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
- kubectl

Argo CD Command line tool

Follow the instructions on this page to install the ArgoCD tool on your local computer.

kubectl

Follow the instructions on this page to install the kubectl on your local computer.

- Puzzle ITC GmbH

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

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 CDs core components are the API Server, the Repository Server and the Application Controller

- Puzzle ITC GmbH

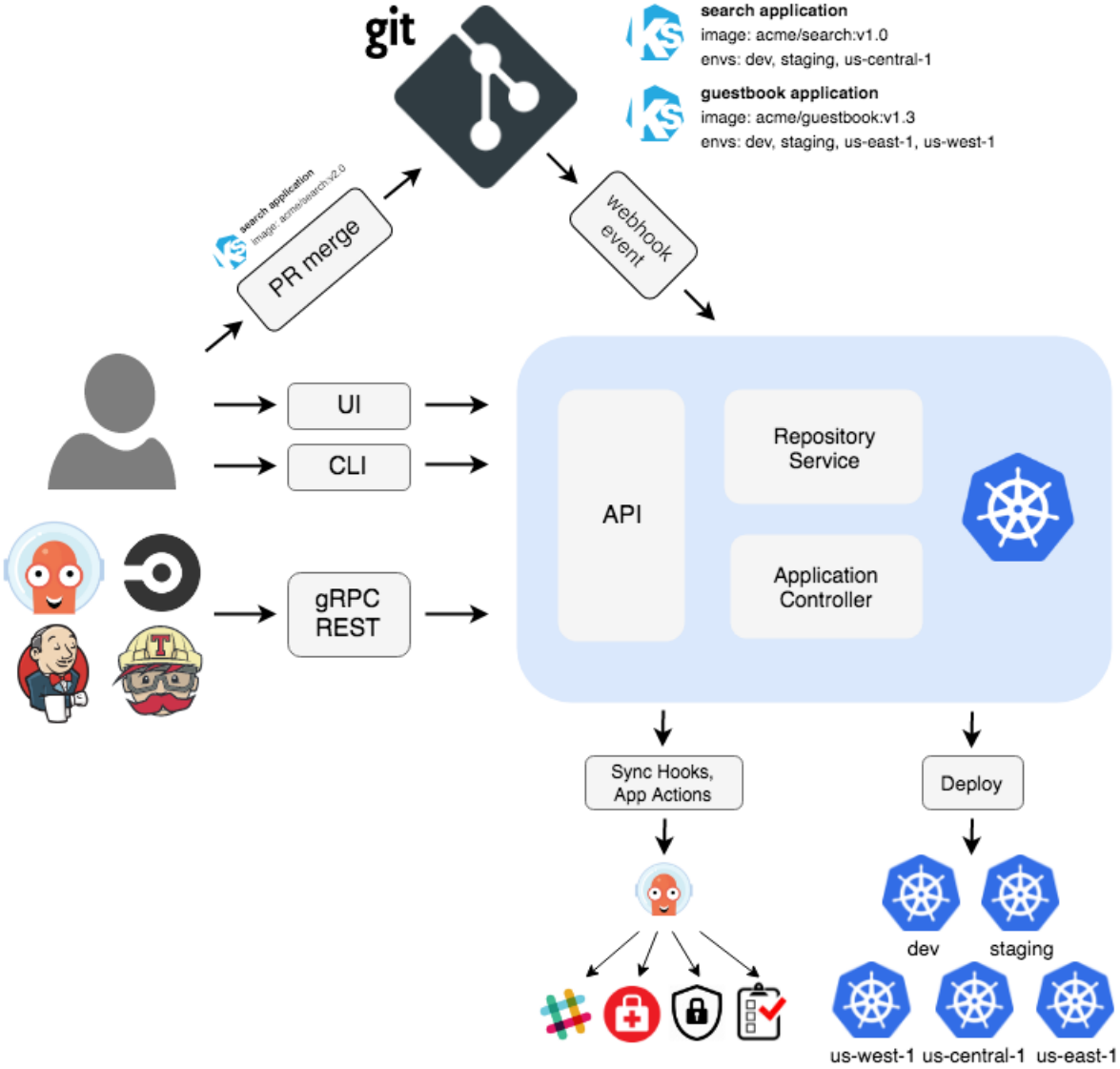


Image and component description source: https://argoproj.github.io/argo-cd/

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:

- 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:

```
kubectl version
```

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.

```
mkdir argocd-training && cd argocd-training
```

Task 1.1.2: Login to ArgoCD

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.

```
argocd login argocd.training.cluster.acend.ch --grpc-web --username $USER
```

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 1.2.1: Getting familiar with the CLI

Print out the help of the CLI by typing

```
argocd --help
```

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] .

```
argocd controls a Argo CD server

Usage:
  argocd [flags]
  argocd [command]

Available Commands:
  account     Manage account settings
  admin       Contains a set of commands useful for Argo CD administrators and requires direct Kubernetes access
  app         Manage applications
  cert        Manage repository certificates and SSH known hosts entries
  cluster     Manage cluster credentials
  completion  output shell completion code for the specified shell (bash or zsh)
  context     Switch between contexts
  gpg         Manage GPG keys used for signature verification
  help        Help about any command
  login       Log in to Argo CD
  logout      Log out from Argo CD
  proj        Manage projects
  relogin     Refresh an expired authenticate token
  repo        Manage repository connection parameters
  repocreds   Manage repository connection parameters
  version     Print version information

Flags:
      --auth-token string               Authentication token
      --client-crt string               Client certificate file
      --client-crt-key string           Client certificate key file
      --config string                   Path to Argo CD config (default "/home/bbuehlmann/.argocd/config")
      --core                            If set to true then CLI talks directly to Kubernetes instead of talking to Argo CD API server
      --grpc-web                        Enables gRPC-web protocol. Useful if Argo CD server is behind proxy which does not support HTTP2.
      --grpc-web-root-path string       Enables gRPC-web protocol. Useful if Argo CD server is behind proxy which does not support HTTP2. Set web root.
  -H, --header strings                  Sets additional header to all requests made by Argo CD CLI. (Can be repeated multiple times to add multiple headers, also supports comma separated headers)
  -h, --help                            help for argocd
      --http-retry-max int              Maximum number of retries to establish http connection to Argo CD server
      --insecure                        Skip server certificate and domain verification
      --logformat string                Set the logging format. One of: text|json (default "text")
      --loglevel string                 Set the logging level. One of: debug|info|warn|error (default "info")
      --plaintext                       Disable TLS
      --port-forward                    Connect to a random argocd-server port using port forwarding
      --port-forward-namespace string   Namespace name which should be used for port forwarding
      --server string                   Argo CD server address
      --server-crt string               Server certificate file
```

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.

```
argocd app create --help
```

Task 1.2.2: 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:

```
source <(argocd completion bash)
```

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.

```
echo "source <(argocd completion bash)" >> ~/.bashrc
source ~/.bashrc
```

