that depend on this base image and update your running applications with the new image. When implementing this *Image Builder* pattern, the cluster knows both—the build of an image and its deployment—and can automatically do a redeployment if a base image changes. In "OpenShift Build" on page 346, we'll see how OpenShift implements such automation.

Having seen the benefits of building images on the platform, let's look at what techniques exist for creating images in a Kubernetes cluster.

Solution

As of 2023, a whole zoo of in-cluster container image-build techniques exists. While all target the same goal of building images, each tool adds a twist, making it unique and suitable for specific situations.

Figure 30-1 contains the essential image-building techniques as of 2023 for building container images within a Kubernetes cluster.

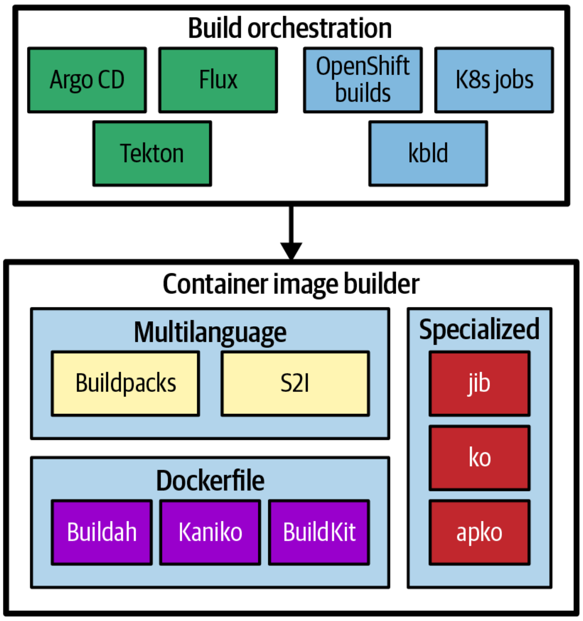

Figure 30-1. Container image builds within Kubernetes

This chapter contains a brief overview of most of these techniques. You can find more details about these tools by following the links in "More Information" on page 353. Please note that while many of the tools described here are matured and used in production projects, there are no guarantees that some of those projects still exist when you read these lines. Before using one, you should check whether the project is still alive and supported.

Categorizing these tools is not straightforward as they are partly overlapping or dependent on one another. Each of these tools has a unique focus, but for in-cluster builds, we can identify these high-level categories:

Container image builder
: These tools create container images within the cluster. There is some overlap of these tools, and they vary, but all of them can run without privileged access. You can also run these tools outside the cluster as CLI programs. The sole purpose of these builders is to create a container image, but they don't care about application redeployments.

Build orchestration
: These tools operate on a higher level of abstraction and eventually trigger the container image builder for creating images. They also support build-related tasks like updating the deployment descriptors after the image has been built. CI/CD systems, as described previously, are typical examples of orchestrators.

Container Image Builder

One of the essential prerequisites for building images from within a cluster is creating images without having privileged access to the node host. Various tools exists that fulfill this prerequisite, and they can be roughly categorized according to how the container image is specified and built.

Rootless Builds

When building within Kubernetes, the cluster has complete control over the build process. Because of this, the cluster needs higher security standards to protect against potential vulnerabilities. One way to improve security during builds is to run them without root privileges, a practice known as *rootless builds*. There are many ways to achieve rootless builds in Kubernetes that allow you to build without elevated privileges.

Docker successfully brought container technologies to the masses thanks to its unmatched user experience. Docker is based on a client-server architecture with a daemon running in the background and taking instructions via a REST API from its client. This daemon needs root privileges mainly for network and volume management reasons. Unfortunately, this imposes a security risk, as untrusted processes can escape their container, and an intruder could get control of the whole host. This concern applies not only when running containers but also when building container images because building also happens within a container when the Docker daemon executes arbitrary commands.

Most of the in-cluster build techniques described in this chapter allow container images to be built in a nonprivileged mode to reduce that attack surface, which is very useful for locked-down Kubernetes clusters.

Dockerfile-Based builders

The following builders are based on the well-known Dockerfile format for defining the build instructions. All of them are compatible on a Dockerfile level, and they either work completely without talking to a background daemon or talk via a REST API remotely with a build process that is running in a nonprivileged mode:

Buildah and Podman
: Buildah and its sister Podman are potent tools for building OCI-compliant images without a Docker daemon. They create images locally within the container before pushing them to an image registry. Buildah and Podman overlap in functionality, with Buildah focusing on building container images (though Podman can also create container images by wrapping the Buildah API). The difference is shaped more clearly in this README.

