Chapter 2: Packaging with Helm

In this chapter, you'll learn more about working with container images. In particular, I will help you understand the concepts of Helm charts.

Modern distributions

As already mentioned in Chapter 2, the most difficult task when turning code into a useful product is creating a distributable package from the application. It is no longer just a matter of zipping all files together and putting them somewhere. We have to take care of several meta-artifacts. Some of these might not belong to the developer, but to the owner of the application.

Distribution takes place on two levels:

- *Internal* distribution: making a containerized application available to the IT team within your own organization. Even if this task requires "only" fitting into the company's CI/CD chain, this step can take some training. It is the subject of Chapter 4.
- *External* distribution: making your containerized application available for customers or other third parties. That is the subject of this chapter.

These types of distribution have much in common. In fact, before you can make my apps available for others (external distribution), you might have to put it into the local CI/CD chain (internal distribution). Kubernetes is there to automate most of the tasks.

Using an external registry with Docker or Podman

As already briefly described, in order to externally distribute an application in the modern and generally expected manner, you need a repository. The repository could be either a public image repository such as Quay.io or Docker Hub, or a private repository that is accessible by your customers. This chapter uses the Quay.io public repository for its examples, but the principles and most of the commands apply to other types of repositories as well.

You can easily get a free account on Quay.io, but it must be publicly accessible. That means everybody can read your repositories, but only you or people to whom you specifically grant permissions can write to the repositories.

The Docker and Podman build tools

Along with creating an account on Quay.io, install either Docker or Podman on your local machine. This section offers an introduction to building an image and uploading it to your repository with those tools.

Docker versus Podman and Buildah

Docker was a game-changing technology when it was first introduced in the early 2010 decade. It made containers a mainstream technology, before Google released Kubernetes.

However, Docker requires a daemon to run on the system hosting the tool, and therefore must be run with root (privileged) access. This is usually unfeasible in Kubernetes, and certainly on Red Hat OpenShift.

Podman was then invented and became a popular replacement for Docker. Podman performs basically the same tasks as Docker and has a compatible interface where almost all the commands and arguments are the same. Podman is very lightweight, and—crucially—can be run as an ordinary user account without a daemon.

However, Podman as a full Docker Desktop replacement currently runs only on GNU/Linux.

Podman internally uses Buildah to build container images. According to the official GitHub page, Buildah is the main tool for building images that conform to the Open Container Initiative (OCI) standard. The documentation states, "Buildah's commands replicate all the commands that are found in a Dockerfile. This allows building images with and without a Dockerfile, without requiring any root privileges."

To build a container image inside OpenShift (for example, using Source-to-Image (S2I) or a Tekton pipeline), you should directly use Buildah instead.

But because everything you can do with Buildah is part of Podman anyway, there is no CLI client for macOS or Windows. The documentation states to use Podman instead.

Your first build

With Docker or Podman in place, along with a repository on Quay.io or another service, check out the demo repository for this article. In `person-service/src/main/docker` you can find the file that configures builds for your application. The file is called simply `Dockerfile`. To build the demo application, based on the Quarkus Java framework, change into the top-level directory containing the demo files. Then enter the following commands, plugging in your Quay.io username and password:

Create your New Docker:

Example https://quay.io/repository/wdovey/person-service

```
$ export username=replaceme
$ docker login quay.io -u <username> -p <password>
$ cd artifacts/person-service
$ mvn clean package -DskipTests
$ docker build -f src/main/docker/Dockerfile.jvm -t
quay.io/$username/person-service .
```

