Published by Red Hat Developer

Getting GitOps

A practical platform with OpenShift, Argo CD, and Tekton

Wanja Pernath

Foreword by Florian Heubeck, Principal Engineer at MediaMarktSaturn Technology & Java User Group Ingolstadt Leader

Contents

Contents

Contents

Foreword

The job of a software developer has never been as approachable as it is nowadays. There are plenty of free resources provided by great communities to start from, allowing you to achieve quick successes with only little effort. But only a few of us remain in the role of a pure programmer, and young professionals face different challenges from those we encountered one to two decades ago. The DevOps mindset and company restructurings made us all software engineers.

The term "software engineer" is not just a rebranding of these creepy, nocturnal keyboard maniacs. It mainly means two things. First, building software is established as an engineering discipline (one that must be learned). Second, it's not just about producing source code, but also creating and operating complex systems composed of the right components for every purpose.

That's where two of the most exciting technologies of our time come in. No, I'm not talking about virtual backgrounds in video conferences and the unmute button. I'm talking about Quarkus and Kubernetes, of course! Quarkus satisfies both a developer's needs and Kubernetes's demands, and it enables actual software engineering in the cloud.

Although Quarkus prepares Java ecosystem inhabitants well for the cloud, there's some more of the "operations part" to consider (remember: engineering, not only developing). In regards to operations, another "thing" has become quite popular—at least as a keyword, but considered closely, is that the most natural operating model for Kubernetes: GitOps [1].

There were (and still are) many discussions and sophisticated concepts about how to solve continuous builds and deployments (CI/CD). Yes, that's a very important topic, but honestly—in the world of Kubernetes—it's a piece of cake. Building, testing, and integrating software modules is straightforward, especially when targeting containers. And continuous delivery? That can be really complicated, I admit—that's why we actually don't do it by ourselves.

We've known "Infrastructure as Code" for some time; it has become common practice in most cloud environments. With GitOps, we do the same regarding application deployment and configuration. The desired state of Kubernetes resources is described declaratively and versioned in Git. Operators running in the target environment are continuously validating and reconciling this state. The process of continuous delivery is fully automated by specialized components such as the projects from the Cloud Native Computing Foundation (CNCF) [2], Argo CD and Flux CD, and of course, the built-ins of OpenShift.

How all of this nicely plays together, peppered with lots of examples, can be found in this book. When my team adopted Kubernetes, GitOps wasn't widely known yet. Sticking to the principles of GitOps fostered the transition toward independent engineering teams and increased the overall quality of our solutions. Since then, we have helped many teams succeed with the same journey. This job becomes much easier now that we can simply hand out the book you're looking at now. So take Wanja's red pill and follow him down the rabbit hole of modern software delivery.

—Florian Heubeck
Principal Engineer at MediaMarktSaturn Technology
Java User Group Ingolstadt Leader
March 2022

[1]: https://opengitops.dev/

[2]: https://www.cncf.io/

Introduction

During my day-to-day job as a Technical Enablement Manager at Red Hat, I frequently explain and demonstrate to interested developers and architects the benefits of using Red Hat OpenShift and modern Kubernetes platforms for their projects. I started writing a series of blog posts [1] about it to record what I was sharing in my classes.

Then I realized that so many of these concepts require more explanation. Students tell me they're overwhelmed with all the fast-moving news around Kubernetes and OpenShift. They don't understand all the technologies involved or the benefits of using them in their projects. They wish to have a practical guide taking them through the whole story. So I decided not just to talk about that, but also to write a book about it.

As there are a lot of books available to explain the theoretical justification for the benefits of GitOps and Kubernetes, I decided to start from the other side: I have a use case and want to go through it from the beginning to the end.

What to Expect From This Book

You might already know about GitOps. You might understand the benefits of using Kubernetes from a developer's perspective because it allows you to set up deployments using declarative configurations. You might also be familiar with container images, Tekton Pipelines, or Argo CD.

But have you already used all of these technologies together in one project? Have you already gotten your hands dirty with Tekton? With Argo CD?

If you want to delve down into GitOps, you might benefit from somebody giving you some hints here and there.

That is why I wrote this book. We'll do GitOps from scratch.

The Use Case

Typically, most modern software projects require you to design and implement one or more services. This service could be a RESTful microservice or a reactive front end. To provide an example that will be meaningful to a large group of readers, I have decided to write a RESTful microservice called `person-service`, which requires a third-party software stack (in my case, a PostgreSQL database).

This service has API methods to read data from the database and update, create, and delete data using JSON. Thus, it's a simple CRUD service (create, read, update, and delete).

Chapter Overview

This book proceeds the way you would when developing your own services. The initial question is always (after understanding the business requirements, of course), what software stack to use. In my case, I have decided to use the Java language with the Quarkus [2] framework. **Chapter 1** explains the benefits of Quarkus and shows you how to use it to develop your microservice.

Once you've developed your code, you need to understand how to move it to your target platform. That is the subject of **Chapter 2**: Understanding container images and all of the Kubernetes manifest files, and easily modifying them for later use.

After you've decided on your target platform, you might also want to determine how to distribute your application. This task includes packaging that application, which in turn involves choosing the right package format. **Chapter 3** discusses possible solutions in detail, with examples.

Once you understand how to package and distribute your application, we will set up a process to automate the tasks of building the sources and deploying your service to your test and production stages. **Chapter 4** explains how to use Tekton to set up an integration pipeline for your service.

And finally, **Chapter 5** sets up GitOps for your service.

Acknowledgments

I started writing a series of blog posts on automated application packaging and distribution with Red Hat OpenShift back in January 2021. I initially wanted to have some notes to support my day-to-day work.

The positive feedback I got from readers around the world motivated me to continue writing on this topic. Then one day, my manager asked me, *Wouldn't it be great to create a book out of these posts?* And *Getting GitOps* was born.

I want to thank all of the people who helped make this possible:

- My manager, Günter Herold, who came up with the idea for the book.
- Hubert Schweinesbein, who gave the project the final okay.
- Markus Eisele, who helped me find the right contacts.
- Florian Heubeck, for writing the foreword.
- My editor, Andrew Oram, for putting my words into proper and understandable English.
- My girlfriend, for her patience.
- My dad, for believing in me.
- The cats, for helping me write—and for all their "feedback."
- Finally, thanks to all of you for reading.

Summary

This book aims to guide your journey to GitOps with OpenShift and Kubernetes.

Thank you for reading.

References

[1] https://www.opensourcerers.org/2021/04/26/automated-application-packaging-and-distribution-with-openshift-basic-development-principles-part-14/

[2] https://quarkus.io

About the Examples

All of the examples in this book are available on GitHub: https://github.com/wpernath/book-example [1]

Prerequisites

You will need the following software:

- Red Hat OpenShift 4.8.x (see the next section for instructions)
- Maven 3.8.3
- Java JDK 11 or later
- git
- Docker
- The OpenShift client (`oc`) matching the version of the OpenShift cluster
- An editor to work with, such as Visual Studio Code, Eclipse, or IntelliJ
- OpenShift also needs to have the following Operators installed:
- OpenShift GitOps
- OpenShift Pipelines
- Crunchy Postgres for Kubernetes by Crunchy Data

How to Get an OpenShift Instance

This section describes the options you have for using OpenShift.

Using the Developer Sandbox for Red Hat OpenShift

This solution is the easiest one, but unfortunately limited: You can't create new projects (namespaces), and you're not allowed to install additional operators. You can use this solution for Chapters 1 and 2 and the Helm chart section of Chapter 3.

To use this option, go to the Developer Sandbox for Red Hat OpenShift [2] and register for free.

Using OpenShift Local

Red Hat OpenShift Local (formerly Red Hat CodeReady Containers) provides a single-node OpenShift installation for Windows, macOS, and Linux. It runs OpenShift on an embedded virtual machine. You have all the flexibility of an external OpenShift cluster without the need for three or more master nodes. You also can install additional Operators.

This solution requires the following resources on your local machine:

- 9GB free memory
- 4 CPU cores
- 50GB free hard disk space

Go to GitHub [3] for a list of releases and have a look at the official documentation [4].

Using Single Node OpenShift (SNO)

With this solution, you have the most flexibility in using your OpenShift installation. But this choice also requires most resources. You should have a dedicated spare machine with the following specs to use SNO:

- 8 vCPU cores
- 32GB free memory
- 150GB free hard disk space

Visit the Red Hat Console [5] to start the installation process. After installation, look at my OpenShift Config script [6], which you can find on GitHub. This script creates persistent volumes, makes the internal registry non-ephemeral, creates a cluster-admin user, and installs necessary operators and a CI environment with Nexus (a maven repository) and Gogs (a Git repository).

The Container Image Registry

You can use any Docker-compliant registry to store your container images. I am using Quay.io for all of my images. The account is available for free and doesn't limit upload/download rates. Once registered, go to Account Settings → User Settings and generate an encrypted password. Quay.io will give you some options to store your password hash, such as a Kubernetes-secret, which you can then directly use as push-/pull secrets.

The free account, however, limits you to creating only public repositories. As a result, anybody can read from your repository, but only you are allowed to write and update your image.

Once you've created your image in the repository, you have to check the image properties and make sure that the repository is public. By default, Quay.io [7] creates private repositories.

How the Examples are Structured

You can find all of the examples on GitHub: https://github.com/wpernath/book-example [8]

Fork the repository and use it as you desire.

Chapter 1

The folder `person-service` contains the Java sources of the Quarkus example. If you want to deploy it on OpenShift, make sure to first install a PostgreSQL server, either via Crunchy Data or by instantiating the `postgresql-persistent` template as follows:

```
$ oc new-app postgresql-persistent \
    -p POSTGRESQL_USER=wanja \
    -p POSTGRESQL_PASSWORD=wanja \
    -p POSTGRESQL_DATABASE=wanjadb \
    -p DATABASE_SERVICE_NAME=wanjaserver
```

Chapter 2

This chapter introduces Kubernetes configuration and the Kustomize tool. The folders containing the configuration files used in this chapter follow:

- **`raw-kubernetes`** contains the raw Kubernetes manifest files.
- **`ocp-template`** contains the OpenShift Template file.
- **`kustomize`** contains a set of basic files for use with Kustomize.
- **`kustomize-ext`** contains a set of advanced files for use with Kustomize.

Chapter 3

This chapter is about Helm Charts and Kubernetes Operators. The corresponding folders are **`helm-chart`** and **`kube-operator`**.

Chapter 4

This chapter is about Tekton and OpenShift Pipelines. The sources can be found in the folder **`tekton`**. Also have a look at the **`pipeline.sh`** script, which installs all the necessary Tasks and resources if you call it with the **`init`** parameter:

```
$ pipeline.sh init
configmap/maven-settings configured
persistentvolumeclaim/maven-repo-pvc configured
persistentvolumeclaim/builder-pvc configured
task.tekton.dev/kustomize configured
task.tekton.dev/maven-caching configured
pipeline.tekton.dev/build-and-push-image configured
```

You can start the pipeline by executing:

```
$ pipeline.sh start -u wpernath -p <your-quay-token>
pipelinerun.tekton.dev/build-and-push-image-run-20211125-163308 created
```

Chapter 5

This chapter is about using Tekton and Argo CD. The sources can be found in the **`gitops`** folder. To initialize these tools, call:

```
$ ./pipeline.sh init [--force] --git-user <user> \
    --git-password <pwd> \
    --registry-user <user> \
    --registry-password <pwd>
```

This call (if given the **`--force`** flag) creates the following namespaces and Argo CD applications:

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

The following command starts the development pipeline discussed in Chapter 5:

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

Whenever the pipeline is successfully executed, you should see an updated message on the **`person-service-config`** Git repository.

You should also see that Argo CD has initiated a synchronization process, which ends with a redeployment of the Quarkus application.

To start the staging pipeline, call:

```
$ ./pipeline.sh stage -r v1.0.1-testing
```

This creates a new branch in Git called **`release-v1.0.1-testing`**, uses the current dev image, tags it on Quay.io, and updates the **`stage`** config in Git.

In order to apply the changes, you need to either merge the branch directly or create a pull request and then merge the changes.

References

[1] https://github.com/wpernath/book-example

[2] https://developers.redhat.com/developer-sandbox

[3] https://github.com/code-ready/crc/releases

[4] https://access.redhat.com/documentation/en-us/red_hat_codeready_containers/1.33/html-single/getting_started_guide/index

[5] https://console.redhat.com/openshift/assisted-installer/clusters/~new

[6] https://github.com/wpernath/openshift-config

[7] https://quay.io/

[8] https://github.com/wpernath/book-example

Chapter 1

Creating the Sample Service

This book needs a good example to demonstrate the power of working with Kubernetes and Red Hat OpenShift. Because it's an application that will be familiar and useful to most readers, we'll create a REST-based microservice in Java that reads data from and writes data to a database.

I have been coding for over two decades with Java and Java Enterprise Edition (Java EE). I have used most of the frameworks out there. (Does anyone remember Struts or JBoss Seam or SilverStream?) I've even created code generators to make my life easier with Enterprise JavaBeans (EJB) (1.x and 2.x). Those frameworks and ideas tried to minimize development effort, but they all had drawbacks.

Then, back in 2020, when I thought that there was nothing out that could really, positively surprise me, I had a look at Quarkus [1.1]. It enchanted me because it provided interfaces to all the common open source tools for containerization and cloud deployment. What's more, it included a dev mode that took away the tedious compilation tasks.

Note: For a thorough introduction to Quarkus, check out the e-book *Quarkus for Spring Developers** [1.2] from Red Hat Developer.

First Steps

Quarkus has a Get Started [1.3] page. Go there to have a look at how to install the command-line interface (CLI) tool, which is called `quarkus`.

After you've installed `quarkus`, create a new project by executing:

```
$ quarkus create app org.wanja.demo:person-service:1.0.0
Looking for the newly published extensions in registry.quarkus.io
-----------

applying codestarts...
📚 java
🔨 maven
📦 quarkus
📝 config-properties
🔧 dockerfiles
🔧 maven-wrapper
🚀 resteasy-codestart

-----------
[SUCCESS] ✅  quarkus project has been successfully generated in:
--> /Users/wpernath/Devel/quarkus/person-service
-----------
Navigate into this directory and get started: quarkus dev
```

A successful initialization creates an initial Maven project with the following structure:

```
$ tree
.
|── README.md
|── mvnw
|── mvnw.cmd
|── pom.xml
|── src
    ├── main
    │   ├── docker
    │   │   ├── Dockerfile.jvm
    │   │   ├── Dockerfile.legacy-jar
    │   │   ├── Dockerfile.native
    │   │   └── Dockerfile.native-distroless
    │   ├── java
    │   │   └── org
    │   │       └── wanja
    │   │           └── demo
    │   │               └── GreetingResource.java
    │   └── resources
    │       ├── META-INF
    │       │   └── resources
    │       │       └── index.html
    │       └── application.properties
    └── test
        └── java
            └── org
                └── wanja
                    └── demo
                        ├── GreetingResourceTest.java
                        └── NativeGreetingResourceIT.java

15 directories, 13 files
```

If you want to test what you have done so far, call:

```
$ mvn quarkus:dev
```

Or if you prefer to use the Quarkus CLI tool, you can also call:

```
$ quarkus dev
```

These commands compile all the sources and start the development mode of your project, where you don't need to specify any runtime environment (Tomcat, Red Hat JBoss, etc.).

Let's have a look at the generated `GreetingResource.java` file, which you can find under `src/main/java/org/wanja/demo`:

```
package org.wanja.demo;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/hello")
public class GreetingResource {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return "Hello RESTEasy";
    }
}
```

If `quarkus:dev` is running, you should have an endpoint reachable at `localhost:8080/hello` in a browser on that system. Let's have a look. For testing REST endpoints, you can use `curl` or a newer client called httpie [1.4]. Here I used httpie:

```
$ http :8080/hello
HTTP/1.1 200 OK
Content-Type: text/plain;charset=UTF-8
content-length: 14

Hello RESTEasy
```

Let's go a little bit deeper. We'll change the string `Hello RESTEasy` and call the service again (but without restarting `quarkus dev`—that's a key point to make).

```
$ http :8080/hello
HTTP/1.1 200 OK
Content-Type: text/plain;charset=UTF-8
content-length: 7

Hi Yay!
```

OK, this is getting interesting now. You don't have to recompile or restart Quarkus to see your changes in action.

Because it's limiting to put hard-coded strings directly into Java code, let's switch to feeding in the strings from a configuration file, as described in the Quarkus documentation about configuring your application [1.5]. To reconfigure the application, open `src/main/resources/application.properties` in your preferred editor and create a new property. For example:

```
app.greeting=Hello, dear quarkus developer!
```

Then go into the `GreetingResource` and create a new property on the class level:

```
package org.wanja.demo;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import org.eclipse.microprofile.config.inject.ConfigProperty;

@Path("/hello")
public class GreetingResource {

    @ConfigProperty(name="app.greeting")
    String greeting;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return greeting;
    }
}
```

Test your changes by calling the REST endpoint again:

```
$ http :8080/hello
HTTP/1.1 200 OK
Content-Type: text/plain;charset=UTF-8
content-length: 25

Hello, quarkus developer!
```

Again, you haven’t recompiled or restarted the services. Quarkus is watching for any changes in the source tree and takes the required actions automatically.

This is already great. Really. But let’s move on.

Creating a Database Client

The use case for this book should be richer than a simple hello service. We want to have a database client that reads from and writes to a database. After reading the corresponding documentation [1.6], I decided to use Panache here, as it dramatically reduces the work I have to do.

First, you need to add the required extensions to your project. The following command installs a JDBC driver for PostgreSQL and everything to be used for ORM:

```
$ quarkus ext add quarkus-hibernate-orm-panache quarkus-jdbc-postgresql
Looking for the newly published extensions in registry.quarkus.io
[SUCCESS] ✅ Extension io.quarkus:quarkus-hibernate-orm-panache has been installed
[SUCCESS] ✅ Extension io.quarkus:quarkus-jdbc-postgresql has been installed
```

Java Code for Database Operations

The next step is to create an entity. We’ll call it `Person`, so you’re going to create a `Person.java` file.

```
package org.wanja.demo;

import javax.persistence.Column;
import javax.persistence.Entity;

import io.quarkus.hibernate.orm.panache.PanacheEntity;

@Entity
public class Person extends PanacheEntity {
    @Column(name=”first_name”)
    public String firstName;

    @Column(name=”last_name”)
    public String lastName;

    public String salutation;
}
```

According to the docs, this should define the `Person` entity, which maps directly to a `person` table in our PostgreSQL database. All public properties will be mapped automatically to the corresponding entity in the database. If you don’t want that, you need to specify the `@Transient` annotation.

You also need a **`PersonResource`** class to act as a REST endpoint. Let's create that simple class:

```
package org.wanja.demo;

import java.util.List;

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import io.quarkus.panache.common.Sort;

@Path("/person")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class PersonResource {

    @GET
    public List<Person> getAll() throws Exception {
        return Person.findAll(Sort.ascending("last_name")).list();
    }
}
```

Right now, this class has exactly one method, **`getAll()`**, which simply returns a list of all persons sorted by the **`last_name`** column.

Enabling the Database

Next, we need to tell Quarkus that we want to use a database. Then we need to find a way to start a PostgreSQL database locally. But one step at a time.

Open the **`application.properties`** file and add some properties there:

```
quarkus.hibernate-orm.database.generation=drop-and-create
quarkus.hibernate-orm.log.format-sql=true
quarkus.hibernate-orm.log.sql=true
quarkus.hibernate-orm.sql-load-script=import.sql

quarkus.datasource.db-kind=postgresql
```

And then make a simple SQL import script to fill some basic data into the database. Create a new file called **`src/main/resources/import.sql`** and put the following lines in there:

```
insert into person(id, first_name, last_name, salutation) values
  (nextval('hibernate_sequence'), 'Doro', 'Pesch', 'Ms');
insert into person(id, first_name, last_name, salutation) values
  (nextval('hibernate_sequence'), 'Bobby', 'Brown', 'Mr');
insert into person(id, first_name, last_name, salutation) values
  (nextval('hibernate_sequence'), 'Kurt', 'Cobain', 'Mr');
insert into person(id, first_name, last_name, salutation) values
  (nextval('hibernate_sequence'), 'Nina', 'Hagen', 'Mrs');
insert into person(id, first_name, last_name, salutation) values
  (nextval('hibernate_sequence'), 'Jimi', 'Henrix', 'Mr');
insert into person(id, first_name, last_name, salutation) values
  (nextval('hibernate_sequence'), 'Janis', 'Joplin', 'Ms');
```

```
insert into person(id, first_name, last_name, salutation) values
  (nextval(‘hibernate_sequence’), ‘Joe’, ‘Cocker’, ‘Mr’);
insert into person(id, first_name, last_name, salutation) values
  (nextval(‘hibernate_sequence’), ‘Alice’, ‘Cooper’, ‘Mr’);
insert into person(id, first_name, last_name, salutation) values
  (nextval(‘hibernate_sequence’), ‘Bruce’, ‘Springsteen’, ‘Mr’);
insert into person(id, first_name, last_name, salutation) values
  (nextval(‘hibernate_sequence’), ‘Eric’, ‘Clapton’, ‘Mr’);
```

You can now restart **`quarkus dev`** with everything you need:

```
$ quarkus dev
2021-12-15 13:39:47,725 INFO  [io.qua.dat.dep.dev.DevServicesDatasourceProcessor]
  (build-26) Dev Services for the default datasource (postgresql) started.
Hibernate:

    drop table if exists Person cascade
__  ____  __  _____   ___  __ ____  ______
 --/ __ \/ / / / _ | / _ \/ //_/ / / / __/
 -/ /_/ / /_/ / __ |/ , _/ ,< / /_/ /\ \
--\___\_\____/_/ |_/_/|_/_/|_|\____/___/
2021-12-15 13:39:48,869 WARN  [org.hib.eng.jdb.spi.SqlExceptionHelper]
  (JPA Startup Thread: <default>) SQL Warning Code: 0, SQLState: 00000

2021-12-15 13:39:48,870 WARN  [org.hib.eng.jdb.spi.SqlExceptionHelper]
  (JPA Startup Thread: <default>) table “person” does not exist, skipping
Hibernate:

    drop sequence if exists hibernate_sequence
2021-12-15 13:39:48,872 WARN  [org.hib.eng.jdb.spi.SqlExceptionHelper]
  (JPA Startup Thread: <default>) SQL Warning Code: 0, SQLState: 00000
2021-12-15 13:39:48,872 WARN  [org.hib.eng.jdb.spi.SqlExceptionHelper]
  (JPA Startup Thread: <default>) sequence “hibernate_sequence” does not exist,
   skipping
Hibernate: create sequence hibernate_sequence start 1 increment 1
Hibernate:

    create table Person (
       id int8 not null,
        first_name varchar(255),
        last_name varchar(255),
        salutation varchar(255),
        primary key (id)
    )

Hibernate:
    insert into person(id, first_name, last_name, salutation) values
      (nextval(‘hibernate_sequence’), ‘Doro’, ‘Pesch’, ‘Mrs’)
Hibernate:
    insert into person(id, first_name, last_name, salutation) values
      (nextval(‘hibernate_sequence’), ‘Bobby’, ‘Brown’, ‘Mr’)
```

The first time I started Quarkus, I expected exceptions because no PostgreSQL database was installed locally on my laptop. And yet, no exception upon startup. How could that be?

