Welcome

Knowledge Prerequisites

Prerequisites

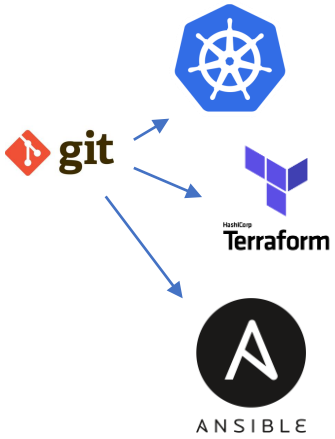
Containers

kubernetes

What is GitOps

What is GitOps

1. Infrastructure as code, Configuration as code.

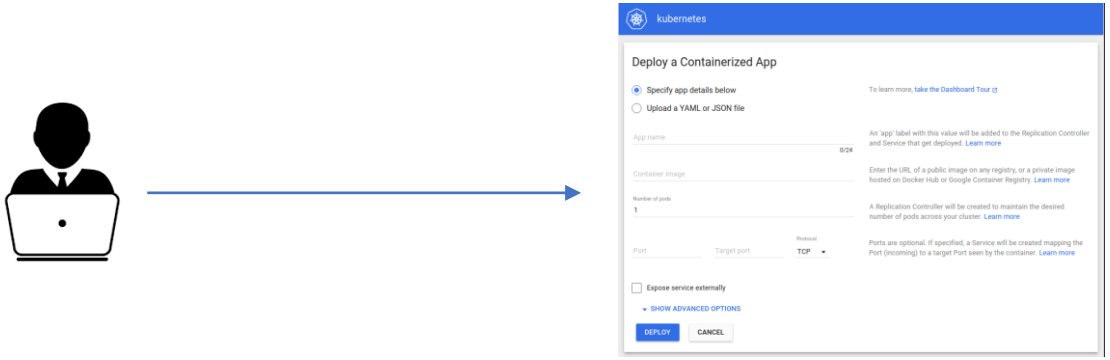


2. Collaboration + change mechanism (Code review)

3. Continues integration and continues delivery pipelines

Before

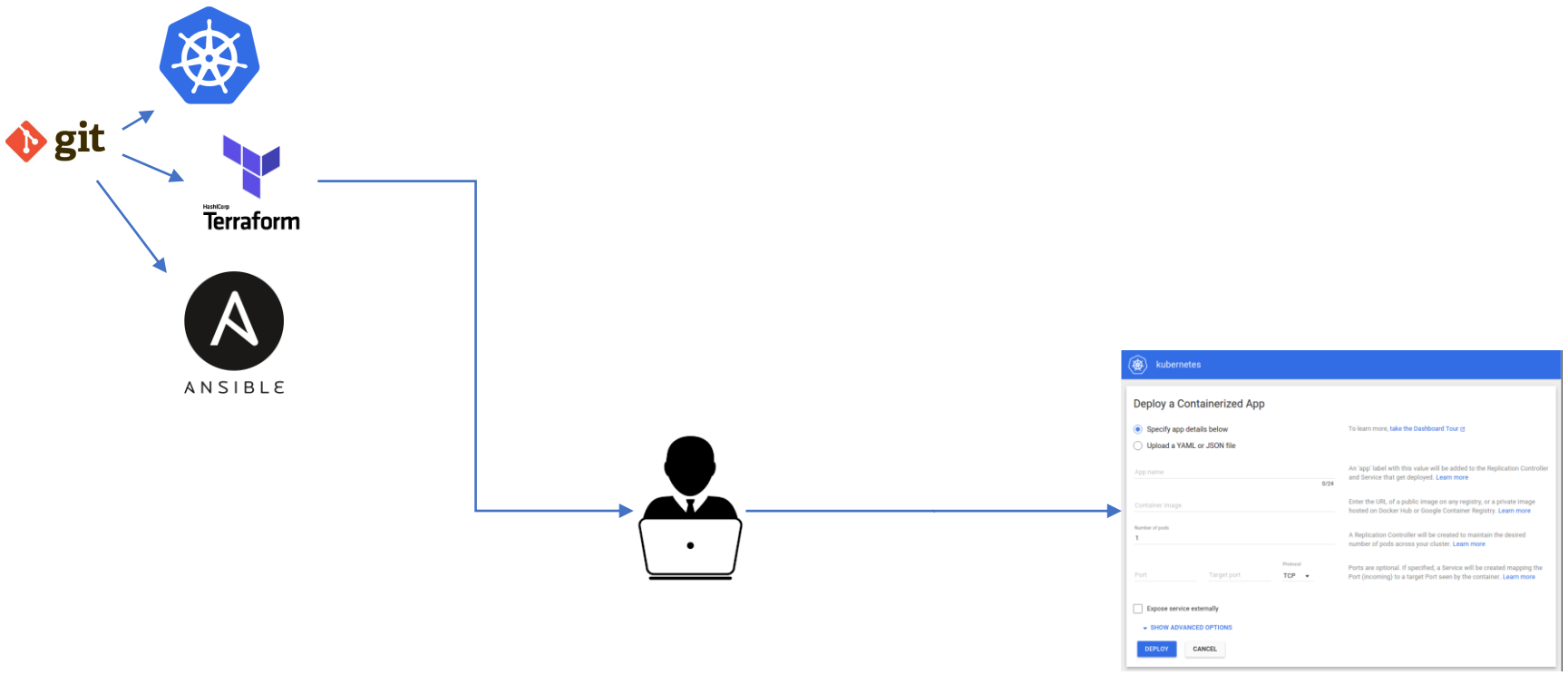
- Manually creating/configuring resources



❌ You cannot reproduce the same infrastructure or environments.

Bad practices

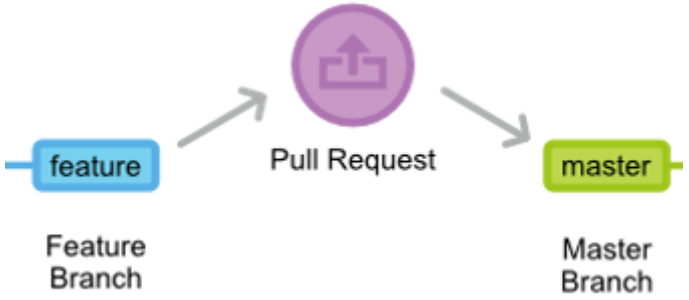
- IaC in git but applied manually without CI/CD



Why GitOps

- Versioned IaC.
- Collaboration by code reviews.
- CI : automated validation and tests.
- CD: automated deployment.

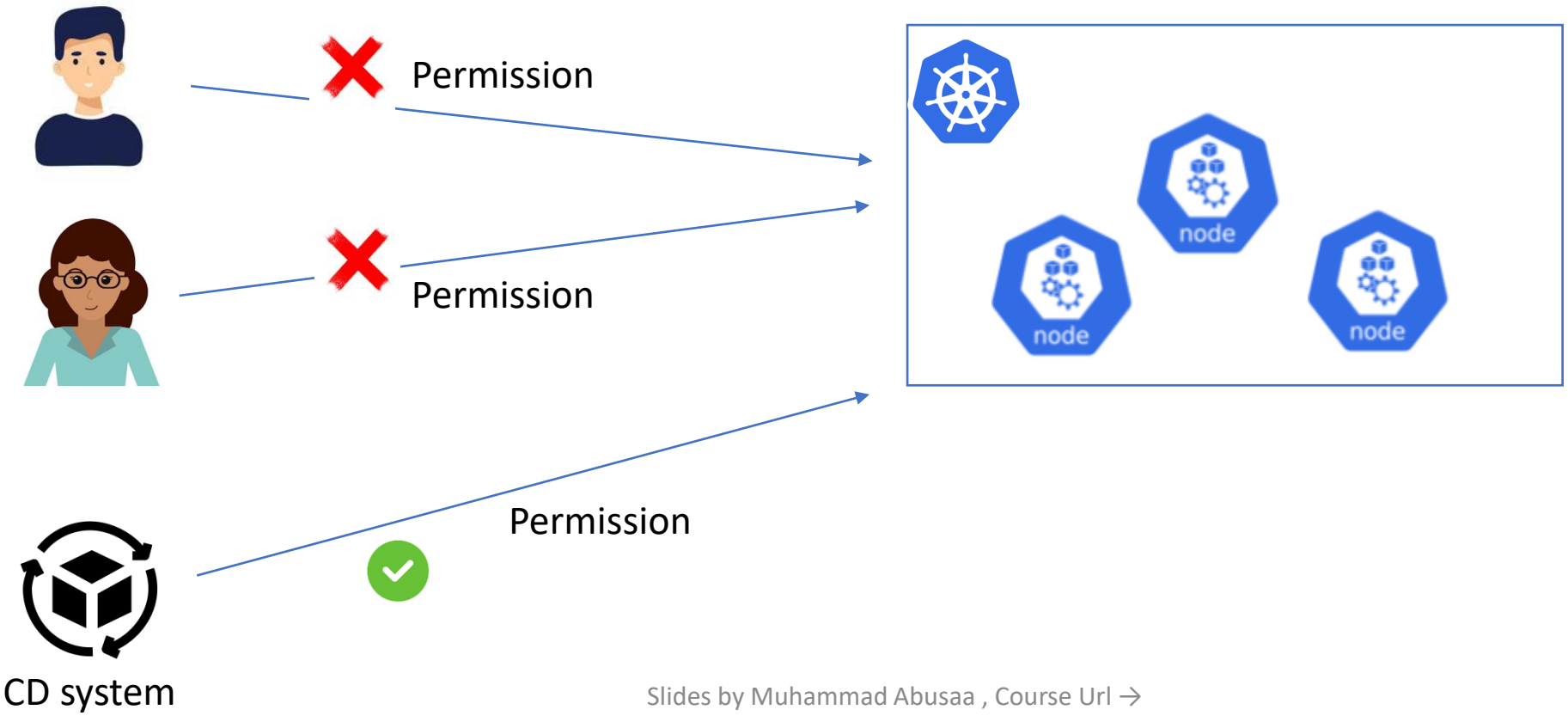
+



+

Why GitOps

- Improved Security.

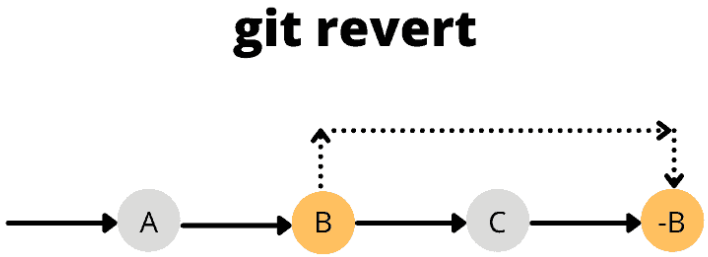


Why GitOps

- Rollbacks.

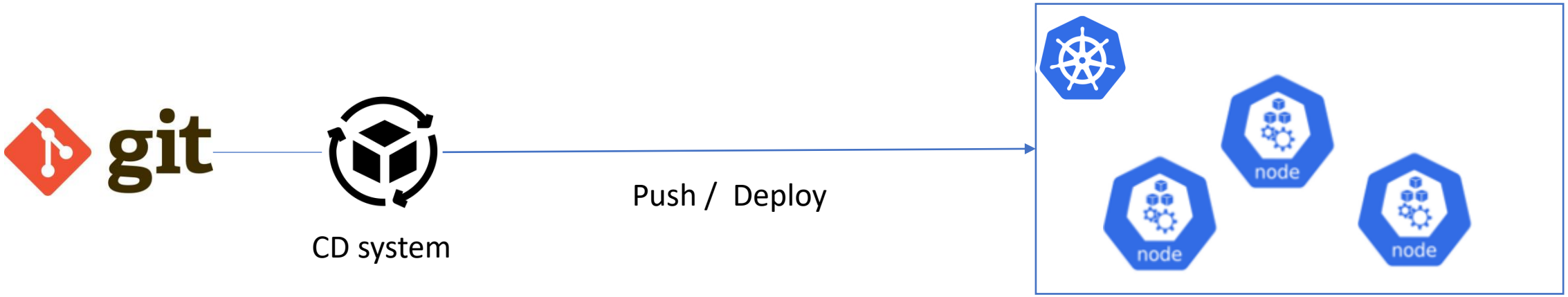
Bad version deployed

Easy rollback with git revert



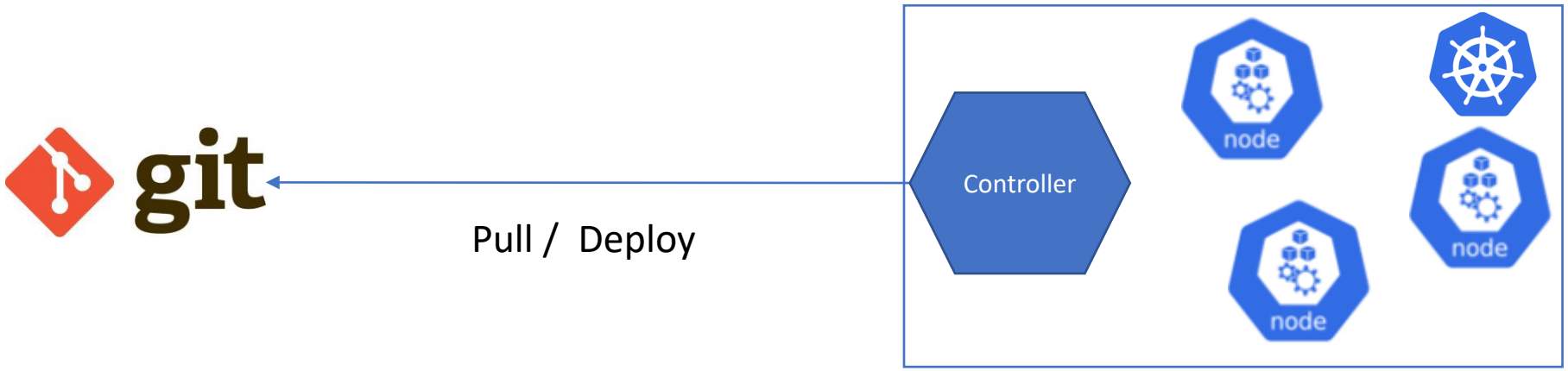
GitOps Push vs Pull model

- Push model



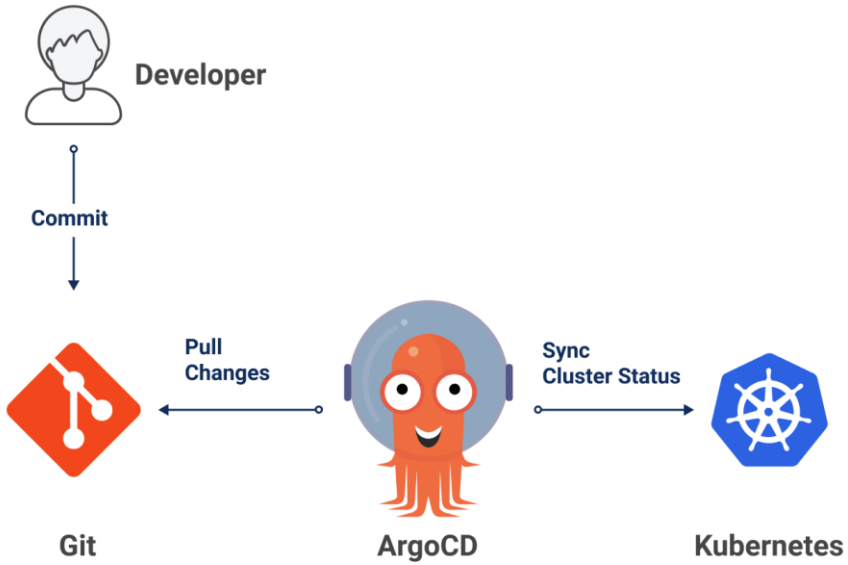
GitOps Push vs Pull model

- Pull model



ArgoCD

- ArgoCD Is a Gitops-based CD with pull model design



Intro to ArgoCD

Slides by Muhammad Abusaa , Course Url →
https://bit.ly/3JRtKKS

What is ArgoCD

- ArgoCD is a GitOps continues delivery tool for Kubernetes.

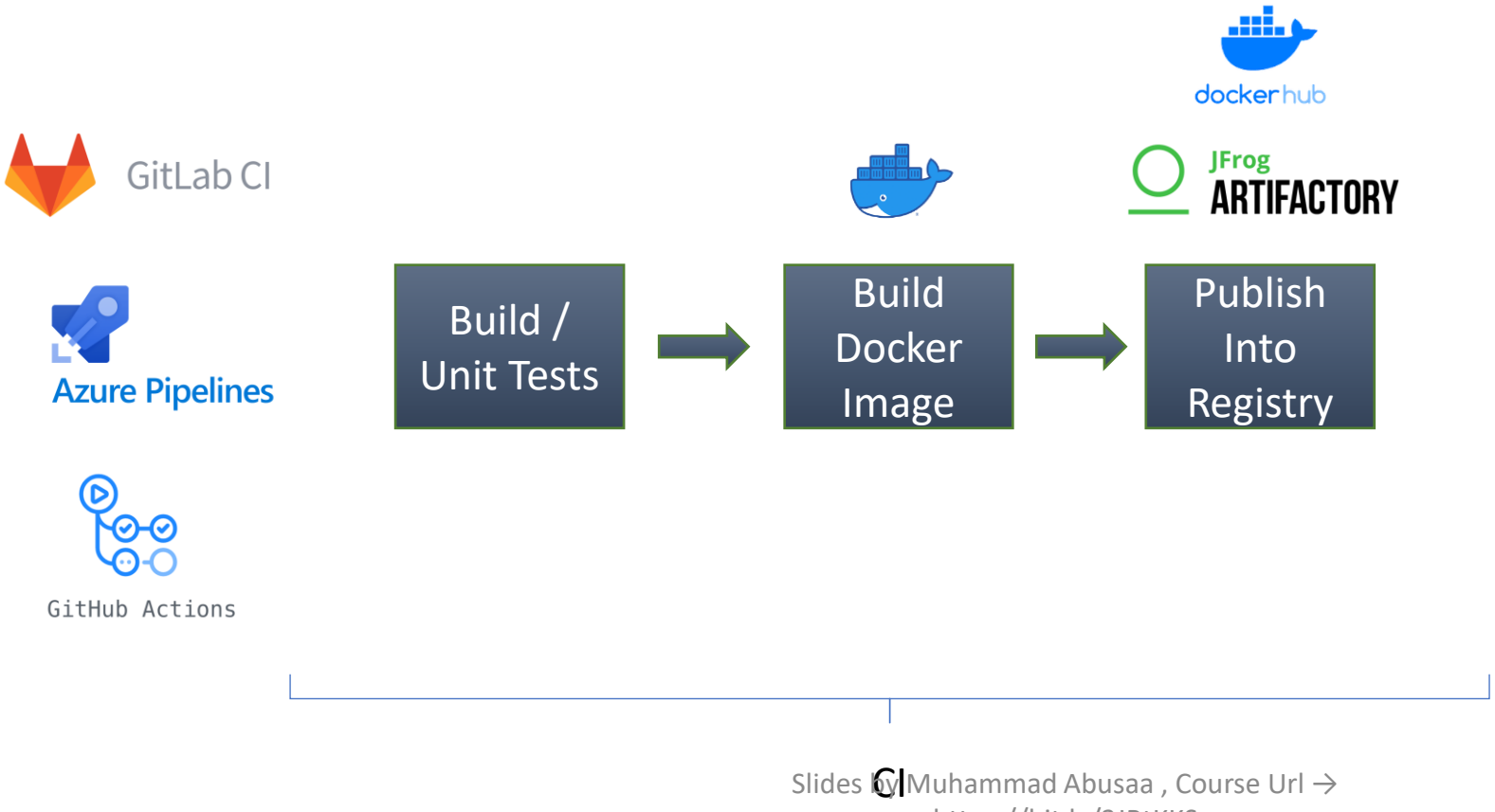
Git Repo
(Source of Manifests)

ArgoCD

(Destination) Cluster

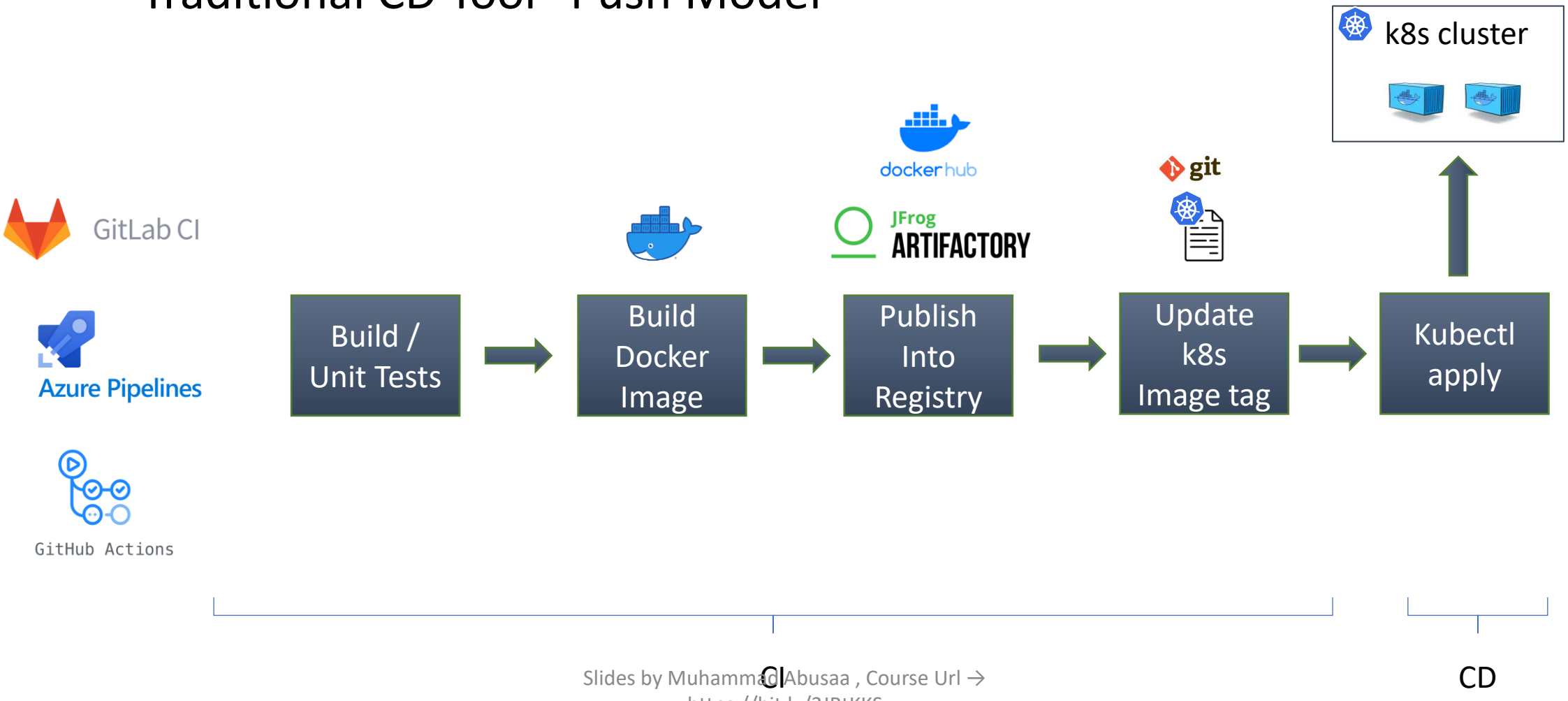
Is ArgoCD a CI tool ?

