

Setup

When not using the preinstalled Web IDE provided by your Trainer, it's also possible to use your local computer.

Local Environment Requirements

In this Training its required to have the following tools locally installed on your computer:

- git
- git bash on Windows
- Argo CD CLI
- oc Tool (OpenShift Client) *Only when on OpenShift*
- kubectl

Argo CD Command line tool

Follow the instructions on [this](#) page to install the ArgoCD tool on your local computer.

oc tool

Follow the instructions on [this](#) page to install the oc tool on your local computer.

kubectl

Follow the instructions on [this](#) page to install the kubectl on your local computer.

Labs

[Argo CD](#) is a part of the [Argo Project](#) and affiliated under the [Cloud Native Computing Foundation \(CNCF\)](#) . The project is just under three years old, completely open source, and primarily implemented in Go.

As the name suggests, Argo CD takes care of the continuous delivery aspect of CI/CD. Continuous integration is handled by a CI tool such as GitLab CI/CD, Jenkins, Tekton or GitHub Actions. The core of Argo CD consists of a Kubernetes controller, which continuously compares the live-state with the desired-state. The live-state is tapped from the Kubernetes API, and the desired-state is persisted in the form of manifests in YAML or JSON in a Git repository. Argo CD helps to point out deviations of the states, to display the deviations or to autonomously restore the desired state.

Argo CD is deployed and operated on a Kubernetes-based container platform. It is possible to connect multiple Kubernetes and OpenShift clusters to one Argo CD instance.

[Argo CD](#) is a declarative, GitOps continuous delivery tool for Kubernetes.

[GitOps](#) is a way to do Kubernetes cluster management and application delivery. It works by using Git as a single source of truth for declarative infrastructure and applications. With GitOps, the use of software agents can alert on any divergence between Git with what's running in a cluster, and if there's a difference, Kubernetes reconcilers automatically update or rollback the cluster depending on the case. With Git at the center of your delivery pipelines, developers use familiar tools to make pull requests to accelerate and simplify both application deployments and operations tasks to Kubernetes.

Managing Kubernetes resources using a GitOps approach brings the following benefits:

- The definition of the manifests is done in a declarative way and a tool ensures the comparison between the desired and live manifests. The differences between the desired configurations and the actually applied manifests can be easily seen at any time.
- Rollbacks to older versions are easily possible with git revert or via the used GitOps tool (provided that the application also supports this).
- Manual adjustments directly on the container platform are immediately visible and can be automatically overwritten (self-healing).
- The git commit history is also a detailed audit log
- The developers describe the infrastructure in already known formats and tools like yaml and git.

Argo CD follows the GitOps pattern of using Git repositories as the source of truth for defining the desired application state.

Warning

Argo CD provides a mechanism to override the parameters of Argo CD applications. [The Argo CD parameter overrides](#) feature is provided mainly as a convenience to developers and is intended to be used in dev/test environments, vs. production environments.

Many consider this feature as anti-pattern to GitOps. So only use this feature when no other option is available!

Kubernetes manifests can be specified in several ways:

- [kustomize](#) applications
- [helm](#) charts
- [ksonnet](#) applications (deprecated)
- [jsonnet](#) files
- Plain directory of YAML/json manifests
- Any custom config management tool configured as a config management plugin

- acend gmbh

Argo CD automates the deployment of the desired application states in the specified target environments. Application deployments can track updates to branches, tags, or pinned to a specific version of manifests at a Git commit. See tracking strategies for additional details about the different tracking strategies available.

For a quick 10 minute overview of Argo CD, check out the demo presented to the Sig Apps community meeting:

Argo CD Architecture

Argo CD's core components are the API Server, the Repository Server and the Application Controller

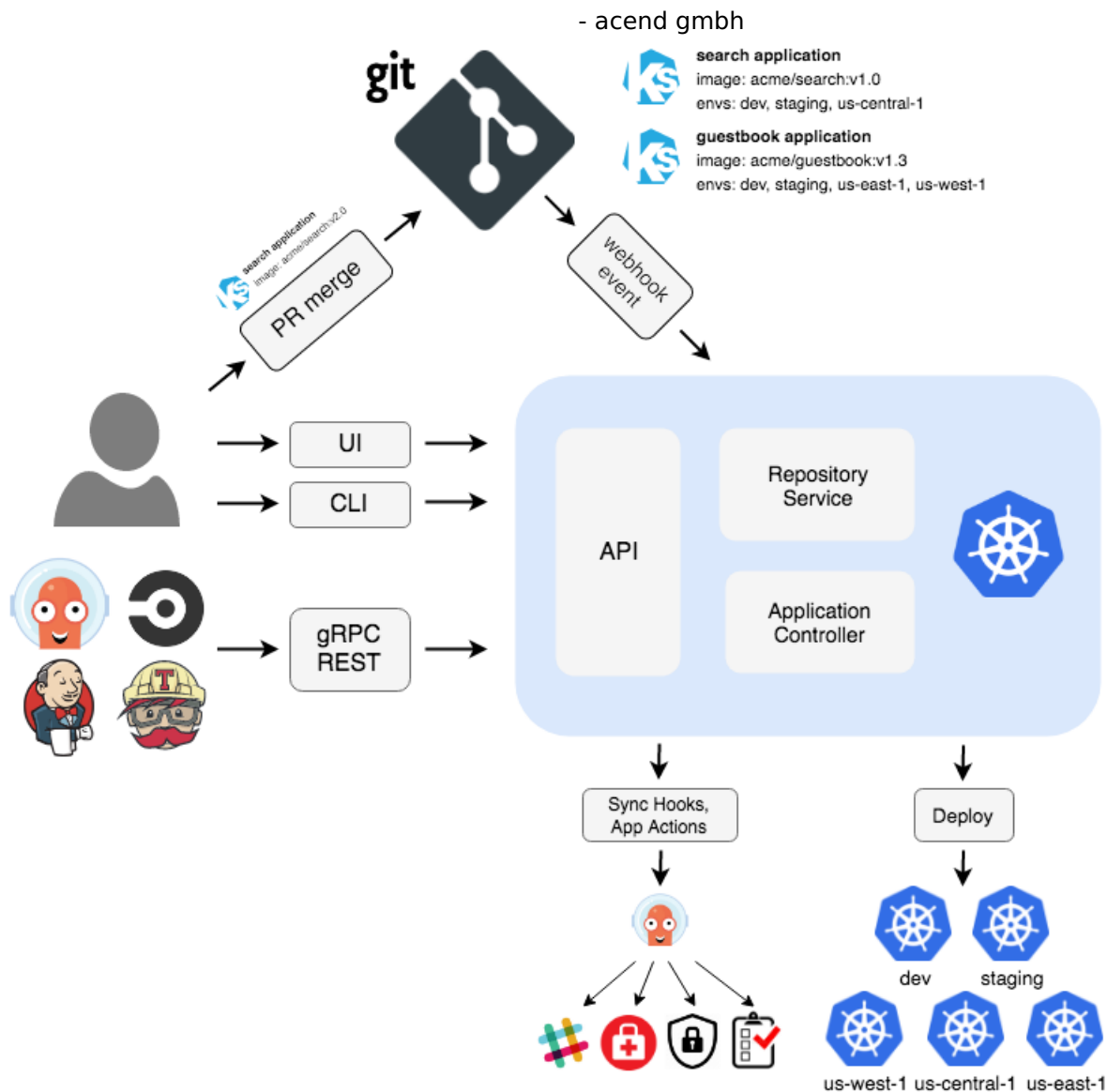


Image and component description source: <https://argoproj.github.io/argo-cd/>

API Server

The API server is a gRPC/REST server which exposes the API consumed by the Web UI, CLI, and CI/CD systems. It has the following responsibilities:

- application management and status reporting
- invoking of application operations (e.g. sync, rollback, user-defined actions)
- repository and cluster credential management (stored as K8s secrets)
- authentication and auth delegation to external identity providers
- RBAC enforcement
- listener/forwarder for Git webhook events

Repository Server

The repository server is an internal service which maintains a local cache of the Git repository holding the application manifests. It is responsible for generating and returning the Kubernetes manifests when provided the following inputs:

- acend gmbh

- repository URL
- revision (commit, tag, branch)
- application path
- template specific settings: parameters, ksonnet environments, helm values.yaml

Application Controller

The application controller is a Kubernetes controller which continuously monitors running applications and compares the current, live state against the desired target state (as specified in the repo). It detects OutOfSync application state and optionally takes corrective action. It is responsible for invoking any user-defined hooks for lifecycle events (PreSync, Sync, PostSync)

Argo CD Core Concepts

Those core Concepts exist in Argo CD:

- **Clusters**: pre configured Kubernetes Clusters (including OpenShift)
- **Repositories** : pre configured git repositories, including repository credentials (ssh, username-password).
- **Applications** : A group of Kubernetes resources, represented in a git repository. Usually the Kubernetes resources which will be applied in a Kubernetes namespace. Represented as [CRD](#) .
- **Projects** : A logical grouping of Argo CD applications. Various restrictions can be defined on project level. Useful when multiple Teams work with the same Argo CD instance

1. Getting started

Task 1.1: Web IDE

The first thing we're going to do is to explore our lab environment and get in touch with the different components.

The namespace with the name corresponding to your username is going to be used for all the hands-on labs. And you will be using the `argocd tool` or the ArgoCD webconsole, to verify what resources and objects Argo CD created for you.

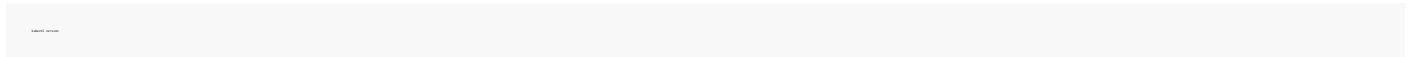
Note

You can also use your local installation of the cli tools. Make sure you completed [the setup](#) before you continue with this lab.

Note

The URL and Credentials to the Web IDE will provided by the teacher. Use Chrome for the best experience.

Once you're successfully logged into the web IDE open a new Terminal by hitting `CTRL + SHIFT + `` or clicking the Menu button -> Terminal -> new Terminal and check the installed kubectlversion by executing the following command:



The Web IDE Pod consists of the following tools:

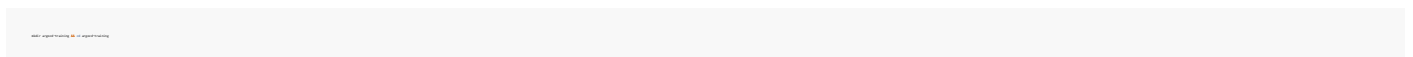
- oc
- kubectl
- kustomize
- helm
- kubectx
- kubens
- tekton cli
- odo
- argocd

The files in the home directory under `/home/project` are stored in a persistence volume, so please make sure to store all your persistence data in this directory.

