테크톤 101

TEKTON CD FOR KUBERNETES

2024-05-14

테크톤 랩 구성

TKN IN K8S

하이퍼브이 설치 및 구성

이 부분은 강사와 함께 진행한다.

테크톤 소개

CNCF TEKTON

TEKTON

테크톤은 CNCF에서 공식으로 지정한 Continuous Delivery도구이다. 테크톤은 엄밀히 말하자면 절차적인 자동화 도구라고 보시면 된다. 테크톤은 Continuous Integration기능도 제공하나, 엄밀하게 말하자만 SCM에 구성이 되어 있는 소스코드를 가져와서 이미지 빌드를 할 수 있도록 지원한다. 테크톤은 개발자에게 다음과 같은 기능 제공한다.

1. build
2. test
3. deploy

테크톤에서의 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 VS JENKINS

테크톤과 젠킨스는 기능 및 역할이 다르다. 젠킨스는 플러그인 및 혹은 로컬 언어(Local Language)를 사용해서 작업 순서를 구성한다. 젠킨스도 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 Groovy의 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"
                }
            ]
        }'''
```

빵야! ';=- **

TEKTON VS JENKINS

테크톤과 젠킨스의 제일 큰 차이점은 동작 방식이다. 젠킨스는 쿠버네티스와 통합된 형태가 아니고, 본래 자바에서 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/

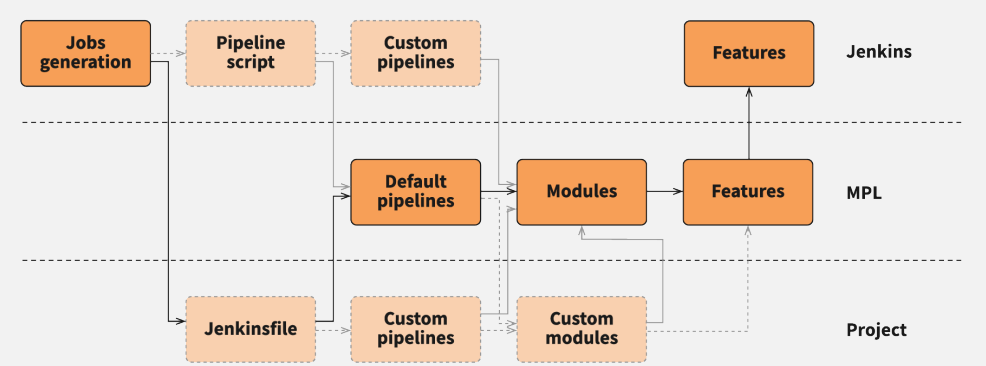
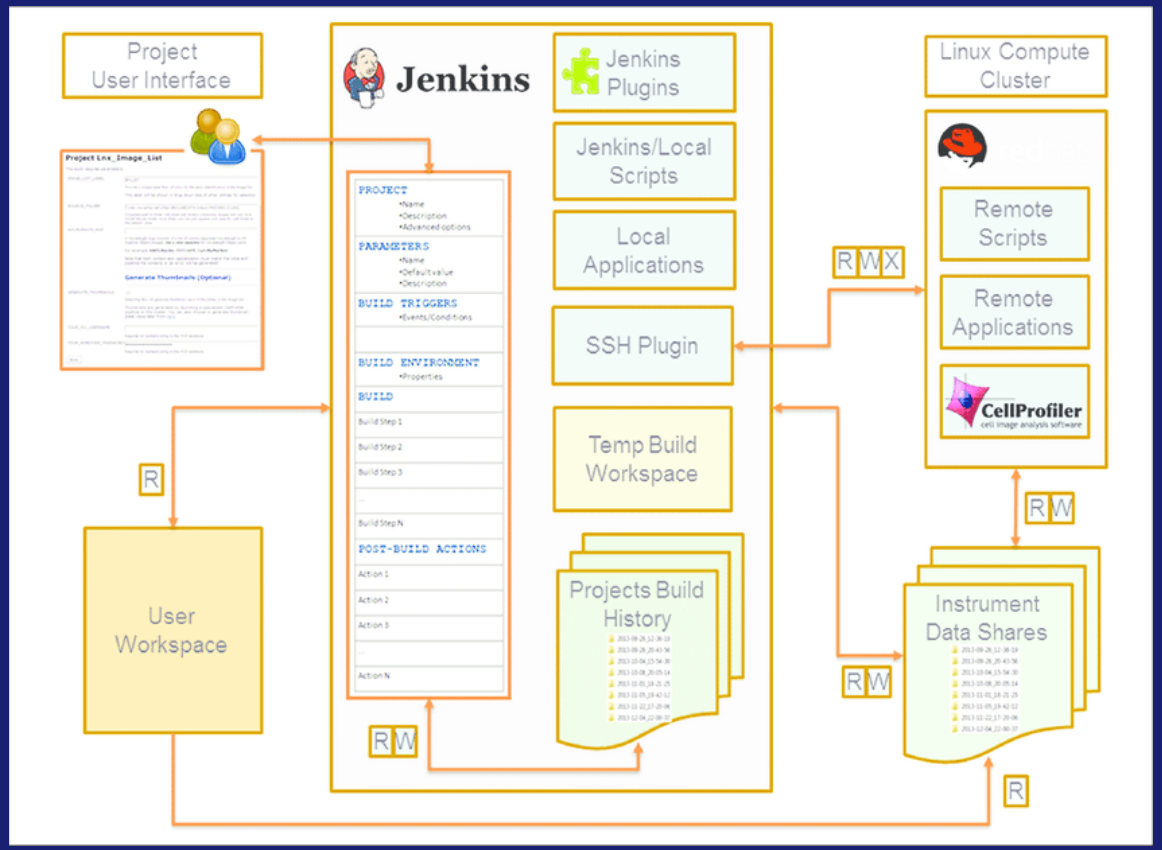