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

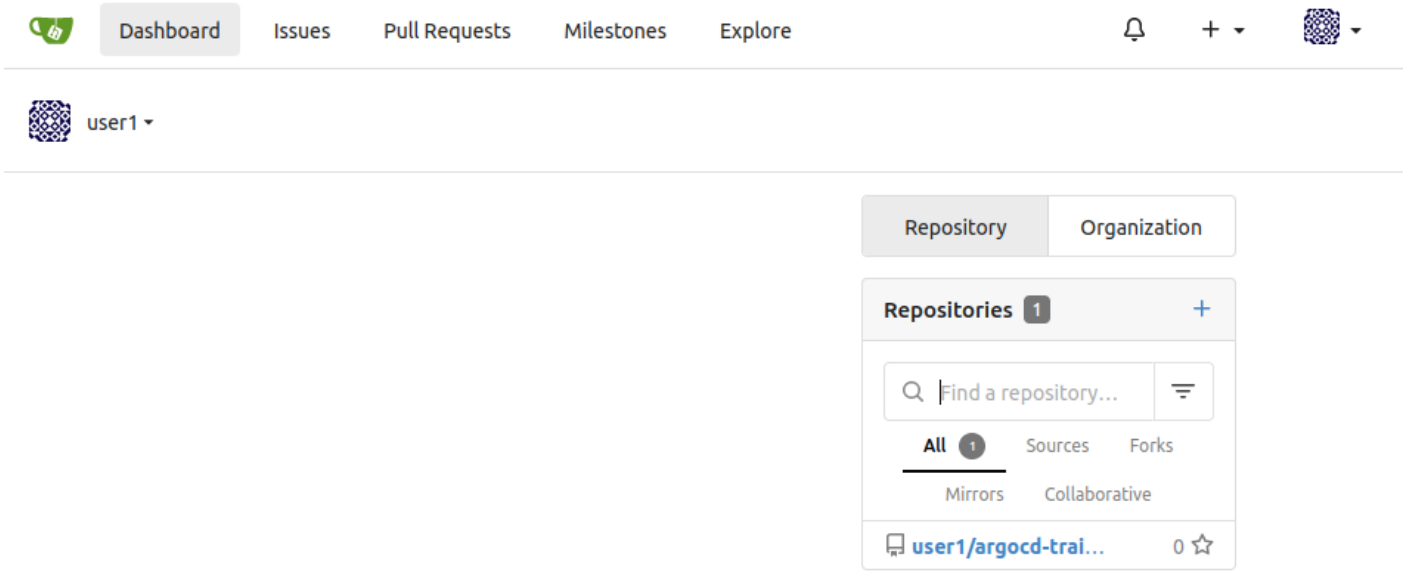
Task 2.1: Login to the Gitea and Clone the Repo

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

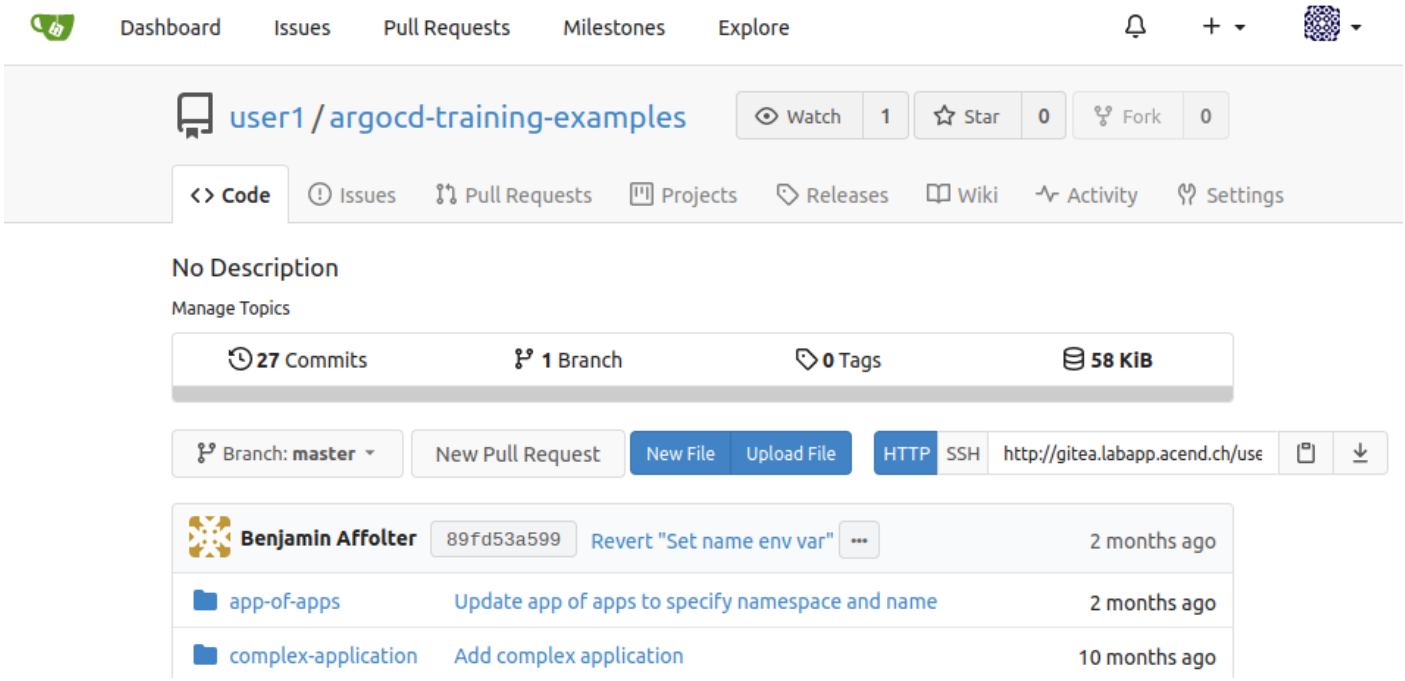
Open your webbrowser and navigate to https://gitea.training.cluster.acend.ch . Login with the training credentials provided by the trainer (Login Button is in the upper right corner).

> **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.



The **URL** of the Git repository, we'll be working with, will look like `https://gitea.training.cluster.acend.ch/<username>/argocd-training-examples.git` .

Within the Web IDE we set the `USER` environment variable to your personal `<username>` .

```
```

Verify that with the following command:

```
```

The `USER` variable will be used as part of the commands to make the lab experience more comfortable for you.

Note

If you're **not** using our lab webshell to execute the labs, make sure to set the `USER` environment variable accordingly with the following command `export USER=<username>`

Clone the forked repository to your local workspace:

```
```

... or the corresponding URL if you have choosen to use your own Git Server.