Quarkus Dev Services

Every developer has faced situations where they wanted to quickly test some new feature or fix a bug in an existing application. The workflow is mostly the same:

- Set up the local IDE.
- Clone the source code repository.
- Check dependencies for databases or other infrastructure software components.
- Install the dependencies locally (a Redis server, an Infinispan server, a database, ApacheMQ, or whatever is needed).
- Make sure everything is set up correctly.
- Creating and implementing the bug fix or the feature.

In short, it takes quite some time before you actually start implementing what you have to implement.

This is where Quarkus Dev Services come into play. As soon as Quarkus detects a dependency on a third-party component (a database, MQ, cache, etc.) and you have Docker Desktop installed on your developer machine, Quarkus starts the component for you. You don't have to configure anything. It just happens.

Have a look at the official Quarkus documentation [1.7] to see which components are currently supported in this manner in `dev` mode.

Testing the Database Client

So you don't have to install and configure a PostgreSQL database server locally on your laptop. That's great. Let's test the service now to prove that it works:

```
$ http :8080/person
HTTP/1.1 500 Internal Server Error
Content-Type: text/html;charset=UTF-8
content-length: 113

Could not find MessageBodyWriter for response object of type:
  java.util.ArrayList of media type: application/json
```

OK. Well. It does not work. Why? We need a `MessageBodyWriter` for this response type. Looking at the class `PersonResource`, you can see that we are directly returning a response of type `java.util.List<Person>`, and we have a global producer annotation of `application/json`. We need a component that translates the result into a JSON string.

This can be done through the `quarkus-resteasy-jsonb` or `quarkus-resteasy-jacksonb` extension. We are going to use the first one by executing:

```
$ quarkus ext add quarkus-resteasy-jsonb
[SUCCESS] ✅  Extension io.quarkus:quarkus-resteasy-jsonb has been installed
```

If you now call the endpoint again, you should see the correctly resolved and formatted output:

```
$ http :8080/person
HTTP/1.1 200 OK
Content-Type: application/json
content-length: 741

```

```
[
    {
        "firstName": "Bobby",
        "id": 2,
        "lastName": "Brown",
        "salutation": "Mr"
    },
    {
        "firstName": "Eric",
        "id": 11,
        "lastName": "Clapton",
        "salutation": "Mr"
    },
    {
        "firstName": "Kurt",
        "id": 4,
        "lastName": "Cobain",
        "salutation": "Mr"
    },
...
```

Finalizing the CRUD REST Service

For a well-rounded create, read, update, and delete (CRUD) service, you still have to implement methods to add, delete, and update a person from the list. Let's do that now.

Creating a New Person

The code snippet to create a new person is quite easy. Just implement another method, annotate it with `@POST` and `@Transactional`, and that's it.

```
   @POST
    @Transactional
    public Response create(Person p) {
        if (p == null || p.id != null)
            throw new WebApplicationException("id != null");
        p.persist();
        return Response.ok(p).status(200).build();
    }
```

The only relevant method we call in this method is `persist()`, called on a given `Person` instance. This is known as the active record pattern [1.8].

Let's have a look to see whether it works:

```
$ http POST :8080/person firstName=Carlos lastName=Santana salutation=Mr
HTTP/1.1 200 OK
Content-Type: application/json
content-length: 69

{
    "firstName": "Carlos",
    "id": 12,
    "lastName": "Santana",
    "salutation": "Mr"
}
```

The returned JSON indicates that we did what we intended.

Updating an Existing Person

The same is true for updating a person. Use the **`@PUT`** annotation and make sure you are providing a path parameter, which you have to annotate with **`@PathParam`**:

```
    @PUT
    @Transactional
    @Path("{id}")
    public Person update(@PathParam Long id, Person p) {
        Person entity = Person.findById(id);
        if (entity == null) {
            throw new WebApplicationException("Person with id of " + id
              + " does not exist.", 404);
        }
        if(p.salutation != null ) entity.salutation = p.salutation;
        if(p.firstName != null )  entity.firstName = p.firstName;
        if(p.lastName != null)    entity.lastName = p.lastName;
        return entity;
    }
```

Then test it:

```
$ http PUT :8080/person/6 firstName=Jimi lastName=Hendrix
HTTP/1.1 200 OK
Content-Type: application/json
content-length: 66

{
    "firstName": "Jimi",
    "id": 6,
    "lastName": "Hendrix",
    "salutation": "Mr"
}
```

Deleting an Existing Person

Finally, let's create a **`delete`** method, which works in the same way as the **`update()`** method:

```
    @DELETE
    @Path("{id}")
    @Transactional
    public Response delete(@PathParam Long id) {
        Person entity = Person.findById(id);
        if (entity == null) {
            throw new WebApplicationException("Person with id of " + id +
              " does not exist.", 404);
        }
        entity.delete();
        return Response.status(204).build();
    }
```

And let's check whether it works:

```
$ http DELETE :8080/person/1
HTTP/1.1 204 No Content
```

This is a correct response with a code in the 200 range.

Preparing for CI/CD

Until now, everything you did was for local development. With just a few lines of code, you were able to create a complete database client. You did not even have to worry about setting up a local database for testing.

But how can you specify real database properties when entering test or production stages?

Quarkus supports configuration profiles [1.9]. Properties marked with a given profile name are used only if the application runs in that particular profile. By default, Quarkus supports the following profiles:

- **`dev`**: Gets activated when you run your application via **`quarkus dev`**
- **`test`**: Gets activated when you are running tests
- **`prod`**: The default profile if the application is not started in the **`dev`** profile

In our case, you want to specify database-specific properties only in **`prod`** mode. If you specified a database URL in dev mode, for example, Quarkus would try to use that database server instead of starting the corresponding Dev Services as you want.

Our configuration therefore is:

```
# only when we are developing
%dev.quarkus.hibernate-orm.database.generation=drop-and-create
%dev.quarkus.hibernate-orm.sql-load-script=import.sql

# only in production
%prod.quarkus.hibernate-orm.database.generation=update
%prod.quarkus.hibernate-orm.sql-load-script=no-file

# Datasource settings...
# note, we only set those props in prod mode
quarkus.datasource.db-kind=postgresql
%prod.quarkus.datasource.username=${DB_USER}
%prod.quarkus.datasource.password=${DB_PASSWORD}
%prod.quarkus.datasource.jdbc.url=jdbc:postgresql://${DB_HOST}/${DB_DATABASE}
```

Quarkus also supports the use of property expressions [1.10] For instance, if your application is running on Kubernetes, you might want to specify the datasource username and password via a secret. In this case, use the **`${PROP_NAME}`** expression format to refer to the property that was set in the file. Those expressions are evaluated when they are read. The property names are either specified in the **`application.properties`** file or come from environment variables.

Your application is now prepared for CI/CD (continuous integration/continuous delivery) and production. We'll get to that later in this book.

Moving the Application to OpenShift

Quarkus provides extensions to generate manifest files for Kubernetes or OpenShift [1.11] Let's add the extensions to our **`pom.xml`** file:

```
$ quarkus ext add jib openshift
```

The **`jib`** extension helps you generate a container image out of the application. The **`openshift`** extension generates the necessary manifest files to deploy the application on, well, OpenShift.

Let's specify the properties accordingly:

```
# Packaging the app
quarkus.container-image.builder=jib
quarkus.container-image.image=quay.io/wpernath/person-service:v1.0.0
quarkus.openshift.route.expose=true
quarkus.openshift.deployment-kind=Deployment

# resource limits
quarkus.openshift.resources.requests.memory=128Mi
quarkus.openshift.resources.requests.cpu=250m
quarkus.openshift.resources.limits.memory=256Mi
quarkus.openshift.resources.limits.cpu=500m
```

Now build the application container image via:

```
$ mvn package -Dquarkus.container-image.push=true
```

This command also pushes the image to Quay.io [1.12] as **`quay.io/wpernath/person-service:v1.0.0`**. Quarkus is using Jib [1.13] to build the image.

After building the image, you can install the application in OpenShift by applying the manifest file:

```
$ oc apply -f target/kubernetes/openshift.yml
service/person-service configured
imagestream.image.openshift.io/person-service configured
deployment.apps/person-service configured
route.route.openshift.io/person-service configured
```

Then, create a PostgreSQL database instance in the same namespace from the corresponding template. You can install the database from the OpenShift console by clicking +Add → Developer Catalog → Database → PostgreSQL and filling in meaningful properties for the service name, username, password, and database name. You could alternatively execute the following command from the shell to instantiate a PostgreSQL server in the current namespace:

```
$ oc new-app postgresql-persistent \
    -p POSTGRESQL_USER=wanja \
    -p POSTGRESQL_PASSWORD=wanja \
    -p POSTGRESQL_DATABASE=wanjadb \
    -p DATABASE_SERVICE_NAME=wanjaserver
```

Suppose you've specified the database properties in **`application.properties`** like this:

```
%prod.quarkus.datasource.username=${DB_USER:wanja}
%prod.quarkus.datasource.password=${DB_PASSWORD:wanja}
%prod.quarkus.datasource.jdbc.url=jdbc:postgresql:
  //${DB_HOST:wanjaserver}/${DB_DATABASE:wanjadb}
```

Quarkus takes the values after the colon as defaults, which means you don't have to create those environment values in the **`Deployment`** file for this test. But if you want to use a secret or ConfigMap, have a look at the corresponding extension [1.14] for Quarkus.

After restarting the **`person-service`** you should see that the database is used and that the **`person`** table was created. But there is no data in the database because you've defined the corresponding property to be used in dev mode only.

So fill the database now:

```
$ http POST http://person-service.apps.art8.ocp.lan/person
  firstName=Jimi lastName=Hendrix salutation=Mr

$ http POST http://person-service.apps.art8.ocp.lan/person
  firstName=Joe lastName=Cocker salutation=Mr

$ http POST http://person-service.apps.art8.ocp.lan/person
  firstName=Carlos lastName=Santana salutation=Mr
```

You should now have three singers in the database. To verify, call:

```
$ http http://person-service.apps.art8.ocp.lan/person
HTTP/1.1 200 OK

[
    {
        "firstName": "Joe",
        "id": 2,
        "lastName": "Cocker",
        "salutation": "Mr"
    },
    {
        "firstName": "Jimi",
        "id": 1,
        "lastName": "Hendrix",
        "salutation": "Mr"
    },
    {
        "firstName": "Carlos",
        "id": 3,
        "lastName": "Santana",
        "salutation": "Mr"
    }
]
```

Becoming Native

Do you want to create a native executable out of your Quarkus application? You can do that easily by running:

```
$ mvn package -Pnative -DskipTests
```

However, this command requires you to set up GraalVM [1.15] locally. GraalVM is a Java compiler that creates native executables from Java sources. If you don't want to install and set up GraalVM locally or if you're always building for a container runtime, you could instruct Quarkus to do a container build [1.16] as follows:

```
$ mvn package -Pnative -DskipTests -Dquarkus.native.container-build=true
```

If you also define `quarkus.container-image.build=true`, Quarkus will produce a native container image, which you could then use to deploy to a Kubernetes cluster.

Try it. And if you're using OpenShift 4.9, you could have a look at the `Observe` register within the Developer Console. This page monitors the resources used by a container image.

My OpenShift 4.9 instance is installed on an Intel NUC with a Core i7 with six cores and 64GB of RAM. Using a native image instead of a JVM one changes quite a few things:

- Startup time decreases from 1.2sec (non-native) to 0.03sec (native).
- Memory usage decreases from 120MB (non-native) to 25MB (native).
- CPU utilization drops to 0.2% of the requested CPU time.

Summary

Using Quarkus dramatically reduces the lines of code you have to write. As you have seen, creating a simple REST CRUD service is a piece of cake. If you then want to move your application to Kubernetes, it's just a matter of adding another extension to the build process.

Thanks to Dev Services, you're even able to do fast prototyping without worrying about installing many third-party applications, such as databases.

Minimizing the amount of boilerplate code makes your application easier to maintain and lets you focus on what you need to do: Implement the business case.

This is why I fell in love with Quarkus.

In Chapter 2, we'll have a deeper look into working with images on Kubernetes and OpenShift.

References

[1.1] https://quarkus.io

[1.2] https://red.ht/quarkus-spring-devs

[1.3] https://quarkus.io/get-started/

[1.4] https://httpie.io

[1.5] https://quarkus.io/guides/config

[1.6] https://quarkus.io/guides/hibernate-orm-panache

[1.7] https://quarkus.io/guides/dev-services

[1.8] https://quarkus.io/guides/hibernate-orm-panache#solution-1-using-the-active-record-pattern

[1.9] https://quarkus.io/guides/config-reference#profiles

[1.10] https://quarkus.io/guides/config-reference#property-expressions

[1.11] https://quarkus.io/guides/deploying-to-openshift

[1.12] https://quay.io

[1.13] https://github.com/GoogleContainerTools/jib

[1.14] https://quarkus.io/guides/kubernetes-config

[1.15] https://www.graalvm.org

[1.16] https://quarkus.io/guides/building-native-image#container-runtime

Chapter 2

Deployment Basics

This chapter discusses how applications are deployed in Kubernetes and OpenShift, what manifest files are involved, and how to change the files so that you can redeploy your application into a new, clean namespace without rebuilding it.

The chapter also discusses OpenShift Templates and Kustomize, tools that help automate those necessary file changes.

Introduction and Motivation

As someone with a long history of developing software, I really like containers and Kubernetes because those technologies increase my productivity. They free me from waiting to get what I need (a remote testing system, for example) from the operations department.

On the other hand, writing applications for a container environment—especially microservices—can easily become quite complex because I suddenly also have to maintain artifacts that do not necessarily belong to me:

- ConfigMaps and secrets (well, I have to store my application configuration somehow, anyway)
- The `deployment.yaml` file
- The `service.yaml` file
- An `ingress.yaml` or `route.yaml` file
- A `PersistentVolumeClaim.yaml` file

In native Kubernetes, I have to take care of creating and maintaining those artifacts. Thanks to the Source-to-Image concept in OpenShift, I don't have to worry about most of those manifest files because they will be generated for me.

The following commands create a new project named `book-dev` in OpenShift, followed by a new application called `person-service`. The application is based on the Java builder image `openjdk-11-ubi8` and takes its source code from GitHub. The final command effectively publishes the service so that applications from outside of OpenShift can interact with it:

```
$ oc new-project book-dev
$ oc new-app java:openjdk-11-ubi8~https://github.com/wpernath/book-example.git
  --context-dir=person-service --name=person-service --build-env
  MAVEN_MIRROR_URL=http://nexus.ci:8081/repository/maven-public/
$ oc expose service/person-service
route.route.openshift.io/person-service exposed
```

If you don't have a local Maven mirror, omit the `--build-env` parameter from the second command. The `--context-dir` option lets you specify a subfolder within the Git repository with the actual source files.

OpenShift generates the security settings, deployment, image, route, and service for you (as well as some OpenShift-specific files, such as `DeploymentConfig` or `ImageStream`). These OpenShift conveniences allow you to focus entirely on application development.

In order to let the example start successfully, we have to create a PostgreSQL database server as well. Just execute the following command (we will discuss it later):

```
$ oc new-app postgresql-persistent \
    -p POSTGRESQL_USER=wanja \
    -p POSTGRESQL_PASSWORD=wanja \
    -p POSTGRESQL_DATABASE=wanjadb \
    -p DATABASE_SERVICE_NAME=wanjaserver
```

Basic Kubernetes Files

So what are the necessary artifacts in an OpenShift application deployment?

- `Deployment`: A deployment connects the image with a container and provides various runtime information, including environment variables, startup scripts, and config maps. This configuration file also defines the ports used by the application.
- `DeploymentConfig`: This file is specific to OpenShift, and contains mainly the same functionality as a `Deployment`. If you're starting today with your OpenShift tour, use `Deployment` instead of this file.
- `Service`: A service contains the runtime information Kubernetes needs to load balance your application over different instances (pods).
- `Route`: A route defines the external URL exposed by your application. Requests from clients are received at this URL.
- `ConfigMap`: ConfigMaps contain, well, configurations for the application.
- `Secret`: Like a ConfigMap, a secret contains hashed password information.

Once those files are automatically generated, you can get them by using `kubectl` or `oc`:

```
$ oc get deployment
NAME             READY   UP-TO-DATE   AVAILABLE   AGE
person-service   1/1     1            1           79m
```

By specifying the `-o yaml` option, you can get the complete descriptor:

```
$ oc get deployment person-service -o yaml
apiVersion: apps/v1
kind: Deployment
metadata:
[...]
```

Just pipe the output into a new `.yaml` file, and you're done. You can directly use this file to create your application in a new namespace (except for the image section). But the generated file contains a lot of text you don't need, so it's a good idea to pare it down. For example, you can safely remove the `managedFields` section, big parts of the `metadata` section at the beginning, and the `status` section at the end of each file. After stripping the file down to the relevant parts (shown in the following listing), add it to your Git repository:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: person-service
  name: person-service
spec:
  replicas: 1
  selector:
    matchLabels:
      deployment: person-service
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      labels:
        deployment: person-service
    spec:
      containers:
        - image: image-registry.openshift-image-registry.svc:
                 5000/book-dev/person-service:latest
          imagePullPolicy: IfNotPresent
          name: person-service
          ports:
            - containerPort: 8080
              protocol: TCP
      restartPolicy: Always
```

Do the same with **`Route`** and **`Service`**. That's all for the present. You can now create your application in a new namespace by entering:

```
$ oc new-project book-test
$ oc policy add-role-to-user system:image-puller system:serviceaccount:
  book-test:default --namespace=book-dev
$ oc apply -f raw-kubernetes/service.yaml
$ oc apply -f raw-kubernetes/deployment.yaml
$ oc apply -f raw-kubernetes/route.yaml
```

The **`oc policy`** command is necessary to grant the **`book-test`** namespace access to the image in the namespace **`book-dev`**. Without this command, you'd get an error message in OpenShift saying that the image was not found, unless you enter commands as an admin user.

This section has described one way of getting the required files. Of course, if you have more elements to your application, you will need to export the files defining those elements as well. If you have defined objects of type **`PersistentVolumeClaim`**, **`ConfigMap`**, or **`Secret`**, you need to export them and strip them down as well.

This simple example has shown how you can export your application's manifest files to redeploy it into another clean namespace. Typically, you have to change some fields to reflect differences between environments, especially for the **`Deployment`** file.

For example, it does not make sense to use the latest image from the **`book-dev`** namespace in the **`book-test`** namespace. You'd always have the same version of your application in the development and test environments. To allow the environments to evolve separately, you have to change the image in the **`Deployment`** on every stage you're using. You could do this manually, of course. But let's find some ways to automate it.

YAML Parser (yq)

To maintain different versions of configuration files, the first tool that most likely pops into your mind is the lightweight command-line YAML parser, **`yq`** [2.1].

There are many ports available for most operating systems. On macOS, you can install it via Homebrew [2.2]:

```
$ brew install yq
```

To read the name of the image out of the `Deployment` file, you could enter:

```
$  yq e '.spec.template.spec.containers[0].image' \
raw-kubernetes/deployment.yaml \
image-registry.openshift-image-registry.svc:5000/book-dev/[email protected]
```

To change the name of the image, you could enter:

```
$ yq e -i '.spec.template.spec.containers[0].image = "image-registry.openshift-
  image-registry.svc:5000/book-dev/person-service:latest"' \
raw-kubernetes/deployment.yaml
```

This command updates the `Deployment` in place, changing the name of the image to `person-service:latest`.

The following process efficiently creates a staging release:

- Tag the currently used image in `book-dev` to something more meaningful, such as `person-service:v1.0.0-test`.
- Use `yq` to change the image name in the deployment.
- Create a new namespace.
- Apply the necessary `Deployment`, `Service`, and `Route` configuration files as shown earlier.

This process could easily be scripted in a shell script, for example:

```
#!/bin/bash
oc tag book-dev/something@some-other-thing book-dev/person-service:stage-v1.0.0
yq e -i ...
oc new-project ...
oc apply -f deployment.yaml
```

You can find more details on this topic in my article Release Management with Open-Shift: Under the hood [2.3].

Using a tool such as `yq` seems to be the easiest way to automate the processing of Kubernetes manifest files. However, this process forces you to create and maintain a script with each of your projects. It might be the best solution for small teams and small projects, but as soon as you're responsible for more applications, the demands could quickly get out of control.

So let's discuss other solutions.

OpenShift Templates

OpenShift Templates provides an easy way to create a single file out of the required configuration files and add customizable parameters to the unified file. As the name indicates, the service is offered only on OpenShift and is not portable to a generic Kubernetes environment.

First, create all the standard configurations shown near the beginning of this chapter (such as `route.yaml`, `deployment.yaml`, and `service.yaml`). However, you don't have to separate the configurations into specific files. Next, to create a new template file, open your preferred editor and create a file called `template.yaml`. The header of that file should look like this:

```
apiVersion: template.openshift.io/v1
kind: Template
name: service-template
metadata:
  name: service-template
  annotation:
    tags: java
    iconClass: icon-rh-openjdk
    openshift.io/display-name: The person service template
    description: This Template creates a new service
objects:
```

Then add the configurations you want to combine into this file right under the `objects` tag. Values that you want to change from one system to another should be specified as parameters in the format `$(parameter)`. For instance, a typical service configuration might look like this in example `template.yaml`:

```
 - apiVersion: v1
    kind: Service
    metadata:
      labels:
        app: ${APPLICATION_NAME}
      name: ${APPLICATION_NAME}
    spec:
      ports:
      - name: 8080-tcp
        port: 8080
        protocol: TCP
      selector:
        app: ${APPLICATION_NAME}
```

Then define the parameters in the `parameters` section of the file:

```
parameters:
- name: APPLICATION_NAME
  description: The name of the application you'd like to create
  displayName: Application Name
  required: true
  value: person-service
- name: IMAGE_REF
  description: The full image path
  displayName: Container Image
  required: true
  value: image-registry.openshift-image-registry.svc:5000/book-dev/
         person-service:latest
```

Now for the biggest convenience offered by OpenShift Templates: Once you have instantiated a template in an OpenShift namespace, you can use the template to create applications within the graphical user interface (UI):

```
$ oc new-project book-template
$ oc policy add-role-to-user system:image-puller system:serviceaccount:
  book-template:default --namespace=book-dev
$ oc apply -f ocp-template/service-template.yaml*
template.template.openshift.io/service-template created
```

Now, open the OpenShift web console, choose the project, click +Add, and choose the Developer Catalog. You should find a template called `service-template` (Figure 2.1). This is the one we've created.

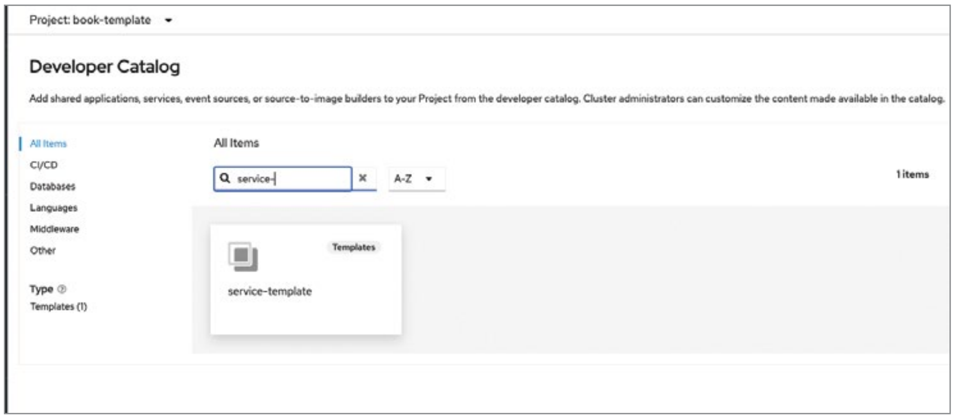
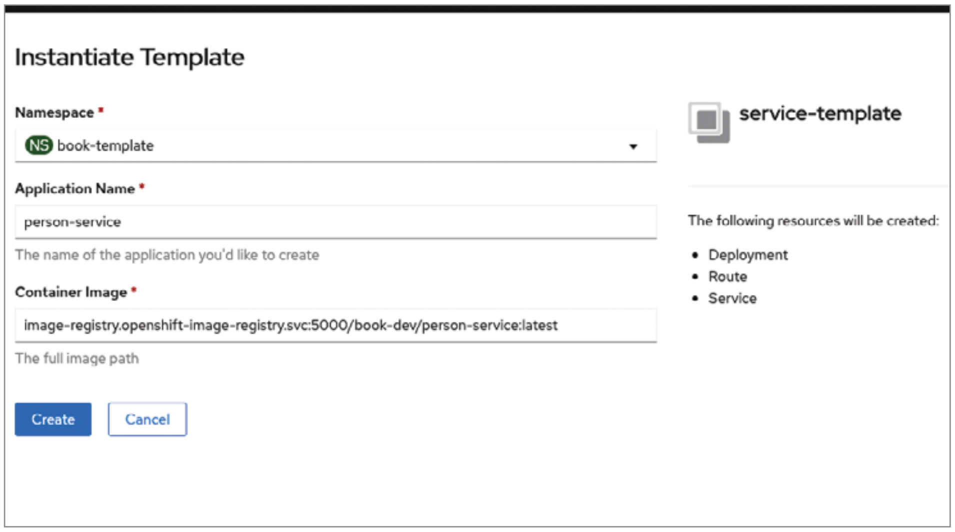

Project: book-template
Developer Catalog
Add shared applications, services, event sources, or source-to-image builders to your Project from the developer catalog. Cluster administrators can customize the content made available in the catalog.
All Items
CI/CD
Databases
Languages
Middleware
Other
Type
Templates (1)
All Items
service
A-Z
1 items
Templates
service-template

Figure 2.1: The Developer Catalog after adding the template.

Instantiate the template and fill in the required fields (Figure 2.2).



Namespace *
NS book-template

Application Name *
person-service
The name of the application you'd like to create

Container Image *
image-registry.openshift-image-registry.svc:5000/book-dev/person-service:latest
The full image path

The following resources will be created:

- Deployment
- Route
- Service

Figure 2.2: Template instantiation with required fields.

Then click Create. After a short time, you should see the application's deployment progressing. Once it is finished, you should be able to access the route of the application.

There are also several ways to create an application instance out of a template without the UI. You can run an `oc` command to do the work within OpenShift:

```
$ oc new-app service-template -p APPLICATION_NAME=simple-service
--> Deploying template "book-template/service-template" to project book-template

     * With parameters:
        * Application Name=simple-service
        * Container Image=image-registry.openshift-image-registry.svc:
          5000/book-dev/person-service:latest

--> Creating resources ...
    route.route.openshift.io "simple-service" created
    service "simple-service" created
    deployment.apps "simple-service" created
--> Success
    Access your application via route
     'simple-service-book-template.apps.art3.ocp.lan'
    Run 'oc status' to view your app.
```

Finally, you can process the template locally:

```
$ oc process service-template APPLICATION_NAME=process-service -o yaml | oc
  apply -f -
route.route.openshift.io/process-service created
service/process-service created
deployment.apps/process-service created
```

Whatever method you choose to process the template, results show up in your Topology view for the project (Figure 2.3).

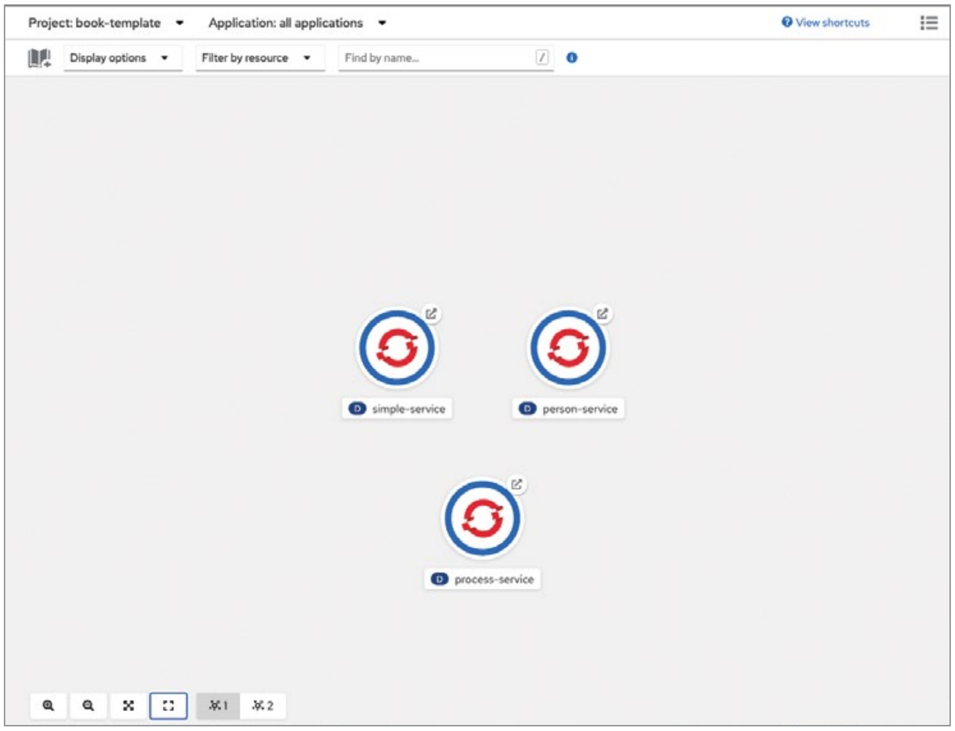


Figure 2.3: The OpenShift UI after using the template in several ways.

Creating and maintaining an OpenShift Template is fairly easy. Parameters can be created and set in intuitive ways. I personally like the deep integration into OpenShift's developer console and the `oc` command.

I would like OpenShift Templates even better if I could extend the development process to other teams. I would like to be able to create a template of a standard application (including a `BuildConfig`, etc.), and import it into the global `openshift` namespace so that all users could reuse my base—just like the other OpenShift Templates shipped with any OpenShift installation.

Unfortunately, OpenShift Templates is for OpenShift only. If you are using a local Kubernetes installation and a production OpenShift version, the template is not easy to reuse. But if your development and production environments are completely based on OpenShift, you should give it a try.

Kustomize

Kustomize is a command-line tool that edits Kubernetes YAML configuration files in place, similar to `yq`. Kustomize tends to be easy to use because usually, only a few properties of configuration files have to be changed from stage to stage. Therefore, you start by creating a base set of files (`Deployment`, `Service`, `Route` etc.) and apply changes through Kustomize for each stage. Kustomize's patch mechanism takes care of merging the files together.

Kustomize is very handy if you don't want to learn a new templating engine and maintain a file that could easily contain thousands of lines, as happens with OpenShift Templates.

Kustomize was originally created by Google and is now a subproject of Kubernetes. The command-line tools, such as `kubectl` and `oc`, have most of the necessary functionality built-in.

How Kustomize Works

Let's have a look at the files in a Kustomize directory:

```
$ tree kustomize
kustomize
├── base
│   ├── deployment.yaml
│   ├── kustomization.yaml
│   ├── route.yaml
│   └── service.yaml
└── overlays
    ├── dev
    │   ├── deployment.yaml
    │   ├── kustomization.yaml
    │   └── route.yaml
    └── stage
        ├── deployment.yaml
        ├── kustomization.yaml
        └── route.yaml

4 directories, 10 files
```

The top-level directory contains the `base` files and an `overlays` subdirectory. The `base` files define the resources that Kubernetes or OpenShift need to deploy your application. You should be familiar with these files from the previous sections of this chapter.

Only `kustomization.yaml` is new. Let's have a look at this file:

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

commonLabels:
  org: wanja.org

resources:
- deployment.yaml
- service.yaml
- route.yaml
```

This file defines the resources for the deployment (`Deployment`, `Service`, and `Route`) and adds a section called `commonLabels`. Those labels will be applied to all resources generated by Kustomize.

The following commands process the files and deploy our application on OpenShift:

```
$ oc new-project book-kustomize
$ oc apply -k kustomize/overlays/dev
service/dev-person-service created
deployment.apps/dev-person-service created
route.route.openshift.io/dev-person-service created
```

If you also install the Kustomize command-line tool (for example, with **`brew install kustomize`** on macOS), you can debug the output:

```
$ kustomize build kustomize/overlays/dev
apiVersion: v1
kind: Service
metadata:
  annotations:
    stage: development
  labels:
    app: person-service
    org: wanja.org
    variant: development
  name: dev-person-service
spec:
  ports:
  - name: 8080-tcp
    port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    deployment: person-service
    org: wanja.org
    variant: development
  sessionAffinity: None
  type: ClusterIP
---
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  annotations:
    stage: development
  labels:
    app: person-service
    org: wanja.org
    variant: development
  name: dev-person-service
spec:
  port:
    targetPort: 8080-tcp
  to:
    kind: Service
    name: dev-person-service
    weight: 100
  wildcardPolicy: None
---
[...]
```

A significant benefit of Kustomize is that you have to maintain only the differences between each stage, so the overlay files are quite small and clear. If a file does not change between stages, it does not need to be duplicated.

Kustomize fields such as `commonLabels` or `commonAnnotations` can specify labels or annotations that you would like to have in every metadata section of every generated file. `namePrefix` specifies a prefix for Kustomize to add to every `name` tag.

The following command merges the files for the staging overlay:

```
$ kustomize build kustomize/overlays/stage
```

The following output shows that all filenames have `staging-` as a prefix. Additionally, the configuration has a new `commonLabel` (the `variant: staging` line) and annotation (`note: we are on staging now`):

```
$ kustomize build kustomize/overlays/stage
[...]

apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    note: We are on staging now
    stage: staging
  labels:
    app: person-service
    org: wanja.org
    variant: staging
  name: staging-person-service
spec:
  progressDeadlineSeconds: 600
  replicas: 2
  selector:
    matchLabels:
      deployment: person-service
      org: wanja.org
      variant: staging
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      annotations:
        note: We are on staging now
        stage: staging
      labels:
        deployment: person-service
        org: wanja.org
        variant: staging
    spec:
      containers:
      - env:
        - name: APP_GREETING
          value: Hey, this is the STAGING environment of the App
        image: image-registry.openshift-image-registry.svc:
               5000/book-dev/person-service:latest
---
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  annotations:
    note: We are on staging now
    stage: staging
  labels:
    app: person-service
    org: wanja.org
    variant: staging
  name: staging-person-service
spec:
  port:
    targetPort: 8080-tcp
  to:
    kind: Service
    name: staging-person-service
    weight: 100
  wildcardPolicy: None
```

The global **`org`** label is still specified. You can deploy the stage to OpenShift with the following command:

```
$ oc apply -k kustomize/overlays/stage
```

More Sophisticated Kustomize Examples

Instead of using **`patchStrategicMerge`** files, you could just maintain a **`kustomization.yaml`** file containing everything. Here is an example:

```
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
  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
```

There are specific fields in newer versions of Kustomize (version 4.x and above) that help you maintain your overlays even better. For example, if all you have to do is change the tag of the target image, you could simply use the **`images`** field array specifier shown in the previous listing.

The **`patches`** parameter can issue a patch on a list of target files, such as replacing the target service name of the **`Route`** (as shown in the following listing) or adding health checks for the application in the **`Deployment`** file:

```
# 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
```

The following patch applies the file **`apply-health-checks.yaml`** to the **`Deployment`**:

```
# apply some patches
patches:
  # apply health checks to deployment
  - path: apply-health-checks.yaml
    target:
      version: v1
      kind: Deployment
      name: person-service
```

The following file is the patch itself and gets applied to the **`Deployment`**:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: person-service
spec:
  template:
    spec:
      containers:
      - name: person-service
        readinessProbe:
          httpGet:
            path: /q/health/ready
            port: 8080
            scheme: HTTP
          timeoutSeconds: 1
          periodSeconds: 10
          successThreshold: 1
          failureThreshold: 3
        livenessProbe:
          httpGet:
            path: /q/health/live
            port: 8080
            scheme: HTTP
          timeoutSeconds: 2
          periodSeconds: 10
          successThreshold: 1
          failureThreshold: 3
```

You can even generate the ConfigMap based on fixed parameters or properties files:

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

Starting with Kubernetes release 1.21 (which is reflected in OpenShift 4.8.x), `oc` and `kubectl` contain advanced Kustomize features from version 4.0.5. Kubernetes 1.22 (OpenShift 4.9.x) will contain features of Kustomize 4.2.0.

Before Kubernetes 1.21 (OpenShift 4.7.x and earlier) `oc apply -k` does not contain recent Kustomize features. So if you want to use those features, you need to use the `kustomize` command-line tool and pipe the output to `oc apply -f`.

```
$ kustomize build kustomize-ext/overlays/stage | oc apply -f -
```

For more information and even more sophisticated examples, have a look at the Kustomize home page [2.4] as well as the examples in the official GitHub repository [2.5].

Using Kustomize and Argo CD

Using Kustomize is relatively straightforward. You don't really have to learn a templating DSL. You just need to understand the processes of patching and merging. Kustomize makes it easy for CI/CD practitioners to separate the configuration of an application for every stage. And because Kustomize is a tightly integrated Kubernetes subproject, you don't have to worry that it will suddenly disappear.

Argo CD has built-in support for Kustomize as well, so if you're doing CI/CD with Argo CD, you can still use Kustomize.

Summary

In this chapter we saw how to build an application using OpenShift's Source-to-Image (S2I) technology, along with the YAML parser, OpenShift Templates, and Kustomize. These are the base technologies for automating application deployment and packaging.

Now you have an understanding of which artifacts need to be taken into account when you want to release your application and how to modify those artifacts to make sure that the new environment is capable of handling your application.

In the next chapter, we'll discuss Helm charts and Kubernetes Operators for application packaging and distribution.

References

[2.1] https://github.com/mikefarah/yq

[2.2] https://brew.sh

[2.3] https://www.opensourcerers.org/2017/09/19/release-management-with-openshift-under-the-hood/

[2.4] https://kustomize.io/

[2.5] https://github.com/kubernetes-sigs/kustomize/tree/master/examples

Chapter 3

Packaging with Helm and Kubernetes Operators

In this chapter, you'll learn more about working with container images. In particular, we will dive into the concepts of Helm charts and Kubernetes Operators.

Modern Distributions

As mentioned in Chapter 2, the most challenging task when turning code into a useful product is creating a distributable package from the application. It is no longer simply 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 the owner of the application.

Distribution takes place on two levels:

- Internal distribution: making a containerized application available to the IT team within your 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 applications available for others (external distribution), you might have to put them 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, you need a repository to externally distribute an application in the modern and generally expected manner. The repository could be either a public image repository such as Quay.io [3.1] or Docker Hub [3.2], or a private repository that is accessible by your customers. This chapter uses the Quay.io [3.3] public repository for its examples, but the principles and most of the commands also apply to other types of repositories.

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, you'll need to install either Docker [3.4] or Podman [3.5] 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 thus 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. If you are working on a Windows or macOS system, you have to set up a remote Linux system to make full use of Podman [3.6]. You might also have a look at a newer version of CodeReady Containers (version 2.x onwards) [3.7], which aims to become a full Docker Desktop replacement for Windows and macOS using Podman underneath.

Podman 4.x onwards [3.8] massively simplifies the way to setup the required remote Linux system with a new API. The following command initializes a new host:

```
$ podman machine init
```

And the following command starts the local VM:

```
$ podman machine start
```

If you previously have used Podman on your Windows or MacOS system, have a look at the system connection list. It should return something like this. If there is still your old system connection available, then either delete it or make it non-default.

