TEKTON 101

2025-04-23

테크톤 랩 구성

Hyper V

OpenStack

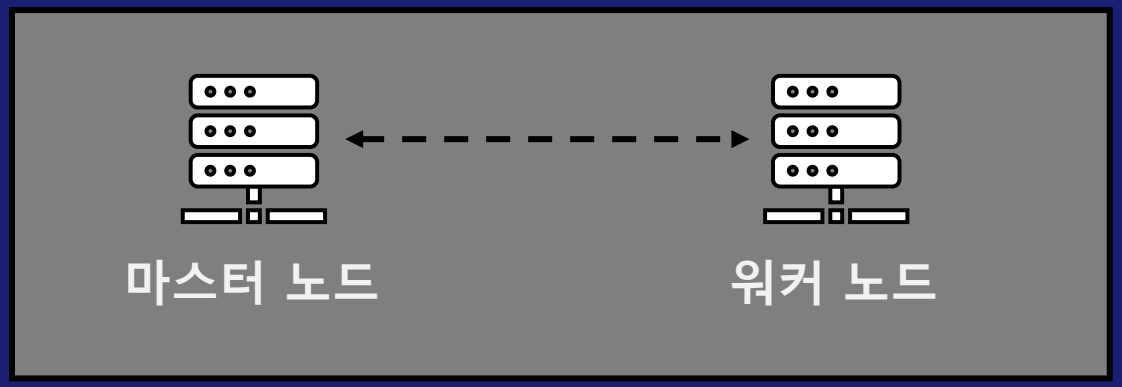
Hyper-V

랩

가상머신 구성

자원 상태에 따라서 다르지만, 기본적으로 마스터/워커 노드 가상머신으로 하나씩 생성이 되어 있으면 된다.

각 가상머신에는 eth0, eth1 혹은 ens0, ens1형식으로 NIC카드 두 개를 가지고 있다. 0번은 default-network, 1번은 static-network를 구성한다.



master/worker node
eth0: DHCP

master/worker node
master[eth1]: 10.10.20.1
worker[eth1]: 10.10.20.2

하이퍼브이 설치 및 구성

하이퍼브이 기반으로 진행하는 경우, 수강생이 직접 랩 아키텍처에 맞게 적절하게 구성한다.

하이퍼브이 네트워크

네트워크 설정은 아래와 같이 구성한다. 먼저, 내부 네트워크를 생성한다.

```
powershell> New-VMSwitch -name InternalSwitch -SwitchType Internal
powershell> ADD-VMNetworkAdapter -VMName master -Switchname Internal
```

외부 네트워크를 생성한다. 기본 "default"네트워크는 아이피가 계속 변경이 된다.

```
powershell> New-VMSwitch -SwitchName "StaticNATSwitch" -SwitchType Internal
powershell> Get-NetAdapter
powershell> New-NetIPAddress -IPAddress 192.168.0.1 -PrefixLength 24 -
InterfaceIndex <INDEX>
powershell> New-NetNat -Name StaticNATSwitch /
 -InternalIPInterfaceAddressPrefix 192.168.0.0/24
```

오픈스택

랩

OPENSTACK

OS는 오픈스택을 통해서 제공하고 있다. 강사의 가이드에 따라서 오픈스택에서 가상머신 구성 후, 쿠버네티스 설치를 진행한다.

`codelab`에서 `cicd-lab.yaml`를 사용하여 가상머신을 프로비저닝 한다. 쿠버네티스 설치는 시간이 오래 걸리기 때문에 `duststack-k8s-auto`를 사용해서 쿠버네티스 설치 한다.

HelmChart설치

HelmChart를 통해서 Tekton설치를 진행한다. 대시보드 설치는 옵션이다.

```
# helm repo add tekton https://cicd.tekton.dev/helm/chartshelm repo update
# helm install tekton-pipelines tekton/tekton-pipeline \
--namespace tekton-pipelines --create-namespace
# helm install tekton-triggers tekton/tekton-trigger \
--namespace tekton-pipelines
# kubectl apply -f
https://github.com/tektoncd/dashboard/releases/latest/download/tekton-
dashboard-release.yaml
# kubectl get crds | grep tekton
# kubectl get pods -n tekton-pipelines
```

에디터 및 확인

랩

hello.yaml

올바르게 동작하는 확인하기 위해서 **hello.yaml**파일을 생성한다.

```
# vi hello.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: hello
spec:
  steps:
    - image: quay.io/centos/centos
      command:
        - /bin/bash
        - -c
        - echo "Hello World"
```

hello.yaml

적용 및 올바르게 동작하는지 확인한다. **tkn**명령어가 없는 경우 다운로드 받아서 적절한 위치에 설치한다.

```
# kubectl apply -f hello.yaml
# mkdir ~/bin/
# tar xzf tkn_0.32.0_Linux_x86_64.tar.gz -C ~/bin/
# tkn task list
NAME    DESCRIPTION   AGE
hello                 55 seconds ago
```

설치 디버깅

테크톤이 올바르게 동작하지 않으면 아래 명령어 실행. 구 버전에서 종종 API문제로 발생.

```
# kubectl delete \
validatingwebhookconfigurations.admissionregistration.k8s.io
config.webhook.pipeline.tekton.dev
# kubectl delete \
validatingwebhookconfigurations.admissionregistration.k8s.io
validation.webhook.pipeline.tekton.dev
# kubectl delete \
mutatingwebhookconfigurations.admissionregistration.k8s.io
webhook.pipeline.tekton.dev
```

에디터 설정(vim)

파일 작성을 원활하게 하기 위해서 `yamllint, ale` 그리고 `neovim`기반으로 진행한다. `neovim`과 `vim+ale`은 동작 방식이 다르기 때문에 선호하는 방식으로 선택 후 설치를 진행한다.

```
# curl -sS https://webi.sh/vim-ale | sh
# dnf install epel-release -y
# dnf search yamllint
yamllint.noarch : A linter for YAML files
# dnf install yamllint -y
# dnf install neovim-ale -y
```

.vimrc

VIM, NeoVIM은 둘 다 같은 설정 파일을 공유한다. VI계열을 사용하는 경우 아래와 같이 옵션을 설정한다. 다만, ALE, YAMLLINT는 꼭 설치가 되어 있어야 한다.

```
# cat <<EOF> ~/.vimrc
autocmd FileType yaml setlocal ts=2 sts=2 sw=2 expandtab

set foldlevelstart=20

let g:ale_echo_msg_format = '[%linter%] %s [%severity%]'
let g:ale_sign_error = '✗'
let g:ale_sign_warning = '⚠️'
let g:ale_lint_on_text_changed = 'never'
EOF
```

에디터 설정(nano)

나노 에디터 사용하는 경우 아래처럼 작업을 수행한다.

```
# dnf search nano
# dnf install nano -y
# cat <<EOF> ~/.nanorc
set tabsize 2
set tabstospaces
EOF
```

.nanorc

```
# nano /usr/share/nano/yaml.nanorc
syntax "YAML" "\.ya?ml$"
header "^(---|\u2550\u2550\u2550)" "%YAML"
color magenta "^\s*[\$A-Za-z0-9_-]+\:"
color brightmagenta "^\s*@[\$A-Za-z0-9_-]+\:"
color white ":\s.+$"
icolor brightcyan " (y|yes|n|no|true|false|on|off)$"
color brightred " [[:digit:]]+(\.[[:digit:]]+)?"
color red "\[" "\]" ":\s+[\u25B7]" "^\s*- "
color green "(^| )!!(binary|bool|float|int|map|null|omap|seq|set|str) "
color brightwhite "#.*$"
```

.nanorc

```
color ,red ":\w.+$"
color ,red ":'.+$"
color ,red ":".+$"
color ,red "\s+$"
color ,red "['\"][^['\"]]*$"
color yellow "['\"].*['\"]"
color brightgreen ":( |$)"
```

tkn CLI

```
# tkn completion zsh > tkn_completion.zsh
# tkn completion bash > tkn_completion.bash
# source tkn_completion.bash
```

tmux

```
# dnf install tmux -y
# cat <<EOF> ~/.tmux.conf
set -g mouse on
set -g prefix C-a
unbind C-b
bind C-a send-prefix
EOF
```

테크톤

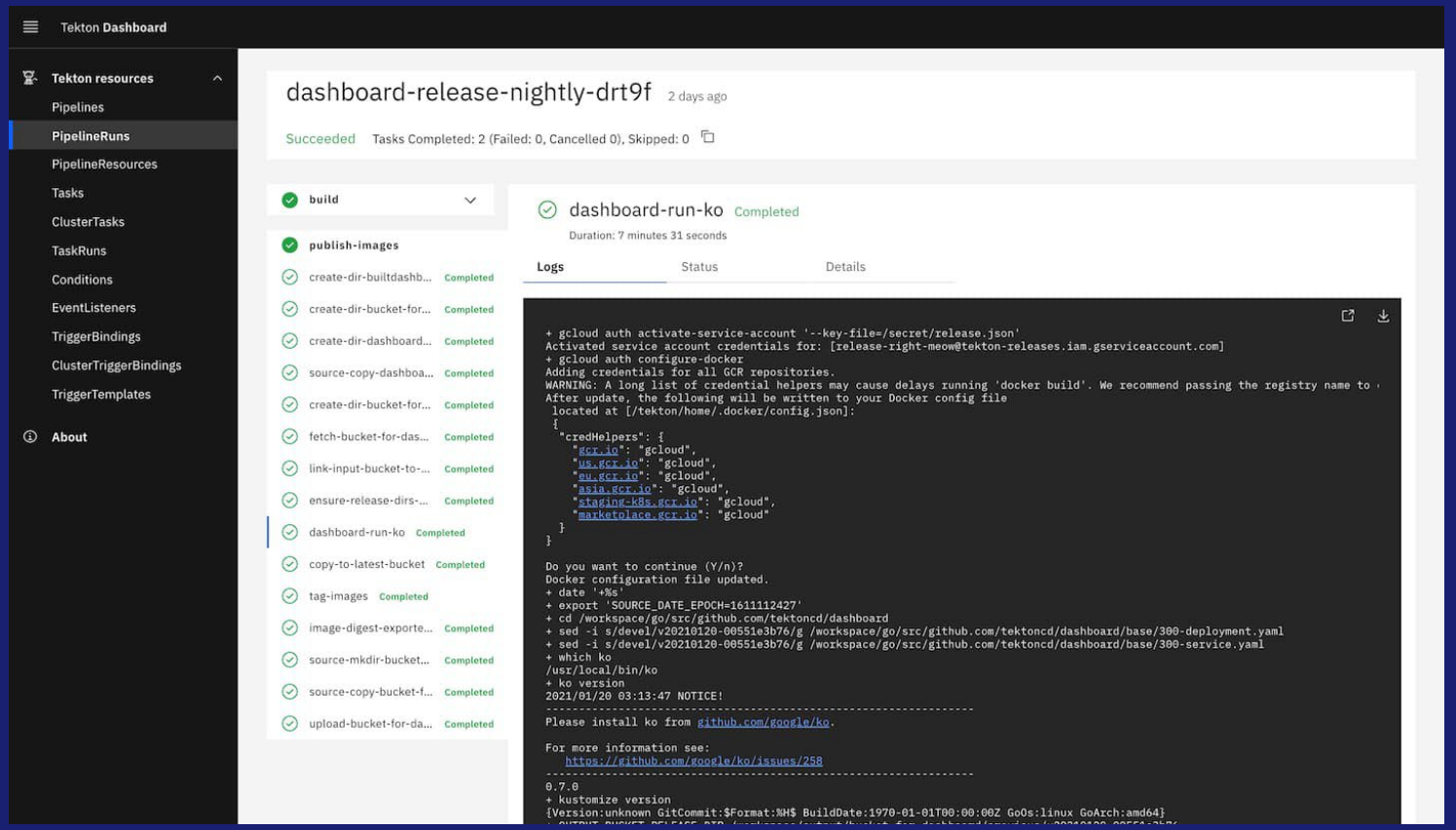
대시보드

Tekton dashboard

테크톤 대시 보드는 간단하게 아래 명령어로 설치가 가능하다.

```
# kubectl apply --filename https://storage.googleapis.com/tekton-releases/dashboard/latest/release.yaml
# kubectl --namespace tekton-pipelines port-forward svc/tekton-dashboard 9097:9097
# kubectl --namespace tekton-pipelines port-forward svc/tekton-dashboard --address <ETH0_IP_ADDRESS> 9097:9097
host]# firefox http://<ETH0_IP_ADDRESS>:9097/
```

Tekton dashboard



테크톤 소개

CNCF TEKTON

TEKTON

테크톤은 CNCF에서 공식으로 지정한 **Continuous Delivery**도구이다.

테크톤은 엄밀히 말하자면 **절차적인 자동화 도구**라고 보시면 된다. 테크톤은 Continuous Integration기능도 제공하나, SCM에 구성이 되어 있는 소스코드를 가져와서 이미지 빌드를 할 수 있도록 지원한다.

테크톤은 개발자에게 다음과 같은 기능 제공한다.

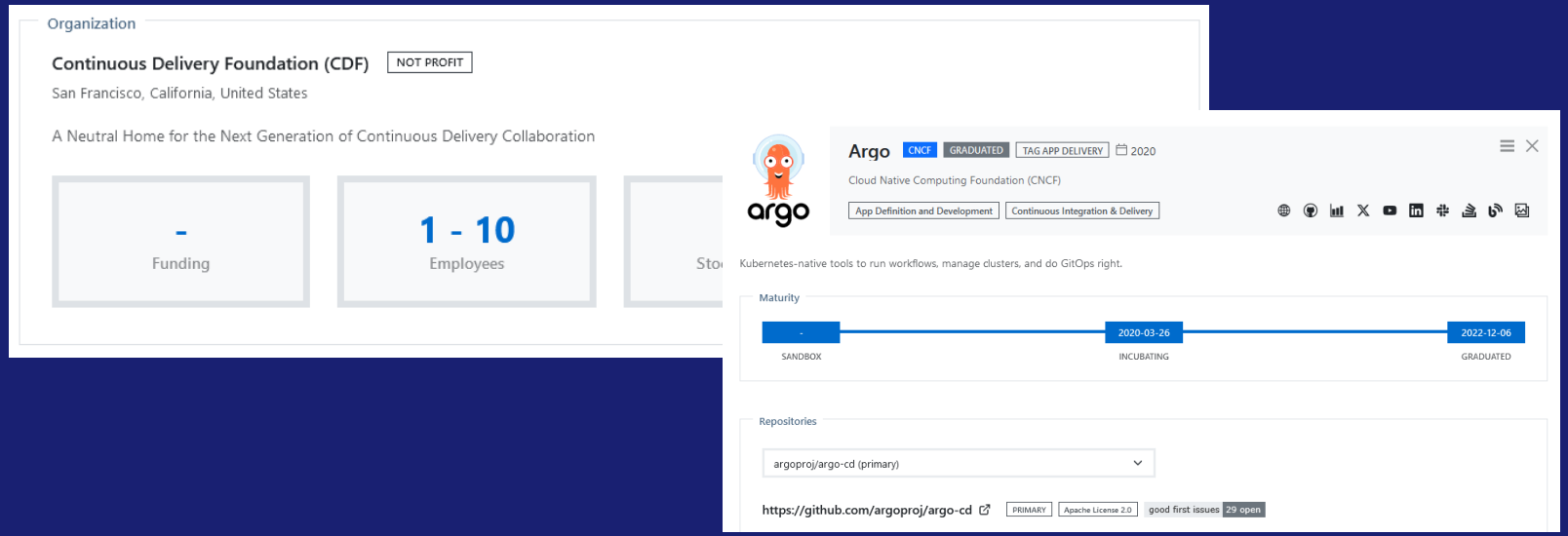
1. build
2. test
3. deploy

테크톤에서 모든 자원은 쿠버네티스와 동일하게 YAML기반으로 제공한다. 코드 기반으로 각각 작업들을 구성하며, 해당작업을 작업공간 및 파이프라인을 통해서 처리가 가능하다. 개발자가 손쉽게 YAML코드 기반으로 애플리케이션을 서비스로 배포가 가능하다.

TEKTON

테크톤은 CNCF사이트 소개가 되어 있다. NONE-PROFIT 오픈소스 프로젝트이다. 테크톤은 CNCF에서 아직 GRADUATED는 아니지만, CNCF에서 오픈소스로 보증하는 프로젝트이다.

ArgoCD는 CNCF에서 GRADUATED된 프로젝트이다.



TEKTON VS JENKINS

테크톤과 **젠킨스**는 기능 및 역할이 다르다. 젠킨스는 **플러그인 및 혹은 로컬 언어(Local Language)**를 사용해서 작업 순서를 구성한다.

젠킨스 Groovy라는 스크립트 명령어를 통해서 파이프라인 및 작업 수행이 가능하지만, 간단하게 파이프라인 및 자바 JDK와 같은 도구를 지원한다. 다른 의미로 테크톤 보다 더 제한적인 범위로 지원한다.

처음 시작하는 사용자에게는 학습 난이도가 낮은 테크톤 기반으로 구성을 권장하며, 추가적인 기능이 필요한 경우 플러그인 제작 및 설치가 별도로 필요하지 않는다. 또한, YAML문법을 사용하기 때문에 젠킨스의 Groovy언어 문법 및 구조체를 복잡하게 학습할 필요가 없다.