- No



ArgoCD vs Traditional CD tools

- Traditional CD Tool “Push Model”



ArgoCD vs Traditional CD tools

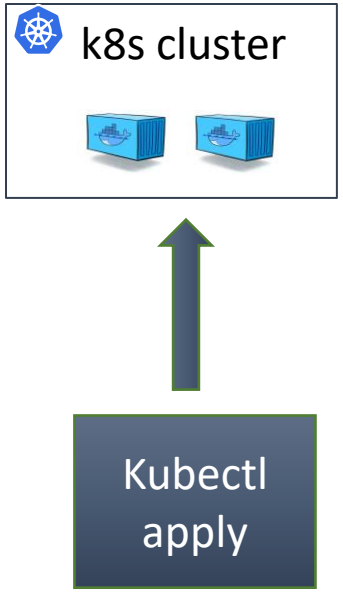
- Traditional CD Tool “Push Model”

GitLab CI

Azure Pipelines

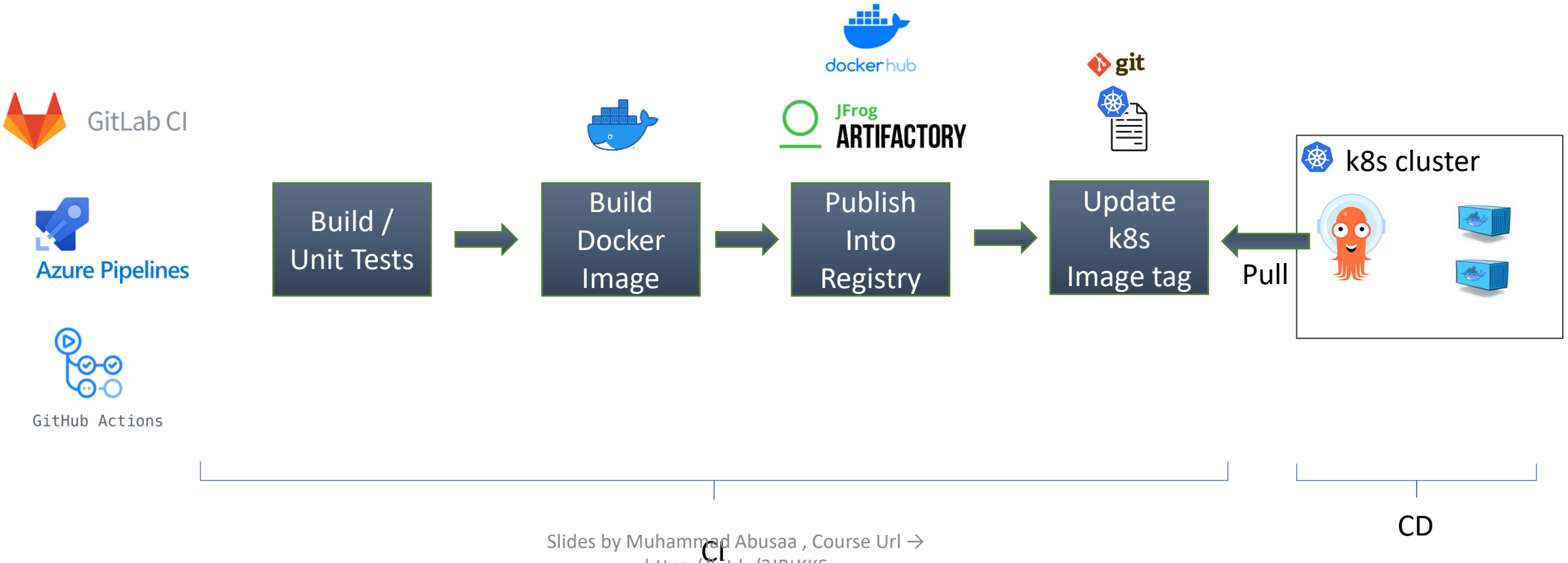
GitHub Actions

- Install needed clients on CICD agents.
  - Kubectl
  - Helm
- Grant access in CICD system to apply deployments into k8s cluster.
  - Configure kubeconfig
- Open connectivity between CICD systems and your k8s cluster “security concern”.



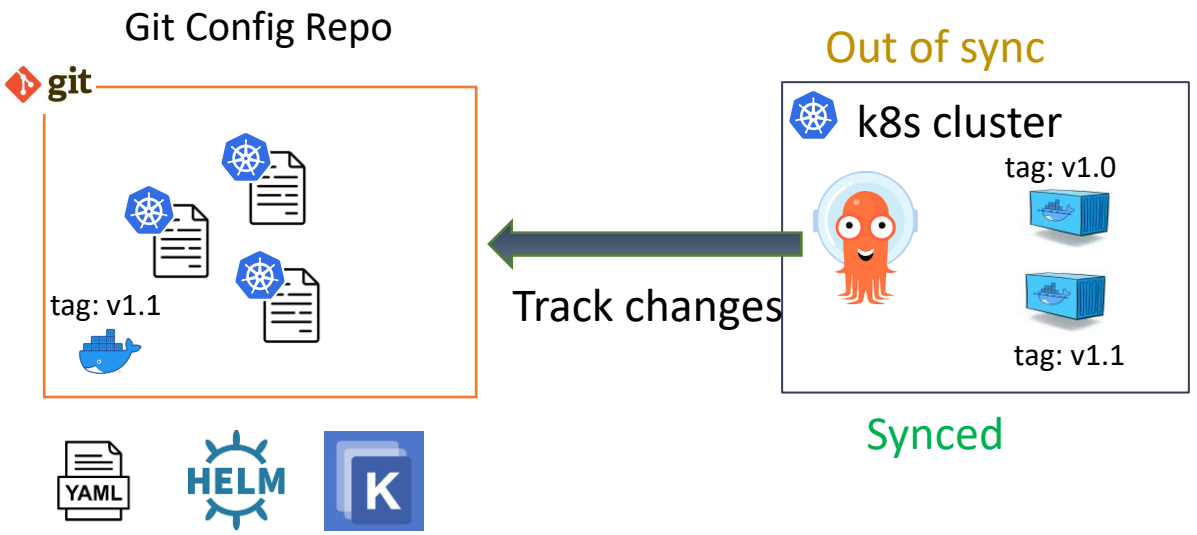
ArgoCD vs Traditional CD tools

- ArgoCD



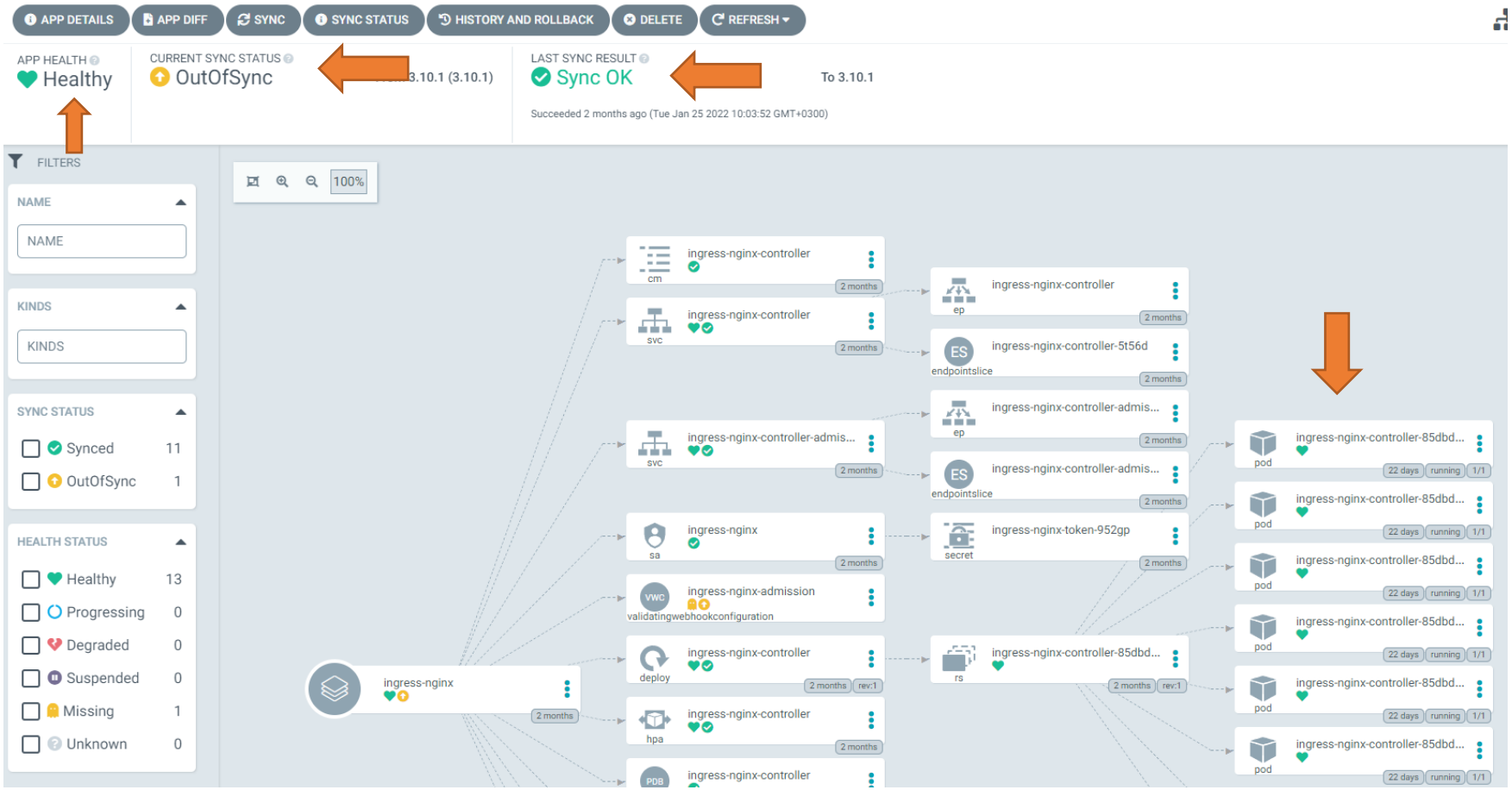
ArgoCD vs Traditional CD tools

- ArgoCD



Why ArgoCD

- State Visibility
  - Last Sync status
  - Health status
  - Current Status
  - Real time updates



Slides by Muhammad Abusaa , Course Url →
https://bit.ly/3JRtKKS

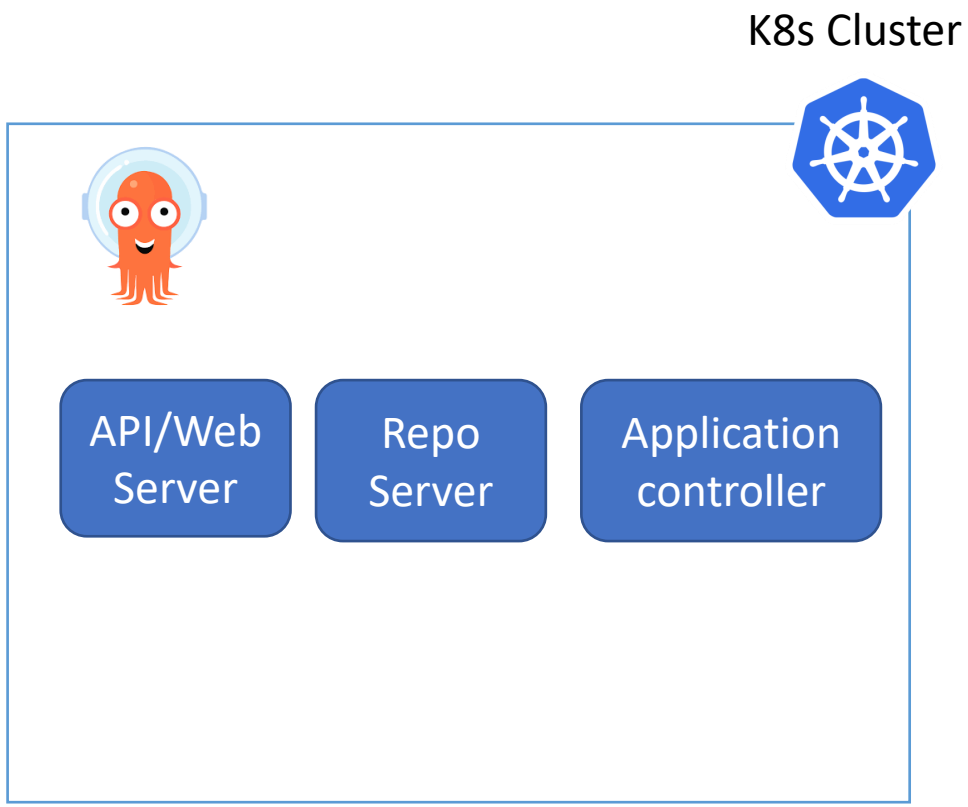
Why ArgoCD

- Git as the source of truth.
  - Developer and DevOps engineer will update the Git code only.
- Keep your cluster in sync with Git.
- Easy rollback.
- More security : Grant access to ArgoCD only.
- Disaster recovery solution : You easily deploy the same apps to any k8s cluster.

ArgoCD Architecture

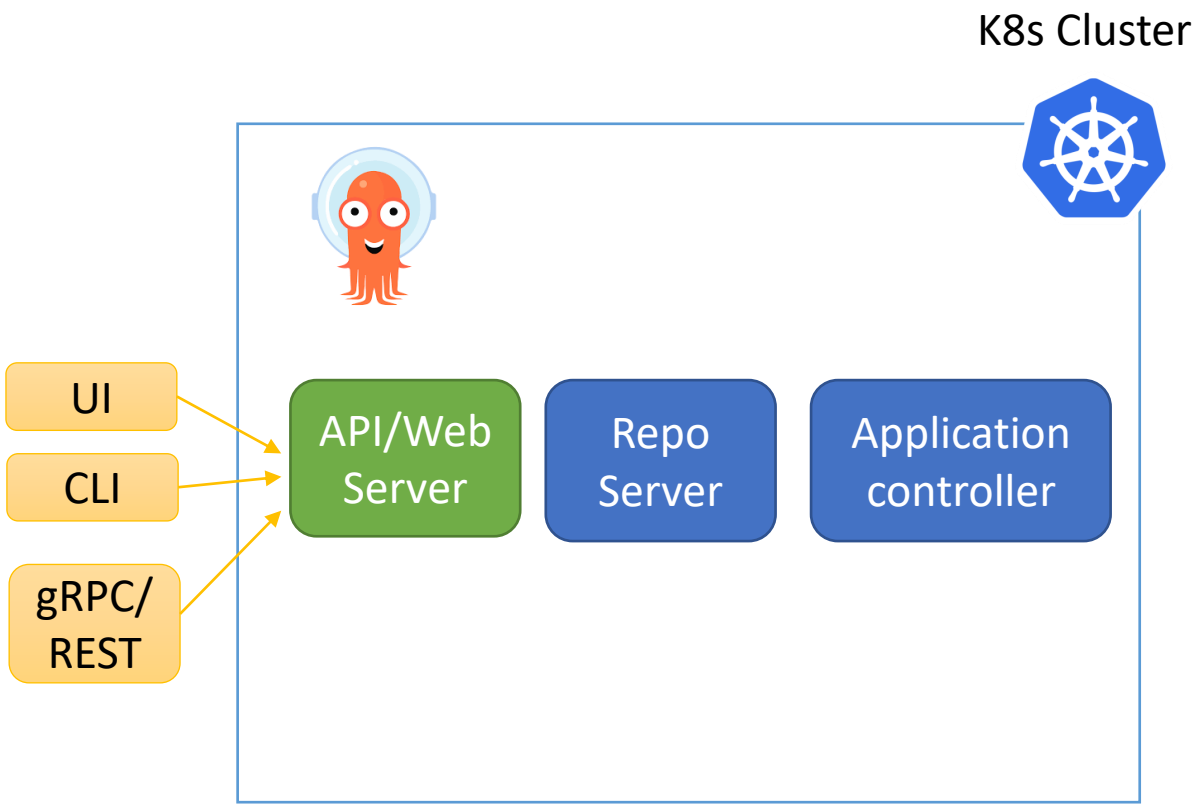
Components

- ArgoCD consist of 3 main components
  - ArgoCD Server (API + Web Server).
  - ArgoCD Repo Server.
  - ArgoCD Application Controller.



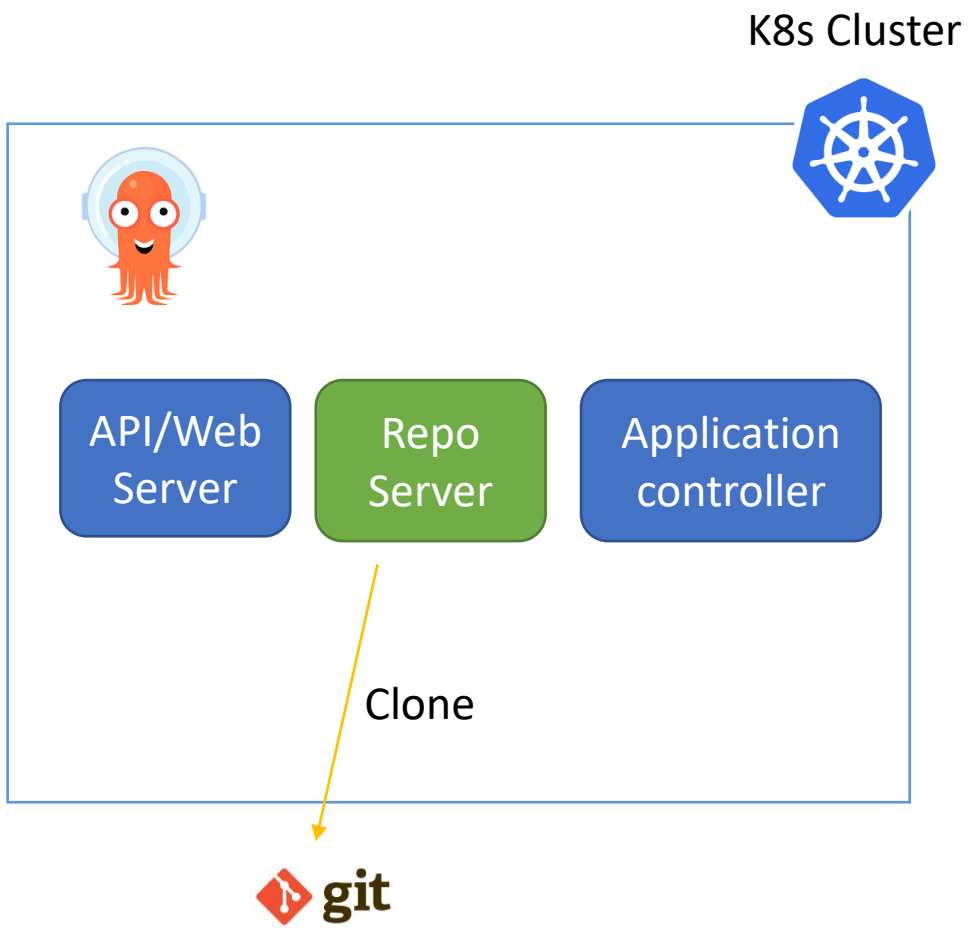
Components / API – Web Server

- Its a gRPC/REST server which exposes the API consumed by the Web UI, CLI.
  - Application management (Create, Update, Delete).
  - Application operations (ex: Sync, Rollback)
  - Repos and clusters management.
  - Authentication.



Components / Repo Server

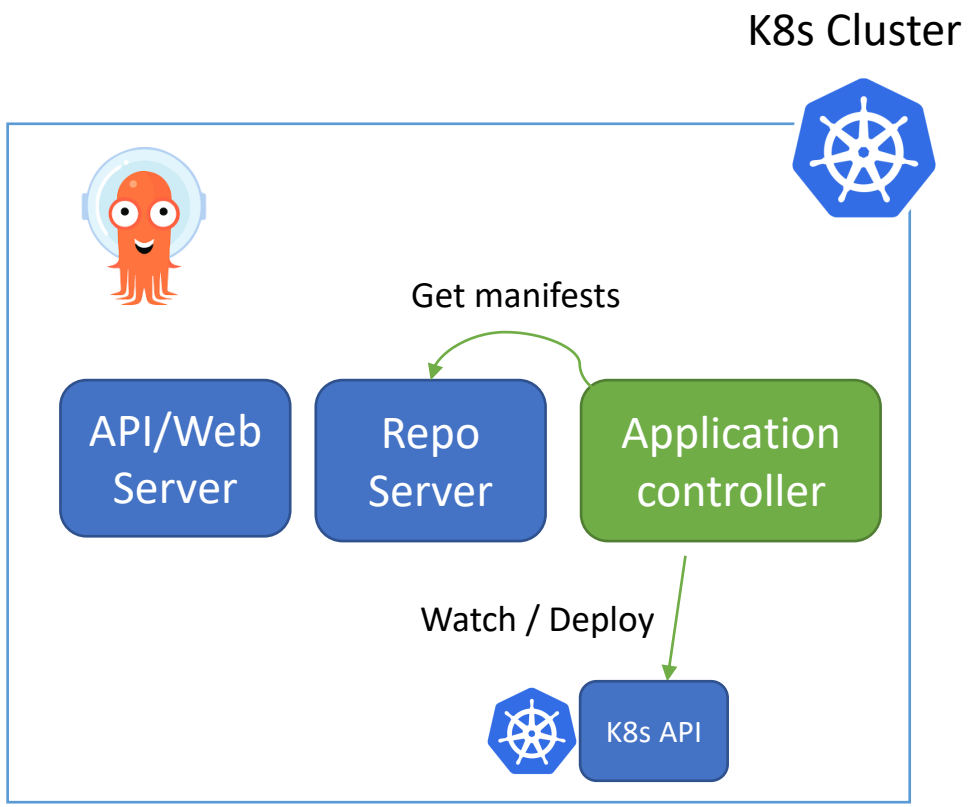
- Its an internal service that responsible of cloning remote git repos and generate the needed k8s manifests.
    - Clone git repo.
    - Generate k8s manifests.



Components / Application controller

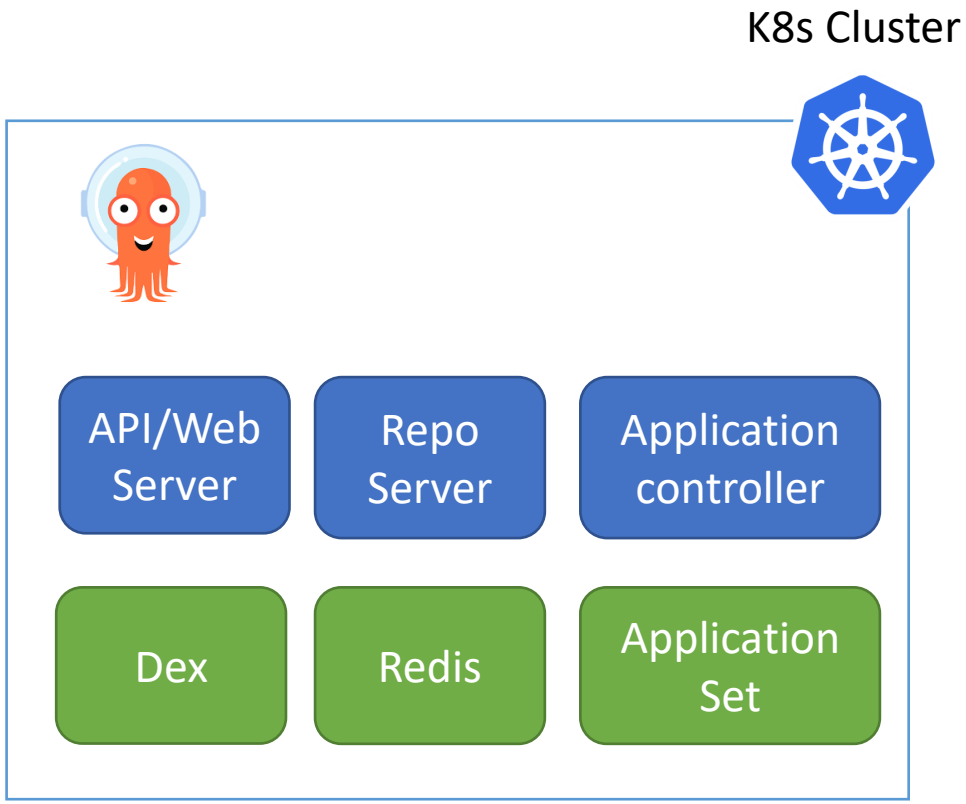
- Its a Kubernetes controller which continuously monitors running applications and compares the current, live state against the desired target state.

  - Communicate with Repo server to get the generated manifests.
  - Communicate with k8s API to get actual cluster state.
  - Deploy apps manifests to destination clusters.
  - Detects OutofSync Apps and take corrective actions “If needed”.
  - Invoking user-defined hooks for lifecycle events (PreSync, Sync, PostSync).