```
$ podman system connection list
Name                        URI                              Identity                    Default
podman-machine-default      ssh://core@localhost:58845/      /Users/wpernath/.ssh/       true
                            run/user/501/podman/podman.sock podman-machine-default
podman-machine-default-root ssh://root@localhost:58845/      /Users/wpernath/            false
                            run/podman/podman.sock          .ssh/podman-machine-default
```

Podman internally uses Buildah [3.9] to build container images. According to the official GitHub page [3.10], Buildah is the main tool for building images that conform to the Open Container Initiative (OCI) [3.11] 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) [3.12] or a Tekton pipeline), you should directly use Buildah.

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 [3.13]. In **`person-service/src/main/docker`**, you can find the file that configures builds for your application. The file is simply called **`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:

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

The first command logs into your Quay.io account. The second command builds the application with Maven [3.14]. The third, finally, creates a Docker image out of the application. 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`**:

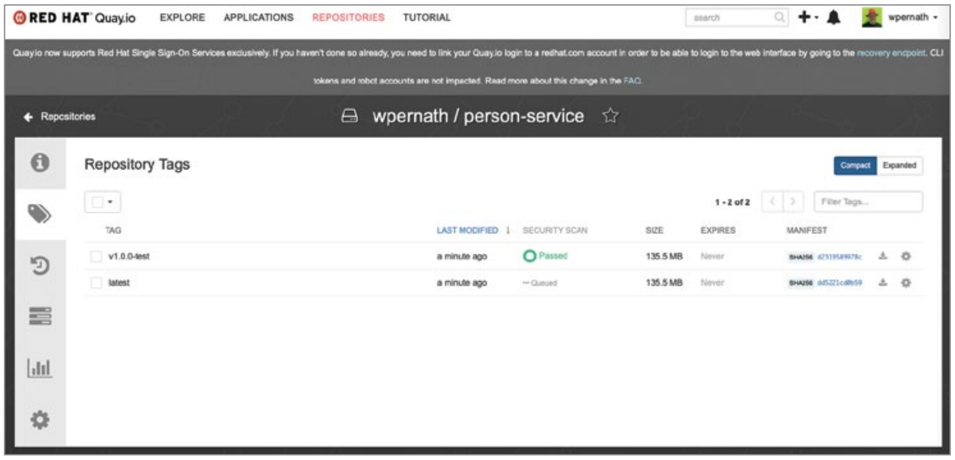
```
$ alias docker=podman
```

Setting up Podman on any non-Linux system is a little bit tricky, as mentioned. You need access to a Linux system, running either directly on one of your systems or in a virtual machine. The Linux system basically works as the execution unit for the Podman client. The documentation mentioned earlier [3.15] shows you how that works.

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

```
$ docker push quay.io/wpernath/person-service -a
```

This will push all (`-a`) locally stored images to the external repository, including all tags (Figure 3-1). Now you can use the image in your OpenShift environment.



Quay.io now supports Red Hat Single Sign-On Services exclusively. If you haven't done so already, you need to link your Quay.io login to a redhat.com account in order to be able to login to the web interface by going to the recovery endpoint. CLI tokens and robot accounts are not impacted. Read more about this change in the FAQ.

TAG	LAST MODIFIED	SECURITY SCAN	SIZE	EXPIRES	MANIFEST
v1.0.0-test	a minute ago	Passed	135.5 MB	Never	SHA256 d7319589978c
latest	a minute ago	Queued	135.5 MB	Never	SHA256 dd5221cd8b59

Figure 3-1. The person-service on quay.io after you've pushed it via docker push.

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

Because Docker and 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

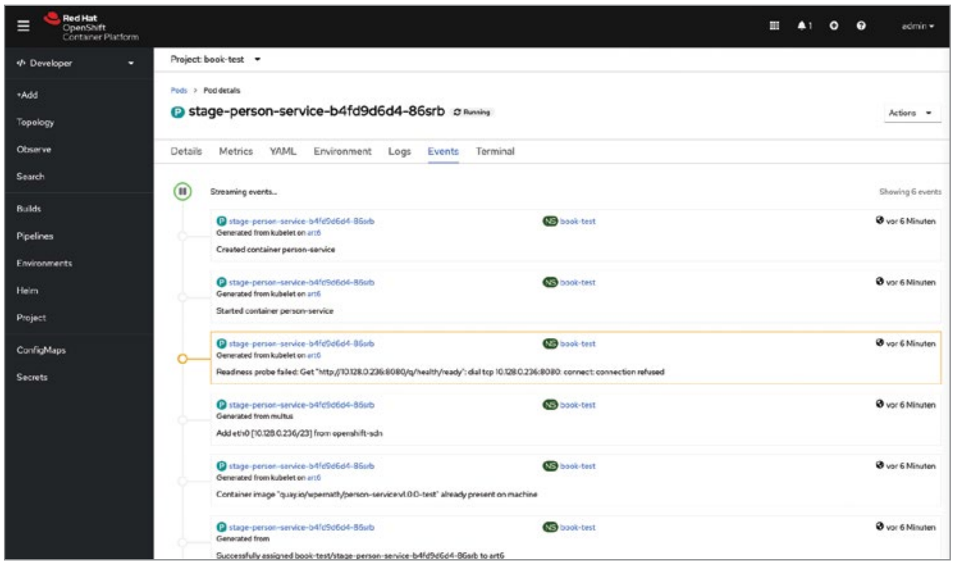
Now that your image is stored in Quay.io, test it to see whether everything has successfully worked out. For this, we will use our Kustomize example from the previous chapter.

First of all, make sure that the **`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
$ oc apply -k 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 Figure 3-2.



```
stage-person-service-b4fd9d6d4-86srb        book-test        vor 6 Minuten
Generated from kubelet on art6
Created container person-service

stage-person-service-b4fd9d6d4-86srb        book-test        vor 6 Minuten
Generated from kubelet on art6
Started container person-service

stage-person-service-b4fd9d6d4-86srb        book-test        vor 6 Minuten
Generated from kubelet on art6
Readiness probe failed: Get "http://10.128.0.236:8080/q/health/ready": dial tcp 10.128.0.236:8080: connect: connection refused

stage-person-service-b4fd9d6d4-86srb        book-test        vor 6 Minuten
Generated from multus
Add eth0 [10.128.0.236/23] from openshift-sdn

stage-person-service-b4fd9d6d4-86srb        book-test        vor 6 Minuten
Generated from kubelet on art6
Container image "quay.io/wpernath/person-service:v1.0.0-test" already present on machine

stage-person-service-b4fd9d6d4-86srb        book-test        vor 6 Minuten
Generated from
Successfully assigned book-test/stage-person-service-b4fd9d6d4-86srb to art6
```

Figure 3-2: Event log for dev-person-service.

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; in most cases, the **`podman`** CLI 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 [3.16] is another command-line tool that helps you work with container images and different image registries without the heavyweight Docker daemon. On macOS, you can easily install Skopeo via `brew`:

```
$ brew install skopeo
```

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

```
$ skopeo copy \
   docker://quay.io/wpernath/person-service:latest \
   docker://quay.io/wpernath/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 [3.17] and a Kubernetes Operator [3.18]. 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 can 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.

First, you need to download and install the Helm CLI [3.19]. On macOS you can easily do this via:

```
$ brew install helm
```

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:

```
$ 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: something@some-other-thing
```

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/
```

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
```

Package, Install, Upgrade, and Roll Back Your Helm Chart

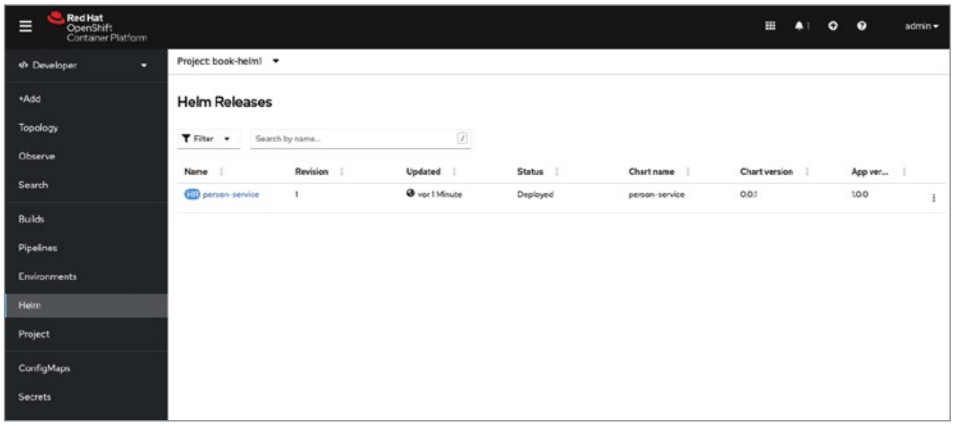
Now that you have a very simple Helm chart, you can package it:

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

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 (Figure 3-3).



Name	Revision	Updated	Status	Chart name	Chart version	App ver...
HR person-service	1	vor 1 Minute	Deployed	person-service	0.0.1	1.0.0

Figure 3-3: OpenShift console with our Helm chart.

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. It's also available via the OpenShift UI in Developer→Helm→Release Notes). You can enter your release notes there as a Markdown-formatted text file like the following example. 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 [3.20] 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 [3.21] is an implementation of the templating engine [3.22] of the Go [3.23] 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 of commands help you debug your templates first.

- The `helm lint` command checks whether your chart follows 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 [3.24] come into play.

A hook is a Kubernetes resource (like a job or a pod) that 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 lets you assign weights to hooks 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 the required subcharts into the **`helm/charts`** folder. This can 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 easy and mostly self-explanatory. 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.

Kubernetes Operators

Our simple Quarkus application makes few demands of an administrator or of its hosting environment, whether plain Kubernetes or OpenShift. The application is a stateless, web-based application that doesn't require any special treatment by an administrator.

If you use the Helm chart to install the application (or even install it manually into OpenShift via `oc apply -f`), Kubernetes understands how to manage it out of the box pretty well. Kubernetes's control loop mechanism knows what the desired state of the application is (based on its various YAML files) and compares the desired state continually with the application's current state. Any deviations from the desired state are fixed automatically. For instance, if a pod has just died, Kubernetes takes care of restarting it. If you have uploaded a new version of the image, Kubernetes re-instantiates the whole application with the new image.

That's pretty easy.

But what happens if your application requires some complex integrations into other applications not built for Kubernetes, such as a database? Or if you want to back up and restore stateful data? Or need a clustered database? Such requirements typically require the special know-how of an administrator.

A Kubernetes Operator embodies those administrative instructions. It creates a package that contains, in addition to the information Kubernetes needs to deploy your application, the know-how of an administrator to maintain the complex, stateful part of the application.

Of course, this makes an Operator way more complex than a Helm chart because all the logic needs to be implemented before putting it into the Operator. There are officially three ways to implement an Operator:

- Create it from Ansible.
- Create it from a Helm chart.
- Develop everything in Go.

Unofficially (not supported right now), you can also implement the Operator's logic in other programming languages, such as Java via a Quarkus extension [3.25].

An Operator creates, watches, and maintains CRDs. This means that it provides new API resources to the cluster, such as a `Route` or `BuildConfig`. Whenever someone creates a new resource via `oc apply` based on the CRD, the Operator knows what to do. The Operator handles all the logic behind that mechanism (just think about the work necessary to set up a clustered database or back up and restore the persistent volume of a database). It makes extensive use of the Kubernetes API.

If you need to have full control over everything, you have to create the Operator with Go or (unofficially) with Java. Otherwise, you can make use of an Ansible-based or Helm-based Operator. The Operator SDK and the base packages in Ansible and Helm take care of the Kubernetes API calls. So you don't have to learn Go now in order to build your first Operator.

Creating a Helm-based Operator

To create an Operator, you need to install the Operator SDK [3.26]. On macOS, you can simply execute:

```
$ brew install operator-sdk
```

Generating the project structure

In this example, we'll create an Operator based on the Helm chart created earlier in the chapter:

```
$ mkdir kube-operator
$ cd kube-operator
$ operator-sdk init \
   --plugins=helm --helm-chart=../helm-chart \
   --domain wanja.org --group charts \
   --kind PersonService \
   --project-name person-service-operator
Writing kustomize manifests for you to edit...
Creating the API:
$ operator-sdk create api --group charts --kind PersonService --helm-chart
  ../helm-chart
Writing kustomize manifests for you to edit...
Created helm-charts/person-service
Generating RBAC rules
I1027 13:19:52.883020   72377 request.go:665] Waited for 1.017668738s due to
  client-side throttling, not priority and fairness, request:
  GET:https://api.art6.ocp.lan:6443/apis/controlplane.operator.openshift.io/
      v1alpha1?timeout=32s
WARN[0002] The RBAC rules generated in config/rbac/role.yaml are based on the
  chart's default manifest. Some rules might be missing for resources that are
  only enabled with custom values, and some existing rules may be overly broad.
  Double-check the rules generated in config/rbac/role.yaml to ensure they meet
  the Operator's permission requirements.
```

These commands initialize the project for the Operator based on the chart found in the `../helm-chart` folder. A `PersonService` CRD should also have been created in `config/crd/bases`. The following listing shows the complete directory structure generated by the commands:

```
$ tree
.
├── Dockerfile
├── Makefile
├── PROJECT
├── config
│   ├── crd
│   │   ├── bases
│   │   │   └── charts.wanja.org_personservices.yaml
│   │   └── kustomization.yaml
│   ├── default
│   │   ├── kustomization.yaml
│   │   ├── manager_auth_proxy_patch.yaml
│   │   └── manager_config_patch.yaml
│   ├── manager
│   │   ├── controller_manager_config.yaml
│   │   ├── kustomization.yaml
│   │   └── manager.yaml
│   ├── manifests
│   │   └── kustomization.yaml
│   ├── prometheus
│   │   ├── kustomization.yaml
│   │   └── monitor.yaml
│   ├── rbac
│   │   ├── auth_proxy_client_clusterrole.yaml
│   │   ├── auth_proxy_role.yaml
│   │   ├── auth_proxy_role_binding.yaml
│   │   ├── auth_proxy_service.yaml
│   │   ├── kustomization.yaml
│   │   ├── leader_election_role.yaml
│   │   ├── leader_election_role_binding.yaml
│   │   ├── personservice_editor_role.yaml
│   │   ├── personservice_viewer_role.yaml
│   │   ├── role.yaml
│   │   ├── role_binding.yaml
│   │   └── service_account.yaml
│   ├── samples
│   │   ├── charts_v1alpha1_personservice.yaml
│   │   └── kustomization.yaml
│   └── scorecard
│       ├── bases
│       │   └── config.yaml
│       ├── kustomization.yaml
│       └── patches
│           ├── basic.config.yaml
│           └── olm.config.yaml
├── helm-charts
│   └── person-service
│       ├── Chart.yaml
│       ├── templates
│       │   ├── NOTES.txt
│       │   ├── config-map.yaml
│       │   ├── deployment.yaml
│       │   ├── post-install-hook.yaml
│       │   ├── route.yaml
│       │   └── service.yaml
│       └── values.yaml
└── watches.yaml

15 directories, 41 files
```

The Helm-based Operator uses the **`watches.yaml`** file to watch changes on the API. So whenever you create a new resource based on the CRD, the underlying logic knows what to do.

Now have a look at the **`Makefile`**. There are three parameters in it that you should change:

- **`VERSION`**: Whenever you change something in the project file (and have running instances of the Operator somewhere), increment the number.

- **`IMAGE_TAG_BASE`**: This is the base name of the images that the makefile produces. Change the name to something like **`quay.io/wpernath/person-service-operator`**.
- **`IMG`**: This is the name of the image within our Helm-based Operator. Change the name to something like **`$(IMAGE_TAG_BASE):$(VERSION)`**.

Build the Docker image

Now build and push the Docker image of your Operator:

```
$ make docker-build
docker build -t quay.io/wpernath/person-service-operator:0.0.1 .
[+] Building 6.3s (9/9) FINISHED

[...]

 => [2/4] COPY watches.yaml /opt/helm/watches.yaml                           0.1s
 => [3/4] COPY helm-charts  /opt/helm/helm-charts                            0.0s
 => [4/4] WORKDIR /opt/helm                                                  0.0s
 => exporting to image                                                       0.0s
 => => exporting layers                                                      0.0s
 => => writing image
   sha256:b5f909021fe22d666182309e3f30c418c80d3319b4a834c5427c5a7a71a42edc    0.0s
 => => naming to quay.io/wpernath/person-service-operator:0.0.1

$ make docker-push
docker push quay.io/wpernath/person-service-operator:0.0.1
The push refers to repository [quay.io/wpernath/person-service-operator]
5f70bf18a086: Pushed
fe540cb9bcd7: Pushed
ddaee5130ba6: Pushed
4ea9a10139f9: Mounted from operator-framework/helm-operator
5a5ce86c51f0: Mounted from operator-framework/helm-operator
3a40d0007ffb: Mounted from operator-framework/helm-operator
0b911edbb97f: Mounted from wpernath/person-service
54e42005468d: Mounted from wpernath/person-service
0.0.1: digest: sha256:
      acd3f89a7e0788b226d5016df765f61a5429cf5cef6118511a0910b9f4a04aaf size: 1984
```

In your repository on Quay.io, you now have a new image called **`person-service-operator`**. This image contains the logic to manage the Helm chart. The image exposes the CRD and the new Kubernetes API.

Test your Operator

There are currently three different ways to run an Operator, listed in the official SDK tutorial [3.27]. The easiest way to test your Operator is to run:

```
$ make deploy
cd config/manager && /usr/local/bin/kustomize edit set image controller=quay.io/
  wpernath/person-service-operator:0.0.1
/usr/local/bin/kustomize build config/default | kubectl apply -f -
namespace/person-service-operator-system created
customresourcedefinition.apiextensions.k8s.io/personservices.charts.wanja.org created
serviceaccount/person-service-operator-controller-manager created
role.rbac.authorization.k8s.io/person-service-operator-leader-election-role created
clusterrole.rbac.authorization.k8s.io/person-service-operator-manager-role created
clusterrole.rbac.authorization.k8s.io/person-service-operator-metrics-reader created
clusterrole.rbac.authorization.k8s.io/person-service-operator-proxy-role created
rolebinding.rbac.authorization.k8s.io/person-service-operator-leader-election-
  rolebinding created
clusterrolebinding.rbac.authorization.k8s.io/person-service-operator-manager-
  rolebinding created
clusterrolebinding.rbac.authorization.k8s.io/person-service-operator-proxy-
  rolebinding created
configmap/person-service-operator-manager-config created
service/person-service-operator-controller-manager-metrics-service created
deployment.apps/person-service-operator-controller-manager created
```

This command creates a `person-service-operator-system` namespace and installs all the necessary files into Kubernetes. This namespace contains everything needed to handle the request to create a `PersonService` but does not contain any instances of your newly created custom resource.

To see your `PersonService` in action, create a new namespace and apply a new instance of the `PersonService` through the following configuration file, located in `config/samples/charts_v1alpha1_personservice`:

```
apiVersion: charts.wanja.org/v1alpha1
kind: PersonService
metadata:
  name: my-person-service1
spec:
  # Default values copied from <project_dir>/helm-charts/person-service/values.yaml
  config:
    greeting: Hello from inside the operator!!!!
  deployment:
    image: quay.io/wpernath/person-service
    includeHealthChecks: false
    replicas: 1
    version: v1.0.0-test
The following commands create the service:
$ oc new-project book-operator
$ oc apply -f config/samples/charts_v1alpha1_personservice
personservice.charts.wanja.org/my-person-service1 configured
```

This will take a while. But after some time, you will notice a change in the Topology view of OpenShift, showing that the Helm chart was deployed.

To delete everything, delete all instances of the CRD you've created:

```
$ oc delete personservice/my-person-service1
$ make undeploy
```

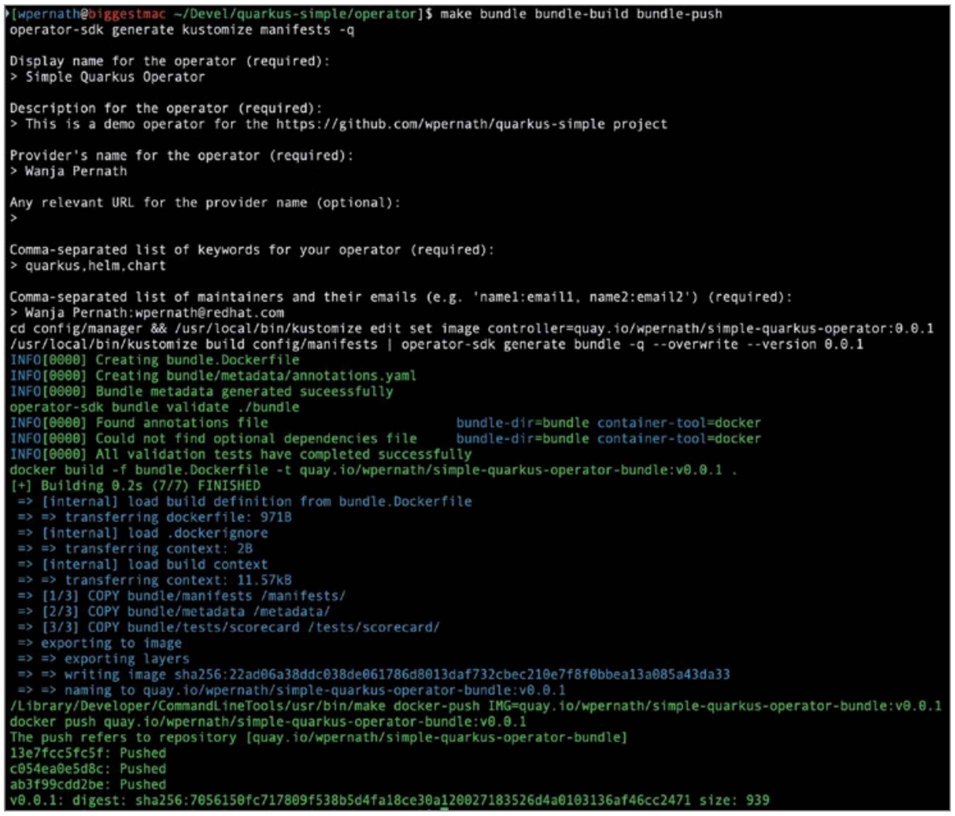
Build and run the Operator bundle image

To release your Operator, you have to create an Operator bundle. This bundle is an image with the metadata and manifests used by the Operator Lifecycle Manager (OLM), which takes care of every Operator deployed on Kubernetes. To create the bundle, enter:

```
$ make bundle
operator-sdk generate kustomize manifests -q
cd config/manager && /usr/local/bin/kustomize edit set image controller=quay.io/
  wpernath/person-service-operator:0.0.3
/usr/local/bin/kustomize build config/manifests | operator-sdk generate bundle
  -q --overwrite --version 0.0.3
INFO[0000] Creating bundle.Dockerfile
INFO[0000] Creating bundle/metadata/annotations.yaml
```

```
INFO[0000] Bundle metadata generated suceessfully
operator-sdk bundle validate ./bundle
INFO[0000] All validation tests have completed successfully
```

The bundle generator asks you a few questions to configure the bundle. Have a look at Figure 3-4 for the output.



```
[wpernath@biggestmac ~/Devel/quarkus-simple/operator]$ make bundle bundle-build bundle-push
operator-sdk generate kustomize manifests -q

Display name for the operator (required):
> Simple Quarkus Operator

Description for the operator (required):
> This is a demo operator for the https://github.com/wpernath/quarkus-simple project

Provider's name for the operator (required):
> Wanja Pernath

Any relevant URL for the provider name (optional):
>

Comma-separated list of keywords for your operator (required):
> quarkus,helm,chart

Comma-separated list of maintainers and their emails (e.g. 'name1:email1, name2:email2') (required):
> Wanja Pernath:wpernath@redhat.com
cd config/manager && /usr/local/bin/kustomize edit set image controller=quay.io/wpernath/simple-quarkus-operator:0.0.1
/usr/local/bin/kustomize build config/manifests | operator-sdk generate bundle -q --overwrite --version 0.0.1
INFO[0000] Creating bundle.Dockerfile
INFO[0000] Creating bundle/metadata/annotations.yaml
INFO[0000] Bundle metadata generated suceessfully
operator-sdk bundle validate ./bundle
INFO[0000] Found annotations file                       bundle-dir=bundle container-tool=docker
INFO[0000] Could not find optional dependencies file    bundle-dir=bundle container-tool=docker
INFO[0000] All validation tests have completed successfully
docker build -f bundle.Dockerfile -t quay.io/wpernath/simple-quarkus-operator-bundle:v0.0.1 .
[+] Building 0.2s (7/7) FINISHED
 => [internal] load build definition from bundle.Dockerfile
 => => transferring dockerfile: 971B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load build context
 => => transferring context: 11.57kB
 => [1/3] COPY bundle/manifests /manifests/
 => [2/3] COPY bundle/metadata /metadata/
 => [3/3] COPY bundle/tests/scorecard /tests/scorecard/
 => exporting to image
 => => exporting layers
 => => writing image sha256:22ad06a38ddc038de061786d8013daf732cbec210e7f8f0bbea13a085a43da33
 => => naming to quay.io/wpernath/simple-quarkus-operator-bundle:v0.0.1
/Library/Developer/CommandLineTools/usr/bin/make docker-push IMG=quay.io/wpernath/simple-quarkus-operator-bundle:v0.0.1
docker push quay.io/wpernath/simple-quarkus-operator-bundle:v0.0.1
The push refers to repository [quay.io/wpernath/simple-quarkus-operator-bundle]
13e7fcc5fc5f: Pushed
c054ea0e5d8c: Pushed
ab3f99cdd2be: Pushed
v0.0.1: digest: sha256:7056150fc717809f538b5d4fa18ce30a120027183526d4a0103136af46cc2471 size: 939
```

Figure 3-4. Building the bundle.