Kaniko
: Kaniko is one backbone of the Google Cloud Build service and is deliberately targeted for running as a build container in Kubernetes. Within the build container, Kaniko still runs with UID 0, but the Pod holding the container itself is nonprivileged. This requirement prevents the usage of Kaniko in clusters that disallow running as a root user in a container, like in OpenShift. We see Kaniko in action in "Build Pod" on page 342.

BuildKit
: Docker extracted its build engine into a separate project, BuildKit, which can be used independently of Docker. It inherits from Docker its client-server architecture with a BuildKit daemon running in the background, waiting for build jobs. Usually, this daemon runs directly in the container that triggers the build, but it can also run in a Kubernetes cluster to allow distributed rootless builds. BuildKit introduces a Low-Level Build (LLB) definition format supported by multiple frontends. LLB allows complex build graphs and can be used for arbitrary complex build definitions. BuildKit also supports features that go beyond the original Dockerfile specification. In addition to Dockerfiles, BuildKit can use other frontends to define the container image's content via LLB.

Multilanguage builders

Many developers care only that their application gets packaged as container images and not so much about how this is done. To cover this use case, multilanguage builders exist to support many programming platforms. They detect an existing project, like a Spring Boot application or generic Python build, and select an opinionated image build flow accordingly.

Buildpacks have been around since 2012 and were initially introduced by Heroku to allow you to push developer's code directly to their platform. Cloud Foundry picked up that idea and created a fork of buildpacks that eventually led to the infamous `cf push` idiom that many considered the gold standard of Platform as a Service (PaaS). In 2018, the different forks of Buildpacks united under the umbrella of the CNCF and are now known as *Cloud Native Buildpacks* (CNB). Besides individual buildpacks for different programming languages, CNB introduce a lifecycle for transforming source code to executable container images.

The lifecycle can roughly be divided into three main phases:[1]

- In the *detect* phase, CNB iterate over a list of configured buildpacks. Each buildpack can decide whether it fits for the given source code. For example, a Java-based buildpack will raise its hand when it detects a Maven *pom.xml*.
- All buildpacks that survived the detect phase will be called in the *build* phase to provide their part for the final, possibly compiled artifact. For example, a buildpack for a Node.js application calls `npm install` to fetch all required dependencies.
- The last step in the CNB lifecycle is an *export* to the final OCI image that gets pushed to a registry.

CNB target two personas. The primary audience includes *Developers* who want to deploy their code onto Kubernetes or any other container-based platform. The other is *Buildpack Authors*, who create individual buildpacks and group them into so-called *builders*. You can choose from a list of prefactored buildpacks and builders or create your own for you and your team. Developers can then pick up those buildpacks by referencing them when running the CNB lifecycle on their source code. Several tools are available for executing this lifecycle; you'll find a complete list at the Cloud Native Buildpacks site.

For using CNB within a Kubernetes cluster, the following tasks are helpful:

- `pack` is a CLI command to configure and execute the CNB lifecycle locally. It requires access to an OCI container runtime engine like Docker or Podman to run Builder images that hold the list of buildpacks to use.
- CI steps like Tekton build tasks or GitHub actions that call the lifecycle directly from a configured Builder image.

1 CNB cover more phases. The entire lifecycle is explained on the Buildpacks site.

- `kpack` comes with an Operator that allows you to configure and run buildpacks within a Kubernetes cluster. All the core concepts of CNB, like Builder or Buildpacks, are reflected directly as CustomResourceDefinitions. `kpack` is not yet part of the CNB project itself, but as of 2023 is about to be absorbed.

Many other platforms and projects have adopted CNB as their build platform of choice. For example, Knative Functions use CNB under the hood to transform Function code to container images before they get deployed as Knative services.

OpenShift's Source-to-Image (S2I) is another opinionated building method with builder images. S2I takes you directly from your application's source code to executable container images. We will look closely at S2I in "OpenShift Build" on page 346.

Specialized builders

Finally, specialized builders with an opinionated way of creating images exist for specific situations. While their scope is narrow, their strong opinion allows for a highly optimized build flow that increases flexibility and decreases build times. All these builders perform a rootless build. They create the container image without running arbitrary commands as with a Dockerfile `RUN` directive. They create the image layers locally with the application artifacts and push them directly to a container image registry:

Jib
: Jib is a pure Java library and build extension that integrates nicely with Java build tools like Maven or Gradle. It creates separate image layers directly for the Java build artifacts, its dependencies, and other static resources to optimize image rebuild times. Like the other builders, it speaks directly with a container image registry for the resulting images.

ko
: For creating images from Golang sources, ko is a great tool. It can directly create images from remote Git repositories and update Pod specifications to point to the image after it has been built and pushed to a registry.