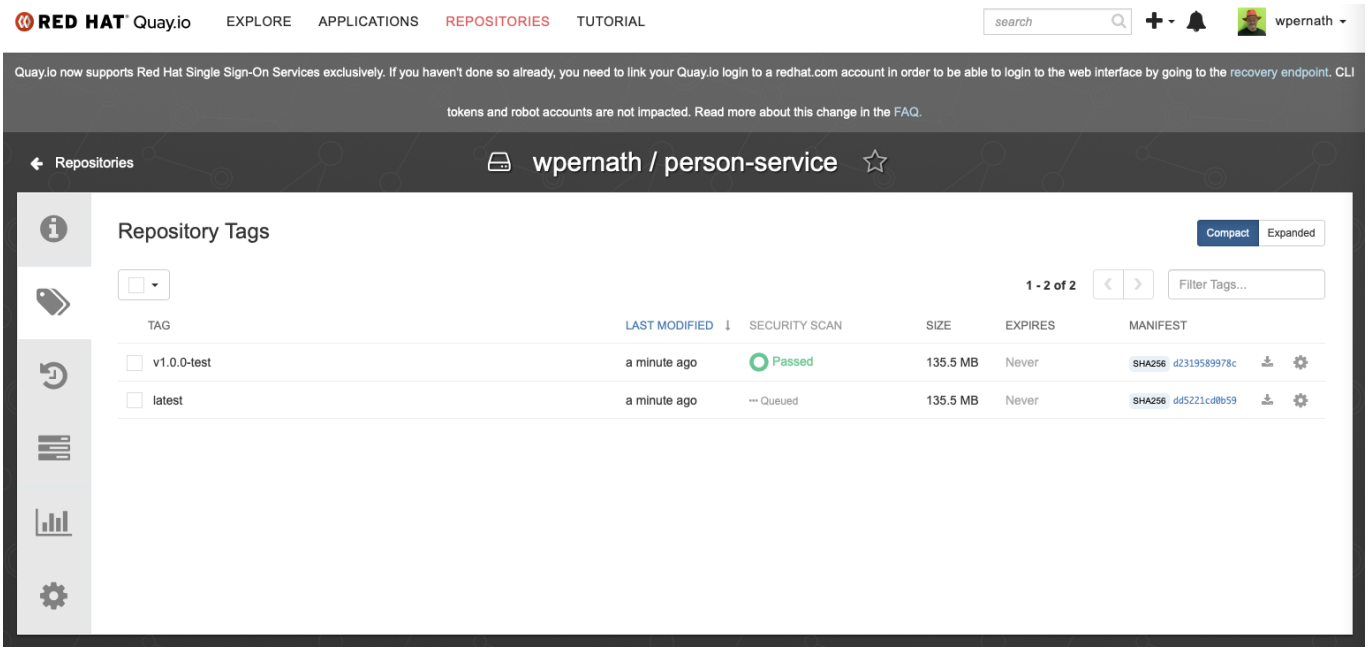
The first command logs into your Quay.io account. The second command builds the application with Maven. The third, finally, creates a Docker image out of the app. If you choose to use Podman, simply substitute `podman` for `docker` in your Docker commands, because Podman supports the same arguments. You could also make scripts that invoke Docker run with Podman instead, by aliasing the string `docker` to `podman`:

```
$ alias docker=podman
```

When your image is ready, you need to push the image to your repository:

```
$ docker push quay.io/$username/person-service
```

This will push all (`-a`) locally stored images to the external repository, including all tags (Figure 1). Now you can use the image in your OpenShift environment .



And that's it. This workflow works for all Docker-compliant registries.

Because Docker or Podman are only incidental to this chapter, I will move on to our main topics and let you turn to the many good articles out on the Internet to learn about how to build images.

Testing the image

Now that you have your image stored in Quay.io, test it to see whether everything has successfully worked out. For this, you are going to use our Kustomize example from the previous chapter.

First of all, make sure that the `artifacts/kustomize-ext/overlays/dev/kustomization.yaml` file looks like this:

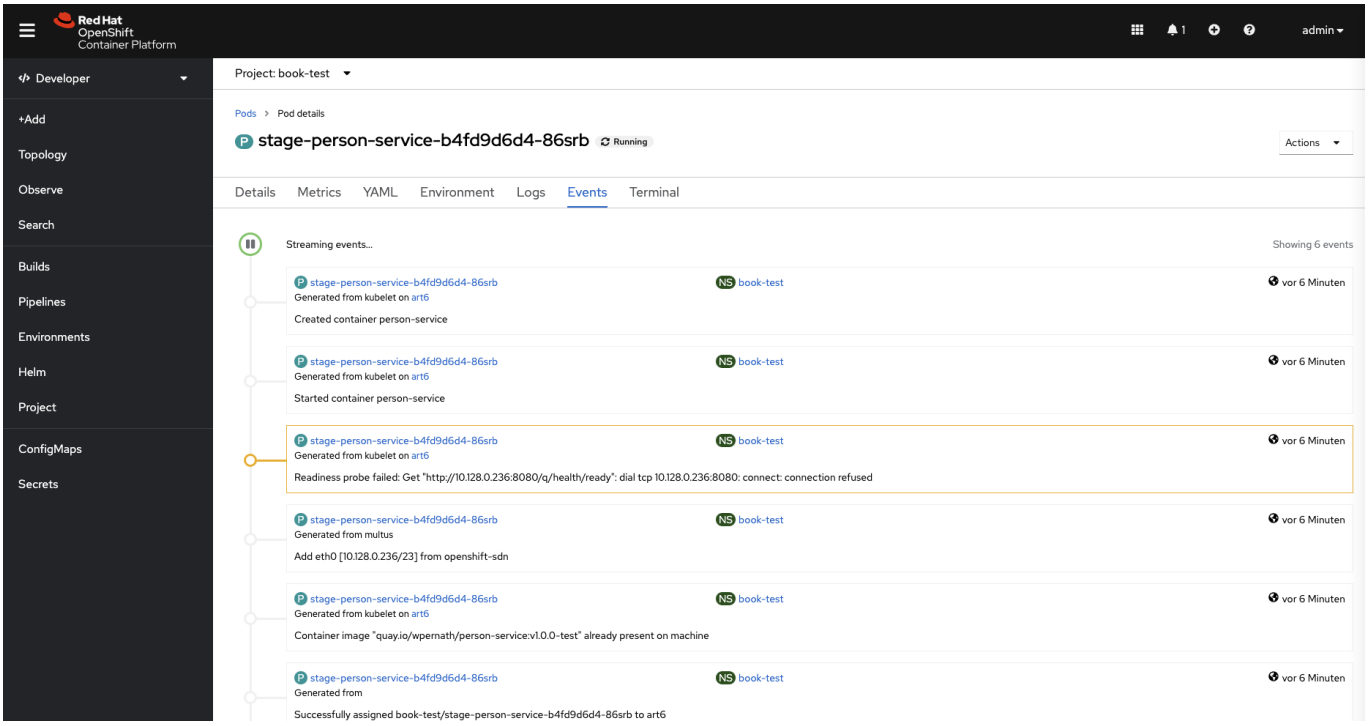
```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
- ../../base

namePrefix: dev-
commonLabels:
  variant: development


# replace the image tag of the container with latest
images:
- name: image-registry.openshift-image-registry.svc:5000/book-dev/person-
service:latest
  newName: quay.io/wpernath/person-service
```

```
    newTag: latest

  # generate a configmap
  configMapGenerator:
    - name: app-config
      literals:
        - APP_GREETING=We are in DEVELOPMENT mode

  # this patch needs to be done, because kustomize does not change
  # the route target service name
  patches:
  - patch: |-
      - op: replace
        path: /spec/to/name
        value: dev-person-service
    target:
      kind: Route
```

Then simply execute the following commands to install your application:

```
$ oc login <your openshift cluster>
$ oc new-project book-test
$ cd ../..
$ oc apply -k artifacts/kustomize-ext/overlays/dev
configmap/dev-app-config-t9m475fk56 created
service/dev-person-service created
deployment.apps/dev-person-service created
route.route.openshift.io/dev-person-service created
```

The resulting event log should look like Image 2.



If you're on any other operating system than Linux, Podman is a little bit complicated to use right now. The difficulty is not with the (CLI) tool. The `podman` CLI in most cases is identical to the `docker` CLI. Instead, the difficulty lies in installation, configuration, and integration into non-Linux operating systems.

This is unfortunate, because Podman is much more lightweight than Docker. And Podman does not require root access. So if you have some time—or are already developing your applications on Linux—try to set up Podman. If not, continue to use Docker.

Working with Skopeo

Skopeo is another command line tool that helps you work with container images and different image registries without the heavyweight Docker daemon.

You can use `skopeo` to tag an image in a remote registry:

```
$ export username=replaceme
$ skopeo copy \
   docker://quay.io/$username/person-service:latest \
   docker://quay.io/$username/person-service:v1.0.1-test
```

But you can also use Skopeo to mirror a complete repository by copying all images in one go.

Next steps after building the application

Now that your image is stored in a remotely accessible repository, you can start thinking about how to let a third party easily install your application. Two mechanisms are popular for this task: Helm charts and a Kubernetes Operator. The rest of this chapter introduces these tools.

Helm Charts

Think about Helm charts as a package manager for Kubernetes applications, like RPM or `.deb` files for Linux. Once a Helm chart is created and hosted on a repository, everyone is able to install, update, and delete your chart from a running Kubernetes installation.