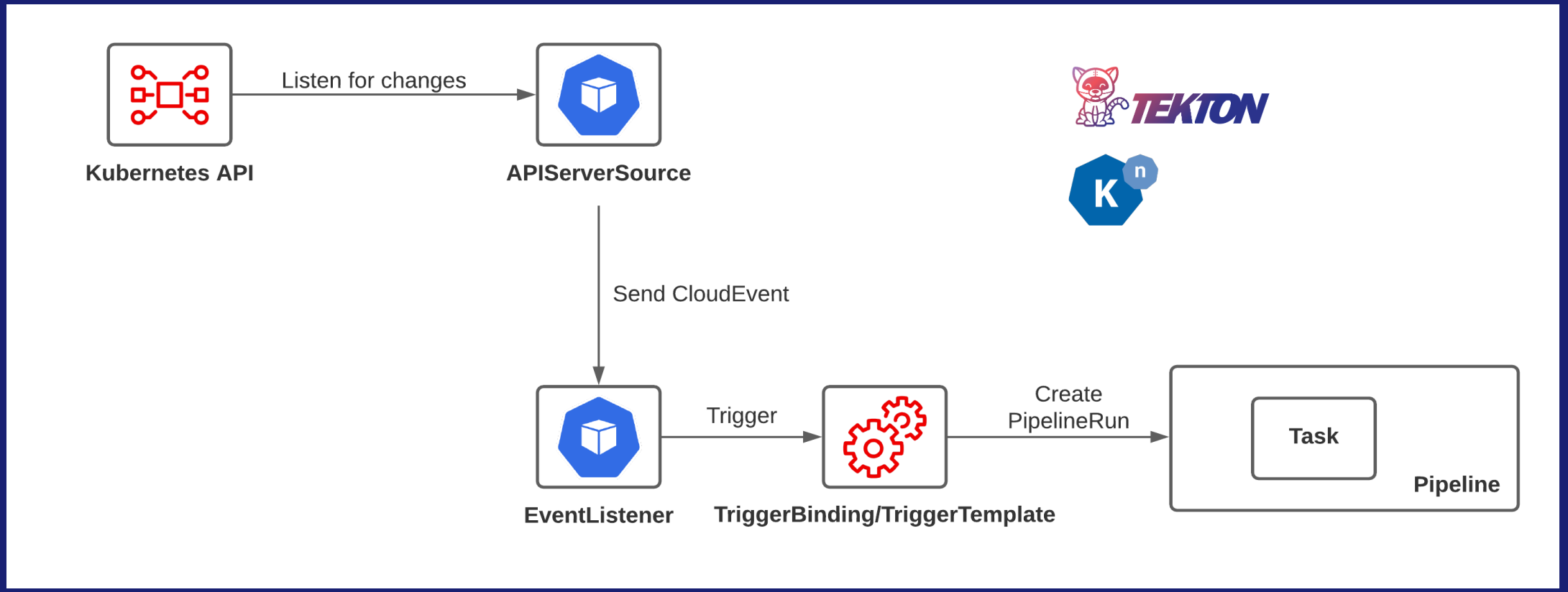


Fig 2. The MPL process

JENKINS


Project
User Interface
Jenkins
Jenkins
Plugins
Jenkins/Local
Scripts
Local
Applications
SSH Plugin
Temp Build
Workspace
Projects Build
History
Linux Compute
Cluster
Remote
Scripts
Remote
Applications
CellProfiler
cell image analysis software
Instrument
Data Shares
User
Workspace
PROJECT
PARAMETERS
BUILD TRIGGERS
BUILD ENVIRONMENT
BUILD
POST-BUILD ACTIONS
R
RW
RWX
RW
RW
R

TEKTON


Listen for changes
Kubernetes API
APIServerSource
TEKTON
Send CloudEvent
Trigger
Create
PipelineRun
Task
Pipeline
EventListener
TriggerBinding/TriggerTemplate

클라우드 네이티브에서 CI/CD

CI/CD

Jenkins/Tekton

2024-05-14

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 Abstration:** 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.
- **Plugable:** 쿠버네티스에 확장 기능 추가가 가능하다.

테크톤 설치

Installation on Kubernetes

가상머신 구성

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

쿠버네티스 클러스터 구성(OS)