TEKTON VS JENKINS

젠킨스 Groovy 문법 형식은 아래와 같다. 하양식이 아닌, 선언 및 문법을 통해서 Groovy코드를 작성해야 한다. 물론, 선언 형식에 대해서도 학습이 필요하다. 그와 반대로 테크톤은 YAML형식으로 단순하게 선언 및 작성이 가능하다.

```
node {
    git url: 'https://github.com/jfrogdev/project-examples.git'
    def server = Artifactory.server "SERVER_ID"
    def downloadSpec =
            '''{
            "files": [
                {
                    "pattern": "libs-snapshot-local/*.zip",
                    "target": "dependencies/",
                    "props": "p1=v1;p2=v2"
                }
            ]
        }'''
```

TEKTON VS JENKINS

기능 비교하면 다음과 같다.

항목	Jenkins	Tekton
개발 주체	Jenkins Community (원래는 Kohsuke Kawaguchi)	Google + CD Foundation
아키텍처	모놀리식(Monolithic), 플러그인 기반	쿠버네이티브(Kubernetes Native), CRD 기반
설치/운영 방식	독립 실행형 서버 또는 컨테이너	Kubernetes 클러스터 내에서 실행되는 파이프라인 리소스
파이프라인 정의	Groovy 기반 DSL (Jenkinsfile)	YAML 기반 CRD (Pipeline, Task 등)
확장성	플러그인을 통한 확장 가능, 그러나 종속성 충돌 위험	Kubernetes 자원을 활용한 확장 (Pod, PVC 등)
CI/CD 목적성	CI 중심에서 출발 → CD는 추가 확장	CD를 염두에 둔 구조, 특히 GitOps 및 배포 중심

TEKTON VS JENKINS

앞에 내용 계속 이어서…

항목	Jenkins	Tekton
보안	많은 플러그인은 권한 문제 발생 가능	Kubernetes의 RBAC 기반 보안 정책 활용 가능
사용성	GUI 친화적, 설정 직관적	CLI/yaml 중심으로 비교적 학습 곡선 있음
에이전트 실행	Jenkins 노드로 실행 (Master-Agent 구조)	각 Task가 Pod로 실행, 병렬 처리 용이
상태 저장	Jenkins 자체 DB 사용 (또는 외부 DB)	Kubernetes 리소스로 상태 관리 (ConfigMap, PVC 등)
커뮤니티 및 문서	매우 활발하고 오래된 커뮤니티	상대적으로 작지만 CNCF 지원 및 문서 지속 증가
적합한 환경	소규모~대규모 전통적인 CI/CD 환경	클라우드 네이티브, 쿠버네티스 중심의 CD 환경
예시 사용처	GitHub Actions과 연동한 기존 서버 중심 파이프라인	GKE, OpenShift 등 쿠버네티스 플랫폼 기반 자동화 파이프라인

TEKTON VS JENKINS

테크톤과 젠킨스의 제일 큰 차이점은 동작 방식이다.

젠킨스는 쿠버네티스와 통합된 형태가 아니다. 본 기능은 자바 빌드용도 사용하다가 쿠버네티스와 통합된 상태이다. 그러한 이유로 젠킨스는 완벽하게 쿠버네티스와 통합이 되지 않는다.

테크톤은 기본적으로 쿠버네티스 클러스터 기반으로 구성 및 작성이 되었다. 젠킨스보다 손쉽게 사용하며, 별도로 관리 모듈을 만들지 않고 사용이 가능하다.

https://www.jenkins.io/blog/2019/01/08/mpl-modular-pipeline-library/

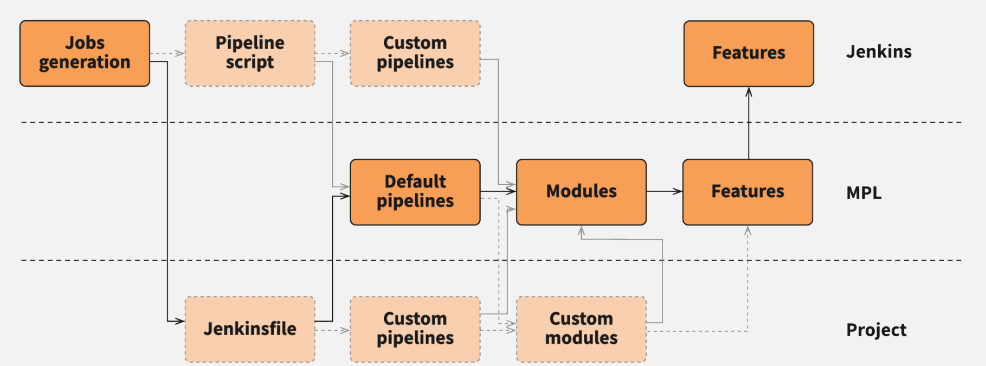
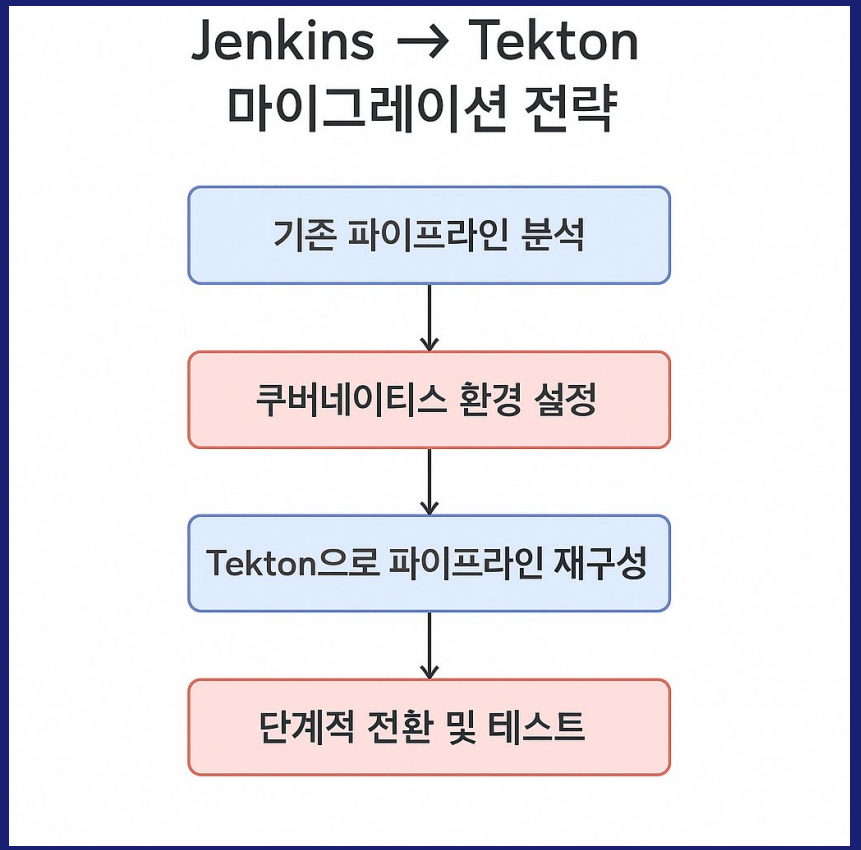
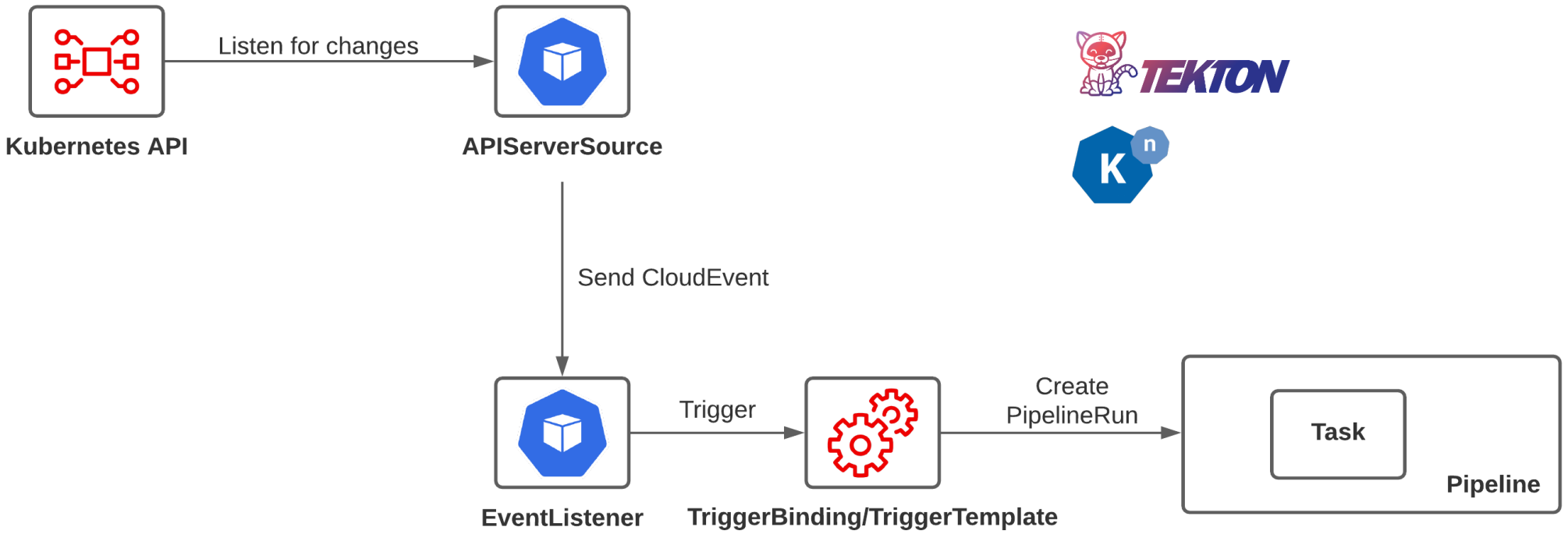


Fig 2. The MPL process

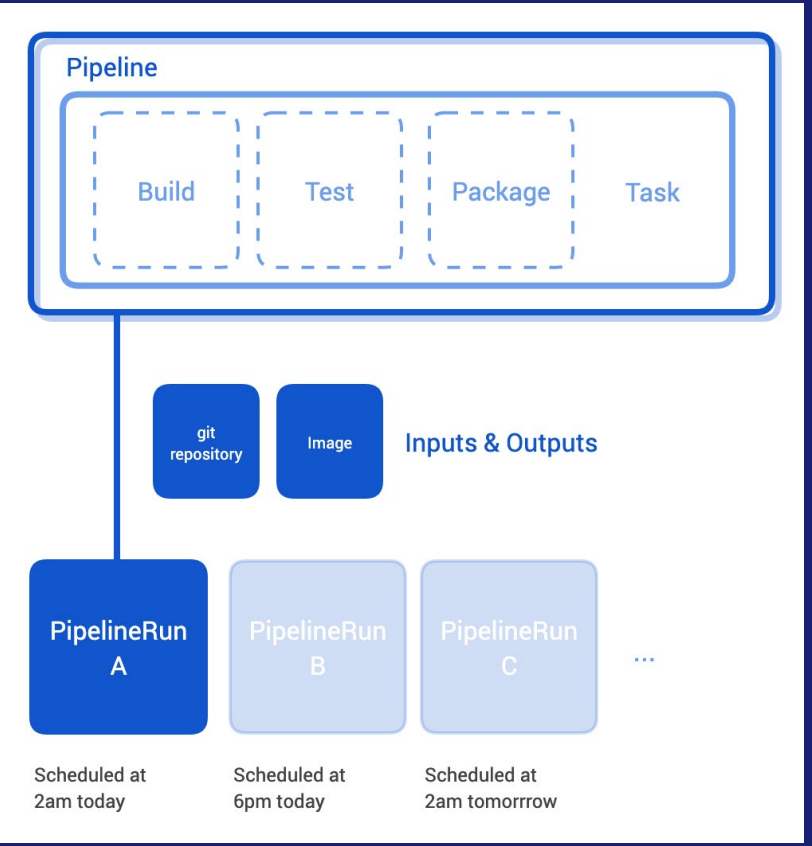
마이그레이션


Jenkins → Tekton
마이그레이션 전략
기존 파이프라인 분석
쿠버네이티스 환경 설정
Tekton으로 파이프라인 재구성
단계적 전환 및 테스트

IN KUBERNETES



TEKTON PIPELINE



클라우드 네이티브에서 CI/CD

CI/CD

Jenkins/Tekton

클라우드 네이티브

클라우드 네이티브는 수많은 컴퓨팅 개념 중 하나이다. 클라우드 네이티브에 제일 큰 차이점은 대다수 오픈소스 기반으로 개념들이 구성이 되어 있으며,

이를 기반으로 Cloud Native Computing Foundation(CNCF)라는 파운데이션을 만들었다. 테크톤은 CD영역에서 포함이 되며, 이를 통해서 K-Native기반의 자동화를 구성한다.

CNCF/K-Native

많이 혼돈 Cloud Native와 K-Native이다. K-Native는 쿠버네티스 기반으로 서버리스 서비스 구성이 목적이다.

K-Native는 컨테이너 기반으로 다음과 목적을 가지고 있다. 클라우드 네이티브는 국가 혹은 기업마다 목표 및 목적이 다르다.

- **Simpler Abstraction:** YAML기반으로 CRD구성이 가능하다.
- **Auto Scaling:** 오토 스케일링 기반으로 0부터 확장 혹은 축소가 가능하다.
- **Progressive Rollouts:** 롤 아웃 상태를 전략에 따라서 구성이 가능하다.
- **Event Integration:** 모든 소스의 이벤트에 대해서 핸들링이 가능하다.
- **Handle Events:** 작업 이벤트를 트리거를 통해서 관리한다.
- **Pluggable:** 쿠버네티스에 확장 기능 추가가 가능하다.

CR/CRD

CR

보통 Custom Resource라고 부르며, 쿠버네티스 API를 통해서 생성이 된다.

대표적인 자원은 `Pod`, `ConfigMap` 그리고 `Secrets`이 있다. CR자원은 별도로 설치 및 구성이 필요 없으며 쿠버네티스 클러스터에서 지원 및 제공한다.

CRD

CRD Custom Resource Definition의 약자이다. 쿠버네티스 네이티브 API가 아닌, 추가적으로 설치한 애플리케이션에 자원을 생성 시, CRD를 통해서 구성한다. 테크톤도 CRD를 통해서 자원을 생성 및 구성한다.

테크톤 자원

Stepping

Tasks

STEP

기본자원

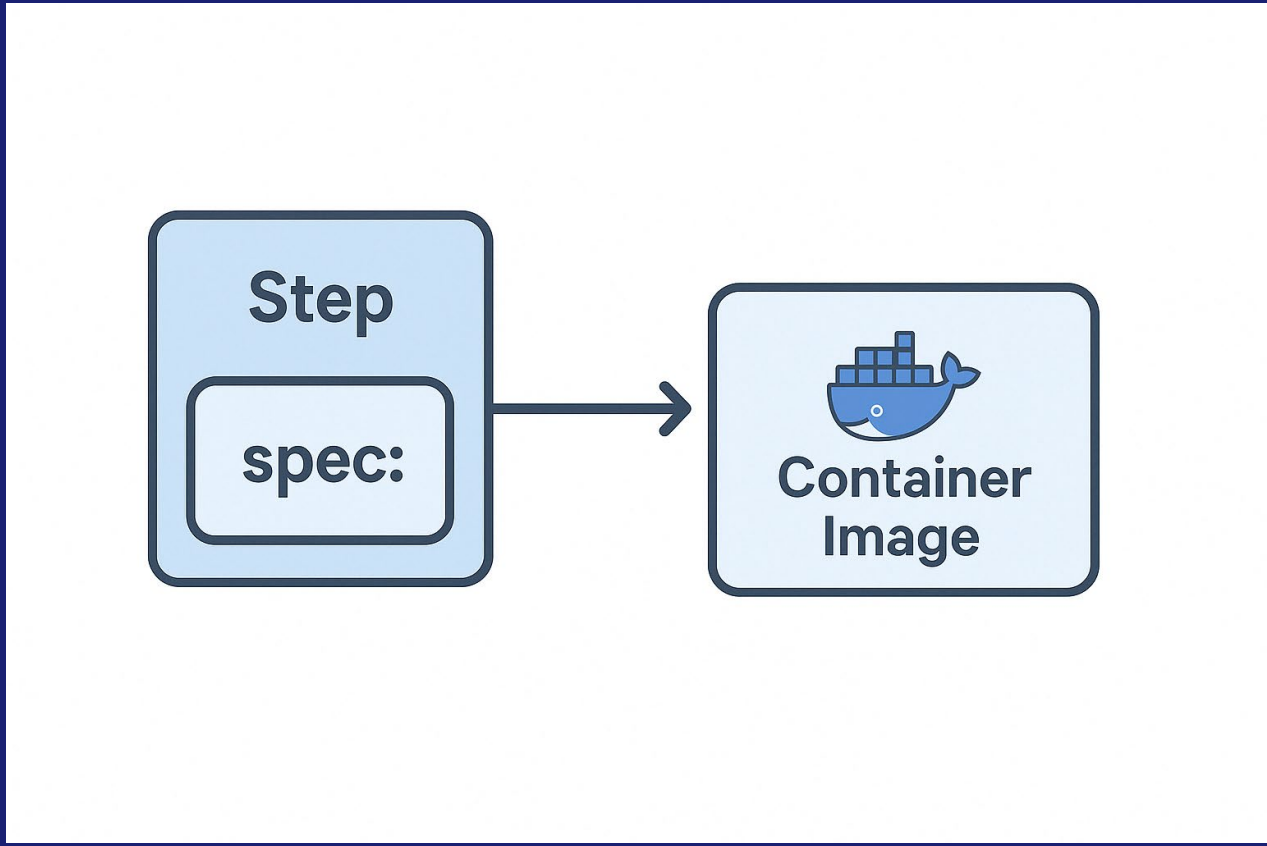
STEP

단계(Step)은 제일 기본적인 유닛이며, 이를 기반으로 파이프라인 구성이 된다. 최소 한 개의 작업이라도 단계에 구성이 되어야 한다. 이를 통해서 CI/CD 작업 수행 단계를 효율적으로 구성 및 운영이 가능하다.