Let's first have a look at how to create a Helm chart to install your application on OpenShift.

Helm has its own directory structure for storing necessary files. Helm allows you to create a basic template structure with everything you need (and even more). The following command creates a Helm chart structure for a new chart called `foo`:

```
$ helm create foo
```

But I think it's better not to create a chart from a template, but to start from scratch. To do so, enter the following commands to create a basic file system structure for your new Helm chart:

```

```

```
$ cd artifacts/
$ mv helm-chart helm-chart.bk
$ mkdir helm-chart
$ mkdir helm-chart/templates
$ touch helm-chart/Chart.yaml
$ touch helm-chart/values.yaml
```

Done. This creates the directory structure for your chart. You now have to put some basic data into `Chart.yaml`, using the following example as a guide to plugging in your own values:

```
apiVersion: v2
name: person-service
description: A helm chart for the basic person-service used as example in
the book
home: https://github.com/wpernath/book-example
type: application
version: 0.0.1
appVersion: "1.0.0"
sources:
    - https://github.com/wpernath/book-example
maintainers:
    - name: Wanja Pernath
      email: wpernath@redhat.com
```

You are now done with your first Helm chart. Of course, right now it does nothing special. You have to fill the chart with some content. So now copy the following files from the previous chapter into the `helm-chart/templates` folder:

```
$ cp kustomize-ext/base/*.yaml helm-chart/templates/
$ rm -fr helm-chart/templates/kustomization.yaml
```

The directory structure of your Helm chart now looks like this:

```
$ tree helm-chart
helm-chart
├── Chart.yaml
├── templates
│   ├── config-map.yaml
│   ├── deployment.yaml
│   ├── route.yaml
│   └── service.yaml
└── values.yaml

1 directory, 6 files
```

Packaging, installing, upgrading and rolling back your Helm chart

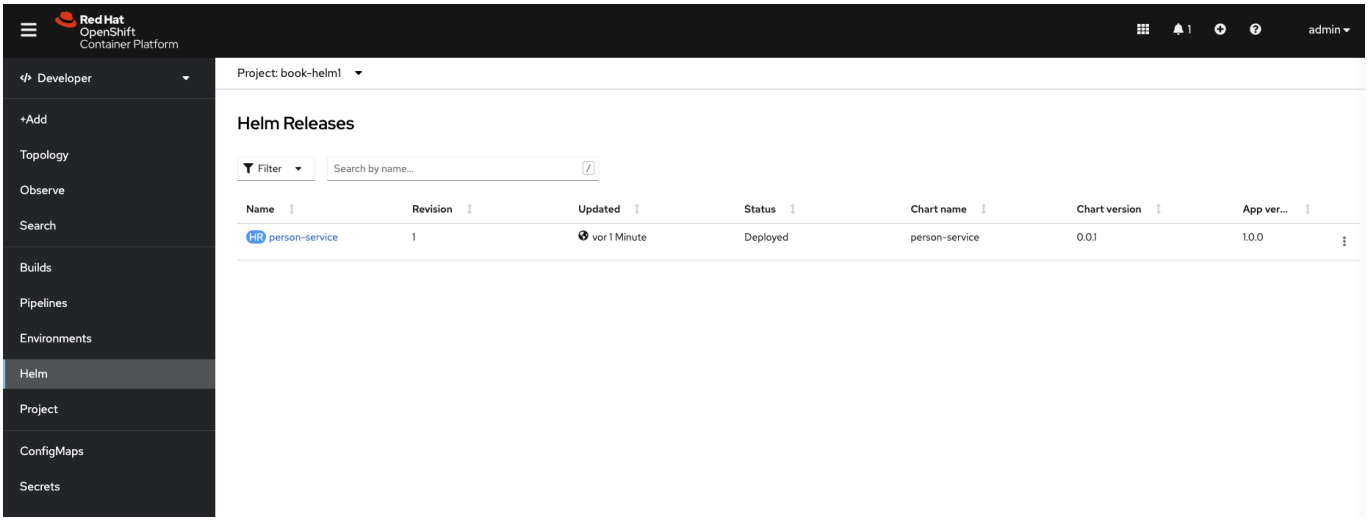
Now that you have a very simple Helm chart, you can package it:

```
$ helm package helm-chart
Successfully packaged chart and saved it to: person-service-0.0.1.tgz
```

And the following command installs the Helm chart into a newly created OpenShift project called `book-helm1`:

```
$ oc new-project book-helm1
$ helm install person-service person-service-0.0.1.tgz
NAME: person-service
LAST DEPLOYED: Mon Oct 25 17:10:49 2021
NAMESPACE: book-helm1
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

If you now go to the OpenShift console, you should see the Helm release (Image 3).



You can get the same overview via the command line. The first command that follows shows a list of all installed Helm charts in your namespace. The second command provides the update history of a given Helm chart.