Task 1.1.1: Local Workspace Directory

During the lab, you'll be using local files (eg. YAML resources) which will be applied in your lab project.

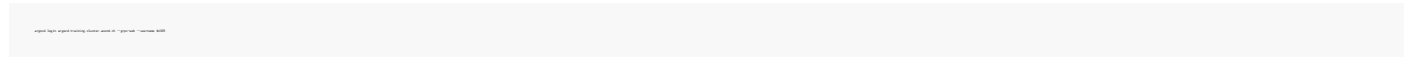
Create a new folder for your `<workspace>` in your Web IDE (for example `argocd-training` under `/home/project/argocd-training`). Either you can create it with `right-mouse-click -> New Folder` or in the Web IDE terminal



- acend gmbh

Task 1.1.2: Login on ArgoCD using argocd CLI

You can access Argo CD via Web UI (Credentials are provided by your teacher) or using the CLI. The Argo CD CLI Tool is already installed on the web IDE.

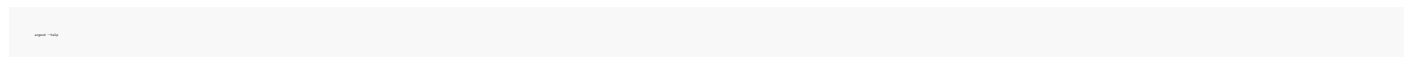


Task 1.2: Argo CD CLI

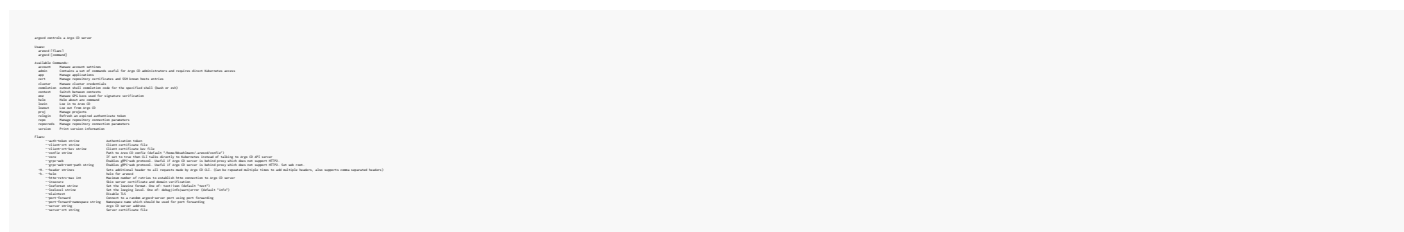
The [Argo CD CLI](#) is a powerful tool to manage Argo CD and different applications. It's a self contained binary written in Go and available for Linux, Mac OS and Windows. Thanks to the fact that the CLI is implemented in Go, it can be easily integrated into scripts and build servers for automation purposes.

Task 1.2: Getting familiar with the CLI

Print out the help of the CLI by typing

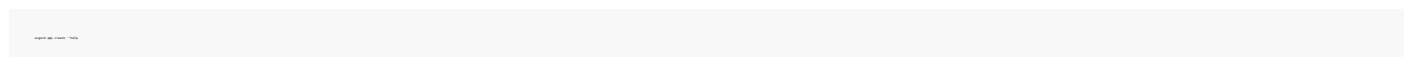


You will see a list with the available commands and flags. If you prefer to browse the manual in the browser you'll find it in the [online documentation](#) .



The `--help` flag is available for every command and subcommand of the CLI. Beside the documentation for every flag and subcommand in the current context, it prints out example command lines for the most common use cases.

Now get the help of the `app create` subcommand and find the examples and documentation of the flags.



Task 1.2: Autocompletion

Note

This step is only needed, when you're not working with the Web IDE we've provided. The autocompletion is already installed in the Web IDE

A productivity booster when working with the CLI is the autocompletion feature. It can be used for `bash` and `zsh` shells. You can enable the autocompletion for the current `bash` with the following command:

- acend gmbh

After typing `argocd` you can autocomplete the subcommands with a double tap the tabulator key. This works even for deployed artifacts on the cluster: A double tab after `argocd app get` lists all defined Argo CD applications.

To install autocompletion permanently you can add the following command in the `~/.bashrc` file.

Find further information in the [official documentation](#)

2. Simple Example

In this lab you will learn how to deploy a simple application using Argo CD.

Our lab setup consists of the following components:

- Git Server (Gitea): <https://gitea.training.cluster.acend.ch>
- Argo CD Server: <https://argocd.training.cluster.acend.ch>
- Kubernetes Cluster

Task 2.1: Login to the Gitea and Clone the Repo

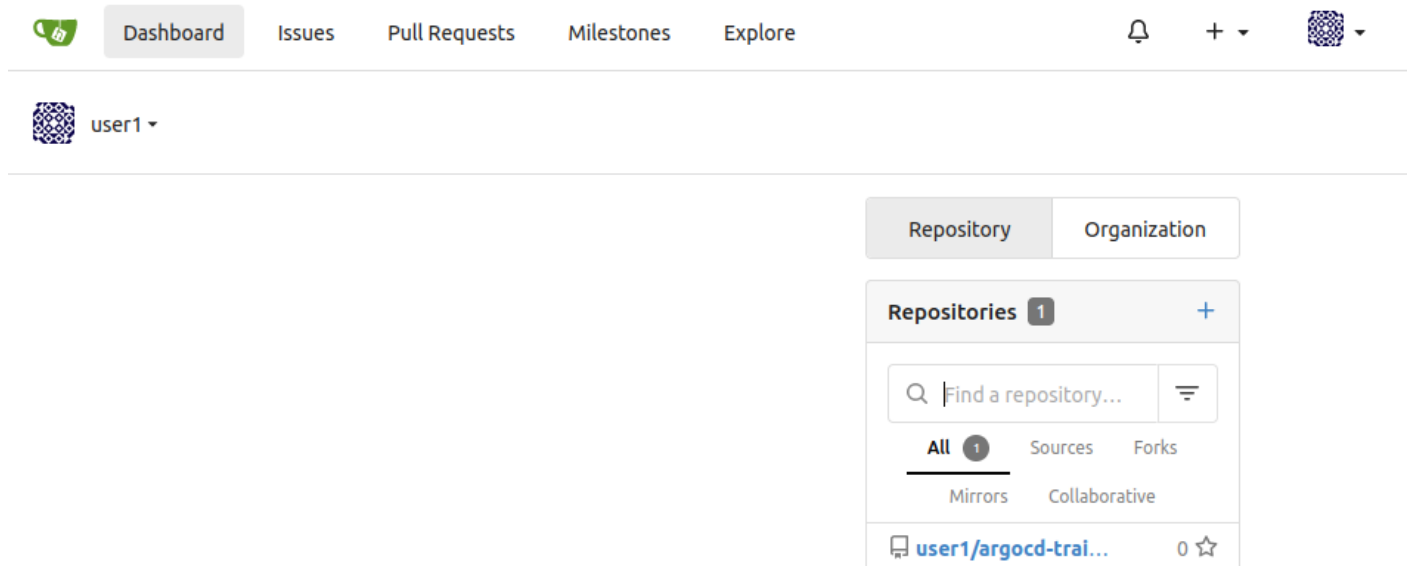
For this Training we've installed a Git Server under <https://gitea.training.cluster.acend.ch> . We also forked the Argo CD Example Repo for your `<username>` .

Open your Webbrowser and navigate to <https://gitea.training.cluster.acend.ch> . Login with the training credentials provided by the trainer (Login Button is in the upper right corner).

Note

Users which have a personal Github account can just fork the Repository [argocd-training-examples](#) to their personal account. To fork the repository click on the top right of the Github on *Fork*.

The Git Repository is available under your Repositories



By clicking on the repository link in the repository list you get to the detail page.

- acend gmbh

And we also want git to store our Password for the whole day so that we don't need to login every single time we push something.

Then use the following command to verify whether the git config for username and email were correctly added:

Task 2.2: Deploying the resources with Argo CD

Now we want to deploy the resource manifests contained in the cloned repository with Argo CD to demonstrate the basic features of Argo CD.

To deploy the resources using the Argo CD CLI use the following command:

Expected output: application 'argo-<username>' created

Note

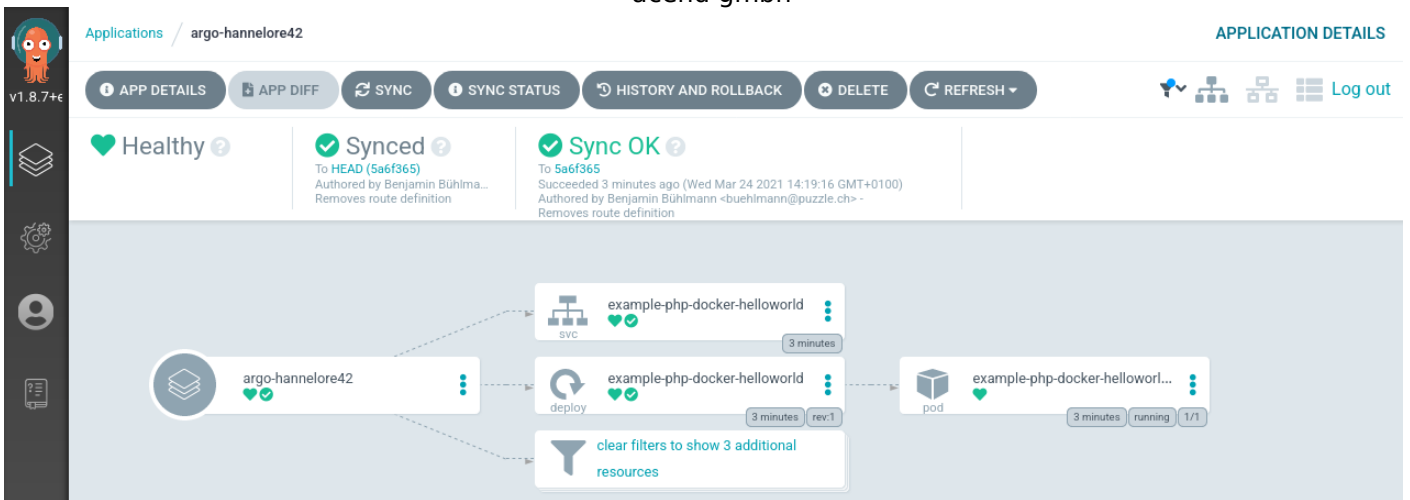
We don't need to provide Git credentials because the repository is readable for non-authenticated users as well