```
master/worker]# cat <<EOF>> /etc/hosts
10.10.20.1 tknmaster.example.com tknmaster
10.10.20.2 tknworker.example.com tknworker
EOF
master]# nmcli con mod eth1 ipv4.address 10.10.20.1/16 ipv4.method manual
master]# nmtui edit eth1
master]# nmcli con up eth1
worker]# nmcli con mod eth1 ipv4.address 10.10.20.2/16 ipv4.method manual
worker]# nmtui edit eth1
worker]# nmcli con up eth1
worker/master]# ip a s eth1
master]# hostnamectl set-hostname tknmaster.example.com
worker]# hostnamectl set-hostname tknworker.example.com
```

쿠버네티스 클러스터 구성(OS)

```
master/worker]# cat <<EOF>> /etc/modules-load.d/k8s-modules.conf
br_netfilter
overlay
EOF
```

쿠버네티스 클러스터 구성(OS)

```
master/worker]# modprobe br_netfilter && modprobe overlay
master/worker]# lsmod | grep -e br_netfilter -e overlay
master/worker]# cat <<EOF> /etc/sysctl.d/99-k8s.conf
net.bridge.bridge-nf-call-iptables  = 1
net.ipv4.ip_forward                 = 1
net.bridge.bridge-nf-call-ip6tables = 1
EOF
master/worker]# sysctl --system
```

저장소 설정

```
master/worker]# cat <<EOF | tee /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://pkgs.k8s.io/core:/stable:/v1.28/rpm/
enabled=1
gpgcheck=1
gpgkey=https://pkgs.k8s.io/core:/stable:/v1.28/rpm/repodata/repomd.xml.key
EOF
```

쿠버네티스 도구 설치

```
master]# dnf install kubeadm kubelet kubectl -y
node]# dnf install kubeadm kubelet -y
```

CRI-O 저장소 설정

```
master/worker]# cat <<EOF | tee /etc/yum.repos.d/cri-o.repo
[cri-o]
name=CRI-O
baseurl=https://pkgs.k8s.io/addons:/cri-o:/prerelease:/main/rpm/
enabled=1
gpgcheck=1
gpgkey=https://pkgs.k8s.io/addons:/cri-
o:/prerelease:/main/rpm/repodata/repomd.xml.key
EOF
```

쿠버네티스 클러스터 구성(OS)

```
master]# dnf repolist
master]# dnf search cri-o
master/worker]# dnf install cri-o -y && systemctl enable --now cri-o
master/worker]# systemctl enable --now cri-o
master/worker]# systemctl is-active cri-o
> active
```

OS설정

```
master/worker]# systemctl stop firewalld && systemctl disable firewalld
master/worker]# systemctl disable firewalld
master/worker]# dnf install iproute-tc -y
master/worker]# sed -i '/ swap / s/^/#/' /etc/fstab
master/worker]# sed -i s/^SELINUX=.*$/SELINUX=permissive/
/etc/selinux/config
```

런타임 설정

```
master/node]# systemctl status cri-o
master/node]# systemctl disable firewalld && systemctl stop firewalld
master/node]# getenforce
master/node]# setenforce 0
master/node]# swapon -s
master/node]# swapoff -a
```

쿠버네티스 클러스터 구성

```
master/worker]# systemctl enable --now kubelet
master]# kubeadm init --apiserver-advertise-address=10.10.20.1 \
--pod-network-cidr=10.10.0.0/16 \
--service-cidr=10.90.0.0/16
> kubeadm join 192.168.90.250:6443 --token kzu7ci.jylu1yzdcwt85c20 \
> --discovery-token-ca-cert-hash \
> sha256:15a5b5e9c5463ca9c359ec96c8677ddd62615fe3afcf986e4b6703e6cbcdef0b
```

쿠버네티스 클러스터 구성

```
worker]# kubeadm join 192.168.90.100:6443 --token kzu7ci.jylu1yzdcwt85c20 \
        --discovery-token-ca-cert-hash
sha256:15a5b5e9c5463ca9c359ec96c8677ddd62615fe3afcf986e4b6703e6cbcdef0b
master]# export KUBECONFIG=/etc/kubernetes/admin.conf
master]# kubectl get pods -A
master]# kubectl get nodes
```

쿠버네티스 네트워크(calico)

```
master]# kubectl create -f
https://raw.githubusercontent.com/projectcalico/calico/v3.26.1/manifests/tigera-
operator.yaml

## 10분 정도 기다리고 다음단계 진행

master]# curl -o custom-resources.yaml
https://raw.githubusercontent.com/tangt64/duststack-k8s-auto/master/roles/cni/cni-
calico/templates/custom-resources.yaml

master]# vi custom-resources.yaml

> cidr: 10.10.0.0/16

master]# kubectl apply -f custom-resources.yaml

master]# kubectl get pods -Aw

## Calico POD가 전부 올라오면 테크톤 설치 진행
```

쿠버네티스 테크톤 설치

```
master]# kubectl apply --filename https://storage.googleapis.com/tekton-releases/pipeline/latest/release.yaml
master]# wget https://github.com/tektoncd/cli/releases/download/v0.32.0/tkn_0.32.0_Linux_x86_64.tar.gz
master]# kubectl apply --filename https://storage.googleapis.com/tekton-releases/dashboard/latest/release.yaml
master]# dnf install git -y
master]# kubectl get pods -n tekton-pipelines
```

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