```
$ helm list
$ helm history person-service
```

If you create a newer version of the chart, upgrade the release in your Kubernetes namespace by entering:

```
$ helm upgrade person-service person-service-0.0.5.tgz
Release "person-service" has been upgraded. Happy Helming!
```

```
NAME: person-service
LAST DEPLOYED: Tue Oct 26 19:15:07 2021
NAMESPACE: book-helm1
STATUS: deployed
REVISION: 7
TEST SUITE: None
```

The `rollback` argument helps you return to a specified revision of the installed chart. First get the history of the installed chart by entering:

```
$ helm history person-service
REVISION    UPDATED                     STATUS      CHART
APP VERSION DESCRIPTION
1           Tue Oct 26 19:24:10 2021    superseded  person-service-0.0.5
v1.0.0-test Install complete
2           Tue Oct 26 19:24:46 2021    deployed    person-service-0.0.4
v1.0.0-test Upgrade complete
```

Then roll back to revision 1 by entering:

```
$ helm rollback person-service 1
Rollback was a success! Happy Helming!
```

The history shows you that the rollback was successful:

```
$ helm history person-service
REVISION    UPDATED                     STATUS      CHART
APP VERSION DESCRIPTION
1           Tue Oct 26 19:24:10 2021    superseded  person-service-0.0.5
v1.0.0-test Install complete
2           Tue Oct 26 19:24:46 2021    superseded  person-service-0.0.4
v1.0.0-test Upgrade complete
3           Tue Oct 26 19:25:48 2021    deployed    person-service-0.0.5
v1.0.0-test Rollback to 1
```

If you are not satisfied with a given chart, you can easily uninstall it by running:

```
$ helm uninstall person-service
release "person-service" uninstalled
```

New content for the chart

Another nice feature to use with Helm charts is a `NOTES.txt` file in the `helm-chart/templates` folder. This file will be shown right after installation of the chart. And it's also available via the OpenShift (UI) in **Developer→Helm→Release Notes**). Basically, you can enter your release notes there as a Markdown-formatted text file like the following. The nice thing is that you can insert named parameters defined in the `values.yaml` file:

```
# Release NOTES.txt of person-service helm chart
- Chart name: {{ .Chart.Name }}
- Chart description: {{ .Chart.Description }}
- Chart version: {{ .Chart.Version }}
- App version: {{ .Chart.AppVersion }}

## Version history
- 0.0.1 Initial release
- 0.0.2 release with some fixed bugs
- 0.0.3 release with image coming from quay.io and better parameter
substitution
- 0.0.4 added NOTES.txt and a configmap
- 0.0.5 added a batch Job for post-install and post-upgrade
```

Parameters? Yes, of course. Sometimes you need to replace standard settings, as we did with the OpenShift Templates or Kustomize. To unpack the use of parameters, let's have a closer look into the `values.yaml` file:

```
deployment:
    image: quay.io/wpernath/person-service
    version: v1.0.0-test
    replicas: 2
    includeHealthChecks: false

config:
    greeting: 'We are on a newer version now!'
```

You are just defining your variables in the file. The following YAML configuration file illustrates how to insert the variables, through curly braces:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_GREETING: |-
    {{ .Values.config.greeting | default "Yeah, it's openshift time" }}
```

This ConfigMap defines a parameter named `APP_GREETING` with the content of the variable `.Values.config.greeting` (the initial period must be present). If this variable is not defined, the default

will be used.

Helm also defines some built-in objects with predefined parameters:

- `.Release` can be used after the Helm chart is installed in a Kubernetes environment. The parameter defines variables such as `.Name` and `.Namespace`.
- `.Capabilities` provides information about the Kubernetes cluster where the chart is installed.
- `.Chart` provides access to the content of the `Chart.yaml` file. Any data in that file is accessible.
- `.Files` provides access to all non-special files in a chart. You can't use this parameter to access template files, but you can use it to read and parse other files in your chart, for example to generate the contents of a ConfigMap.

