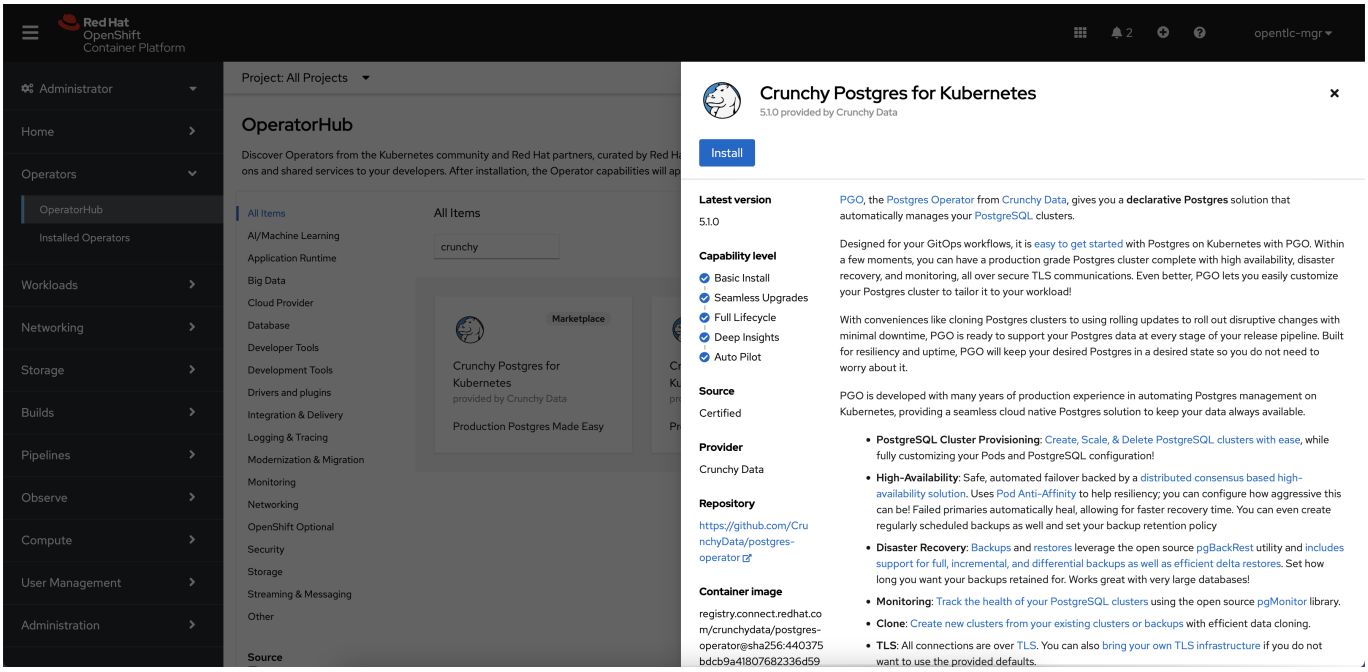
Chapter 1: Deployment Basics

This discusses how apps 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.

Installing the Crunchy Postgres for Kubernetes Operator

The Crunchy Postgres for Kubernetes Operator can easily be installed in OpenShift. Just log in as a user with cluster-admin rights and switch to the **Administrator** perspective of the OpenShift console. Then go to the **Operators** menu entry and select **OperatorHub**. In the search field, start typing "crunchy " and select the Crunchy Postgres for Kubernetes Operator when its panel is shown. Make sure it's the **Certified** one and **NOT** Marketplace. Accept all the defaults and click install.



Once the Operator is installed, the Postgres service should be available to the entire cluster.

Introduction and Motivation

As someone with a long history of developing software, I like containers and Kubernetes a lot, because those technologies increase my own 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 micro-services—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
- etc.

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 app named `person-service`. The app is based on the Java builder image `openjdk-11-ubi8` and takes its source code coming from GitHub. The final command effectively publishes the service so that apps 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
$ 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.

