Chapter 4: GitOps and Argo CD

The previous chapters discussed the basics of modern application development with Kubernetes. This chapter shows you how to integrate a project into Kubernetes native pipelines to do your CI/CD and automatically deploy your application out of a pipeline run. We discuss the risks and benefits of using GitOps and Argo CD in your project and give you some hints on how to use it with Red Hat OpenShift.

Introduction to GitOps

I can imagine a reader complaining, "We are still struggling to implement DevOps and now you're coming to us with yet another new fancy acronym to help solve all the issues we still have?" This is something I have heard when I first talked about GitOps during a customer engagement.

The short answer is: DevOps is a cultural change in your enterprise, meaning that developers and operations people should talk to each other instead of doing their work secretly behind big walls.

GitOps is an evolutionary way of implementing continuous deployments for the cloud and Kubernetes. The idea behind GitOps is to use the same version control system you're using for your code to store formal descriptions of the infrastructure desired in the test or production environment. These descriptions can be updated as the needs of the environment change, and can be managed through version control just like source code. You automatically gain a history of all the deployments you've done. After each change, an automated process runs (either through a manual step or through automation) to make the production environment match the desired state. The term "healing" is often applied to the process that brings the actual state of the system in sync with the desired state.

Motivation Behind GitOps

But why Git? And why now? And what does Kubernetes has to do with all that?

As described earlier in this book, you already should be maintaining a formal description of your infrastructure. Each application you're deploying on Kubernetes has a bunch of YAML files that are required to run your application. Adding those files to your project in a Git repository is just a natural step forward. And if you have a tool that could read those files from the repository and apply them to a specified Kubernetes namespace...wouldn't that be great?

Well, that's what GitOps accomplishes. And Argo CD is one of the available tools to help you do GitOps.

What Does a Typical GitOps Process Look Like?

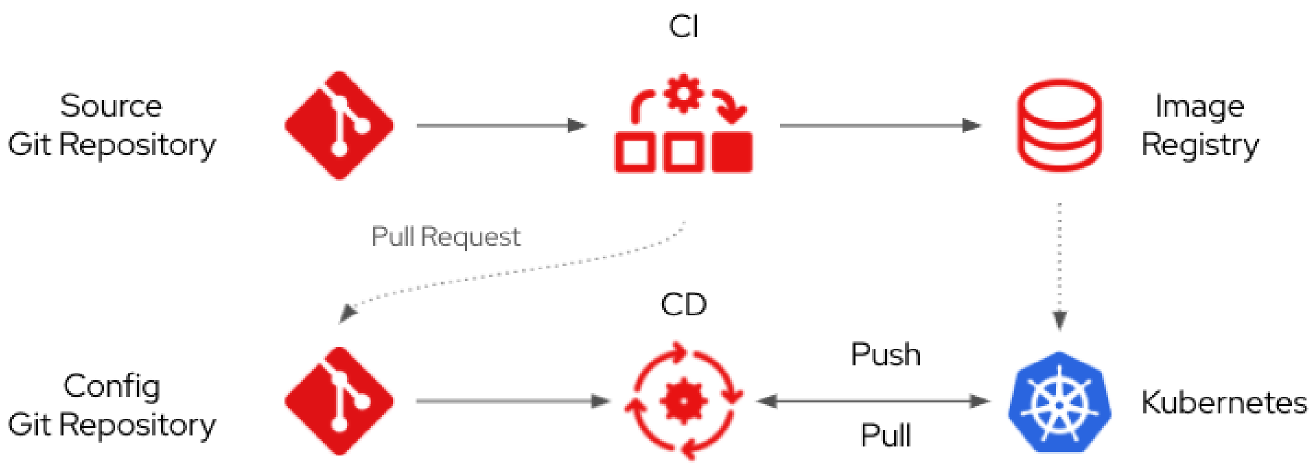
One of the questions people ask most often about GitOps is this: Is it just another way of doing CI/CD? The answer to this question is simply No. GitOps takes care of only the CD part, the delivery part.

Without GitOps, the developer workflow looks like this:

1. A developer implements a change request.
2. Once the developer commits the changes to Git, an integration pipeline is triggered.
3. This pipeline compiles the code, runs all automated tests, and creates and pushes the image.
4. Finally, the pipeline automatically installs the application on the test system.

With GitOps, the developer workflow looks somewhat different (see Image 1):

1. A developer implements a change request.
2. Once the developer commits the changes to Git, an integration pipeline is triggered.
3. This pipeline compiles the code, runs all automated tests, and creates and pushes the image.
4. The pipeline automatically updates the configuration files' directory in the Git repository to reflect the changes.
5. The CD tool sees a new desired state in Git, which is then synchronized to the Kubernetes environment.



So you're still using your pipeline based on Tekton, Jenkins, or whatever to do CI. GitOps then takes care of the CD part.

Argo CD Concepts

Right now (as of version 2.0), the concepts behind Argo CD are quite easy. You register an Argo application that contains pointers to the necessary Git repository with all the application-specific descriptors such as `Deployment`, `Service`, etc., and to the Kubernetes cluster. You might also define an Argo project, which defines various defaults such as:

- Which source repositories are allowed
- Which destination servers and namespaces can be deployed to
- A whitelist of cluster resources to deploy, such as deployments, services, etc.
- Synchronization windows

Each application is assigned to a project and inherits the project's settings. A `default` project is created in Argo CD and contains reasonable defaults.

Once the application is registered, you can manually start a sync to update the actual environment. Alternatively, Argo CD starts "healing" the application automatically, if the synchronization policy is set to do so.

The Use Case: Implementing GitOps for our person-service App

We've been using the person service over the course of this book. Let's continue to use it and create a GitOps workflow for it. You can find all the resources discussed here in the `gitops` folder within the `book-example` repository on GitHub.

We are going to set up Argo CD (via the OpenShift GitOps Operator) on OpenShift 4.9 (via Red Hat CodeReady Containers). We are going to use Tekton to build a pipeline, which updates the person-service-config Git repository with the latest image digest of the build. Argo CD should then detect the changes and should start a synchronization of our application.

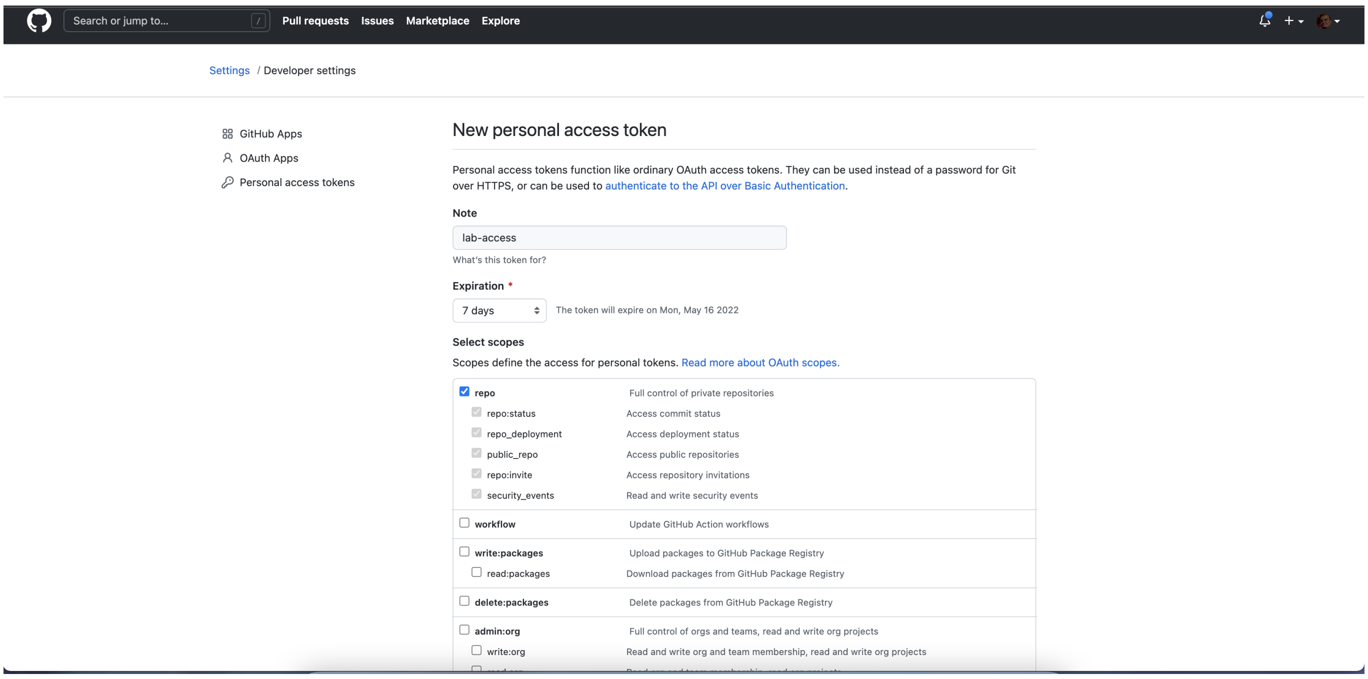
> **Note**: Typically, a GitOps pipeline does not directly push changes into the main branch of a configuration repository. Instead, the pipeline should commit files into a feature branch or release branch and should create a pull request, so that committers can review changes before they are merged to the main branch.

The Application Configuration Repository

First of all, let's create a new repository for our application configuration: `person-service-config`.

Just create a new remote Git repository for example on GitHub.com and copy the URL (for example, `https://github.com/USERNAME/person-service-config.git`).

Create a personal Token using the offical guide



New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to authenticate to the API over Basic Authentication.

Note

lab-access

What's this token for?

Expiration *

7 days — The token will expire on Mon, May 16 2022

Select scopes

Scopes define the access for personal tokens. Read more about OAuth scopes.