Apko
: Apko is a unique builder that uses Alpine's Apk packages as building blocks instead of Dockerfile scripts. This strategy allows for the easy reuse of building blocks when creating multiple similar images.

This list is only a selection of the many specialized build techniques. All of them have a very narrow scope of what they can build. The advantage of this opinionated approach is that they can optimize build time and image size because they know precisely about the domain in which they operate and can make strong assumptions.

Now that we have seen some ways to build container images, let's jump one abstraction level higher and see how we can embed the actual build in a broader context.

Build Orchestrators

Build orchestrators are CI and CD platforms like Tekton, Argo CD, or Flux. Those platforms cover your application's entire automated management lifecycle, including building, testing, releasing, deploying, security scanning, and much more. There are excellent books that cover those platforms and bring it all together, so we won't go into the details here.

In addition to general-purpose CI and CD platforms, we can use more specialized orchestrators to create container images:

OpenShift builds
: One of the oldest and most mature ways of building images in a Kubernetes cluster is the *OpenShift build* subsystem. It allows you to build images in several ways. We take a closer look at the OpenShift way of building images in "OpenShift Build" on page 346.

kbld
: kbld is part of Carvel, a toolset for building, configuring, and deploying on Kubernetes. kbld is responsible for building containers with one of the builder technologies we described in "Container Image Builder" on page 337 and updating resource descriptors with a reference to the images that have been built. The technique for updating the YAML files is very similar to how ko works: kbld looks for `image` fields and sets their values to the coordinates of the freshly built image.

Kubernetes Job
: You can also use standard Kubernetes Jobs for triggering builds with any of the image builders from "Container Image Builder" on page 337. Jobs are described in detail in Chapter 7, "Batch Job". Such a Job wraps a build Pod specification for defining the runtime parts. The build Pod picks up the source code from a remote source repository and uses one of the in-cluster builders to create the appropriate image. We'll see such a Pod in action in "Build Pod" on page 342.

What Happened to Knative Build?

In the first edition of this book, we described Knative Build as one possibility for creating container images from within the cluster. As time has shown, Knative as an umbrella project was too small for the community, so Knative Build was split off from Knative and transformed into a new project, Tekton, with a much larger scope than only building container images. Tekton a is a full-featured CI solution that fully integrates into Kubernetes and uses CustomResourceDefinitions as described in Chapter 28 as the basis for the description of the CI pipelines.

While Knative Build is history now, it was an excellent lesson about how open source communities evolve and can transform in unexpected ways. Keep this in mind, as it might happen to other popular projects too.

Build Pod

To carve out the essential ingredients of typical in-cluster builds, let's start minimally and use a Kubernetes Pod for performing a complete build and deploy cycle. These build steps are illustrated in Figure 30-2.

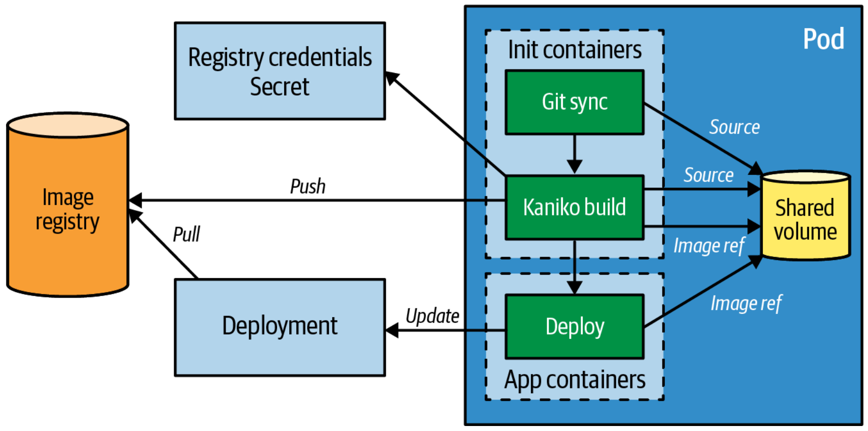

Figure 30-2. In-cluster container image build with a build Pod

The following tasks are representative of all build orchestrators and cover all aspects of creating container images:

- Check out the source code from a given remote Git repository.
- For a compiled language, perform a local build within the container.
- Build the application with one of the techniques described in "Container Image Builder" on page 337.
- Push the image to a remote image registry.
- Optionally, update a deployment with the new image reference, which will trigger a redeployment of the application following the strategies described in Chapter 3, "Declarative Deployment".