Additional Components

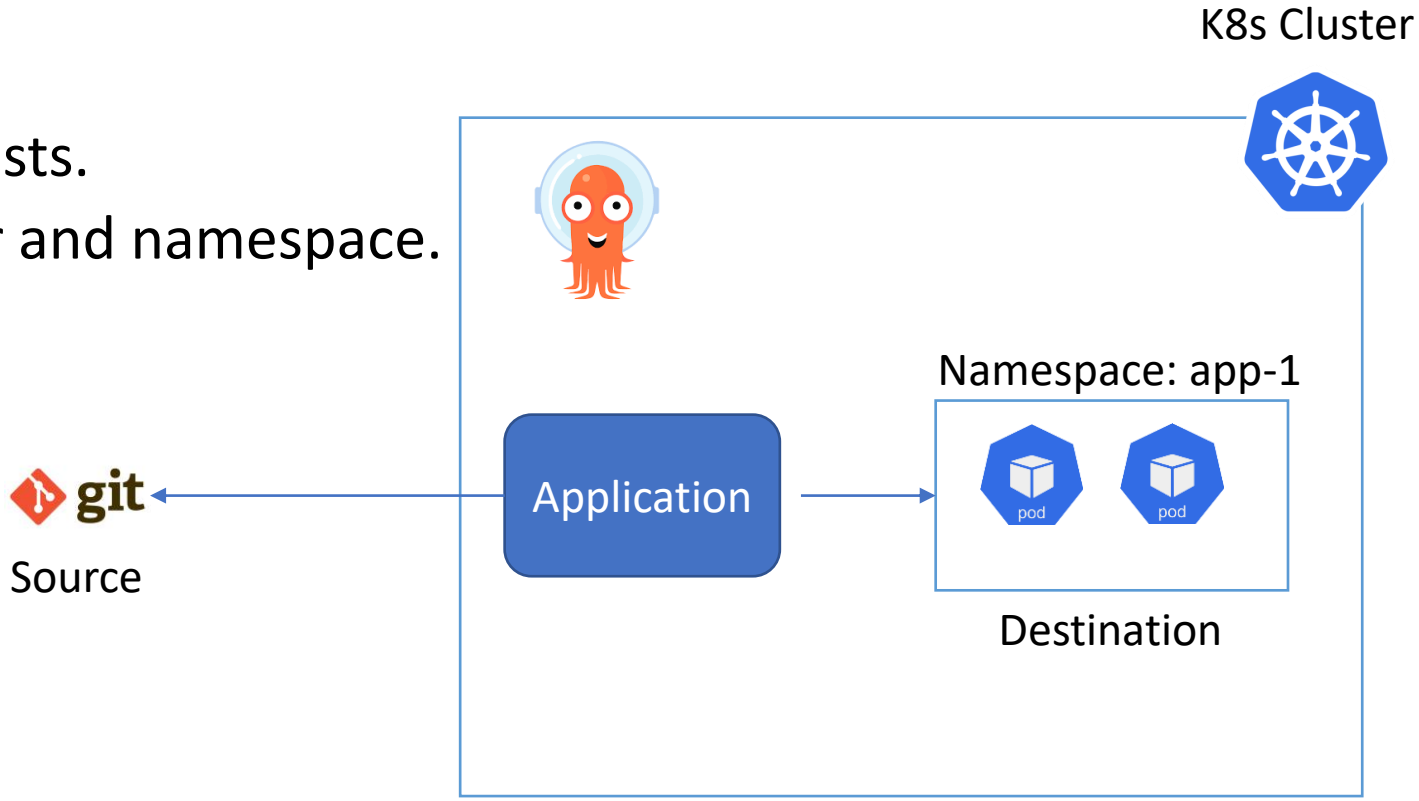
- Redis: used for caching.
- Dex: identity service to integrate with external identity providers.
- ApplicationSet Controller : It automates the generation of Argo CD Applications.



Core Concepts

Application

- Defines source and destination to deploy group of k8s resources.
  - Source : k8s manifests.
  - Destination: cluster and namespace.

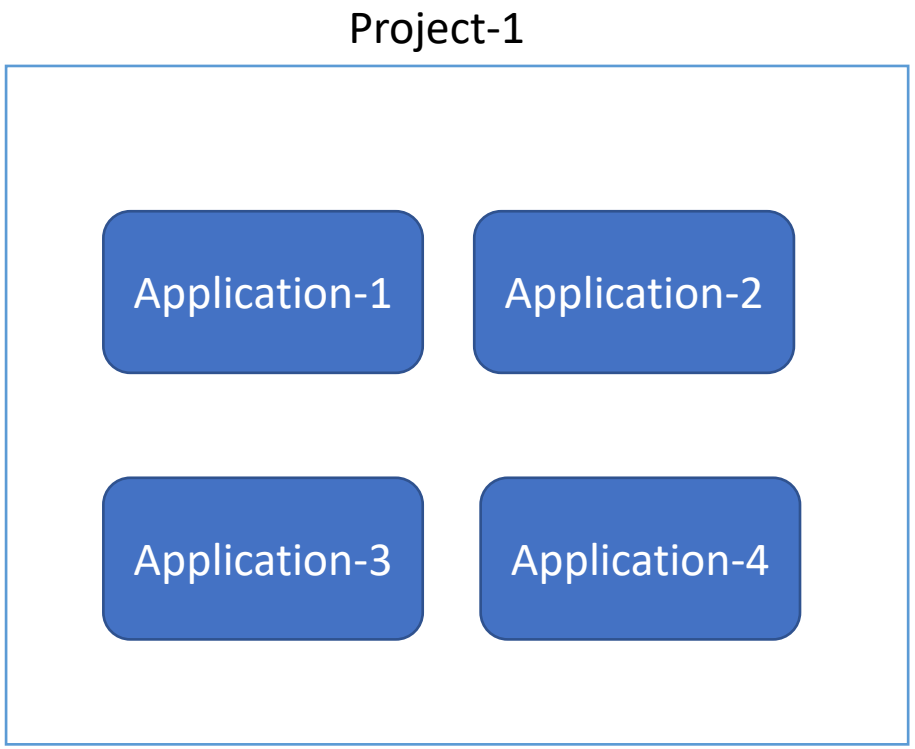


Application source - tools

- ArgoCD supports the below tools as source
  - Helm charts
  - Kustomize application
  - Directory of Yaml files.
  - Jsonnet

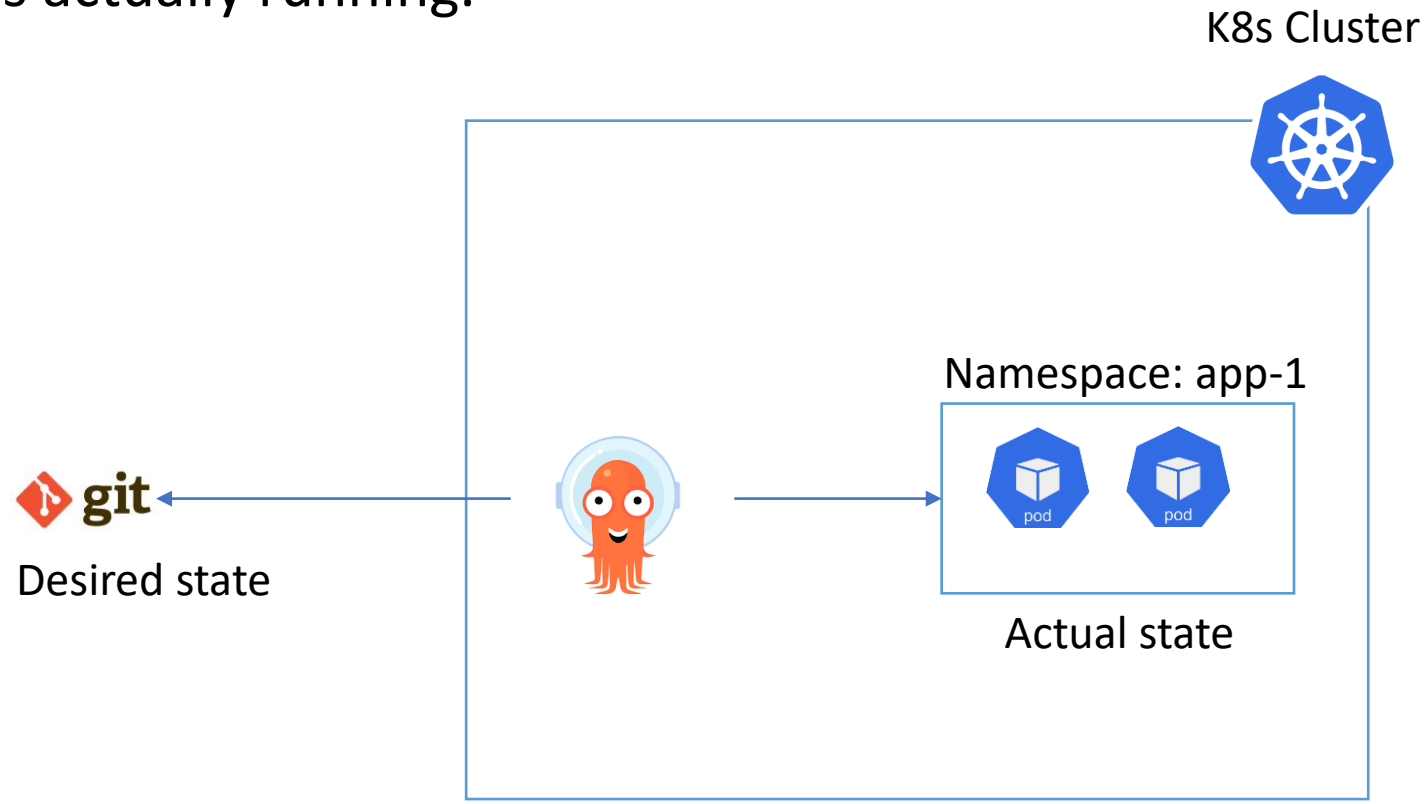
Project

- Projects provide a logical grouping of applications.
  - Useful when ArgoCD is used by multiple teams.
    - Allow only specific sources “trusted git repos”.
    - Allow apps to be deployed into specific clusters and namespaces.
    - Allow specific resources to be deployed “deployments, Statefulsets .. etc”.



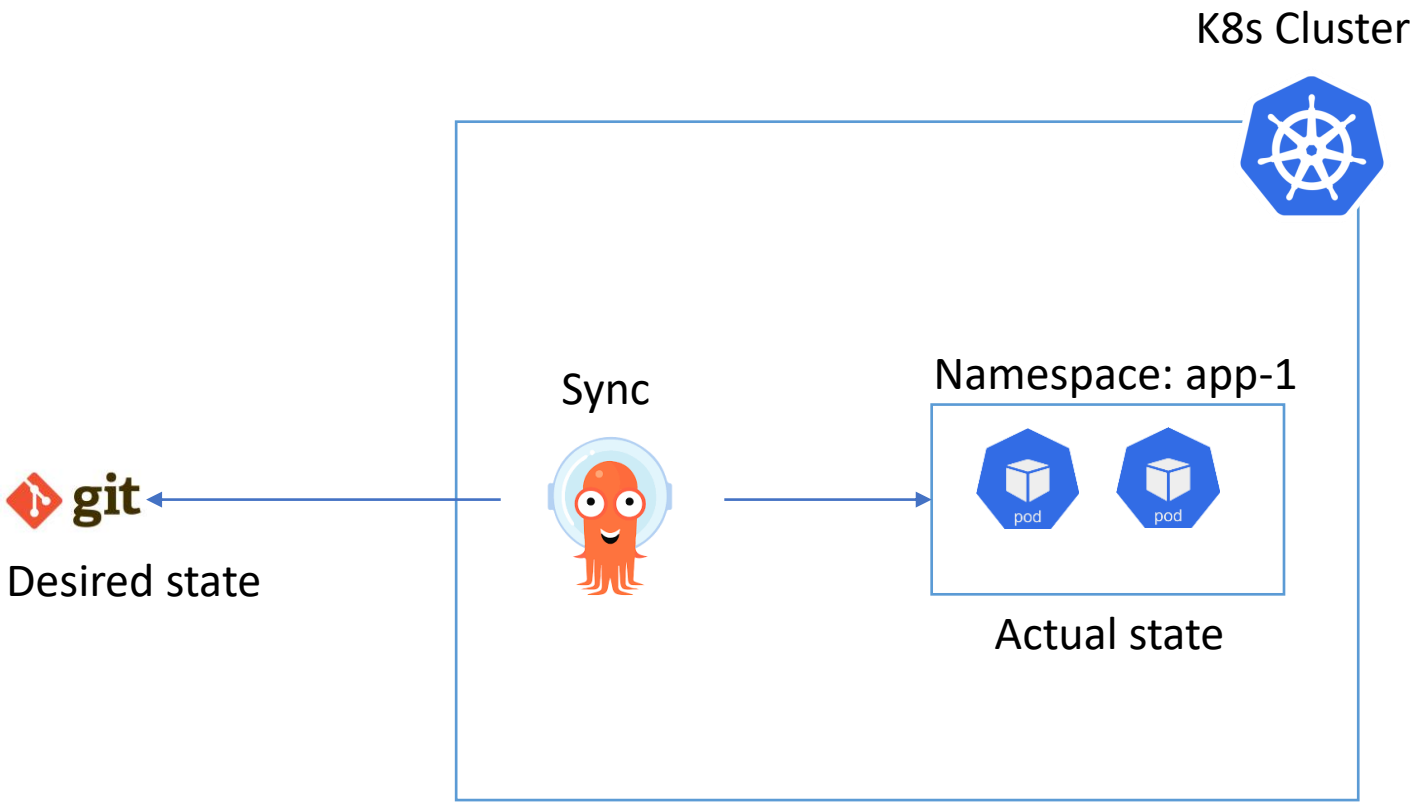
Desired state vs Actual state

- Desired state : described in git.
- Actual state: what is actually running.



Sync

- The process of making desired state = actual state.



Refresh (Compare)

- Compare the latest code in Git with the live state. Figure out what is different.
- ArgoCD automatically refreshes every 3 minutes.

Installation options

Prerequisite - k8s cluster

- You need a running k8s cluster.
  - Minikube.
  - Docker Desktop.
  - Kind.
  - Rancher Desktop.
  - Full cluster.

Installation options

- Non High availability setup
  - Suitable for evaluation or dev/testing environments.
- High availability setup.
  - Recommended for production.
  - You need at least 3 worker nodes.
- Light installation “Core”
  - Suitable if ArgoCD is used by administrators only. UI and API server is not installed for end users.
  - By default its installed as Non-HA.

Privileges options

- ArgoCD provides two options for in-cluster privileges.
  - **Cluster-admin privileges**: where ArgoCD has the cluster-admin access to deploy into the cluster that runs in.
  - **Namespace level privileges**: Use this manifest set if you do not need Argo CD to deploy applications in the same cluster that Argo CD runs in

Manifests Installation options

- Yaml Manifests :

https://github.com/argoproj/argo-cd/blob/master/manifests/install.yaml

- Helm chart:

https://github.com/argoproj/argo-helm/tree/master/charts/argo-cd

- Kustomize

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

namespace: argocd
resources:
- https://raw.githubusercontent.com/argoproj/argo-cd/v2.0.4/manifests/ha/install.yaml
```

Practice: Install Argo CD

- You need to have a Kubernetes cluster, If you don't already have one, you can install docker desktop or minikube or rancher desktop.
- Create namespace for Argo CD.
- Install Non highly available setup.
- Use this manifest: https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml

Practice: Get Initial Admin Password

- Get the initial admin password that is stored as a secret in Argo CD namespace.
- Convert the secret from base64 into plain text.

Accessing ArgoCD server (API + UI)

Exposing ArgoCD Server (API + UI)

- By default ArgoCD server is not exposed with external endpoint.

Expose by using:

- Service : LoadBalancer.
  - Change the argocd-server service type to LoadBalancer
- Ingress: Use your preferred ingress controller
  - Create an ingress resource that point into argocd-server service.
- Port-forward: simply you can use this to access locally on your machine
  - kubectl port-forward svc/argocd-server -n argocd 8080:443

Practice: Access Argo CD Server Web UI

- Use kubectl port forward to access Argo CD Web UI.
- Use the localhost port 8080 to forward the traffic into Argo CD Server service.

ArgoCD CLI

ArgoCD CLI

- You can interact with ArgoCD via
  - CLI
  - Web UI
  - Rest/gRPC API

- CLI is useful when you need to interact with ArgoCD in CI pipelines.

ArgoCD CLI - Commands

- You manage everything using CLI
  - Manage applications.
  - Manage Repos.
  - Manage clusters.
  - Admin tasks.
  - Manage projects.
  - And more.

ArgoCD CLI - Installation

- ArgoCD CLI can be installed into all platforms
  - **Linux** :

    1. brew install argocd
    or
    2. yay -Sy argocd-bin
    Or using curl
    3. curl -sSL -o /usr/local/bin/argocd https://github.com/argoproj/argo-cd/releases/latest/download/argocd-linux-amd64

  - **Mac** :

    1. brew install argocd

    or using curl

    2. VERSION=$(curl --silent "https://api.github.com/repos/argoproj/argo-cd/releases/latest" | grep '"tag_name"' | sed -E 's/.*"([^"]+)".*/\1/')

    curl -sSL -o /usr/local/bin/argocd https://github.com/argoproj/argo-cd/releases/download/$VERSION/argocd-darwin-amd64

  - **Windows** :

    $version = (Invoke-RestMethod https://api.github.com/repos/argoproj/argo-cd/releases/latest).tag_name

    $url = "https://github.com/argoproj/argo-cd/releases/download/" + $version + "/argocd-windows-amd64.exe"
    $output = "argocd.exe" Invoke-WebRequest -Uri $url -OutFile $output

ArgoCD CLI - Login

- You need to login to ArgoCD Server before using any command.

  argocd login <ARGOCD_SERVER>

- After successful login you can try using commands

  ex: argocd cluster list

Practice: Install Argo CD CLI

- Install Argo CD CLI on your machine.
- Follow official docs for your preferred platform (linux or mac or windows)
  https://argo-cd.readthedocs.io/en/stable/cli_installation/

- Expose Argo CD Server whether using port forward or ingress or load balancer service.
- Use CLI to login to Argo CD using admin user and password.
- Verify that you can get data from Argo CD by running a any command such as argocd cluster list.

Applications

ArgoCD Application

- Application is a Kubernetes resource object representing a deployed application instance in an environment.

- It is defined by two key pieces of information:

  - **Source**: reference to the desired state in Git (repository, revision, path)
  - **Destination**: reference to the target cluster and namespace.

ArgoCD Application

- Applications can be created using below options
  - Declaratively “Yaml”. (Recommended)
  - Web UI
  - CLI

Application
“declarative”

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
```

Argo application object “CRD”

Name of application

Destination cluster

Source of manifests

Application

“Web UI”

CREATE CANCEL

GENERAL

Application Name
app-1

Project
default

SYNC POLICY
Manual

SOURCE

Repository URL
https://github.com/mabusaa/argocd-example-apps.git

Revision
master

Path
guestbook

DESTINATION

Cluster URL
https://kubernetes.default.svc

Namespace
default

Application
"CLI"

argocd app create app-1 --repo https://github.com/mabusaa/argocd-example-apps.git --path guestbook --dest-server https://kubernetes.default.svc --dest-namespace default

Practice: Create Argo CD Application

- Define an Argo CD application declaratively using Yaml.
  - Source: Use this directory manifests as a source : https://github.com/mabusaa/argocd-example-apps/tree/master/guestbook
  - Destination: use local Kubernetes cluster.
- Apply the Argo CD Application using kubectl.
- View the app in Web and sync it, also verify resources are created in destination cluster.

Practice: Create Argo CD Application using Web UI

- Navigate to web UI on your browser and create an application with below info
  - Source: Use this directory manifests as a source : https://github.com/mabusaa/argocd-example-apps/tree/master/guestbook
  - Destination: use local Kubernetes cluster.
- Sync the app, and verify resources are created in destination cluster.

Practice: Create Argo CD Application using CLI

- Open your terminal and login to Argo CD using CLI.
- Create an application using CLI with below info:
  - Source: Use this directory manifests as a source : https://github.com/mabusaa/argocd-example-apps/tree/master/guestbook
  - Destination: use local Kubernetes cluster.
- Sync the app, and verify resources are created in destination cluster.

Tools Detection

Tools supported

- ArgoCD supports the below tools as source
  - Helm charts
  - Kustomize application
  - Directory of Yaml files.
  - Jsonnet

Tool - Explicitly Specifying Directory of Yaml

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
    directory:
      recurse: true
```

Directory of Yaml

Tool - Explicitly Specifying Helm

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: helm-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
    helm:
      releaseName: guestbook
```

Helm

Tool - Explicitly Specifying kustomize

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: guestbook-kustomize
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
    kustomize:
      version: v3.5.4
```

kustomize

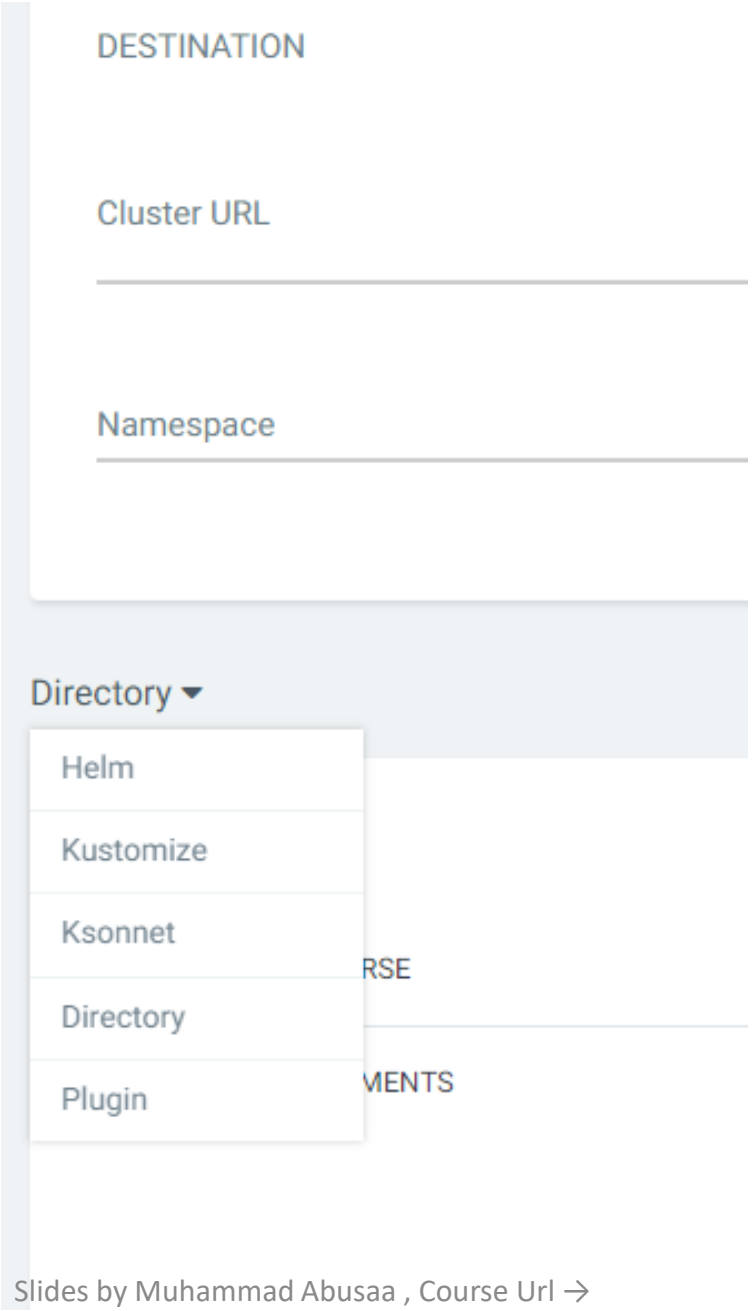
Tool – Not Set, Auto Detected

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
```

How ArgoCD Detects Tools

- If you don’t explicitly specify a tool, then its detected as follows:
  - Helm charts : if there is a file as Chart.yaml
  - Kustomize: if there's a kustomization.yaml, kustomization.yml, or Kustomization
  - Otherwise it is assumed to be a plain Yaml **directory** application.

Tool - Explicitly Specifying In Web UI



Helm Options

Helm Sources

- Helm Applications an be deployed from two sources
  - Git Repo.
  - Helm Repo.

Helm – From Git Repo

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: helm-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
```

From Helm Repo

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    chart: sealed-secret
    repoURL: "https://bitnami-labs.github.io/sealed-secrets"
    targetRevision: 1.16.1  # For Helm, this refers to the chart version.
```

Chart name

Helm repo Url

Chart version

Options

- ArgoCD provies the below for options
  - Release name.
  - Values files.
  - Parameters.
  - File parameters.
  - Values as block file.

Helm – Release name

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: helm-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
    helm:
      releaseName: # override release name (defaults to application name)
```

Release name

Helm – Values files

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: helm-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
    helm:
      valuesFiles: # can set multi values files, (defaults to values.yaml in source repo)
       - values-prod.yaml
```

Values files

Helm – Parameters

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: helm-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
    helm:
      parameters: # override any values in a values.yaml
       - name: “service.type”
         value: “LoadBalancer”
       - name: “image.tag”
         value: “v2”
```

parameters

Helm – File Parameters

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: helm-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
    helm:
      fileParameters: # set parameter values from a file
       - name: config
         value: files/config.json
```

File parameters

Helm – Values as block

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: helm-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
    helm:
        values: |
            ingress:
              enabled: true
              path: /
              hosts:
                - mydomain.example.com
```

Block file

Practice: Helm Options

- Define and create Argo CD Application declaratively with below info:
  - Source: Use this **helm chart** manifests as a source : https://github.com/mabusaa/argocd-example-apps/tree/master/helm-guestbook
  - Destination: use local Kubernetes cluster.
- Sync the app, and verify resources are created in destination cluster.
- After that, update the app definition and set the helm release name.
- Re-sync the application and prune the old resources.

Directory of files Options

Directory - Options

- ArgoCD provides the below as options
  - Recursive: include all files in sub-directories.
  - Jsonnet
    - External Vars : list of external variables for Jsonnet.
    - Top level Arguments.

Directory – Recurse

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path:  guestbook-with-sub-directories
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: master
    directory:
      recurse: true # include all sub-driectories
```

Recursive

Directory – Jsonnet External Variables

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path:  guestbook-with-sub-directories
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: master
    directory:
      jsonnet:
        extVars:
        - name: service
          value: “internal”
```

External variables

Directory – Jsonnet Top Level Arguments

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path:  guestbook-with-sub-directories
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: master
    directory:
      jsonnet:
        tlas:
        - name: service
          value: “internal”
          code: false
```

Top level arguments

Practice: Directory Options

- Define and create Argo CD Application declaratively with below info:
  - Source: Use this **Directory of Yaml** manifests as a source : https://github.com/mabusaa/argocd-example-apps/tree/master/guestbook-with-sub-directories
  - Destination: use local Kubernetes cluster.
- Sync the app, and verify resources are created in destination cluster.
- After that, update the app definition and **set the recurse to true**.
- Re-sync the application.