Note

If you want to deploy it in a different namespace, make sure the namespaces exists before synching the app

Once the application is created, you can view its status:

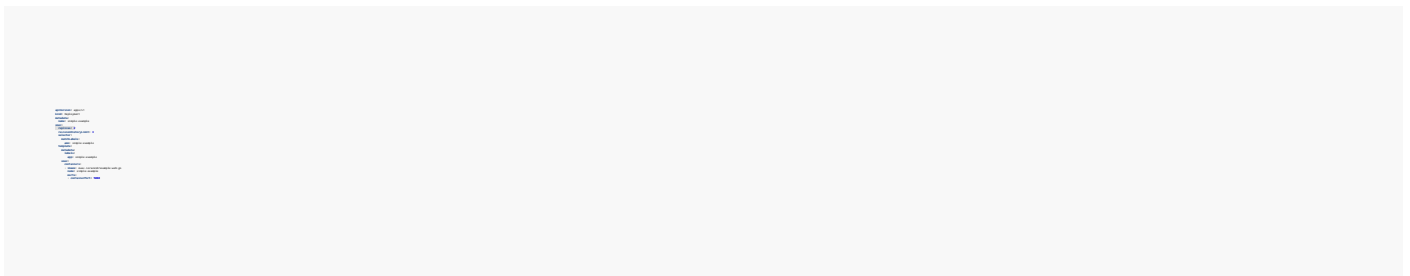
The application status is initially in OutOfSync state. To sync (deploy) the resource manifests, run:



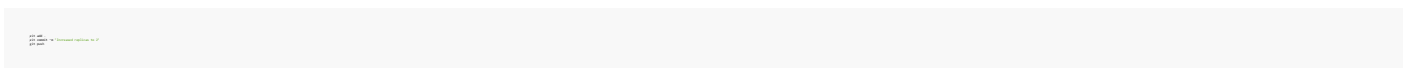
Task 2.3: Automated Sync Policy and Diff

When there is a new commit in your Git repository, the Argo CD application becomes OutOfSync. Let's assume we want to scale up our `Deployment` of the example application from 1 to 2 replicas. We will change this in the `Deployment` manifest.

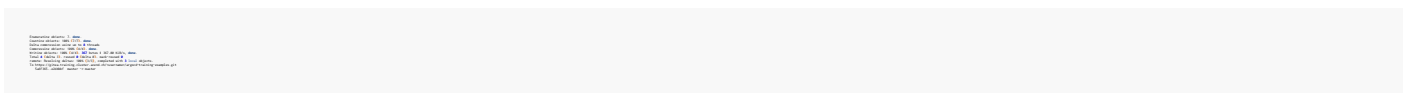
Increase the number of replicas in your file `<workspace>/example-app/deployment.yaml` to 2.



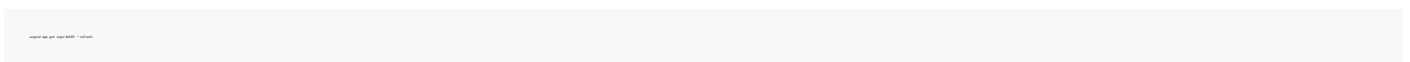
Commit the changes and push them to your personal remote Git repository. After the Git push command a **password** input field will appear at the top of the Web IDE.



After a successful push you should see the following output



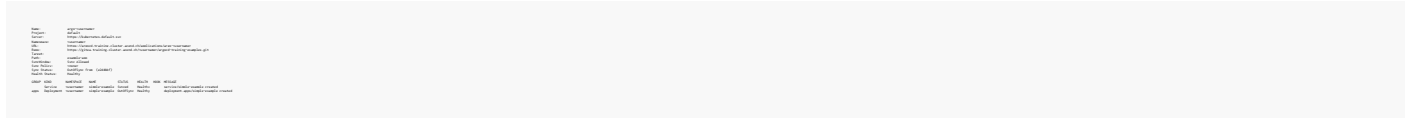
Check the state of the resources by cli:



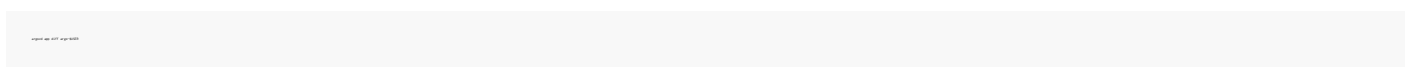
- acend gmbh

The parameter `--refresh` triggers an update against the Git repository. Out of the box Git will be polled by Argo CD in a predefined interval (defaults to 3 minutes). To use a synchronous workflow you can use webhooks in Git. These will trigger a synchronization in Argo CD on every push to the repository.

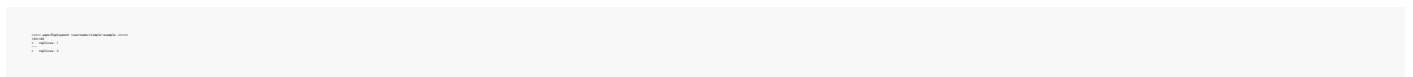
You will see that the Deployment is now OutOfSync:



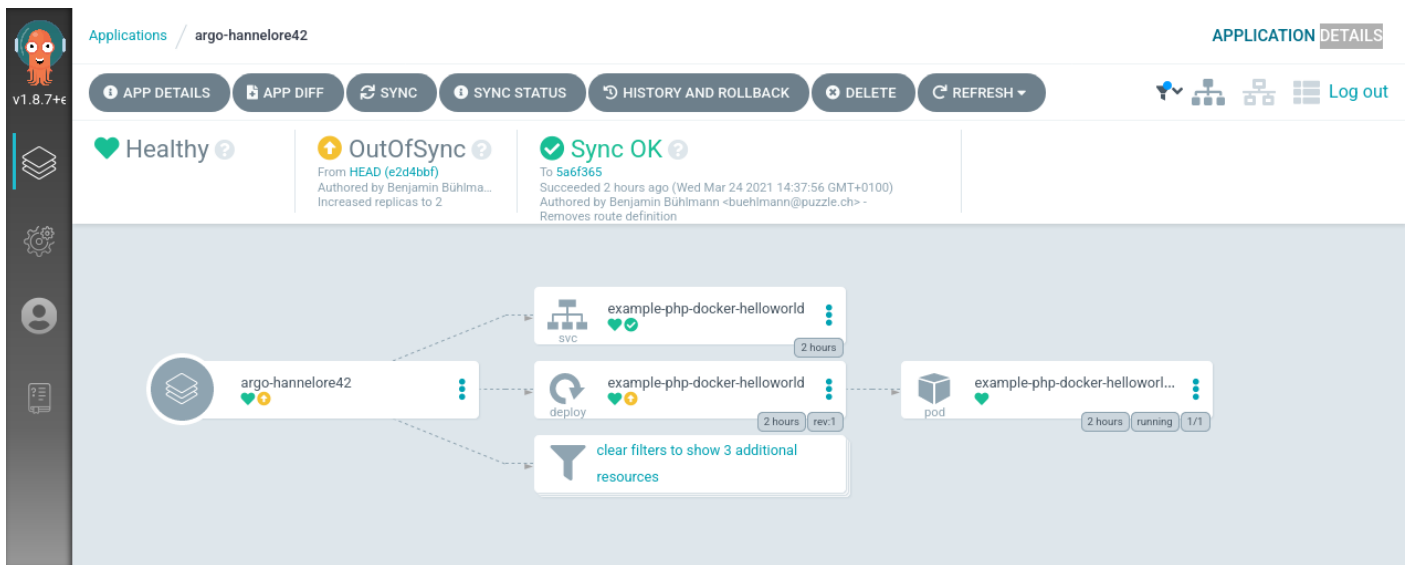
When an application is OutOfSync then your deployed 'live state' is no longer the same as the 'target state' which is represented by the resource manifests in the Git repository. You can inspect the differences between live and target state by cli:



which should give you an output similar to:



Now open the web console of Argo CD and go to your application. The deployment `simple-example` is marked as 'OutOfSync':



With a click on Deployment > Diff you will see the differences:

The screenshot shows the Argo CD interface for an application named 'example-php-docker-helloworld' in the 'hannelore42' namespace. The application is currently in an 'OutOfSync' state, indicated by a yellow warning icon, but its health is 'Healthy' (green heart). The 'SUMMARY' tab shows the following details:

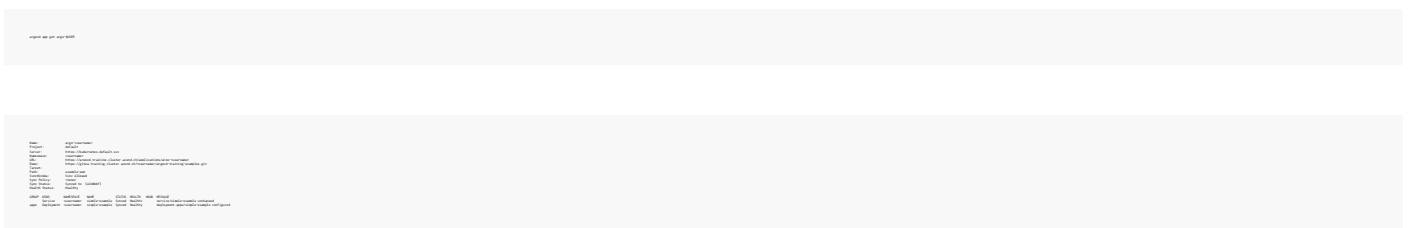
KIND	Deployment
NAME	example-php-docker-helloworld
NAMESPACE	hannelore42
CREATED_AT	03/24/2021 14:37:56
STATUS	OutOfSync
HEALTH	Healthy

Below the summary, a 'DIFF' view compares the 'LIVE MANIFEST' and 'DESIRED MANIFEST'. The 'Compact diff' option is selected. The diff shows a change in the number of replicas:

Line	Field	Value
99	spec:	
100	progressDeadlineSeconds:	600
101	replicas:	1 (Live) / 2 (Desired)
102	revisionHistoryLimit:	3
103	selector:	

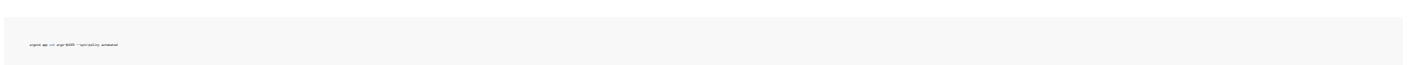
Now click `Sync` on the top left and let the magic happens ;) The application will be scaled up to 2 replicas and the resources are in Sync again.