Run the generator every time you change the **`VERSION`** field in the makefile:

```
$ make bundle-build bundle-push
docker build -f bundle.Dockerfile -t
  quay.io/wpernath/person-service-operator-bundle:v0.0.3 .
[+] Building 0.2s (7/7) FINISHED
 => [internal] load build definition from bundle.Dockerfile                 0.0s
 => => transferring dockerfile: 987B                                        0.0s
 => [internal] load .dockerignore                                           0.0s
 => => transferring context: 2B                                             0.0s
 => [internal] load build context                                           0.0s
 => => transferring context: 11.89kB                                        0.0s
 => [1/3] COPY bundle/manifests /manifests/                                 0.0s
 => [2/3] COPY bundle/metadata /metadata/                                   0.0s
 => [3/3] COPY bundle/tests/scorecard /tests/scorecard/                     0.0s
 => exporting to image                                                      0.0s
 => => exporting layers                                                     0.0s
 => => writing image
  sha256:7f712b903cbdbf12b6f34189cdbca404813ade4d7681d3d051fb7f6b2e12d0f5    0.0s
 => => naming to quay.io/wpernath/person-service-operator-bundle:v0.0.3      0.0s
/Library/Developer/CommandLineTools/usr/bin/make docker-push
  IMG=quay.io/wpernath/person-service-operator-bundle:v0.0.3
docker push quay.io/wpernath/person-service-operator-bundle:v0.0.3
```

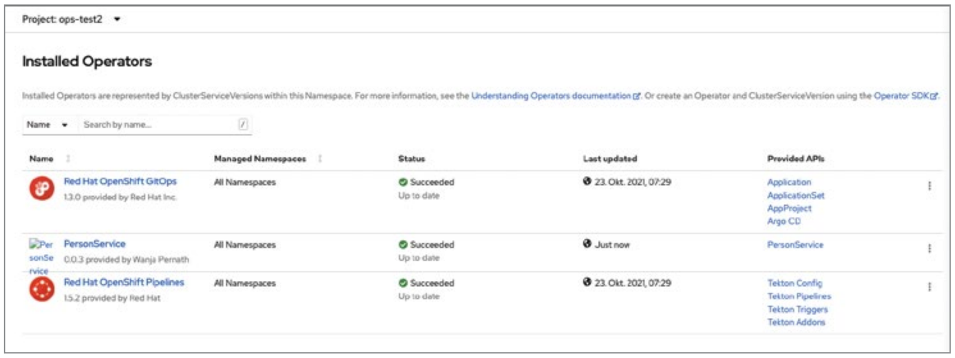
```
The push refers to repository [quay.io/wpernath/person-service-operator-bundle]
413502b46f1b: Pushed
25ab801d3fd7: Pushed
f01cda5511c8: Pushed
v0.0.3: digest:
  sha256:780980b7b7df76faf7be01a7aebcdaaabc4b06dc85cc673f69ab75b58c7dca0c size: 939
```

The bundle image gets pushed to Quay.io as **`quay.io/wpernath/person-service-operator-bundle`**.

Finally, to install the Operator, enter:

```
$ operator-sdk run bundle quay.io/wpernath/person-service-operator-bundle:v0.0.3
```

This command installs the Operator into OpenShift and registers it with the Operator Lifecycle Manager (OLM). After this, you'll be able to watch and manage your Operator via the OpenShift UI, just like the Operators that come with OpenShift (Figure 3-5).



Project: ops-test2

Installed Operators

Installed Operators are represented by ClusterServiceVersions within this Namespace. For more information, see the Understanding Operators documentation. Or create an Operator and ClusterServiceVersion using the Operator SDK.

Name | Search by name...

Name	Managed Namespaces	Status	Last updated	Provided APIs
Red Hat OpenShift GitOps 1.3.0 provided by Red Hat Inc.	All Namespaces	Succeeded Up to date	23. Okt. 2021, 07:29	Application ApplicationSet AppProject Argo CD
PersonService 0.0.3 provided by Wanja Pernath	All Namespaces	Succeeded Up to date	Just now	PersonService
Red Hat OpenShift Pipelines 1.5.2 provided by Red Hat	All Namespaces	Succeeded Up to date	23. Okt. 2021, 07:29	Tekton Config Tekton Pipelines Tekton Triggers Tekton Addons

Figure 3-5: The installed Operator in the OpenShift UI.

You can create a new instance of the service in the UI by clicking Installed Operators→PersonService→Create PersonService, or by executing the following at the command line:

```
$ oc apply -f config/samples/charts_v1alpha1_personservice
personservice.charts.wanja.org/my-person-service1 configured
```

Shortly after requesting the new instance, you should see the deployed Helm chart of your **`PersonService`**.

Cleaning Up

If you want to get rid of this Operator, just enter:

```
$ operator-sdk cleanup person-service-operator --delete-all
```

The **`cleanup`** command needs the project name, which you can find in the **`PROJECT`** file.

Summary of Operators

Creating an Operator just as a replacement for a Helm chart does not really make sense because Operators are much more complex to develop and maintain. And what we've done so far is just the tip of the iceberg. We haven't really touched the Kubernetes API.

However, as soon as you need more influence over creating and maintaining your application and its associated resources, think about building an Operator. Fortunately, the Operator SDK and the documentation can help you with the first steps.

Summary

This chapter described how to build images. You learned more about the various command-line tools, including `skopeo`, `podman`, and `buildah`. You also saw how to create a Helm Chart and a Kubernetes Operator. Finally, 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.

References

[3.1] https://quay.io/

[3.2] https://hub.docker.com

[3.3] https://quay.io/

[3.4] https://www.docker.com

[3.5] https://podman.io/getting-started/installation

[3.6] https://github.com/containers/podman/blob/master/docs/tutorials/mac_win_client.md

[3.7] https://crc.dev/crc/#about-presets_gsg

[3.8] https://podman.io/releases/2022/02/22/podman-release-v4.0.0.html

[3.9] https://buildah.io/

[3.10] https://github.com/containers/buildah

[3.11] https://opencontainers.org/

[3.12] https://github.com/openshift/source-to-image

[3.13] https://github.com/wpernath/book-example

[3.14] https://maven.apache.org/what-is-maven.html

[3.15] https://github.com/containers/podman/blob/master/docs/tutorials/mac_win_client.md

[3.16] https://github.com/containers/skopeo

[3.17] https://helm.sh

[3.18] https://operatorframework.io

[3.19] https://helm.sh/docs/intro/install/

[3.20] https://helm.sh/docs/chart_template_guide/builtin_objects/

[3.21] https://helm.sh/docs/chart_template_guide/getting_started/

[3.22] https://pkg.go.dev/text/template?utm_source=godoc

[3.23] https://golang.org

[3.24] https://helm.sh/docs/topics/charts_hooks/

[3.25] https://github.com/quarkiverse/quarkus-operator-sdk

[3.26] https://sdk.operatorframework.io/docs/installation/

[3.27] https://sdk.operatorframework.io/docs/building-operators/helm/tutorial/

Chapter 4

CI/CD with Tekton Pipelines

Chapters 2 and 3 covered the basics of application packaging with Kustomize, Helm charts, and Operators. We also saw how to handle images and all the metadata required for working with Kubernetes.

This chapter will discuss how to integrate complex tasks, such as building and deploying applications, into Kubernetes using Tekton. Continuous integration and continuous development (CI/CD) are represented in Tekton as *pipelines* that combine all the steps you need to accomplish what you want. And Tekton makes it easy to write a general pipeline that you can adapt to many related tasks.

Tekton and OpenShift Pipelines

Tekton [4.1] is an open source framework to create pipelines for Kubernetes and the cloud. This means that there is no central tool you need to maintain, such as Jenkins. You just have to install a Kubernetes Operator into your Kubernetes cluster to provide custom resource definitions (CRDs). Based on those CRDs, you can create tasks and pipelines to compile, test, deploy, and maintain your application.

OpenShift Pipelines [4.2] is based on Tekton and adds a nice GUI to the OpenShift developer console. The Pipelines Operator is free to use for every OpenShift user.

Tekton Concepts

Tekton has numerous objects, but the architecture is quite easy to understand. The key concepts are:

- **Step**: A process that runs in its own container and can execute whatever the container image provides. A step does not stand independently but must be embedded in a task.
- **Task**: A set of steps running in separate containers (known as "pods" in Kubernetes). A task could be, for example, a compilation process using Maven. One step would be to check the Maven `settings.xml` file. The second step could be to execute the Maven goals (compile, package etc.).
- **Pipeline**: A set of tasks executed either in parallel or (in a simpler case) one after another. A pipeline can be customized through *parameters*.
- **PipelineRun**: A collection of parameters to submit to a pipeline. For instance, a build-and-deploy pipeline might refer to a PipelineRun that contains technical input (for example, a ConfigMap and PersistentVolumeClaim) as well as non-technical parameters (for example, the URL for the Git repository to clone, the name of the target image, etc.)

Internally, Tekton creates a TaskRun object for each task it finds in a PipelineRun.

To summarize: A pipeline contains a list of tasks, each containing a list of steps. One of Tekton's benefits is that you can share tasks and pipelines with other people, because a pipeline just specifies what to do in a given order. So if most of your projects have a similar pipeline, share and reuse it.

Install the tkn CLI

Tekton comes with a command-line tool called `tkn`, which you can easily install on macOS by issuing:

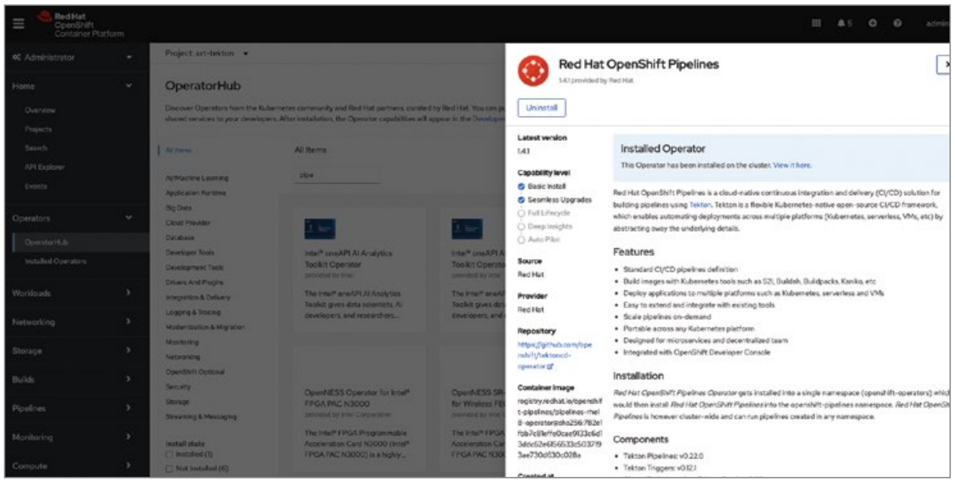
```
$ brew install tektoncd-cli
```

Check the official Tekton homepage to see how to install the tool on other operating systems.

Install OpenShift Pipelines on Red Hat OpenShift

The process in this chapter requires version 1.4.1 or higher of the OpenShift Pipelines Operator. To install that version, you also need a recent 4.7 OpenShift cluster, which you could install, for example, via Red Hat CodeReady Containers [4.3]. Without these tools, you won't have access to workspaces (which you need to define).

To install OpenShift Pipelines, you must be cluster-admin. Go to the OperatorHub, search for "pipelines," and click the **Install** button. There is nothing more to do now, as the Operator maintains everything for you (Figure 4-1).



Red Hat OpenShift Pipelines
1.4.1 provided by Red Hat

Latest version
1.4.1

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

Source
Red Hat

Provider
Red Hat

Repository
https://github.com/openshift/tektoncd-operator

Container Image
registry.redhat.io/openshift-pipelines/pipelines-rhel8-operator@sha256:782e1fbb7c8effe0cae9133c6d13d6c52e6f66533e503719 3ae730d530c028a

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

Red Hat OpenShift Pipelines is a cloud-native continuous integration and delivery (CI/CD) solution for building pipelines using Tekton. Tekton is a flexible Kubernetes-native open-source CI/CD framework, which enables automating deployments across multiple platforms (Kubernetes, serverless, VMs, etc) by abstracting away the underlying details.

Features
- Standard CI/CD pipelines definition
- Build images with Kubernetes tools such as S2I, Buildah, Buildpacks, Kaniko, etc
- Deploy applications to multiple platforms such as Kubernetes, serverless and VMs
- Easy to extend and integrate with existing tools
- Scale pipelines on-demand
- Portable across any Kubernetes platform
- Designed for microservices and decentralized team
- Integrated with OpenShift Developer Console

Installation
Red Hat OpenShift Pipelines Operator gets installed into a single namespace (openshift-operators) which would then install *Red Hat OpenShift Pipelines* into the openshift-pipelines namespace. *Red Hat OpenShift Pipelines* is however cluster-wide and can run pipelines created in any namespace.

Components
- Tekton Pipelines: v0.22.0
- Tekton Triggers: v0.12.1

Figure 4-1: Using the OpenShift UI to install OpenShift Pipelines Operator.

After a while, you'll notice a new GUI entry in both the Administrator and the Developer UI (Figure 4-2).

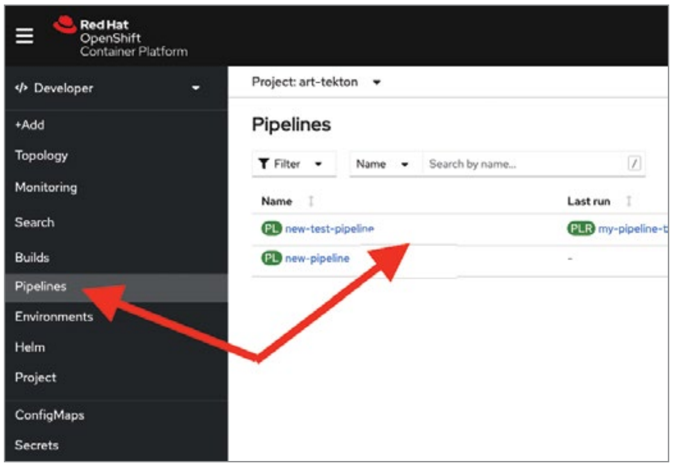


Figure 4-2: New UI entries in the OpenShift GUI after you've installed the Pipelines Operator.

Create a pipeline for person-service

For our person-service [4.4], we will create a Tekton pipeline for a simple deployment task. The pipeline compiles the source, creates a Docker image based on Jib [4.5], pushes the image to Quay.io [4.6], and uses Kustomize [4.7] to apply that image to an OpenShift project called `book-tekton`.

Sounds easy? It is. Well, mostly.

Why use Jib to build the container image? Well, that's easily explained: Right now, there are three different container image build strategies available with Quarkus:

- Docker
- Source-to-Image (S2I)
- Jib

The Docker strategy uses the `docker` binary to build the container image. But the `docker` binary is not available inside a Kubernetes cluster (as mentioned in Chapter 3), because Docker is too heavyweight and requires root privileges to run the daemon.

S2I requires creating `BuildConfig`,`DeploymentConfig`, and `ImageStream`\ objects specific to OpenShift, but these are not available in vanilla Kubernetes clusters.

So in order to stay vendor-independent, we have to use Jib for this use case.

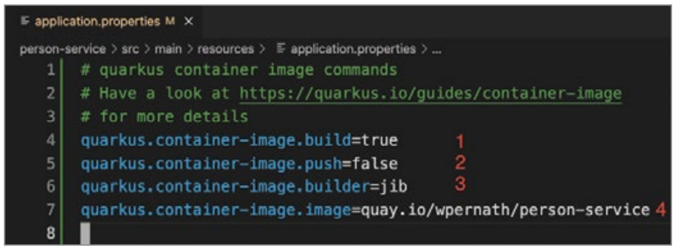
Of course, you could also use other tools to create your container image inside Kubernetes. But to keep this Tekton example clean and simple, we are reusing what Quarkus provides. So we can simply set a few Quarkus properties in `application.properties` to define how Quarkus should package the application. Then we'll be able to use exactly *one* Tekton task to compile, package, and push the application to an external registry.

Make sure that your Quarkus application is using the required Quarkus extension `container-image-jib`. If your `pom.xml` file does not include the `quarkus-container-image-jib` dependency, add it by executing:

```
$ mvn quarkus:add-extension -Dextensions="container-image-jib"
[INFO] Scanning for projects...
[INFO]
[INFO] -------------------< org.wanja.book:person-service >--------------------
[INFO] Building person-service 1.0.0
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- quarkus-maven-plugin:2.4.2.Final:add-extension (default-cli)
  @ person-service ---
[INFO] Looking for the newly published extensions in registry.quarkus.io
[INFO] [SUCCESS] ✅  Extension io.quarkus:quarkus-container-image-jib has been
  installed
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  5.817 s
[INFO] Finished at: 2021-11-22T09:34:05+01:00
[INFO] ------------------------------------------------------------------------
```

Then have a look at Figure 4-3 to see what properties are needed for Quarkus to build, package, and push the image. The following properties need to be set in `application.properties`:

- `quarkus.container-image.build`: Set this to `true` to ensure that a `mvn package` command builds a container image.
- `quarkus.container-image.push`: This is optional and required only if you want to push the image directly to the registry. I don't intend to do so, so I set the value to `false`.
- `quarkus.container-image.builder`: This property selects the method of building the container image. We set the value to `jib` to use Jib [4.8].
- `quarkus.container-image.image`: Set this to the complete name of the image to be built, including the domain name.


application.properties M ×
person-service > src > main > resources > application.properties > ...

```
# quarkus container image commands
# Have a look at https://quarkus.io/guides/container-image
# for more details
quarkus.container-image.build=true          1
quarkus.container-image.push=false          2
quarkus.container-image.builder=jib         3
quarkus.container-image.image=quay.io/wpernath/person-service 4
```

Figure 4-3: Application properties of the person-service.

Now check out the source code [4.9], have a look at `person-service/src/main/resources/application.properties`, change the image property to meet your needs, and issue:

```
$ mvn clean package -DskipTests
```

This command compiles the sources and builds the container image. If you want to push the resulting image to your registry, simply call:

```
$ mvn package -DskipTests -Dquarkus.container-image.push=true
```

After a while, Quarkus will generate and push your image to your registry. In my case, it's `quay.io/wpernath/person-service`.

Inventory Check: What Do We Need?

To create our use case, you need the following tools:

- `git`: To fetch the source from GitHub.
- `maven`: To reuse most of what Quarkus provides.
- `kustomize`: To change our Deployment to point to the new image.
- OpenShift client: To apply the changes we've made in the previous steps.