각 단계에서는 명령어를 통해서 작업이 수행이 되는데, 예를 들어서 이미지 빌드를 위해서 소스코드를 내려받기 후 컴파일 과정이 필요하면, 이 부분을 단계에 명시한다. 각 단계에는 일반적으로 명령어를 통해서 어떠한 작업을 수행할지 명시한다. 자세한 내용은 뒤에서 더 다룬다.

```
spec:
  steps:
    - image: quay.io/centos/centos
      command:
        - /bin/bash
        - -c
        - echo "Hello World"
```

STEP



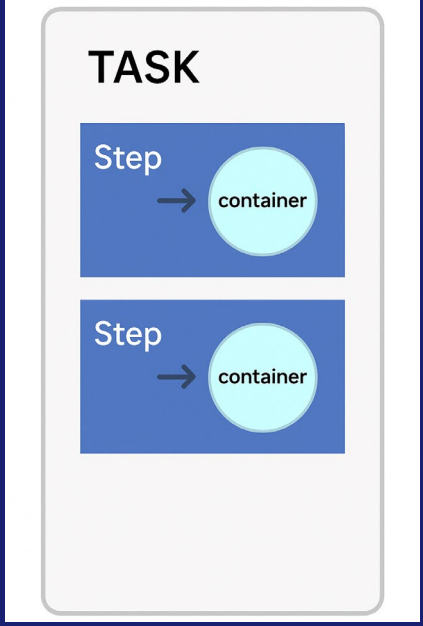
TASK, PIPELINE

기본 자원

TASK

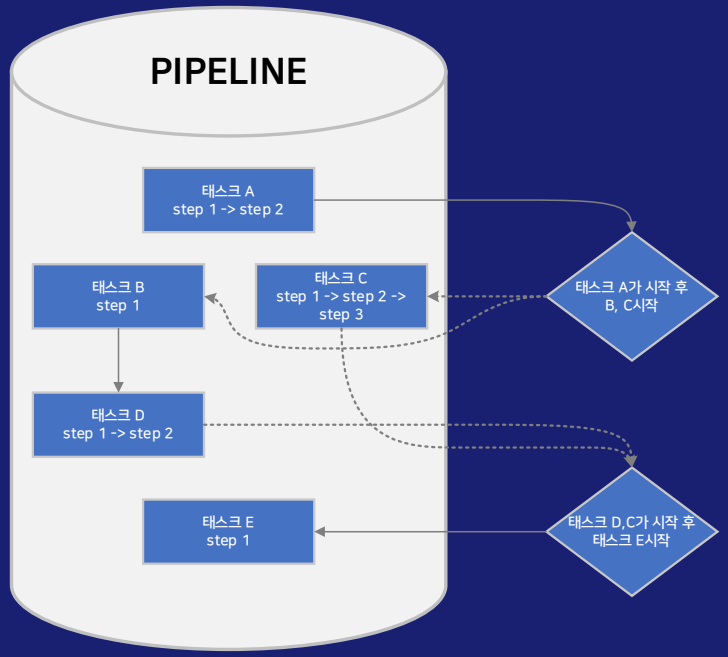
파이프 라인은 **작업의 순서를 정하는 자원**이다.

쉽게 생각하면 GW-BASIC처럼, 각각 작업에 순서를 걸어서 순차적으로 작업을 수행한다. 파이프 라인은 추상적인 자원이며, 이 자원은 기본적으로 태스크를 감싸고 있다.



PIPELINE

모든 **스텝** 및 **태스크**에는 작업 순서가 있기 때문에, 작성된 순서대로 작업이 하나씩 수행이 된다. 이러한 형태를 가지고 있는 자동화 도구는 **앤서블/솔트**와 같은 도구들이 있으며, 이들은 테크톤과 동일하게 YAML기반으로 작업을 작성 및 구성하게 된다.



PIPELINE

아래는 파이프라인 생성 YAML코드이다. 파이프라인은 반드시 어떤 작업(task)를 수행할지 명시해야 한다.

```
# vim first-pipeline.yaml
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: first-pipeline
spec:
  tasks:
    - name: first
      taskRef:
        name: first-task
```

TASK

태스크(tasks)는 파이프라인에 안에서는 작업에 대한 격리를 한다.

여기서 말하는 **작업은 단계(step)**를 이야기 하며, 각각 단계들은 태스크에서 수행이 된다. 각 태스크는 단계들의 **순서(sequence)**를 가지고 있으며, 이를 통해서 하나의 작업을 통해서 모든 작업을 수행한다.

쿠버네티스 클러스터에서 Pod를 생성하기 위해서 이미지 작업 및 Pod생성과 같은 과정이 필요하다. 이러한 부분을 각각 단계로 구성하여 작업을 수행 후, 최종적으로 API를 통해서 Pod생성 및 PV/PVC와 같은 자원을 구성 및 연결을 수행한다.

이러한 **작업 단계를 모아서 동작하는 영역이 파이프라인**이며, 작업(task)는 단계(step)에 명시된 작업들을 분리 및 격리하여 수행 할 수 있도록 한다. 작업을 구성하면, 이들은 **파이프 라인을** 통해서 구성 및 실행이 된다.

TASK YAML

아래는 간단하게 태스크(task)를 생성하는 YAML이다.

```
# vi first-task.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: first-task
spec:
  steps:
  - name: first-task
    image: quay.io/centos/centos
    command:
      - echo "hello world"
```

생성 자원 확인

생성된 자원은 아래와 같이 명령어로 확인이 가능하다.

```
# tkn pipeline list
# tkn task list
# tkn pipeline start first-pipeline
# tkn pipeline list
NAME             AGE              LAST RUN                    STARTED          DURATION   STATUS
first-pipeline   14 minutes ago   first-pipeline-run-86x7v   19 seconds ago   ---        Running
```

기본 자원

Stepping

Task

Pipe

STEP

단계(스탭)은 특정 작업을 수행하는 부분이다. 단계는 다음과 같이 작업을 수행한다.

1. 소스코드 내려받기
2. 컴파일 및 명령어 수행
3. 프로그램 패키징

총 3단계를 통해서 작업을 한다. 이미지가 비공개 저장소에 있는 경우, `ImagePullSecret`를 통해서 내려받기가 가능하다. 작성 방법은 아래와 같이 작성이 가능하다.

STEP

아래와 같이 작업을 생성한다.

```
# vi demo-task-1.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: demo-task-1
spec:
  steps:
    - image: quay.io/centos/centos:stream9
      command:
        - /bin/bash
        - -c
        - echo "Hello World"
```

STEP

생성한 **demo-task-1**를 아래와 같이 생성한다.

```
# kubectl apply -f demo-task-1.yaml
# tkn task ls
NAME           DESCRIPTION   AGE
demo-task-1                  7 seconds ago
hello                        3 minutes ago
# tkn task start demo-task-1
# tkn task start demo-task-1 --showlog
```

STEP FOR MULTI STEP

한 개 이상의 단계가 있는 경우, 아래와 같이 작성이 가능하다.

```
# vi demo-multi-step.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: demo-multi-step
spec:
  steps:
    - name: first
      image: quay.io/centos/centos:stream9
      command:
        - /bin/bash
        - -c
        - echo "First step"
```

STEP FOR MULTI STEP

```
    - name: second
      image: quay.io/centos/centos:stream9
      command:
        - /bin/bash
        - -c
        - echo "Second step"
# tkn task list
# tkn task start demo-multi-step
# tkn taskrun logs demo-multi-step-run-hlvgx -f
```

STEP IMAGE

단계 구성 시, 사용하는 컨테이너 이미지는 사용자가 원하는 이미지로 변경이 가능하다. 여기서는 일반적으로 많이 사용하는 centos이미지 기반으로 구성하였다. 이미지는 최소 한 개가 구성 및 선언이 되어야 한다.

문법은 아래와 같이 선언이 된다.

```
  - name: second
    image: quay.io/centos/centos
```

현재 연습 예제에서는 centos기반으로 테스트를 수행 및 진행한다. 만약, 다른 배포판 사용을 원하는 경우, 다른 컨테이너 이미지를 사용하여도 된다.

STEP 실행 및 상태 확인

올바르게 실행이 되었는지 아래 명령어로 확인한다.

```
# tkn task list
# tkn task start demo-multi-step --showlog
# tkn task start demo-multi-step --showlog --no-color
# kubectl get tasks
# kubectl get taskruns
```

STEP 스크립트 사용

다중 명령어를 실행하거나 혹은 스크립트를 실행하는 경우, 아래와 같이 처리가 가능하다.

```
- name: step-in-script
  image: quay.io/centos/centos:stream9
  script: |
    #!/usr/bin/env bash
    echo "Install a package"
    dnf install httpd -y
    dnf clean all
    echo "All commands ran!"
```

STEP IN SCRIPT

위의 내용을 코드로 변경하면 아래와 같다.

```
$ vi demo-step-script.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: demo-step-script
spec:
```

STEP IN SCRIPT

위의 내용 계속 이어서…

```
    steps:
    - image: quay.io/centos/centos:stream9
      script: |
        #!/usr/bin/env bash
        echo "Install a package"
        dnf install httpd -y
        dnf clean all
        echo "All commands ran!"
```

STEP IN SCRIPT

아래 명령어 적용 및 확인한다.

```
# kubectl apply -f demo-step-script.yaml
# tkn task list
# tkn task start demo-step-script --showlog
```

TASK PARAMETER

앞에서 간단하게 단계와 작업을 동시에 작성 및 실행 하였다. 하지만, 매번 작업을 구성할 때마다 단계에 들어가는 값을 변경할 수 없기 때문에, 재사용을 위해서 파라메터 형식으로 변경한다.

파라메터, 즉 변수 형태로 하는 경우, "param"항목에서만 변경하면서 사용이 가능하다. 좀 더 효율적으로 운영 및 구성이 가능하다.

위와 같이 값의 이름, 그리고 유형을 적어주면, 쉘 텍스트 입력 받는 형식과 비슷하게 값을 읽어온다. 예로, 앞에서 사용하였던 명령어 실행을 예로 든다.

```
params:
  - name: username
    type: string
```

TASK PARAMETER

아래와 같이 작성하면, 변수를 통해서 좀 더 효율적으로 코드 사용이 가능하다.

```
command:
  - /bin/bash
  - -c
  - echo "Hello $(params.who)
```

TASK PARAMETER

위의 두 개 예제를 가지고 다음과 같이 테크톤 작업형식으로 작성이 가능하다. 아래 슬라이드처럼 파일을 작성한다.

```
# vi demo-task-param.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: demo-task-param
spec:
  params:
  - name: username
    type: string
```

TASK PARAMETER

위의 내용 계속 이어서…

```
steps:
  - image: quay.io/centos/centos:stream9
    command:
      - /bin/bash
      - -c
      - echo "Hello $(params.username)"
```

TASK PARAMETER

적용 후, 실행하면 다음과 같이 실행 및 확인이 된다.

```
# kubectl -f demo-task-param.yaml
# tkn task list
# tkn task start demo-task-param --showlog
? Value for param `username` of type `string`? tang
TaskRun started: demo-task-param-run-9rpcz
Waiting for logs to be available...
[unnamed-0] Hello tang
# tkn task start demo-task-param --showlog -p username=tang
```

TASK ARRAY PARAMETER

파라메터를 배열로 처리가 가능하다. 다만, 좀 더 복잡하게 코드를 구성해야 한다. 기존 코드에 아래와 같이 파일 이름을 변경 후 작성한다.

```
# vi demo-task-param-array.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: demo-task-param-array
spec:
  params:
  - name: users
    type: array
```

TASK ARRAY PARAMETER

파라메터를 배열로 처리가 가능하다. 다만, 좀 더 복잡하게 코드를 구성해야 한다. 기존 코드에 아래와 같이 파일 이름을 변경 후 작성한다.

```
  steps:
  - name: list-users
    image: quay.io/centos/centos:stream9
    args:
    - $(params.users[*])
    command:
    - /bin/bash
    - -c
    - for ((i=1;i≤$#;i++)); do echo "$#" "$i" "${!i}"; done
```

TASK PARAMETER ARRAY

테크톤에 작업 등록 후 작업을 수행한다.

```
# kubectl apply -f demo-task-param-array.yaml
# tkn task list
# tkn task start demo-task-param-array --showlog --use-param-defaults
test1, test2, test3
```

TASK DEFAULT PARAMETER

파라메터에 기본값 설정이 필요한 경우 default라는 변수 키워드를 사용한다. 미리 사용할 기본값을 미리 작성한

```
# vi demo-task-param-default.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: demo-task-param-default
spec:
  params:
  - name: users
    type: array
    default:
      - test1
      - test2
      - test3
```

TASK DEFAULT PARAMETER

앞에 내용 계속 이어서…

```
  steps:
  - name: list-users
    image: quay.io/centos/centos:stream9
    args:
    - $(params.users[*])
    command:
    - /bin/bash
    - -c
    - for ((i=0;i≤$#;i++)); do echo "$#" "$i" "${!i}"; done
```

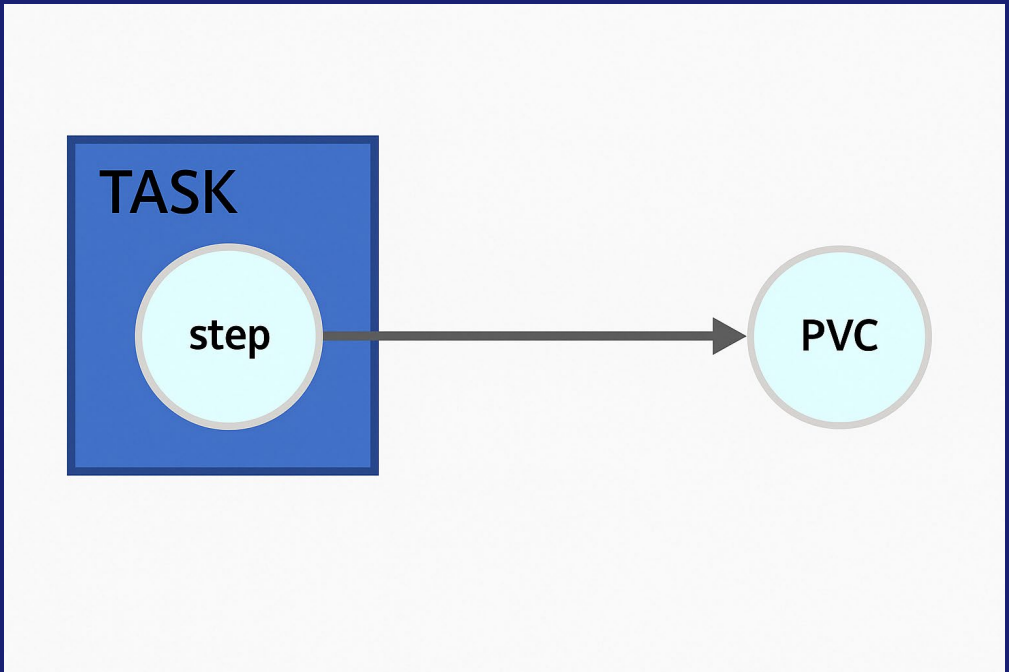
TASK DEFAULT PARAMETER

기본 변수 값을 쿠버네티스에 등록 후 실행한다.

```
# kubectl apply -f demo-task-param-default.yaml
# tkn task list
# tkn task start demo-task-param-default --use-param-defaults --showlog
```

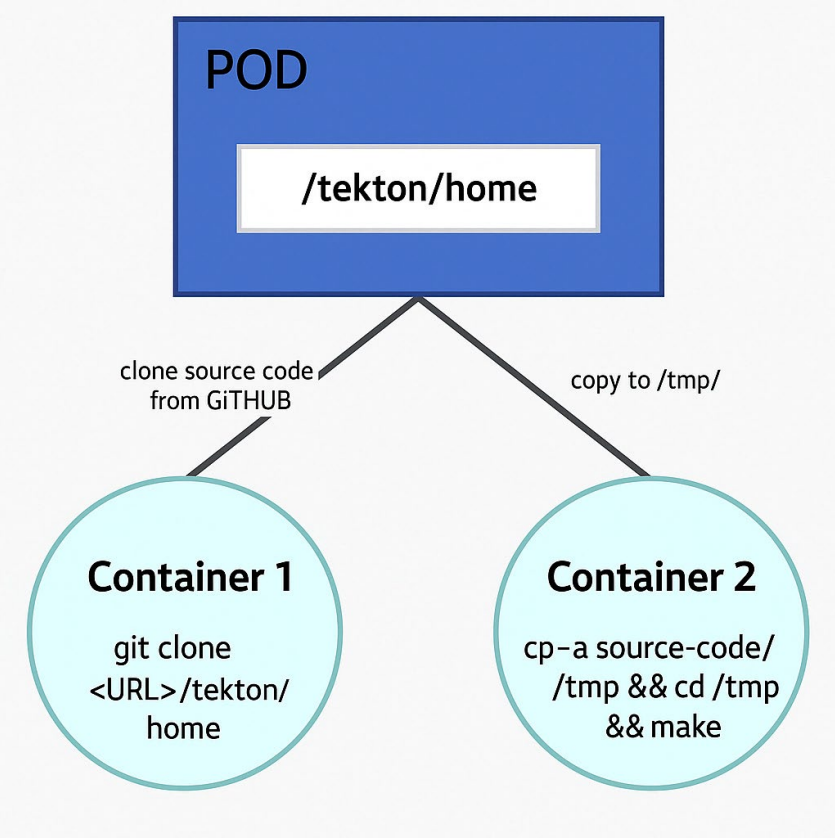
공유 데이터

테크톤이 동작하면 컨테이너가 하나의 포드(Pod)에서 동작하기 때문에, 데이터 공유가 가능하다. 데이터를 공유 하기 위해서 쿠버네티스의 PV/PVC를 통해서 공유가 가능하다.