```
master]# kubectl apply -f hello.yaml

master]# mkdir ~/bin/

master]# tar xzf tkn_0.32.0_Linux_x86_64.tar.gz -C ~/bin/

master]# tkn task list

> NAME    DESCRIPTION   AGE

> hello                 55 seconds ago

## 테크톤이 올바르게 동작하지 않으면 아래 명령어 실행

master]# kubectl delete validatingwebhookconfigurations.admissionregistration.k8s.io
config.webhook.pipeline.tekton.dev

master]# kubectl delete validatingwebhookconfigurations.admissionregistration.k8s.io
validation.webhook.pipeline.tekton.dev

master]# kubectl delete mutatingwebhookconfigurations.admissionregistration.k8s.io
webhook.pipeline.tekton.dev
```

에디터 설정(vim)

파일 작성을 원활하게 하기 위해서 yamllint, ale 그리고 neovim기반으로 진행한다. neovim과 vim+ale은 동작 방식이 다르기 때문에 선호하는 방식으로 선택 후 설치를 진행한다.

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

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

.nanorc

```
syntax "YAML" "\.ya?ml$"

header "^(---|===)" "%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+[|>]" "^\s*- "

color green "(^| )!!(binary|bool|float|int|map|null|omap|seq|set|str) "

color brightwhite "#.*$"

color ,red ":\w.+$"

color ,red ":'.+$"

color ,red ":".+$"

color ,red "\s+$"

color ,red "['\"][^['\"]]*$"

color yellow "['\"].*['\"]"

color brightgreen ":( |$)"
```

kubectl

```
master]# mkdir ~/.kube/
master]# cp /etc/kubernetes/admin.conf ~/.kube/config
master]# unset KUBECONFIG
master]# kubectl cluster-info
> Kubernetes control plane is running at https://192.168.90.250:6443
> CoreDNS is running at https://192.168.90.250:6443/api/v1/namespaces/kube-system/ser
>
> To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

kubectl(bash)

```
master]# kubectl completion bash > kubectl_completion.sh
master]# source kubectl_completion.sh
```

kubectl(zsh)

```
master]# dnf install zsh -y
master]# usermod -s /bin/zsh root
master]# sh -c "$(curl -fsSL
https://raw.githubusercontent.com/ohmyzsh/ohmyzsh/master/tools/install.sh)"
master]# kubectl completion zsh > ~/kubectl_completion.zsh
master]# source kubectl_completion.zsh
```

tkn CLI

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

tmux

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

Tekton dashboard

```
master]# kubectl apply --filename https://storage.googleapis.com/tekton-releases/dashboard/latest/release.yaml

master]# kubectl --namespace tekton-pipelines port-forward svc/tekton-dashboard 9097:9097

master]# kubectl --namespace tekton-pipelines port-forward svc/tekton-dashboard --address <ETH0_IP_ADDRESS> 9097:9097

host]# firefox http://<ETH0_IP_ADDRESS>:9097/
```

테크톤 자원

Stepping

Tasks

기본 자원 소개

- Step

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"
```

기본 자원 소개

- Tasks

PIPELINE

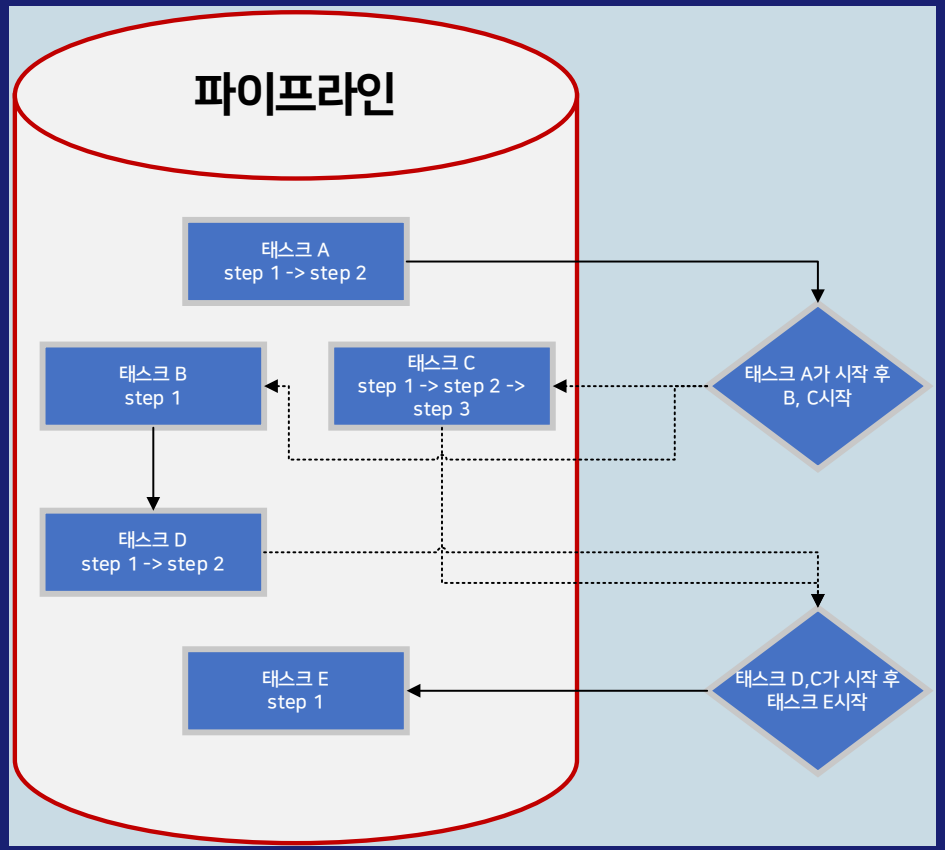
파이프 라인은 **작업의 순서를 정하는 자원**이다. 쉽게 생각하면 GW-BASIC처럼, 각각 작업에 순서를 걸어서 순차적으로 작업을 수행한다. 파이프 라인은 추상적인 자원이며, 이 자원은 기본적으로 태스크를 감싸고 있다.

PIPELINE

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
```