Double-check the status by cli



Argo CD can automatically sync an application when it detects differences between the desired manifests in Git, and the live state in the cluster. A benefit of automatic sync is that CI/CD pipelines no longer need direct access to the Argo CD API server to perform the deployment. Instead, the pipeline makes a commit and push to the Git repository with the changes to the manifests in the tracking Git repo.

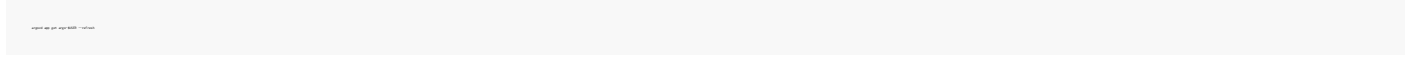
To configure automatic sync run (or use the UI):



From now on Argo CD will automatically apply all resources to Kubernetes every time you commit to the Git repository.

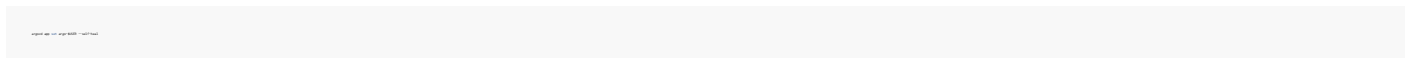
- acend gmbh

Decrease the replicas count to 1 and push the updated manifest to remote. Wait for a few moments and see check that ArgoCD will scale the deployment of the example app down to 1 replica. The default polling interval is 3 minutes. If you don't want to wait you can force a refresh by clicking `Refresh` in the UI or by cli:

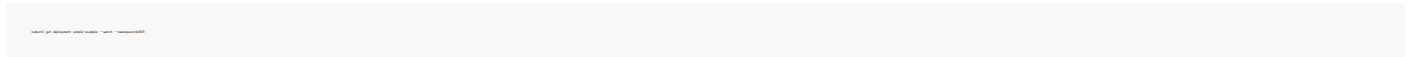


Task 2.4: Automatic Self-Healing

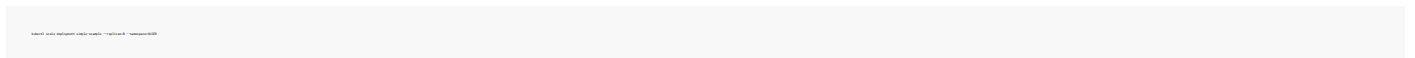
By default, changes made to the live cluster will not trigger automatic sync. To enable automatic sync when the live cluster's state deviates from the state defined in Git, run:



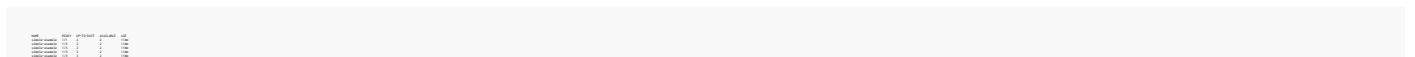
Watch the deployment `simple-example` in a separate terminal



Let's scale our `simple-example` Deployment and observe whats happening:



Argo CD will immediately scale back the `simple-example` Deployment to 1 replicas. You will see the desired replicas count in the watched Deployment.

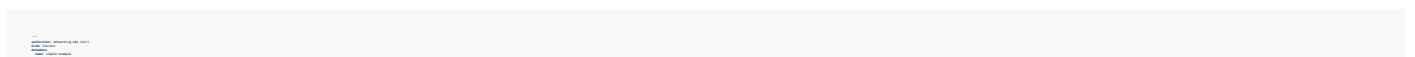


This is a great way to enforce a strict GitOps principle. Changes which are manually made on deployed resource manifests are reverted immediately back to the desired state by the ArgoCD controller.

Task 2.5: Expose Application

This is an optional task.

To expose an application we need to specify a so called `ingress` resource. Create an `ingress.yaml` file next to the `deployment.yaml` in the `example-app` directory with the following content.



Commit and Push the changes again, like you did before:

- acend gmbh

The Service was successfully deleted by Argo CD because the manifest was removed from git. See the HEALTH and MESSAGE of the previous console output.

Task 2.7: State of ArgoCD

Argo CD is largely built stateless. The configuration is persisted as native Kubernetes objects. And those are stored in Kubernetes *etcd*. There is no additional storage layer needed to run ArgoCD. The Redis storage under the hood acts just as a throw-away cache and can be evicted anytime without any data loss.

The configuration changes made on ArgoCD objects through the UI or by cli tool `argocd` are reflected in updates of the ArgoCD Kubernetes objects `Application` and `AppProject` in the `argocd` namespace.

Let's list all Kubernetes objects of type `Application` (short form: `app`)

```
argocd app list
```

```
NAME      STATUS      SYNC STATUS  HEALTH STATUS
```

You will see the application which we created some chapters ago by cli command `argocd app create...` . To see the complete configuration of the `Application` as *yam*/use:

```
argocd app get <app-name>
```

You even can edit the `Application` resource by using:

```
argocd app edit <app-name>
```

This allows us to manage the ArgoCD application definitions in a declarative way as well. It is a common pattern to have one ArgoCD application which references n child Applications which allows us a fast bootstrapping of a whole environment or a new cluster. This pattern is well known as the *App of apps* pattern.

Task 2.8: Accessing a private Git repository

The Git repository we have imported to Gitea is public available for the whole world. When accessing a private repository we have to provide credentials in form of a username/password pair or a ssh private key. In this task you will learn how to access a protected repo from Argo CD.

First make the Git repository in Gitea private by checking the option `Visibility: Make Repository Private` under `Settings -> Repository` . Now sync the app again.

```
argocd app sync <app-name>
```

You will see the following error

- acend gmbh

Argo CD can't any longer access the protected repository without providing credentials for authentication. Next assign credentials to used Git repository. You have to provide the Gitea password interactively.

Note

You can provide the password through the cli by using the flag `--password`.

Now the sync should work. Argo CD use the configured credentials to authenticate against your repository in Gitea.

You can define [credential templates](#) when using the same credential for multiple Git repositories. The configured credentials are used for each Git repository beginning with the configured URL. The following command will create a credential which matches all git repositories for your username (e.g. `https://<username>@gitea.training.cluster.acend.ch/<username>`)

Finally make your personal Git repository public again for the following labs. Uncheck the option `Visibility: Make Repository Private` under `Settings -> Repository` in the Gitea UI.

Note

TLS certificates and SSH private keys are supported alternative authentication methods by Argo CD. Proxy support can be configured as well in the repository settings.

Have a look in the [documentation](#) for detailed information about accessing private repositories.

Task 2.9: Delete the Application

You can cascading delete the ArgoCD Application with the following command:

Hit `y` to confirm the deletion and this will delete the `Application` manifests of ArgoCD and all created resources by this application. In our case the `Application`, `Deployment` and `Service` will be deleted. With the flag `--cascade=false` only the ArgoCD `Application` will be deleted and the created resources `Deployment` and `Service` remain untouched.

3. Resource Hooks

In this Lab you are going to learn about [Resource Hooks](#) .

Resource Hooks

Hooks allow to run scripts before, during and after the Argo CD **sync** operation is running. They give you more control over the sync process. They can also run when the sync operation fails for example. The concept is very similar to the concept of [Helm Hooks](#) . Argo CD supports many Helm hooks by mapping the Helm annotations onto Argo CD's own hook annotations. You can see the full mapping of the Helm hooks [in the ArgoCD documentation](#)

Some examples when hooks can be useful:

- `PreSync` hook. Upgrading a Database, Performing a migration before deploying a new version of the application.
- `PostSync` hook. Run integration, smoke and other tests after the deployment to verify its status.
- `Sync` hook. Allows to run more complex deployment strategies. e.g.: Blue-Green or Canary Deployments
- `SyncFail` hook. Clean up a failed deployment.

Hooks are annotated `argocd.argoproj.io/hook: <hook>` Kubernetes resources in the source repository, which Argo CD will apply during the sync operation.

A `PreSync` Hook to run a database migration might therefore look like this:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: migrate
  annotations:
    argocd.argoproj.io/hook: PreSync
spec:
  template:
    spec:
      containers:
      - name: migrate
        image: postgres:12
        command:
        - /bin/sh
        - -c
        - 'psql -h $PGHOST -U $PGUSER -d $PGDATABASE -c "ALTER TABLE users ADD COLUMN new_col VARCHAR(255);"'
```

It's basically a [Kubernetes Job](#) which starts a Pod that executes some sort of code.

Note

Named hooks (i.e. ones with `/metadata/name`) will only be created once. If you want a hook to be re-created each time either use `BeforeHookCreation` policy or `/metadata/generateName`.

Note

Hooks are not run during a [selective sync](#)

Hook Deletion Policies

The hook deletion policy defines when a hook should be deleted. It's also configured with an annotation `argocd.argoproj.io/hook-delete-policy` on the hook resource.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: migrate
  annotations:
    argocd.argoproj.io/hook: PreSync
    argocd.argoproj.io/hook-delete-policy: HookSucceeded,HookFailed,BeforeHookCreation
spec:
  template:
    spec:
      containers:
      - name: migrate
        image: postgres:12
        command:
        - /bin/sh
        - -c
        - 'psql -h $PGHOST -U $PGUSER -d $PGDATABASE -c "ALTER TABLE users ADD COLUMN new_col VARCHAR(255);"'
```

- `HookSucceeded` : will be deleted after the hook succeeded
- `HookFailed` : will be deleted after a hook failed
- `BeforeHookCreation` : Any hook resource will be deleted before the new one is created.

Task 3.1: Hook Example

In this task we're going to deploy an [example](#) which has `pre` and `post` hooks.

Create the new application `argo-hook-$USER` with the following command. It will create a service, a deployment and two hooks as soon as the application is synced.

- PreSync: before Job
- Sync: Deployment with name `pre-post-sync-hook`
- PostSync: after Job

```
argo create --name argo-hook-$USER --service <service> --deployment <deployment> --pre-sync-hook <pre-hook> --post-sync-hook <post-hook>
```

Sync the application

```
argo sync --application argo-hook-$USER
```

And verify the deployment:

```
argo get --application argo-hook-$USER --namespace <namespace>
```

Or in the web UI.

Task 3.2: Post-hook Curl (Optional)

Alter the post sync hook command from `sleep` to `curl https://acend.ch` (Could be used to send a notification to a Chat channel) The `curl` command is not available in the minimal `quay.io/acend/example-web-go` image. You can use `quay.io/acend/example-web-python` or different image.