공유 홈

테크톤의 고유한 디렉터리를 사용하여 POD에서 컨테이너간 데이터 공유가 가능하다.



기본 자원 소개

share directory

result

공유 데이터

테크톤은 특정 디렉터리 위치에 데이터를 임시적 혹은 영구적으로 공유가 가능하다. 테크톤 컨테이너가 실행 및 동작하면 다음과 같은 디렉터리를 생성하여 테크톤에 연결한다. 예약된 디렉터리 및 변수는 다음과 같다.

공유 데이터

공유 데이터 위치는 다음과 같다.

디렉터리 경로	용도 설명	접근 주체
`/workspace`	Task 및 Step 간 공유 작업 공간 (Pipeline level에서 연결된 워크스페이스)	사용자 정의
`/tekton/home`	컨테이너의 기본 작업 디렉터리 (HOME). git clone 등의 작업 경로로 사용	Tekton Controller
`/tekton/creds`	인증 정보가 저장되는 디렉터리. git, docker 등 Secret/ServiceAccount 기반 인증 정보 주입 위치	Tekton 시스템
`/tekton/results`	Step 실행 결과(예: 출력 값)를 저장하는 디렉터리. results 필드와 연동됨	Tekton 시스템
`/tekton/run`	TaskRun 실행 정보 (metadata 등)가 저장되는 임시 경로 (일부 구현체에서 사용)	Tekton 내부용
`/tekton/tools`	일부 도구/실행 파일들(executable)이나 CLI 바이너리가 임시로 설치되는 위치	Tekton 시스템
`/tekton/tmp`	임시 파일 저장소. 스크립트 작업 중 중간 파일 생성 시 활용됨	각 Step

공유 데이터(HOME) 1

간단하게 기본 테크톤 공유 디렉터리를 생성한다. 아래 처럼 예제 파일을 구성 및 생성한다.

```
# vi tekton-share-home.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: tekton-share-home
spec:
  steps:
  - name: write
    image: quay.io/centos/centos:stream9
    script: |
      echo "location is $(pwd)"
      echo ~
      echo ~tekton
      touch /tekton/home/tkn-message.txt
```

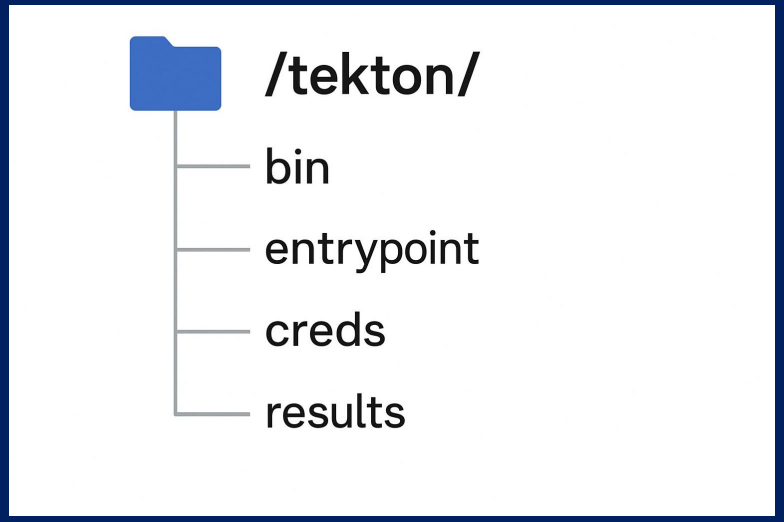
공유 데이터(HOME) 2

테크톤 홈 디렉터리를 확인하기 위해서 몇가지 작업을 추가한다.

```
- name: read
   image: quay.io/centos/centos:stream9
   script: |
     echo "Listing /tekton/home"
     ls -lR /tekton/home
- name: tree
   image: quay.io/centos/centos:stream9
   script: |
     dnf install tree -y
     tree -L 2 /tekton/
     tree -L 2 /workspace/
```

공유 데이터(HOME) 3

```
# kubectl apply -f tekton-share-home.yaml
# tkn task list
# tkn task start tekton-share-home --showlog
```



참고

위의 기능은 최신 테크톤 버전에서는 동작하지 않는다. 자세한 정보는 아래 링크를 참고한다.

https://github.com/tektoncd/pipeline/pull/3878

테크톤 포드에서는 더 이상 앞에서 언급한 디렉터리 **"/tekton/home", "/workspace"**는 변수를 통해서 접근이 가능하다.

https://github.com/tektoncd/pipeline/blob/main/docs/variables.md

공유 데이터(RESULT)

특정 작업이 수행 후, 결과값에 대해서 저장을 하기 위해서 **`result`**라는 지시자 사용이 가능하다. 결과(result)를 사용하면, 완료된 작업의 내용을 **`/tekton/results`**에서 접근 및 사용이 가능하다. 결과(result)는 쿠버네티스의 PV/PVC를 사용하지 않는다.

위의 내용들은 앞서 이야기한 테크톤 디렉터리에서 불러와서 화면에 출력한다. 이전에 사용한 내용을 응용하여 **결과(result)**를 사용하도록 한다.

TEKTON RESUTL

```
$ vi tekton-result.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: tekton-result
spec:
  results:
    - name: welcome
      description: welcome message
```

TEKTON RESUTL

앞에 내용 계속 이어서…

```
  steps:
    - name: write
      image: quay.io/centos/centos:stream9
      command:
        - /bin/bash
      args:
        - "-c"
        - echo "Welcome to Tekton world" > $(results.welcome.path)
```

TEKTON RESUTL

앞에 내용 계속 이어서…

```
    - name: read
      image: quay.io/centos/centos:stream9
      command:
        - /bin/bash
      args:
        - "-c"
        - cat $(results.welcome.path)
$ kubectl apply -f tekton-result.yaml
$ tkn task list
$ tkn task start tekton-result --showlog
```

KUBERNETES VOLUME

쿠버네티스에서 사용하는 볼륨 자원, `secret, configmap` 그리고 `Persistent Volume, Persistent Volume Claim` 전부 사용이 가능하다. 여기에서는 간단하게 쿠버네티스 볼륨 자원들을 테크톤에서 활용해보도록 한다.

더 많은 자원은 뒤에서 추후 활용 및 다루기로 한다.

1. `secret`
2. `configmap`
3. PV/PVC

TEKTON CONFIGMAP

테크톤에서 ConfigMap에 접근 할 수 있도록 아래와 같이 테스트용 CM자원을 생성한다.

```
# vi kubernetes-demo-configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: user-list
data:
  user1: helix
  user2: tang
  user3: suse
```

TEKTON CONFIGMAP

테크톤에서 CM에 접근이 가능한지 아래 예제로 테스트한다.

```
# vi tekton-demo-configmap.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: tekton-demo-configmap
spec:
  volumes:
    - name: users
      configmap:
        name: user-list
```

TEKTON CONFIGMAP

```
steps:
  - name: username-list
    image: quay.io/centos/centos
    volumeMounts:
      - name: users
        mountPath: /var/username-list
    script: |
      echo "$(cat /var/username-list/user1) is normal user"
      echo "$(cat /var/username-list/user2) is root user"
      echo "$(cat /var/username-list/user3) is admin user"
```

TEKTON CONFIGMAP(CM)

```
# kubectl apply -f kubernetes-demo-configmap.yaml
# kubectl apply -f tekton-demo-configmap.yaml
# tkn task list
# tkn task start tekton-demo-configmap --showlog
```

디버깅

테크톤

DIGGING and DIGGING

디버깅

디버깅 위한 명령어는 다음과 같이 지원 및 제공한다.

```
# tkn task start tekton-configmap --showlog
# kubectl run --image=quay.io/centos/centos:stream9
# kubectl get taskrun tekton-configmap-run-<ID> -o yaml
# tkn taskrun list
```

디버깅

아래와 같이 YAML파일 생성 후 다음과 같이 명령어를 통해서 디버깅을 시도한다.

```
# vi failforfail.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: failforfail
spec:
  steps:
  - image: quay.io/dollar/dollar
    command:
      - echo "Hello failure"
```

디버깅

아래와 같이 YAML파일 생성 후 다음과 같이 명령어를 통해서 디버깅을 시도한다.

```
# kubectl apply -f failforfail.yaml
# tkn task start failforfail --showlog
# kubectl get tr
# kubectl get tr failforfail-run-<ID> -o yaml
```

워크스페이스

데이터 공유

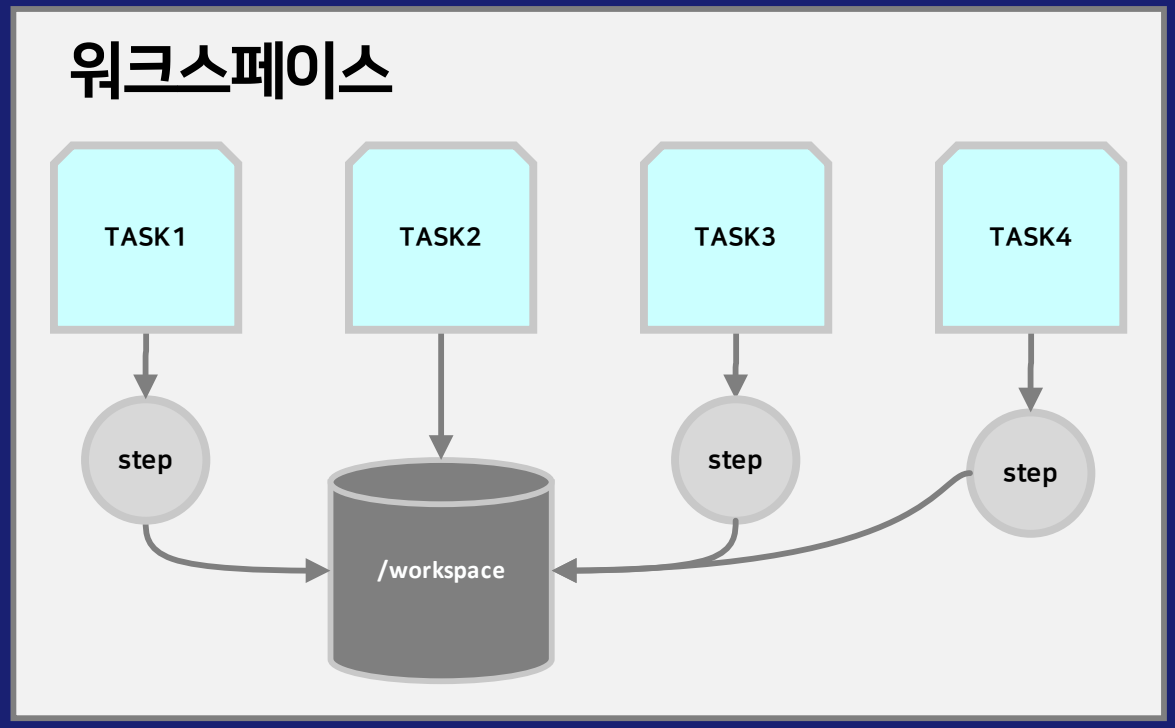
워크스페이스

소개

2025-04-23

워크스페이스

워크 스페이스는 작업에서 생성 된 데이터를 저장한다. 저장은 POD의 **특정 공유 볼륨** 혹은 PVC를 통해서 컨테이너 데이터를 저장한다.



워크스페이스

워크스페이스와 비슷한 **결과(results)**경우에는 각 단계에서 발생한 결과을 저장할 수 있지만, 여기에는 최대 크기가 4k(4096bytes)에 대한 제한이 있다.

https://github.com/tektoncd/pipeline/issues/4060

상태	이유	설명
False	TaskRunResultTooLarge	**"TaskRun"**에서 크기가 4096바이트를 넘어가는 경우, 이 옵션을 통해서 허용한다.

지원하는 자원 형태

워크스페이스에서 지원하는 저장소 형태는 다음과 같다.

- emptyDir
- ConfigMap
- Secret

쿠버네티스에서 제공하는 볼륨 `Persistent Volume, Persistent Volume Claim`를 지원한다.

PV/PVC를 사용한 경우, 앞에서 학습하였던 `finally`를 사용하여 종료 시, 내부 데이터를 제거해야 한다. 그렇지 않는 경우, 기존 데이터가 남아 있기 때문에, 다른 컨테이너 사용자에게 기존 내용이 노출이 될 수도 있다.

- **`PersistentVolume`**
- **`PersistentVolumeClaim`**

문법 선언

워크스페이스를 선언하기 위해서 다음과 같이 YAML파일에 작성한다.

workspaces: 부분은 워크스페이스에서 생성한 이름을 보통 적는다.

명시한 이름은 `/workspaces/<WORKSPACE_DIR_NAME>/`으로 생성이 된다. 예를 들어서 위에서 작성한 `source`이름으로 작성한 경우 `/workspace/source`으로 디렉터리 생성이 된다.

workingDir:은 명령어가 수행할 작업 위치를 명시하며, 보통 테크톤 변수를 통해서 선언이 가능하다.

```
  workspaces:
     - name: source
  steps:
     - name: clone
       image: quay.io/centos/centos:stream9
       workingDir: $(workspaces.source.path)
```

저장소 형식

워크스페이스

저장소

앞에서 이야기 하였지만, 워크스페이스에서는 쿠버네티스에서 사용하는 저장소 유형을 전부 사용이 가능하다. 하지만, 데이터가 많이 발생하는 경우 일반적으로 다음과 같이 사용한다.

1. emptyDir
2. PV/PVC
3. StorageClass(권장)

가급적이면 스토리지 클래스(StorageClass)으로 구성한다. 기본적으로 **NFS CSI(Container Storage Interface)**가 구성이 안되어 있다는 조건으로 `emptyDir:`를 사용한다.

저장소 사용예

앞에서 사용했던 예제 기반으로 설명하면 다음과 같다.

```
  workspaces:
     - name: source
  steps:
     - name: clone
       image: quay.io/centos/centos:stream9
       workingDir: $(workspaces.source.path)
```

별도로 명시가 되지 않는 경우, `emptyDir:`으로 구성이 되며, 만약 PVC를 통해서 저장소를 전달하는 경우 다음처럼 작성한다. 미리 구성한 PVC의 이름을 작성하면 된다.

```
  workspaces:
     - name: output
       persistentVolumeClaim:
         claimName: tekton-tutorial-sources
```

첫 워크스페이스 생성

워크스페이스

간단한 워크스페이스 생성

예를 들어서 깃헙에서 특정 소스코드를 복제 후, 실행하는 작업이 필요한 경우, 이때 워크스페이스가 필요하다. 간단하게 만든 워크스페이스는 다음과 같은 작업을 수행한다.

1. 깃헙에서 소스코드, **`duststack-osp-auto`**를 복제한다.
2. **앤서블**이 실행이 가능하도록 패키지를 설치한다.(RPM, PIP)
3. 복제한 코드를 앤서블 명령어로 간단하게 코드 검증 및 확인한다.

간단한 워크스페이스 예제

저장소는 간단하게 **emptyDir**를 사용하며, 이를 통해서 소스코드를 저장 후 앤서블을 실행한다.

```
# vi demo-workspace-clone-github.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: demo-workspace-clone-github
spec:
  params:
    - name: repo
      type: string
      description: Git Repository clone from github
      default: https://github.com/tangt64/duststack-osp-auto
  workspaces:
    - name: source
```

간단한 워크스페이스 예제

앞에 내용 계속 이어서…

```
  steps:
    - name: clone
      image: quay.io/centos/centos:stream9
      workingDir: $(workspaces.source.path)
      script: |
        dnf install git -y
        git clone -v $(params.repo) $(workspaces.source.path)
    - name: list
      image: quay.io/centos/centos:stream9
      workingDir: $(workspaces.source.path)
      script: |
        ls -l $(workspaces.source.path)
```

간단한 워크스페이스 예제

앞에 내용 계속 이어서…

```
    - name: ansible
      image: quay.io/centos/centos:stream9
      workingDir: $(workspaces.source.path)/duststack-osp-auto
      script: |
        dnf install ansible -y
        ansible-playbook --syntax-check playbooks/osp-install.yaml
```

실행

다음 명령어로 실행한다.

```
# kubectl apply -f demo-workspace-clone-github.yaml
# tkn task start demo-clone-GitHub --showlog
? Value for param `repo` of type `string`
? Please give specifications for the workspace: source
? Name for the workspace : source
? Value of the Sub Path :
? Type of the Workspace : emptyDir
? Type of EmptyDir :
```

워크스페이스에서 작업 실행

워크스페이스

TaskRun

태스크 실행(TaskRun)은 특정 작업을 실행 시 사용한다.