Kustomize Options

Options

- ArgoCD provides the below for options
  - Name prefix: appended to resources.
  - Name suffix: appended to resources.
  - Images : to override images.
  - Common labels: set labels on all resources.
  - Common annotations: set annotations on all resources.
  - Version: explicitly set kustomize version.

Kustomize – Name Prefix

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: kustomize-guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: kustomize-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
    kustomize:
      namePrefix: staging- # adds  a prefix to all resources names
```

prefix name

Kustomize – Name Suffix

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: kustomize-guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: kustomize-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
    kustomize:
      nameSuffix: -staging  # adds  a suffix to all resources names
```

suffix name

Kustomize – Override Images

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: kustomize-guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: kustomize-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
    kustomize:
      images:
       - gcr.io/heptio-images/ks-guestbook-demo:0.2
```

Override images

Kustomize – Common Labels

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: kustomize-guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: kustomize-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
    kustomize:
      commonLabels:
        app: demo
        appVersion: 1.0
```

labels

Kustomize – Common Annotations

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: kustomize-guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: kustomize-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
    kustomize:
      commonAnnotations:
        app: demo
        appVersion: 1.0
```

annonations

Kustomize – version

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: kustomize-guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: kustomize-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
    kustomize:
      version: v3.5.4
```

version

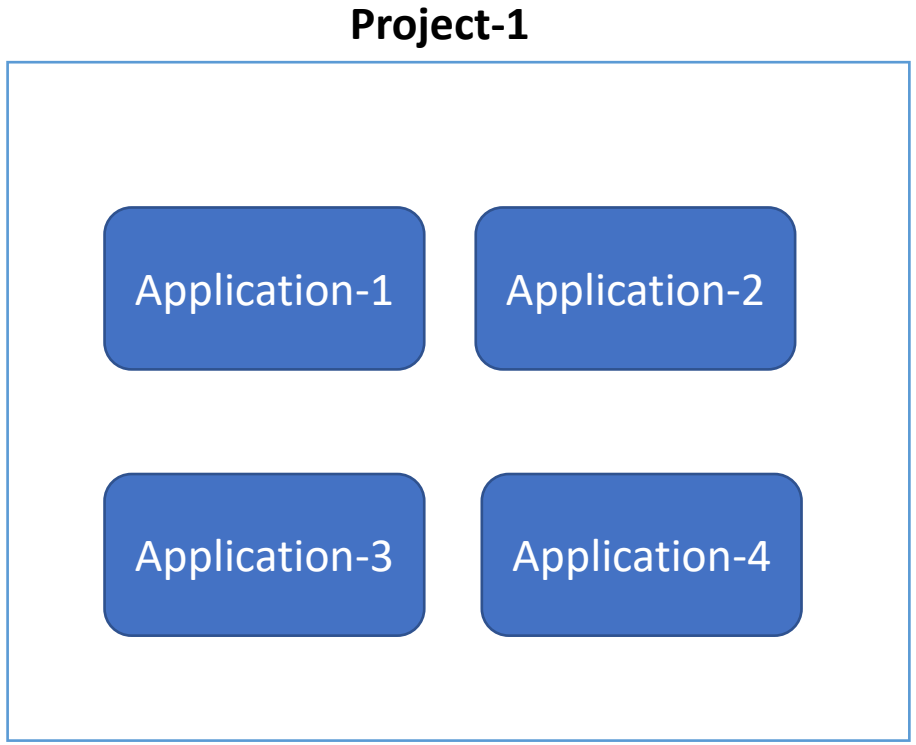
Practice: Kustomize Options

- Define and create Argo CD Application declaratively with below info:
  - Source: Use this **Kustomize** manifests as a source : https://github.com/mabusaa/argocd-example-apps/tree/master/kustomize-guestbook.
  - Destination: use local Kubernetes cluster.
- Apply the Argo CD application and explore the resources in Web UI.
- After that, update the app definition and **set name prefix and common labels**.
- Sync the application.

Why Projects

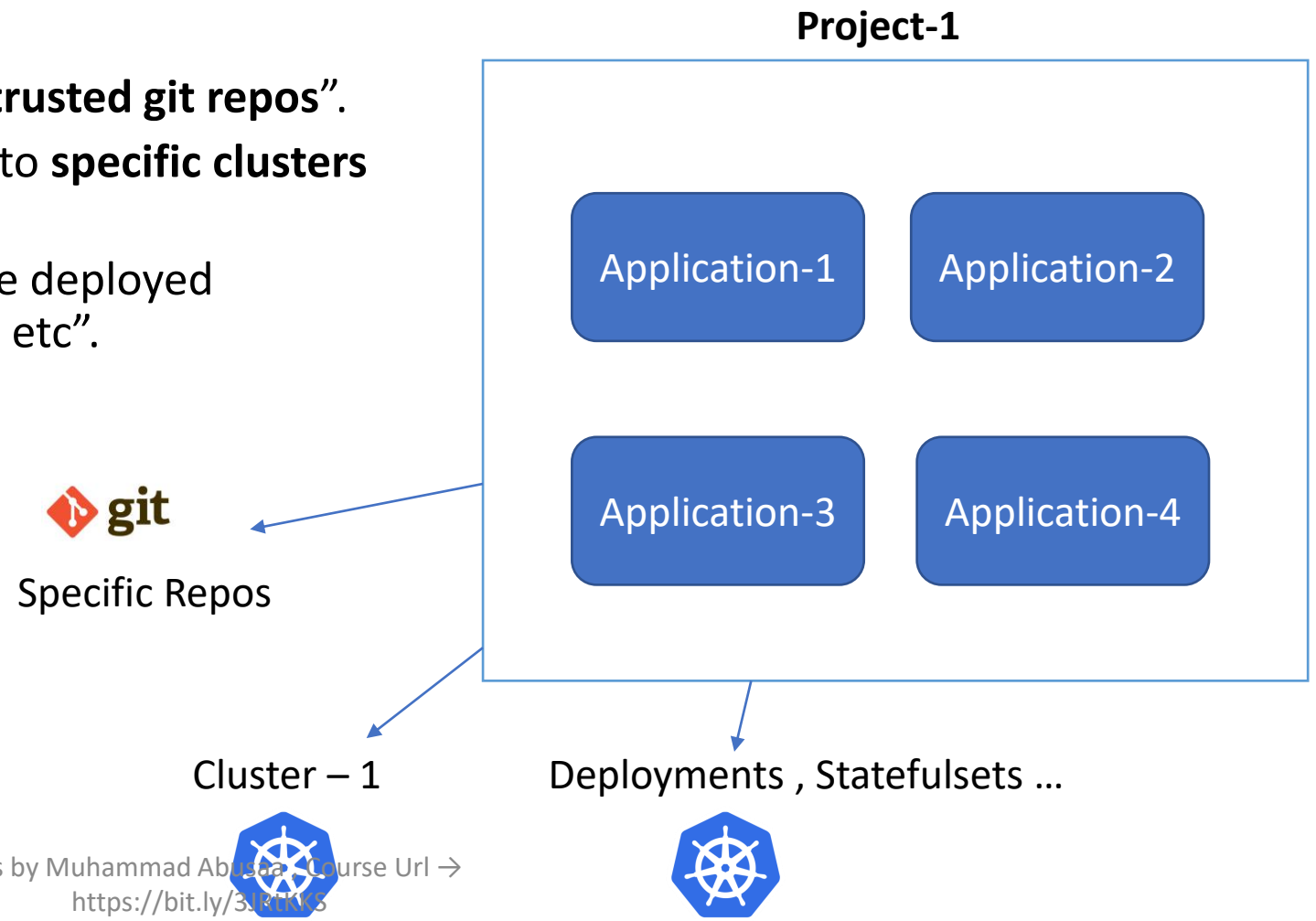
Application Grouping

- Projects provide a logical grouping of applications.



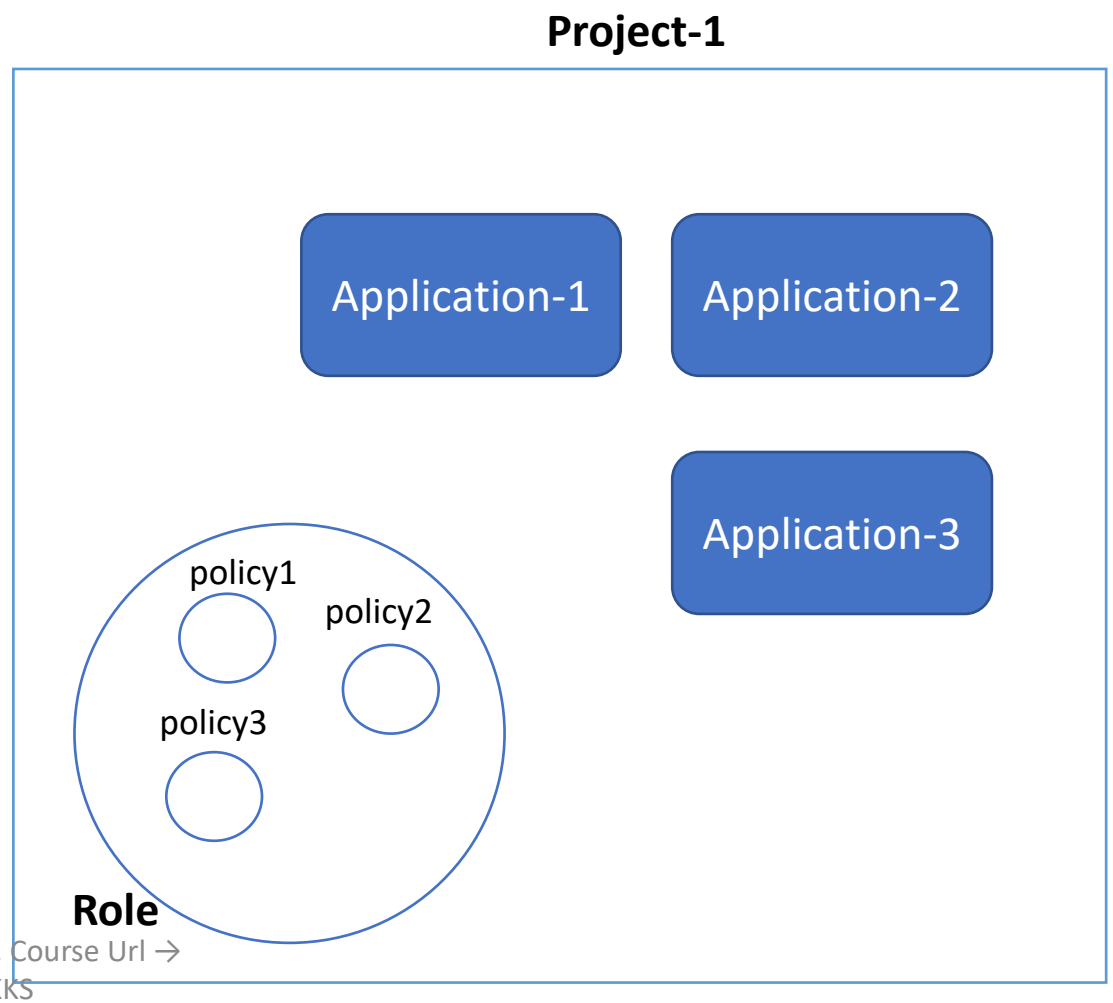
Access Restrictions

- Useful when ArgoCD is used by multiple teams.
  - Allow only specific sources “**trusted git repos**”.
  - Allow apps to be deployed into **specific clusters and namespaces**.
  - Allow **specific resources** to be deployed “Deployments, Statefulsets .. etc”.



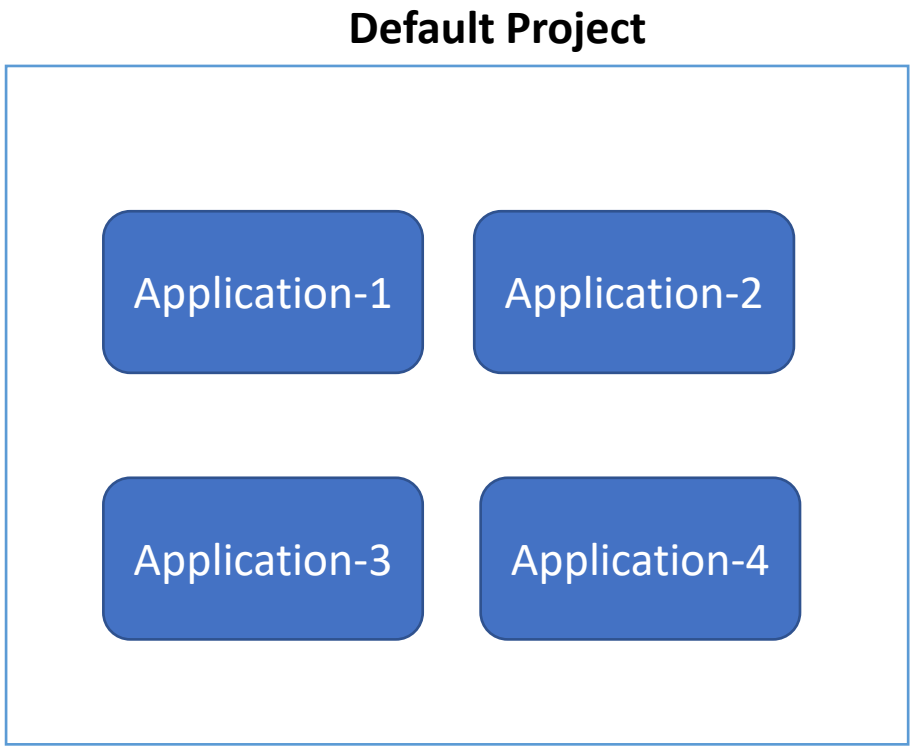
Project Roles Feature

- Enables you to create a role with set of policies “permissions” to grant access to a project's applications.
  - You can use it to grant CI system a specific access to project applications.
    - It must be associated with JWT.
  - You can use it to grant oidc groups a specific access to project applications.



Default Project

- ArgoCD creates a default project once you install it.



Creating Projects

Creating Projects Options

- Declaratively.
- CLI.
- Web UI.

Project - Yaml

```yaml
apiVersion: argoproj.io/v1alpha1
kind: AppProject
metadata:
  name: project-1
  namespace: argocd
spec:
  description: project description
  sourceRepos:
    - “*”
  destinations:
    - server: “*”
      namespace: “*“
  clusterResourceWhitelist:
    - group: “*”
      kind: “*“
 namespaceResourceWhitelist:
    - group: “*”
      kind: “*"
```

Specific destination

```yaml
apiVersion: argoproj.io/v1alpha1
kind: AppProject
metadata:
  name: project-1
  namespace: argocd
spec:
  description: project description
  sourceRepos:
    - "*"
  destinations:  # Only permit applications to deploy to the ns-1 namespace in the same cluster
    - server: https://kubernetes.default.svc
      namespace: "ns-1"
  clusterResourceWhitelist:
    - group: "*"
      kind: "*"
 namespaceResourceWhitelist:
    - group: "*"
      kind: "*"
```

Specific Source Repo

```yaml
apiVersion: argoproj.io/v1alpha1
kind: AppProject
metadata:
  name: project-1
  namespace: argocd
spec:
  description: project description
  sourceRepos: # Only permit this Git repos
    - “https://github.com/mabusaa/argocd-example-apps.git”
  destinations:
    - server: https://kubernetes.default.svc
      namespace: “ns-1“
  clusterResourceWhitelist:
    - group: “*”
      kind: “*“
 namespaceResourceWhitelist:
    - group: “*”
      kind: “*"
```

Allow specific Cluster-scoped resources

```yaml
apiVersion: argoproj.io/v1alpha1
kind: AppProject
metadata:
  name: project-1
  namespace: argocd
spec:
  description: project description
  sourceRepos:
    - "*"
  destinations:
    - server: https://kubernetes.default.svc
      namespace: "ns-1"
  clusterResourceWhitelist: # Deny all cluster-scoped resources from being created, except for Namespace
    - group: ""
      kind: "Namespace"
 namespaceResourceWhitelist:
    - group: "*"
      kind: "*"
```

Allow specific Namespace-scoped resources

```yaml
apiVersion: argoproj.io/v1alpha1
kind: AppProject
metadata:
  name: project-1
  namespace: argocd
spec:
  description: project description
  sourceRepos:
    - "*"
  destinations:
    - server: https://kubernetes.default.svc
      namespace: "ns-1"
  clusterResourceWhitelist:
    - group: ""
      kind: "Namespace"
  namespaceResourceWhitelist: # Deny all namespaced-scoped resources from being created, except for Deployment
    - group: "apps"
      kind: "Deployment"
```

Blacklist specific Namespace-scoped resources

```yaml
apiVersion: argoproj.io/v1alpha1
kind: AppProject
metadata:
  name: project-1
  namespace: argocd
spec:
  description: project description
  sourceRepos:
    - "*"
  destinations:
    - server: https://kubernetes.default.svc
      namespace: "ns-1"
  clusterResourceWhitelist:
    - group: ""
      kind: "Namespace"
  namespaceResourceBlacklist:  # Allow all namespaced-scoped resources to be created, except for NetworkPolicy
    - group: ""
      kind: "NetworkPolicy"
```

Set project in application

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: project-1
  source:
    path: helm-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
```

Practice: Creating Projects

- Define and create Argo CD Project declaratively with below info:
  - Allow all sources (repos)
  - Allow all destinations
  - Allow all cluster and namespace scoped resources
- Then, define and create and Argo CD Application that is related to the new project.

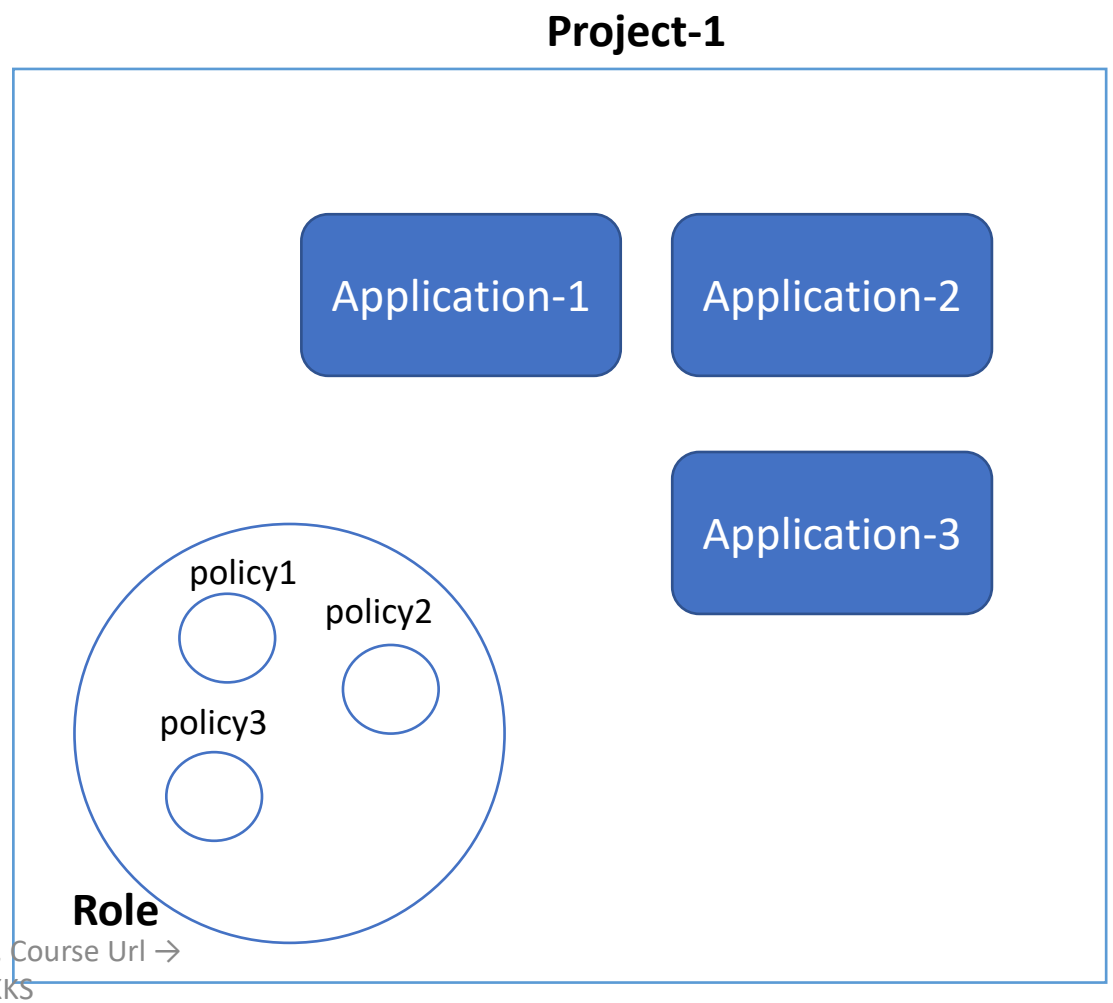
Practice: Creating Project and Allowing Specific Destinations

- Define and create Argo CD Project declaratively with below info:
  - Allow all sources (repos)
  - Allow **local cluster** and namespace **ns-1** only as destinations
  - Allow all cluster and namespace scoped resources
- Then, define and create and Argo CD Application that is using **test-1** namespace as destination, and verify that Argo CD deny this application.
- Then update the application with **ns-1** namespace, and sync the application.

Project roles

Project Roles Feature

- Enables you to create a role with set of policies “permissions” to grant access to a project's applications.
  - You can use it to grant CI system a specific access to project applications.
    - It must be associated with JWT.
  - You can use it to grant oidc groups a specific access to project applications.



Project Role

```yaml
apiVersion: argoproj.io/v1alpha1
kind: AppProject
metadata:
  name: project-1
  namespace: argocd
spec:
  description: project description
  sourceRepos:
    - "*"
  destinations:
    - server: "*"
      namespace: "*"
  clusterResourceWhitelist:
    - group: "*"
      kind: "*"
 roles:
  - name: ci-role
    description: Sync privileges for demo-project
    policies:
    - p, proj:demo-project:ci-role, applications, sync, demo-project/*, allow
```

Projectname:rolename

Action: sync,get,create,delete,update,override

Application name: * means all

Creating a token

- Project roles is not useful without generating a JWT.
- Generated tokens are not stored in ArgoCD.

- To create a token using CLI
  argocd proj role create-token PROJECT ROLE-NAME

  Expected output:

```
Create token succeeded for proj:demo-project:ci-role.
  ID: 867b66fe-969f-41d7-aba0-d9fbfe3548c5
  Issued At: 2022-05-10T23:28:09+03:00
  Expires At: Never
  Token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJhcmdvY2QiLCJzdWIiOiJwcm9qOmRlbW8tcHJvamVjdDpjaS1yb2xlIiwibmJmIjoxNjUyMjE0NDg5LCJpYXQiOjE
2NTIyMTQ0ODksImp0aSI6Ijg2N2I2NmZlLTk2OWYtNDFkNy1hYmEwLWQ5ZmJmZTM1NDhjNSJ9.a5ErBoVNxhj3QZE-_PgsUZK0Ud-ZcsWSa-k8QZj87qw
```

Using the token in CLI

- A user can leverage tokens in the cli by either passing them in using the --auth-token flag or setting the ARGOCD_AUTH_TOKEN environment variable.

Ex: argocd cluster list --auth-token token-value

Practice: Creating a Project With Role

- Define and create Argo CD Project declaratively with below specs:
  - Allow all sources (repos)
  - Allow all destinations
  - Allow all cluster and namespace scoped resources.
  - Define a **role** with has **sync permission** to all applications in the same project.
- Then, create a token related to this role.
- Then, try to delete an application using this token. Argo CD should deny this action.

Private Git Repos

Private Git Repos

- Public repos can be used directly in application.
- Private repos needs to be registered in ArgoCD with proper authentication before using it in applications.
- ArgoCD support connecting to private repos using below ways:
  - HTTPs: using username and password or access token.
  - SSH: using ssh private key.
  - GitHub / GitHub Enterprise : GitHub App credentials.
- Private repos credentials are stored in normal k8s secrets.
- You can register repos using declarative approach, cli and web UI.

Git Repo using HTTPs

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: private-repo
  namespace: argocd
  labels:
    argocd.argoproj.io/secret-type: repository
stringData:
  type: git
  url: https://github.com/argoproj/private-repo
  password: my-password
  username: my-username
```

Git Repo using HTTPs - insecure

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: private-repo
  namespace: argocd
  labels:
    argocd.argoproj.io/secret-type: repository
stringData:
  type: git
  url: https://github.com/argoproj/private-repo
  password: my-password
  username: my-username
  insecure: true
```

Git Repo using HTTPs and Tls

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: private-repo
  namespace: argocd
  labels:
    argocd.argoproj.io/secret-type: repository
stringData:
  type: git
  url: https://github.com/argoproj/private-repo
  password: my-password
  username: my-username
  tlsClientCertData: ....
  tlsClientCertKey: ....
```

Git Repo using SSH

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: private-repo
  namespace: argocd
  labels:
    argocd.argoproj.io/secret-type: repository
stringData:
  type: git
  url: https://github.com/argoproj/private-repo
 sshPrivateKey: |
    -----BEGIN OPENSSH PRIVATE KEY-----
    ...
    -----END OPENSSH PRIVATE KEY-----
```

Git Repo using GitHub App

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: private-repo
  namespace: argocd
  labels:
    argocd.argoproj.io/secret-type: repository
stringData:
  type: git
  url: https://github.com/argoproj/private-repo
 githubAppID: 1
  githubAppInstallationID: 2
  githubAppPrivateKey: |
    -----BEGIN OPENSSH PRIVATE KEY-----
    ...
    -----END OPENSSH PRIVATE KEY-----
```

Practice: Adding Private Git Repo using Https

- Define and add a private git repo to Argo CD.
- Verify its connected successfully in repositories page in Web UI
- Then, create an Argo CD application that use the private repo as source of manifests.