Change the working directory to the cloned git repository:

```
cd argocd-training-examples/example-app
```

When using the Web IDE: Configure the Git Client and verify the output

```
git config user.name "bb20"
git config user.email bb20@puzzle.training.cluster.acend.ch"
```

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.

```
git config credential.helper 'cache --timeout=86400'
```

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

```
git config --local --list
```

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:

```
argocd app create argo-bb20 --repo https://gitea.training.cluster.acend.ch/bb20/argocd-training-examples.git --path example-app --dest-server https://kubernetes.default.svc --dest-namespace bb20
```

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:

```
argocd app get argo-bb20
```

```
Project:        default
Server:         https://kubernetes.default.svc
Namespace:      bb20
URL:            https://argocd-training.cluster.acend.ch/applications/argo-username
Repo:           https://gitea.training.cluster.acend.ch/username/argocd-training-examples.git
Target:
Path:           example-app
SyncWindow:     Sync Allowed
Sync Policy:    <none>
Sync Status:    OutOfSync from (5a57d42)
Health Status:  Missing

GROUP  KIND        NAMESPACE  NAME       STATUS     HEALTH   HOOK  MESSAGE
       Service     username   simple-example  OutOfSync  Missing
apps   Deployment  username   simple-example  OutOfSync  Missing
```

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

```
argocd app sync argo-$USER
```

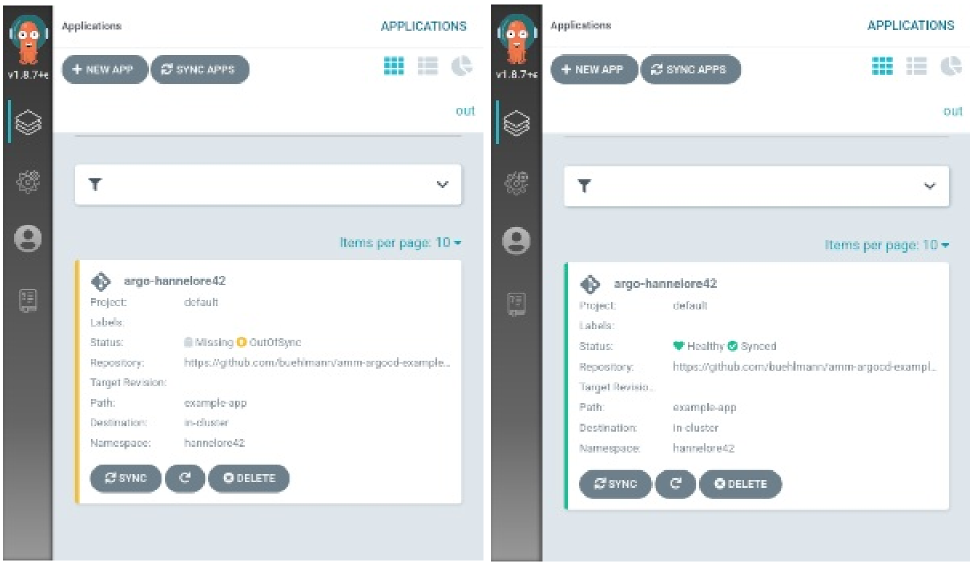
This command retrieves the manifests from the git repository and performs a `kubectl apply` on them. Because all our manifests has been deployed manually before, no new rollout of them will be triggered on Kubernetes. But form now on, all resources are managed by Argo CD. Congrats, the first step in direction GitOps! :)

You should see an output similar to the following lines:

```
TIMESTAMP                  GROUP        KIND   NAMESPACE  NAME            STATUS    HEALTH        HOOK  MESSAGE
2021-03-24T14:19:16+01:00            Service  <username>  simple-example  OutOfSync  Missing
2021-03-24T14:19:16+01:00   apps  Deployment  <username>  simple-example  OutOfSync  Missing
2021-03-24T14:19:16+01:00            Service  <username>  simple-example    Synced  Healthy

Name:               argo-<username>
Project:            default
Server:             https://kubernetes.default.svc
Namespace:          <username>
URL:                https://argocd.training.cluster.acend.ch/applications/argo-<username>
Repo:               https://gitea.training.cluster.acend.ch/<username>/argocd-training-examples.git
Target:
Path:               example-app
SyncWindow:         Sync Allowed
Sync Policy:        <none>
Sync Status:        Synced to  (5a6f365)
Health Status:      Progressing

Operation:          Sync
Sync Revision:      5a6f365d8a65d1dab0761fc0d32b90f7eb480354
Phase:              Succeeded
Start:              2021-03-24 14:19:16 +0100 CET
Finished:           2021-03-24 14:19:16 +0100 CET
Duration:           0s
Message:            successfully synced (all tasks run)

GROUP  KIND        NAMESPACE   NAME            STATUS  HEALTH       HOOK  MESSAGE
       Service     <username>  simple-example  Synced  Healthy            service/simple-example created
apps   Deployment  <username>  simple-example  Synced  Progressing        deployment.apps/simple-example created
```

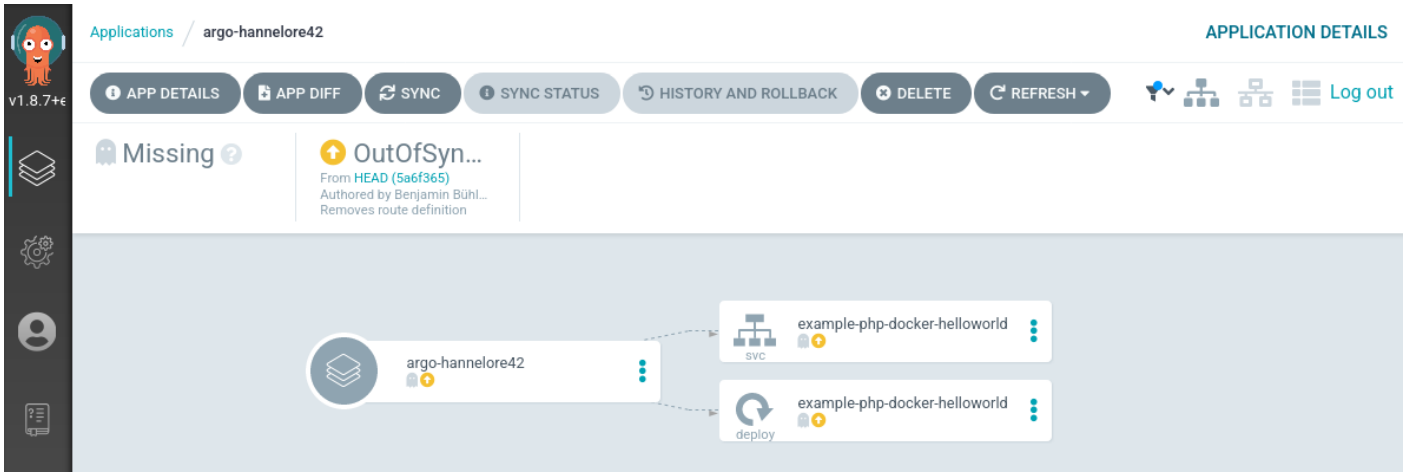
Check the [Argo CD UI] to browse the application and their components.