예를 들어서 앞에서 만든 작업, **demo-clone-github**를 **tr(TaskRun)**를 통해서 실행한다. 이전에는 작업을 실행 시 `tkn task start <TASK_ID>`를 통해서 실행하였다. 태스크 실행을 사용하는 경우, 코드 기반으로 작업 수행이 가능하다.

```
# tkn task start demo-clone-GitHub --showlog
```

TaskRun

다음처럼 변경 및 실행이 가능하다.

```
# vi demo-tr-clone-github.yaml
apiVersion: tekton.dev/v1beta1
kind: TaskRun
metadata:
  generateName: demo-tr-clone-github
spec:
  workspaces:
    - name: source
      emptyDir: {}
  taskRef:
    name: demo-workspace-clone-github
```

TaskRun

아래 명령어로 실행한다.

```
# kubectl apply -f demo-tr-clone-github.yaml
# kubectl create -f demo-tr-clone-github.yaml
# tkn taskrun list
# tkn taskrun start demo-tr-clone-github
# tkn taskrun logs demo-tr-clone-github
```

파이프라인 추가

워크스페이스

파이프라인에 워크스페이스

파이프라인에 워크스페이스를 추가해서 사용하면, 손쉽게 반복적으로 사용이 가능하다.

기존에 사용하였던 깃헙 코드를 조금 더 단순하게 만들어서 테스트를 한다. 한 개의 파일에 파이프라인, 태스크 두 개의 내용이 작성이 된다.

```
# vi demo-workspace-clone-github-pipeline.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: demo-workspace-clone-github-pipeline
spec:
  params:
    - name: repo
      type: string
      description: Git Repository clone from github
      default: https://github.com/tangt64/duststack-osp-auto
```

파이프라인에 워크스페이스

앞에 내용 계속 이어서…

```
  workspaces:
    - name: source
  steps:
    - name: clone
      image: quay.io/centos/centos:stream9
      workingDir: $(workspaces.source.path)
      script: |
        dnf install git -y
        git clone -v $(params.repo)
```

파이프라인에 워크스페이스

앞에 내용 계속 이어서…

```
---
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: demo-workspace-list-github-pipeline
spec:
  workspaces:
    - name: gitcode
```

파이프라인에 워크스페이스

앞에 내용 계속 이어서…

```
  steps:
    - name: list
      image: quay.io/centos/centos:stream9
      workingDir: $(workspaces.source.path)
      script: |
        ls -l
```

파이프라인에 워크스페이스

위에서 작성한 태스크를 파이프 라인으로 생성 및 구성한다.

```
# vi demo-clone-list-from-pipeline.yaml
---
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: demo-clone-list-from-pipeline
spec:
  workspaces:
    - name: gitcode
```

파이프라인에 워크스페이스

앞에 내용 계속 이어서…

```
  tasks:
    - name: clone
      taskRef:
        name: demo-workspace-clone-github-pipeline
      workspaces:
      - name: source
        workspace:  gitcode
```

파이프라인에 워크스페이스

앞에 내용 계속 이어서…

```
  - name: list
      taskRef:
        name: demo-workspace-list-github-pipeline
    workspaces:
      - name: source
        workspace: gitcode
    runAfter: ['clone']
      - clone
```

워크스페이스

파이프라인에 영구적 데이터 저장소 구성

2025-04-23

PVC기반 워크스페이스

PVC기반으로 구성하기 위해서는 먼저 쿠버네티스 클러스터에 PV/PVC가 구성이 되어 있어야 한다. NFS서버가 구성이 안되어 있으면 아래와 같이, NFS서버 및 CSI-NFS드라이버를 설치한다. 버전 적절하게 선택한다.

https://github.com/kubernetes-csi/csi-driver-nfs

https://github.com/kubernetes-csi/csi-driver-nfs/blob/master/docs/install-csi-driver-v4.7.0.md

```
# curl -skSL https://raw.githubusercontent.com/kubernetes-csi/csi-driver-nfs/v4.7.0/deploy/install-driver.sh | bash -s v4.7.0 --
# kubectl -n kube-system get pod -o wide -l app=csi-nfs-controller
# kubectl -n kube-system get pod -o wide -l app=csi-nfs-node
```

NFS서버

컨트롤 노드에서 NFS서버를 구성한다. **showmount**명령어로 올바르게 조회가 되는지 확인한다.

```
# dnf install nfs-utils
# vi /etc/exports
/workspaces/  *(rw)
# mkdir -m 777 -p /workspaces
# systemctl enable --now nfs-server
# export -avrs
# showmount -e master.example.com
```

SC/PVC

기본 스토리지 클래스 및 PVC를 생성한다.

```
# kubectl apply -f storageclass-configure.yaml
# kubectl apply -f tekton-pvc.yaml
# kubectl get sc
nfs-csi  (default)      nfs.csi.k8s.io      Delete                    Immediate
false                   4m12s
# kubectl get pvc
tekton-pvc      Bound       pvc-fc50e03e-dd99-4ec8-946a-958e7fcc33ee      1Gi
RWX             nfs-csi        13s
```

테크톤 PVC 연결

워크스페이스에 사용하는 저장소를 PVC로 연결해서 사용한다. 이번에는 Pipeline에 설정하는게 아니라, Pipeline Run를 통해서 임시적으로 사용할 PVC를 명시한다.

```yaml
apiVersion: tekton.dev/v1beta1
kind: PipelineRun
metadata:
  generateName: clone-and-list-pr-
spec:
  pipelineRef:
    name: demo-workspace-clone-list-github-pipeline
  workspaces:
    - name: gitcode
      persistentVolumeClaim:
        claimName: tekton-pvc
```

`name:`지시자를 사용하여도 상관은 없다. 다만, 이름이 고정으로 되기 때문에, 두 번 실행은 되지 않는다.

테크톤 PVC 연결

앞에 내용 계속 이어서…

```
# kubectl create -f  demo-workspace-pipeline-run-clone-and-list.yaml
# tkn pr list
```

테크톤 PVC 템플릿

고정이 아닌, 필요할 때 마다 PVC를 생성해서 워크스페이스를 구성하는 경우 아래와 같이 구성한다.

```
# vi demo-workspace-pipeline-run-clone-and-list-pvc-template.yaml
apiVersion: tekton.dev/v1beta1
kind: PipelineRun
metadata:
  generateName: clone-and-list-pr-
spec:
  pipelineSpec:
    workspaces:
      - name: gitcode
```

테크톤 PVC 템플릿

앞에 내용 계속 이어서…

```
    tasks:
      - name: clone
        taskRef:
          name: demo-workspace-clone-github-pipeline
        workspaces:
          - name: source
            workspace: gitcode
```

테크톤 PVC 템플릿

앞에 내용 계속 이어서…

```
      - name: list
        taskRef:
          name: demo-workspace-list-github-pipeline
        workspaces:
          - name: source
            workspace: gitcode
        runAfter:
          - clone
# kubectl create -f demo-workspace-pipeline-run-clone-and-list-pvc-
template.yaml
# tkn pipelinerun list
```

테크톤 PVC 템플릿

고정이 아닌, 필요할 때 마다 PVC를 생성해서 워크스페이스를 구성하는 경우 아래와 같이 구성한다.

```
  workspaces:
  - name: gitcode
    volumeClaimTemplate:
      spec:
        accessModes:
        - ReadWriteOnce
        resources:
          requests:
            storage: 1Gi
# kubectl create -f demo-workspace-pipeline-run-clone-and-list-pvc-
template.yaml
# tkn pipelinerun list
```

연습문제

워크스페이스

연습문제

다음과 같이 파이프라인 및 태스크를 구성해서 작업을 수행한다.

1. 모든 작업은 quay.io/centos/centos:stream9 이미지를 사용한다.
2. 레지스트리 서버를 간단하게 생성 및 구성한다.
    1. 마스터 노드에 docker-register서버를 구성한다.
    2. 포트번호는 5000번을 사용해서 동작하도록 한다.
    3. 구성은 Podman기반으로 한다.
3. 첫 번째 태스크는 다음과 같은 작업을 수행한다.
    1. 사용자가 명시한 주소의 소스코드를 내려받기 한다.
    2. 소스코드는 buildeah를 통해서 이미지 빌드를 수행한다.
    3. 해당 이미지는 이미지 빌드가 완료가 되면 내부 레지스트리 서버에 저장한다.
4. 세 번째 태스크는 다음과 같은 작업을 수행한다.
    1. 구성된 이미지 기반으로 웹 서비스를 구성한다.
    2. kubectl명령어로 빌드한 이미지 기반으로 웹 서비스를 구성한다.
    3. 해당 웹 서비스는 노드포트 38080으로 접근이 가능해야 한다.
5. 모든 작업들은 파이프라인으로 수행이 되어야 한다.

파이프라인

소개

소개

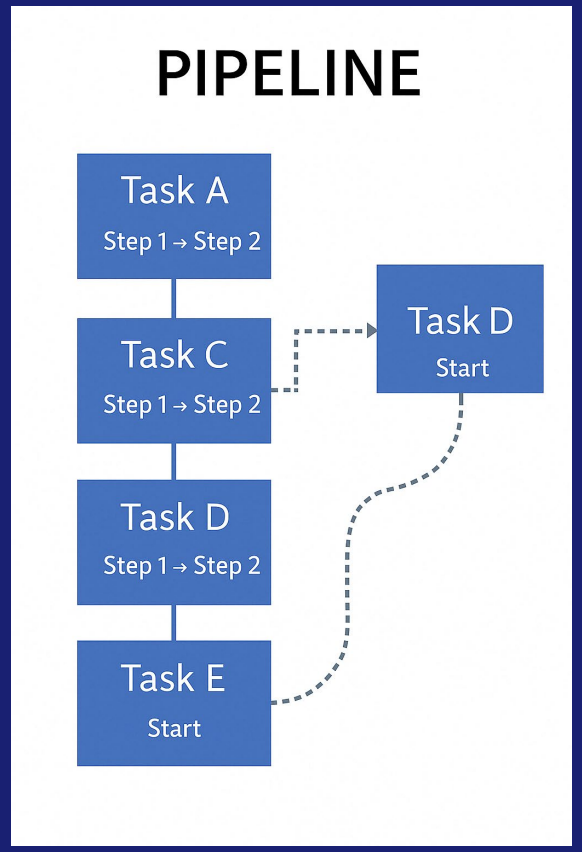
파이프라인은 앞서 사용하였던 단계(Step), 작업(Task)를 조합하여 실행한다.

또한, 파이프라인은 쿠버네티스에서 CI/CD인터페이스를 통해서 작업을 수행 시, 테크톤은 파이프라인을 호출하여 작업을 실행 및 수행한다. 예를 들어서 이전에 생성한 작업에 "복제", "컴파일", "검증"과 같은 작업이 있었다면, 이를 태스크를 통해서 작업을 수행한다.

태스크의 주요 목적은 최대한 자원을 재사용과 손쉽게 작업을 사용자가 원하는 순서대로 구성하는 게 주요 목적이다. 파이프 라인도 YAML기반으로 작성이 된다.

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
name: demo-pipeline
labels:
  key: value
spec:
```

소개


PIPELINE
Task A
Step 1 → Step 2
Task C
Step 1 → Step 2
Task D
Step 1 → Step 2
Task E
Start
Task D
Start

간단한 파이프 라인 #1

간단하게 파이프 라인을 구성한다. 앞에서 학습 내용을 잘 기억하고 구성한다.

```
# vi pipeline-hello-world.yaml
apiVersion: tekton.dev/v1
kind: Task
metadata:
  name: hello-world
spec:
  steps:
    - name: echo
      image: alpine
      script: |
        echo "Hello Tekton!"
```

간단한 파이프 라인 #1

앞에 내용 계속 이어서…

```
---
apiVersion: tekton.dev/v1
kind: Pipeline
metadata:
  name: hello-pipeline
spec:
  tasks:
    - name: say-hello
      taskRef:
        name: hello-world
# kubectl apply -f pipeline-hello-world.yaml
```

간단한 파이프 라인 #1

간단하게 `step, task`를 생성 후, 파이프 라인으로 작업을 호출한다.

위의 코드는 쉽게 구성 및 구현하기 위해서 task와 pipeline를 한 파일에 동시 선언 하였다. 쉽게 실행하기 위해서 `pipelinerun`이라는 자원을 통해서 실행이 가능하다.

```
# vi hello-pipelinerun.yaml
apiVersion: tekton.dev/v1
kind: PipelineRun
metadata:
  name: hello-pipelinerun
spec:
  pipelineRef:
    name: hello-pipeline
```

간단한 파이프 라인 #1

아래와 같이 확인 및 디버깅이 가능하다.

```
# tkn pipelinerun list
NAME                  STARTED          DURATION     STATUS
hello-pipelinerun     2 minutes ago    5s           Succeeded

# tkn pipelinerun describe hello-pipelinerun
Name:         hello-pipelinerun
Status:       Succeeded
StartTime:    2 minutes ago
Duration:     5s
Tasks:
 NAME         STATUS
 • say-hello  Succeeded
```

간단한 파이프 라인 #1

아래와 같이 확인 및 디버깅이 가능하다.

```
# tkn pipelinerun logs hello-pipelinerun -f
[say-hello : echo] Hello Tekton!
```

간단한 파이프 라인 #2

이번에는 앞에 내용에 깃/파일 목록 출력 관련된 작업을 출력하는 내용이다. 테크톤에서 제일 많이 하는 작업은 앞서 이야기 하였지만 다음과 같다.

1. `git clone`
2. `build`
3. `package`
4. `application image push`
5. `deploy to kubernetes cluster`

이 중, 1번과 2번만 파이프라인으로 수행한다.

간단한 파이프 라인 #2

먼저 깃 서버에서 소스코드를 클론 받는다.

```
# vi pipelinerun-git.yaml
apiVersion: tekton.dev/v1
kind: Task
metadata:
  name: fetch-repo
spec:
  workspaces:
    - name: shared
  steps:
    - name: clone
      image: alpine/git
      script: |
        git clone https://github.com/tektoncd/pipeline.git /workspace/shared
```

간단한 파이프 라인 #2

앞에 내용 계속 이어서…

```
---
apiVersion: tekton.dev/v1
kind: Task
metadata:
  name: list-files
spec:
  workspaces:
    - name: shared
  steps:
    - name: ls
      image: alpine
      script: |
        ls -al /workspace/shared
```

간단한 파이프 라인 #2

앞에 내용 계속 이어서…

```
---
apiVersion: tekton.dev/v1
kind: Pipeline
metadata:
  name: git-pipeline
spec:
  workspaces:
    - name: shared
```

간단한 파이프 라인 #2

앞에 내용 계속 이어서…

```
  tasks:
    - name: fetch
      taskRef:
        name: fetch-repo
      workspaces:
        - name: shared
          workspace: shared
    - name: list
      runAfter: ["fetch"]
      taskRef:
        name: list-files
      workspaces:
        - name: shared
          workspace: shared
```

간단한 파이프 라인 #2

앞에 내용 계속 이어서…

```
  tasks:
    - name: fetch
      taskRef:
        name: fetch-repo
      workspaces:
        - name: shared
          workspace: shared
    - name: list
      runAfter: ["fetch"]
      taskRef:
        name: list-files
      workspaces:
        - name: shared
          workspace: shared
```

간단한 파이프 라인 #2

적용 후 아래와 같이 결과를 확인한다.

```
# tkn pipelinerun list
NAME                STARTED          DURATION     STATUS
git-pipelinerun     10 seconds ago   8s           Succeeded
```

간단한 파이프 라인 #2

적용 후 아래와 같이 결과를 확인한다.

```
# tkn pipelinerun describe git-pipelinerun
Name:         git-pipelinerun
Namespace:    default
Status:       Succeeded
StartTime:    2025-04-20T16:15:00Z
Completion:   2025-04-20T16:15:08Z
Duration:     8s

Tasks:
 NAME      TASK NAME   STARTED          DURATION     STATUS
 • fetch   fetch-repo  10 seconds ago   5s           Succeeded
 • list    list-files  5 seconds ago    3s           Succeeded
```

간단한 파이프 라인 #2

적용 후 아래와 같이 결과를 확인한다.

```
# tkn pipelinerun logs git-pipelinerun -f
[fetch : clone] Cloning into '/workspace/shared' ...
[fetch : clone] remote: Enumerating objects: 10, done.
[fetch : clone] remote: Counting objects: 100% (10/10), done.
[fetch : clone] remote: Compressing objects: 100% (8/8), done.
[fetch : clone] Receiving objects: 100% (10/10), done.