Practice: Adding Private Git Repo using SSH

- Define and add a private git repo to Argo CD.
- Verify its connected successfully in repositories page in Web UI
- Then, create an Argo CD application that use the private repo as source of manifests.

Helm Repos

Helm Repos

- Public standard Helm repos can be used directly in application.
- Non standard Helm repositories have to be registered explicitly.
- Private Helm repos needs to be registered in ArgoCD with proper authentication before using it in applications.
- ArgoCD support connecting to private Helm repos using username/password and tls cert/key.
- Registering Helm repos in ArgoCD can be done declaratively , CLI and Web UI.

Private Helm Repo - declaratively

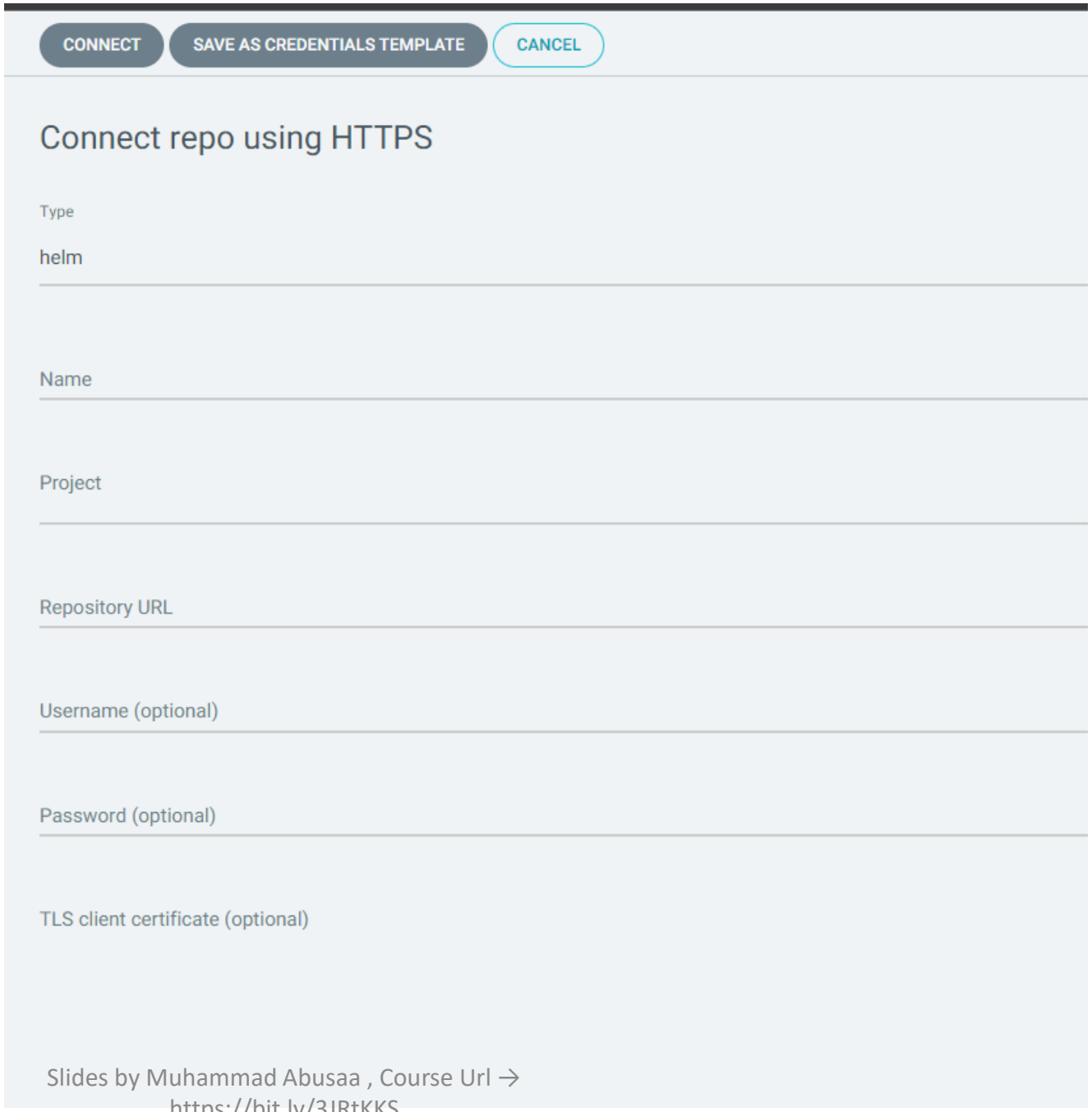
```yaml
apiVersion: v1
kind: Secret
metadata:
  name: private-repo
  namespace: argocd
  labels:
    argocd.argoproj.io/secret-type: repository
stringData:
  type: helm
  name: argo
  url: https://argoproj.github.io/argo-helm
  password: my-password
  username: my-username
  tlsClientCertData: ....
  tlsClientCertKey: ....
```

Private Helm Repo - CLI

- # Add a private Helm repository named 'stable' via HTTPS

argocd repo add https://charts.helm.sh/stable --type helm --name stable --username test --password test

Private Helm Repo – Web UI



Credential Templates

Credential Templates

- Used If you want to use the same credentials for multiple repositories in your organization without having to repeat credential configuration.
- Defined as same as repositories credentials information, with different label value "*argocd.argoproj.io/secret-type: repo-creds*".
- In order for ArgoCD to use a credential template for any given repository, the following conditions must be met:
  - The URL configured for a credential template (e.g. https://github.com/mabusaa) must match as prefix for the repository URL (e.g. https://github.com/mabusaa/argocd-example-apps).
  - The repository must either not be configured at all, or if configured, must not contain any credential information.
- Registering credentials in ArgoCD can be done declaratively , CLI and Web UI.

Credential Templates - declaratively

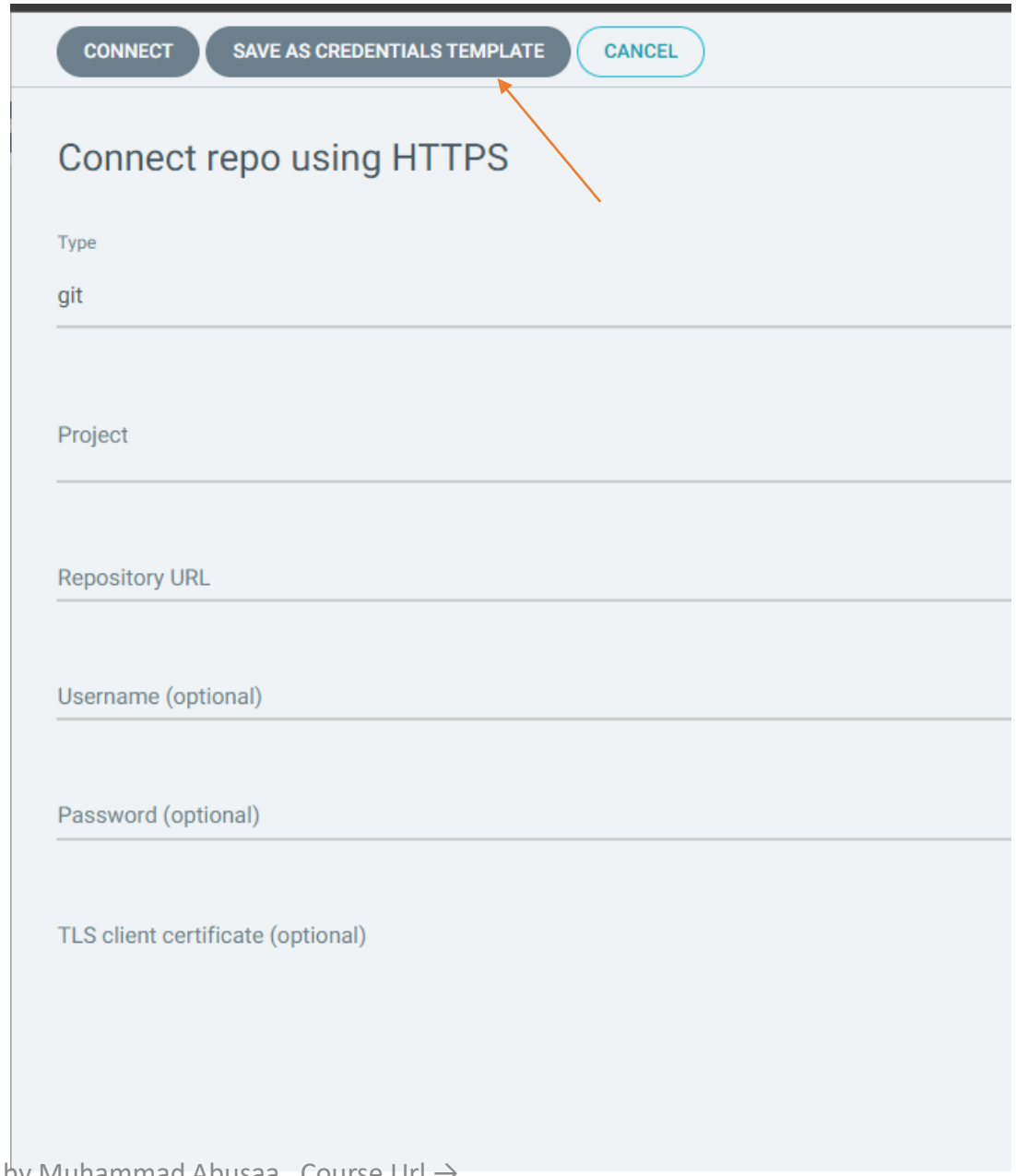
```yaml
apiVersion: v1
kind: Secret
metadata:
  name: private-repo-creds
  namespace: argocd
  labels:
    argocd.argoproj.io/secret-type: repo-creds
stringData:
  type: git
  url: https://github.com/mabusaa
  password: my-password
  username: my-username
```

Credential Templates - CLI

- # Add credentials with user/pass authentication to use for all repositories under https://git.example.com/repos

argocd repocreds add https://git.example.com/repos/ --username git --password secret

Credential Templates – Web UI



Practice: Credential Templates

- Define and add a credential template to Argo CD using Https or SSH.
- Then, verify that its working fine by creating an Argo CD application that uses a private repo as source of manifests.

Automated Sync

Automated Sync

- By default, ArgoCD polls Git repositories every 3 minutes to detect changes to the manifests.
- Argo CD can automatically sync apps when it detects differences between the desired manifests in Git, and the live state in the cluster.
  - No need to do manual sync anymore.
  - CI/CD pipelines no longer need direct access.
- Notes:
  - An automated sync will only be performed if the application is OutOfSync.
  - Automatic sync will not reattempt a sync if the previous sync attempt against the same commit-SHA and parameters had failed.
  - Rollback cannot be performed against an application with automated sync enabled.

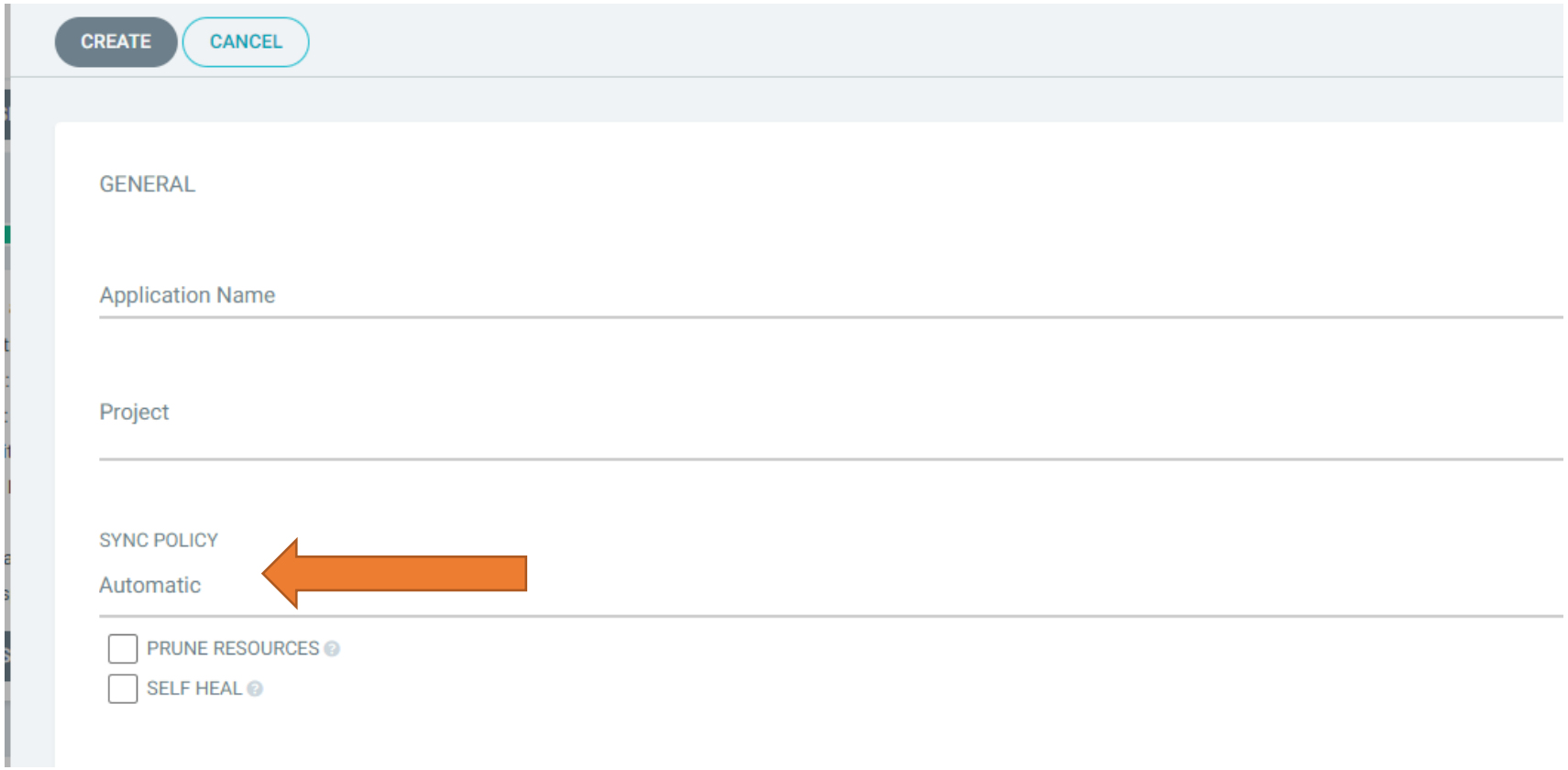
Auto Sync – Declaratively

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: kustomize-guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: kustomize-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
  syncPolicy:
    automated: {}
```

Auto Sync – CLI

argocd app create nginx-ingress --repo https://charts.helm.sh/stable --helm-chart nginx-ingress --revision 1.24.3 --dest-namespace default --dest-server https://kubernetes.default.svc **--sync-policy automated**

Auto Sync – Web UI



Practice: Automated Sync

- Define an Argo CD application and enable the automated sync.
  - Manifests as source : https://github.com/mabusaa/argocd-example-apps/tree/master/guestbook-with-sub-directories.
- Apply the application using kubectl and verify that its synced directly.
- Then, verify auto syncing by increasing the replicas in deployment manifests and refresh the application.

Automated Pruning

Automated Pruning

- Default – no prune: when automated sync is enabled, by default for safety automated sync will not delete resources when Argo CD detects the resource is no longer defined in Git.

- Pruning can be enabled to delete resources automatically as part of the automated sync.

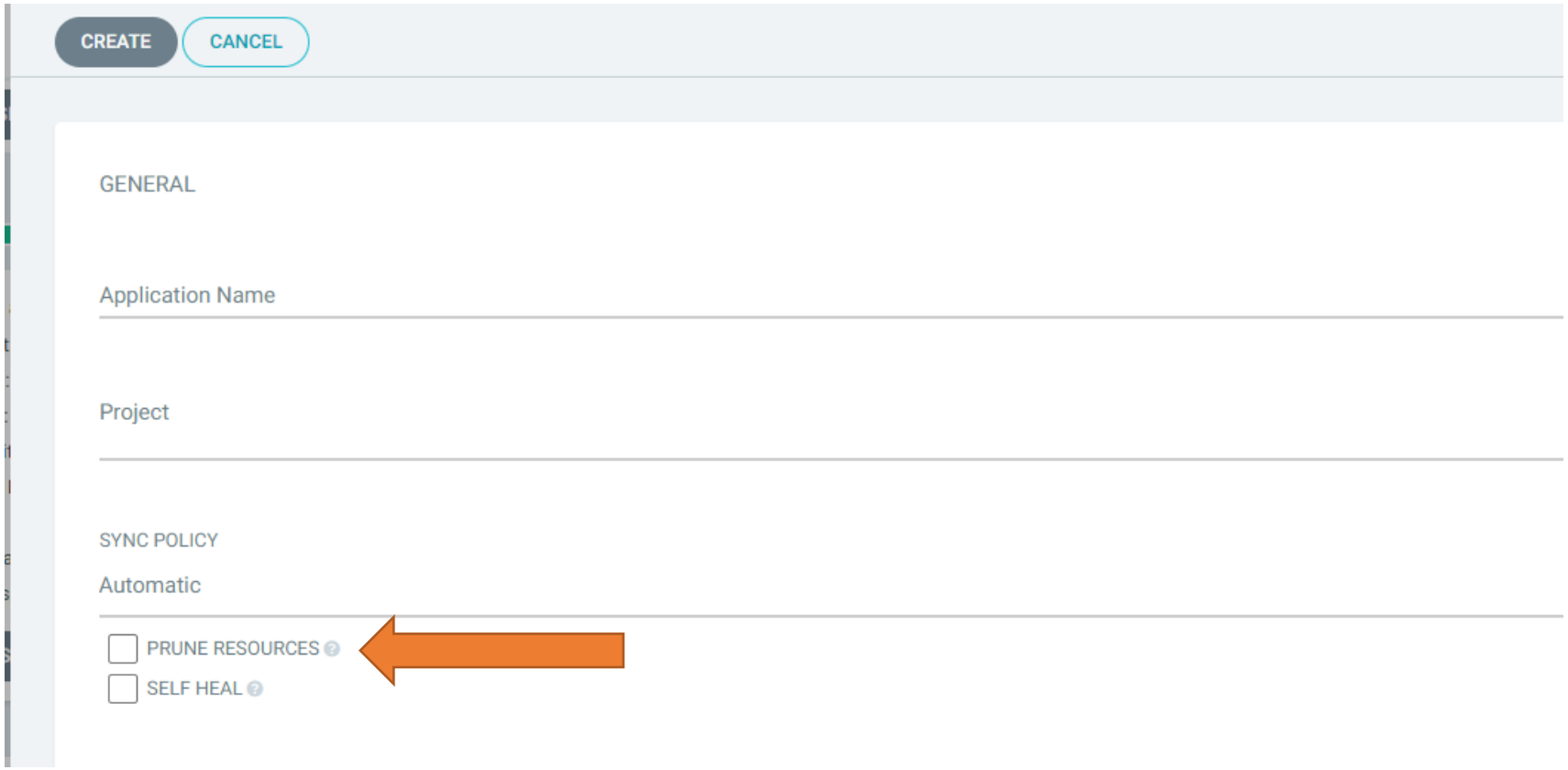
Auto Prune – Declaratively

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: kustomize-guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: kustomize-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
  syncPolicy:
    automated:
      prune: true
```

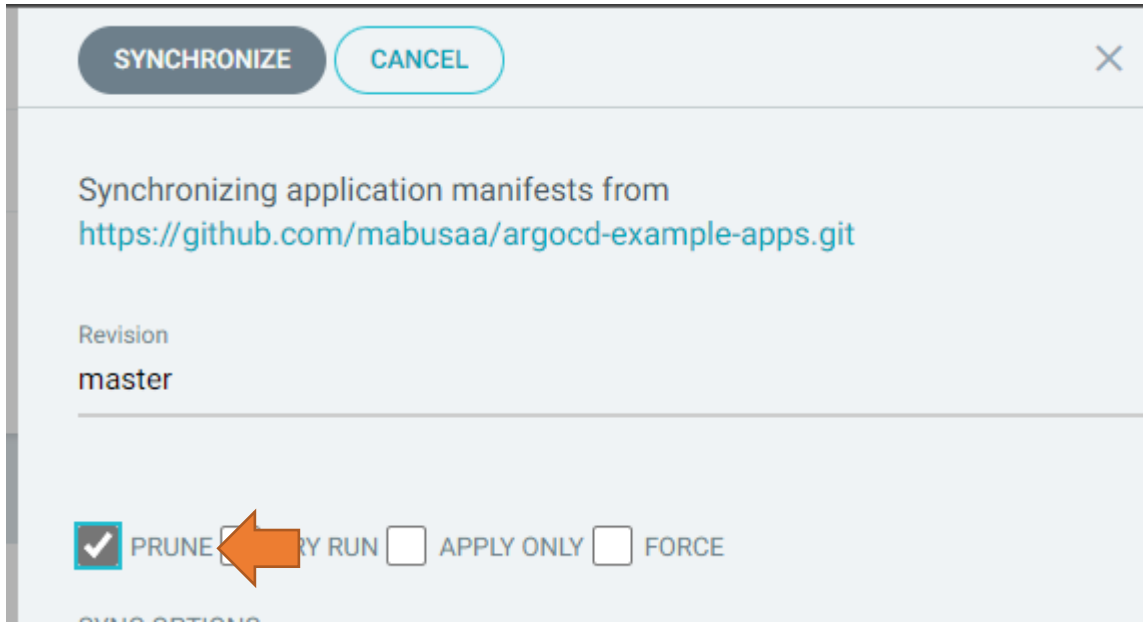
Auto Prune – CLI

argocd app create nginx-ingress --repo https://charts.helm.sh/stable --helm-chart nginx-ingress --revision 1.24.3 --dest-namespace default --dest-server https://kubernetes.default.svc **--auto-prune**

Auto Prune – Web UI



Prune manually – manual sync



Practice: Automated Pruning

- Define an Argo CD application and enable the automated sync only.
  - Manifests as source : https://github.com/mabusaa/argocd-example-apps/tree/master/guestbook-with-sub-directories.
- Apply the application using kubectl and verify that its synced directly.
- Delete the service file from git repo and notice that its not deleted from destination cluster.
- Then, enable the auto pruning in application definition and verify the service is deleted from destination cluster.

Automated Self Healing

Automated Self Healing

- By default, changes that are made to the live cluster will not trigger automated sync.
- ArgoCD has a feature to enable self healing when the live cluster state deviates from Git state.