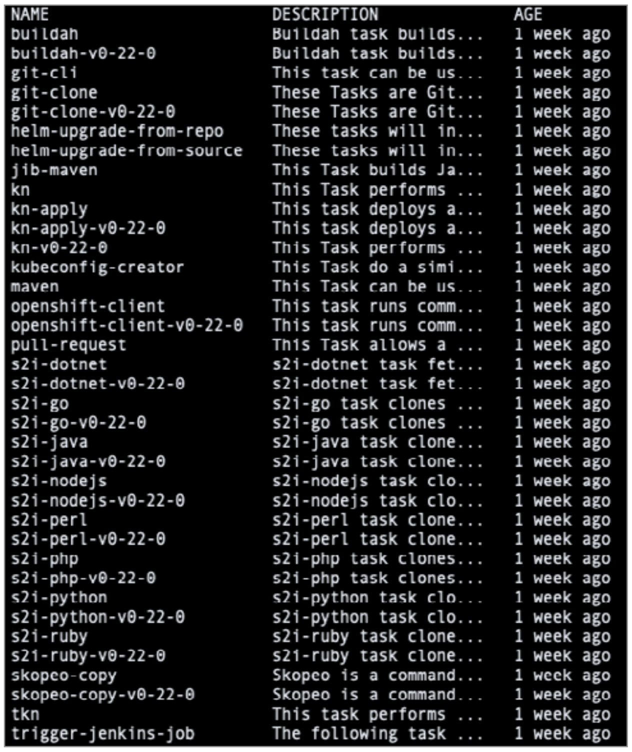
Some of them can be set up for you by OpenShift. So now log into your OpenShift cluster, create a new project, and list all the available ClusterTasks:

```
$ oc login .....
$ oc new-project book-tekton
$ tkn ct list
```

Figure 4-4 shows all the available ClusterTasks created after installing the OpenShift Pipelines Operator. It seems you have most of what we need:

- `git-clone`
- `maven`
- `openshift-client`

Note: What is the difference between a task and a ClusterTask? A ClusterTask is available globally in all projects, whereas a task is available only locally per project and must be installed into each project where you want to use it.



```
NAME                        DESCRIPTION              AGE
buildah                     Buildah task builds...   1 week ago
buildah-v0-22-0             Buildah task builds...   1 week ago
git-cli                     This task can be us...   1 week ago
git-clone                   These Tasks are Git...   1 week ago
git-clone-v0-22-0           These Tasks are Git...   1 week ago
helm-upgrade-from-repo      These tasks will in...   1 week ago
helm-upgrade-from-source    These tasks will in...   1 week ago
jib-maven                   This Task builds Ja...   1 week ago
kn                          This Task performs ...   1 week ago
kn-apply                    This task deploys a...   1 week ago
kn-apply-v0-22-0            This task deploys a...   1 week ago
kn-v0-22-0                  This Task performs ...   1 week ago
kubeconfig-creator          This Task do a simi...   1 week ago
maven                       This Task can be us...   1 week ago
openshift-client            This task runs comm...   1 week ago
openshift-client-v0-22-0    This task runs comm...   1 week ago
pull-request                This Task allows a ...   1 week ago
s2i-dotnet                  s2i-dotnet task fet...   1 week ago
s2i-dotnet-v0-22-0          s2i-dotnet task fet...   1 week ago
s2i-go                      s2i-go task clones ...   1 week ago
s2i-go-v0-22-0              s2i-go task clones ...   1 week ago
s2i-java                    s2i-java task clone...   1 week ago
s2i-java-v0-22-0            s2i-java task clone...   1 week ago
s2i-nodejs                  s2i-nodejs task clo...   1 week ago
s2i-nodejs-v0-22-0          s2i-nodejs task clo...   1 week ago
s2i-perl                    s2i-perl task clone...   1 week ago
s2i-perl-v0-22-0            s2i-perl task clone...   1 week ago
s2i-php                     s2i-php task clones...   1 week ago
s2i-php-v0-22-0             s2i-php task clones...   1 week ago
s2i-python                  s2i-python task clo...   1 week ago
s2i-python-v0-22-0          s2i-python task clo...   1 week ago
s2i-ruby                    s2i-ruby task clone...   1 week ago
s2i-ruby-v0-22-0            s2i-ruby task clone...   1 week ago
skopeo-copy                 Skopeo is a command...   1 week ago
skopeo-copy-v0-22-0         Skopeo is a command...   1 week ago
tkn                         This task performs ...   1 week ago
trigger-jenkins-job         The following task ...   1 week ago
```

Figure 4-4: Available ClusterTasks in OpenShift after installation of the Pipelines Operator.

You're missing just the **`kustomize`** task. You'll create one later, but first, we want to take care of the rest of the tasks.

Analyzing the Necessary Tasks

If you want to have a look at the structure of a task, you can easily do so by executing the following command:

```
$ tkn ct describe <task-name>
```

The output explains all the task parameters, together with other necessary information such as its inputs and outputs.

By specifying the **`-o yaml`** parameter, you can view the YAML source definition of the task.

The **`git-clone`** task allows many parameters, but most of them are optional. You just have to specify **`git-url`** and **`git-revision`**. You also have to specify a workspace for the task.

What are Workspaces?

Remember that Tekton is running every task (and all steps inside a task) as a separate pod. If the application running on the pod writes to some random folder, nothing gets really stored. So if we want one step of the pipeline to read and write data that is shared with other steps (and yes, we do want this), we have to find a way to do that.

That is where workspaces come in. They could be a PersistentVolumeClaim (PVC), a ConfigMap, etc. A task that either requires a place to store data (such as git-clone) or needs to have access to data coming from a previous step (such as Maven) defines a workspace. If the task is embedded into a pipeline, the workspace is defined for every task in the pipeline. The PipelineRun (or, in the case of a single running task, the TaskRun) finally creates the mapping between the defined workspace and corresponding storage.

In our example, we need two workspaces:

- A PVC where the git-clone task clones the source code to and from the place where the Maven task compiles the source.
- A ConfigMap with the `maven-settings` file you need in your environment.

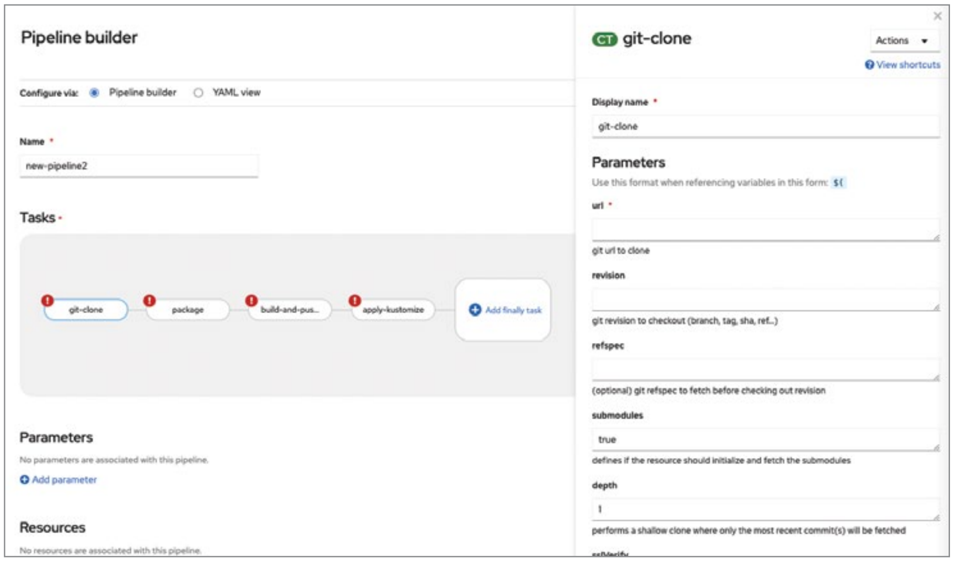
Building the Pipeline

Once you know what tasks you need to build your pipeline, you can start creating it. There are two ways of doing so:

- Via a code editor as a YAML file.
- In the OpenShift developer console.

Note: Remember that you should have at least version 1.4.1 of the OpenShift Pipelines Operator installed.

For your first try, I recommend building the pipeline via the graphical OpenShift developer console (Figure 4-5). Then export and see what it looks like. The rest of this section focuses on that activity.



Pipeline builder

Configure via: [✓ Pipeline builder] YAML view

Name *
new-pipeline2

Tasks *

Parameters
No parameters are associated with this pipeline.
Add parameter

Resources
No resources are associated with this pipeline.

CT git-clone

Display name *
git-clone

Parameters
Use this format when referencing variables in this form: ${

url *

git url to clone

revision

git revision to checkout (branch, tag, sha, ref...)

refspec

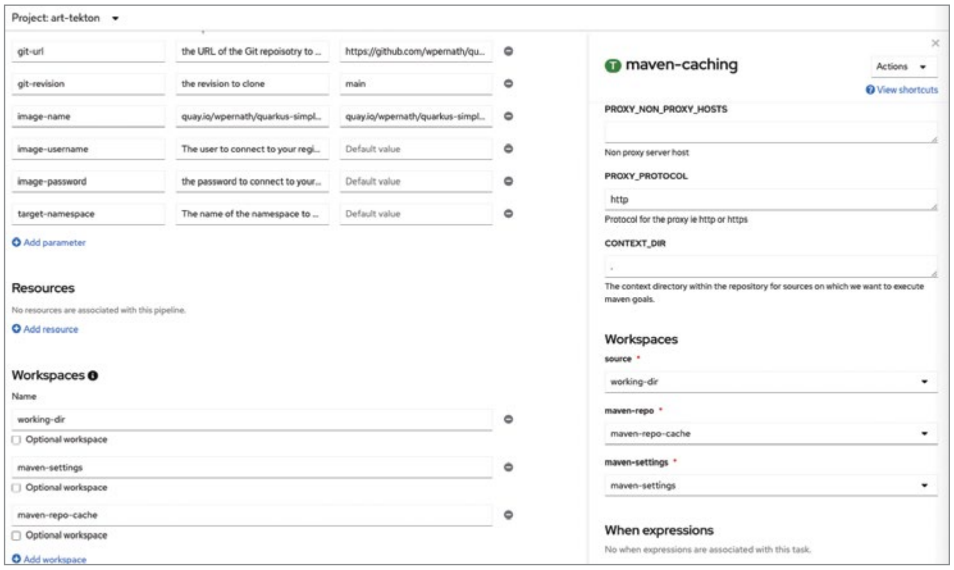
(optional) git refspec to fetch before checking out revision

submodules
true
defines if the resource should initialize and fetch the submodules

depth
1
performs a shallow clone where only the most recent commit(s) will be fetched

Figure 4-5: Pipeline builder.

You have to provide parameters to each task and link the required workspaces to the tasks. You can easily do that by using the GUI (Figure 4-6).


Project: art-tekton

git-url	the URL of the Git repoisotry to ...	https://github.com/wpernath/qu...
git-revision	the revision to clone	main
image-name	quay.io/wpernath/quarkus-simpl...	quay.io/wpernath/quarkus-simpl...
image-username	The user to connect to your regi...	Default value
image-password	the password to connect to your...	Default value
target-namespace	The name of the namespace to ...	Default value

Add parameter

Resources

No resources are associated with this pipeline.

Add resource

Workspaces

Name

working-dir

Optional workspace

maven-settings

Optional workspace

maven-repo-cache

Optional workspace

Add workspace

maven-caching

PROXY_NON_PROXY_HOSTS

Non proxy server host

PROXY_PROTOCOL

http

Protocol for the proxy ie http or https

CONTEXT_DIR

.

The context directory within the repository for sources on which we want to execute maven goals.

Workspaces

source *: working-dir

maven-repo *: maven-repo-cache

maven-settings *: maven-settings

When expressions

No when expressions are associated with this task.

Figure 4-6: Linking the workspaces from Pipeline to task.

You need to use the `maven` task twice, using the `package` goal:

1. To simply compile the source code.
2. To execute the `package` goal with the following parameters that instruct quarkus to build and push the image [4.10]:
   - `-Dquarkus.container-image.push=true`
   - `-Dquarkus.container-image.builder=jib`
   - `-Dquarkus.container-image.image=$(params.image-name)`
   - `-Dquarkus.container-image.username=$(params.image-username)`
   - `-Dquarkus.container-image.password=$(params.image-password)`

Once you've done all that and have clicked the Save button, you're able to export the YAML file by executing:

```
$ oc get pipeline/build-and-push-image -o yaml >
  tekton/pipelines/build-and-push-image.yaml
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: build-and-push-image
spec:
  params:
  - default: https://github.com/wpernath/book-example.git
    description: the URL of the Git repoisotry to build
    name: git-url
    type: string
....
```

You can easily re-import the pipeline file by executing:

```
$ oc apply -f tekton/pipelines/build-and-push-image.yaml
```

Task Parameter Placement

One of Tekton's primary aims is to provide tasks and pipelines that are as reusable as possible. This means making each task as general-purpose as possible.

If you're providing the necessary parameters directly to each task, you might repeat the settings over and over again. For example, in our case, we are using the Maven task for compiling, packaging, image generation, and pushing. Here it makes sense to take the parameters out of the specification of each task. Instead, put them on the pipeline level under a property called `params` (as shown in the following listing) and refer to them inside the corresponding task by specifying them by their name in the syntax `$(params.parameter-name)`.

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: build-and-push-image
spec:
  params:
    - default: 'https://github.com/wpernath/book-example.git'
      description: Source to the GIT
      name: git-url
      type: string
    - default: main
      description: revision to be used
      name: git-revision
      type: string
[...]
  tasks:
    - name: git-clone
      params:
        - name: url
          value: $(params.git-url)
        - name: revision
          value: $(params.git-revision)
[...]
      taskRef:
        kind: ClusterTask
        name: git-clone
      workspaces:
        - name: output
          workspace: shared-workspace
[...]
```

Creating a New Task: Kustomize

Remember that our default OpenShift Pipelines Operator installation didn't include Kustomize. Because we want to use it to apply the new image to our Deployment, we have to look for a proper task in Tekton Hub [4.11]. Unfortunately, there doesn't seem to be one available, so we have to create our own.

We first need to have a proper image that contains the `kustomize` executable. The `Dockerfile` for this project is available in the kustomize-ubi repository on GitHub [4.12], and the image is available in its repository on Quay.io [4.13].

Now let's create a new Tekton task:

```
apiVersion: tekton.dev/v1beta1
kind: Task
```

```
metadata:
  name: kustomize
  labels:
    app.kubernetes.io/version: "0.4"
  annotations:
    tekton.dev/pipelines.minVersion: "0.12.1"
    tekton.dev/tags: build-tool
spec:
  description: >-
    This task can be used to execute kustomze build scripts and to apply the
      changes via oc apply -f
  workspaces:
    - name: source
      description: The workspace holding the cloned and compiled quarkus source.
  params:
    - name: kustomize-dir
      description: Where should kustomize look for kustomization in source?
    - name: target-namespace
      description: Where to apply the kustomization to
    - name: image-name
      description: Which image to use. Kustomize is taking care of it
  steps:
    - name: build
      image: quay.io/wpernath/kustomize-ubi:latest
      workingDir: $(workspaces.source.path)
      script: |

        cd $(workspaces.source.path)/$(params.kustomize-dir)

        DIGEST=$(cat $(workspaces.source.path)/target/jib-image.digest)

        kustomize edit set image
          quay.io/wpernath/simple-quarkus:latest=$(params.image-name)@$DIGEST

        kustomize build $(workspaces.source.path)/$(params.kustomize-dir) >
          $(workspaces.source.path)/target/kustomized.yaml

    - name: apply
      image: 'image-registry.openshift-image-registry.svc:5000/openshift/cli:latest'
      workingDir: $(workspaces.source.path)
      script: |
        oc apply -f $(workspaces.source.path)/target/kustomized.yaml -n
          $(params.target-namespace)
```

Paste this text into a new file called `kustomize-task.yaml`. As you can see from the file's contents, this task requires a workspace called `source` and three parameters: `kustomize-dir`, `target-namespace`, and `image-name`. The task contains two steps: `build` and `apply`.

The build step uses the Kustomize image to set the new image and digest. The apply step uses the internal OpenShift CLI image to apply the Kustomize-created files in the `target-namespace` namespace.

To load the `kustomize-task.yaml` file into your current OpenShift project, simply execute:

```
$ oc apply -f kustomize-task.yaml
task.tekton.dev/kustomize configured
```

Putting It All Together

We have created a pipeline containing four tasks: `git-clone`, `package`, `build-and-push-image`, and `apply-kustomize`. We then provided the necessary parameters to each task and the pipeline and connected workspaces to it.

Now we have to create the PersistentVolumeClaim (PVC) and a ConfigMap named `maven-settings`, which will then be used by the corresponding PipelineRun.

Creating a maven-settings ConfigMap

If you have a working `maven-settings` file, you can easily reuse it with the Maven task. Simply create it via:

```
$ oc create cm maven-settings --from-file=/your-maven-settings --dry-run=client
  -o yaml > maven-settings-cm.yaml
```

If you need to edit the ConfigMap, feel free to do it right now and then execute to import the ConfigMap into your current project:

```
$ oc apply -f maven-settings-cm.yaml
```

Creating a PersistentVolumeClaim

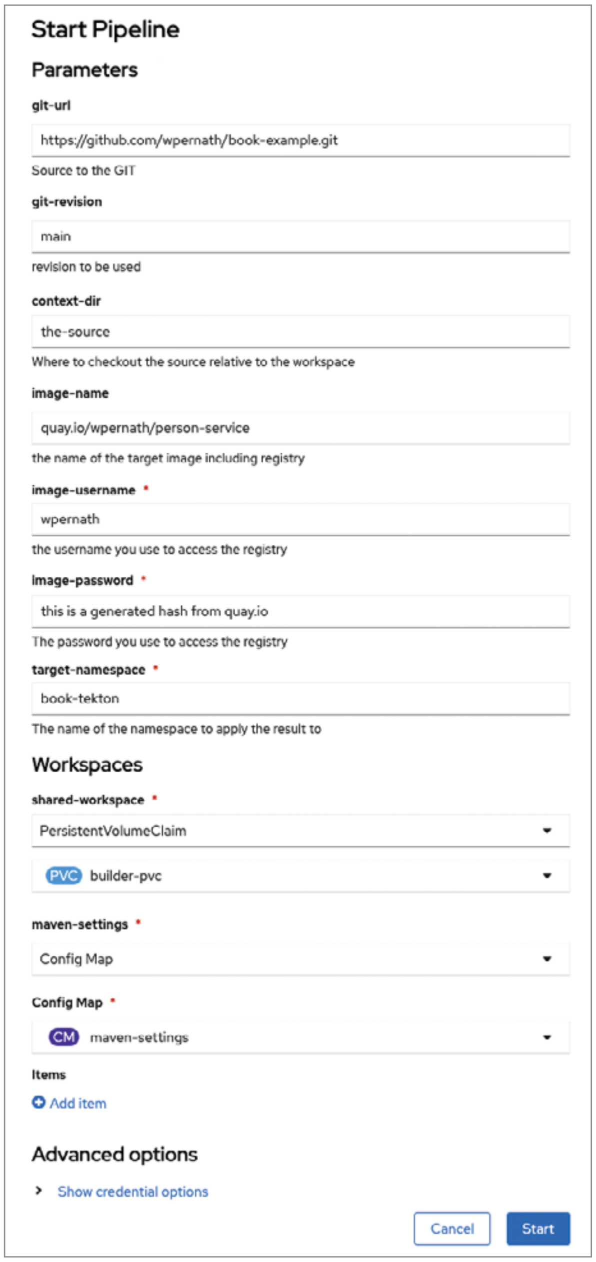
Create a new file with the following content and execute `oc apply -f` to import it into your project:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: builder-pvc
spec:
  resources:
    requests:
      storage: 10Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
```

This file reserves a PVC with the name `builder-pvc` and a requested storage of 10GB. It's important to use `persistentVolumeReclaimPolicy: Retain` here, as we want to reuse build artifacts from the previous builds—more on this requirement later in this chapter.

Run the Pipeline

Once you have imported all your artifacts into your current project, you can run the pipeline. To do so, click on the **Pipelines** entry on the left side of the Developer Perspective of OpenShift, choose your created pipeline, and select **Start** from the **Actions** menu on the right side. After you've filled in all necessary parameters (Figure 4-7), you can start the PipelineRun.



Start Pipeline

Parameters

git-url	https://github.com/wpernath/book-example.git
	Source to the GIT
git-revision	main
	revision to be used
context-dir	the-source
	Where to checkout the source relative to the workspace
image-name	quay.io/wpernath/person-service
	the name of the target image including registry
image-username *	wpernath
	the username you use to access the registry
image-password *	this is a generated hash from quay.io
	The password you use to access the registry
target-namespace *	book-tekton
	The name of the namespace to apply the result to

Workspaces

shared-workspace *	PersistentVolumeClaim
	PVC builder-pvc
maven-settings *	Config Map
Config Map *	CM maven-settings
Items	Add item

Advanced options

Figure 4-7: Starting the pipeline with all parameters.

The Logs and Events cards of the OpenShift Pipeline Editor show, well, all the logs and events. If you prefer to view these things from the command line, use `tkn` to follow the logs of the PipelineRun:

```
$ tkn pr
```

The output shows the available actions for PipelineRuns. To list each PipelineRun and its status, enter:

```
$ tkn pr list
NAME                                      STARTED        DURATION     STATUS
build-and-push-image-run-20211123-091039  1 minute ago   54 seconds   Succeeded
build-and-push-image-run-20211122-200911  13 hours ago   2 minutes   Succeeded
build-and-push-image-ru0vni               13 hours ago   8 minutes    Failed
```

To follow the logs of the last run, execute:

```
$ tkn pr logs -f -L
```

If you omit the `-L` option, `tkn` lets you choose from the list of PipelineRuns. You can also log, list, cancel, and delete PipelineRuns.

Visual Code has a Tekton Pipeline extension that you can also use to edit, build, and execute pipelines.

Creating a PipelineRun Object

In order to start the pipeline via a shell (or from any other application you're using for CI/CD), you need to create a PipelineRun object, which looks like the following:

```
apiVersion: tekton.dev/v1beta1
kind: PipelineRun
metadata:
  name: $PIPELINE-run-$(date "+%Y%m%d-%H%M%S")
spec:
  params:
    - name: git-url
      value: https://github.com/wpernath/book-example.git
    - name: git-revision
      value: main
    - name: context-dir
      value: the-source
    - name: image-name
      value: quay.io/wpernath/person-service
    - name: image-username
      value: wpernath
    - name: image-password
      value: *****
    - name: target-namespace
      value: book-tekton
  workspaces:
    - name: shared-workspace
      persistentVolumeClaim:
        claimName: builder-pvc
    - configMap:
        name: maven-settings
      name: maven-settings
  pipelineRef:
    name: build-and-push-image
  serviceAccountName: pipeline
```

Most of the properties of this object are self-explanatory. Just one word on the `serviceAccountName` property: Each PipelineRun runs under a given service account, which means that all pods started along the pipeline run inside this security context.

