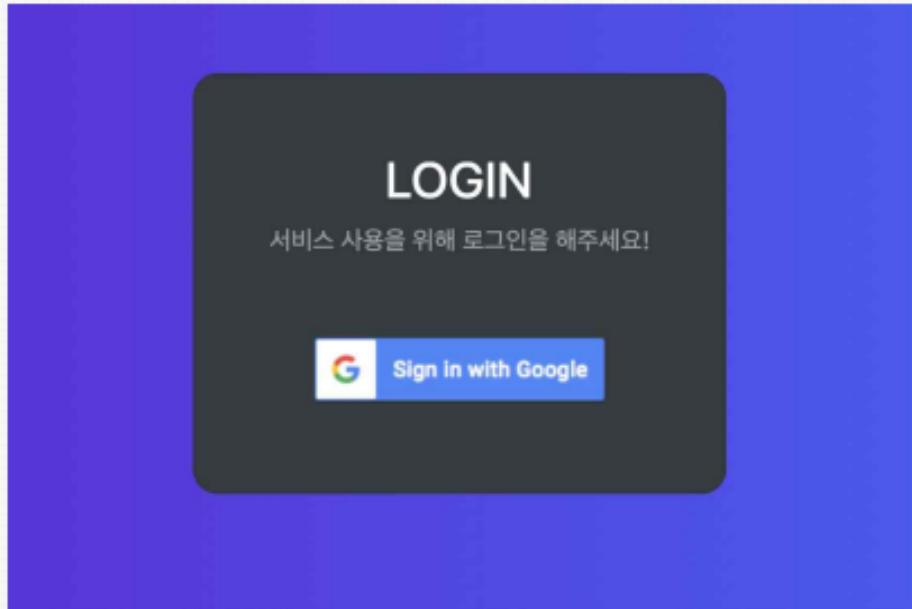
그리고 반환된 권한과 인가를 임시로

저장하고 임시로 저장된 요청이

검증에서 통과하면 해당 요청 사항을 제거하고 통과시킨다.

## 로그인 화면 구현

---



localhost:8080/login에 접속하고  
Oauth를 이용하여 로그인한다.

## Oauth 계정 선택

### 계정 선택

taewoo(으)로 이동

-  **태우**   
 qkrxodn6035@gmail.com
-  **daebak**   
 qkrxodn1502@gmail.com
-  **김태우**   
 rkdixodn6035@gmail.com
-  **다른 계정 사용**

계속 진행하기 위해 Google에서 내 이름, 이메일 주소, 언어 설정 설정, 프로필 사진을 taewoo(과)와 공유합니다.

localhost:8080/login에 접속하면 계정을 선택할 수 있는 화면이 리다이렉트 된다.

## Local 접속

### 계정 선택

taewoo(으)로 이동



### My Blog

블로그에 오신 것을 환영합니다.

The screenshot shows a simple blog article creation form. It includes fields for the title ('제목'), content ('내용'), and a '보내기' (Send) button at the bottom.

글쓰기
제목1
내용1
<button>보내기</button>

localhost:8080/login에 접속하면 계정을 선택할 수 있는 화면이 리다이렉트 된다.

로그인을 하면 article로 리다이렉트 된다.

블로그 db 접속

## My Blog

블로그에 오신 것을 환영합니다.

ID	AUTHOR	CONTENT	CREATED_AT	TITLE	UPDATED_AT
1	user1	내용1	2024-03-31 21:58:40.792371	제목1	2024-03-31 21:58:40.792371
2	user2	내용2	2024-03-31 21:58:40.793115	제목2	2024-03-31 21:58:40.793115
3	user3	내용3	2024-03-31 21:58:40.793272	제목3	2024-03-31 21:58:40.793272
4	cknaxodn6035@gmail.com	gdgdgd	2024-03-31 21:59:33.592771	gdgd	2024-03-31 21:59:33.592771

글을 등록하면 인메모리에 db에도 내용을 확인할 수 있다.

THANK YOU

감사합니다.