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.

TASK

태스크(tasks)는 파이프라인에 안에서는 작업에 대한 격리를 한다. 여기서 말하는 **작업**은 **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.

예를 들어서, 쿠버네티스 클러스터에서 Pod를 생성하기 위해서 이미지 구성 및 내려받기, 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 파이프라 인**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 소개

- Stepping
- Task
- Pipe

STEP

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

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

```
$ 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
```

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

```
$ 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 IN SCRIPT

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:
  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

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

```
$ 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.

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

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

TASK PARAMETER

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

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

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

TASK PARAMETER-1

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

TASK PARAMETER-1

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

TASK PARAMETER

```
$ 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
  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 'default'라는 변수 키워드를 사용한다. 미리 사용할 기본값을 미리 작성한다.

```
$ 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
  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
```

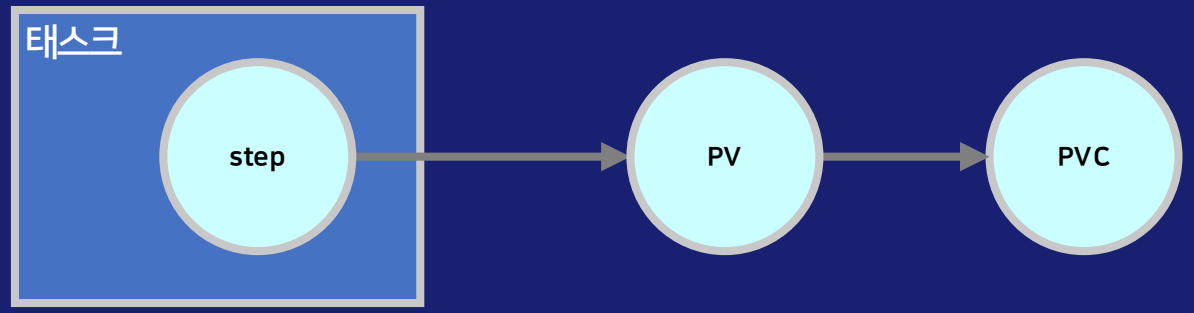
TASK DEFAULT PARAMETER

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
```

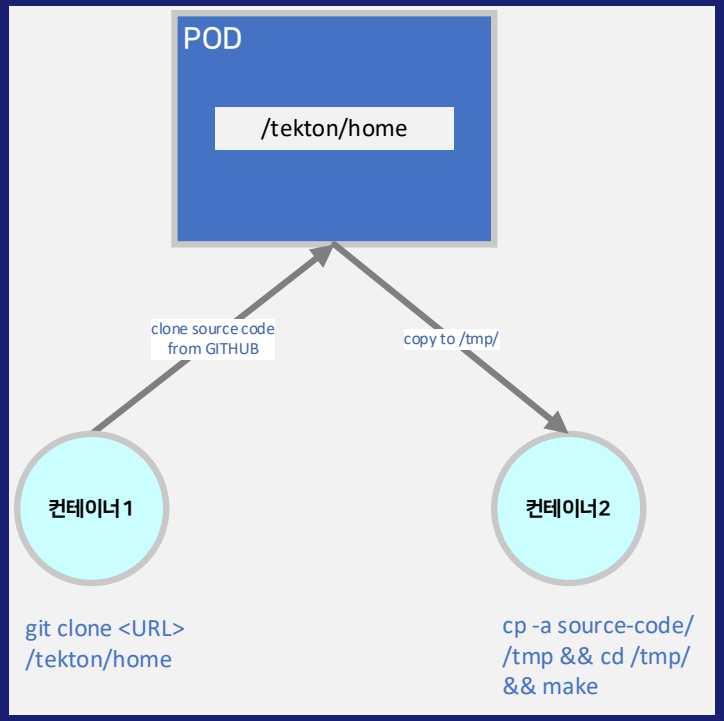
공유 데이터

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.

이름	설명
$HOME	테크톤 컨테이너 실행 시, 테크톤 홈 디렉터리 정보 가지고 있는 변수.
/tekton/home	테크톤 홈 디렉터리. 테크톤에서 사용하는 기본 홈 디렉터리.
/tekton/creds	테크톤에서 사용하는 secret와 같은 정보가 저장되는 위치.
/tekton/results	테크톤에서 실행된 결과 내용이 저장.
/workspace	워크 스페이스에서 생성된 내용 공유.