OpenShift Pipelines creates a default service account for you called `pipeline`. If you have secrets that you want to make available to your PipelineRun, you have to connect them with the service account name. But this requirement is out of scope for this chapter of the book; we'll return to secrets in the next chapter.

The `tekton/pipeline.sh` shell script creates a full version of this PipelineRun based on input parameters.

Optimizing the Pipeline

As earlier output from the logs showed, the first pipeline run takes quite a long time to finish: in my case, approximately eight minutes. The second pipeline still took two minutes. I was running the pipelines on a home server with modest resources. When you use Tekton on your build farms, run times should be much lower because you're running on dedicated server hardware.

But still, the pipelines at this point take way too long.

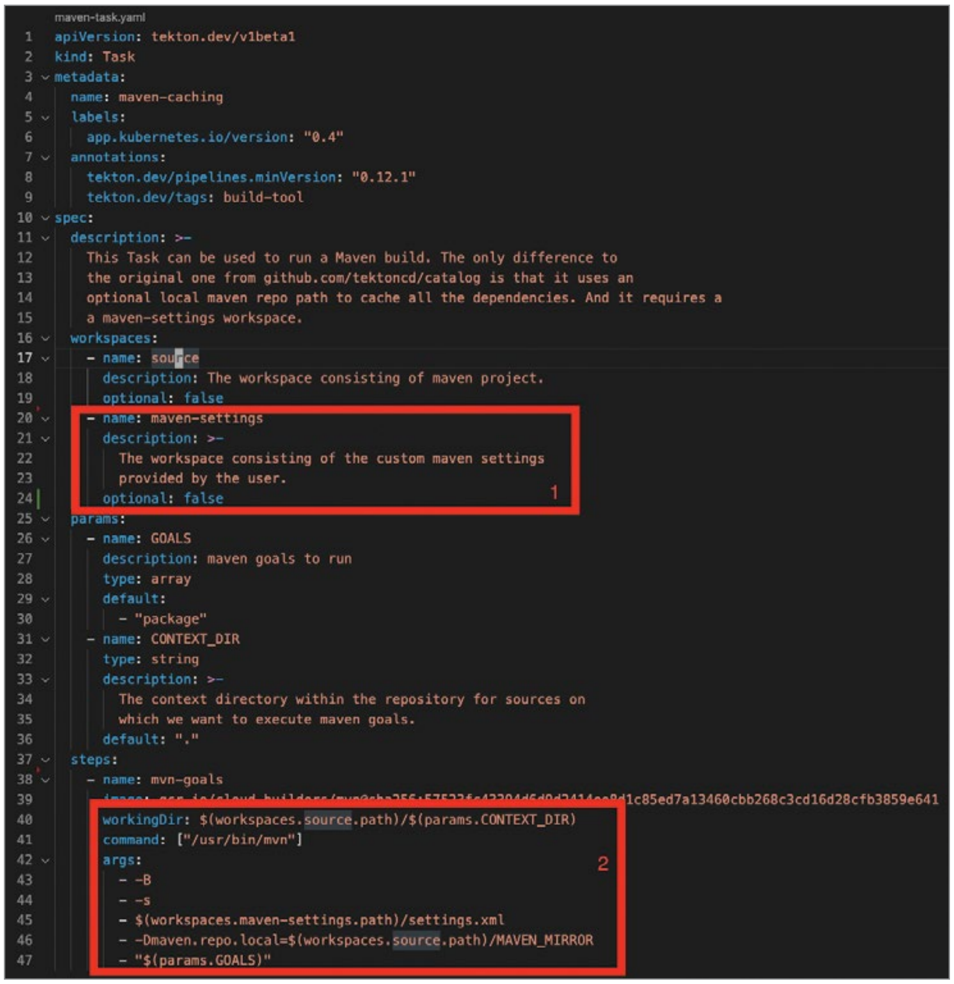
Looking at the logs, you can see that the `maven` task takes a long time to finish. This is because Maven is downloading the necessary artifacts again and again on every run. Depending on your Internet connection, this takes some time, even if you're using a local Maven mirror.

On your developer machine, Maven uses the `$HOME/.m2` folder as a cache for the artifacts. The same is true when you run Maven from a task. However, because each PipelineRun runs on a separate set of pods, `$HOME/.m2` is not properly defined, which means the whole cache gets invalidated once the PipelineRun finishes.

Maven allows you to specify `-Dmaven.repo.local` to provide a different path to a local cache. This option is what you can use to solve the problem.

I have created a new Maven task (`maven-caching`), which you can find in the book's example repository [4.14]. The file was originally just a copy of the one from Tekton Hub. But then I decided to remove the init step, which was building a `maven-settings.xml` file based on some input parameters. Instead, I removed most of the parameters and added a ConfigMap with must-have `maven-settings`. I believe this makes everything much easier.

As Figure 4-8 shows, you now have only two parameters: `GOALS` and `CONTEXT_DIR`.



```yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: maven-caching
  labels:
    app.kubernetes.io/version: "0.4"
  annotations:
    tekton.dev/pipelines.minVersion: "0.12.1"
    tekton.dev/tags: build-tool
spec:
  description: >-
    This Task can be used to run a Maven build. The only difference to
    the original one from github.com/tektoncd/catalog is that it uses an
    optional local maven repo path to cache all the dependencies. And it requires a
    a maven-settings workspace.
  workspaces:
    - name: source
      description: The workspace consisting of maven project.
      optional: false
    - name: maven-settings
      description: >-
        The workspace consisting of the custom maven settings
        provided by the user.
      optional: false
  params:
    - name: GOALS
      description: maven goals to run
      type: array
      default:
        - "package"
    - name: CONTEXT_DIR
      type: string
      description: >-
        The context directory within the repository for sources on
        which we want to execute maven goals.
      default: "."
  steps:
    - name: mvn-goals
      image: gcr.io/cloud-builders/mvn@sha256:57523fc43394d6d9d2414ee8d1c85ed7a13460cbb268c3cd16d28cfb3859e641
      workingDir: $(workspaces.source.path)/$(params.CONTEXT_DIR)
      command: ["/usr/bin/mvn"]
      args:
        - -B
        - -s
        - $(workspaces.maven-settings.path)/settings.xml
        - -Dmaven.repo.local=$(workspaces.source.path)/MAVEN_MIRROR
        - "$(params.GOALS)"
```

Figure 4-8. Simplified Maven task.

The important properties for the **`maven`** call is shown in the second red box of Figure 4-8. These properties call **`maven`** with the **`maven-settings`** property and with the parameter indicating where to store the downloaded artifacts.

One note on artifact storage: When I was testing this example, I realized that if the **`git-clone`** task clones the source to the root directory of the PVC (when no **`subdirectory`** parameter is given on task execution), the next start of the pipeline will delete everything from the PVC again. And in that case, we once again have no artifact cache.

So you have to provide a **`subdirectory`** parameter (in my case, I used a global property called **`the-source`**) and provide exactly the same value to the **`CONTEXT_DIR`** parameter in the **`maven`** calls.

The changes discussed in this section reduce our **`maven`** calls dramatically, in my case from 8 minutes to 54 seconds:

```
$ tkn pr list
NAME                      STARTED        DURATION     STATUS
build-and-push-image-123  1 minute ago   54 seconds   Succeeded
build-and-push-image-ru0  13 hours ago   8 minutes    Succeeded
```

Summary

Tekton is a powerful tool for creating CI/CD pipelines. Because it is based on Kubernetes, it uses extensive concepts from Kubernetes and reduces the maintenance of the tool itself. If you want to start your first pipeline quickly, try to use the OpenShift Developer UI, which you get for free if you're installing the Operator. This gives you a nice base to start your tests. However, at some point—especially when it comes to optimizations—you need a proper editor to code your pipelines.

One of the biggest advantages of Tekton over other CI/CD tools such as Jenkins is that you can reuse all your work for other projects and applications. If you want to standardize the way your pipelines work, build one pipeline and simply specify different sets of parameters for different situations. PipelineRun objects make this possible. You can easily reuse the pipeline we created in this chapter for all Quarkus-generated applications. Just change the `git-url` and `image-name` parameters. Isn't that great?

And even if you're not satisfied with all the tasks you get from Tekton Hub, use them as bases and build your own iterations out of them, as we did with the optimized Maven task and the Kustomize task in this chapter.

Tekton is not the easiest technology available for CI/CD pipelines, but it is definitely one of the most flexible.

However, we have not even talked about Tekton security [4.15] and how we can provide, for example, secrets to access your Git repository or the image repository. And we cheated a little bit with image generation because we used the mechanism Quarkus provides. There are other ways of creating images using a dedicated Buildah task.

The next chapter discusses Tekton security, GitOps, and Argo CD.

References

[4.1] https://tekton.dev

[4.2] https://cloud.redhat.com/blog/introducing-openshift-pipelines

[4.3] https://github.com/code-ready/crc

[4.4] https://github.com/wpernath/book-example/tree/main/person-service

[4.5] https://github.com/GoogleContainerTools/jib

[4.6] https://quay.io/repository/wpernath/quarkus-simple-wow

[4.7] https://www.opensourcerers.org/2021/04/26/automated-application-packaging-and-distribution-with-openshift-part-12/

[4.8] https://github.com/GoogleContainerTools/jib

[4.9] https://github.com/wpernath/book-example

[4.10] https://quarkus.io/guides/container-image#quarkus-container-image-jib_quarkus.jib.base-jvm-image

[4.11] https://hub.tekton.dev

[4.12] https://github.com/wpernath/kustomize-ubi

[4.13] https://quay.io/repository/wpernath/kustomize-ubi

[4.14] https://raw.githubusercontent.com/wpernath/book-example/main/tekton/tasks/maven-task.yaml

[4.15] https://tekton.dev/docs/pipelines/auth/

Chapter 5

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 [5.1] in your project as well as 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?" I heard this the first time I talked about GitOps during a customer engagement.

The short answer is that DevOps is a cultural change in your enterprise, meaning developers and operations people should talk to each other instead of secretly doing their work 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 should already maintain 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 had 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 often ask about GitOps is this: Is it just another way of doing CI/CD? The answer to this question is no. GitOps takes care of only the CD, 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 Figure 5-1):

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.

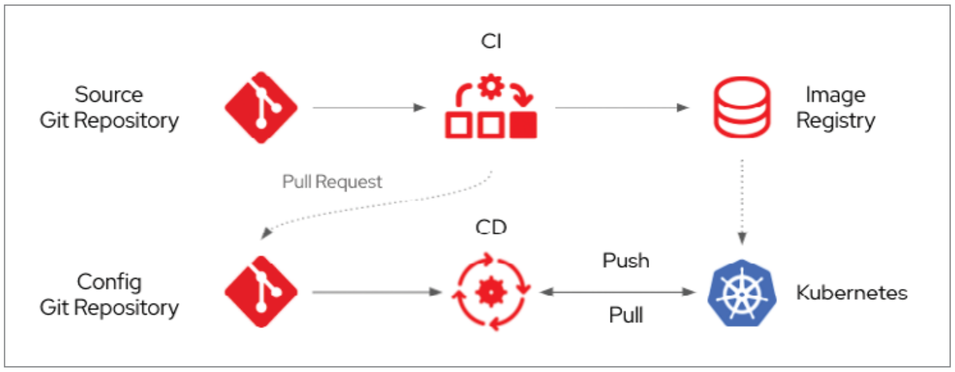


Figure 5-1: The GitOps delivery model.

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 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.

Use Case: Implementing GitOps for our person-service Application

We've been using the person-service [5.2] throughout this book. Let's now 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 [5.3]) on OpenShift 4.9 (via Red Hat OpenShift Local [5.4]). We are going to use Tekton to build a pipeline [5.5], which updates the person-service-config [5.6] Git repository with the latest image digest of the build. Argo CD should then detect the changes and start synchronizing 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`.

Create a new remote Git repository and copy the URL (for example, `https://github.com/wpernath/person-service-config.git`). Then jump to the shell, create a new empty folder somewhere, and issue the following commands:

```
$ mkdir person-service-config
$ git init -b main
$ git remote add origin https://github.com/wpernath/person-service-config.git
```

One of the main concepts behind GitOps is to represent your application's configuration and build parameters 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 [5.7], we will create a new repository containing the artifacts that we built in earlier chapters using Kustomize:

```
$ tree
└── config
    ├── base
    │   ├── config-map.yaml
    │   ├── deployment.yaml
    │   ├── kustomization.yaml
    │   ├── route.yaml
    │   └── service.yaml
    └── overlays
        ├── dev
        │   └── kustomization.yaml
        ├── prod
        │   └── kustomization.yaml
        └── stage
            ├── apply-health-checks.yaml
            ├── change-env-value.yaml
            └── kustomization.yaml

6 directories, 10 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 hundreds 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:

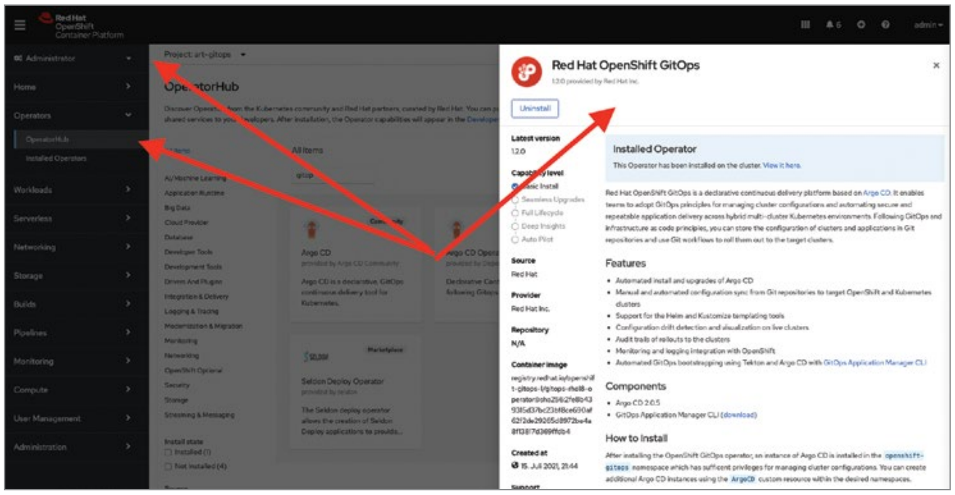
```
$ mkdir config
$ cp -r ../book-example/kustomize-ext/ config/
$ git add config
$ git commit -am 'initial commit'
$ git push -u origin main
```

Note: The original `kustomization.yaml` file already contains an image section. You should remove this first.

Install the OpenShift GitOps Operator

Because the OpenShift GitOps Operator is offered free of charge to OpenShift users and comes well preconfigured, I am focusing on its use. If you want to bypass the Operator and dig into Argo CD installation, feel free to have a look at the official guides [5.8].

The OpenShift GitOps Operator can easily be installed in OpenShift [5.9]. 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 (Figure 5-2). Start typing “gitops” in the search field and select the GitOps Operator when its panel is shown.



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@sha256:2fe8b439385d37bc23bf8ce690af62f2de29265d8972be4a8f13817d369ffcb4

Created at
15. Juli 2021, 21:44

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.

Figure 5-2: Installing the OpenShift GitOps Operator.

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.

At the time of this writing, Argo CD is not yet configured to use OpenShift authentication, so you have to get the password of the admin user by getting the value of the `openshift-gitops-cluster` secret in the `openshift-gitops` namespace:

```
$ oc get secret openshift-gitops-cluster -n openshift-gitops
  -ojsonpath='{.data.admin\.password}' | base64 -d
```

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
```

Create an Argo CD Application

The easiest way to create a new Argo application is by using the GUI provided by Argo CD (Figure 5-3).

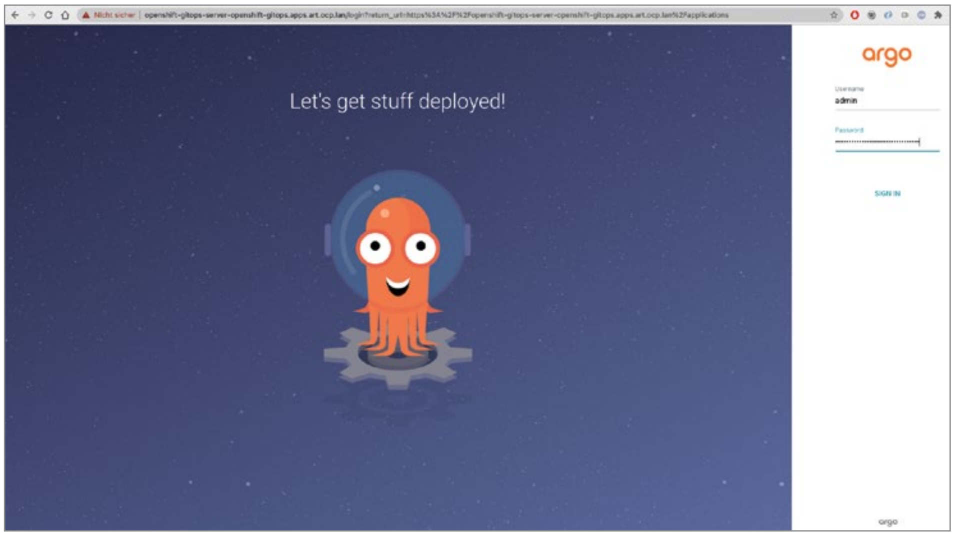
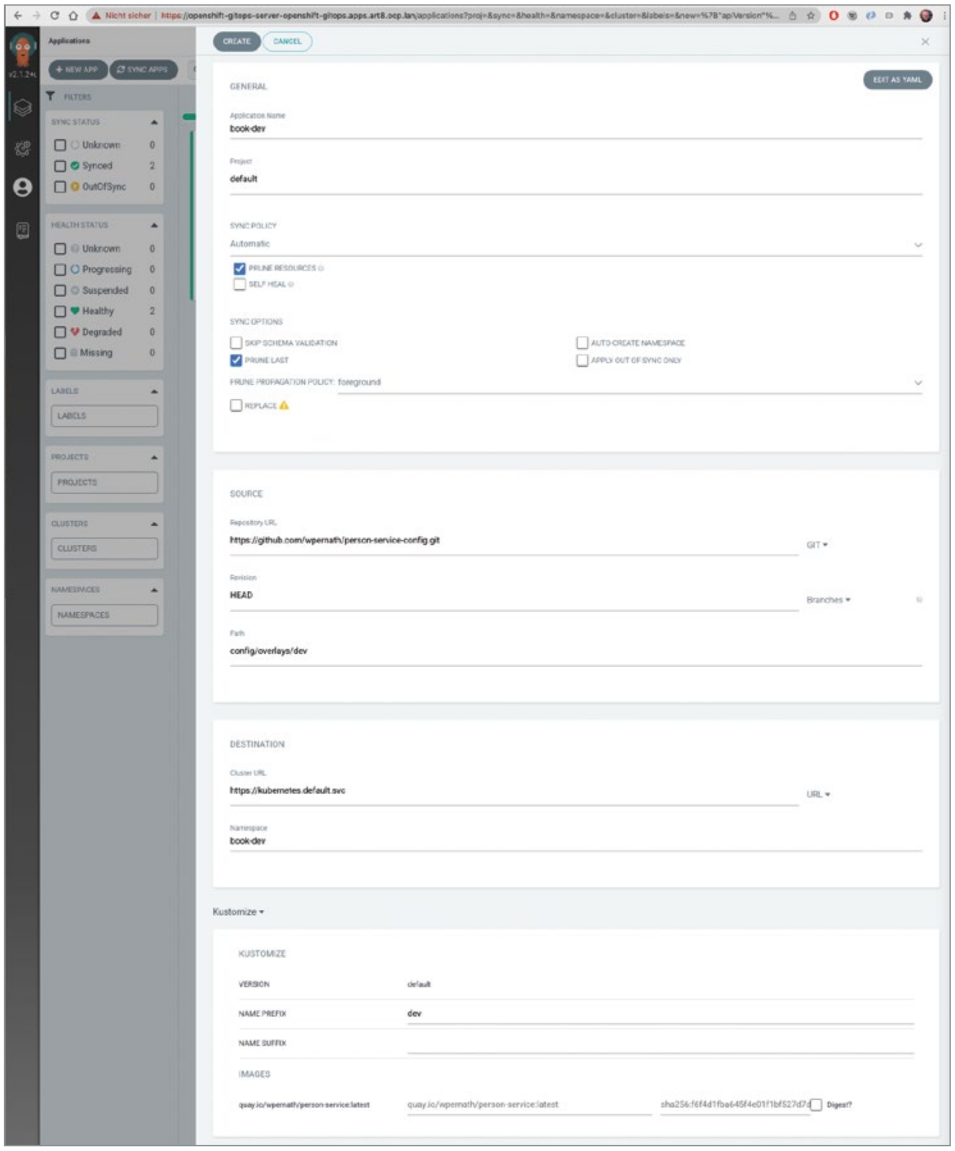


Figure 5.3: Argo CD on OpenShift.

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 Figure 5-4 as follows:

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

Figure 5-4: Creating a new application in Argo CD.