[list : ls] total 24
[list : ls] drwxr-xr-x    3 root     root          4096 Apr 20 16:15 .github
[list : ls] -rw-r--r--    1 root     root          1234 Apr 20 16:15 README.md
[list : ls] -rw-r--r--    1 root     root          5678 Apr 20 16:15 main.go
[list : ls] -rw-r--r--    1 root     root          2345 Apr 20 16:15 pipeline.yaml
```

간단한 파이프 라인 #3

파라메터를 파이프라인에서 전달 시, 다음과 같이 구성한다. 파일 이름은 `pipelinerun-result.yaml`으로 작성한다.

#3에서는 어떠한 방식으로 파이프라인에 다양한 변수 값 전달이 가능한지 학습한다.

간단한 파이프 라인 #3

에디터로 아래와 같이 YAML파일을 작성한다.

```yaml
apiVersion: tekton.dev/v1
kind: Task
metadata:
  name: generate-message
spec:
  results:
    - name: output
      description: message text
  steps:
    - name: echo
      image: alpine
      script: |
        echo -n "메시지 전달 완료!" > $(results.output.path)
```

간단한 파이프 라인 #3

앞에 내용 계속 이어서…

```
---
apiVersion: tekton.dev/v1
kind: Task
metadata:
  name: print-message
spec:
  params:
    - name: input
  steps:
    - name: echo
      image: alpine
      script: |
        echo "전달된 메시지: $(params.input)"
```

간단한 파이프 라인 #3

앞에 내용 계속 이어서…

```
---
apiVersion: tekton.dev/v1
kind: Pipeline
metadata:
  name: result-pipeline
spec:
```

간단한 파이프 라인 #3

앞에 내용 계속 이어서…

```
  tasks:
    - name: generate
      taskRef:
        name: generate-message
    - name: print
      taskRef:
        name: print-message
      params:
        - name: input
          value: $(tasks.generate.results.output)
```

간단한 파이프 라인 #3

올바르게 파이프라인 실행에 등록이 되었는지 확인한다.

```
# kubectl apply -f pipelinerun-result.yaml
# pipelinerun.tekton.dev/result-pipelinerun created

# tkn pipelinerun list
NAME                 STARTED          DURATION     STATUS
result-pipelinerun   5 seconds ago    4s           Succeeded
```

간단한 파이프 라인 #3

올바르게 자원이 실행이 되었는지 확인한다.

```
# tkn pipelinerun describe result-pipelinerun
Name:         result-pipelinerun
Namespace:    default
Status:       Succeeded
StartTime:    2025-04-20T16:30:00Z
Completion:   2025-04-20T16:30:04Z
Duration:     4s

Tasks:
 NAME      TASK NAME          STARTED          DURATION     STATUS
 • generate   generate-message   4s ago           2s           Succeeded
 • print      print-message      2s ago           2s           Succeeded
```

간단한 파이프 라인 #3

실행된 결과를 로그를 통해서 확인한다.

```
# tkn pipelinerun logs result-pipelinerun -f
[generate : echo] 메시지 전달 완료!
[print : echo] 전달된 메시지: 메시지 전달 완료!
```

간단한 파이프 라인 #4

이번에는 파이프라인 기반으로 빌드 하는 방법에 대해서 학습한다. 여기서는 Go언어 기반으로 사용한다. 각 언어별 프레임워크에 따라서 지원 방법이 다르기 때문에, 다른 언어를 사용하는 경우 별도로 매뉴얼을 확인한다.

코드는 다음 슬라이드부터 시작.

간단한 파이프 라인 #4

코드는 아래와 같이 작성한다.

```
# vi git-clone-task.yaml
apiVersion: tekton.dev/v1
kind: TaskRun
metadata:
  name: git-clone-run
spec:
  taskRef:
    name: git-clone
    bundle: gcr.io/tekton-releases/catalog/upstream/git-clone:0.9
```

간단한 파이프 라인 #4

코드는 아래와 같이 작성한다.

```
  params:
    - name: url
      value: "https://github.com/gookhyun/go-hello-world.git"
    - name: revision
      value: "main"
    - name: subdirectory
      value: ""
    - name: deleteExisting
      value: "true"
  workspaces:
    - name: output
      persistentVolumeClaim:
        claimName: go-source-pvc
```

간단한 파이프 라인 #4

앞에 내용 계속 이어서…

```
  workspaces:
    - name: output
      persistentVolumeClaim:
        claimName: go-source-pvc
# kubectl apply -f git-clone-task.yaml
```

간단한 파이프 라인 #4

컴파일 부분을 작업으로 구성한다.

```
# vi go-build-task.yaml
apiVersion: tekton.dev/v1
kind: Task
metadata:
  name: go-build
spec:
  workspaces:
    - name: source
```

간단한 파이프 라인 #4

앞에 내용 계속 이어서…

```
  steps:
    - name: build
      image: golang:1.20
      workingDir: $(workspaces.source.path)
      script: |
        #!/bin/sh
        go mod tidy
        go build -o hello-app
# kubectl apply -f go-build-task.yaml
```

간단한 파이프 라인 #4

위의 두 개 내용을 이어서 파이프라인으로 구성한다.

```
# vi build-pipeline.yaml
apiVersion: tekton.dev/v1
kind: Pipeline
metadata:
  name: go-pipeline
spec:
  workspaces:
    - name: shared-workspace
```

간단한 파이프 라인 #4

앞에 내용 계속 이어서…

```
tasks:
   - name: fetch-repo
     taskRef:
       name: git-clone
       bundle: gcr.io/tekton-releases/catalog/upstream/git-clone:0.9
     params:
       - name: url
         value: "https://github.com/gookhyun/go-hello-world.git"
       - name: revision
         value: "main"
       - name: subdirectory
         value: ""
       - name: deleteExisting
         value: "true"
```

간단한 파이프 라인 #4

앞에 내용 계속 이어서…

```
    workspaces:
      - name: output
        workspace: shared-workspace
  - name: build-go
    runAfter:
      - fetch-repo
    taskRef:
      name: go-build
    workspaces:
      - name: source
        workspace: shared-workspace
```

간단한 파이프 라인 #4

앞에 내용 계속 이어서…

```
# vi build-pipeline-run.yaml
apiVersion: tekton.dev/v1
kind: PipelineRun
metadata:
  name: go-pipeline-run
spec:
  pipelineRef:
    name: go-pipeline
  workspaces:
    - name: shared-workspace
      persistentVolumeClaim:
        claimName: go-source-pvc
```

간단한 파이프 라인 #4

PVC가 구성이 안된 경우, 아래와 같이 구성 및 생성.

```
# vi go-source-pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: go-source-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

간단한 파이프 라인 #4

올바르게 구성이 되면, 다음과 같이 화면에 출력이 된다.

```
[build] go: downloading github.com/gin-gonic/gin v1.8.1
[build] go: downloading ...
[build] Compiled binary: hello-app
```

간단한 파이프 라인 #5

소스코드를 컴파일 하였으면, 쿠버네티스에서 사용이 가능하도록 Containerfile로 패키징이 필요하다. 패키징 하기 위해서 다음과 같은 두 가지 작업이 필요하다.

1. Containerfile 혹은 Dockerfile
2. 이미지 빌드 후, 보관할 image-registry서버
3. 효율적인 빌드를 위한 buildah사용

위와 같은 조건으로 작업을 수행한다.

간단한 파이프 라인 #5

디렉터리 구성은 다음과 같이 한다.

구성 요소	설명
`git-clone`	Git 저장소 복제 (Tekton Catalog)
`build-image`	Buildah or Kaniko 기반 이미지 빌드
`push-image`	레지스트리에 푸시
`apply-deploy`	kubectl apply로 쿠버네티스에 배포
`PipelineRun`	전체 실행 흐름
`PVC`	Git clone과 빌드 공유 목적
`ServiceAccount`	Registry 인증 및 쿠버네티스 권한 포함

간단한 파이프 라인 #5

컨테이너 파일은 보통 다음과 같이 작성한다. 여기서는 Go언어 기반의 helloworld프로그램 가지고 진행한다.

```
# vi Containerfile
FROM golang:1.20 AS builder
WORKDIR /app
COPY . .
RUN go build -o hello-app

FROM debian:bullseye-slim
COPY --from=builder /app/hello-app /hello-app
ENTRYPOINT ["/hello-app"]
```

간단한 파이프 라인 #5

컴파일 후, 이미지로 빌드 및 배포를 구성한다.

```
apiVersion: tekton.dev/v1
kind: Task
metadata:
  name: buildah-go-image
spec:
  params:
    - name: IMAGE
      description: Target image name
    - name: CONTEXT
      description: Build context path
      default: .
  workspaces:
    - name: source
```

간단한 파이프 라인 #5

컴파일 후, 이미지로 빌드 및 배포를 구성한다.

```
# vi go-buildah-task.yaml
apiVersion: tekton.dev/v1
kind: Task
metadata:
  name: buildah-go-image
spec:
  params:
    - name: IMAGE
      description: Target image name
    - name: CONTEXT
      description: Build context path
      default: .
```

간단한 파이프 라인 #5

앞에 내용 계속 이어서…

```
  workspaces:
    - name: source
  steps:
    - name: build-push
      image: quay.io/buildah/stable:v1.27.0
      securityContext:
        privileged: true
      workingDir: $(workspaces.source.path)/$(params.CONTEXT)
      script: |
        buildah bud -f Containerfile -t $(params.IMAGE) .
        buildah push $(params.IMAGE)
# kubectl apply -f go-buildah-task.yaml
```

간단한 파이프 라인 #5

이번에는 이미지를 쿠버네티스에 배포하는 작업을 구성한다. 여기서는 SA 및 RBAC구성하지 않았기에 올바르게 동작하지 않는다.

```
# apply-deploy-task.yaml
apiVersion: tekton.dev/v1
kind: Task
metadata:
  name: apply-deploy
spec:
  params:
    - name: YAML_PATH
      default: k8s/deployment.yaml
```

간단한 파이프 라인 #5

앞에 내용 계속 이어서…

```
  workspaces:
    - name: source
  steps:
    - name: apply
      image: bitnami/kubectl:latest
      script: |
        kubectl apply -f $(workspaces.source.path)/$(params.YAML_PATH)
```

간단한 파이프 라인 #5

파이프라인 생성한다. 앞에서 만든 작업을 하나의 워크 플로우로 호출한다.

```
# pipeline-build.yaml
apiVersion: tekton.dev/v1
kind: Pipeline
metadata:
  name: go-app-cicd
spec:
  params:
    - name: IMAGE
  workspaces:
    - name: shared-workspace
```

간단한 파이프 라인 #5

파이프라인 생성한다. 앞에서 만든 작업을 하나의 워크 플로우로 호출한다.

```
  tasks:
    - name: fetch-repo
      taskRef:
        name: git-clone
        bundle: gcr.io/tekton-releases/catalog/upstream/git-clone:0.9
      params:
        - name: url
          value: "https://github.com/gookhyun/go-hello-world.git"
        - name: revision
          value: "main"
      workspaces:
        - name: output
          workspace: shared-workspace
```

간단한 파이프 라인 #5

앞에 내용 계속 이어서…

```
    - name: build-push
      runAfter: [fetch-repo]
      taskRef:
        name: buildah-go-image
      params:
        - name: IMAGE
          value: "$(params.IMAGE)"
        - name: CONTEXT
          value: "."
      workspaces:
        - name: source
          workspace: shared-workspace
```

간단한 파이프 라인 #5

앞에 내용 계속 이어서…

```
    - name: deploy
      runAfter: [build-push]
      taskRef:
        name: apply-deploy
      params:
        - name: YAML_PATH
          value: "k8s/deployment.yaml"
      workspaces:
        - name: source
          workspace: shared-workspace
# kubectl apply -f pipeline-build.yaml
```

간단한 파이프 라인 #5

앞에서 구성한 파이프라인을 실행한다.

```
# vi pipeline-build-run.yaml
apiVersion: Tekton.dev/v1
kind: PipelineRun
metadata:
  name: go-app-cicd-run
spec:
  pipelineRef:
    name: go-app-cicd
  params:
    - name: IMAGE
      value: registry.dustbox.kr/tangt64/go-app:latest
```

간단한 파이프 라인 #5

앞에 내용 계속 이어서…

```
  workspaces:
    - name: shared-workspace
      persistentVolumeClaim:
        claimName: go-source-pvc
  serviceAccountName: pipeline-sa
# kubectl apply -f pipeline-build-run.yaml
```

간단한 파이프 라인 #5

쿠버네티스 클러스터에 배포할 deployment, service를 생성한다.

```
# vi deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: go-hello
spec:
  replicas: 1
  selector:
    matchLabels:
      app: go-hello
```

간단한 파이프 라인 #5

앞에 내용 계속 이어서…

```
  template:
    metadata:
      labels:
        app: go-hello
    spec:
      containers:
        - name: go-hello
          image: registry.dustbox.kr/gookhyun/go-app:latest
          ports:
            - containerPort: 8080
```

간단한 파이프 라인 #5

앞에 내용 계속 이어서…

이 부분은 서비스 부분이다. 별도로 만들지 말고 하나의 파일로 작성 및 구성한다.

```
---
apiVersion: v1
kind: Service
metadata:
  name: go-hello-svc
spec:
  selector:
    app: go-hello
  ports:
    - port: 80
      targetPort: 8080
```

간단한 파이프 라인 #5

클러스터에서 사용할 SA계정 및 레지스트리 인증 계정 정보를 등록한다.

```
# vi deployment-sa-secret.yaml
apiVersion: v1
kind: Secret
metadata:
  name: regcred
  annotations:
    tekton.dev/docker-0: https://registry.dustbox.kr
type: kubernetes.io/basic-auth
stringData:
  username: gookhyun
  password: xxxxxx
```

간단한 파이프 라인 #5

위의 내용 계속 이어서…

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: pipeline-sa
secrets:
  - name: regcred
# kubectl apply -f deployment-sa-secret.yaml
```

PIPELINE ORDERING

파이프라인은 자유롭게 **작업 순서 변경(ordering)**이 가능하다. 이를 통해서 각 작업마다 원하는 방식으로 작업 순서를 지정할 수 있다. 이미 위에서 기능 학습을 하였지만, **작업(task) 및 단계(step)**은 위치에 따라서 실행하는 순서가 달라진다.

그래서 태스크에서 구성한 순서가 아닌, 사용자가 원하는 순서대로 진행을 하고 싶은 경우, YAML코드에 사용자가 원하는 작업 이름을 순서대로 명시하면 된다.

아래 슬라이드의 내용처럼, 간단하게 여러 개의 작업을 구성하여 파이프라인을 통해서 실행 순서를 재구성 한다.

PIPELINE ORDERING - TASK

```
$ vi pipeline-ordering-task.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: pipeline-ordering-task
spec:
  params:
    - name: task-name
      type: string
    - name: time
      type: string
      default: ""
```

PIPELINE ORDERING - TASK

다음과 같이 작업에 대해서 순서 선언 및 지정이 가능하다. 먼저, 실험용 작업을 구성한다.

```
# vi pipeline-ordering-task.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: pipeline-ordering-task
spec:
  params:
    - name: task-name
      type: string
    - name: time
      type: string
      default: ""
```

PIPELINE ORDERING - TASK

위의 내용 계속 이어서…

```
  steps:
    - name: first-task
      image: quay.io/centos/centos:stream9
      command:
        - /bin/bash
      args: ['-c', 'echo Ran to the first task $(params.task-name)']
    - name: second-task
      image: quay.io/centos/centos:stream9
      command:
        - /bin/bash
      args: ['-c', 'echo Ran to the second task $(params.time)']
```

PIPELINE ORDERING - TASK

위의 내용 계속 이어서…

```
    - name: logger
      image: quay.io/centos/centos:stream9
      command:
        - /bin/bash
      args: ['-c', 'echo Ran to the last task $(date +%d/%m/%Y %T) - Task
$(params.task-name) Completed']
# kubectl apply -f pipeline-ordering-task.yaml
```

PIPELINE ORDERING - PIPELINE

앞에서 작성한 작업들을 파이 라인으로 순서대로 실행하도록 작성한다.

```
# vi pipeline-ordering-pipeline.yaml
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: pipeline-ordering-pipeline
```

PIPELINE ORDERING - PIPELINE

위의 내용 계속 이어서…

```
spec:
  tasks:
  - name: first
    params:
      - name: task-name
        value: A
      - name: time
        value: "2PM"
   taskRef:
      name: first-task
```

PIPELINE ORDERING - PIPELINE

위의 내용 계속 이어서…

```
  - name: second
    params:
      - name: task-name
        value: "This is the Task 2"
    taskRef:
      name: first-task
  - name: third
    params:
      - name: task-name
        value: "This is the Task 3"
      - name: time
        value: "3PM"
```

PIPELINE ORDERING - PIPELINE

위의 내용 계속 이어서…

```
  - name: fourth
    params:
      - name: task-name
        value: "This is the Task 4"
    taskRef:
      name: logger
```

PIPELINE ORDERING - RUN

적용한다.

```
# kubectl apply -f pipeline-ordering-task.yaml
# kubectl apply -f pipeline-ordering-pipeline.yaml
# tkn pipeline start pipeline-ordering-pipeline --showlog
```

PIPELINE ORDERING - RESULT(TASK)

앞에서 사용하였던 ordering에 추가적으로 result를 같이 붙여서 응용한다. 먼저, 작업을 구성한다.