Edit the hook under `pre-post-sync-hook/post-sync-job.yaml` accordingly, commit and push the changes and trigger the sync operation.

```
apiVersion: argoproj.io/v1alpha1
kind: Hook
name: post-sync-job
type: PostSync
command:
- curl
- https://acend.ch
```

Task 3.3: Delete the Application

Delete the application after you've explored the Argo CD Resources and the managed Kubernetes resources.

```
argo delete --application argo-hook-$USER
```

4. Sync Phases and Waves

In this Lab you are going to learn about [Sync Phases and Waves](#) .

Sync Phases and Waves

At a high-level, Argo CD executes the sync operation in the three phases pre-sync, sync and post-sync.

Within each phase you can have one or more waves, that allows you to ensure certain resources are healthy before subsequent resources are synced.

When Argo CD starts a sync, it orders the resources in the following precedence:

- The phase
- The wave they are in (lower values first)
- By kind (e.g. namespaces first)
- By name

It then determines the number of the next wave to apply. This is the first number where any resource is out-of-sync or unhealthy.

It applies resources in that wave.

It repeats this process until all phases and waves are in-sync and healthy.

How to specify waves and phases

Pre-sync and post-sync can only contain hooks defined on annotations `argocd.argoproj.io/hook: PreSync` .

You can specify the wave in the sync phase by setting an annotation `argocd.argoproj.io/sync-wave` . Hooks and resources are assigned to wave zero by default. The wave can be negative, so you can create a wave that runs before all other resources.

Task 4.1: Sync Wave Example

Let's now get our hands on a sync wave example.

Create the new application `argo-wave-$USER` with the following command. The Application consist of the following resources, phases and waves:

- PreSync
 - Job: upgrade-sql-schema
- Sync Wave 0
 - Deployment: backend
 - Service: backend
- Sync Wave 1
 - Job: maintenance-page-up
- Sync Wave 2
 - Deployment: frontend
 - Service: frontend
- Sync Wave 3
 - Job: maintenance-page-down

- acend gmbh

Sync the application:

And verify the deployment:

Task 4.2: Delete the Application

Delete the application after you've explored the Argo CD Resources and the managed Kubernetes resources.

5. Tools

In this Lab you are going to learn about different [application source tools](#) .

Tools

As mentioned in the [introduction](#) Argo CD supports many different formats in which the Kubernetes manifests can be defined:

- [kustomize](#) applications
- [helm](#) charts
- [ksonnet](#) applications (deprecated)
- [jsonnet](#) files
- Plain directory of YAML/json manifests
- Any custom config management tool configured as a config management plugin

So far you have been using **plain YAML** manifest in the previous labs.

Warning

Argo CD provides a mechanism to override the parameters of Argo CD applications. [The Argo CD parameter overrides](#) feature is provided mainly as a convenience to developers and is intended to be used in dev/test environments, vs. production environments.

Many consider this feature as anti-pattern to GitOps. So only use this feature when no other option is available!

Tool Detection

When the build tool is not specified explicitly in the [Argo CD Application](#) CRD it will be detected:

- Helm if there's a file matching `Chart.yaml` .
- Kustomize if there's a `kustomization.yaml` , `kustomization.yml` , OR `Kustomization`
- jsonnet if there's a `*.jsonnet` file.

You are now going to deploy an application in the different formats.

You can also find additional examples [here](#) .

5.1 Helm

This lab explains how to use [Helm](#) as manifest format together with Argo CD.

Helm Introduction

[Helm](#) is a [Cloud Native Foundation](#) project to define, install and manage applications in Kubernetes.

It can be used to package multiple Kubernetes resources into a single logical deployment unit.

Helm Charts are configured using `values.yaml` files. (e.g. images, image tags, hostnames, ...).

When using `helm` charts together with Argo CD we can specify the `values.yaml` like this:

- acend gmbh

The `--values` flag can be repeated to support multiple values files.

Info

Values files must be in the same git repository as the Helm chart. The files can be in a different location in which case it can be accessed using a relative path relative to the root directory of the Helm chart.

Helm Parameters

Similar to when using `helm` directly (`helm install <release> --set replicaCount=2 ./mychart --namespace <namespace>`), you are able to overwrite values from the `values.yaml`, by setting parameters.

Warning

Argo CD provides a mechanism to override the parameters of Argo CD applications. [The Argo CD parameter overrides](#) feature is provided mainly as a convenience to developers and is intended to be used in dev/test environments, vs. production environments.

Many consider this feature as anti-pattern to GitOps. So only use this feature when no other option is available!

Helm Release Name

By default, the Helm release name is equal to the Application name to which it belongs. Sometimes, especially on a centralised ArgoCD, you may want to override that name, and it is possible with the `release-name` flag on the cli:

Warning

Please note that overriding the Helm release name might cause problems when the chart you are deploying is using the `app.kubernetes.io/instance` label. ArgoCD injects this label with the value of the Application name for tracking purposes.

Helm Hooks

[Helm hooks](#) are similar to the Argo CD Hooks from [lab 4](#).

Further Docs

Read more about the helm integration in the [official documentation](#)

Task 5.1.1: Deploy the simple-example as Helm Chart

- acend gmbh

Let's deploy the simple-example from lab 1 using a [helm chart](#) .

First you'll have to create a new Argo CD application.

```
argocd app create simple-example --repo https://github.com/acend/helm-charts --path helm-chart --dest-namespace default
```

Sync the application

To sync (deploy) the resources you can simply click sync in the web UI or execute the following command:

```
argocd app sync simple-example
```

And verify the deployment:

```
argocd app get simple-example
```

Tell the application to sync automatically, to enable self-healing and auto-prune

```
argocd app set simple-example --sync-policy=Automated --prune
```

Task 5.1.2: Scale the deployment to 2 replicas

We can set the `helm` parameter with the following command:

```
argocd app set simple-example --helm-set=replicas=2
```

Warning

Only use this way of setting params in dev and test stages. Not for Production!

Since the `sync-policy` is set to `automated` the second pod will be deployed immediately.

Task 5.1.3: Ingress

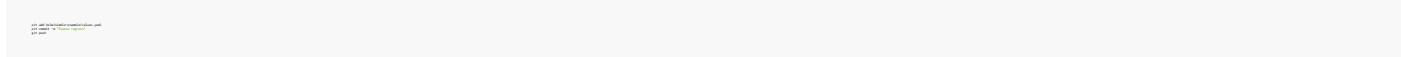
The proper and production ready way of overwriting values is by doing it in git.

Change the `helm/simple-example/values.yaml` file in your git repository

```
helm chart lint simple-example --strict
```

Commit and push the changes to your repository.

- acend gmbh



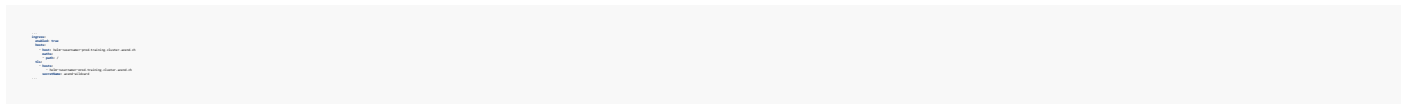
Open your Browser and verify whether you can access the application.

Task 5.1.4: Create a second application representing the production stage

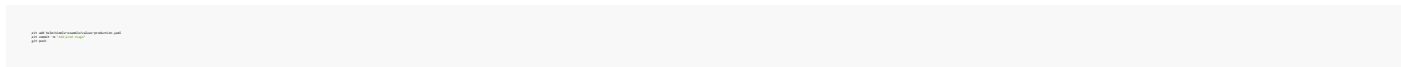
Let's now also deploy an application for the production stage.

Create a new `values.yaml` file for the production stage: `helm/simple-example/values-production.yaml` And copy the content from the default `helm/simple-example/values.yaml` file.

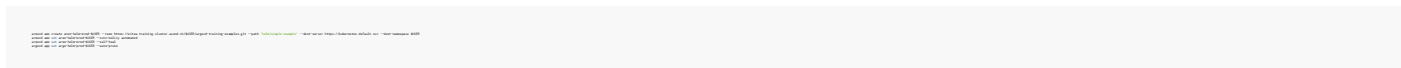
Change the host in the `helm/simple-example/values-production.yaml` to the production url



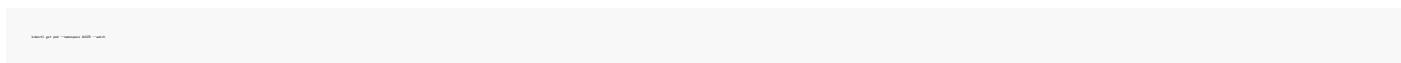
Commit and push the changes to your repository.



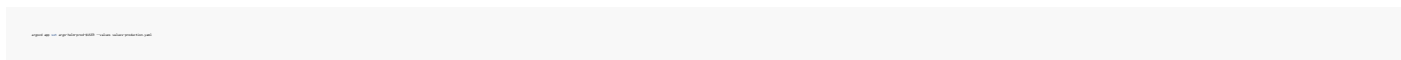
Let's create the production stage Argo CD application with the name `argo-helm-prod-$USER` and enable automated sync, self-healing and pruning.



And verify the deployment:



Tell the Argo CD app to use the `values-production.yaml` values file



Change for example the ingress hostname to something different in the `values-production.yaml` and verify whether you can access the new hostname.

Task 5.1.4: Delete the Applications

Delete the applications after you've explored the Argo CD Resources and the managed Kubernetes resources.

- acend gmbh

© 2019 acend gmbh

5.2 Kustomize

This lab explains how to use [kustomize](#) as manifest format together with Argo CD.

Kustomize Introduction

[Kustomize](#) introduces a template-free way to customize application configuration that simplifies the use of off-the-shelf applications. It is built into `kubectl` and `oc` with the command `kubectl apply -k` or `oc apply -k`.

It uses a concept called overlays, which allows to reduce redundant configuration for multiple stages (e.g. dev, prod, test) without a use of a template language.

Argo CD supports kustomize manifests out of the box.

Kustomize Overlays

When you want to use Kustomize with an overlay, you have to point the Argo Application to the Overlay

Kustomize Configuration

The following configuration options are available for Kustomize:

- `namePrefix` is a prefix appended to resources for Kustomize apps
- `nameSuffix` is a suffix appended to resources for Kustomize apps
- `images` is a list of Kustomize image overrides
- `commonLabels` is a string map of an additional labels
- `commonAnnotations` is a string map of an additional annotations

Use the following command to set those parameters:

```
...
```

Further Docs

Read more about the kustomize integration in the [official documentation](#)

Task 5.2.1: Deploy the simple-example with kustomize

Let's deploy the simple-example from lab 1 using [kustomize](#).