✓ repo	Full control of private repositories
✓ repo:status	Access commit status
✓ repo_deployment	Access deployment status
✓ public_repo	Access public repositories
✓ repo:invite	Access repository invitations
✓ security_events	Read and write security events
☐ workflow	Update GitHub Action workflows
☐ write:packages	Upload packages to GitHub Package Registry
☐ read:packages	Download packages from GitHub Package Registry
☐ delete:packages	Delete packages from GitHub Package Registry
☐ admin:org	Full control of orgs and teams, read and write org projects
☐ write:org	Read and write org and team membership, read and write org projects

Then jump to the shell, create a new empty folder somewhere, and issue the following commands:

```
$ mkdir person-service-config
$ cd person-service-config
$ echo "# person-service-config" >> README.md
$ git init
$ git add README.md
$ git commit -m "first commit"
$ git branch -M main
$ git remote add origin https://github.com/waynedovey/person-service-
config.git
```

One of the main concepts behind GitOps is to represent the configuration and build parameters of your application as a Git repository. This repository could be either part of the source code repository or separate. As I am a big fan of *separation of concerns*, we will create a new repository containing the artifacts that we built in earlier chapters using Kustomize:

```
$ tree ../GitOps-Workshop/artifacts/kustomize-final/
../GitOps-Workshop/artifacts/kustomize-final/
└── config
    ├── base
    │   ├── deployment.yaml
    │   ├── kustomization.yaml
    │   ├── postgres.yaml
    │   ├── post-sync-hook.yaml
    │   ├── route.yaml
    │   └── service.yaml
    └── overlays
        ├── dev
        │   └── kustomization.yaml
        ├── prod
        │   ├── apply-health-checks.yaml
        │   └── kustomization.yaml
        └── stage
            ├── apply-health-checks.yaml
            └── kustomization.yaml

6 directories, 11 files
```

Of course, there are several ways to structure your config repositories. Some natural choices include:

1. A single configuration repository with all files covering all services and stages for your complete environment
2. A separate configuration repository per service or application, with all files for all stages
3. A separate configuration repository for each stage of each service

This is completely up to you. But option 1 is probably not optimal because combining all services and stages in one configuration repository might make the repository hard to read, and does not promote separation of concerns. On the other hand, option 3 might break up information too much, forcing you to maintain hundredths of repositories for different applications or services. Therefore, option 2 strikes me as a good balance: One repository per application, containing files that cover all stages for that application.

For now, create this configuration repository by copying the files from the `book-example/kustomize_ext` directory into the newly created Git repository:

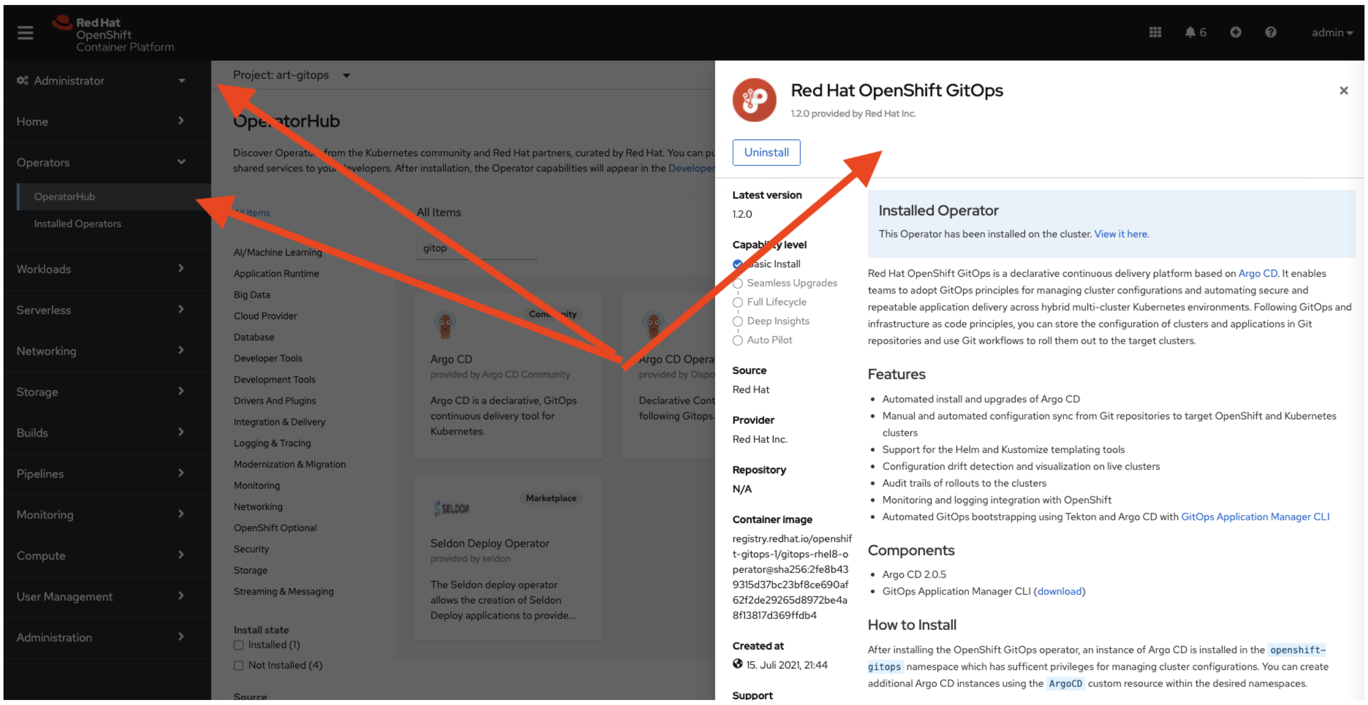
```
$ mkdir config
$ cp -r ../GitOps-Workshop/artifacts/kustomize-final/config/* config/
$ sed -i 's/wpernath/<QuayID>/g' config/base/deployment.yaml
$ sed -i 's/wpernath/<QuayID>/g' config/overlays/dev/kustomization.yaml
$ sed -i 's/wpernath/<QuayID>/g' config/overlays/stage/kustomization.yaml
$ sed -i 's/wpernath/<QuayID>/g' config/overlays/prod/kustomization.yaml
$ git add config
$ git commit -am 'initial commit'
$ git push -u origin main
Username for 'https://github.com':
```

> **Note**: The original `kustomization.yaml` file already contains an image section. This should be removed first.

Installing the OpenShift GitOps Operator

Because the OpenShift GitOps Operator is offered free of charge to OpenShift users and comes quite well preconfigured, I am focusing on its use. If you want to bypass the Operator and dig into Argo CD installation, please feel free to have a look at the official guides.

The OpenShift GitOps Operator can easily be installed in OpenShift. Just log in as a user with cluster-admin rights and switch to the **Administrator** perspective of the OpenShift console. Then go to the **Operators** menu entry and select **OperatorHub** (Image 2). In the search field, start typing "gitops" and select the GitOps Operator when its panel is shown.



Red Hat OpenShift GitOps
1.2.0 provided by Red Hat Inc.

Latest version
1.2.0

Capability level
[✓ Basic Install]
Seamless Upgrades
Full Lifecycle
Deep Insights
Auto Pilot

Source
Red Hat

Provider
Red Hat Inc.

Repository
N/A

Container image
registry.redhat.io/openshift-gitops-1/gitops-rhel8-operator@sha256:2fe8b439315d37bc23bf8ce690af62f2de29265d8972be4a8f13817d369ffdb4

Created at
15. Juli 2021, 21:44

Support

Installed Operator
This Operator has been installed on the cluster. View it here.

Red Hat OpenShift GitOps is a declarative continuous delivery platform based on Argo CD. It enables teams to adopt GitOps principles for managing cluster configurations and automating secure and repeatable application delivery across hybrid multi-cluster Kubernetes environments. Following GitOps and infrastructure as code principles, you can store the configuration of clusters and applications in Git repositories and use Git workflows to roll them out to the target clusters.

Features
- Automated install and upgrades of Argo CD
- Manual and automated configuration sync from Git repositories to target OpenShift and Kubernetes clusters
- Support for the Helm and Kustomize templating tools
- Configuration drift detection and visualization on live clusters
- Audit trails of rollouts to the clusters
- Monitoring and logging integration with OpenShift
- Automated GitOps bootstrapping using Tekton and Argo CD with GitOps Application Manager CLI

Components
- Argo CD 2.0.5
- GitOps Application Manager CLI (download)

How to Install
After installing the OpenShift GitOps operator, an instance of Argo CD is installed in the `openshift-gitops` namespace which has sufficent privileges for managing cluster configurations. You can create additional Argo CD instances using the `ArgoCD` custom resource within the desired namespaces.