The security settings, deployment, image, route, and service are generated for you by OpenShift (and also some OpenShift specific files, such as `ImageStream` or `DeploymentConfig`). These OpenShift conveniences allow you to fully focus on app 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 app 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 that Kubernetes needs to load balance your application over different instances (pods).

- `Route`: A route defines the external URL exposed by your app. Requests from clients are received at this URL.
- `ConfigMap`: ConfigMaps contain—well—configurations for the app.
- `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 app 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 (show 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're now able to create your app 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
$ git clone https://github.com/waynedovey/GitOps-Workshop.git
$ cd GitOps-Workshop
$ oc apply -f artifacts/raw-kubernetes/service.yaml
$ oc apply -f artifacts/raw-kubernetes/deployment.yaml
$ oc apply -f artifacts/raw-kubernetes/route.yaml
```

We have to create a PostgreSQL database server as well. Just execute the following command.

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

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 are entering commands as an admin user.

This section has described one way of getting required files. Of course, if you have more elements to your application, you 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 the manifest files of your app 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 obviously do this manually. But let's find some ways to automate it.

The following process efficiently creates a staging release:

- Tag the currently used image in `book-dev` to something more meaningful, like `person-service:v1.0.0-test`.
- 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/person-service@sha... book-dev/person-service:stage-v1.0.0
oc new-project ...
oc apply -f deployment.yaml
```

More details on this topic can be found in my article Release Management with OpenShift: Under the hood.

Using a tool such as `yq` seems to be the easiest way to automate the processing of Kubernetes manifest files. However, this process makes you 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 apps, the demands could easily 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`), although 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
```

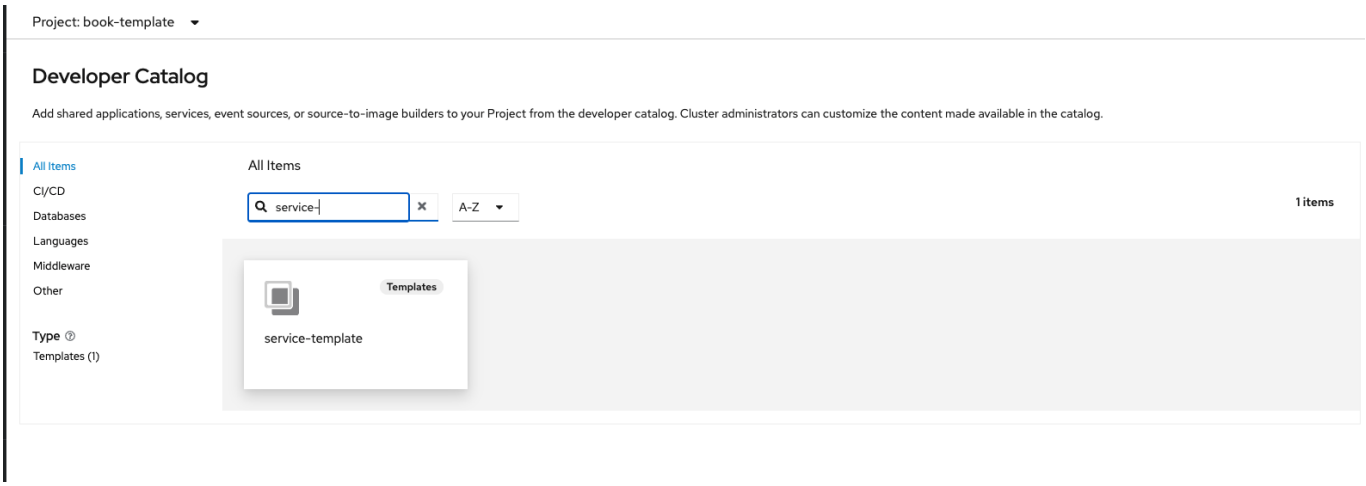
Now for the biggest convenience offered by OpenShift Templates: Once you have instantiated a template in an OpenShift namespace, you can used the template to create applications within the graphical user interface (UI):