First you'll have to create a new Argo CD application.

```
...
```

Sync the application

To sync (deploy) the resources you can simply click sync in the web UI or execute the following command:

- acend gmbh

And verify the deployment:

Tell the application to sync automatically, to enable self-healing and auto-prune

Task 5.2.2: Set a configuration parameter

We can set the `kustomize` configuration parameter with the following command:

And take a look at the application in the web UI or using the command line tool

Warning

Only use this way of setting params in dev and test stages. Not for Production!

Task 5.2.3: Create a second application representing the production stage

Let's now also deploy an application for the production stage.

This does mean we deploy an overlay which specifically configures the production stage.

Let's create the production stage Argo CD application (path: `kustomize/overlays-example/overlays/production`) with the name `argo-kustomize-prod-$USER` and enable automated sync, self-healing and pruning.

And verify the deployment:

- acend gmbh

Task 5.2.4: Delete the Applications

Delete the applications after you've explored the Argo CD Resources and the managed Kubernetes resources.

© 2020 acend gmbh

5.3 Jsonnet (Optional)

This lab explains how to use [jsonnet](#) as manifest format together with Argo CD.

Jsonnet

[Jjsonnet](#) is a templating language which adds the possibility to programmatically work with the underlying data. It basically is a simple extension of [JSON](#).

Let's have a look at an example first. The following jsonnet file

```
1 {
2   name: "example",
3   namespace: "default",
4   labels: {
5     app: "example",
6   },
7   service: {
8     type: "ClusterIP",
9     ports: [
10      {
11        port: 80,
12        targetPort: 8080,
13      },
14    ],
15  },
16  deployment: {
17    replicas: 1,
18    selector: {
19      matchLabels: {
20        app: "example",
21      },
22    },
23    template: {
24      metadata: {
25        labels: {
26          app: "example",
27        },
28      },
29      spec: {
30        containers: [
31          {
32            name: "example",
33            image: "example/example",
34            ports: [
35              {
36                containerPort: 8080,
37              },
38            ],
39          },
40        ],
41      },
42    },
43  },
44 }
```

will render into:

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: example
5   namespace: default
6   labels:
7     app: example
8 spec:
9   type: ClusterIP
10  ports:
11  - port: 80
12    targetPort: 8080
13 ---
14 apiVersion: apps/v1
15 kind: Deployment
16 metadata:
17   name: example
18   namespace: default
19   labels:
20     app: example
21 spec:
22   replicas: 1
23   selector:
24     matchLabels:
25       app: example
26   template:
27     metadata:
28       labels:
29         app: example
30     spec:
31       containers:
32       - name: example
33         image: example/example
34         ports:
35         - containerPort: 8080
```

Among many other features, Jjsonnet can help to reduce duplications.

Further Docs

Read more about the jsonnet integration in the [official documentation](#)

Task 5.3.1: Deploy the simple-example with jsonnet

Let's first explore the files in your local repository under `jsonnet`.

Similar to `helm`, `jsonnet` allows us to extract parameters into a separate file. In our example we extracted all values to the `params.libsonnet` file:

```
1 {
2   name: "example",
3   namespace: "default",
4   labels: {
5     app: "example",
6   },
7   service: {
8     type: "ClusterIP",
9     ports: [
10      {
11        port: 80,
12        targetPort: 8080,
13      },
14    ],
15  },
16  deployment: {
17    replicas: 1,
18    selector: {
19      matchLabels: {
20        app: "example",
21      },
22    },
23    template: {
24      metadata: {
25        labels: {
26          app: "example",
27        },
28      },
29      spec: {
30        containers: [
31          {
32            name: "example",
33            image: "example/example",
34            ports: [
35              {
36                containerPort: 8080,
37              },
38            ],
39          },
40        ],
41      },
42    },
43  },
44 }
```

And the actual template file, containing the kubernetes service and deployment definitions, `simple-application.jsonnet` file:

- acend gmbh

```
1 #!/usr/bin/env bash
2 # This script is used to create a new application in Argo CD.
3 # It takes a repository URL and a path to a jsonnet file as input.
4 # The script will create a new application in Argo CD and sync it.
5 # Usage: ./create-application.sh <repository> <path-to-jsonnet-file>
6
7 # Get the repository URL and path to the jsonnet file
8 REPO_URL=$1
9 JSONNET_PATH=$2
10
11 # Create a new application in Argo CD
12 kubectl apply -f <path-to-application-definition>
13
14 # Sync the application
15 kubectl apply -f <path-to-application-definition>
16
17 # Done
```

Note the first line, where we tell jsonnet where to get params from.

Ok now replace the `<username>` placeholder in the `params.libsonnet` file with your username.

Commit and push the changes to your repository.

```
1 #!/usr/bin/env bash
2 # This script is used to create a new application in Argo CD.
3 # It takes a repository URL and a path to a jsonnet file as input.
4 # The script will create a new application in Argo CD and sync it.
5 # Usage: ./create-application.sh <repository> <path-to-jsonnet-file>
6
7 # Get the repository URL and path to the jsonnet file
8 REPO_URL=$1
9 JSONNET_PATH=$2
10
11 # Create a new application in Argo CD
12 kubectl apply -f <path-to-application-definition>
13
14 # Sync the application
15 kubectl apply -f <path-to-application-definition>
16
17 # Done
```

Create the new Argo CD application.

```
1 #!/usr/bin/env bash
2 # This script is used to create a new application in Argo CD.
3 # It takes a repository URL and a path to a jsonnet file as input.
4 # The script will create a new application in Argo CD and sync it.
5 # Usage: ./create-application.sh <repository> <path-to-jsonnet-file>
6
7 # Get the repository URL and path to the jsonnet file
8 REPO_URL=$1
9 JSONNET_PATH=$2
10
11 # Create a new application in Argo CD
12 kubectl apply -f <path-to-application-definition>
13
14 # Sync the application
15 kubectl apply -f <path-to-application-definition>
16
17 # Done
```

Sync the application

To sync (deploy) the resources you can simply click sync in the web UI or execute the following command:

```
1 #!/usr/bin/env bash
2 # This script is used to create a new application in Argo CD.
3 # It takes a repository URL and a path to a jsonnet file as input.
4 # The script will create a new application in Argo CD and sync it.
5 # Usage: ./create-application.sh <repository> <path-to-jsonnet-file>
6
7 # Get the repository URL and path to the jsonnet file
8 REPO_URL=$1
9 JSONNET_PATH=$2
10
11 # Create a new application in Argo CD
12 kubectl apply -f <path-to-application-definition>
13
14 # Sync the application
15 kubectl apply -f <path-to-application-definition>
16
17 # Done
```

And verify whether your jsonnet Application definition has been successfully synced.

Task 5.3.2: Autosync and scale up

Tell the application to sync automatically, to enable self-healing and auto-prune

```
1 #!/usr/bin/env bash
2 # This script is used to create a new application in Argo CD.
3 # It takes a repository URL and a path to a jsonnet file as input.
4 # The script will create a new application in Argo CD and sync it.
5 # Usage: ./create-application.sh <repository> <path-to-jsonnet-file>
6
7 # Get the repository URL and path to the jsonnet file
8 REPO_URL=$1
9 JSONNET_PATH=$2
10
11 # Create a new application in Argo CD
12 kubectl apply -f <path-to-application-definition>
13
14 # Sync the application
15 kubectl apply -f <path-to-application-definition>
16
17 # Done
```

Now let's change the replicacount of the deployment and scale to 2 pods.

Change the replica param in your `params.libsonnet` to 2

```
1 #!/usr/bin/env bash
2 # This script is used to create a new application in Argo CD.
3 # It takes a repository URL and a path to a jsonnet file as input.
4 # The script will create a new application in Argo CD and sync it.
5 # Usage: ./create-application.sh <repository> <path-to-jsonnet-file>
6
7 # Get the repository URL and path to the jsonnet file
8 REPO_URL=$1
9 JSONNET_PATH=$2
10
11 # Create a new application in Argo CD
12 kubectl apply -f <path-to-application-definition>
13
14 # Sync the application
15 kubectl apply -f <path-to-application-definition>
16
17 # Done
```

Commit and push the changes to your repository.

- acend gmbh



And verify the result in the ArgoCD Ui or by using the following command, this might take a little while to happen, depending on how many trainees are currently working on the labs. Hint: hit refresh to speed up the process.



Task 5.3.4: Delete the Applications

Delete the applications after you've explored the Argo CD Resources and the managed Kubernetes resources.



6. Multiple Applications

When it comes to managing a larger amount of application or bootstrapping whole environments, it's not very practical to manage the ArgoCD application manually using the CLI Tool `argocd app create`.

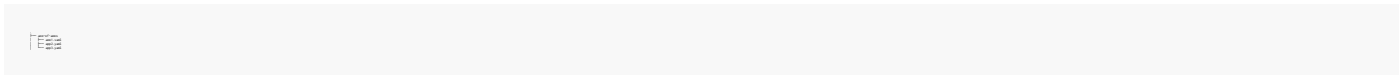
In ArgoCD there are two approaches which allow us to manage a set of applications. In the following two sub chapters you'll learn how the App of Apps and ApplicationSet pattern work.

6.1 App of Apps

The [App of apps](#) pattern is a declarative specification of one ArgoCD app that consists only of **other ArgoCD applications**. This way we have the possibility to deploy multiple apps within just one single App definition.

In Lab 2.7 you learnt how ArgoCD stores its state as Application Custom Resources. The basic idea behind the App of Apps pattern therefore is to store those Application Custom Resources within an Argo CD Application.

First let us examine our ArgoCD example repository with the child applications.

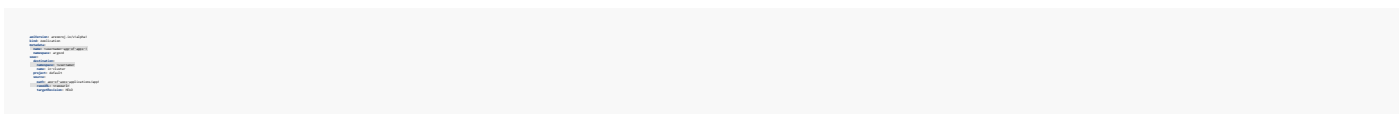


As we can see the directory consists of three ArgoCD applications. Each of them has its own source repository pointing to the corresponding repository containing a kubernetes deployment yaml file.

Task 6.1.1: Specify the Application Resources

To deploy the app of apps into our namespace we need to edit the three application custom Resources (`app-of-apps/apps/*`):

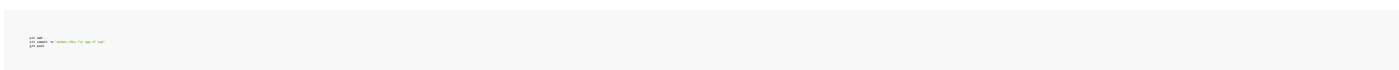
- Replace all occurrences `<username>` in the **three** yaml files.
- Set the correct `<repourl>` eg. (`https://gitea.training.cluster.acend.ch/<username>/argocd-training-examples.git`)



Make sure to also commit and push your changes to the git repository.

Note

Please make sure, to update all three application files



Task 6.1.2: Create Argo CD Application

- acend gmbh

Now let us create the parent Application which deploys our child applications as Custom Resources. Note the three parameters

- `--sync-policy automated` Set the sync policy to automated. This ensures that our child applications will be created and synced per default
- `--self-heal` Enable self heal and ensure that the parent application reconciles the child application
- `--auto-prune` Ensure that if the parent application gets deleted, it also deletes their child applications

```
-----
```

Expected output: application 'argo-aoa-<username>' created

Explore the Argo parent application in the web UI.

As you can see our newly created parent app consists of another three apps.

Note that the child application resources are not synced automatically. This is because an ArgoCD application only syncs their direct child resources. To sync the child apps, either click on sync in the ArgoCD UI or set the sync policy to automated.

Task 6.1.3: Delete the Application

Delete the application after you've explored the Argo CD Resources and the managed Kubernetes resources.