자세한 내용은 뒤에서 더 다루도록 한다. 기본적인 사용 방법은 다음과 같다.

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
```

```
[tree] /tekton/
[tree] |-- bin
[tree] |   `-- entrypoint
[tree] |-- creds
[tree] |-- home
[tree] |-- results
```

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-1

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

TEKTON RESUTL-2

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

TEKTON RESUTL-3

```
$ 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 SECRET

```
$ 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

```
$ tkn task start tekton-configmap --showlog

$ kubectl get taskrun tekton-configmap-run-<ID> -o yaml
```

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

기본 자원

연습문제

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:
```

파이프 라인 구성(task)

```
$ vi demo-pipeline1-task.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: demo-pipeline1-task
spec:
  steps:
    - image: quay.io/centos/centos:stream9
      command:
        - /bin/bash
      args: ['-c', 'echo I am the demo pipeline task']
```

파이프 라인 구성(task)

```
$ vi demo-pipeline1-pipeline.yaml
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: demo-pipeline1-pipeline
spec:
  tasks:
  - name: first
    taskRef:
      name: demo-pipeline1-task
```

파이프 라인 구성(task)

```
$ kubectl apply -f demo-pipeline1-task.yaml
$ kubectl apply -f demo-pipeline1-pipeline.yaml
$ tkn pipeline list
$ tkn task list
$ tkn task start demo-pipeline1-task --showlog
$ tkn pipeline start demo-pipeline1-pipeline --showlog
```

DICE TASK-1

```
$ vi demo-pipeline-dice-task.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: demo-pipeline-dice-task
spec:
  steps:
  - name: welcome-message
    image: quay.io/centos/centos:stream9
    command:
      - /bin/bash
    args: ['-c', 'echo Rolling dice']
```

DICE TASK-2

```
- name: dice-random
  image: quay.io/centos/centos:stream9
  script: |
    dnf install php-cli -y
    php -r "echo rand(1,6);"
```

WELCOME-MESSAGE TASK

```
$ vi demo-pipeline-welcome-message-task.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: welcome-message
spec:
  steps:
  - name: welcome-message
    image: quay.io/centos/centos:stream9
    command:
      - /bin/bash
    args: ['-c', 'echo "Welcome Gambling world!"']
```

DICE PIPELINE

```
$ vi demo-pipeline-dice-pipeline.yaml
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: demo-pipeline-dice-pipeline
spec:
  tasks:
  - name: message
    taskRef:
      name: welcome-message
  - name: roll
    taskRef:
      name: demo-pipeline-dice-task
```

DICE RUN

```
$ kubectl apply -f demo-pipeline-dice-task.yaml
$ kubectl apply -f demo-pipeline-dice-pipeline.yaml
$ kubectl apply -f demo-pipeline-welcome-message-task.yaml
$ tkn task list
$ tkn pipeline list
$ tkn task start welcome-message --showlog
$ tkn task start demo-pipeline-dice-task --showlog
$ tkn pipeline start demo-pipeline-dice-pipeline --showlog
$ tkn pipeline logs
```

파이프라인 파라메터(DEMO)

파이프라인을 지속적으로 반복하다 보면, 같은 작업에 다른 요청 값을 통해서 작업이 수행이 된다. 이러한 형태는 쿠버네티스에서 빈번하게 발생이 된다. 테크톤 기반의 CI/CD에서는 이를 파라메터(Parameter)를 통해서 쉽게 구현이 가능하다.

```
$ vi demo-pipeline-dice-pipeline2-task-param.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: demo-pipeline-dice-pipeline2-task-param
spec:
  params:
    - name: sideNum
      description: number of sides to the dice
      default: 6
      type: string
```

파이프라인 파라메터(DEMO)

```
  steps:
  - name: welcome-message-param
    image: quay.io/centos/centos:stream9
    command:
      - /bin/bash
      - -c
      - echo "Rolling $(params.sideNum) dice"
- name: dice-random-param
    image: quay.io/centos/centos:stream9
    script: |
      dnf install php-cli -y
      php -r 'echo rand(1,$(params.sideNum));'
```

DICE 파이프라인 2

```
$ vi demo-pipeline-dice-pipeline2.yaml
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: demo-pipeline-dice-pipeline2
spec:
  tasks:
  - name: first
    taskRef:
      name: hello
  - name: roll
    taskRef:
      name: demo-pipeline-dice-pipeline2-task-param
```

파이프라인 파라메터

```
$ kubectl apply -f demo-pipeline-dice-pipeline2.yaml
$ kubectl apply -f demo-pipeline-dice-pipeline2-task-param.yaml
$ tkn pipeline start demo-pipeline-dice-pipeline2 --showlog
```

파이프라인 파라메터 변경

```
$ vi demo-pipeline-dice-pipeline2-task-param.yaml
> default: 6 -> 8
$ kubectl apply -f emo-pipeline-dice-pipeline2-task-param.yaml
$ tkn pipeline start demo-pipeline-dice-pipeline2 --showlog
```

DICE 파이프라인 3

```
$ vi demo-pipeline-dice-pipeline3.yaml
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: demo-pipeline-dice-pipeline3
spec:
  params:
  - name: sidenum
    type: string
    default: 6
    description: number of sides the dice
```