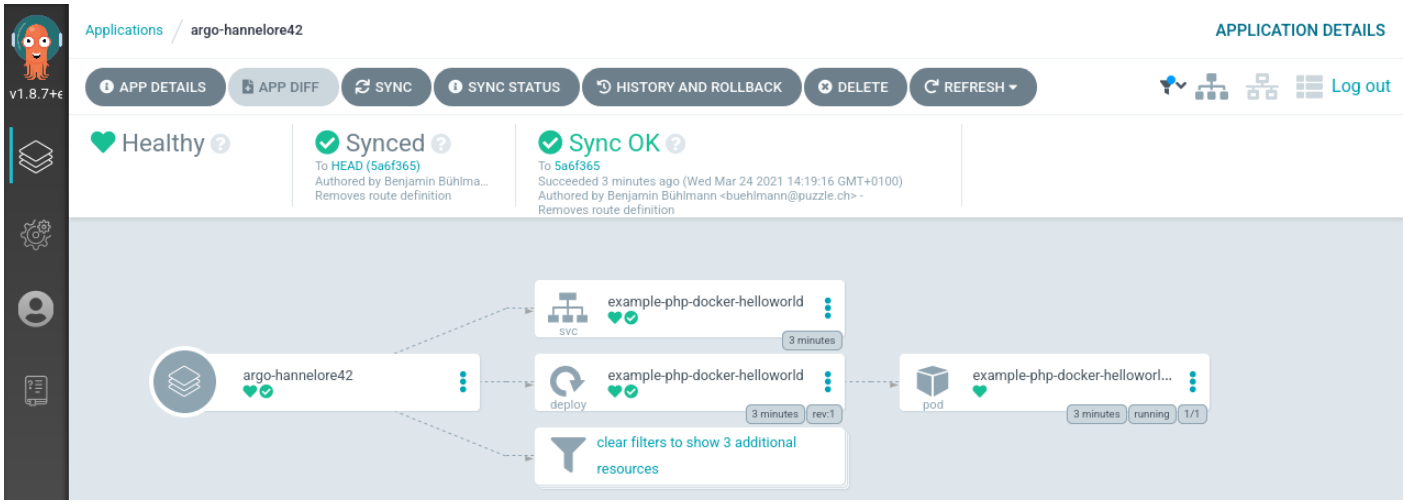
Application overview in unsynced and synced state



Detailed view of a application in unsynced and synced state

- Puzzle ITC GmbH

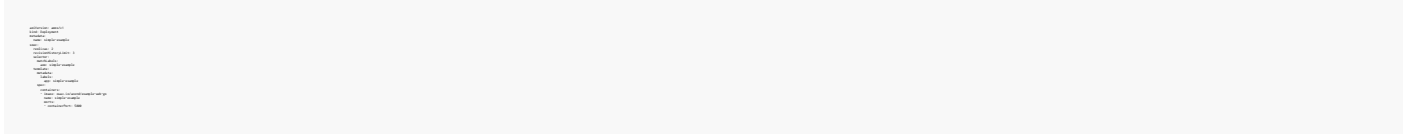




Task 2.3: Automated Sync Policy and Diff

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

```
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 8 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 367 bytes | 367.00 KiB/s, done.
Total 4 (delta 3), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
To https://gitea.training.cluster.acend.ch/<username>/argocd-training-examples.git
   5a6f365..e2d4bbf  master -> master
```

Check the state of the resources by cli:

```
argocd app get argo-$USER --refresh
```

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:

```
Name:               argo-<username>
Project:            default
Server:             https://kubernetes.default.svc
Namespace:          <username>
URL:                https://argocd.training.cluster.acend.ch/applications/argo-<username>
Repo:               https://gitea.training.cluster.acend.ch/<username>/argocd-training-examples.git
Target:
Path:               example-app
SyncWindow:         Sync Allowed
Sync Policy:        <none>
Sync Status:        OutOfSync from  (e2d4bbf)
Health Status:      Healthy

GROUP  KIND        NAMESPACE   NAME            STATUS     HEALTH   HOOK  MESSAGE
       Service     <username>  simple-example  Synced     Healthy        service/simple-example created
apps   Deployment  <username>  simple-example  OutOfSync  Healthy        deployment.apps/simple-example created
```

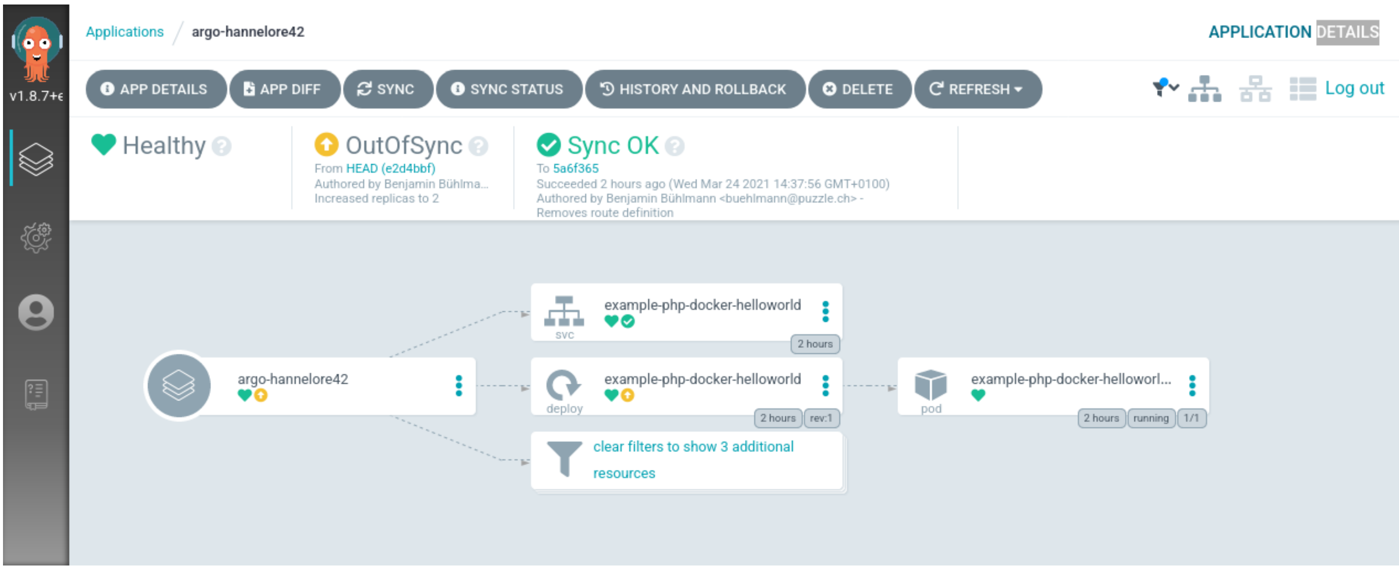
When an application is OutOfSync then your deployed ’live state’ is no longer the same as the ’target state’ which is represented by the resource manifests in the Git repository. You can inspect the differences between live and target state by cli:

```
argocd app diff argo-$USER
```