```
-----
```

6.2 Application Sets

With the ApplicationSet ArgoCD adds support for managing ArgoCD Application across a large number of clusters and environments. Plus it adds the capability of managing multitenant Kubernetes clusters.

ApplicationSets are defined through Custom Resource Definition and are processed by the ApplicationSet controller.

The ApplicationSet provides following features

- The ability to use a single Kubernetes manifest to target multiple Kubernetes clusters with Argo CD
- The ability to use a single Kubernetes manifest to deploy multiple applications from one or multiple Git repositories with Argo CD
- Improved support for monorepos: in the context of Argo CD, a monorepo is multiple Argo CD Application resources defined within a single Git repository
- Within multitenant clusters, improves the ability of individual cluster tenants to deploy applications using Argo CD (without needing to involve privileged cluster administrators in enabling the destination clusters/namespaces)

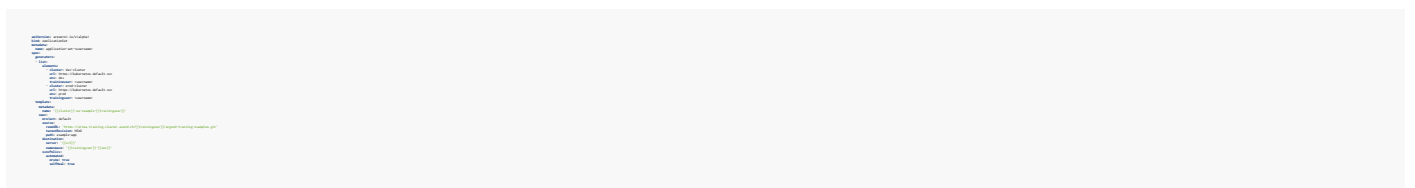
A list of parameters, which come from so called [generators](#) , render the ArgoCD Application Template to create a list of Applications.

The ApplicationSet resources work in a similar way as Helm templates do. You can define a set of placeholders `{{placeholder}}` which then are replaced with the actual value during the processing of the ApplicationSet.

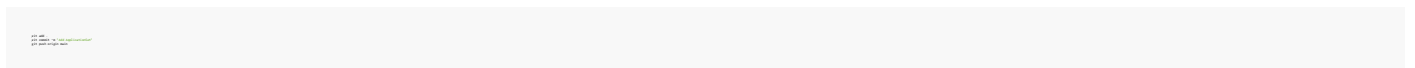
Task 6.2.1: Create an ApplicationSet

First delete the Ingress resource under `~/argocd-training-examples/example-app/ingress.yaml`

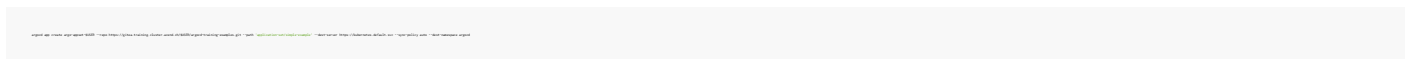
For better understanding we create our first ApplicationSet. Create a yaml file with the following content under `~/argocd-training-examples/application-set/simple-example/application-set.yaml` and replace the `<username>` placeholder with your actual username.



Now let's make sure to apply this to the cluster. But wait, we can either directly apply the yaml or we can create an ArgoCD Application just containing the ApplicationSet. Let's go the GitOps path:



And now create the ArgoCD Application, which references the ApplicationSet definition:



Note

Please notice the `dest-namespace`, `ApplicationSets` needs to be deployed within the `argocd` namespace

You should now be able to see three ArgoCD Applications postfixed with your `<username>` :

- `argo-appset-<username>` The application containing your ApplicationSet.
- `dev-cluster-as-example-<username>` ArgoCD Application for the first set of key value pairs: `dev`
- `prod-cluster-as-example-<username>` ArgoCD Application for the second set of key value pairs: `prod`

Generators

The generators (`generators` spec in the ApplicationSet yml) are the building block on how to specify the list of parameters that will be used to generate the Applications. There are several built in generators. Check out the [official documentation](#) for more information.

You have even the possibility to combine multiple generators together using the Matrix generator.

Matrix generator

The Matrix generator combines the parameters generated by two child generators, iterating through every combination of each generator's generated parameters.

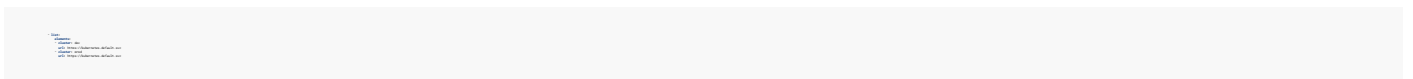
- **SCM Provider Generator + Cluster Generator:** Scanning the repositories of a GitHub organization for application resources, and targeting those resources to all available clusters.
- **Git File Generator + List Generator:** Providing a list of applications to deploy via configuration files, with optional configuration options, and deploying them to a fixed list of clusters.
- **Git Directory Generator + Cluster Decision Resource Generator:** Locate application resources contained within folders of a Git repository, and deploy them to a list of clusters provided via an external custom resource.
- And so on...

Task 6.2.2: Matrix Example ApplicationSet

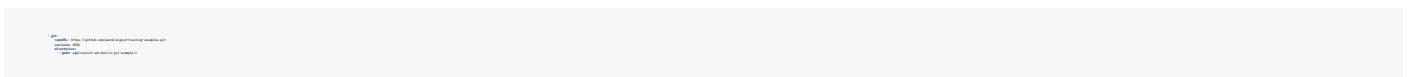
In this lab section we're going to create an ApplicationSet for an multi-environment.

- Multiple Clusters
- Multiple Applications out of a git directory

Since we don't have multiple clusters configured in our ArgoCD Cluster, we're going to use the list generator instead of the cluster generator, with two entries `dev` and `prod` both pointing to the local cluster at <https://kubernetes.default.svc> . The list generator generating values for two clusters `dev` and `prod` looks like this:



The git generator which for the Applications will therefore look like this:



Both generators generate two sets of parameters

- acend gmbh

cluster	url	path	path.basename
dev	https://kubernetes.default.svc	application-set/matrix-git-example/application1	application1
dev	https://kubernetes.default.svc	application-set/matrix-git-example/application2	application2
prod	https://kubernetes.default.svc	application-set/matrix-git-example/application1	application1
prod	https://kubernetes.default.svc	application-set/matrix-git-example/application2	application2

Next let's put everything together, create the Application Set `~/argocd-training-examples/application-set/matrix-example/matrix-example-application-set.yaml` and add the following content:

```
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: matrix-example-application-set
spec:
  generators:
  - clusterName: dev
    namespace: matrix-example
    template:
      metadata:
        name: application1
      spec:
        applicationName: application1
        destination:
          namespace: matrix-example
          server: https://kubernetes.default.svc
        project: matrix-example
  - clusterName: dev
    namespace: matrix-example
    template:
      metadata:
        name: application2
      spec:
        applicationName: application2
        destination:
          namespace: matrix-example
          server: https://kubernetes.default.svc
        project: matrix-example
  - clusterName: prod
    namespace: matrix-example
    template:
      metadata:
        name: application1
      spec:
        applicationName: application1
        destination:
          namespace: matrix-example
          server: https://kubernetes.default.svc
        project: matrix-example
  - clusterName: prod
    namespace: matrix-example
    template:
      metadata:
        name: application2
      spec:
        applicationName: application2
        destination:
          namespace: matrix-example
          server: https://kubernetes.default.svc
        project: matrix-example
```

Make sure to replace all `<username>` occurrences with your username.

Push the changes to your git repository.

```
git commit -m "Add ApplicationSet"
git push
```

And let's create an ArgoCD Application containing the Matrix ApplicationSet with the following command:

```
argocd app create matrix-example --repo https://github.com:acend/matrix-example --path matrix-example --project matrix-example
```

Next check the ArgoCD web ui, you should see the 4 generated ArgoCD applications together with the ArgoCD Application, which contains the ApplicationSet itself.

Task 6.2.3: Delete the Application

Delete the two applications (`argo-appset-$USER` and `argo-appset-matrix-$USER`) after you've explored the Argo CD Resources and the managed Kubernetes resources.