```
$ 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 artifacts/ocp-template/service-template.yaml
template.template.openshift.io/service-template created
```

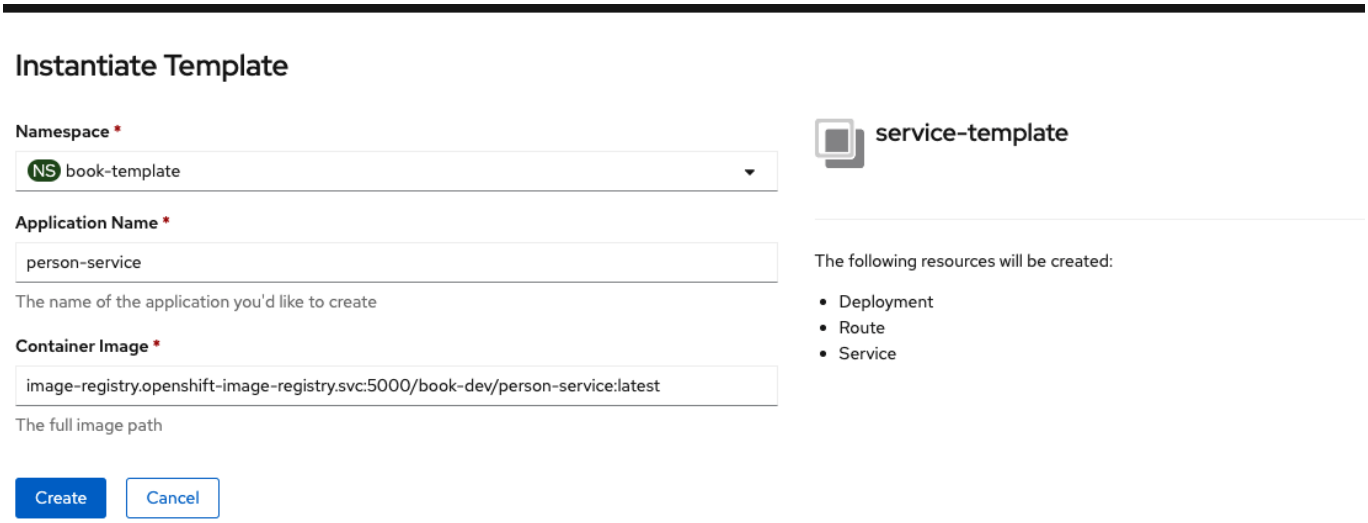
We have to create a PostgreSQL database server as well. Just execute the following command.

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

Just open the OpenShift web console now, choose the project, click **+Add**, and choose the **Developer Catalog**. You should be able to find a template called `service-template` (Image 1). This is the one we've created.



Instantiate the template and fill in the required fields (Image 2).



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.

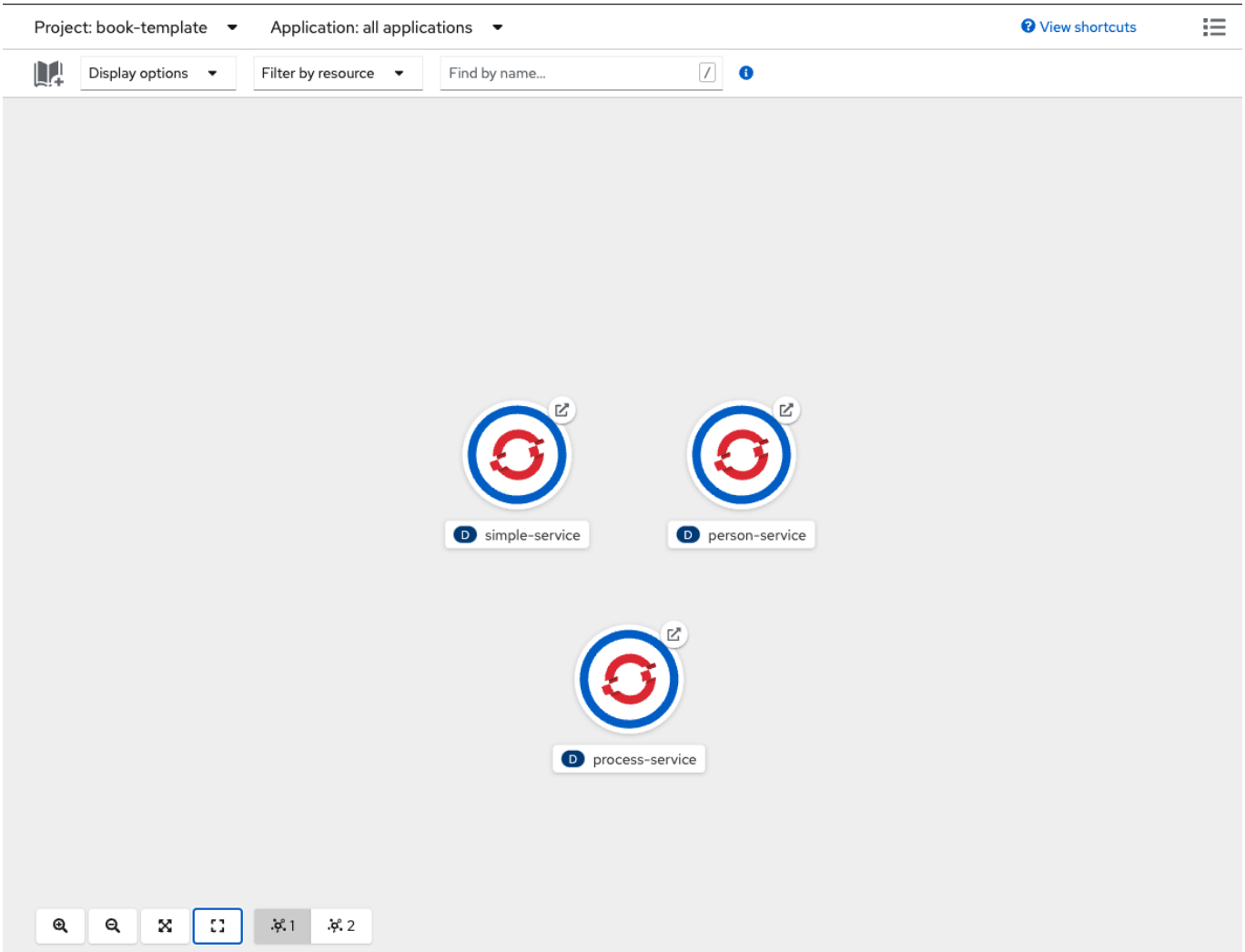
There are also several ways to create an application instance out of a template without the UI. You can run an `oc`command to do the work within OpenShift:

```
$ 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 (Image 3).



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. that only a few fields 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. The patch mechanism of Kustomize 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 artifacts/kustomize
artifacts/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 in order to deploy your application. These files should be well-known 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`) but also 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 policy add-role-to-user system:image-puller
system:serviceaccount:book-kustomize:default \
  --namespace=book-dev
$ oc apply -k artifacts/kustomize/overlays/dev
service/dev-person-service created
deployment.apps/dev-person-service created
route.route.openshift.io/dev-person-service created
```

We have to create a PostgreSQL database server as well. Just execute the following command.

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

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

```
$ kustomize build artifacts/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 big 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 together for the staging overlay.

```
$ kustomize build artifacts/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 artifacts/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 command:

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

More sophisticated Kustomize examples

Instead of using `patchStrategicMerge` files, you could just maintain a `kustomization.yaml` file containing everything. An example looks like:

```
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 (v4.x and above) of Kustomize that help you even better maintain your overlays. 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 before) `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 artifacts/kustomize-ext/overlays/stage | oc apply -f -
```

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

Summary of Kustomize

Using Kustomize is quite easy and 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 also a Kubernetes subproject and is tightly integrated into Kubernetes's tools, you don't have to worry that Kustomize would suddenly disappear.

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

Summary

In this chapter we have learned 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.

The next chapter is about Helm Charts and Kubernetes Operators for application packaging and distribution.