DICE 파이프라인 3

```
tasks:
- name: first
  taskRef:
    name: hello
- name: roll
  params:
  - name: sides
    value: "$(params.sidenum)"
  taskRef:
    name: demo-pipeline-dice-task
```

파이프라인 파라메터

```
$ kubectl apply -f demo-pipeline-dice-pipeline3.yaml
$ tkn pipeline start demo-pipeline-dice-pipeline3 --showlog
$ tkn pipeline start demo-pipeline-dice-pipeline3 -p sidenum=20 --showlog
$ tkn pipeline start demo-pipeline-dice-pipeline3 --use-param-defaults --
showlog
```

REUSE PIPELINE DEMO

```
$ vi demo-pipeline-reuse-task.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: demo-pipeline-reuse-task
spec:
  params:
  - name: text
    type: string
  steps:
  - name: log
    image: quay.io/centos/centos:stream9
    script: |
      DATE=$(date +%d/%m/%Y\ %T)
      echo [$DATE] - $(params.text)
```

REUSE PIPELINE DEMO(pipeline)

```
$ vi demo-pipeline-reuse-pipeline.yaml
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: demo-pipeline-reuse-pipeline
spec:
  tasks:
  - name: say-hello
    taskRef:
      name: demo-pipeline-reuse-task
    params:
    - name: text
      value: "Hello"
```

REUSE PIPELINE DEMO(pipeline)

```
- name: log-something
  taskRef:
    name: hello
  params:
  - name: text
    value: "Something else being logged"
```

REUSE PIPELINE DEMO(RUN)

```
$ kubectl apply -f demo-pipeline-reuse-task.yaml
$ kubectl apply -f demo-pipeline-reuse-pipeline.yaml
$ tkn pipeline start demo-pipeline-reuse-pipeline --showlog
```

PIPELINE ORDERING

파이프라인 소개 시, 파이프라인은 자유롭게 작업 순서를 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

```
  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)']
    - 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']
```

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

PIPELINE ORDERING - PIPELINE

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

PIPELINE ORDERING - RUN

아래와 같이 명령어를 실행한다.

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

2024-05-14

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
  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
  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
```

PIPELINE RUNS

```
$ tkn pipeline start demo-pipeline-dice-task-param-result
$ kubectl get pipelineruns
>
$ kubectl delete pipelineruns <ID>
>
$ kubectl get pipelineruns <ID> -o yaml
```

파이프라인 연습문제

연습문제

2024-05-14

디버깅

디버깅

파이프라인 및 태스크 정리

2024-05-14

디버깅

테크톤에서 디버깅 방식은 두가지가 있다.

1. 'kubectl'명령어를 통한 디버깅
2. 'tkn'명령어를 통한 디버깅

tkn명령어 기반으로 하는 경우, 실행 시, 어떠한 태스크, 파라메터를 찾지 못하는지 메시지를 통해서 확인이 가능하지만, 이미지나 혹은 컨테이너에서 발생한 오류 메시지는 확인이 불가능하다.

그래서 kubectl명령어를 통해서 좀 더 자세하게 확인을 해야 한다.

```
$ kubectl get tasks
$ kubectl get pipeline
$ kubectl describe tasks <POD_ID>
$ kubectl logs tasks <POD_ID>
```

DEBUG-TASK

```
$ 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

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

DEBUG-TASK

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

DEBUG-TASK

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

DEBUG-TASK RUN 1

```
$ 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

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

FINALLY/CLEANUP

Finally, Cleanup작업은 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