The build Pod in our example uses init containers as described in Chapter 15, "Init Container", to ensure that the build steps are running one after the other. In a real-world scenario, you would use a CI system like Tekton to specify and execute these tasks sequentially.

The complete build Pod definition is shown in Example 30-1.

Example 30-1. Build Pod using Kaniko

```
apiVersion: v1
kind: Pod
metadata:
  name: build
spec:
  initContainers:
  - name: git-sync            ❶
    image: k8s.gcr.io/git-sync/git-sync
    args: [
      "--one-time",
      "--depth", "1",
      "--root", "/workspace",
      "--repo", "https://github.com/k8spatterns/random-generator.git",
      "--dest", "main",
      "--branch", "main"]
    volumeMounts:             ❷
    - name: source
      mountPath: /workspace
  - name: build               ❸
    image: gcr.io/kaniko-project/executor
    args:
    - "--context=dir:///workspace/main/"
    - "--destination=index.docker.io/k8spatterns/random-generator-kaniko"
    - "--image-name-with-digest-file=/workspace/image-name"
    securityContext:
      privileged: false       ❹
    volumeMounts:
    - name: kaniko-secret     ❺
      mountPath: /kaniko/.docker
    - name: source            ❻
      mountPath: /workspace
  containers:
  - name: image-update        ❼
    image: k8spatterns/image-updater
    args:
    - "random"
    - "/opt/image-name"
    volumeMounts:
    - name: source
      mountPath: /opt
  volumes:
  - name: kaniko-secret       ❽
    secret:
      secretName: registry-creds
      items:
      - key: .dockerconfigjson
        path: config.json
  - name: source           ❾
    emptyDir: {}
  serviceAccountName: build-pod  ❿
  restartPolicy: Never     ⓫
```

❶ Init container for fetching the source code from a remote Git repository.

❷ Volume in which to store the source code.

❸ Kaniko as build container, storing the created image as a reference in the shared workspace.

❹ Build is running unprivileged.

❺ Secret for pushing to Docker Hub registry mounted at a well-known path so that Kaniko can find it.

❻ Mounting shared workspace for getting the source code.

❼ Container for updating the deployment `random` with the image reference from the Kaniko build.

❽ Secret volume with the Docker Hub credentials.

❾ Definition of a shared volume as an empty directory on the node's local filesystem.

❿ ServiceAccount that is allowed to patch a Deployment resource.

⓫ Never restart this Pod.

This example is quite involved, so let's break it down into three main parts.

First, before being able to build a container image, the application code needs to be fetched. In most cases, the source code is picked up from a remote Git repository, but other techniques are available. For development purposes, it is convenient to get the source code from your local machine so that you don't have to go over a remote source repository and mess up your commit history with triggering commits. Because the build happens within a cluster, that source code must be uploaded somehow to your build container. Another possibility is to distribute the source code packaged in a container image and distribute it via a container image registry.

In Example 30-1, we use an init container to fetch the source code from our source Git repository and store it in a shared Pod volume `source` of type `emptyDir` so that it can later be picked up by the build process container.

Second, after the application code is retrieved, the actual build happens. In our example, we use Kaniko, which uses a regular Dockerfile and can run entirely unprivileged. We again use an init container to ensure that the build starts only after the source code has been fully fetched. The container image is created locally on disk, and we also configure Kaniko to push the resulting image to a remote Docker registry.

The credentials for pushing to the registry are picked up from a Kubernetes Secret. We describe Secrets in detail in Chapter 20, "Configuration Resource".

Luckily, for the particular case of authentication against a Docker registry, we have direct support from `kubectl` for creating such a secret that stores this configuration in a well-known format:

```
kubectl create secret docker-registry registry-creds \
    --docker-username=k8spatterns \
    --docker-password=********* \
    --docker-server=https://index.docker.io/
```

For Example 30-1, such a secret is mounted into the build container under a given path so that Kaniko can pick it up when pushing the created image. In Chapter 25, "Secure Configuration", we explain how such a secret can be stored securely so that it can't be forged.

The final step is to update an existing Deployment with the newly created image. This task is now performed in the actual application container of the Pod.[2] The referenced image is from our example repository and contains just a `kubectl` binary that patches the specified Deployment with the new image name with the following call, shown in Example 30-2.