Because Helm's templating engine is an implementation of the templating engine of the Go programming language, you also have access to functions and flow control. For example, if you want to write only certain parts of the `deployment.yaml` file, you can do something like:

```
{{- if .Values.deployment.includeHealthChecks }}
<do something here>
{{- end }}
```

Do you see the hyphen (`-`) at the beginning of the reference? This is necessary to get a well formatted YAML file after the template has been processed. If you removed the `-`, you would have empty lines in the YAML.

Debugging Templates

Typically, after you execute `helm install`, all the generated files are sent directly to Kubernetes. A couple commands help you debug your templates first.

- The `helm lint` command checks whether your chart is following best practices.
- The `helm install --dry-run --debug` command renders your files without sending them to Kubernetes.

Defining a hook

It is often valuable to include sample data with an application for mocking and testing. But if you want to install a database with example data as part of your Helm chart, you need to find a way of initializing the database. This is where Helm hooks come into play.

Basically, a hook is just another Kubernetes resource (such as a job or a pod), which gets executed when a certain event is triggered. An event could take place at one of the following points:

- pre-install
- post-install
- pre-upgrade
- post-upgrade
- pre-delete
- post-delete

- pre-rollback
- post-rollback

The type of hook gets configured via the `helm.sh/hook` annotation. This annotation allows you to assigns weights to hooks in order to specify the order in which they run. Lower-numbered weights run before higher-number ones, for each type of event.

The following listing defines a new hook as a Kubernetes `Job` with `post-install` and `post-upgrade` triggers:

```
apiVersion: batch/v1
kind: Job
metadata:
    name: "{{ .Release.Name }}"
    labels:
        app.kubernetes.io/managed-by: {{ .Release.Service | quote }}
        app.kubernetes.io/instance: {{ .Chart.Name | quote }}
        app.kubernetes.io/version: {{ .Chart.AppVersion }}
        "helm.sh/chart": "{{ .Chart.Name }}-{{ .Chart.Version }}"
    annotations:
        # This is what defines this resource as a hook. Without this line,
the
        # job is considered part of the release.
        "helm.sh/hook": post-install,post-upgrade
        "helm.sh/hook-weight": "-5"
        "helm.sh/hook-delete-policy": before-hook-creation
spec:
    template:
        metadata:
            name: {{ .Chart.Name }}
            labels:
                "helm.sh/chart": "{{ .Chart.Name }}-{{ .Chart.Version }}"
        spec:
            restartPolicy: Never
            containers:
            - name: post-install-job
              image: "registry.access.redhat.com/ubi8/ubi-minimal:latest"
              command:
              - /bin/sh
              - -c
              - |
                echo "WELCOME TO '{{ .Chart.Name }}-{{ .Chart.Version }}'
"
                echo "-------------------------------------------------"
                echo "Here we could now do initialization work."
                echo "Like filling our DB with some data or what's so
ever"
                echo "..."

                sleep 10
```

As the example shows, you can also execute a script of your design as part of the install process.

Subcharts and CRDs

In Helm, you can define subcharts. Whenever your chart gets installed, all dependent subcharts are installed as well. Just put required subcharts into the `helm/charts` folder. This could be quite handy if your application requires the installation of a database or other dependent components.

Subcharts must be installable without the main chart. This means that each subchart has its own `values.yaml` file. You can override the values in the subchart's file within your main chart's `values.yaml`.

If your chart requires the installation of a custom resource definition (CRD)—for example, to install an Operator—simply put the CRD into the `helm/crds` folder of your chart. Keep in mind that Helm does *not* take care of deinstalling any CRDs if you want to deinstall your chart. So installing CRDs with Helm is a one-shot operation.

Summary of Helm charts

Creating a Helm chart is quite easy and mostly self-explaining. Features such as hooks help you do some initialization after installation.

If Helm charts do so much, so well, why would you need another package format? Well, let's next have a look at Operators.

Summary

This chapter has described how to build images. You've learned more about the various command-line tools (`skopeo`, `podman`, `buildah` etc.). You also saw how to create a Helm Chart. And you should know how to decide when to use each tool.

The next chapter of this book will talk about Tekton pipelines as a form of internal distribution.