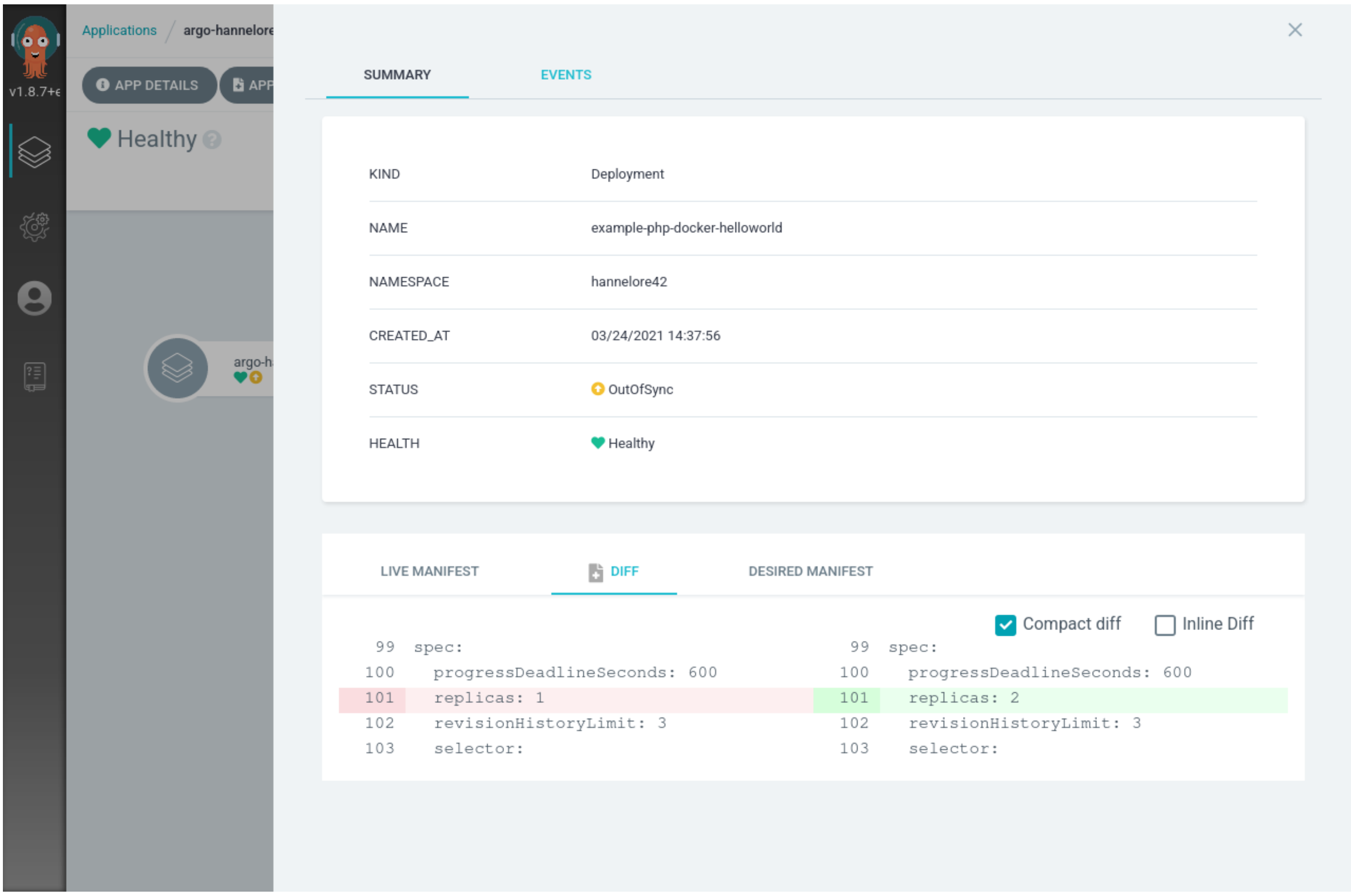
which should give you an output similar to:

```
===== apps/Deployment <username>/simple-example ======
101c102
<   replicas: 1
---
>   replicas: 2
```

Now open the web console of Argo CD and go to your application. The deployment `simple-example` is marked as ‘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 with a click on Deployment > Diff:



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

Double-check the status by cli

```
argocd app get argo-$USER
```

```
Name:               argo-<username>
Project:            default
Server:             https://kubernetes.default.svc
Namespace:          <username>
URL:                https://argocd.training.cluster.acend.ch/applications/argo-<username>
Repo:               https://gitea.training.cluster.acend.ch/<username>/argocd-training-examples.git
Target:
Path:               example-app
SyncWindow:         Sync Allowed
Sync Policy:        <none>
Sync Status:        Synced to  (e2d4bbf)
Health Status:      Healthy

GROUP  KIND        NAMESPACE   NAME            STATUS  HEALTH   HOOK  MESSAGE
       Service     <username>  simple-example  Synced  Healthy        service/simple-example unchanged
apps   Deployment  <username>  simple-example  Synced  Healthy        deployment.apps/simple-example configured
```

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):

```
argocd app set argo-$USER --sync-policy automated
```

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

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:

```
argocd app get argo-$USER --refresh
```

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:

```
argocd app set argo-$USER --self-heal
```

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

```
kubectl get deployment simple-example --watch --namespace=$USER
```

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

```
kubectl scale deployment simple-example --replicas=5 --namespace=$USER
```

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

```
NAME             READY   UP-TO-DATE   AVAILABLE   AGE
simple-example   1/1     1            1           114s
simple-example   1/5     1            1           115s
simple-example   1/5     1            1           115s
simple-example   1/5     1            1           115s
simple-example   1/5     1            1           115s
simple-example   1/5     1            1           115s
simple-example   1/1     1            1           116s
```

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.

```
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: simple-example
spec:
  rules:
    - host: simple-example-<username>-training.cluster.acend.ch
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: simple-example
                port:
                  name: http
  tls:
    - hosts:
        - simple-example-<username>-training.cluster.acend.ch
```

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

```
git add .
git commit -m "Remove application"
git push
```

After ArgoCD syncs the changes, you can access the example applications url: `https://simple-example-<username>.training.cluster.acend.ch`

Verify using the following command:

```
curl https://simple-example-%USER.training.cluster.acend.ch
```

The result should look similar to this:

```
<h1> Welcome</h1>
```

Task 2.6: Pruning

You probably asked yourself: how can I delete deployed resources on the container platform? Argo CD can be configured to delete resources that no longer exist in the Git repository.

First delete the files `service.yaml` and `ingress.yaml` from Git repository and push the changes:

```
git add .
git add -u$ git commit -m "Remove service and ingress" $$ git push
```

Check the status of the application with

```
argocd app get argo-%USER --refresh
```

You will see that even with auto-sync and self-healing enabled the status is still OutOfSync

```
GROUP                 KIND        NAMESPACE  NAME                   STATUS     HEALTH   HOOK  MESSAGE
networking.k8s.io     Ingress     test-argo  simple-example         OutOfSync  Healthy        Ingress networking.k8s.io/simple-example created
                      Service     test-argo  simple-example         OutOfSync  Healthy
apps                  Deployment  test-argo  simple-example         Synced     Healthy
```

Now enable the auto pruning explicitly:

```
argocd app set argo-%USER --auto-prune
```

Recheck the status again

```
argocd app get argo-%USER --refresh
```

```
GROUP   KIND        NAMESPACE  NAME                   STATUS   HEALTH   HOOK  MESSAGE
argocd  Service     frontpage  standard-example       Synced   Healthy        service unchanged
apps    Deployment  frontpage  standard-example       Synced   Healthy        deployment.apps/standard-example unchanged
```

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 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`)

```
kubectl get application --namespace=argocd
```

```
NAME          SYNC STATUS   HEALTH STATUS
argo-rollouts   Synced        Healthy
```

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 *yaml* use:

```
kubectl get applications.argoproj.io -o yaml --namespace=argocd
```

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

```
kubectl edit applications.argoproj.io/bgd01 --namespace=argocd
```

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 argo-bgd01
```

You will see the following error

```
FATA[0000] rpc error: code = FailedPrecondition desc = authentication required
```

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.

```
argocd repo add https://gitea.training.cluster.acend.ch/k8000/argocd-training-examples.git --username k8000
```

> **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.