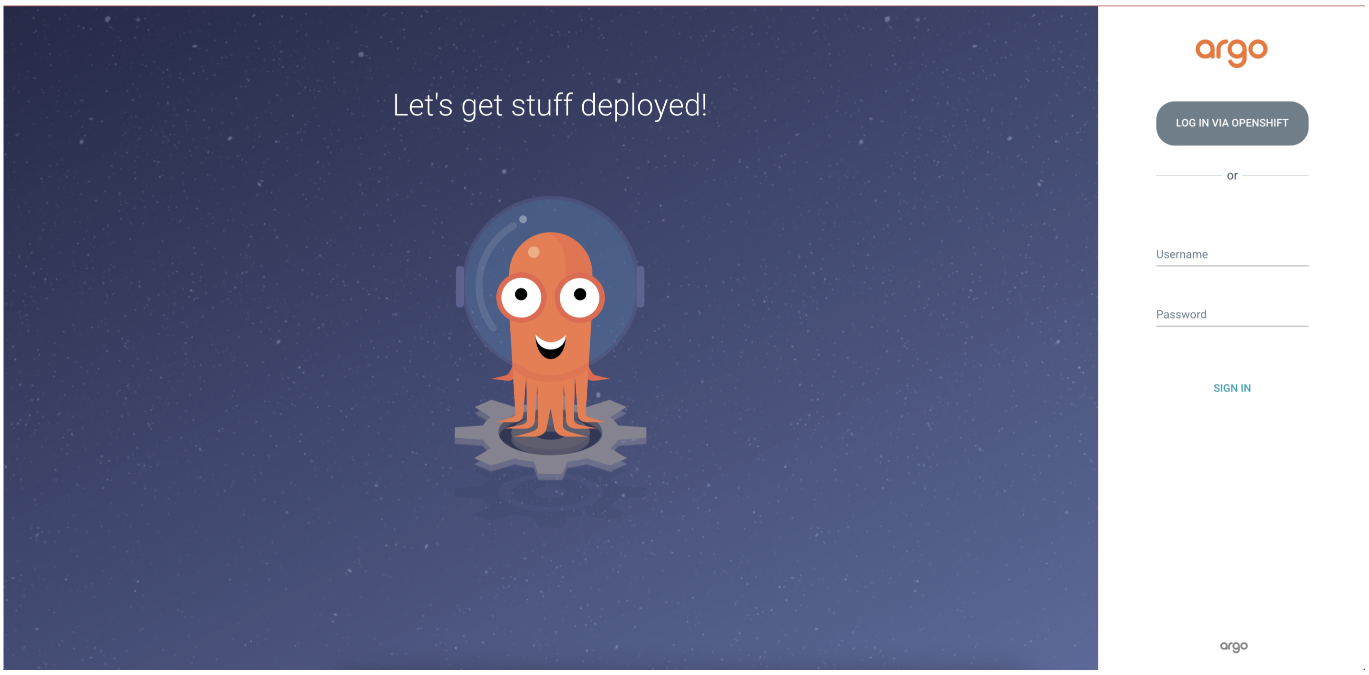
Once the Operator is installed, it creates a new namespace called `openshift-gitops` where an instance of Argo CD is installed and ready to be used.

And this is how to get the URL of your Argo CD instance:

```
$ oc get route openshift-gitops-server -ojsonpath='{.spec.host}' -n
openshift-gitops
```

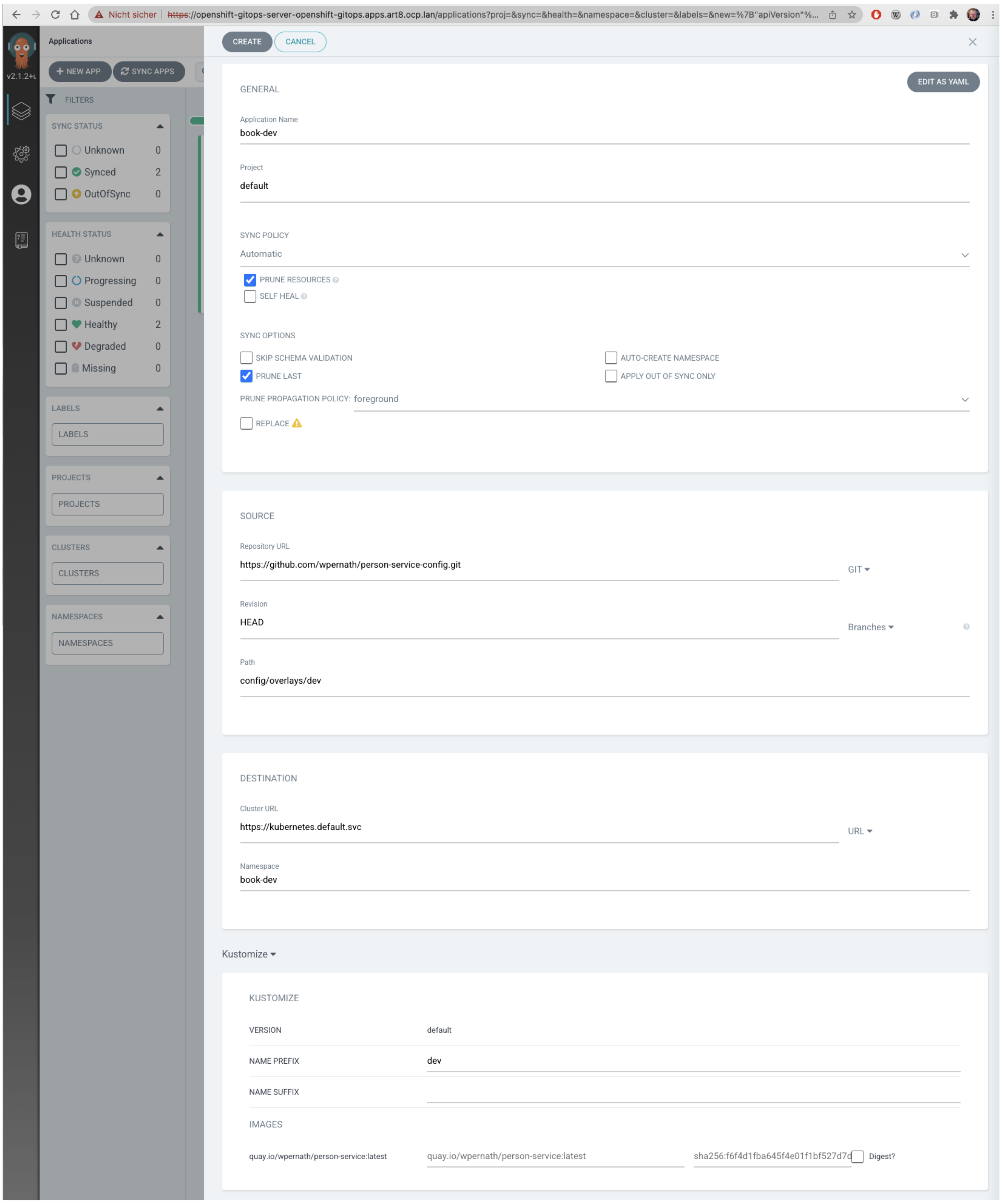
Creating a New Argo Application

The easiest way to create a new Argo application is by using the GUI provided by Argo CD (Image 3).



Go to the URL and log in using `admin` as the user and the password you got as described in the previous section. Click **New App** and fill in the required fields shown in Image 4, as follows:

1. Application Name: We'll use `book-dev`, the same name as our repository.
2. Project: In our case it's `default`, the project created during Argo CD installation.
3. Sync Policy: Choose whether you want automatic synchronization, which is enabled by the **SELF HEAL** option.
4. Repository URL: Specify your directory with the application metadata (Kubernetes resources).
5. Path: This specifies the subdirectory within the repository that points to the actual files.
6. Cluster URL: Specify your Kubernetes instance.
7. Namespace: This specifies the OpenShift or Kubernetes namespace to deploy to.



GENERAL

Application Name
book-dev

Project
default

SYNC POLICY
Automatic

[✓] PRUNE RESOURCES
[ ] SELF HEAL

SYNC OPTIONS

[ ] SKIP SCHEMA VALIDATION
[✓] PRUNE LAST
[ ] AUTO-CREATE NAMESPACE
[ ] APPLY OUT OF SYNC ONLY

PRUNE PROPAGATION POLICY: foreground

[ ] REPLACE

SOURCE

Repository URL
https://github.com/wpernath/person-service-config.git GIT

Revision
HEAD Branches

Path
config/overlays/dev

DESTINATION

Cluster URL
https://kubernetes.default.svc URL

Namespace
book-dev

Kustomize

KUSTOMIZE

VERSION	default
NAME PREFIX	dev
NAME SUFFIX	

IMAGES

quay.io/wpernath/person-service:latest quay.io/wpernath/person-service:latest sha256:f6f4d1fba645f4e01f1bf527d7d [ ] Digest?

After filling out the fields, click **Create**. All Argo CD objects of the default Argo CD instance will be stored in the `openshift-gitops` namespace, from which you can export them via:

```
$ oc get Application/book-dev -o yaml -n openshift-gitops > book-dev-
app.yaml
```