```
$ 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

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

디버깅 연습문제

연습문제

2024-05-14

워크스페이스

데이터 공유

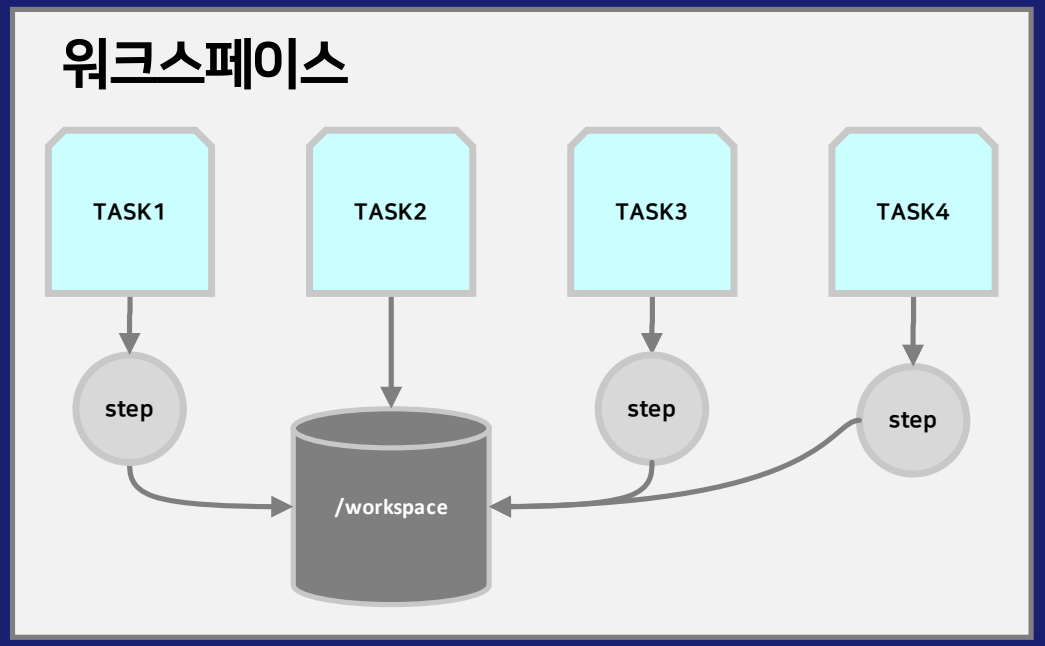
워크스페이스

소개

워크스페이스

워크스페이스는 테크톤에서 작업 시 사용하는 일종의 **워크벤치(workbench)**이다. 워크벤치는 태스크에 구성되어 있는 **단계(step)**들에서 발생한 데이터를 서로 공유가 가능하다.

이러한 방식으로 공유를 해야 하는 이유는, 특정 태스크에서 발생한 대 용량 데이터를 다른 컨테이너, 즉 **작업(task)**에서 사용하려는 경우, 반드시 워크스페이스를 통해야 한다.



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 형태이외, Persistent Volume, Persistent Volume Claim를 지원한다. PV/PVC를 사용한 경우, 앞에서 학습하였던 finally를 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:
    - name: source
steps:
    - name: clone
      image: quay.io/centos/centos:stream9
      workingDir: $(workspaces.source.path)
```

"workspaces:"부분은 워크스페이스에서 생성한 이름을 보통 적는다.

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

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

2024-05-14

워크스페이스

저장소 형식

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`

PV/PVC경우에는 최소 NFS3/4기반으로 구성해야 되며, 스토리지 클래스(StorageClass)으로 구성한다. 기본적으로 **`NFS CSI(Container Storage Interface)`**가 구성이 안되어 있다는 조건으로 "emptyDir"를 사용한다.

NFS CSI설치를 원하는 경우, 앞서 쿠버네티스 클러스터 설치에서 추가적으로 진행하도록 한다.

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
```

워크스페이스

첫 워크스페이스 생성

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. 앤서블이 실행이 가능하도록 패키지를 설치한다.(RPM, PIP)
3. 복제한 코드를 앤서블 명령어로 간단하게 코드 검증 및 확인한다.

간단한 워크스페이스 예제

저장소는 간단하게 emptyDir를 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

각 작업을 아래와 같이 **구성/설치 및 실행**한다.

```
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)
  - 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`? (Default is `https://github.com/tangt64/duststack-osp-auto`) https://github.com/tangt64/duststack-osp-auto
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 :
```

워크스페이스

워크스페이스에서 작업 실행(task runs) 사용하기

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
```

이 명령어를 다음처럼 변경이 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
```

워크스페이스

워크스페이스 파이프라인 추가

2024-05-14

파이프라인에 워크스페이스

파이프라인에 워크스페이스를 추가해서 사용하면, 좀 더 손쉽게 반복적으로 사용이 가능하다. 기존에 사용하였던 깃헙 코드를 조금 더 단순하게 만들어서 테스트를 한다.

```
# 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
  workspaces:
    - name: source-pipeline
 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)
```

파이프라인에 워크스페이스

위의 파일에 계속 이어서 작성한다.

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

```
---
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: demo-clone-list-from-pipeline
spec:
  workspaces:
    - name: codebase
  tasks:
    - name: clone
      taskRef:
        name: clone
    workspaces:
      - name: source
        workspace:  codebase
```

파이프라인에 워크스페이스

위에서 작성한 태스크를 파이프 라인으로 생성 및 구성한다.

```
  - name: list
      taskRef:
        name: list
    workspaces:
      - name: source
        workspace: codebase
    runAfter:
      - clone
```

워크스페이스

파이프라인에 영구적 데이터 저장소 구성

2024-05-14

2024-05-14

워크스페이스

최종(finally)를 사용한 작업 정리

2024-05-14

2024-05-14

워크스페이스

파이프라인 실행(pipeline runs) 사용하기

2024-05-14

워크스페이스

템플릿에 쿠버네티스 볼륨 클레임 사용하기

2024-05-14

2024-05-14

조건식

조건식 활용 및 사용하기

2024-05-14

조건식

소개

2024-05-14

조건식

조건식, 파라메터 사용하기

2024-05-14

2024-05-14

조건식

조건식, 파라메터 사용하기

2024-05-14

2024-05-14

조건식

값 입력 받기

2024-05-14

조건식

결과값 기반으로 조건식 작성

2024-05-14

인증

기본 인증 및 SSH

2024-05-14

조건식

소개

트리거

조건식 활용 및 사용하기

2024-05-14

트리거

소개

2024-05-14

트리거

설치

2024-05-14

트리거

로컬 클러스터

2024-05-14

트리거

트리거 템플릿/바인딩/이벤트 리스너

2024-05-14

2024-05-14

트리거

트리거 수행 및 실행

2024-05-14

랩

최종학습 랩

2024-05-14