```
argocd app sync argo-k8000
```

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>)

```
argocd repocreds add https://gitea.training.cluster.acend.ch/k8000 --username k8000
```

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:

```
argocd app delete argo-k8000
```

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:

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.

- `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

```
argocd app create argo-hook-$USER --repo https://gitea.training.cluster.acend.ch/$USER/argocd-training-examples.git --path pre-post-sync-hook --dest-server https://kubernetes.default.svc --dest-namespace $USER
```

Sync the application

```
argocd app sync argo-hook-$USER
```

And verify the deployment:

```
kubectl get pod --namespace $USER --watch
```

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: batch/v1
kind: Job
metadata:
  name: ...
  annotations:
    argocd.argoproj.io/hook: PostSync
    argocd.argoproj.io/hook-delete-policy: HookSucceeded
spec:
  template:
    spec:
      containers:
        - name: ...
          image: quay.io/acend/example-web-python
          command: ["curl", "https://acend.ch"]
      restartPolicy: Never
```

Task 3.3: Delete the Application

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

```
argocd app delete 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

```
argocd app create argo-wave-$USER --repo https://gitea.training.cluster.acend.ch/$USER/argocd-training-examples.git --path 'sync-wave' --dest-server https://kubernetes.default.svc --dest-namespace $USER
```

Sync the application:

```
argocd app sync argo-wave-$USER
```

And verify the deployment:

```
kubectl get pod --namespace $USER --watch
```

Task 4.2: Delete the Application

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

```
argocd app delete argo-wave-$USER
```

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:

```
argocd app set argo-helm-8689 --values values-production.yaml
```

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.

```
argocd app set argo-helm-8689 --parameter replicaCount=4
```

> **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:

```
argocd app set argo-helm-8689 --release-name myrelease
```

> **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

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 argo-tute-kd09 --repo https://gitea.training.cluster.acend.ch/kd09/argocd-training-examples.git --path 'helm/simple-example' --dest-server https://kubernetes.default.svc --dest-namespace kd09 --values values.yaml
```

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 argo-tute-kd09
```

And verify the deployment:

```
kubectl get pod --namespace kd09 --watch
```

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

```
argocd app set argo-tute-kd09 --sync-policy automated
argocd app set argo-tute-kd09 --self-heal
argocd app set argo-tute-kd09 --auto-prune
```

Task 5.1.2: Scale the deployment to 2 replicas

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

```
argocd app set argo-tute-kd09 --parameter replicaCount=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

```
ingress:
  enabled: true
  hosts:
    - host: kd<namespace>.training.cluster.acend.ch
      paths:
        - path: /
          pathType: Prefix
  tls:
    - hosts:
        - kd<namespace>.training.cluster.acend.ch
      secretName: acend-wildcard
```

Commit and push the changes to your repository.

```
git add helm/simple-example/values.yaml
git commit -m "Update replicas"
git push
```

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

```
ingress:
  enabled: true
  hosts:
    - host: helm-<username>-prod.training.cluster.ucont.ch
      paths:
        - path: /
  tls:
    - hosts:
        - helm-<username>-prod.training.cluster.ucont.ch
      secretName: ucont-wildcard
```

Commit and push the changes to your repository.

```
git add helm/simple-example/values-production.yaml
git commit -m "add prod stage"
git push
```

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

```
argocd app create argo-helm-prod-$USER --repo https://github.com/puzzle/training-cluster-ucont.ch/$USER/argocd-training-examples.git --path "helm/simple-example" --dest-server https://kubernetes.default.svc --dest-namespace $USER
argocd app set argo-helm-prod-$USER --sync-policy automated
argocd app set argo-helm-prod-$USER --self-heal
argocd app set argo-helm-prod-$USER --auto-prune
```

And verify the deployment:

```
kubectl get pod --namespace $USER -watch
```

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

```
argocd app set argo-helm-prod-$USER --values values-production.yaml
```

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

Task 5.1.5: Delete the Applications

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

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:

```
argocd app set argo-kustomize-$USER --nameprefix=<namePrefix>
```

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.

```
argocd app create argo-kustomize-$USER --repo https://gitea.training.cluster.acend.ch/$USER/argocd-training-examples.git --path 'kustomize/simple-example' --dest-server https://kubernetes.default.svc --dest-namespace $USER
```

Sync the application

To sync (deploy) the resources you can simply click sync in the web UI

- Puzzle ITC GmbH

or execute the following command:

```
argocd app sync argo-kustomize-%USER
```

And verify the deployment:

```
kubectl get pod --namespace %USER --watch
```

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

```
argocd app set argo-kustomize-%USER --sync-policy automated
argocd app set argo-kustomize-%USER --auto-heal
argocd app set argo-kustomize-%USER --auto-prune
```

Task 5.2.2: Set a configuration parameter

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

```
argocd app set argo-kustomize-%USER --kustomize-image
```

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

```
argocd app get argo-kustomize-%USER
```

> **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.

```
argocd app create argo-kustomize-prod-%USER --repo https://github.com/puzzle/training-ocp-gitops.git --path 'kustomize/overlays-example/overlays/production' --dest-server https://kubernetes.default.svc --dest-namespace %USER
argocd app set argo-kustomize-prod-%USER --sync-policy automated
argocd app set argo-kustomize-prod-%USER --auto-heal
argocd app set argo-kustomize-prod-%USER --auto-prune
```

And verify the deployment:

```
kubectl get pod --namespace %USER --watch
```

Task 5.2.4: Delete the Applications

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

```
argocd app delete argo-kuard
argocd app delete argo-kuard-local
```

5.3. Jsonnet (Optional)

This lab explains how to use jsonnet as manifest format together with Argo CD.

Jsonnet

Jsonnet 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

```
{
  application1: {
    name: "jsonnet-application1",
    basepath: "/application/test",
    path: self.basepath + "/" + self.name,
  },
  application2: self.application1 { name: "jsonnet-application2" },
}
```

will render into:

```
{
  "application1": {
    "basepath": "/application/test",
    "name": "jsonnet-application1",
    "path": "/application/test/jsonnet-application1"
  },
  "application2": {
    "basepath": "/application/test",
    "name": "jsonnet-application2",
    "path": "/application/test/jsonnet-application2"
  }
}
```

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

Further Docs

Read more about the jsonnet integration in the official documentation

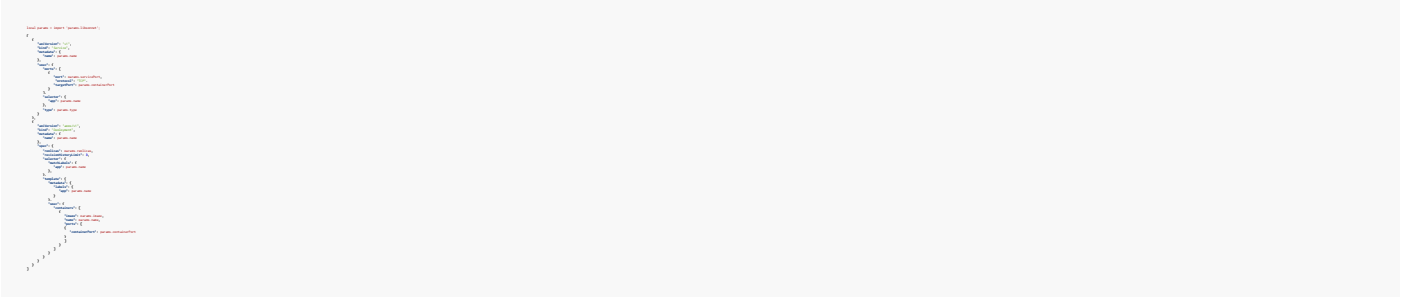
Task 5.3.1: Deploy the simple-example with jsonnet

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:

```
{
  containerPort: 5000,
  image: "quay.io/acend/example-web-go",
  name: "argo-jsonnet-example-<username>",
  replicas: 1,
  servicePort: 5000,
  type: "ClusterIP",
}
```

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



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.