To create an application object in a new Kubernetes instance, open the `book-dev-app.yaml` file exported by the previous command in your preferred editor:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: book-dev
  namespace: openshift-gitops
spec:
  destination:
    namespace: book-dev
    server: https://kubernetes.default.svc
  project: default
  source:
    path: config/overlays/dev
    repoURL: https://github.com/wpernath/person-service-config.git
    targetRevision: HEAD
  syncPolicy:
    automated:
      prune: true
    syncOptions:
    - PruneLast=true
```

Remove the metadata from the object file so that it looks like the listing just shown, and then enter the following command to import the application into the predefined Argo CD instance:

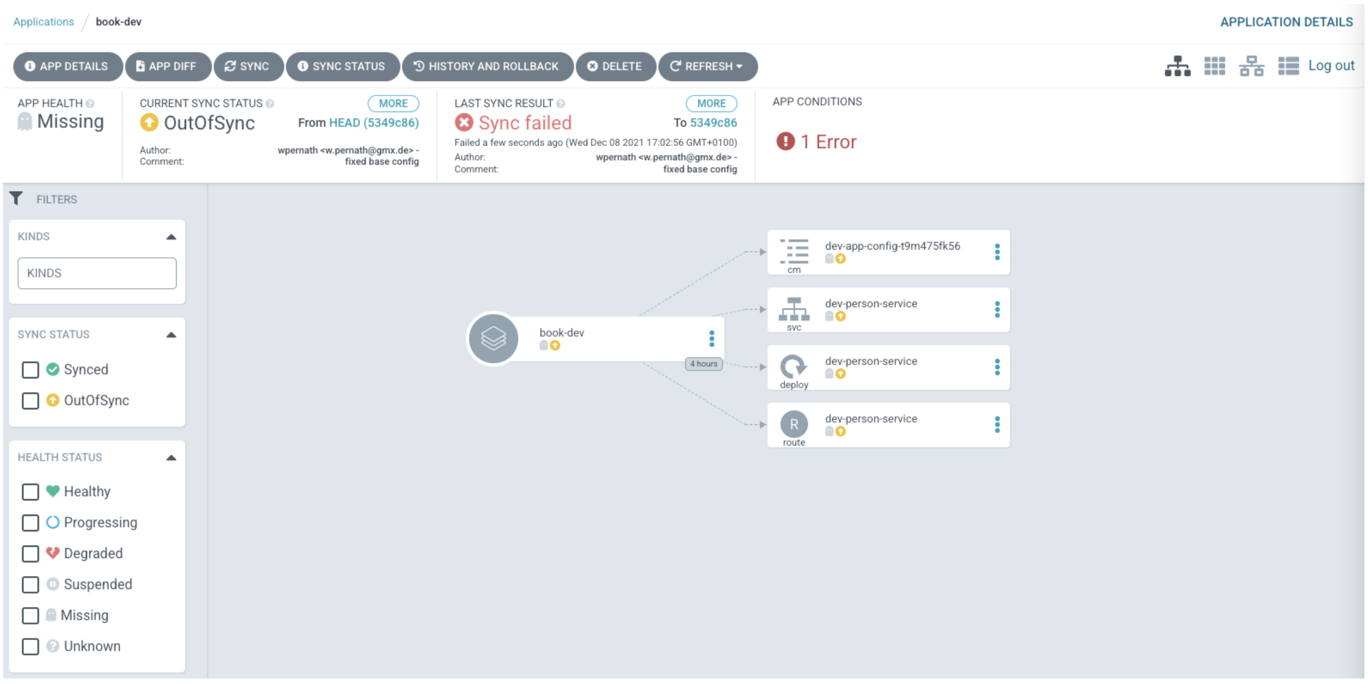
```
$ oc apply -f book-dev-app.yaml -n openshift-gitops
```

Now import the application into the predefined Argo CD instance. Please note that you have to import the application into the `openshift-gitops` namespace. Otherwise it won't be recognized by the default Argo CD instance running after you've installed the OpenShift GitOps operator.

First Synchronization

As you've chosen to do an automatic synchronization, Argo CD will immediately start synchronizing the configuration repository with your OpenShift target server. However, you might notice that the first synchronization takes quite a while and breaks without doing anything except to issue an error message

(Image 5).



Applications / book-dev

APP HEALTH: Missing

CURRENT SYNC STATUS: OutOfSync From HEAD (5349c86)
Author: wpernath <w.pernath@gmx.de> -
Comment: fixed base config

LAST SYNC RESULT: Sync failed To 5349c86
Failed a few seconds ago (Wed Dec 08 2021 17:02:56 GMT+0100)
Author: wpernath <w.pernath@gmx.de> -
Comment: fixed base config

APP CONDITIONS: 1 Error

The error arises because the service account of Argo CD does not have the necessary authority to create typical resources in a new Kubernetes namespace. You have to enter the following command for each namespace Argo CD is taking care of:

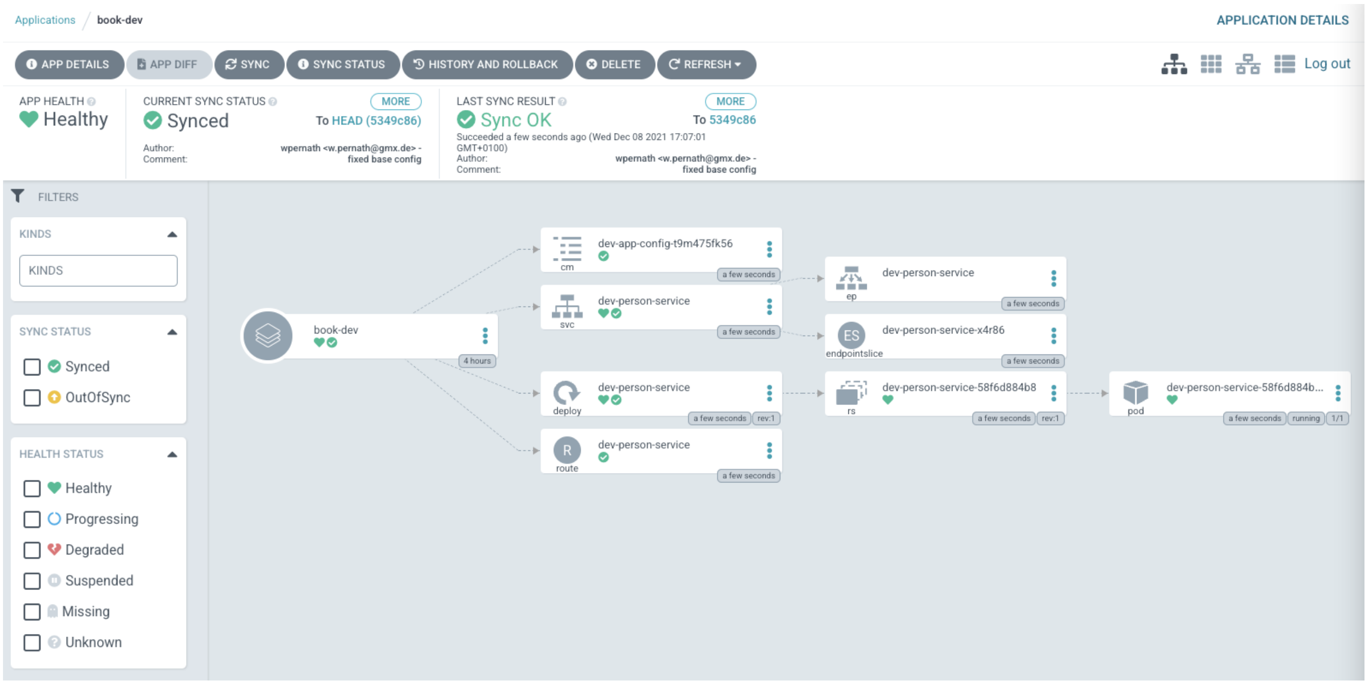
```
$ oc policy add-role-to-user admin \
  system:serviceaccount:openshift-gitops:openshift-gitops-argocd-
application-controller \
  -n book-dev
```

Alternatively, if you prefer to use a YAML description file for this task, create something like the following:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: book-dev-role-binding
  namespace: book-dev
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: admin
subjects:
- kind: ServiceAccount
  name: openshift-gitops-argocd-application-controller
  namespace: openshift-gitops
```

> **Note**: You could also provide cluster-admin rights to the Argo CD service account. This would have the benefit of granting Argo CD to everything on its own. The drawback is that Argo is then superuser of your Kubernetes cluster. This might not be very secure.

After you've given the service account the necessary role, you can safely click **Sync** and Argo CD will do the synchronization (Image 6).



APP HEALTH
Healthy

CURRENT SYNC STATUS
Synced
To HEAD (5349c86)
Author: wpernath <w.pernath@gmx.de> -
Comment: fixed base config

LAST SYNC RESULT
Sync OK
To 5349c86
Succeeded a few seconds ago (Wed Dec 08 2021 17:07:01 GMT+0100)
Author: wpernath <w.pernath@gmx.de> -
Comment: fixed base config

If you chose automatic sync during configuration, any change to a file in the application's Git repository will cause Argo CD to check what has changed and start the necessary actions to keep the environment in sync.

Automated setup

In order to automatically create everything you need to let Argo CD start synchronizing your config repository with a Kubernetes cluster, you have to create the following files. Please have a look at `~/GitOps-Workshop/artifacts/gitops/argocd/`.

Argo CD application config file

The Argo CD application config file is named `book-apps.yaml`. This file contains the Application instructions for Argo CD discussed earlier:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: book-dev
  namespace: openshift-gitops
spec:
  destination:
    namespace: book-dev
    server: https://kubernetes.default.svc
  project: default
  source:
    path: config/overlays/dev
    repoURL: https://github.com/wpernath/person-service-config.git
    targetRevision: HEAD
  syncPolicy:
    automated:
      prune: true
    syncOptions:
    - PruneLast=true
```

A File to Create the Target Namespace

Because we are talking about an automated setup, you also need to automatically create the target namespace. This can be achieved via the `ns.yaml` file, which looks like:

```
apiVersion: v1
kind: Namespace
metadata:
  annotations:
    openshift.io/description: ""
    openshift.io/display-name: "DEV"
  labels:
    kubernetes.io/metadata.name: book-dev
  name: book-dev
spec:
  finalizers:
  - kubernetes
```

The Role Binding

As described earlier, you need a role binding that makes sure the service account of Argo CD is allowed to create and modify the necessary Kubernetes objects. This can be done via the `roles.yaml` file:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: book-dev-role-binding
  namespace: book-dev
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: admin
subjects:
- kind: ServiceAccount
  name: openshift-gitops-argocd-application-controller
  namespace: openshift-gitops
```

Use Kustomize to Apply All Files in One Go

Until now, you have had to apply all the preceding files separately. Using Kustomize, you're able to apply all files in one go. To accomplish this simplification, create a `kustomization.yaml` file, which looks like:

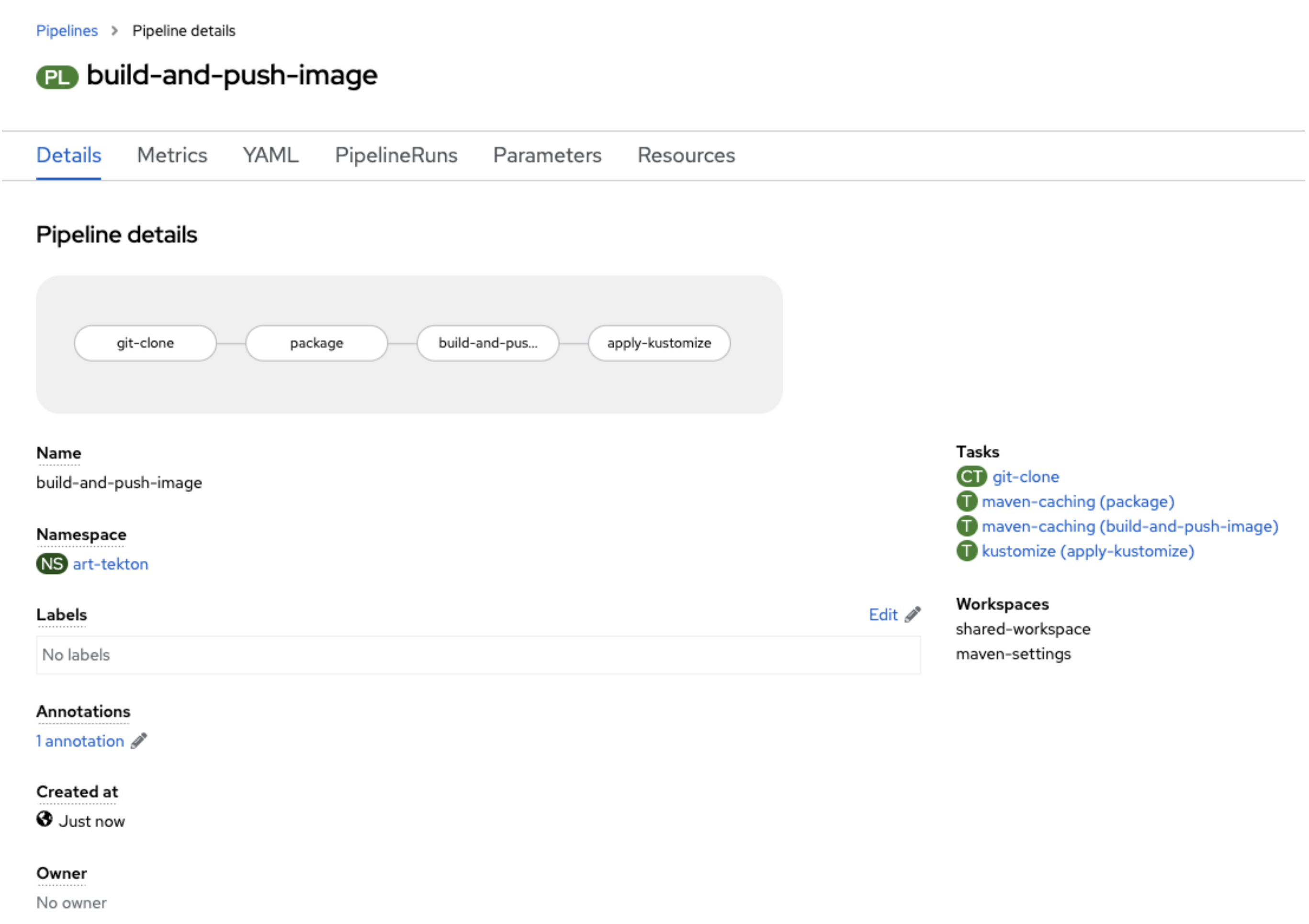
```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
- ns.yaml
- roles.yaml
- book-apps.yaml
- postgresql.yaml
```

To install everything in one go, you simply have to execute the following command:

```
$ echo "REPLACE <username> with github ID"
$ sed -i 's/wpernath/<username>/g' ~/GitOps-
Workshop/artifacts/gitops/argocd/book-apps.yaml
$ oc apply -k ~/GitOps-Workshop/artifacts/gitops/argocd/
```

Creating a Tekton Pipeline to Update person—service-config (Overview Only)

We now want to change our pipeline from the previous chapter (Figure 7) to be more GitOps'y. But what exactly needs to be done?



Pipelines > Pipeline details

PL build-and-push-image

Details Metrics YAML PipelineRuns Parameters Resources

Pipeline details

Name
build-and-push-image

Namespace
NS art-tekton

Labels
No labels

Annotations
1 annotation

Created at
Just now

Owner
No owner

Tasks
CT git-clone
T maven-caching (package)
T maven-caching (build-and-push-image)
T kustomize (apply-kustomize)

Workspaces
shared-workspace
maven-settings

The current pipeline is a development pipeline, which will be used to:

- Compile and test the code.
- Create a new image.
- Push that image to an external registry (in our case, quay.io).
- Use Kustomize to change the image target.
- Apply the changes via the OpenShift CLI to a given namespace.

In GitOps, we don’t do deployments through pipelines anymore. The final step of our pipeline is to update our `person-service-config` Git repository with the new version of the image.

The current pipeline is a development pipeline, which will be used to:

1. Compile and test the code.
2. Create a new image.
3. Push that image to an external registry (in our case Quay.io)).
4. Use Kustomize to change the image target.
5. Apply the changes via the OpenShift CLI to a given namespace.

In GitOps, we don't do pipeline-centric deployments anymore. As explained earlier, the final step of our pipeline just updates our `person-service-config` Git repository with the new version of the image. Instead of the `apply-kustomize` task, we are creating and using the `git-update-deployment` task as final step. This task should clone the config repository, use Kustomize to apply the image changes, and finally push the changes back to GitHub.com.

A Word On Tekton Security

Because we want to update a private repository, we first need to have a look at Tekton authentication. Tekton uses specially annotated secrets with either a `<username>`/`<password>` combination or an SSH key. The authentication then produces a `~/.gitconfig` file (or for an image repository, a `~/.docker/config.json` file) and maps it into the step's pod via the run's associated ServiceAccount. That's easy, isn't it? Configuring the process looks like:

```
apiVersion: v1
kind: Secret
metadata:
  name: git-user-pass
  annotations:
    tekton.dev/git-0: https://github.com
type: kubernetes.io/basic-auth
stringData:
  username: <cleartext username>
  password: <cleartext password>
```

Once you've filled in the `username` and `password`, you can apply the secret into the namespace where you want to run your newly created pipeline.

```
$ oc new-project book-ci
$ oc apply -f secret.yaml
```

Now you need to either create a new ServiceAccount for your pipeline or update the existing one, which was generated by the OpenShift Pipeline Operator. The pipeline runs completely within the security context of the provided ServiceAccount.

Let's use on a new ServiceAccount. To see which other secrets this ServiceAccount requires, execute:

```
$ oc get sa/pipeline -o yaml
```

Copy the secrets to your own ServiceAccount:

```
apiVersion: v1
kind: ServiceAccount
    metadata:
    name: pipeline-bot
secrets:
- name: git-user-pass
```

You don't need to copy the following generated secrets to your ServiceAccount, because they will be linked automatically with the new ServiceAccount by the Operator:

- `pipeline-dockercfg-`: The default secret for reading and writing images from and to the internal OpenShift registry.
- `pipeline-token-`: The default secret for the `pipeline` ServiceAccount. This is used internally.

You also have to create two RoleBindings for the ServiceAccount. Otherwise, you can't reuse the PersistenceVolumes we've been using so far:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: piplinebot-rolebinding1
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: pipelines-scc-clusterrole
subjects:
  - kind: ServiceAccount
    name: pipeline-bot
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: piplinebot-rolebinding2
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: edit
subjects:
  - kind: ServiceAccount
    name: pipeline-bot
```

The `edit` role is mainly used if your pipeline needs to change any Kubernetes metadata in the given namespace. If your pipeline doesn't do things like that, you can safely ignore that role. In our case, we don't necessary need the edit role.

The git-update-deployment Tekton Task

Now that you understand Tekton authentication and have created all the necessary manifests, you are able to focus on the `git-update-deployment` task.

Remember, we want to have a task that does the following:

- Clone the configuration Git repository.
- Update the image digest via Kustomize.
- Commit and push the changes back to the repository.

This means you need to create a task with at least the following parameters:

- `GIT_REPOSITORY`: The configuration repository to clone.
- `CURRENT_IMAGE`: The name of the image in the `deployment.yaml` file.
- `NEW_IMAGE`: The name of the new image to deploy.
- `NEW_DIGEST`: The name of the digest of the new image to deploy. This digest is generated in the `build-and-push-image` step that appears in both the Chapter 4 version and this chapter's version of the pipeline.
- `KUSTOMIZE_PATH`: The path within the `GIT_REPOSITORY` with the `kustomization.yaml` file.

And of course, you need to create a workspace to hold the project files.

Let's have a look at the steps within the task:

```
    steps:
      - name: update-digest
        image: quay.io/wpernath/kustomize-ubi:latest
        workingDir: $(workspaces.workspace.path)/the-config
        script: |
          cd $(params.KUSTOMIZATION_PATH)
          kustomize edit set image
$(params.CURRENT_IMAGE)=$(params.NEW_IMAGE)@$(params.NEW_DIGEST)

          cat kustomization.yaml

      - name: git-commit
        image: docker.io/alpine/git:v2.26.2
        workingDir: $(workspaces.workspace.path)/the-config
        script: |
          git config user.email "wpernath@redhat.com"
          git config user.name "My Tekton Bot"
          git add $(params.KUSTOMIZATION_PATH)/kustomization.yaml
          git commit -am "[ci] Image digest updated"