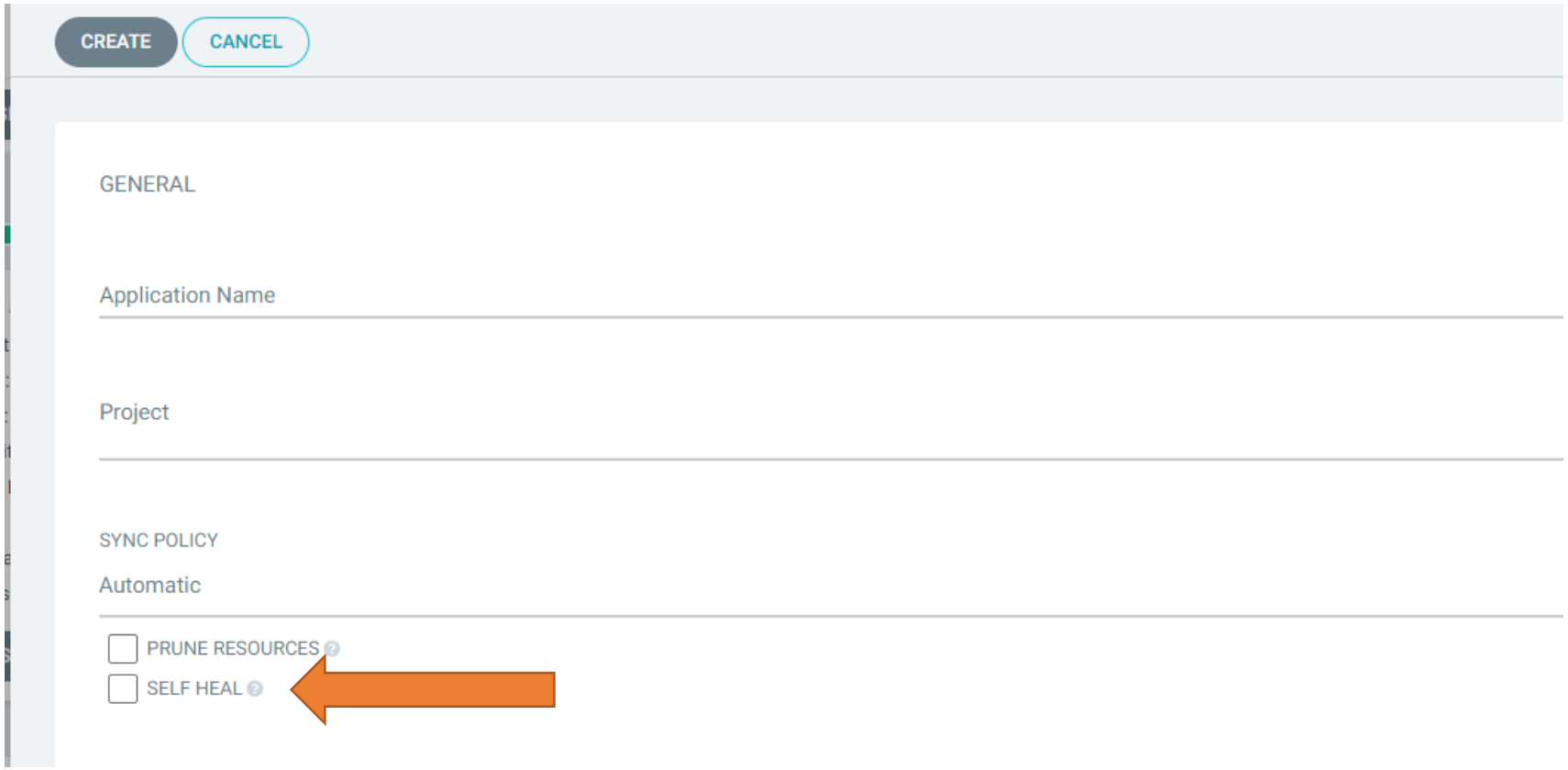
Auto Self Heal – Declaratively

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: kustomize-guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: kustomize-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
  syncPolicy:
    automated:
      selfHeal: true
```

Auto Self Heal– CLI

argocd app create nginx-ingress --repo https://charts.helm.sh/stable --helm-chart nginx-ingress --revision 1.24.3 --dest-namespace default --dest-server https://kubernetes.default.svc **--self-heal**

Auto Self Heal – Web UI



Practice: Automated Self Healing

- Define an Argo CD application and enable the automated sync only.
  - Manifests as source : https://github.com/mabusaa/argocd-example-apps/tree/master/guestbook-with-sub-directories.
- Apply the application using kubectl and verify that its synced directly.
- scale the deployment replicas into 3 using **kubectl**, and note how Argo CD will NOT do auto correction (healing) for the actual state.
- Then, enable the automated self healing in application definition.
- Try to scale the deployment to 10 replicas, and notice how Argo CD will do the auto-healing automatically.

Sync Options

Sync options

- Users can customize how resources are synced between target cluster and desired state.
- Most of the options available at application level.

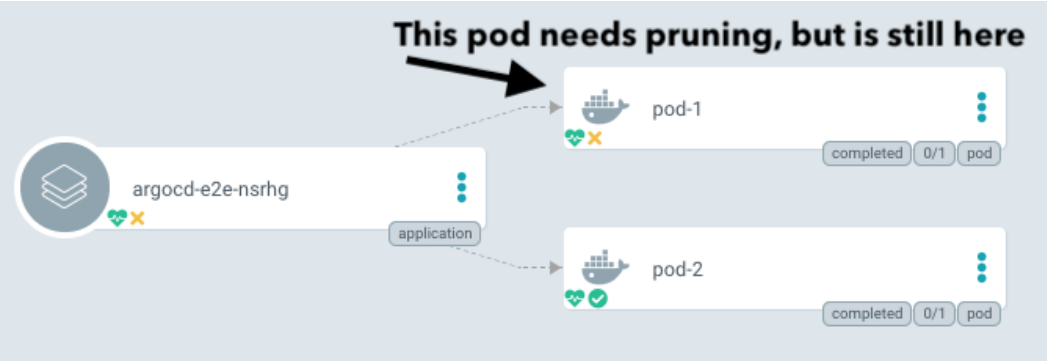
```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: example
spec:
 ....
  syncPolicy:
    syncOptions:
```

- Some of the options available using resources annotations.

```
metadata:
   annotations:
      argocd.argoproj.io……..
```

No Prune

- ArgoCD can prevent an object from being pruned.



- The app will be in out of sync but still does not prune the resource.
- In the resource itself, can be used as annotation as below:

```
metadata:
  annotations:
    argocd.argoproj.io/sync-options: Prune=false
```

No Prune – Specific resource

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    argocd.argoproj.io/sync-options: Prune=false
  name: guestbook-ui
spec:
  replicas: 2
  revisionHistoryLimit: 3
.....
```

ArgoCD will not prune the resource even if deleted in Git

Disable Kubectl Validation

- Some resources need to be applied without validating the resources “kubectl apply --validate=false”.
- You can achieve this in ArgoCD by at application level or resource level.
  - Application level:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: example
spec:
 ....
  syncPolicy:
    syncOptions:
      - Validate=false
```

  - Resource level using annotation :

```
metadata:
  annotations:
    argocd.argoproj.io/sync-options: Validate=false
```

Disable Kubectl Validation – Application level

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: kustomize-guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: kustomize-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
  syncPolicy:
   syncOptions:
    - Validate=false
```

 # ArgoCD will not validate resources when applying

Disable Kubectl Validation – Specific resource

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    argocd.argoproj.io/sync-options: Validate=false
  name: guestbook-ui
spec:
  replicas: 2
  revisionHistoryLimit: 3
.....
```

ArgoCD will not validate when applying the resource

Selective Sync

- When syncing using auto sync ArgoCD applies every object in the application.
- Selective sync option will sync only out-of-sync resources. You need when you have thousands of resources in which sync take a long time and puts pressure on Api server.
- Can be applied at application level only:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: example
spec:
 ....
  syncPolicy:
    syncOptions:
      - ApplyOutOfSyncOnly=true
```

Selective Sync – Application level

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: kustomize-guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: kustomize-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
  syncPolicy:
   syncOptions:
    - ApplyOutOfSyncOnly=true
```

ArgoCD sync resources with status out of sync only

Prune Last

- ArgoCD can control the sequence of creation/pruning resources, aka waves.
- You can prune some resources to happen as final using “Prune Last”.
- You can achieve this in ArgoCD by at application level or resource level.
  - Application level:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: example
spec:
 ....
  syncPolicy:
    syncOptions:
      - PruneLast=true
```

  - Resource level using annotation :

```
metadata:
  annotations:
    argocd.argoproj.io/sync-options: PruneLast=true
```

Prune Last – Application level

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: kustomize-guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: kustomize-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
  syncPolicy:
   syncOptions:
    - PruneLast=true
```

 # ArgoCD will prune this application last if it were deployed as part of multiple applications.

Prune Last – Specific resource

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    argocd.argoproj.io/sync-options: PruneLast=true
  name: guestbook-ui
spec:
  replicas: 2
  revisionHistoryLimit: 3
.....
```

ArgoCD will prune this resource as last one.

Replace Resources

- By default ArgoCD use “kubectl apply” to deploy the resources changes.
- In some cases you need to “Replace/Recreate” the resources, ArgoCD can do this by using replace=true.
- You can achieve this in ArgoCD by at application level or resource level.
  - Application level:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: example
spec:
 ....
  syncPolicy:
    syncOptions:
      - Replace=true
```

  - Resource level using annotation :

```
metadata:
  annotations:
    argocd.argoproj.io/sync-options: Replace=true
```

Replace – Application level

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: kustomize-guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: kustomize-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
  syncPolicy:
   syncOptions:
    - Replace=true
```

ArgoCD will replace all resources during the sync operation

Replace – Specific resource

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    argocd.argoproj.io/sync-options: Replace=true
  name: guestbook-ui
spec:
  replicas: 2
  revisionHistoryLimit: 3
.....
```

ArgoCD will replace the resource during the sync operation

Fail on Shared Resource

- By default ArgoCD will apply the resources even if it was available in multiple applications.
- You can configure the sync to fail if any resource is found in other applications by using FailOnSharedResource=true.
- Can be applied at application level only:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: example
spec:
 ....
  syncPolicy:
    syncOptions:
      - FailOnSharedResource=true
```

Fail on Shared Resource – Application level

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: kustomize-guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: kustomize-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
  syncPolicy:
   syncOptions:
    - FailOnSharedResource=true
```

 # ArgoCD sync will faile if any resource in this application is found in other applications.

Practice: Manifests No Pruning

- Define an Argo CD application and enable the automated sync only.
  - Manifests as source : https://github.com/mabusaa/argocd-example-apps/tree/master/sync-options/no-prune.
- Include argocd.argoproj.io/sync-options: Prune=false annotation in all manifests.
- Apply the application using kubectl and verify that its synced directly.
- Try to delete one resource from git repo, and verify its not deleted in the target cluster

Practice: Selective Sync

- Define an Argo CD application and enable the automated sync only.
  - Manifests as source : https://github.com/mabusaa/argocd-example-apps/tree/master/sync-options/selective-sync .
- Argo CD Application should only sync the changed manifests.
- Apply the application using kubectl and verify that its synced directly.
- Update deployment replicas and verify the sync result.

Practice: Fail On Shared Resources

- Define **two** Argo CD applications and enable the automated sync only.
  - Manifests as source : https://github.com/mabusaa/argocd-example-apps/tree/master/guestbook .
- Both applications should use the same manifests and same destination cluster and namespace.
- Notice the warnings in web UI.
- Then, On any of the applications enable fail on shared resources option and re-create the application, Argo CD will not be able to sync the application.

Practice: Replace Resources

- Define an Argo CD application and enable the automated sync only.
  - Manifests as source : https://github.com/mabusaa/argocd-example-apps/tree/master/sync-options/replace .
- Include argocd.argoproj.io/sync-options: Replace=true annotation in one of the manifests.
- Apply the application using kubectl and verify that its synced directly.
- Verify how Argo CD will replace the resource on each sync.
- Finally, Apply the replace option at the application and verify that all resources are re-created on each sync operation.

Tracking Strategies

Tracking Strategies

- ArgoCD provides several options to track manifests (sources) whether its in Git repos or Helm repos.

- Git repos tracking :
  - Commit SHA (good for production).
  - Tags (good for production).
  - Branch tracking (ex: main branch).
  - Symbolic reference (HEAD).

- Helm repos tracking : Helm always use semantic versioning
  - Specific version v1.2
  - Range 1.2.* or >=1.2.0 <1.3.0.
  - Latest * or >=0.0.0

Git – using tag

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: example
  namespace: argocd
spec:
  destination:
    namespace: default
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: guestbook
    repoURL: "https://github.com/mabusaa/argocd-example-apps.git"
    targetRevision: v1
```

 # using tag name

Git – commit SHA

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: example
  namespace: argocd
spec:
  destination:
    namespace: default
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: guestbook
    repoURL: "https://github.com/mabusaa/argocd-example-apps.git"
    targetRevision: 2455bb6
```

 # using the short commit id or the full commit id

Git – branch

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: example
  namespace: argocd
spec:
  destination:
    namespace: default
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: guestbook
    repoURL: "https://github.com/mabusaa/argocd-example-apps.git"
    targetRevision: main          # branch name
```

Git – symbolic reference

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: example
  namespace: argocd
spec:
  destination:
    namespace: default
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: guestbook
    repoURL: "https://github.com/mabusaa/argocd-example-apps.git"
    targetRevision: HEAD          # symbolic reference
```

Helm – Specific version

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    chart: sealed-secret
    repoURL: "https://bitnami-labs.github.io/sealed-secrets"
    targetRevision: 1.16.1          # specific version
```

Helm – Range

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    chart: sealed-secret
    repoURL: "https://bitnami-labs.github.io/sealed-secrets"
    targetRevision: '>=3.0.0 <4.1.0'      # recent version that is smaller than 4.1.0
```

Helm – Latest version

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    chart: sealed-secret
    repoURL: "https://bitnami-labs.github.io/sealed-secrets"
    targetRevision: *                    # latest version
```

Practice: Tracking Git Tag

- Create a tag for your manifests repo with name v1.
- Define an Argo CD application that track git tag v1.
- Apply the application using kubectl and verify its synced and working as expected.

Practice: Tracking Commit SHA

- Define an Argo CD application that track a certain commit SHA.
- Apply the application using kubectl and verify its synced and working as expected.

Practice: Tracking HEAD

- Define an Argo CD application that track HEAD.
- Apply the application using kubectl and verify its synced.
- Update some manifests (ex: replicas) and notice how Argo CD will sync the application into the latest commit Id.

Practice: Tracking Helm Chart Range

- Define an Argo CD application that track and deploy a helm chart with major version 1 only, and it should always sync the latest minor or patch version.
  - Use this repo Url : https://bitnami-labs.github.io/sealed-secrets
  - Chart name : sealed-secrets
- Apply the application using kubectl and verify its synced.

Practice: Tracking Helm Chart Latest Version

- Define an Argo CD application that track and deploy a helm chart latest version, and if there is any new release, it should be discovered and synced by Argo CD.
  - Use this repo Url : https://kubernetes.github.io/ingress-nginx
  - Chart name : ingress-nginx
- Apply the application using kubectl and verify its synced.

Diffing Customization

Diffing Customization

- Argo CD allows you to optionally ignore differences of problematic resources/manifests.
- Examples when you might need diffing customization:
  - A controller or mutating webhook is altering the resources after it was submitted to Kubernetes at runtime in a manner which contradicts Git.
  - A Helm chart is using a template function such as randAlphaNum, which generates different data every time helm template is invoked.
  - There is a bug in the manifest, where it contains extra/unknown fields from the actual K8s spec.
- Diffing customization can be configured at application level or at a system level.

Ignoring differences Options

- Argo CD allows ignoring differences using below options:
  - RFC6902 JSON patches at a specific JSON path (json pointers)
  - JQ path expressions.
  - Ignore differences from fields owned by specific managers defined in metadata.managedFields.

Application level – Json Pointers

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: kustomize-guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: kustomize-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
ignoreDifferences:
  - group: apps
    kind: Deployment
    jsonPointers:
    - /spec/replicas
```

 # Will ignore differences between live and desired states during the diff.
 # this will ignore differences in spec.replicas for all deployments for this application.

Application level – Json Pointers – for specific resource

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: kustomize-guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: kustomize-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
ignoreDifferences:
  - group: apps
    kind: Deployment
    name: guestbook
    jsonPointers:
    - /spec/replicas
```

this will ignore differences in spec.replicas for deployment with name guestbook .

Application level – jq expressions

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: kustomize-guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: kustomize-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
ignoreDifferences:
  - group: apps
    kind: Deployment
    jqPathExpressions:
    - .spec.template.spec.initContainers[] | select(.name == "injected-init-container")
```

Use JQ path expressions to identify list items based on item content

Application level – by specific managers

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: kustomize-guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: kustomize-guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
ignoreDifferences:
  - group: *
    kind: *
    managedFieldsManagers:
    - kube-controller-manager
```

 # will ignore differences from all fields owned by kube-controller-manager for all resources belonging to this application

Practice: Diffing Customization

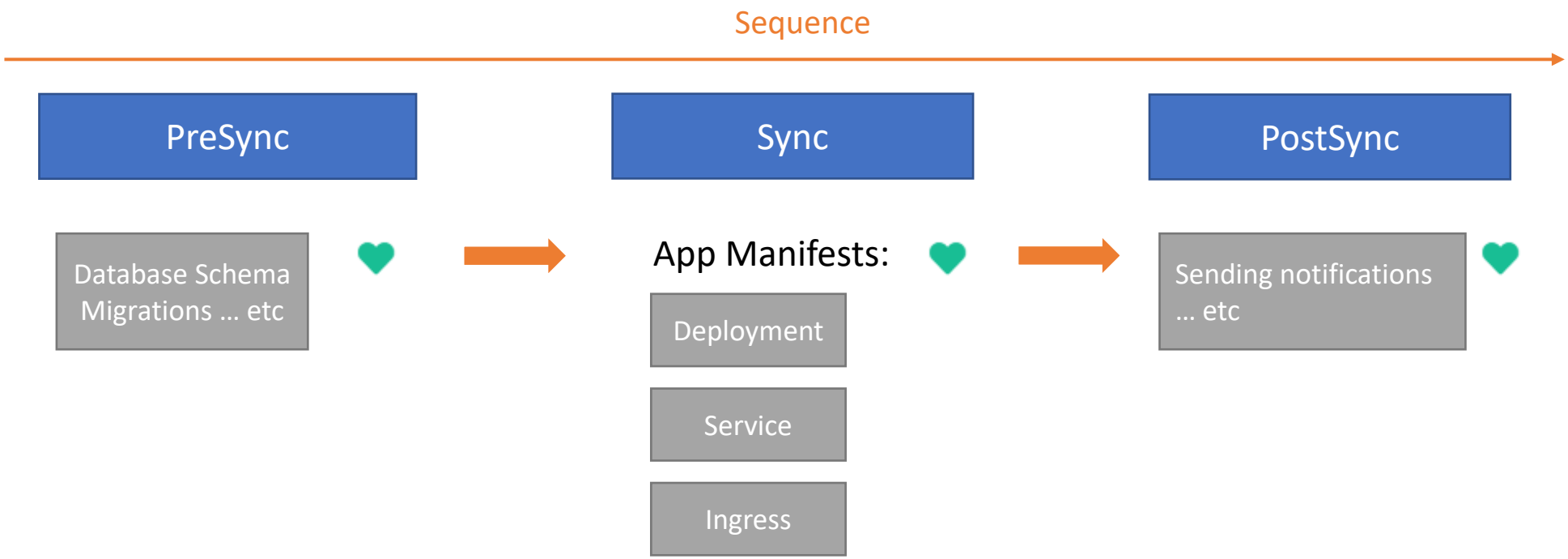
- Define an Argo CD application that ignore the differences for the deployment replicas.
  - Manifests: https://github.com/mabusaa/argocd-example-apps/tree/master/guestbook-with-sub-directories
- Apply the application using kubectl and verify its synced.
- Increase deployment replicas in git repo and verify that the application is still in sync status.

Practice: Real Case Diffing Customization for Istio

- Define an Argo CD application that deploy Istiod helm chart
  - Helm chart Repo: https://istio-release.storage.googleapis.com/charts
  - Chart name: istiod
  - Version: 1.13.4
- Apply the application using kubectl and verify in out of sync status.
- Fix the issue that causing the application to be in out of sync. And verify its in synced status.

Sync Phases & Resources Hooks

Sync Phases


Sequence
PreSync
Sync
PostSync
Database Schema Migrations … etc
App Manifests:
Deployment
Service
Ingress
Sending notifications … etc

Usage

- You can use phases using resources hooks annotation argocd.argoproj.io/hook on app manifests.

- You can add hook annotation to any manifests in your git repo:

```
apiVersion: batch/v1
kind: Job
metadata:
  generateName: database-migrations-
  annotations:
   argocd.argoproj.io/hook: PreSync
```

- Hooks does not run during selective sync.

Resources Hook types

Hook	Description
PreSync	Executes prior to the app manifests
Sync	Executes after all PreSync hooks successfully completed and healthy.
PostSync	Executes after all Sync hooks successfully completed and healthy.
Skip	Argo CD will skip syncing the manifest.
SyncFail	Executes when the sync operation fails.

Hook Deletion Policies

- Hooks recourses can be automatically deleted by several options using this annotation argocd.argoproj.io/hook-delete-policy

Deletion policy	Description
HookSucceeded	The hook resource is deleted after the hook succeeded.
HookFailed	The hook resource is deleted after the hook failed.
BeforeHookCreation	Any existing hook resource is deleted before the new one is created (default policy)

Resources Hook

```
apiVersion: apps/v1
kind: Job
metadata:
  annotations:
    argocd.argoproj.io/hook: PreSync
  name: db-job
spec:
.....
```

This resource will be executed in PreSync phase

Resource Deletion Policy

```
apiVersion: apps/v1
kind: Job
metadata:
  annotations:
    argocd.argoproj.io/hook: PreSync
    argocd.argoproj.io/hook-delete-policy: HookSucceeded
  name: db-job
spec:
.....
```

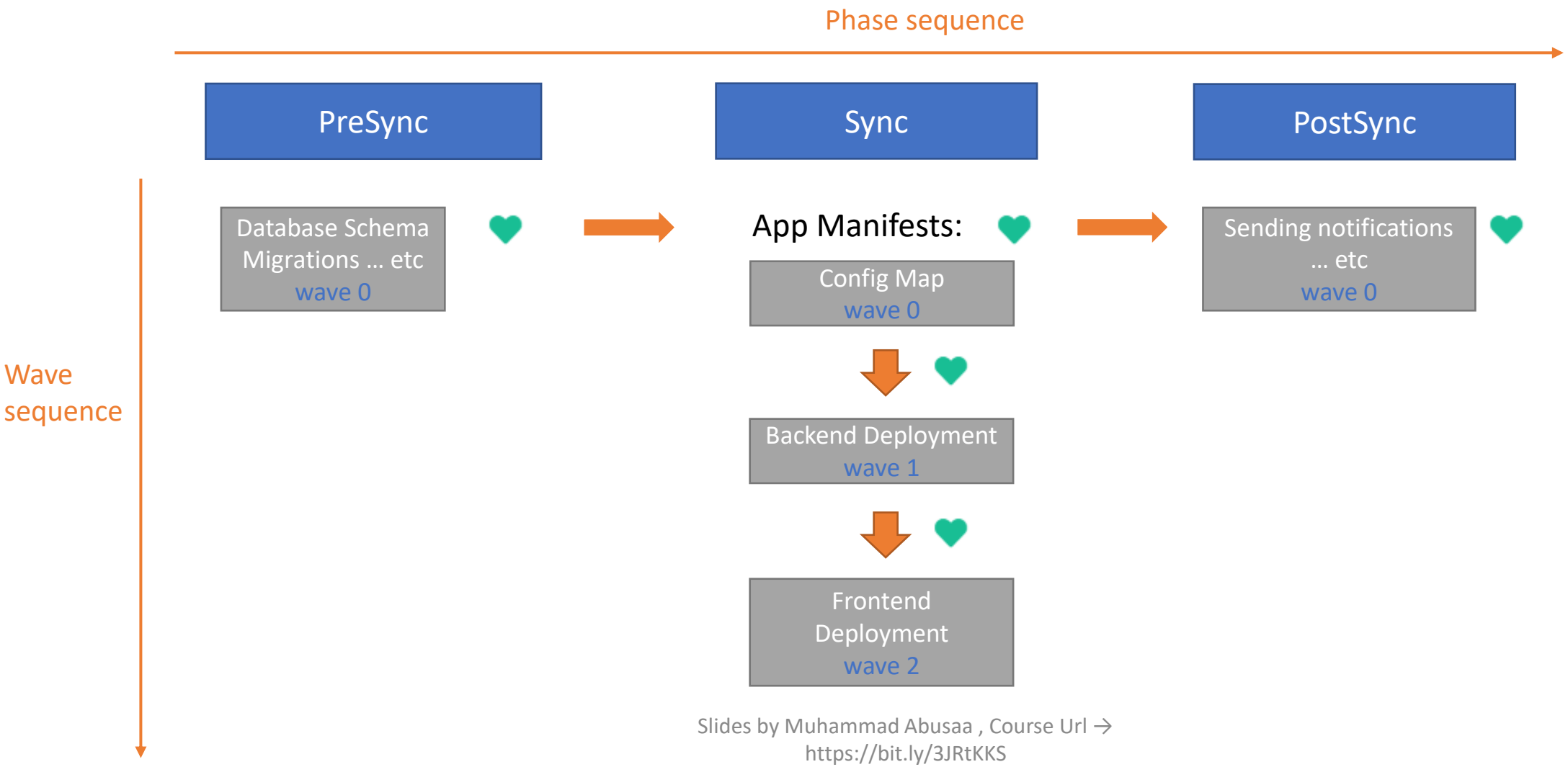
This resource will be executed in PreSync phase, and resource will be deleted after completion successfully

Practice Resources Hooks

- Create an application with manifests synced as below:
  - One job in PreSync phase, this job should be deleted once Succeeded.
  - One job in Sync phase, this job should be deleted if new resource is created.
  - One job in PostSync phase, this job should be deleted if failed.
  - Add a job manifest that will be skipped by ArgoCD.