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
```

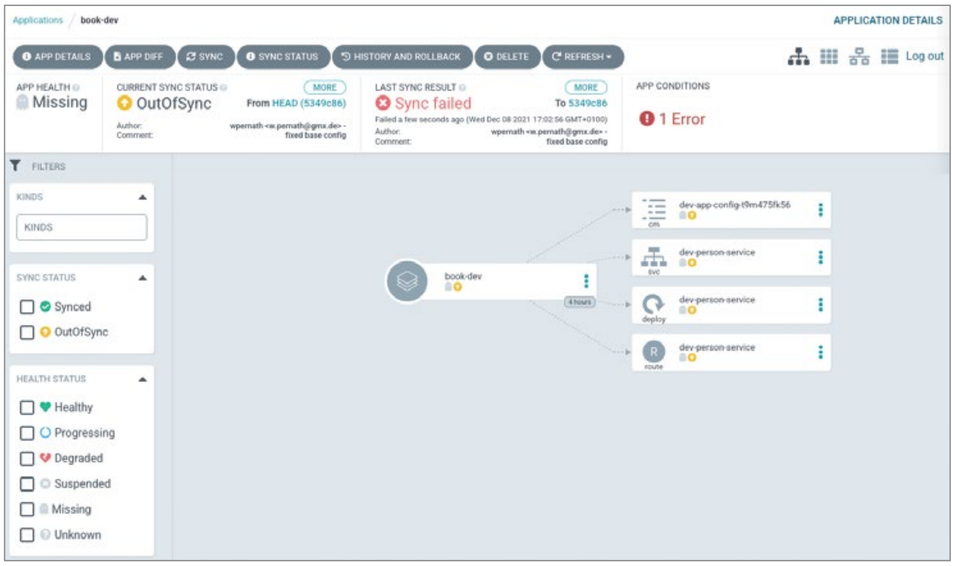
Remove the metadata from the object file so that it looks like the listing just shown, and then enter the following command to import the application into the predefined Argo CD instance:

```
$ 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 (Figure 5-5).



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

book-dev
dev-app-config-t9m475fk56
dev-person-service
dev-person-service
dev-person-service

Figure 5-5: Argo CD UI showing synchronization failure.

The error arises because the Argo CD service account 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 <target namespace>
```

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 a superuser of your Kubernetes cluster. This might not be very secure.

After giving the service account the necessary role, you can safely click Sync, and Argo CD will do the synchronization (Figure 5-6).

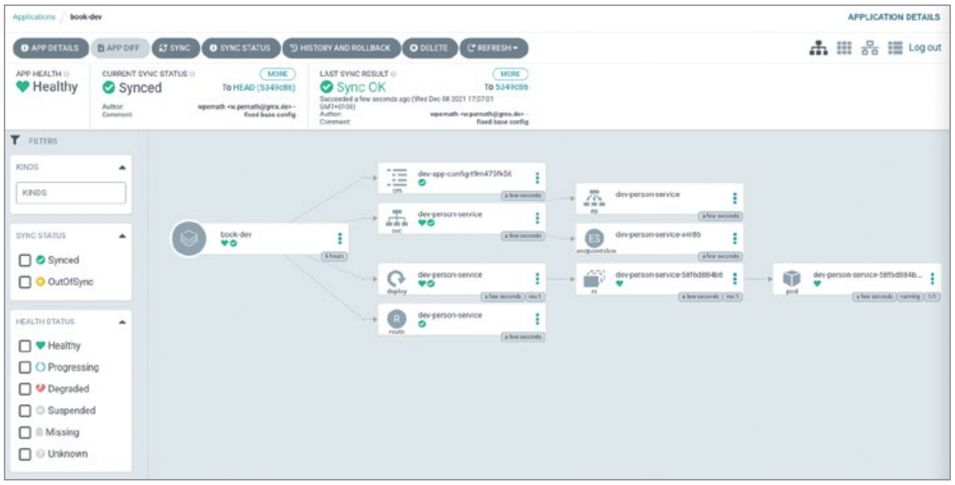


Figure 5-6: Argo CD UI showing successful synchronization.

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

To automatically create everything you need to let Argo CD start synchronizing your config repository with a Kubernetes cluster, you must create the files described in the following sections. See `book-example/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. You can do this 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 can apply all files in one go. To accomplish this simplification, create a **`kustomization.yaml`** file, which looks like:

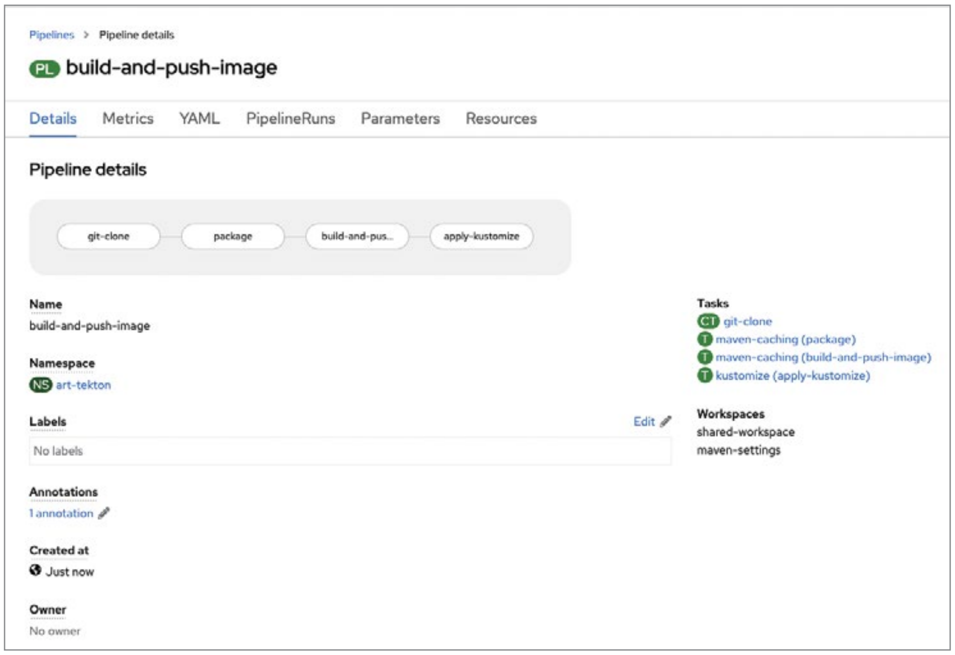
```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

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

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

```
$ oc apply -k book-example/gitops/argocd
```

Create a Tektonik Pipeline to Update person-service-config

We now want to change our pipeline from the previous chapter (Figure 5-7) to be more GitOps-y. But what exactly needs to be done?



PL build-and-push-image

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

Figure 5-7: Tekton pipeline from Chapter 4.

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 [5.10]).
- 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:

- Compile and test the code.
- Create a new image.

- Push that image to an external registry (in our case, quay.io [5.10]).
- Use Kustomize to change the image target.
- 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 create and use the `git-update-deployment` task as the final step. This task should clone the config repository, use Kustomize to apply the image changes, and finally push the changes back to GitHub [5.11].

A Word On Tekton Security

Because we want to update a private repository, we first need to look at Tekton authentication [5.12]. 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 entirely within the security context of the provided ServiceAccount.

Let's use it 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 meta-data 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 can 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 "something@some-other-thing"
       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 [5.13] (which in turn is using Jib [5.14]), 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 created a separate task that does just this step. The step relies on a Tekton feature known as emitting results from a task [5.15].

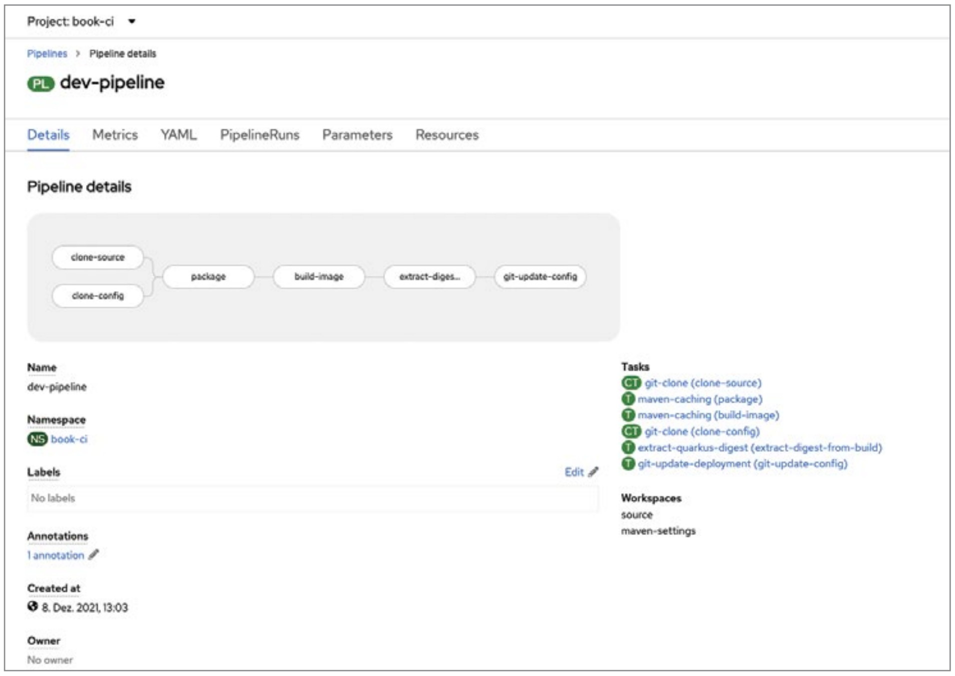
You can define a **results** property within the **spec** section of a task. Each result is stored in **$(results.<result-name>.path)**, where the **<result-name>** component is a string that refers to the data in that result. Results are available in all tasks and on the pipeline level through strings 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. Figure 5-8 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.



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)

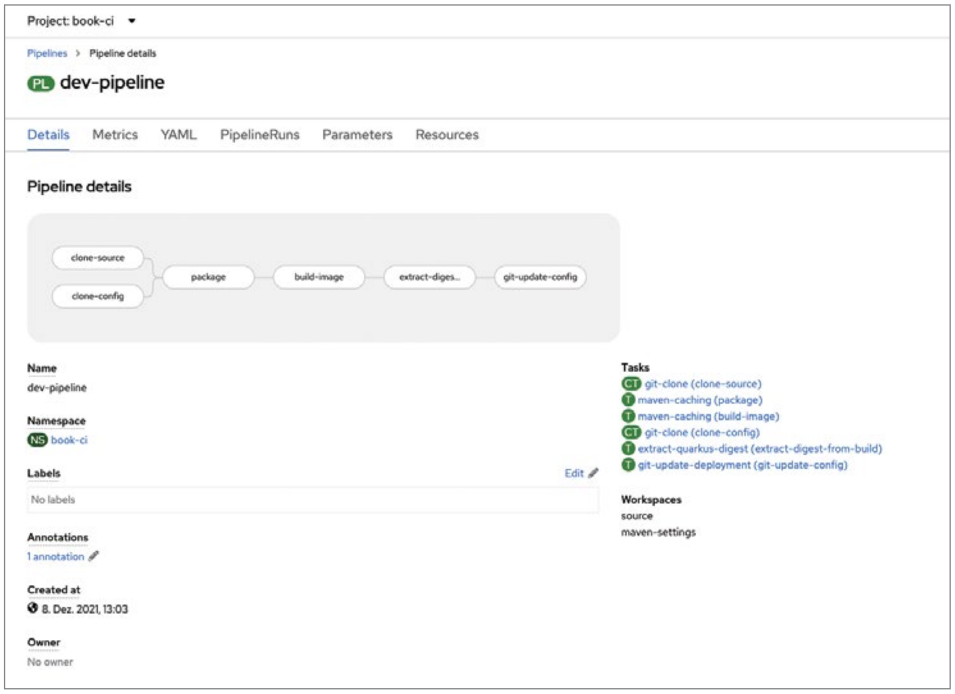
Workspaces
source
maven-settings

Figure 5-8: The gitops-pipeline.

Start by using the previous non-GitOps pipeline [5.16], 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.

Now it's time to summarize everything in a new pipeline. Figure 5-8 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.



Project: book-ci

Pipelines > Pipeline details

PL dev-pipeline

Details Metrics YAML PipelineRuns Parameters Resources

Pipeline details

Name
dev-pipeline

Namespace
NS book-ci

Labels
Edit
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

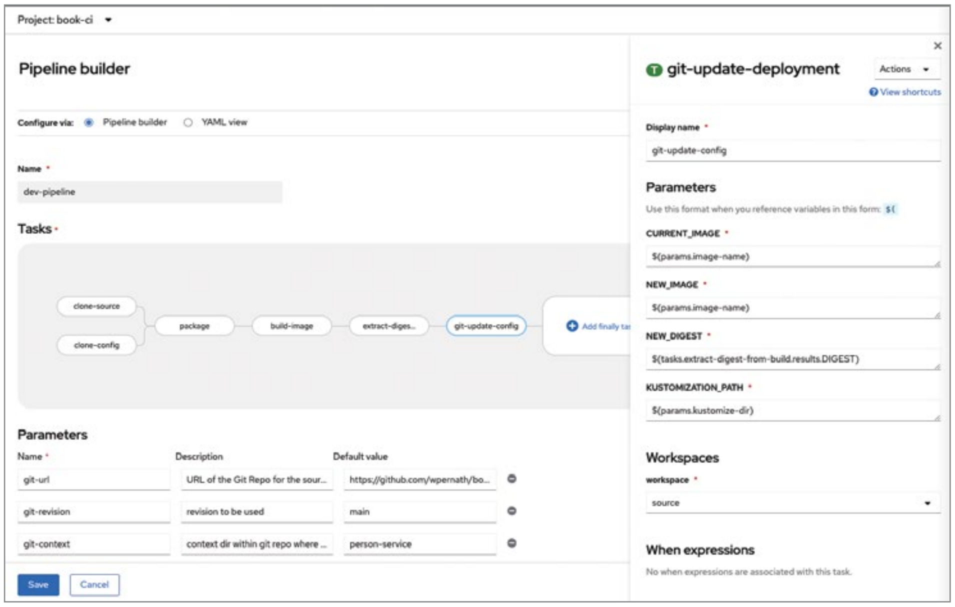
Figure 5-8: The gitops-pipeline.

Start by using the previous non-GitOps pipeline [5.17], 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 Figure 5-9.



Configure via: [✓ Pipeline builder] YAML view

Name *: dev-pipeline

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.

Figure 5-9: Parameter mapping.

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/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:

```
$ ./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>
```

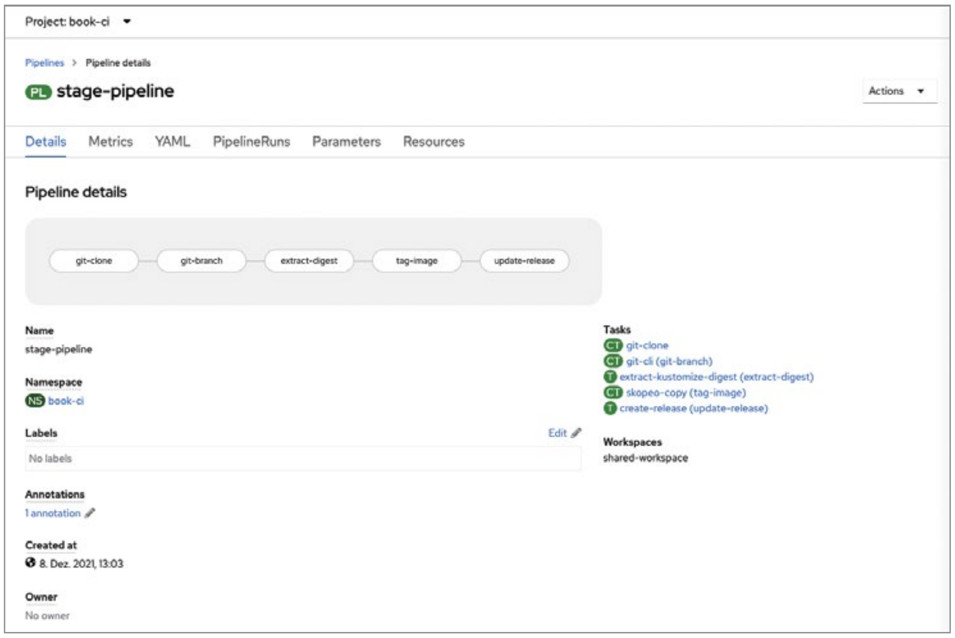
Whenever the pipeline is successfully executed, you should notice an updated message in the **`person-service-config`** Git repository. You should also see that Argo CD has initiated a synchronization process, which ends with a redeployment of the quarkus application.

Refer back to Chapter 4 for more information on starting and testing pipelines.

Create a stage-release Pipeline

What does a staging pipeline look like? We need a process that does the following, in our case (Figure 5-10):

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

Figure 5-10: The staging pipeline.

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 1.2.5:

```
$ ./pipeline.sh stage -r 1.0.0-beta1
```

Pipeline Setup

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 [5.18] 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 from the **`kustomization.yaml`** file. The command looks like:

```
$ yq eval '.images[0].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 can authenticate to Quay.io and do its work.

Creating the Release

The final **`create-release`** task does more or less 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 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 Git was intended to be used by people who 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 application 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 [5.18].

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; it 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. This is probably 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 [5.19] 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:

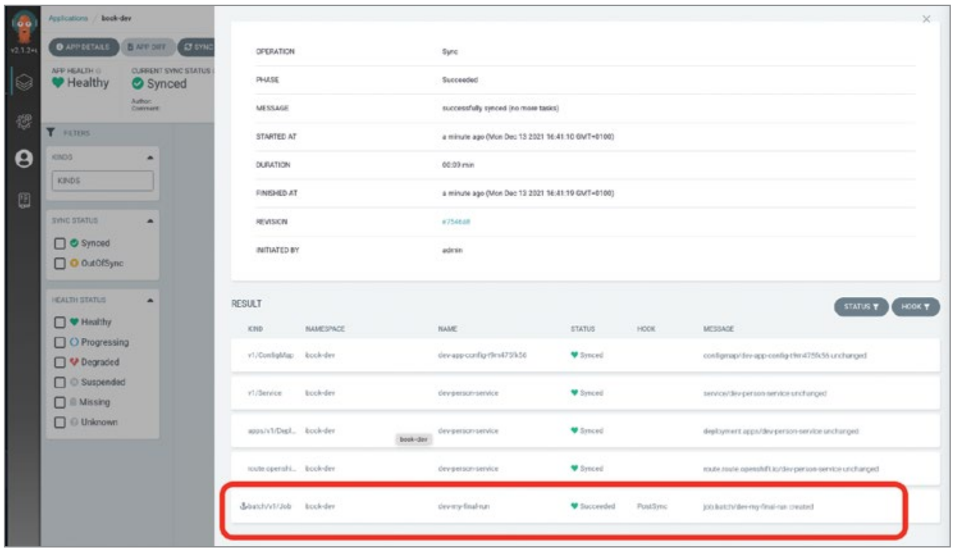
```
apiVersion: batch/v1
kind: Job
metadata:
  generateName: post-sync-run
  name: my-final-run
  annotations:
    argocd.argoproj.io/hook: PostSync
spec:
  ttlSecondsAfterFinished: 100
  template:
    spec:
      restartPolicy: Never
      containers:
        - name: post-install-job
          env:
            - 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 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: Make sure to add your sync jobs to the base `kustomization.yaml` file. Otherwise, they won't be processed.

Figure 5-11 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	e754648
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

Figure 5-11: The synchronization log of Argo CD with a presync and a postsync hook.

Git Repository Management

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

If you want to use a single configuration Git repository for all your applications and stages, keep in mind that Git was never meant to resolve merge conflicts automatically. 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 create and merge release branches automatically.

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 GitHub and Quay.io. You don't want to make that confidential information publicly accessible in a Git repository.

Instead, think about a solution like Hashicorp Vault [5.20], 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 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, making 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:

- 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 consistently 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 standalone applications. Automation becomes a necessity in large environments and fast-changing configurations of hosts.

I hope this book has helped you put together the basic concepts and enable your work with DevOps and GitOps.

References

[5.1] https://argoproj.github.io/argo-cd/

[5.2] https://github.com/wpernath/book-example/tree/main/person-service

[5.3] https://docs.openshift.com/container-platform/4.8/cicd/gitops/installing-openshift-gitops.html

[5.4] https://github.com/code-ready/crc

[5.5] https://www.opensourcerers.org/2021/07/26/automated-application-packaging-and-distribution-with-openshift-tekton-pipelines-part-34-2/

[5.6] https://github.com/wpernath/person-service-config

[5.7] https://deviq.com/principles/separation-of-concerns

[5.8] https://argo-cd.readthedocs.io/en/latest/getting_started/

[5.9] https://docs.openshift.com/container-platform/4.8/cicd/gitops/installing-openshift-gitops.html

[5.10] https://quay.io

[5.11] http://github.com

[5.12] https://tekton.dev/docs/pipelines/auth/

[5.13] https://www.opensourcerers.org/2021/07/26/automated-application-packaging-and-distribution-with-openshift-tekton-pipelines-part-34-2/

[5.14] https://cloud.google.com/blog/products/application-development/introducing-jib-build-java-docker-images-better

[5.15] https://tekton.dev/docs/pipelines/tasks/#emitting-results

[5.16] https://raw.githubusercontent.com/wpernath/book-example/main/tekton/pipelines/tekton-pipeline.yaml

[5.17] https://tekton.dev/docs/pipelines/pipelines/#guard-task-execution-using-whenexpressions

[5.18] https://argo-cd.readthedocs.io/en/stable/user-guide/sync-waves/

[5.19] https://argo-cd.readthedocs.io/en/stable/user-guide/resource_hooks/

[5.20] https://www.hashicorp.com/products/vault/secrets-management

About the Author

Wanja Pernath is a Technical Enablement Manager at Red Hat. He started his career in 1997 as a C and C++ developer and later became a Java and Java Enterprise Edition consultant in 2000. Wanja joined Red Hat Germany in 2007 as a JBoss Solution Architect, working with middleware customers in central and eastern Europe. In 2016 he joined the EMEA Partner Enablement team, where he oversees the OpenShift platform from a development perspective. His work includes CI/CD, GitOps, pipelines and automation, and Kubernetes-native development with Quarkus.

Wanja lives in a very (very!) small town near Munich, Germany.