          git push origin HEAD:main

          RESULT_SHA="$(git rev-parse HEAD | tr -d '\n')"
          EXIT_CODE="$?"
          if [ "$EXIT_CODE" != 0 ]
          then
            exit $EXIT_CODE
          fi
          # Make sure we don't add a trailing newline to the result!
          echo -n "$RESULT_SHA" > $(results.commit.path)
```

Nothing special here. It's the same things we would do via the CLI. The full task and everything related can be found, as always, in the `gitops/tekton/tasks` folder of the repository on GitHub.

Creating an extract-digest Tekton Task

The next question is how to get the image digest. Because we are using the Quarkus image builder (which in turn is using Jib), we need to create either a step or a separate task that provides content to create a `target/jib-image.digest` file.

Because I want to have the `git-update-deployment` task as general-purpose as possible, I have created a separate task that does just this step. The step relies on a Tekton feature known as emitting results from a task.

Within the `spec` section of a task, you can define a `results` property. Each result is stored in `$(results.<result-name>.path)`, where the `<result-name>` component is a string that refers to the data in that result. Results are available in all tasks and on the pipeline level through strings in the format:

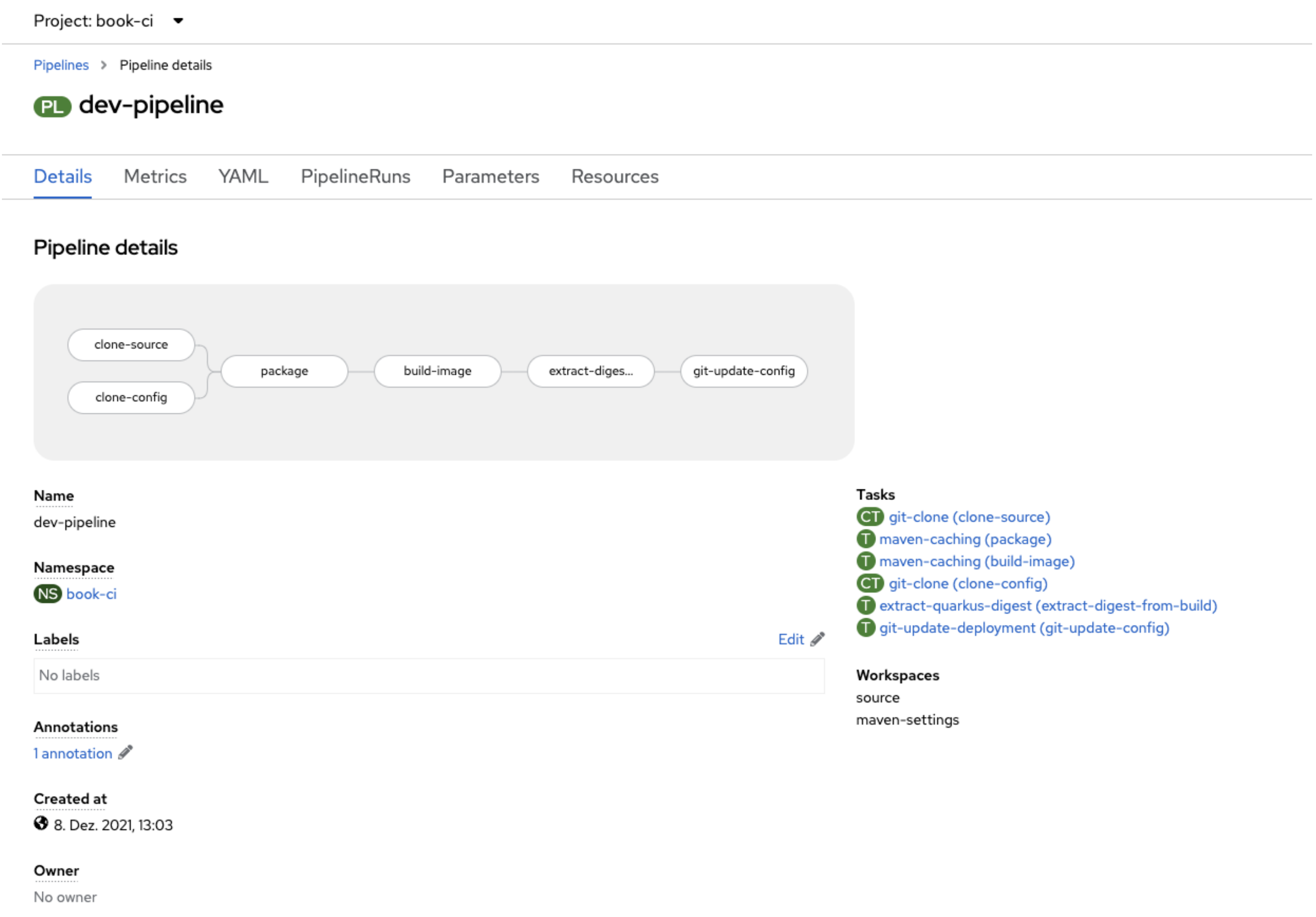
```
$(tasks.<task-name>.results.<result-name>)
```

The following configuration defines the step that extracts the image digest and stores it into a result:

```
spec:
  params:
    - name: image-digest-path
      default: target

  results:
    - name: DIGEST
      description: The image digest of the last quarkus maven build with
JIB image creation

  steps:
    - name: extract-digest
```

```
      image: quay.io/wpernath/kustomize-ubi:latest
      script: |
        # extract DIGEST
        DIGEST=$(cat $(workspaces.source.path)/$(params.image-digest-
path)/jib-image.digest)
        # Store DIGEST into result
        echo -n $DIGEST > $(results.DIGEST.path)
```

Now it's time to summarize everything in a new pipeline. Image 8 shows the tasks. The first three are the same as in the Chapter 4 version of this pipeline. We have added the `extract-digest` step as described in the previous section, and end by updating our repository.



Pipeline details

Name
dev-pipeline

Namespace
NS book-ci

Labels
No labels

Annotations
1 annotation

Created at
8. Dez. 2021, 13:03

Owner
No owner

Tasks
CT git-clone (clone-source)
T maven-caching (package)
T maven-caching (build-image)
CT git-clone (clone-config)
T extract-quarkus-digest (extract-digest-from-build)
T git-update-deployment (git-update-config)

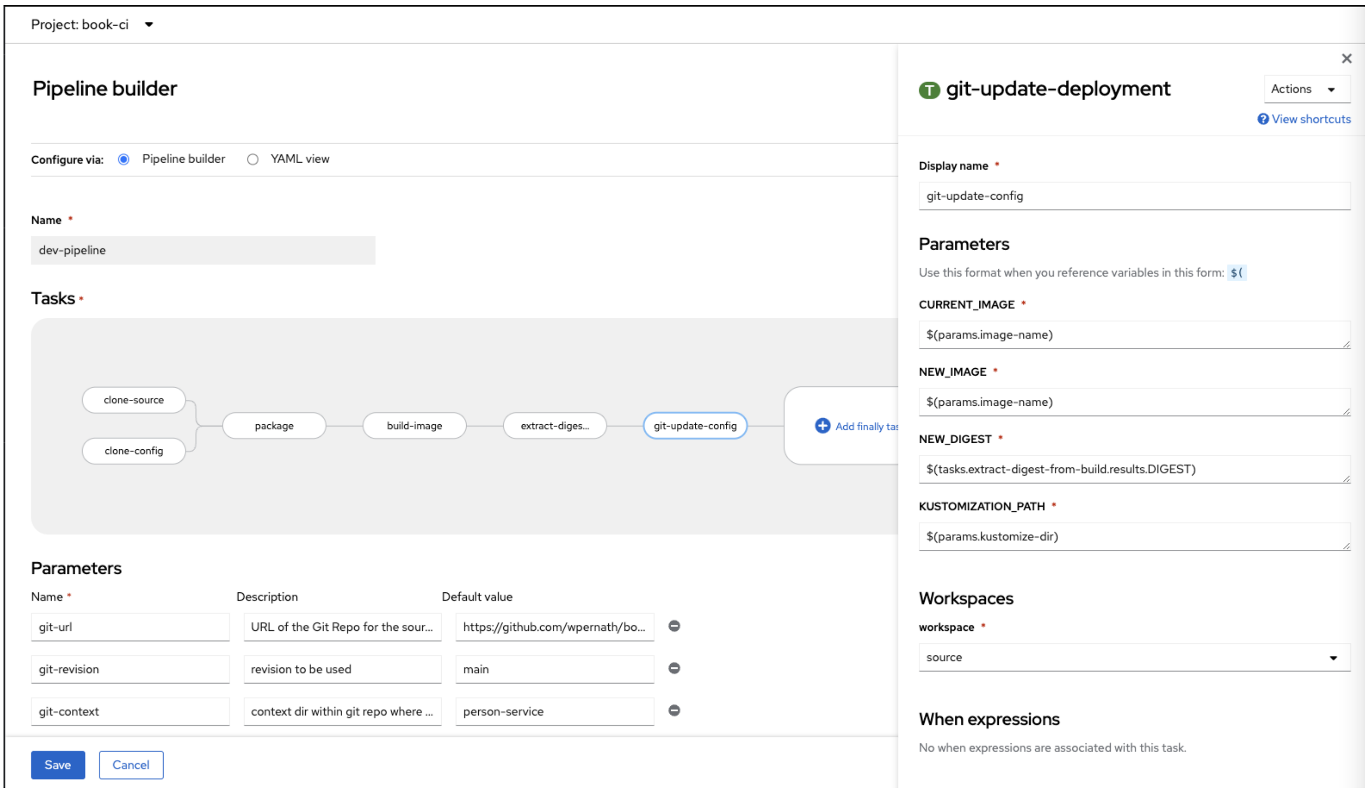
Workspaces
source
maven-settings

Start by using the previous non-GitOps pipeline, which we created in Chapter 4. Remove the last task and add `extract-digest` and `git-update-deployment` as new tasks.

Add a new `git-clone` task at the beginning by hovering over `clone-source` and pressing the plus sign below it to create a new parallel task. Name the new task `clone-config` and fill in the necessary parameters:

- `config-git-url`: This should point to the service configuration repository.
- `config-git-revision`: This is the branch name of the configuration repository to clone.

Map these parameters to the `git-update-deployment` task, as shown in Image 9.



Name: dev-pipeline

Tasks

Parameters

Name	Description	Default value
git-url	URL of the Git Repo for the sour...	https://github.com/wpernath/bo...
git-revision	revision to be used	main
git-context	context dir within git repo where ...	person-service

git-update-deployment

Display name: git-update-config

Parameters

Use this format when you reference variables in this form: $(

CURRENT_IMAGE: $(params.image-name)

NEW_IMAGE: $(params.image-name)

NEW_DIGEST: $(tasks.extract-digest-from-build.results.DIGEST)

KUSTOMIZATION_PATH: $(params.kustomize-dir)

Workspaces

workspace: source

When expressions

No when expressions are associated with this task.

Testing the Pipeline

You can't currently run the pipeline from the user interface because you can't use a different ServiceAccount that lacks the two secrets you need to provide. Therefore, start a pipeline via the CLI. For your convenience, I have created a Bash script called `~/GitOps-Workshop/artifacts/gitops/tekton/pipeline.sh` that can be used to initialize your namespace and start the pipeline.

To create the necessary namespaces and Argo CD Applications, enter the following command, passing your username and password:

```
$ cd ~/GitOps-Workshop/artifacts/gitops/tekton/
$ sed -i 's/github.com\/wpernath/github.com\/<GitHubID>/g' ~/GitOps-
Workshop/artifacts/gitops/tekton/pipelines/dev-pipeline.yaml
$ sed -i 's/github.com\/wpernath/github.com\/<GitHubID>/g' ~/GitOps-
Workshop/artifacts/gitops/tekton/pipelines/stage-release.yaml
$ sed -i 's/wpernath/<QuayID>/g' ~/GitOps-
Workshop/artifacts/gitops/tekton/pipelines/dev-pipeline.yaml
$ sed -i 's/wpernath/<QuayID>/g' ~/GitOps-
Workshop/artifacts/gitops/tekton/pipelines/stage-release.yaml
$ ./pipeline.sh init [--force] --git-user <user> \
    --git-password <pwd> \
    --registry-user <user> \
    --registry-password <pwd>
```

If the `--force` option is included, the command creates the following namespaces and Argo CD applications for you:

- `book-ci`: Pipelines, tasks, and a Nexus instance
- `book-dev`: The current dev stage
- `book-stage`: The most recent stage release

The following command starts the development pipeline.

```
$ ./pipeline.sh build -u <reg-user> \
    -p <reg-password>
```

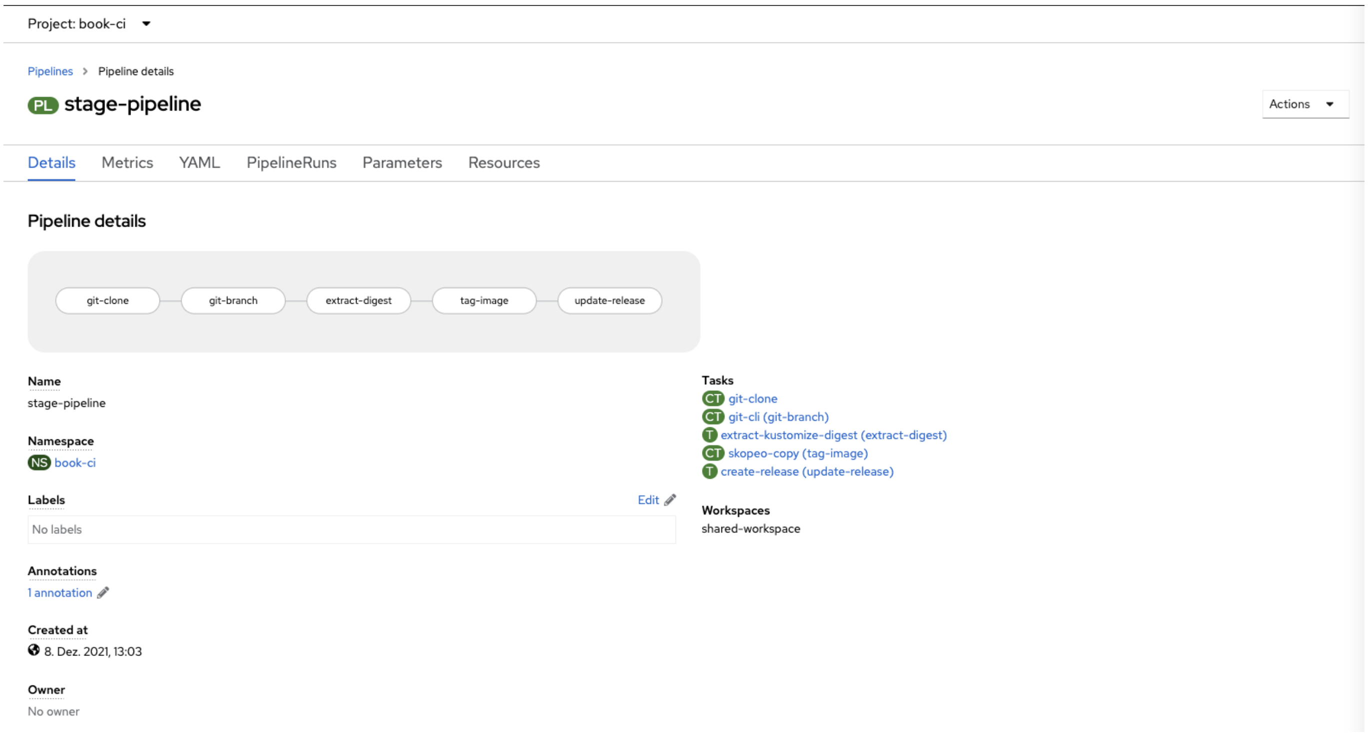
Whenever the pipeline is successfully executed, you should notice an updated message in the `person-service-config` Git repository. And you should notice that Argo CD has initiated a synchronization process, which ends with a redeployment of the quarkus application.

Have a look at Chapter 4 for more information on starting and testing pipelines.

Creating a stage-release Pipeline

What does a staging pipeline look like? We need a process which does the following, in our case (Image 10):

1. Clone the config repository.
2. Create a release branch (e.g., `release-1.2.3`).
3. Get the image digest. In our case, we extract the image out of the current development environment.
4. Tag the image in the image repository (e.g., `quay.up/wpernath/person-service:1.2.3`).
5. Update the configuration repository and point the stage configuration to the newly tagged image.
6. Commit and push the code back to the Git repository.
7. Create a pull or merge request.



Pipeline details

Name
stage-pipeline

Namespace
NS book-ci

Labels
No labels

Annotations
1 annotation

Created at
8. Dez. 2021, 13:03

Owner
No owner

Tasks
CT git-clone
CT git-cli (git-branch)
T extract-kustomize-digest (extract-digest)
CT skopeo-copy (tag-image)
T create-release (update-release)

Workspaces
shared-workspace

These tasks are followed by a manual process where a test specialist accepts the pull request and merges the content from the branch back into the main branch. Then Argo CD takes the changes and updates the running staging instance in Kubernetes.

You can use the Bash script I created as follows to start the staging pipeline, creating release 1.0.1:

```
$ ./pipeline.sh stage -r v1.0.1-test -g
https://github.com/waynedovey/person-service-config.git
```

Setup of the Pipeline

The `git-clone` and `git-branch` steps use existing ClusterTasks, so there is nothing to explain here except one new Tekton feature: Conditional execution of a task by using a "When" expression.

In our case, if a `release-name` is specified, only the `git-branch` task should be executed. The corresponding YAML code in the pipeline looks like:

```
      when:
        - input: $(params.release-name)
          operator: notin
          values:
            - ""
```

The new `extract-digest` task uses `yq` to extract the digest out of the `kustomization.yaml` file. The command looks like:

```
$ yq eval '.images[1].digest'
$(workspaces.source.path)/$(params.kustomize-dir)/kustomization.yaml
```

The result of this call is stored in the task's `results` field.

The tag-image Task

The `tag-image` task uses a `skopeo-copy` ClusterTask, which requires a source image and a target image. The original use case of this task was to copy images from one repository to another (for example, from the local repository up to an external Quay.io repository). However, you can also use this task to tag an image in a repository. The corresponding parameters for the task are:

```
    - name: tag-image
      params:
        - name: srcImageURL
          value: >-
            docker://$(params.target-image)@$(tasks.extract-
digest.results.DIGEST)
        - name: destImageURL
          value: >-
            docker://$(params.target-image):$(params.release-name)
        - name: srcTLSverify
          value: 'false'
        - name: destTLSverify
          value: 'false'
      runAfter:
        - extract-digest
      taskRef:
        kind: ClusterTask
        name: skopeo-copy
[...]
```

`skopeo` uses an existing Docker configuration if it finds one in the home directory of the current user. For us, this means that we have to create another secret with the following content:

```
apiVersion: v1
kind: Secret
metadata:
  annotations:
    tekton.dev/docker-0: https://quay.io
  name: quay-push-secret
type: kubernetes.io/basic-auth
stringData:
  username: <use your quay.io user>
  password: <use your quay.io token>
```

Also update the ServiceAccount accordingly:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: pipeline-bot
secrets:
- name: git-user-pass
- name: quay-push-secret
```

After you've applied the configuration changes, the `skopeo` task is able to authenticate to Quay.io and do its work.

Creating the release

The final `create-release` task does more or less exactly the same work as the task we've already used in the `gitops-dev-pipeline` task:

- Run Kustomize to set the image in the `config/overlays/stage/kustomization.yaml` file.
- Commit and push the changes back to GitHub.

Challenges

Of course, you still face quite some challenges if you're completely switching over to do GitOps. Some challenges spring from the way Kubernetes intrinsically works, which has pros and cons by itself. Other challenges exist because originally, Git was meant to be used by people who are able to analyze merge conflicts and apply them manually.

However, nobody says GitOps would be the only and *de facto* method of doing CD nowadays. If GitOps is not right for your environment, simply don't use it.

But let's discuss some of the challenges and possible solutions.

Order-Dependent Deployments

Although order-dependent deployments are not necessarily a best practice, the reality is that nearly every large application does have dependencies, and these must be fulfilled before the installation process can continue. Two examples:

- Before my app can start, I must have a properly installed and configured database.
- Before I can install an instance of a Kafka service, an Operator must be installed on Kubernetes.

Fortunately, Argo CD has a solution for those scenarios. Like with Helm Charts, you're able to define sync phases and so called *waves*.