Sync Waves

Sync Phases and Waves



Slides by Muhammad Abusaa , Course Url → https://bit.ly/3JRtKKS

Sync Waves

- A Sync-wave is a way to order how Argo CD applies the manifests.
- All manifests have a wave of zero by default.
- You can set these by using the argocd.argoproj.io/sync-wave annotation.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
  annotations:
   argocd.argoproj.io/sync-wave: “-1”
```

- Waves values can be negative or positive.
- ArgoCD starts with the lowest wave value to the highest.
- Next wave will start if previous wave resources are healthy
  - Default delay between waves are 2 seconds.

How ArgoCD starts sync actions

- When Argo CD starts a sync , resources get placed in the following order:
  - The phase.
  - The wave they are in (lower values first).
  - By kind
    - Namespace.
    - Network policy.
    - Resource quota.
    - Limit range.
    - … etc
  - By name.

Sync wave

```
apiVersion: apps/v1
kind: Job
metadata:
  annotations:
    argocd.argoproj.io/hook: PreSync
    argocd.argoproj.io/sync-wave: "-1"
  name: db-job
spec:
.....
```

This resource will be executed in PreSync phase in wave -1

Practice Sync Waves

- Create an application with manifests synced as below:
  - Two database jobs
    - First job in PreSync phase, synced at wave 0.
    - Second job in PreSync phase, synced at wave 1.
  - Backend replicaset and service in Sync phase, synced at wave 0.
  - One job to bring maintenance page up in Sync phase, synced at wave 1.
  - Frontend replicaset and service in Sync phase, synced at wave 2.
  - One job to bring maintenance page down in Sync phase, synced at wave 3.

Defining K8s Clusters

K8s Clusters in ArgoCD

- ArgoCD can deploy application resources into local cluster or remote clusters.
- By default ArgoCD has the permission to deploy into the local cluster where its running.
- We can add remote k8s clusters information including credentials as k8s secrets.
  - Each secret must have label of `argocd.argoproj.io/secret-type: cluster`
  - Each cluster must have the below data:
    - Name.
    - Server (cluster api server url).
    - Config (an option to authenticate to the cluster).
    - namespaces (optional): comma-separated list of namespaces which are accessible in that cluster.
- You can add remote clusters declarativly or using cli.

K8s Clusters in ArgoCD

- There are many options to authenticate to remote clusters:
  - Basic authentication (username and password)
  - Bearer token authentication.
  - IAM authentication configuration (suitable for cloud k8s clusters).
  - External provider command to supply client credentials.

Cluster - declaratively

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: my-cluster
  namespace: argocd
  labels:
    argocd.argoproj.io/secret-type: cluster
stringData:
  name: mycluster
  server: https://mycluster.com
  config: |
    {
      "bearerToken": "<authentication token>",
      "tlsClientConfig": {
        "insecure": false,
        "caData": "<base64 encoded certificate>"
      }
    }
```

Cluster - CLI

argocd cluster add context

where the context is the cluster context name that exist in kubeconfig file

Practice: Remote Clusters

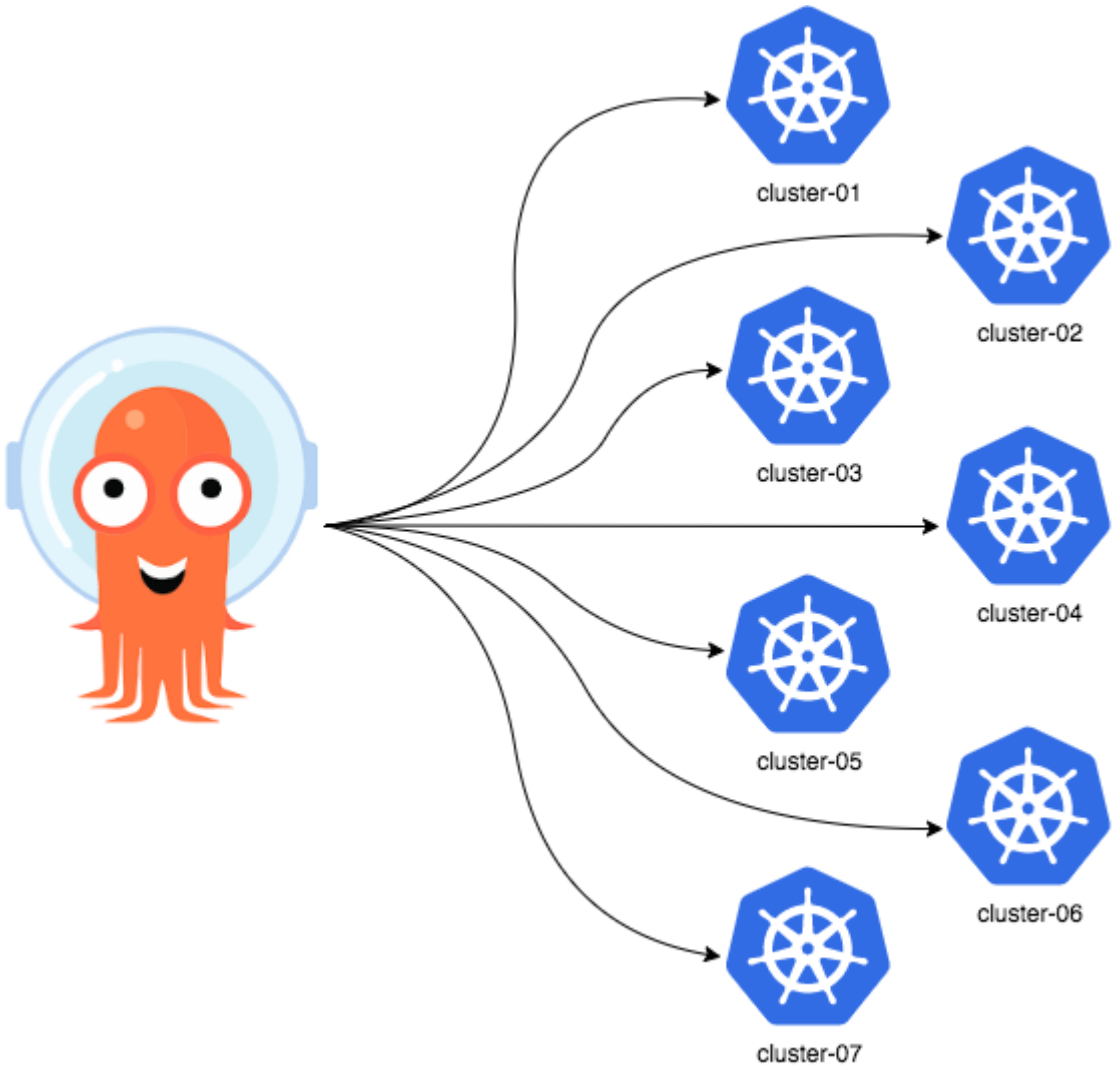
- Create a new k8s cluster in any remote environment (cloud or on-premise or local.)
- Connect ArgoCD to the remote cluster.
- Create an ArgoCD application that deploy manifests into the remote cluster.

What is ApplicationSet

ApplicationSet

Its a controller and CRD that provides:

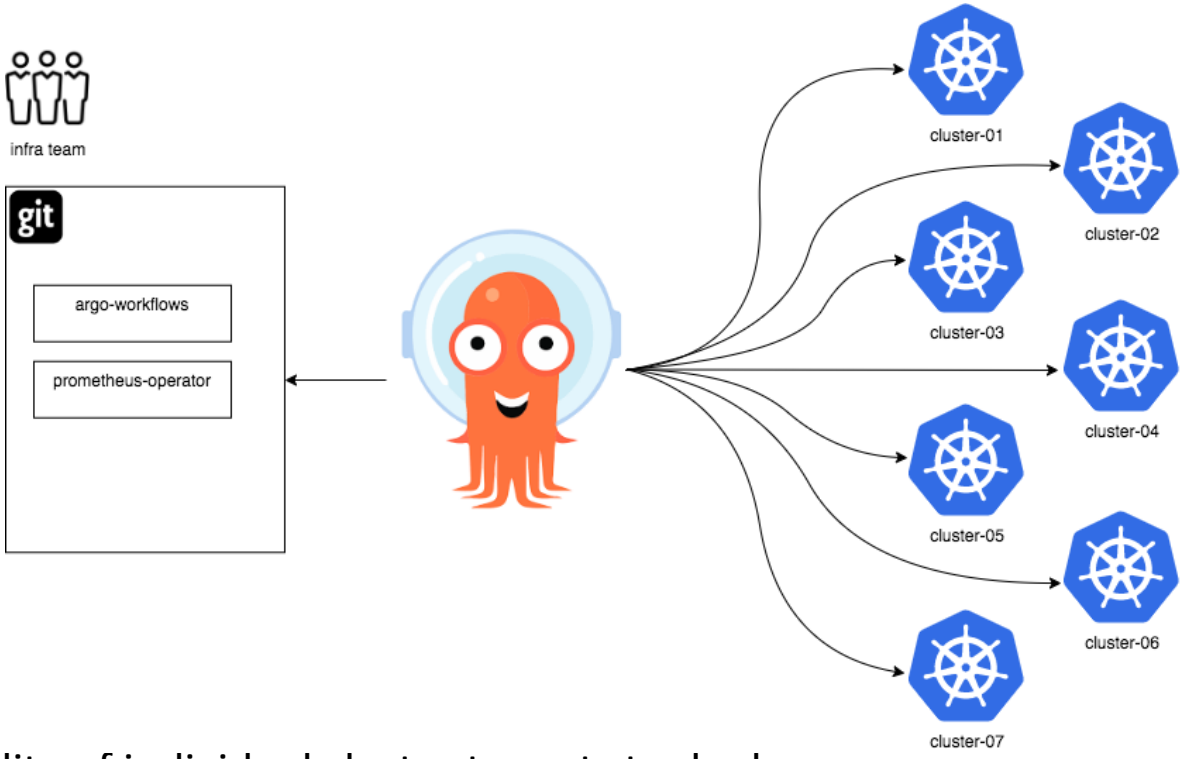
- Automating the applications generations.
- More Flexibility when managing Argo CD Applications across a large number of clusters.
- Ability to make self-service usage on multitenant clusters.



Use cases

- Use a single Kubernetes manifest to deploy into multiple Kubernetes clusters with Argo CD.

- Use a single Kubernetes manifest to deploy multiple applications from one or multiple Git repositories with Argo CD.

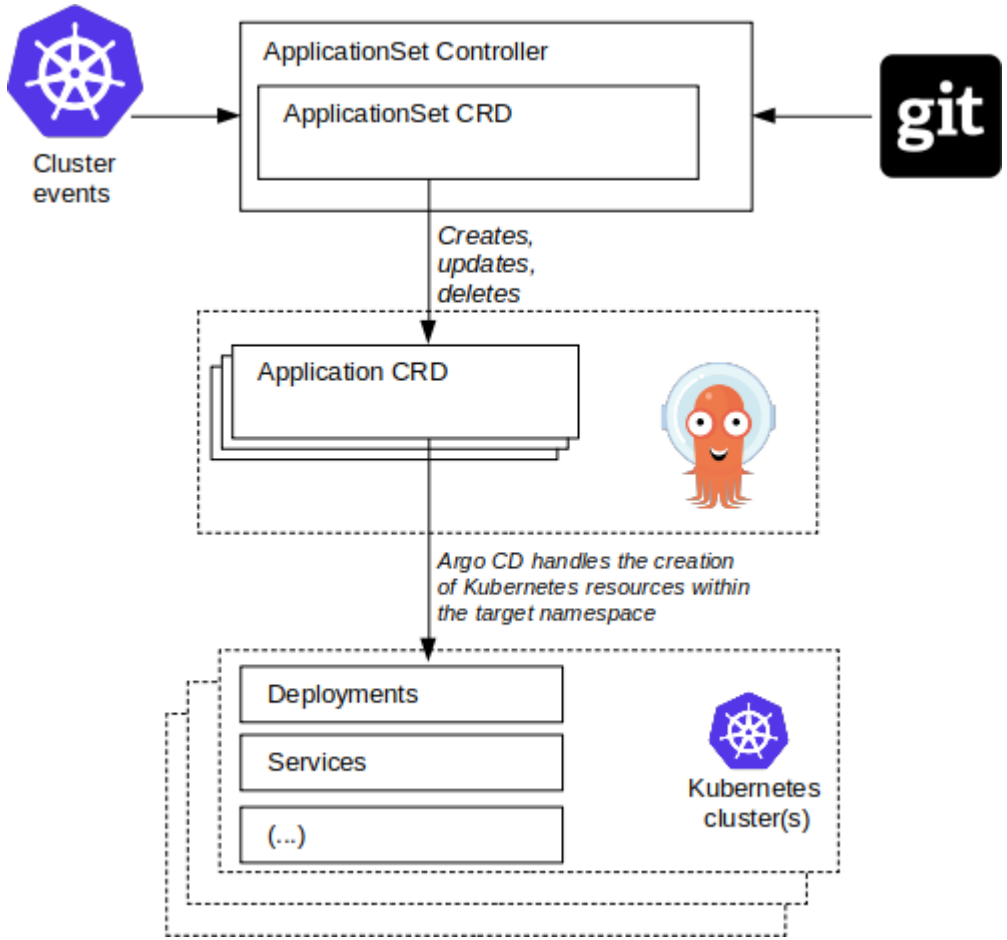


- Multitenancy support, improves the ability of individual cluster tenants to deploy applications using Argo CD.
  Ex: developers can deploy services into specific cluster/namespace without engaging clusters admins.

How it works

- ApplicationSet controller only responsible of creating, updating and deleting application in ArgoCD namespace.

- ApplicationSet does not modify k8s resources. Its Application controller responsibility to deploy resources into destination clusters.



ApplicationSet yaml structure

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: guestbook
  namespace: argocd
spec:
  destination:
    namespace: guestbook
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: guestbook
    repoURL: "https://github.com/argoproj/argocd-example-apps.git"
    targetRevision: HEAD
```

```yaml
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: guestbook
spec:
  generators:
  - list:
    elements:
    - cluster: engineering-prod-1
      url: https://1.2.3.4
    - cluster: engineering-prod-2
      url: https://2.4.6.8
  template:
    metadata:
      name: '{{cluster}}-guestbook'
    spec:
      destination:
        server: '{{url}}'
        namespace: guestbook
      source:
        repoURL: https://github.com/argoproj/argocd-example-apps.git
        targetRevision: HEAD
        path: guestbook/{{cluster}}
```

Argo applicationSet object “CRD”

Name of applicationSet

List generator

List of elements (params)

Name of application

Destination cluster

Source of manifests

Generators

- Generators are responsible for generating parameters, which are then rendered into the application template.

- Generators are primarily based on the data source that they use to generate the template parameters.

Installation

- ArgoCD v2.3+ includes ApplicationSet by default.

```
argocd-application-controller-0                    1/1     Running
argocd-applicationset-controller-66689cbf4b-tpx6f  1/1     Running
argocd-dex-server-64cb85bf46-xndrp                 1/1     Running
argocd-notifications-controller-5f8c5d6fc5-clcm5   1/1     Running
argocd-redis-d486999b7-9j9nm                       1/1     Running
argocd-repo-server-8576d68689-82lh5                1/1     Running
argocd-server-cb57f685d-tsh4s                      1/1     Running
```

- Prior v2.3, ApplicationSet has a unique installation manifest.

kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj-labs/applicationset/v0.3.0/manifests/install.yaml

Generators

Generators

- Generators are responsible for generating parameters and are based on data sources, which can be used to generate parameters into the application template.
- Types of generators
  - List generator
  - Cluster generator
  - Git generator
  - Matrix generator
  - Merge generator
  - SCM provider generator
  - Pull request generator
  - Cluster Decision Resource generator

List Generator

- **List generator**: The List generator allows you to target Argo CD Applications to clusters based on a fixed list of cluster name/URL values.

- any key/value element pair is supported.
- Clusters need to be pre-defined in Argo CD.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: guestbook
spec:
  generators:
  - list:
    elements:
    - cluster: engineering-prod-1
      url: https://1.2.3.4
    - cluster: engineering-prod-2
      url: https://2.4.6.8
template:
  metadata:
    name: '{{cluster}}-guestbook'
  spec:
    destination:
      server: '{{url}}'
      namespace: guestbook
    source:
      repoURL: https://github.com/argoproj/argocd-example-apps.git
      targetRevision: HEAD
      path: guestbook/{{cluster}}
```

List generator

List of elements (params)

Name of application

Destination cluster

Source of manifests

Cluster Generator

- The Cluster generator allows you to generate applications based on the list of clusters defined within Argo CD
- The following parameter values to the Application template for each cluster:
  - name
  - nameNormalized ('name' but normalized to contain only lowercase alphanumeric characters, '-' or '.')
  - server
  - metadata.labels.<key> (for each label in the Secret)
  - metadata.annotations.<key> (for each annotation in the Secret)
- Additional key/value pairs can be set manually via values field.

Cluster generator – targeting all clusters.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: guestbook
spec:
  generators:
  - cluster:  {}
template:
    metadata:
      name: '{{name}}-guestbook'
    spec:
      destination:
        server: '{{server}}'
        namespace: guestbook
      source:
        repoURL: https://github.com/argoproj/argocd-example-apps.git
        targetRevision: HEAD
        path: guestbook
```

All clusters

Cluster generator – targeting specific clusters using labels.

```yaml
kind: Secret
data:
  # (... Cluster details ...)
metadata:
  labels:
    argocd.argoproj.io/secret-type: cluster
    staging: "true"
# (...)
```

Note: By default local cluster is not added as k8s secret, you can add it declaratively or using Web UI by editing the cluster name in clusters page.
Then you can use it by matching labels.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: guestbook
spec:
  generators:
  - cluster:
      selector:
        matchLabels:
          staging: true
template:
    metadata:
      name: '{{name}}-guestbook'
    spec:
      destination:
        server: '{{server}}'
        namespace: guestbook
      source:
        repoURL: https://github.com/argoproj/argocd-example-apps.git
        targetRevision: HEAD
        path: guestbook
```

Targeting clusters with label stage = true

Cluster generator – passing additional values.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: guestbook
spec:
  generators:
  - cluster:
      selector:
        matchLabels:
          type: staging
      values:
        revision: HEAD
  - cluster:
      selector:
        matchLabels:
          type: production
      values:
        revision: stable
template:
  metadata:
    name: '{{name}}-guestbook'
  spec:
    destination:
      server: '{{server}}'
      namespace: guestbook
    source:
      repoURL: https://github.com/argoproj/argocd-example-apps.git
      targetRevision: '{{values.revision}}'
      path: guestbook
```

Targeting clusters with label type = staging

Targeting clusters with label type = production

setting revision based on generator values

Git Generator

- The Git generator: generates parameters based on files or folders that are contained within the Git repository defined within the applicationset resource.

- Contains two subtypes:
  - **Git directory generator**: generates parameters using the directory structure of a specified Git repository.
  - **Git file generator:** generates parameters using the contents of JSON/YAML files found within a specified repository.

Git Generator

- The git generator parameters are:
  - **{{path}}:** The directory paths within the Git repository that match the path wildcard.
  - **{{path.basename}}:** For any directory path within the Git repository that matches the path wildcard, the right-most path name is extracted (**e.g. /directory/directory2 would produce directory2**).
  - **{{path.basenameNormalized}}:** This field is the same as path.basename with unsupported characters replaced with - (e.g. a path of /directory/directory_2, and path.basename of directory_2 would produce directory-2 here).

Git Directory Generator

Suppose you have a Git repository with the following directory structure, applicationset will generate two applications.

```
├── argo-workflows
│   ├── kustomization.yaml
│   └── namespace-install.yaml
└── prometheus-operator
    ├── Chart.yaml
    ├── README.md
    ├── requirements.yaml
    └── values.yaml
```

```yaml
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: cluster-addons
spec:
  generators:
  - git:
      repoURL: https://github.com/argoproj-labs/applicationset.git
      revision: HEAD
      directories:
      - path: examples/git-generator-directory/cluster-addons/*
  template:
    metadata:
      name: '{{path.basename}}'
    spec:
      project: default
      source:
        repoURL: https://github.com/argoproj-labs/applicationset.git
        targetRevision: HEAD
        path: '{{path}}'
      destination:
        server: https://kubernetes.default.svc
        namespace: '{{path.basename}}'
```

Git Directory – exclude feature

- The Git directory generator will automatically exclude folders that begin with . (such as .git).
- The Git directory generator also supports an exclude option in order to exclude directories.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: cluster-addons
spec:
  generators:
  - git:
      repoURL: https://github.com/argoproj-labs/applicationset.git
      revision: HEAD
      directories:
      - path: examples/git-generator-directory/cluster-addons/*
      - path: examples/git-generator-directory/excludes/cluster-addons/exclude-helm-guestbook
        exclude: true
  template:
    metadata:
      name: '{{path.basename}}'
    spec:
      project: default
      source:
        repoURL: https://github.com/argoproj-labs/applicationset.git
        targetRevision: HEAD
        path: '{{path}}'
      destination:
        server: https://kubernetes.default.svc
        namespace: '{{path.basename}}'
```

Git Files Generator

Suppose you have a Git repository with the following structure:

```
├── apps
│   └── guestbook
│       ├── guestbook-ui-deployment.yaml
│       ├── guestbook-ui-svc.yaml
│       └── kustomization.yaml
├── cluster-config
│   └── engineering
│       ├── dev
│       │   └── config.json
│       └── prod
│           └── config.json
└── git-generator-files.yaml
```

config.json

```
{
  "aws_account": "123456",
  "asset_id": "11223344",
  "cluster": {
    "owner": "cluster-admin@company.com",
    "name": "engineering-dev",
    "address": "https://1.2.3.4"
  }
}
```

```yaml
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: guestbook
spec:
  generators:
  - git:
      repoURL: https://github.com/argoproj-labs/applicationset.git
      revision: HEAD
      files:
      - path: "examples/git-generator-files-discovery/cluster-config/**/config.json"
  template:
    metadata:
      name: '{{cluster.name}}-guestbook'
    spec:
      project: default
      source:
        repoURL: https://github.com/argoproj-labs/applicationset.git
        targetRevision: HEAD
        path: "examples/git-generator-files-discovery/apps/guestbook"
      destination:
        server: '{{cluster.address}}'
        namespace: guestbook
```

More Generators

- **Matrix generator**: The Matrix generator may be used to combine the generated parameters of two separate generators.
- **Merge generator**: The Merge generator may be used to merge the generated parameters of two or more generators. Additional generators can override the values of the base generator.
- **SCM Provider generator**: The SCM Provider generator uses the API of an SCM provider (eg GitHub) to automatically discover repositories within an organization.
- **Pull Request generator**: The Pull Request generator uses the API of an SCMaaS provider (eg GitHub) to automatically discover open pull requests within an repository.
- **Cluster Decision Resource generator**: The Cluster Decision Resource generator is used to interface with Kubernetes custom resources that use custom resource-specific logic to decide which set of Argo CD clusters to deploy to.

Practice: List Generator

- Deploy an application manifests into two clusters:
  - Local cluster (Where ArgoCD running)
  - Remote cluster.

  Use any app manifests,

  eg: https://github.com/mabusaa/argocd-example-apps.git , path: guestbook.

- Use ApplicationSet with list generator to achieve this.

- Make sure remote cluster is added to ArgoCD before applying the ApplicationSet.

Practice: Cluster Generator

- Add local cluster as a secret. Add label as `environment=dev` and `non-prod=true`.
- Add one more label `non-prod=true` to remote cluster.