```
# vi demo-pipeline-dice-task-param-result.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: demo-pipeline-dice-task-param-result
spec:
  params:
  - name: sideNum
    description: number of sides to the dice
    default: 6
    type: string
```

PIPELINE ORDERING - RESULT(TASK)

앞에 내용 계속 이어서…

```
  results:
  - name: dice-result
    description: result of dice roll number
  steps:
  - name: rolling-n-rolling-dice
    image: quay.io/centos/centos:stream9
    script: |
      dnf install php-cli -y
      php -r 'echo rand(1,$(params.sideNum));' > dice-result.txt
```

PIPELINE ORDERING - RESULT(PIPELINE)

파이프라인을 구성 및 생성 후 작업을 실행한다.

```
# vi demo-pipeline-dice-pipeline-result.yaml
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: demo-pipeline-dice-pipeline-result
spec:
  params:
  - name: sideNum
    type: string
    default: 6
```

PIPELINE ORDERING - RESULT(PIPELINE)

앞에 내용 계속 이어서…

```
  tasks:
  - name: first
    params:
    - name: text
      value: "The dice sides is $(params.sideNum)"
    taskRef:
      name: logger
```

PIPELINE ORDERING - RESULT(PIPELINE)

아래 내용을 최종적으로 추가 후, 아래와 같이 명령어를 실행한다.

```
  - name: roll
    params:
    - name: sides
      value: "$(params.sideNum)"
    taskRef:
      name: demo-pipeline-dice-param-result
    runAfter:
    - first
# kubectl apply -f demo-pipeline-dice-task-param-result.yaml
# kubectl apply -f demo-pipeline-dice-pipeline-result.yaml
# tkn pipeline start demo-pipeline-dice-pipeline-result --showlog
```

파이프라인 연습문제

연습문제

연습문제

다음과 같이 파이프라인 및 태스크를 구성해서 작업을 수행한다.

1. 모든 작업은 quay.io/centos/centos:stream9 이미지를 사용한다.
2. 첫번째 작업은 사용자에게 메시지를 입력 받는다.
    1. uname이름으로 사용자 이름을 입력 받는다.
    2. 입력 받은 이름은 반드시 출력해야 한다.
    3. 사용자 이름을 uname.txt파일로 저장한다.
3. 두 번째 작업은 사용자의 이메일 주소를 입력 받는다.
    1. uemail이름으로 사용자 메일 주소를 입력 받는다.
    2. 입력 받은 이름은 반드시 출력해야 한다.
    3. 사용자 메일 주소를 uemail.txt파일로 저장한다.
4. 위의 모든 작업을 파이프라인으로 생성 후 동시에 작업을 수행하도록 한다.

디버깅

디버깅

파이프라인 및 태스크 정리

2025-04-23

디버깅

테크톤에서 디버깅 방식은 두가지가 있다.

1. `kubectl`명령어를 통한 디버깅

2. `tkn`명령어를 통한 디버깅

`tkn`명령어 기반으로 하는 경우, 어떠한 **태스크/파라메터**를 찾지 못하는지 메시지를 통해서 확인이 가능하지만, 이미지나 혹은 컨테이너에서 발생한 오류 메시지는 확인이 불가능하다.

그래서 `kubectl`명령어를 통해서 좀 더 자세하게 확인을 해야 한다.

```
# kubectl get tasks                      → kubectl get pods
# kubectl get pipeline                   → kubectl get pods
# kubectl describe pod <TASK_NAME>       → tkn task describe <TASK_NAME>
# kubectl logs tasks <TASK_NAME>         → kubectl log pod/<POD_NAME>
```

2025-04-23

DEBUG-TASK

디버깅하기 위해서 다음과 같이 작성한다.

```
# vi exit-debug-task.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: exit-debug-task
spec:
  params:
    - name: text
      type: string
    - name: exitcode
      type: string
```

DEBUG-TASK

앞에 내용 계속 이어서…

```
  steps:
    - name: log
      image: quay.io/centos/centos:stream9
      command:
        - /bin/bash
      args: ["-c", " echo $(params.text)"]
    - name: exit
      image: quay.io/centos/centos:stream9
      command:
        - /bin/bash
      args: ["-c", "echo 'Exiting with code $(params.exitcode)' && exit
$(params.exitcode)"]
```

DEBUG-TASK

올바르게 실행이 되었는지, 디버깅용 작업을 구성 및 생성한다.

```
$ vi log-and-exit.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: log-and-exit
spec:
  params:
  - name: text
      type: string
  - name: exitcode
    type: string
```

DEBUG-TASK

앞에 내용 계속 이어서…

```
  steps:
  - name: log
    image: quay.io/centos/centos:stream9
    command:
      - /bin/bash
    args: ['-c, 'echo $(params.text)']
```

DEBUG-PIPELINE

위의 작업을 파이프라인을 통해서 호출 및 실행한다.

```
# vi exit-debug-pipeline.yaml
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: exit-debug-pipeline
```

DEBUG-PIPELINE

앞에 내용 계속 이어서…

```
spec:
  tasks:
  - name: clone
    taskRef:
      name: log-and-exit
    params:
      - name: text
        value: "Simulating git clone"
      - name: exitcode
        value: "0"
```

DEBUG-PIPELINE

위의 작업을 파이프라인을 통해서 호출 및 실행한다.

```
# vi exit-debug-pipeline.yaml
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: exit-debug-pipeline
```

DEBUG-PIPELINE

앞에 내용 계속 이어서…

```
spec:
  tasks:
  - name: clone
    taskRef:
      name: log-and-exit
    params:
      - name: text
        value: "Simulating git clone"
      - name: exitcode
        value: "0"
```

DEBUG-PIPELINE

앞에 내용 계속 이어서…

```
  - name: unit-tests
    taskRef:
      name: exit-debug-task
    params:
      - name: text
        value: "Simulating unit testing"
      - name: exitcode
        value: "0"
    runAfter:
      - clone
```

DEBUG-PIPELINE

앞에 내용 계속 이어서…

```
  - name: deploy
    taskRef:
      name: exit-debug-task
    params:
      - name: text
        value: "Simulating deployment"
      - name: exitcode
        value: "0"
    runAfter:
      - unit-tests
```

DEBUG-TASK RUN 1

아래와 같이 자원 등록 및 실행한다.

```
# kubectl apply -f exit-debug-task.yaml
# kubectl apply -f exit-debug-pipeline.yaml
# tkn pipeline start exit-debug-pipeline --showlog
```

DEBUG-TASK VALUE

위의 `exit-debug-task.yaml`파일에 아래와 같이 수정한다. 기존에 "0"을 "1"으로 변경한다.

```
  - name: unit-tests
    taskRef:
      name: exit-debug-task
    params:
      - name: text
        value: "Simulating unit testing"
      - name: exitcode
        value: "0" → "1"
    runAfter:
      - clone
```

DEBUG-TASK RUN 2

다음 명령어로 등록 및 실행한다.

```
# kubectl -f exit-debug-pipeline.yaml
# tkn pipeline start exit-debug-task --showlog
```

FINALLY/CLEANUP

`Finally, Cleanup`작업은 특정 작업이 완료가 된 후, 특정 작업을 내용을 제거 시 사용한다. 보통 이 부분은 PV/PVC를 사용할 때 이 명령어를 사용하여 처리한다.

아래 예제들은 간단하게 메시지 출력 후, PVC의 내용을 `finally`명령어를 통해서 작업 내용을 제거한다.

FINALLY/CLEANUP

작업 마무리 시, 다음과 같이 작업을 수행한다.

```
# vi task-cleanup.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: task-cleanup
spec:
  steps:
  - name: clean
    image: quay.io/centos/centos:stream9
    command:
      - /bin/bash
    args: ['-c', 'echo Cleaning up!']
```

FINALLY/CLEANUP

위의 내용에 다음과 같은 내용을 추가하여 기능을 확장한다.

```
# vi exit-debug-pipeline.yaml
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: exit-debug-pipeline
spec:
  tasks:
…
  finally:
  - name: cleanup-task
    taskRef:
      name: cleanup
```

FINALLY/CLEANUP RUN

다음과 같이 적용 및 확인한다.

```
# kubectl apply -f task-cleanup.yaml
# kubectl apply -f exit-debug-pipeline.yaml
# tkn pipeline start exit-debug-pipeline --showlog
```

디버깅

연습문제

연습문제

아래 작업을 등록하여 올바르게 실행이 되는지 확인한다.

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: echo-task
spec:
  params:
    - name: message
      type: string
  steps:
    - name: echo
      image: alpine
      script: |
        echo "$(params.msg)"
```

연습문제

아래 코드를 올바르게 동작하도록 수정. 파일 이름은 적절하게 생성 후, 파이프라인으로 실행이 되어야 함.

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: generate-result
spec:
  steps:
    - name: echo
      image: alpine
      script: |
        echo "myvalue" > $(results.myoutput.path)
```

연습문제

앞에 내용 계속 이어서…. 위의 파이프라인에서 사용하는 작업파일.

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: debug-pipeline-2
spec:
  tasks:
    - name: generate
      taskRef:
        name: generate-result
    - name: use-result
      taskRef:
        name: echo-task
      params:
        - name: message
          value: "$(tasks.generate.results.myoutput)"
```

연습문제

해당 파이프라인에 어떠한 문제가 있는지 확인 후 코드를 적절하게 수정. 앞에 echo-task와 연결됨.

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: debug-pipeline-2
spec:
  tasks:
    - name: generate
      taskRef:
        name: generate-result
    - name: use-result
      taskRef:
        name: echo-task
      params:
        - name: message
          value: "$(tasks.generate.results.myoutput)"
```

조건식

조건식 활용 및 사용하기

조건식

소개

소개

테크톤에서 조건식은 앤서블과 비슷하게 제공하지만, 기본적으로 Go Lang에서 사용하는 템플릿 형식을 사용한다.

1. https://go.dev/ref/spec
2. https://tekton.dev/docs/triggers/cel_expressions/
3. https://tekton.dev/docs/pipelines/pipelines/

입력 혹은 출력에 결과 내용을 다음과 같이 조건으로 사용이 가능하다.

```
when:
  - input: "true"
    operator: IN
    values: ["true"]
```

조건 명령어

기본적으로 많이 사용하는 조건식 명령어는 아래와 같다.

항목	설명
`input`	비교할 값 (일반적으로 파라미터 또는 결과 값 사용)
`operator`	비교 연산자 (in, notin)
`values`	비교 대상 값 배열
사용 위치	Pipeline 또는 Task 내 when 필드 사용
동작 방식	조건이 만족할 때만 해당 Task 실행

조건 명령어

좀 더 자세한 사용 방법은 아래 표를 참고한다.

구성 요소	설명	예시 구문
input	조건식에 입력되는 값. 생략 시 빈 문자열이 기본값으로 사용됨.	- 고정값 예: "ubuntu" - 변수 예: $(params.image) - Task 결과 예: $(tasks.task1.results.image) - 배열 값 예: $(tasks.task1.results.array-results[1])
operator	입력 값과 values 배열의 관계를 나타내는 연산자. 유효한 연산자를 반드시 지정해야 함.	- in - notin
values	비교 대상이 되는 문자열 배열. 반드시 비어 있지 않아야 함.	- 배열 파라미터 예: ["$(params.images[*])"] - Task 결과 배열 예: ["$(tasks.task1.results.array-results[*])"] - 고정값 포함 예: "ubuntu" - 변수 포함 예: ["$(params.image)"], ["$(tasks.task1.results.image)"]

코드 예제

입력 사용 시, 보통 아래와 같이 코드로 표현한다.

```
- name: conditional-task
  taskRef:
    name: echo-hello
  when:
    - input: "$(params.environment)"
      operator: in
      values: ["prod", "staging"]
```

코드 예제

아래 코드를 보면 알겠지만, 앤서블에서 사용하는 조건식 when과 거의 비슷하게 사용한다. 변수 number에 값이 10이 들어오면, winmsg작업을 실행한다. 조건식은 가급적이면 대문자로 작성한다.

테크톤에서 조건식을 적용하면 다음과 같이 사용이 가능하다.

```
    - name: win
      params:
        - name: text
          value: winner
      taskRef:
        name: winmsg
      when:
        - input: $(params.number)
          operator: in   # IN
          values: ["10"]
```

조건식

조건식, 파라메터 사용하기

2025-04-23

사용하기(IN)

조건식 테스트를 위해서 작업을 아래와 같이 작성한다.

```
# vi demo-expression-task-show-timedate.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: demo-expression-task-time-show-timedate
spec:
```

사용하기(IN)

앞에 내용 계속 이어서…. 컨테이너 내부의 로케일을 임시로 변경 후 날짜를 출력한다.

```
params:
    - name: locale
      type: string
      default: America/New_York
  steps:
    - name: show-timedate
      image: quay.io/centos/centos:stream9
      script: |
        export TZ=$(params.locale)
        DATE=$(date +%d/%m/%Y\ %T)
        echo [$DATE] - $(params.locale)
```

사용하기(IN)

아래와 같이 파이프 라인을 생성한다. 앞에서 생성한 작업에 변수를 전달한다.

```
# vi demo-expression-pipeline-show-timedate.yaml
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: demo-expression-pipeline-time-show-timedate
spec:
  params:
    - name: timezone
      description: set to TZ
      type: string
```

사용하기(IN)

앞에 내용 계속 이어서…

```
tasks:
    - name: timezone
      params:
        - name: timezone
          value: Asia/Seoul
      taskRef:
        name: demo-expression-task-time-show-timedate
      when:
        - input: $(params.timezone)
          operator: in
          values: ["Asia/Seoul"]
```

적용(IN)

아래 명령어로 실행한다.

```
# kubectl apply -f demo-expression-task-show-timedate.yaml
# kubectl apply -f demo-expression-pipeline-show-timedate.yaml
# tkn pipeline list
# tkn task list
```

NOTIN

기존에 사용하던 파일에 다음과 같이 수정한다. **NOTIN**은 말 그대로 조건에 없는 값이 들어오면 실행이 된다.

```
# vi demo-expression-pipeline-show-timedate-notin.yaml
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: demo-expression-pipeline-time-show-timedate
spec:
  params:
    - name: timezone
      description: set to TZ
      type: string
```

NOTIN

기존에 사용하던 파일에 다음과 같이 수정한다. **NOTIN**은 말 그대로 조건에 없는 값이 들어오면 실행이 된다.

```
tasks:
  - name: timezone
    params:
      - name: timezone
        value: Asia/Seoul
    taskRef:
      name: demo-expression-task-time-show-timedate
    when:
      - input: $(params.timezone)
        operator: IN
        values: ["Asia/Seoul"]
```

NOTIN

위의 내용의 마지막에 아래 내용을 추가한다.

```
    - name: timezone-default
      params:
        - name: timezone
          value: Asia/Tokyo
      taskRef:
        name: demo-expression-task-time-show-timedate
      when:
        - input: $(params.timezone)
          operator: NOTIN
          values: ["Asia/Tokyo"]
```

적용(NOTIN)

아래 명령어로 실행한다.

```
# kubectl apply -f demo-expression-pipeline-show-timedate-notin.yaml
# tkn pipeline list
# tkn pipeline start demo-expression-pipeline-show-timedate-notin
```

조건식

연습문제

연습문제

틀린 조건식을 올바르게 동작하도록 수정한다.

연습문제

아래 작업을 등록하여 올바르게 실행이 되는지 확인한다.

```
when:
- input: "$(params.branch)"
  operator: equals
  values: ["main"]
```

연습문제

정답.

```
when:
- input: "$(params.branch)"
  operator: in
  values: ["main"]
```

연습문제

아래 작업을 등록하여 올바르게 실행이 되는지 확인한다.

```
when:
- input: "$(params.environment)"
  operator: notin
  values: "prod"
```

연습문제

정답.

```
when:
- input: "$(params.environment)"
  operator: notin
  values: ["prod"]
```

연습문제

아래 작업을 등록하여 올바르게 실행이 되는지 확인한다.

```
when:
- input: "$(params.git-branch)"
  operator: in
  values: ["main"]
  runAfter:
    - lint
```

연습문제

정답.

```
- name: build
  taskRef:
    name: build-task
  runAfter:
    - lint
  when:
    - input: "$(params.git-branch)"
      operator: in
      values: ["main"]
```

인증

기본 인증 및 SSH

인증

소개

GIT SERVER LOGIN

대다수 깃 서버는 인증을 요구한다. 내부 깃 서버에서 소스코드를 다운로드 받기 위해서 아래와 같이 코드를 작성 후, 저장소에 접근한다.

```
# vi demo-github-secret-task.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: demo-github-list
spec:
  params:
    - name: private-repo
      type: string
```

GIT SERVER LOGIN

위의 내용 계속 이어서…. 여기서는 임시로 POD영역에 디렉터리 생성 후, 코드를 미러링 받는다.

```
  steps:
    - name: clone
      image: alpine/git
      script: |
        mkdir /temp && cd /temp
        git clone $(params.private-repo) .
        ls -lR .
# kubectl apply -f demo-github-secret-task.yaml
```

GIT SERVER ID/PW+TOKEN

깃헙과 같은 시스템에 로그인이 필요하다. 아이디 및 비밀번호로 로그인 하는 경우, 아래와 같이 접근이 가능하다.