Argo CD defines three sync phases:

- PreSync: Before the synchronization starts
- Sync: The actual synchronization phase
- PostSync: After the synchronization is complete

Within each phase, you can define waves to list activities to perform. However, presync and postsync can contain only *hooks*. A hook could be of any Kubernetes type, such as a pod or job, but could also be of type TaskRun or PipelineRun (if the corresponding CRDs are already installed in your Cluster).

Waves can be defined by annotating your Kubernetes resources with the following annotation:

```
metadata:
  annotations:
    argocd.argoproj.io/sync-wave: <+/- number>
```

Argo CD sorts all resources first by the phase, then by the wave, and finally by type and name. If you know that some resources need to be applied before others, simply group them via the annotation. By default, Argo CD uses wave zero for any resources and hooks.

Nondeclarative Deployments

Nondeclarative deployments are simply lists of steps; they are also called *imperative* deployments. For most of us, this is the most familiar type of deployment. For instance, if you're creating a hand-over document for the OPS department, you are providing imperative instructions about how to install the application. And most of us are used to creating installation scripts for more complex applications.

However, the preferred way of installing applications with Kubernetes is through declarative deployments. These specify the service, the deployment, persistent volume claims, secrets, config maps, etc.

If this declaration is not enough, you have to provide a script to configure a special resource—for example, to update the structure of a database or do a backup of the data first.

As mentioned earlier, Argo CD manages synchronization via phases and executes resource hooks when necessary. So you can define a presync hook to execute a database schema migration or fill the database with test data. You might create a postsync hook to do some tiding or health checks.

Let's create such a hook:

```
apiVersion: batch/v1
kind: Job
metadata:
  generateName: post-sync-run
  name: my-final-run
  annotations:
    argocd.argoproj.io/hook: PostSync
spec:
  ttlSecondsAfterFinished: 100
  template:
    spec:
      restartPolicy: Never
      containers:
        - name: post-install-job
          env:
            - name: NAMESPACE
              value: book-dev
          image: "quay.io/wpernath/kustomize-ubi:v4.4.1"
          command:
          - /bin/sh
          - -c
          - |
            echo "WELCOME TO the post installation hook for Argo CD "
            echo "-------------------------------------------------"
            echo "We are now going to fill the database by calling "
            echo "the corresponding REST service"

            SERVICE_URL=person-service.${NAMESPACE}.svc:8080/person
            NUM_PERSONS=$(curl http://$SERVICE_URL/count)

            if [ $NUM_PERSONS -eq 0 ]; then
              echo "There are no persons in the database, filling some"

              http --ignore-stdin --json POST ${SERVICE_URL}
firstName=Carlos lastName=Santana salutation=Mr
              http --ignore-stdin --json POST ${SERVICE_URL} firstName=Joe
lastName=Cocker salutation=Mr
              http --ignore-stdin --json POST ${SERVICE_URL}
firstName=Eric lastName=Clapton salutation=Mr
              http --ignore-stdin --json POST ${SERVICE_URL}
```

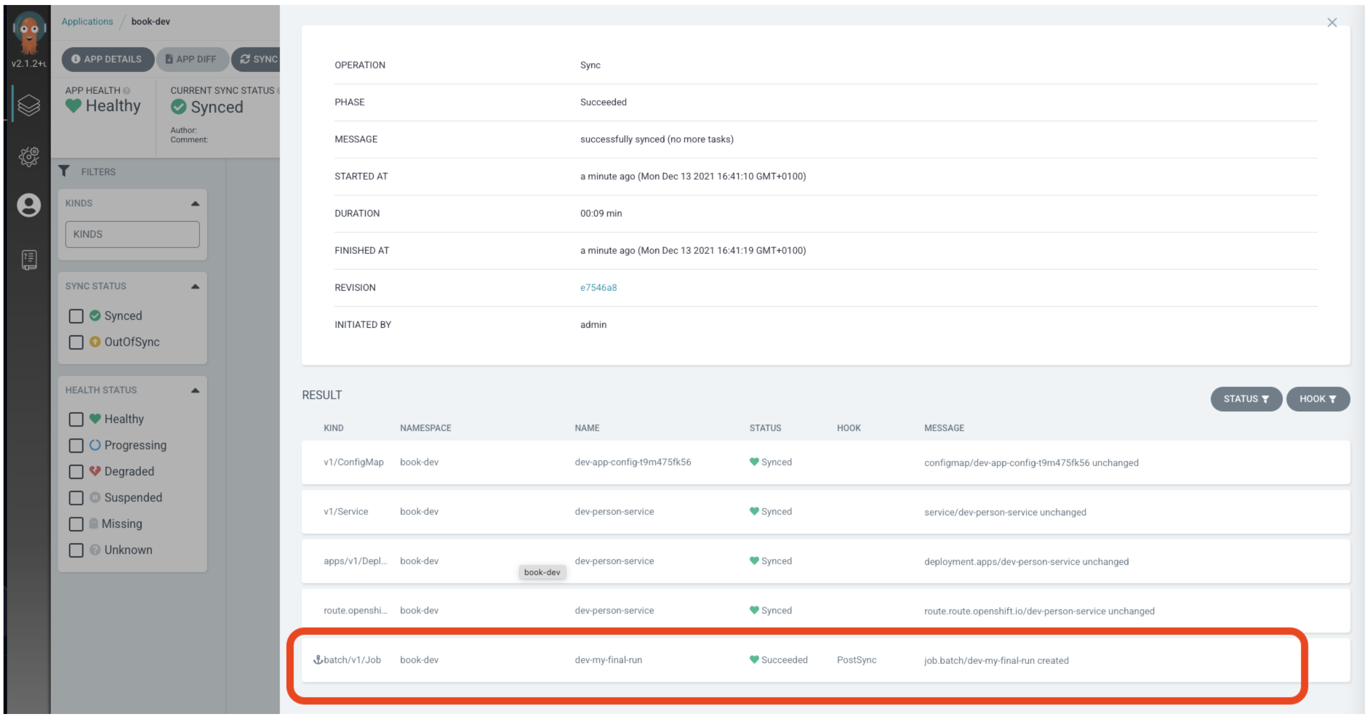
```
firstName=Kurt lastName=Cobain salutation=Mr

            else
              echo "There are already $NUM_PERSONS persons in the
database."
              http --ignore-stdin ${SERVICE_URL}
            fi
```

This hook just runs a simple job that checks to see whether there is any data in the PostgreSQL database and, if not, uses the `person-service` to create data. The only difference between this sync and a "normal" Kubernetes job is the `argocd.argoproj.io/hook` annotation.

As a result of a successful synchronization, you could also start a Tekton PipelineRun or any Kubernetes resource that is already registered in your cluster.

> **Note**: Please make sure to add your sync jobs to the base `kustomization.yaml` file. Otherwise, they won't be processed.

Image 11 shows the results of a sync.



OPERATION	Sync
PHASE	Succeeded
MESSAGE	successfully synced (no more tasks)
STARTED AT	a minute ago (Mon Dec 13 2021 16:41:10 GMT+0100)
DURATION	00:09 min
FINISHED AT	a minute ago (Mon Dec 13 2021 16:41:19 GMT+0100)
REVISION	e7546a8
INITIATED BY	admin

RESULT

KIND	NAMESPACE	NAME	STATUS	HOOK	MESSAGE
v1/ConfigMap	book-dev	dev-app-config-t9m475fk56	Synced		configmap/dev-app-config-t9m475fk56 unchanged
v1/Service	book-dev	dev-person-service	Synced		service/dev-person-service unchanged
apps/v1/Depl...	book-dev	dev-person-service	Synced		deployment.apps/dev-person-service unchanged
route.openshi...	book-dev	dev-person-service	Synced		route.route.openshift.io/dev-person-service unchanged
batch/v1/Job	book-dev	dev-my-final-run	Succeeded	PostSync	job.batch/dev-my-final-run created

Git Repository Management

In a typical enterprise with at least dozen applications, you could easily end up with a lot of Git repositories. This complexity can make it hard to manage them properly, especially in terms of security (controlling who is able to push what).

If you want to use a single configuration Git repository for all your applications and stages, please keep in mind that Git was never meant to automatically resolve merge conflicts. Instead, conflict resolution sometimes needs to be done manually. Be careful in such a case and plan your releases thoroughly. Otherwise, you might end up not being able to automatically create and merge release branches.

Managing Secrets

Another important task requiring a lot of thought is proper secret management. A secret contains access tokens to mission-critical external applications, such as databases, SAP systems, or in our case GitHub and Quay.io. You don't want to make that confidential informations publicly accessible in a Git repository.

Instead, think about a solution like Hashicorp Vault, where you are able to centrally manage your secrets.

Summary

The desire to automatically deploy the latest code into production is as old as information technology, I suppose. Automation *is* important, as it makes everybody in the release chain more productive and helps to create reproducible and well-documented deployments.

There are many ideas and even more tools out there for how to implement automation. The easiest way, of course, is to create scripts for deployment that do exactly what you need. The downside of scripts is that they might become unmanageable after some time. You also tend to do a lot of copying, and then if you want to make a change you need to find and correct each script.

With Kubernetes and GitOps, you can define everything as a file, which helps you store everything in a version control system and use the conveniences it provides.

Kubernetes bases its infrastructure on YAML files, which makes it easy to reuse what you already know as a developer or administrator: Just use Git and store the complete infrastructure description in a repository. You work through the repository to create releases, roll the latest release back if there was a bug in it, keep track of any changes, and create an automated process around deployments.

Other benefits of using Git include:

- Every commit has a description.
- Every commit could be required to be bound to an issue that was previously submitted (i.e., no commit without providing an issue number).
- Everything is auditable.
- Collaboration is facilitated and managed in a consistent manner through pull or merge requests.

This book has taken you through a tour of tools that help with every stage of development, testing, and deployment. Cloud environments such as Kubernetes and OpenShift benefit from different processes than developers traditionally used for stand-alone applications, and automation becomes a necessity in large environments and fast-changing configurations of hosts. I hope this book has helped you put together the information you had before and enable your work with DevOps and GitOps.