1. Practice1: Deploy one application into all clusters:
   - Application name should include cluster name.
   - Use one ApplicationSet manifest with cluster generator to achieve this.

2. Practice2: Deploy one application into all clusters with matching label non-prod=true.
   - Application name should include cluster name.
   - Use one ApplicationSet with cluster generator to achieve this.

Use this git repo as a source:
https://github.com/mabusaa/argocd-example-apps.git , path guestbook

Practice: Git Directory Generator

1. Deploy multiple applications into local cluster.
2. Application name and namespace should be the same as base path name that's in git.
3. Use one ApplicationSet manifest with git directory generator to achieve this.

Deploy all helm charts under path (helmcharts/security-policy-charts).

https://github.com/mabusaa/argocd-example-apps.git

Practice: Matrix Generator

1. Deploy multiple applications into all clusters that has label non-prod=true
2. Application name and namespace should include base path name that's in git and cluster-name where its deployed into.
3. Use one ApplicationSet manifest with matrix generator to achieve this.

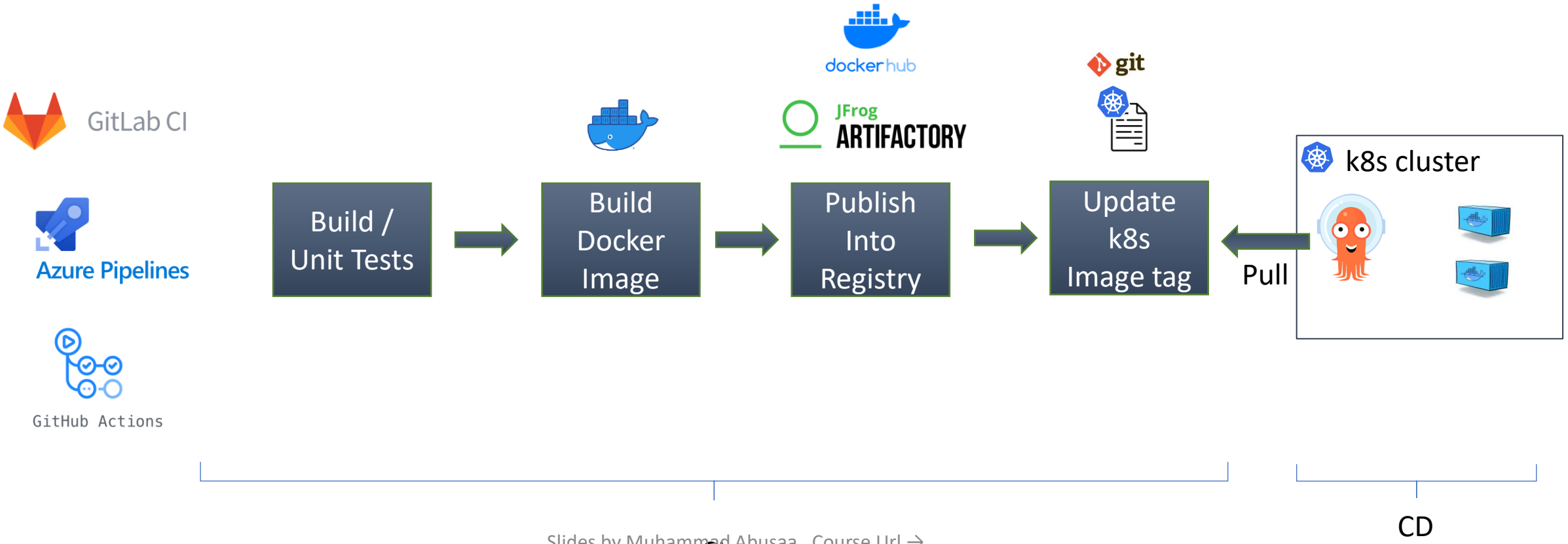
Deploy all helm charts under path (helmcharts/security-policy-charts).

https://github.com/mabusaa/argocd-example-apps.git

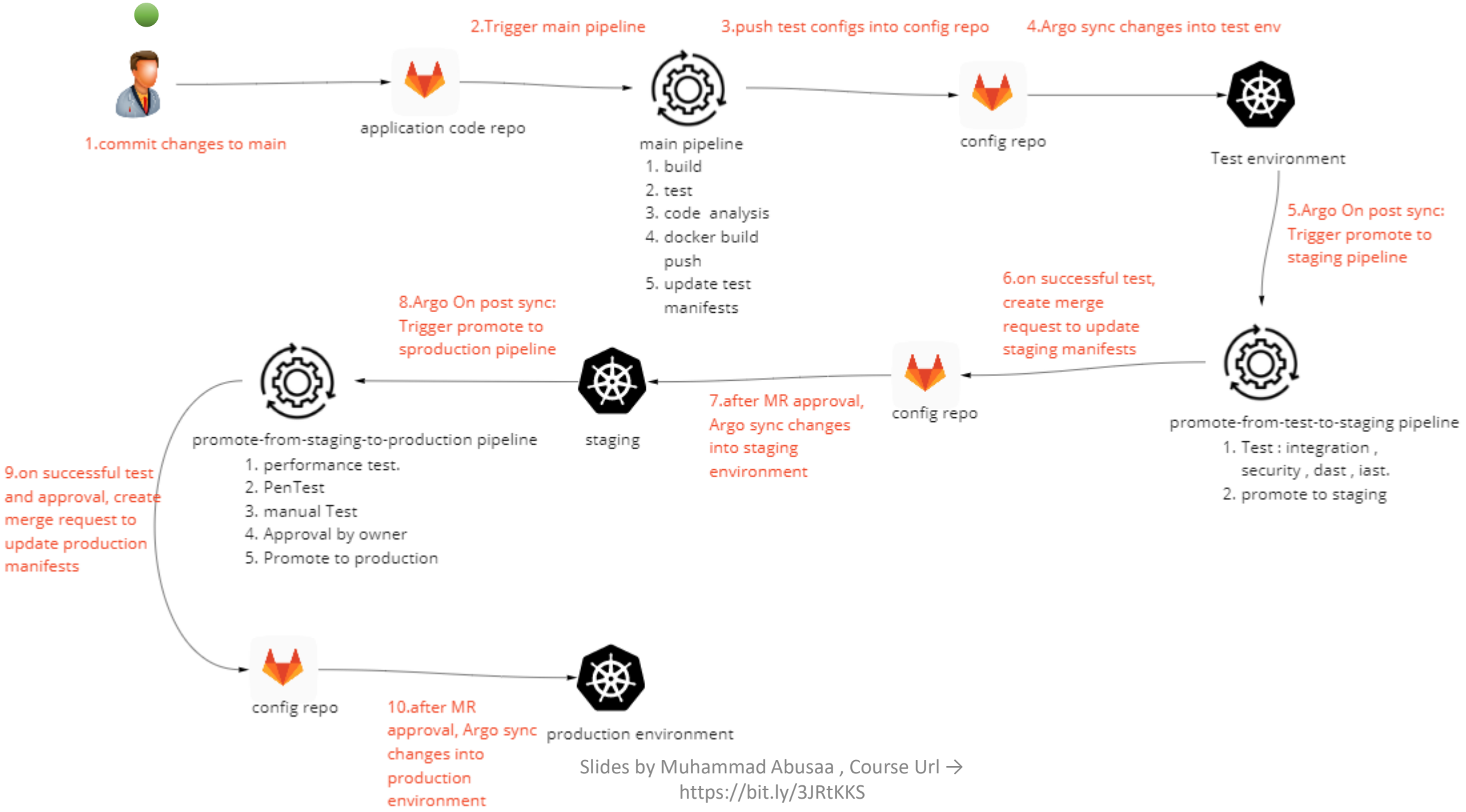
Automation by CI Pipelines

GitOps Pull Model

- Simple CI model



Full Release - Sample flow

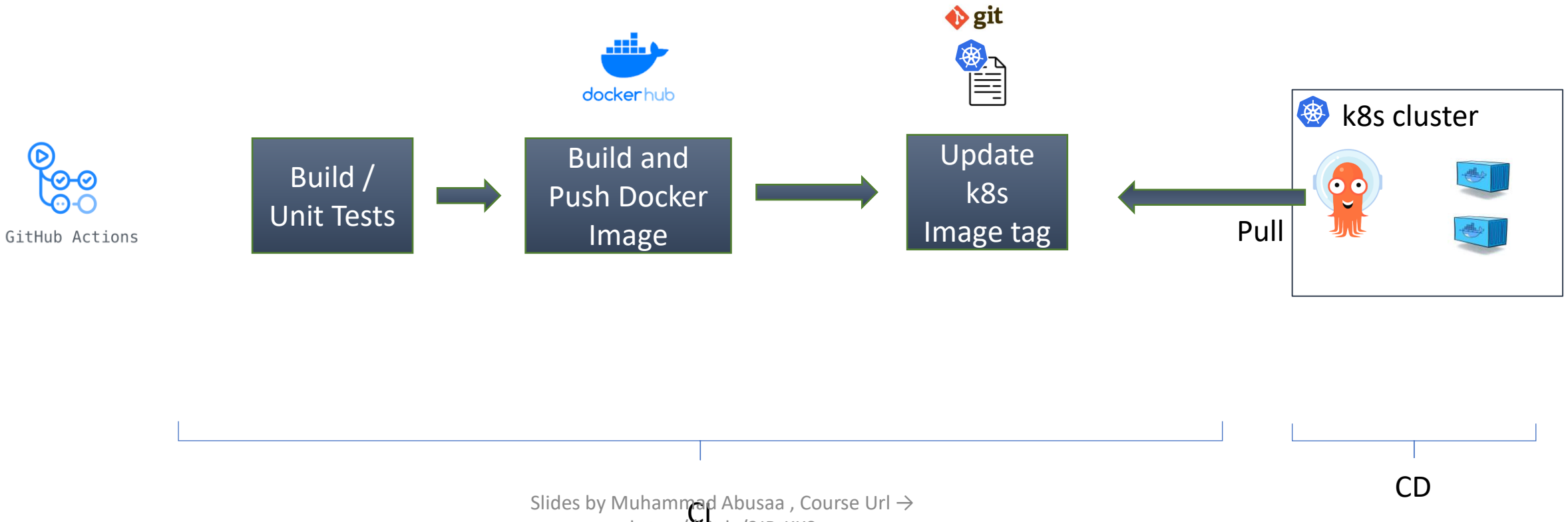


Slides by Muhammad Abusaa , Course Url → https://bit.ly/3JRtKKS

Practice: Basic CI/CD

- Automate a sample webapp deployment into dev environment using CI pipeline and Argo CD.
- Webapp repos:
  - **Webapp main repo**: that contain the code, dockerfile and CI definition.
  - **Manifests (config) repo**: that contains the helm chart related to webapp.
- CI pipeline should consist of the below jobs:
  - **First job**: Build and test code job.
  - **Second job**: Build and push docker image into registry.
    - Use commit SHA for Image tag.
  - **Third job**: Update image tag in app manifests (config) repo.
- Every commit on webapp code repo main branch should trigger the pipeline.
- Create Argo CD application manifest that will track webapp manifests repo (main branch) and enable Auto sync.
- You can use/fork the below repos:
  - Webapp code: https://github.com/mabusaa/argocd-course-webapp
  - Webapp manifests: https://github.com/mabusaa/argocd-course-webapp-config

Basic CICD Practice



How to Structure Git Repos

Separated config repo is recommended

- Separating config repo from application code repo is highly recommended, why:
  - Cleaner Git history of what changes were made.
  - Your application might consist of services that are distributed in multiple Git repositories, but is deployed as a single unit.
  - Separation of access. Maintainers of source code maybe not be the same as the maintainers of application manifests (config).
  - Smaller overhead on Argo CD repo-controller, cloning config repo without the source code.

Slides by Muhammad Abusaa , Course Url → https://bit.ly/3JRtKKS

Environments (config) manifests

- **Types of environments:**
  - Static environments:
    - Test
    - Staging
    - Production
  - On-demand environments : created for a small period for feature testing. Can be created once a pull request created and destroyed once the pull request is closed.
- **How to handle on-demand environments creation/deletion**:
  - **On-demand environments** can be created by two options
    - **Option-1**: Use Application Set Pull Request Generator, it will handle creating ArgoCD applications per open pull request. And it will be Argo CD application will be destroyed once Pull request is closed.
    - **Option-2**: Single branch for on-demand environments, and a folder per environment and each folder will contain the related manifests. (You need to write a script using CI pipeline to create theses folders and manifests per Pull request).

Handling static environments (config) manifests

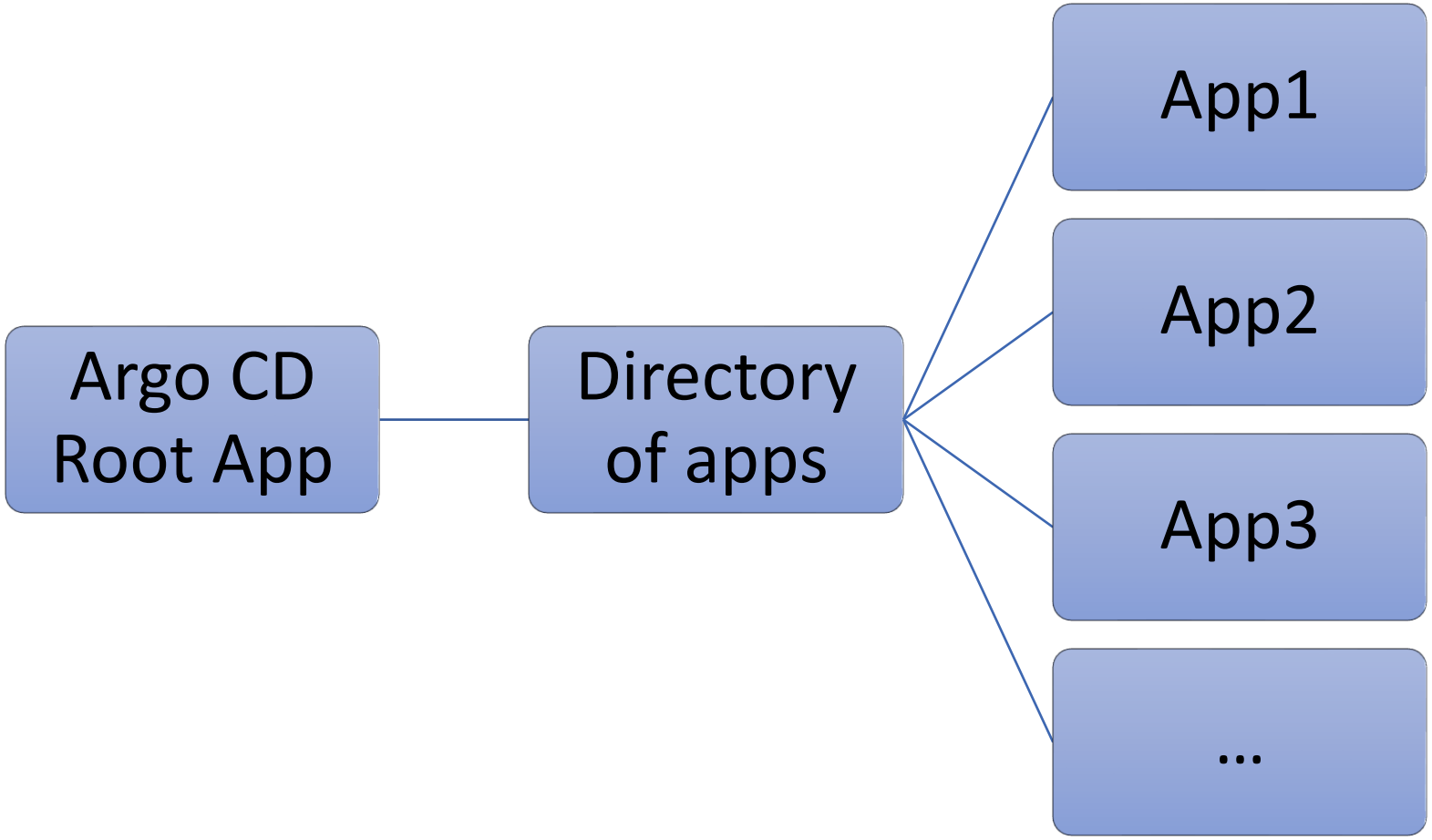
- How to handle static environments config ?
  - **Approach #1** : **Single branch** all static environments:
    - Eg: Main branch, contains the helm chart values files for-each static environment:
      - Values-test.yaml.
      - Values-staging.yaml.
      - Values-prod.yaml.
  - **Approach #2** : **Branch per static environment**
    - Test branch.
    - Staging branch.
    - Production branch. (You can use tags or commits SHA for production environments)

(App of Apps) How to Structure Apps Definitions

Managing many apps using one app

- Managing ArgoCD Apps manually using kubectl is not so effective, and can be solved by using app of apps pattern.
- App of apps: Create one ArgoCD App that responsible of managing other apps.
- Your can structure app of apps in many options
  - Root app that is tracking directory of ArgoCD apps manifests with enabling the recursion.
    - Any new app added to the tracked directory can be synced by ArgoCD directly.
  - Root app that is tracking helm chart that include all other apps.
    - Adding new apps will be within the helm chart , then it can be synced by ArgoCD directly.

Root app tracking directory of apps



Root app tracking directory of apps

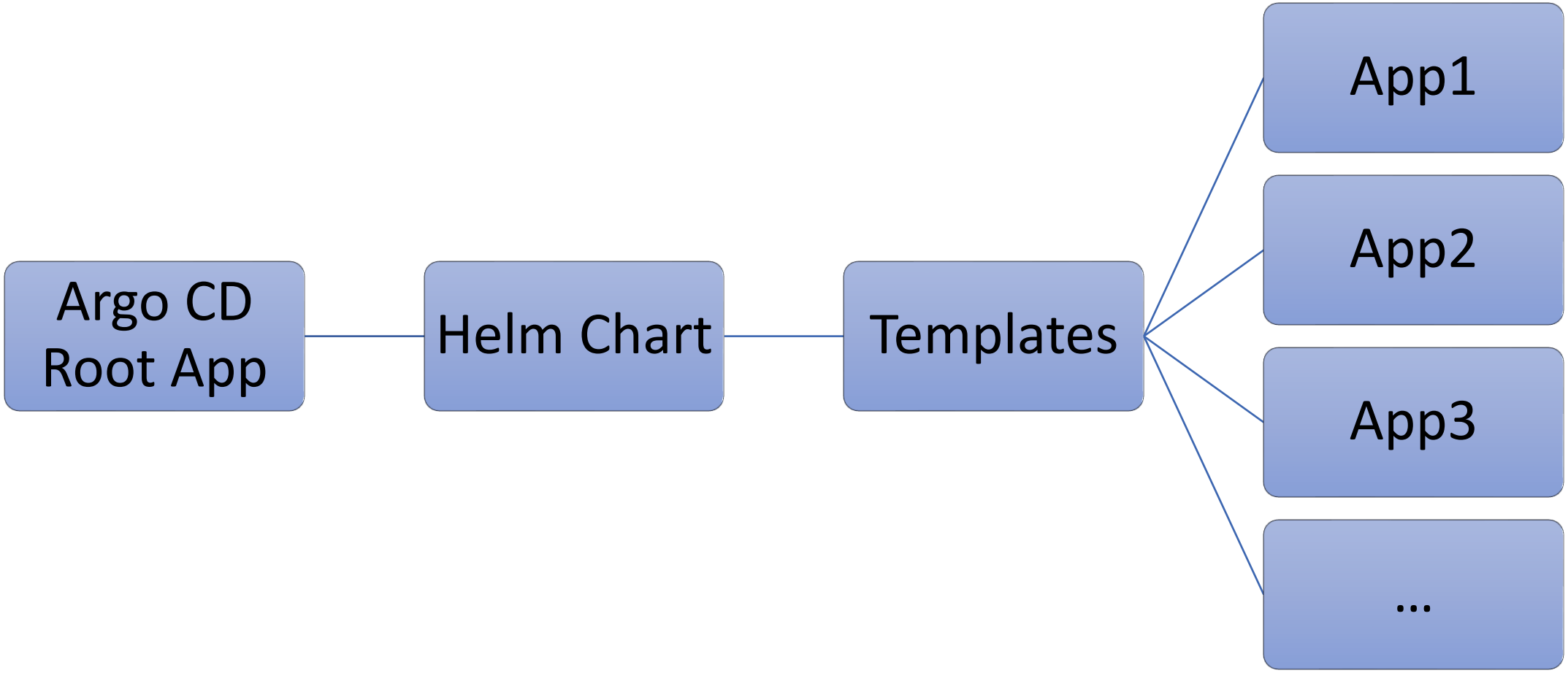
```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
labels:
    app.kubernetes.io/name: root-app-directory-approach
  name: root-app-directory-approach
  namespace: argocd
spec:
  destination:
    namespace: argocd
    server: https://kubernetes.default.svc
project: default
  source:
    path: directory-of-apps
    repoURL: https://github.com/mabusaa/argocd-course-app-of-apps.git
    targetRevision: main
    directory:
      recurse: true
  syncPolicy:
    automated: {}
```

Root app tracking directory of apps

- directory-of-apps
  - app1.yaml
  - app2.yaml
- root-app-directory-approach.yaml
- root-app-helm-approach.yaml

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: app1
  namespace: argocd
spec:
  destination:
    namespace: app1
    server: "https://kubernetes.default.svc"
  project: default
  source:
    path: guestbook
    repoURL: "https://github.com/mabusaa/argocd-example-apps.git"
    targetRevision: master
  syncPolicy:
    syncOptions:
      - CreateNamespace=true
```

Root app tracking helm chart approach



Root app tracking helm chart approach

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
labels:
    app.kubernetes.io/name: root-app-helm-approach
  name: root-app-helm-approach
  namespace: argocd
spec:
  destination:
    namespace: argocd
    server: https://kubernetes.default.svc
project: default
  source:
    path: apps-helm-chart/apps
    repoURL: https://github.com/mabusaa/argocd-course-app-of-apps.git
    targetRevision: main
syncPolicy:
    automated: {}
```

Root app tracking helm chart

- apps-helm-chart\apps
  - templates
    - helm-app1.yaml
    - helm-app2.yaml
    - kustomize-app3.yaml
    - sync-app4.yaml
  - Chart.yaml
  - values.yaml
- directory-of-apps
- root-app-directory-approach.yaml
- root-app-helm-approach.yaml

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: helm-app1
  namespace: argocd
  finalizers:
  - resources-finalizer.argocd.argoproj.io
spec:
  destination:
    namespace: helm-app1
    server: {{ .Values.spec.destination.server }}
  project: default
  source:
    path: helm-guestbook
    repoURL: {{ .Values.spec.source.repoURL }}
    targetRevision: {{ .Values.spec.source.targetRevision }}
  syncPolicy:
    automated:
      allowEmpty: true
      selfHeal: true
    syncOptions:
    - allowEmpty=true
    - CreateNamespace=true
```

Practice: App of Apps

- **Practice-1**: Manage applications that exist in below repo using app of apps pattern:
  - Apps repo: https://github.com/mabusaa/argocd-course-app-of-apps
  - Root app should track directory path `directory-of-apps` and its sub-directories.
  - Apps in above directory should be synced manually.
  - Deploy the root app using kubectl, and then sync all the apps created by the root app.
- **Practice-2**: Manage applications that exist in below repo using app of apps pattern:
  - Apps repo: https://github.com/mabusaa/argocd-course-app-of-apps
  - Root app should track helm chart `apps-helm-chart/apps`
  - Apps within helm chart should be synced automatically.
  - Deploy the root app using kubectl.

Best Practices

Separate config repo

- Use a separated config repo that contains app manifests.

Immutable manifests

- Use immutable manifests in production
  - Avoid using HEAD revision and use tags or commits SHA .

```yaml
source:
  path: guestbook
  repoURL: "https://github.com/
  targetRevision: HEAD
```

```yaml
source:
  path: guestbook
  repoURL: "https://github.com/
  targetRevision: v1
```

Replicas and HPA

- If you want HPA to control the number of replicas , then don’t include replicas in Git.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: guestbook-ui
spec:
  # do not include replicas in git, if HPA is enabled for this deployment
  # replicas: 1
```

Plan for secrets management

- Don’t store plain secrets in git.
- There are several solutions for secrets:
  - Within Git
    - Sealed secrets.
    - SOPS
  - External secrets store:
    - Hashicorp Vault.
    - External Secrets Operator.
    - Cloud secrets store.
      - Aws secret operator
    - … more

Plan for ArgoCD Instances

- Use a separated ArgoCD instance for production.
  - Use HA setup for production.

- Its recommended to have at least two instances:
  - Non-prod instance.
  - Prod instance.

Non-prod instance

Prod instance

App of Apps and ApplicationSet

- Use app of apps to manage ArgoCD application.
- Use Application Set and the power of generators to generate applications.