```
# vi kubernetes-github-secret-token.yaml
apiVersion: v1
kind: Secret
metadata:
  name: kubernetes-github-sa-token
  annotations:
    tekton.dev/git-0: https://github.com
```

GIT SERVER ID/PW+TOKEN

앞에 내용 계속 이어서…

```
type: kubernetes.io/basic-auth
stringData:
  username: tangt64
  password: ghp_token
# kubectl apply -f kubernetes-github-secret-token.yaml
```

GIT SERVER SSH

깃헙이나 혹은 SSH기반으로 접근이 필요한 경우 아래와 같이 작성 후 접근이 가능하다.

```
# vi kubernetes-github-secret-ssh.yaml
apiVersion: v1
kind: Secret
metadata:
  name: kubernetes-github-sa-ssh
  annotations:
    tekton.dev/git-0: github.com
type: kubernetes.io/ssh-auth
```

GIT SERVER SSH

위의 내용 계속 이어서…. SSH키를 아래와 같이 입력한다. known_hosts 문제를 막기 위해서 미리 fingerprint를 넣어준다.

```
stringData:
  ssh-privatekey: |
    -----BEGIN OPENSSH PRIVATE KEY-----
    -----END OPENSSH PRIVATE KEY-----
  known_hosts: github.com,140.82.112.4 ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAQEAq2A7hRGmdnm9tUDbO9IDSwBK6TbQa+PXYPCPy6rbTrTtw7P
HkccKrpp0yVhp5HdEIcKr6pLlVDBfOLX9QUsyCOV0wzfjIJNlGEYsdlLJizHhbn2mUjvSAHQqZE
TYP81eFzLQNnPHt4EVVUh7VfDESU84KezmD5QlWpXLmvU31/yMf+Se8xhHTvKSCZIFImWwoG6mb
UoWf9nzpIoaSjB+weqqUUmpaaasXVal72J+UX2B+2RPW3RcT0eOzQgqlJL3RKrTJvdsjE3JEAvG
q3lGHSZXy28G3skua2SmVi/w4yCE6gbODqnTWlg7+wC604ydGXA8VJiS5ap43JXiUFFAaQ==
# kubectl apply -f kubernetes-github-secret-ssh.yaml
```

SA(TOKEN)

서비스 계정을 생성한다. 바로 secret자원에 접근이 불가능하기 때문에, SA계정을 통해서 접근을 권장한다.

```
# vi kubernetes-github-sa-token.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: kubernetes-github-sa-token
secrets:
  - name: kubernetes-github-sa-token
# kubectl apply -f kubernetes-github-sa-token.yaml
```

SA(SSH)

SSH도 똑같이 SA계정을 생성 및 구성한다.

```
# vi kubernetes-github-sa-ssh.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: kubernetes-github-sa-ssh
secrets:
  - name: kubernetes-github-sa-ssh
# kubectl apply -f kubernetes-github-sa-ssh.yaml
```

TASKRUN(GIT TOKEN)

올바르게 동작하는 TR 구성 후 실행한다.

```
# vi kubernetes-github-taskrun-git-token.yaml
apiVersion: tekton.dev/v1beta1
kind: TaskRun
metadata:
  generateName: git-token-
```

TASKRUN(GIT TOKEN)

앞에 내용 계속 이어서…

```
spec:
  serviceAccountName: kubernetes-github-sa-token
  params:
    - name: private-repo
      value: https://github.com/tangt64/duststack-osp-auto.git
  taskRef:
    name: demo-github-list
# kubectl create -f kubernetes-github-taskrun-git-token.yaml
```

TASKRUN(SSH)

이번에는 SSH 기반으로 접근 및 clone받는 TR를 구성한다.

```
# vi kubernetes-github-taskrun-git-ssh.yaml
apiVersion: tekton.dev/v1beta1
kind: TaskRun
metadata:
  generateName: git-ssh-
```

TASKRUN(SSH)

앞에 내용 계속 이어서…

```
spec:
  serviceAccountName: kubernetes-github-sa-ssh
  params:
    - name: private-repo
      value: tangt64@github.com:tangt64/dustbox-auto-osp
  taskRef:
    name: demo-github-list
# kubectl apply -f kubernetes-github-secret-token.yaml
```

기능확장

HUB

소개

앤서블 갤럭시 기능과 비슷하게 테크톤 허브가 존재. 이를 통해서 좀 더 손쉽게 작업 관리 및 배포가 가능함.

https://hub.tekton.dev/

이 부분은 강사가 데모를 통해서 시연 및 확인.

허브

확장 기능을 통해서 몇가지 기능을 구현한다. 아래와 같이 검색 및 설치를 진행한다.

```
# tkn hub get task
# tkn hub search
# tkn search buildah
# tkn search kubernetes
# tkn search git-clone
# tkn install buildah
# tkn install kubernetes-action
# tkn install git-clone
```

설치(HELM)

설치는 매우 간단하다. 먼저 Helm 명령어를 설치한다. 몇몇 배포판은 패키지로 지원하기도 한다.

```
# curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
| bash
# helm version
```

앞에서 사용한 ingress, loadbalancer를 Helm기반으로 설치한다.

```
# helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
# helm repo update
# helm install ingress-nginx ingress-nginx/ingress-nginx --namespace
ingress-nginx --create-namespace
# kubectl get pods -n ingress-nginx
NAME                                        READY   STATUS    RESTARTS
AGEingress-nginx-controller                 1/1     Running   0          2m
```

TKN/METALLB

트리거가 설치가 되어 있는지 확인한다. 트리거를 사용하기 위해서 로드밸런서 서비스를 구성 후, 테스트 하도록 한다.

```
# tkn version
Client version: 0.36.0
Pipeline version: v0.59.0
Triggers version: v0.26.2
```

METALLB RANGE

L4/L7서비스를 간단하게 MetalLB기반으로 구성한다. MetalLB를 사용하기 어려우면 NodePort기반으로 구성하여도 상관 없다.

```
# helm repo add metallb https://metallb.github.io/metallb
# helm repo update
# helm install metallb metallb/metallb --namespace metallb-system --create-namespace
```

INGRESS

ingress를 Helm기반으로 설치한다.

```
# helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
# helm repo update
# helm install ingress-nginx ingress-nginx/ingress-nginx --namespace
ingress-nginx --create-namespace
# kubectl get pods -n ingress-nginx
NAME                                        READY   STATUS    RESTARTS
AGEingress-nginx-controller                 1/1     Running   0          2m
```

트리거

조건식 활용 및 사용하기

트리거

webhook

소개

트리거(trigger)는 **웹훅(webhooks)**를 통해서 외부에서 테크톤 작업을 실행 시킨다.

예를 들어서 외부/내부에서 사용하는 깃 서버에서 웹훅를 구성하여, 소스코드가 업데이트가 되면 서비스를 업데이트를 위해서 빌드를 자동으로 수행한다.

보통 낮은 사양으로 랩을 진행하기 때문에 여기서는 GITLAB대신 Gogs 및 docker-registry를 사용하여 가볍게 구현 및 시연한다.

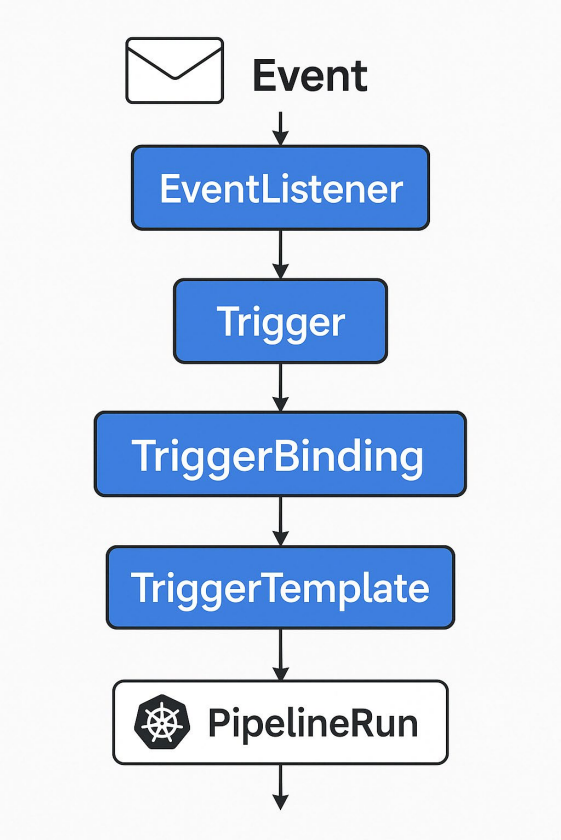
1. MetalLB
2. Gogs
3. docker-registry

설명

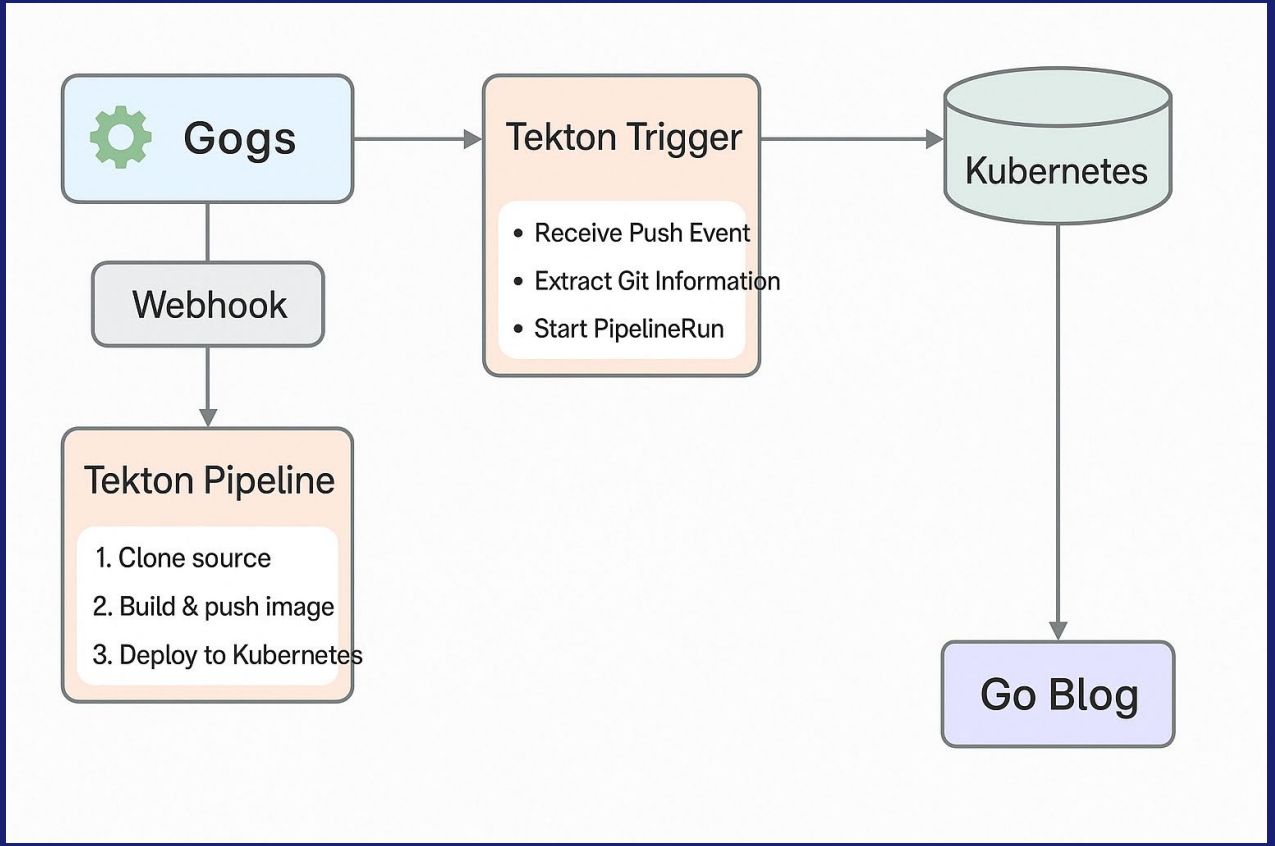
트리거를 구성 및 실행하기 위해서, 어떠한 자원을 외부에서 **웹훅(webhook)**를 통해서 실행이 가능한지 선언해야 한다. 이러한 테크톤 자원은 **트리거 바인딩(trigger binding)**이라고 부른다.

트리거 자원은 별도 자원으로 제공 및 구성이 되기 때문에 따로 설치해야 한다. 이 랩은 앞에서 미리 트리거를 설치하였다.

설명


Event
EventListener
Trigger
TriggerBinding
TriggerTemplate
PipelineRun

설명


Gogs
Webhook
Tekton Pipeline
1. Clone source
2. Build & push image
3. Deploy to Kubernetes
Tekton Trigger
• Receive Push Event
• Extract Git Information
• Start PipelineRun
Kubernetes
Go Blog

설명

트리거 관련 자원을 다음과 같이 테크톤에서 제공하고 있다.

리소스 종류	설명	연관 대상	주요 필드/요소
EventListener	외부 이벤트를 수신하는 진입점. HTTP 엔드포인트 제공.	Trigger	serviceAccountName, triggers, bindings, template
Trigger	수신된 이벤트를 바탕으로 어떤 작업을 할지 정의.	TriggerBinding, Template	name, bindings, template, interceptors
TriggerBinding	이벤트에서 가져올 파라미터 정의.	Event, Trigger	params.name, params.value
Interceptor	이벤트 필터링 또는 인증 처리. (예: GitHub Webhook 검증)	EventListener, Trigger	webhook, params, ref
TriggerTemplate	PipelineRun 또는 TaskRun을 생성하는 템플릿.	Trigger	params, resourcetemplates
PipelineRun/TaskRun	실제 실행되는 파이프라인 또는 태스크 인스턴스.	TriggerTemplate	pipelineRef, params, workspaces

트리거 바인딩

트리거 바인딩 구성.

```
# vi tb-goblog-TriggerBinding.yaml
apiVersion: triggers.tekton.dev/v1beta1
kind: TriggerBinding
metadata:
  name: tb-goblog
spec:
  params:
    - name: gitrevision
      value: $(body.after)
    - name: gitrepositoryurl
      value: $(body.repository.clone_url)
```

트리거 템플릿

트리거 템플릿 구성.

```
# vi tb-goblog-TriggerTemplate.yaml
apiVersion: triggers.tekton.dev/v1beta1
kind: TriggerTemplate
metadata:
  name: tt-goblog
spec:
  params:
    - name: gitrevision
    - name: gitrepositoryurl
```

트리거 템플릿

트리거 템플릿 구성.

```
  resourcetemplates:
    - apiVersion: tekton.dev/v1
      kind: PipelineRun
      metadata:
        generateName: goblog-run-
  spec:
        serviceAccountName: sa-blog
        pipelineRef:
          name: goblog-pipeline
```

트리거 템플릿

트리거 템플릿 구성.

```
    params:
      - name: repo-url
        value: $(params.gitrepositoryurl)
      - name: repo-revision
        value: $(params.gitrevision)
      - name: image-url
        value: 192.168.10.40:5000/gookhyun/goblog:v1
```

트리거 템플릿

스토리지 클래스 기반으로 PVC 요청.

```
      workspaces:
        - name: shared-data
          volumeClaimTemplate:
            metadata:
              name: goblog-pvc
            spec:
              accessModes: ["ReadWriteOnce"]
              resources:
                requests:
                  storage: 1Gi
```

트리거 템플릿

미리 구성된 PVC로 생성 및 구성 시 사용.

```
      workspaces:
        - name: shared-data
          persistentVolumeClaim:
            claimName: goblog-pvc
```

파이프라인

파이프라인.

```
apiVersion: tekton.dev/v1
kind: Pipeline
metadata:
  name: goblog-pipeline
spec:
  params:
    - name: repo-url
    - name: repo-revision
    - name: image-url
  workspaces:
    - name: shared-data
```

파이프라인

파이프라인.

```
  tasks:
    - name: clone-source
      taskRef:
        name: git-clone
        kind: ClusterTask
      params:
        - name: url
          value: $(params.repo-url)
        - name: revision
          value: $(params.repo-revision)
        - name: deleteExisting
          value: "true"
```

파이프라인

파이프라인.

```
      workspaces:
        - name: output
          workspace: shared-data
```

파이프라인(이미지 빌드)

이미지 빌드 부분

```
    - name: build-image
      runAfter: [clone-source]
      taskRef:
        name: buildah
        kind: ClusterTask
      params:
        - name: IMAGE
          value: $(params.image-url)
        - name: CONTEXT
          value: $(workspaces.source.path)
        - name: TLSVERIFY
          value: "false"  # insecure registry 설정
```

파이프라인(이미지 빌드)

이미지 빌드 부분

```
      workspaces:
        - name: source
          workspace: shared-data
```

파이프라인

파이프라인.

```
    - name: deploy-to-k8s
      runAfter: [build-image]
      taskRef:
        name: kubernetes-admin
        kind: ClusterTask
      params:
        - name: script
          value: |
            kubectl rollout restart deployment go-blog -n default
```

이벤트 리스너

이벤트 리스너.

```
apiVersion: triggers.tekton.dev/v1beta1
kind: EventListener
metadata:
  name: el-goblog-listener
spec:
  serviceAccountName: sa-blog
  triggers:
    - name: goblog-trigger
      bindings:
        - ref: tb-goblog
      template:
        ref: tt-goblog
```

스토리지

PVC 저장소 구성.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: goblog-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

랩

최종학습 랩

2025-04-23