Create the 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:

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

Task 5.3.2: Autosync and scale up

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

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

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

Commit and push the changes to your repository.

```
git add kustomization.libsonnet
git commit -m "Scale Guestbook to Four"
git push
```

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.

```
kubectl get pod --namespace $USER --watch
```

Task 5.3.3: Delete the Applications

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

```
argocd app delete argo-jsonnet-$USER
```

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 .`

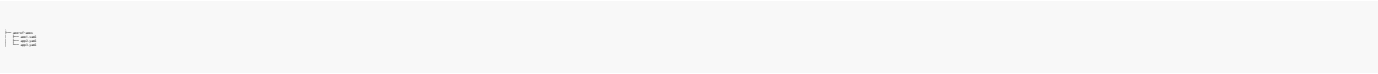
In ArgoCD there are two approaches which allow us to manage a set of applications. In the following two sub chapters you'll learn how the App of Apps and ApplicationSet pattern work.

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.

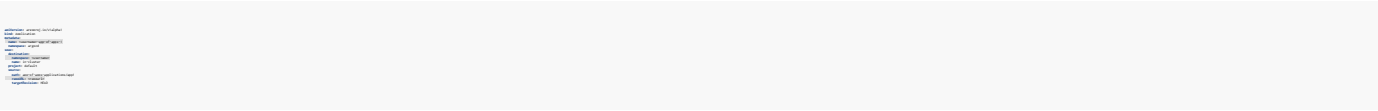


As we can see the directory consists of three ArgoCD applications. Each of them has its own source repository pointing to the corresponding repository containing a kubernetes deployment yaml file.

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

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

- `--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 reconcile the child application
- `--auto-prune` Ensure that if the parent application gets deleted, it also delete the their child applications

```
argocd app create argo-aoa-%USER --repo https://gitea.training.cluster.puzzle.ch/%USER/argocd-training-examples.git --path apps/argo-aoa --dest-server https://kubernetes.default.svc --dest-namespace %USER --sync-policy automated --self-heal --auto-prune
```

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 consits of another three apps.

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

Task 6.1.3: Delete the Application

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

```
argocd app delete argo-aoa-%USER
```

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)

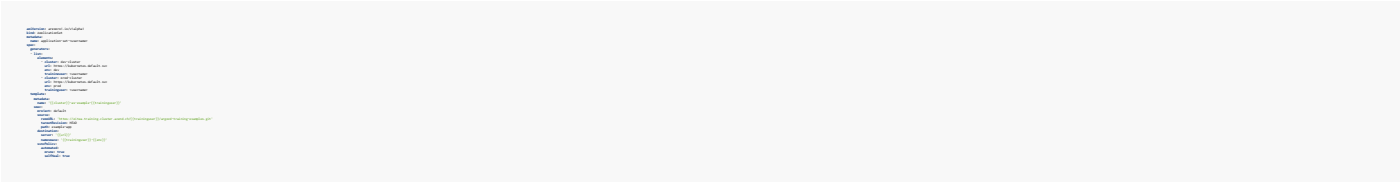
A list of parameters, which come from so called generators , render the ArgoCD Application Template to create a list of Applications.

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 fist 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 yaml) 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.

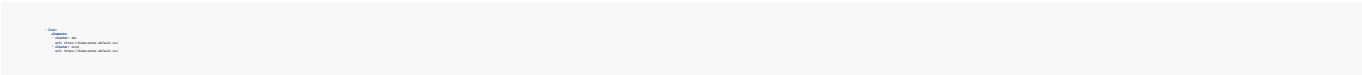
- **SCM Provider Generator + Cluster Generator:** Scanning the repositories of a GitHub organization for application resources, and targeting those resources to all available clusters.
- **Git File Generator + List Generator:** Providing a list of applicatations to deploy via configuration files, with 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

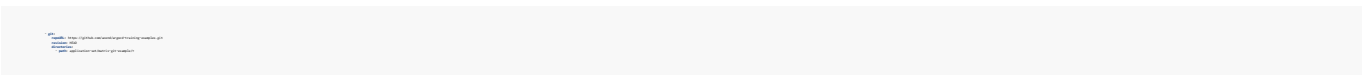
In this lab section we're going to create an ApplicationSet for an multi-environment.

- 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

- Puzzle ITC 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:

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

Push the changes to your git repository.

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

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.

7. Projects

Argo CD applications can be linked to a project which provides a logical grouping of applications. The following configurations can be made on a project:

- Source repositories: Git repositories where application manifests are permitted to be retrieved from
- Destination Clusters: Permitted destination Kubernetes or OpenShift clusters
- Destination Namespaces: Destination namespaces where the manifests are permitted to be deployed to
- Permitted resource kinds to be synced (e.g. `ConfigMap`)
- Sync windows: Time windows when an application is permitted to be synced by Argo CD.
- Roles: Roles and policies assigned to the project. The roles are bound to OIDC groups and/or JWT tokens)
- GPG Signature Keys: GnuPG keys that commits must be signed with in order to be allowed to sync them
- Resource Monitoring: Visualization and monitoring of orphaned resources

In summary, a project defines who can deploy what to which destination. This is very useful to keep the isolation between different user groups working on the same Argo CD instance and enables the capability of multi tenancy.

Task 7.1: Create a new empty project

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

```
argocd proj create project-$USER
argocd proj list
```

You should find in the output your newly created project:

```
NAME                    DESCRIPTION  DESTINATIONS  SOURCES  CLUSTER-RESOURCE-WHITELIST  NAMESPACE-RESOURCE-BLACKLIST  SIGNATURE-KEYS  ORPHANED-RESOURCES
...
default                              *,*           *        */*                         <none>                        <none>          disabled
project-<username>                   <none>        <none>   <none>                      <none>                        <none>          disabled
...
```

Task 7.2: Define permitted sources and destinations

The next step is to deploy a new application and assign it to the created project `project-<username>` by using the flag `--project`

```
argocd app create project-app-$USER --repo https://github.com/acend/argocd-training-examples.git --path 'example-app' --dest-server https://kubernetes.default.svc --dest-namespace $USER --project project-$USER
```

You will receive an error when trying to create the new application

```
FATA[0000] rpc error: code = InvalidArgument desc = application spec is invalid: InvalidSpecError: application repo https://github.com/acend/argocd-training-examples.git is not permitted in project 'project-<username>';InvalidSpecError: application destination {https://kubernetes.default.svc <username>} is not permitted in project 'project-<username>'
```