```
argocd app delete argo-appset-$USER
argocd app delete argo-appset-matrix-$USER
```


- acend gmbh

The same issue would happen because of the missing source repository expression. We will use the wildcard "*" to allow all source repositories.

Now print out the details of the project again

... and you will see the permitted source repository and destination cluster/namespace:

Now you should be able to create a new application linked with the project

Now sync the application manifest

Note

The feature of limiting source repositories and destination clusters/namespaces is a powerful construct of Argo CD as roles and policies can be assigned to projects. With this tool you can enforce a fine grained permission model to control the access of the users to the different clusters and namespaces.

Task 7.3: Deny resources by kind

On a project there is the possibility to restrict the kind of resources that can be synchronized. The restrictions are defined by whitelisting for cluster scoped resources and blacklisted for namespace scoped resources.

Let's extend our existing project and deny the synchronization of `Services`.

Now sync the application

- acend gmbh

The sync operation will fail with the following error

```
error: Service is not permitted to project project namespace.
```

Remove the kind `Service` from the deny list by using `allow-namespace-resource`

```
apiVersion: rbac.authorization.k8s.io/v1
```

... and sync the app again

```
apiVersion: rbac.authorization.k8s.io/v1
```

Task 7.4: Cleanup

Delete the resources created in this chapter by running the following commands:

```
apiVersion: rbac.authorization.k8s.io/v1
```

8. Controlling Deployment Workflows

In this chapter you'll learn how to control the Deployment Workflow even more.

8.1 Tracking and Deployment Strategies

If you are using ArgoCD with Git or Helm tools, ArgoCD gives you the availability to configure different tracking and deployment strategies.

Helm

For Helm you can use Semver versions either to pin to a specific version or to track minor / patch changes.

Use Case	How	Examples
Pin to a version (e.g. in production)	Use the version number	1.2.0
Track patches (e.g. in pre-production)	Use a range	1.2.* or $\geq 1.2.0 < 1.3.0$
Track minor releases (e.g. in QA)	Use a range	1.* or $\geq 1.0.0 < 2.0.0$
Use the latest (e.g. in local development)	Use star range	* or $\geq 0.0.0$

Git

Use Case	How	Notes
Pin to a version (e.g. in production)	Either tag the commit with (e.g. v1.2.0) and use that tag, or using commit SHA.	See commit and version pinning.
Use the latest (e.g. in local development)	Use HEAD or master (assuming master is your master branch).	See HEAD / Branch Tracking

Head / Branch Tracking

Either a branch name or a symbolic reference (like HEAD). For branches ArgoCD will take the latest commit of this branch. This method is often used in development environment where you want to apply the latest changes.

Commit and Version Pinning

The state at the specified Git tag or commit will be applied to the cluster. Pinning can be achieved in two ways. Either you can specify the full semver Git tag (v1.2.0) or a commit SHA. Usually the Git tag offers more flexibility while the commit SHA offers more immutability. Commit pinning is generally the first choice for production environments.

Task 8.1.1: Git version pinning

In this task we're going to configure a version pinning with a Git tag. The goal of this task is to show you how to pin a version from a Git tag and therefore freeze the deployment to specific commits.

- acend gmbh

First we create a Git tag `v1.0.0` and push the tag to the repository. We want to re-create the example application and let it track the created git tag `v1.0.0`.

To create and push a Git tag execute the following command:

```
git tag v1.0.0
git push --tags
```

Re-create the simple application example:

```
git clone https://github.com:acend/example-app.git
cd example-app
git checkout v1.0.0
```

To pin the `v1.0.0` version tag on our application execute the following command:

```
git checkout v1.0.0
```

Increase the number of replicas in your file `<workspace>/example-app/deployment.yaml` to 2. After that commit and push your changes to the Git repository.

```
sed -i 's/replicas: 1/replicas: 2/' deployment.yaml
git add deployment.yaml
git commit -m 'Increase replicas to 2'
git push
```

For committing and pushing your changes to your Git repository, execute following command:

```
git add deployment.yaml
git commit -m 'Increase replicas to 2'
git push
```

Now you can try to sync your application with following command:

```
argocd app sync example-app
```

Check the number of configured replicas on the app deployment.

To see the number of configured replicas execute following command:

```
argocd app get example-app
```

You can see in the command output, the number of replicas didn't changed and remains to one.

```
argocd app sync example-app
```

Let ArgoCD pickup the latest change, for that we have to create a new git tag and tell ArgoCD to track it. Let's create a new Git tag with the patch version `v.1.0.1` and push it to the repository. Then update the revision and resync the ArgoCD app.

- acend gmbh

Execute the following command to create and push a new Git tag

```
git tag v.1.0.1
```

Execute the following command to set the revision to our new Git tag v.1.0.1 .

```
git checkout v.1.0.1
```

With the new created tag, ArgoCD is going to pick up and apply the latest changes and scales up the replica count to 2. First let us sync the changes and check if the ArgoCD App is in Sync.

```
argocd app sync
```

Then display the status with following command:

```
argocd app get
```

If the app is in sync, you can check the number of replicas of the deployment.

```
argocd app get -o json
```

Now you can see in the output that the replica count has changed to 2.

```
argocd app get -o json | jq
```

Task 8.1.2: Delete the Application

You can cascading delete the ArgoCD Application with the following command:

```
argocd app delete
```

8.2 Sync Windows

With Sync windows the user can define at which time applications can be synchronized automatically and manually by Argo CD. Allowed and forbidden time windows can be defined. Sync windows can be restricted to a subset of applications, clusters and namespaces and thus offer great flexibility.

Task 8.2.1: Create application and project

Now we want to create a new empty Argo CD project.

```
argo cd create --project my-project
```

You should see the following message after a successful sync

```
argo cd sync --project my-project
```

Task 8.2.2: Create sync windows

Per default no sync windows are pre-configured in Argo CD. That means manual and automatic sync operations are allowed all the time. Now we want to create a sync window which denies syncs during the day between 08:00 and 20:00.

```
cat <<EOF | kubectl apply -f -
apiVersion: argoproj.io/v1alpha1
kind: SyncWindow
metadata:
  name: deny-syncs
spec:
  cron: "0 8 * * *"
  duration: 12h
  deny: true
  EOF
```

List all registered sync windows for the project.

```
argo cd sync-windows --project my-project
```

..prints out

```
NAME: deny-syncs
CRON: 0 8 * * *
DURATION: 12h
DENY: true
```

The window starts at 08:00 in the morning and lasts for 12 hours and denies all sync operation for all applications.

Note

Paste the cron expression on [Crontab Guru](https://crontab.guru/) to get an explanation of it.

Now try to sync the previously created application

- acend gmbh

This manual sync request will be blocked due to the active sync window with the following output

Note

If there is an active matching allow window and an active matching deny window then syncs will be denied as deny windows override allow windows.

Task 8.2.3: Updating the sync window

Now we want to restrict the defined sync windows just for the application with name `sketchy-app`. We update the existing sync window with the new application name.

Sync the application again

.. which now works because the sync window only applies for applications with the name `sketchy-app`.

Revert the changes and use wildcard `*` again to match all applications

Task 8.2.4: Enabling manual syncs

Now enable the manual sync for the window and try again to sync manually

Which now work flawless. Automatic syncs are still forbidden and will not occur between 08:00 and 20:00.

Task 8.2.5: Housekeeping

Clean up the resources created in this lab

- acend gmbh

Find more detailed information about [Sync Windows in the docs](#) .

9. Additional Concepts

In this chapter you'll learn additional concepts of ArgoCD

9.1 Orphaned Resources

This lab contains demonstrates how to find orphaned top-level resources with Argo CD. Orphaned resources are not managed by Argo CD and could be potentially removed from cluster.

Task 9.1.1: Create application and project

```
...
```

Create new Argo CD project without restrictions for Git source repository (`-src`) nor destination cluster/namespace (`-dest`)

```
...
```

Enable visualization and monitoring of Orphaned Resources for the newly created project `apps-<username>`

```
...
```

Note

The flag `--orphaned-resources` enables the determinability of orphaned resources in Argo CD. After a refresh you will see them in the user interface on the project when selecting the checkbox *Orphaned Resources*. With the flag `--orphaned-resources-warn` enabled, for each Argo CD application with orphaned resources in the destination namespace a warning will be shown in the user interface.

Task 9.1.2: Assign application to project

Assign application to newly created project

```
...
```

Ensure that the application is now assigned to the new project `apps-<username>`

```
...
```

Refresh the application

Task 9.1.3: Create orphaned resource

Now create the orphan service `black-hole` in the same target namespace the Argo CD application has:

```
apiVersion: v1
kind: Service
metadata:
  name: black-hole
spec:
  selector: {}
  ports:
  - port: 80
```

Note

This service will be detected as orphaned resource because it is not managed by Argo CD. All resources which are managed by Argo CD are marked with the label `app.kubernetes.io/instance` per default. The key of the label can be changed with the setting `application.instanceLabelKey`. See [documentation](#) for further details.

Print all resources

```
apiVersion: v1
kind: Service
metadata:
  name: black-hole
spec:
  selector: {}
  ports:
  - port: 80
```

You see in the output that the manually created service `black-hole` is marked as orphaned:

```
NAME          TYPE        PORT   ORPHANED
black-hole    ClusterIP   80     true
```

When viewing the details of the application you will see the warning about the orphaned resource

```
apiVersion: v1
kind: Service
metadata:
  name: black-hole
spec:
  selector: {}
  ports:
  - port: 80
```

```
WARNING: The service 'black-hole' is orphaned. It is not managed by Argo CD. All resources managed by Argo CD are marked with the label 'app.kubernetes.io/instance' per default. The key of the label can be changed with the setting 'application.instanceLabelKey'. See https://argo-cd.readthedocs.io/en/stable/user-guide/orphaned-resources/ for further details.
```

Task 9.1.4: Housekeeping

Clean up the resources created in this lab

```
apiVersion: v1
kind: Service
metadata:
  name: black-hole
spec:
  selector: {}
  ports:
  - port: 80
```

Find more detailed information about [Orphaned Resources in the docs](#) .

9.2 Backup and Restore

Warning

This Lab only works on your local machine.

Backup and Restore

As ArgoCD holds the whole state in native Kubernetes objects it's quite straightforward to make a backup and restore it in case of a disaster recovery.

ArgoCD provides the utility `argocd-util` which is used by ArgoCD internally. The functions `export` and `import` can be used for backup and restore of a ArgoCD instance.

As the tool is contained inside the ArgoCD server image you just can execute it from inside the container.

First find the current version used of ArgoCD server to align the server version with the tool version:

```
argocd --help
```

Export the version (without the postfix `+eb3d1fb`) to environment:

```
export ARGOCD_VERSION=$(argocd --help | grep -o 'v[0-9].[0-9].[0-9]' | head -n 1)
```

Backup

```
argocd-util export --kubeconfig /home/argocd/.kube/config --output-dir /tmp/argocd-backup
```

Note

If you should encounter permission errors like `error loading config file \"/home/argocd/.kube/config\": open /home/argocd/.kube/config: permission denied` you should change temporarily the permission of the kube config:

```
chmod 777 /home/argocd/.kube/config
```

Don't forget to remove the read permission for others after exporting:

```
chmod 600 /home/argocd/.kube/config
```

Restore

The same utility can be used to restore a previously made backup:

```
argocd-util import --kubeconfig /home/argocd/.kube/config --output-dir /tmp/argocd-backup
```

Reference: [Disaster recovery](#)

- acend gmbh