The cause for this error is the missing setting for the allowed destination clusters and namespaces on the project. We will fix that by setting the allowed destination cluster to `https://kubernetes.default.svc` and using the wildcard expression `user*` as allowed namespace names.

```
argocd proj add-destination project-b609 https://kubernetes.default.svc *
```

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

```
argocd proj add-source project-b609 *
```

Now print out the details of the project again

```
argocd proj get project-b609
```

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

```
Destinations:     https://kubernetes.default.svc,user
Namespaces:       *
```

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

```
argocd app create project-app-b609 --repo https://github.com/puzzle/argocd-training-examples.git --path examples/app --dest-server https://kubernetes.default.svc --dest-namespace b609 --project project-b609
```

Now sync the application manifest

```
argocd app sync project-app-b609
```

> **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` .

```
argocd proj deny-namespace-resource project-b609 '' Service
```

Now sync the application

```
argocd app sync project-app-b609
```

The sync operation will fail with the following error

```
...
GROUP  KIND        NAMESPACE   NAME            STATUS   HEALTH   HOOK  MESSAGE
       Service     <username>  simple-example  Unknown  Missing        Resource :Service is not permitted in project project-<username>.
apps   Deployment  <username>  simple-example  Synced   Healthy
FATA[0001] Operation has completed with phase: Failed
```

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

```
argocd proj allow-namespace-resource project-$USER "" Service
```

… and sync the app again

```
argocd app sync project-app-$USER
```

Task 7.4: Cleanup

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

```
argocd app delete project-app-$USER
argocd proj delete project-$USER
```

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 >=1.2.0 <1.3.0
Track minor releases (e.g. in QA)	Use a range	1.* or >=1.0.0 <2.0.0
Use the latest (e.g. in local development)	Use star range	* or >=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 achieved in two ways. Eiteher you can specifiy 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 immutuability. 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 to show you how to pin a version from a Git tag and therefore freeze the deployment to specific commits.

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 origin --tags
```

Re-create the simple application example:

```
argocd app create argocd-example-0609 --repo https://gitea.training.cluster.puzzle.ch/0609/argocd-training-example.git --path simple-app --dest-server https://kubernetes.default.svc --dest-namespace 0609
```

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

```
argocd app set argocd-example-0609 --revision 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.

```
replicas: 2
```

For commiting and pushing your changes to your Git repository, execute follwing command:

```
git add . && git commit -m "feat: deployment replicas to 2" && git push origin
```

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

```
argocd app sync argocd-example-0609
```

Check the number of configured replicas on the app deployment.

To see the number of configured replicas execute follwing command:

```
kubectl describe deployment simple-example
```

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

```
NAME             READY   UP-TO-DATE   AVAILABLE   AGE
simple-example   1/1     1            1           6m52s
```

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 repsitory. Then update the revision and resync the ArgoCD app.

- Puzzle ITC GmbH

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

```
git tag v1.0.1 && git push origin --tags
```

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

```
argocd app set argo-example-9609 --revision v1.0.1
```

With the new created tag, ArgoCD is goingt 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 argo-example-9609
```

Then diplay the status with following command:

```
argocd app get argo-example-9609
```

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

```
kubectl describe deployment simple-example
```

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

```
NAME                 READY   UP-TO-DATE   AVAILABLE   AGE
simple-example   2/2     2            2           5m5s
```

Task 8.1.2: Delete the Application

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

```
argocd app delete argo-example-9609
```

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.

```
argocd proj create -s "*" -d "*,*" project-sync-windows-$USER
argocd app create sync-windows-$USER --repo https://github.com/acend/argocd-training-examples.git --path 'example-app' --dest-server https://kubernetes.default.svc --dest-namespace $USER --project project-sync-windows-$USER
argocd app sync sync-windows-$USER
```

You should see the following message after a successful sync

```
...
Message:            successfully synced (all tasks run)
...
```

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.

```
argocd proj windows add project-sync-windows-$USER \
    --kind deny \
    --schedule "0 8 * * *" \
    --duration 12h \
    --applications "*" \
    --namespaces "" \
    --clusters ""
```

List all registered sync windows for the project.

```
argocd proj windows list project-sync-windows-$USER
```

..prints out

```
ID  STATUS  KIND  SCHEDULE   DURATION  APPLICATIONS  NAMESPACES  CLUSTERS  MANUALSYNC
0   Active  deny  0 8 * * *  12h       *             *           *         Disabled
```

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

> **Note**
> Paste the cron expression on [Crontab Guru] to get an explanation of it.

Now try to sync the previously created application

```
argocd app sync sync-window-9689
```

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

```
FATA[0000] rpc error: code = PermissionDenied desc = cannot sync: blocked by sync window
```

> **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.

```
argocd proj windows update project-sync-window-9689 0 --app-selector "sketchy-app"
```

Sync the application again

```
argocd app sync sync-window-9689
```

.. 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

```
argocd proj windows update project-sync-window-9689 0 --app-selector "*"
```

Task 8.2.4: Enabling manual syncs

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

```
argocd proj windows enable-manual-sync project-sync-window-9689 0
argocd app sync sync-window-9689
```

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

```
argocd proj windows delete project-sync-window-9689 0
argocd app delete sync-window-9689 -y
argocd service project-sync-window-9689
```

- Puzzle ITC 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

```
argocd app create guestbook-ui --repo https://github.com/argoproj/argocd-example-apps.git --path kustomize-guestbook --dest-server https://kubernetes.default.svc --dest-namespace default
argocd app sync argo/guestbook-ui
```

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

```
argocd proj create test
```

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

```
argocd proj set apps-%USER% --orphaned-resources --orphaned-resources-warn
```

> **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

```
argocd app set argo/guestbook-ui --project apps-%USER%
```

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

```
argocd app get argo/guestbook-ui
```

Refresh the application

```
argocd app get --refresh argo-$USER
```

Task 9.1.3: Create orphaned resource

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

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Service
metadata:
  name: black-hole
spec:
  ports:
  - port: 1234
    targetPort: 1234
EOF
```

> **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

```
argocd app resources argo-$USER
```

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

```
GROUP  KIND        NAMESPACE   NAME            ORPHANED
       Service     <username>  simple-example  No
apps   Deployment  <username>  simple-example  No
       Service     <username>  black-hole      Yes
```

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

```
argocd app get --refresh argo-$USER
```

```
...
CONDITION                MESSAGE                              LAST TRANSITION
OrphanedResourceWarning  Application has 1 orphaned resources  2021-09-02 16:20:36 +0200 CEST
...
```

Task 9.1.4: Housekeeping

Clean up the resources created in this lab

```
argocd app delete argo-$USER -y
argocd proj delete apps-$USER
```

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 version | grep server
argocd-server: v2.5.7+eb3d1fb
```

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

```
export ARGOCD=v2.5.7
```

Backup

```
docker run -t -i -v ~/.kube:/home/argocd/.kube --rm argoproj/argocd:$ARGOCD argocd-admin export > backup.yaml
```

> **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 a+r ~/.kube/config
> ```
>
> Don't forget to remove the read permission for others after exporting:
>
> ```
> chmod a-r ~/.kube/config
> ```

Restore

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

```
docker run -t -i -v ~/.kube:/home/argocd/.kube --rm argoproj/argocd:$ARGOCD argocd-admin import - < backup.yaml
```

- Puzzle ITC GmbH

Reference: Disaster recovery