Example 30-2. Update image field in Deployment

```
IMAGE=$(cat $1)                  ❶
PATCH=<<EOT                      ❷
[{
  "op":    "replace",
  "path":  "/spec/template/spec/containers/0/image",
  "value": "$IMAGE"
}]
EOT
```

2 We could have also chosen an init container again here and used a no-op application container, but since the application containers start only after all init containers have been finished, it doesn't matter much where we put the container. In all cases, the three specified containers run after one another.

```
kubectl patch deployment $2 \ ❸
   --type="json" \
   --patch=$PATCH
```

❶ Pickup image name stored by the previous build step in the file */opt/image-name*. This file is provided as the first argument to this script.

❷ JSON path to update the Pod spec with the new image reference.

❸ Patch the deployment given as the second argument (`random` in our example) and trigger a new rollout.

The Pod's assigned ServiceAccount `build-pod` is set up so it can write to this Deployment. Assigning permissions to a ServiceAccount is described fully in Chapter 26, "Access Control". When the image reference is updated in the Deployment, a rollout as described in Chapter 3, "Declarative Deployment", is performed.

You can find the fully working setup in the book's example repository. The build Pod is the simplest way to orchestrate an in-cluster build and redeployment. As mentioned, it is meant for illustrative purposes only.

For real-world use cases, you should use a CI/CD solution like Tekton or a whole build orchestration platform like OpenShift Build, which we describe now.

OpenShift Build

Red Hat OpenShift is an enterprise distribution of Kubernetes. Besides supporting everything Kubernetes supports, it adds a few enterprise-related features like an integrated container image registry, single sign-on support, and a new user interface, and it also adds a native image building capability to Kubernetes. OKD is the upstream open source community edition distribution that contains all the OpenShift features.

OpenShift build was the first cluster-integrated way of directly building images managed by Kubernetes. It supports multiple strategies for building images:

Source-to-Image (S2I)
: Takes the source code of an application and creates the runnable artifact with the help of a language-specific S2I builder image and then pushes the images to the integrated registry.

Docker builds
: Use a Dockerfile plus a context directory and creates an image as a Docker daemon would do.

Pipeline builds
: Map build-to-build jobs of an internally managed Tekton by allowing the user to configure a Tekton pipeline.

Custom builds
: Give you full control over how you create your image. Within a custom build, you have to create the image on your own within the build container and push it to a registry.

The input for doing the builds can come from different sources:

Git
: Repository specified via a remote URL from where the source is fetched.

Dockerfile
: A Dockerfile that is directly stored as part of the build configuration resource.

Image
: Another container image from which files are extracted for the current build. This source type allows for *chained builds*, as shown in Example 30-4.

Secret
: Resource for providing confidential information for the build.

Binary
: Source to provide all input from the outside. This input has to be provided when starting the build.

The choice of which input sources we can use in which way depends on the build strategy. *Binary* and *Git* are mutually exclusive source types. All other sources can be combined or used on a standalone basis. We will see later in Example 30-3 how this works.

All the build information is defined in a central resource object called BuildConfig. We can create this resource either by directly applying it to the cluster or by using the CLI tool `oc`, which is the OpenShift equivalent of `kubectl`. `oc` supports build-specific commands for defining and triggering a build.

Before we look at BuildConfig, we need to understand two additional concepts specific to OpenShift.

An ImageStream is an OpenShift resource that references one or more container images. It is a bit similar to a Docker repository, which also contains multiple images with different tags. OpenShift maps an actual tagged image to an ImageStreamTag resource so that an ImageStream (repository) has a list of references to ImageStreamTags (tagged images). Why is this extra abstraction required? Because it allows OpenShift to emit events when an image is updated in the registry for an ImageStreamTag. Images are created during builds or when an image is pushed to the OpenShift internal registry. That way, the build or deployment controllers can listen to these events and trigger a new build or start a deployment.

To connect an ImageStream to a deployment, OpenShift uses the DeploymentConfig resource instead of the Kubernetes Deployment resource, which can only use container image references directly. However, you can still use vanilla Deployment resources in OpenShift with ImageStreams by adding some OpenShift-specific annotations.

The other concept is a *trigger*, which we can consider as a kind of listener to events. One possible trigger is `imageChange`, which reacts to the event published because of an ImageStreamTag change. As a reaction, such a trigger can, for example, cause the rebuild of another image or redeployment of the Pods using this image. You can read more about triggers and the kinds of triggers available in addition to the `imageChange` trigger in the OpenShift documentation.

Source-to-Image

Let's have a quick look at what an S2I builder image looks like. We won't go into too many details here, but an S2I builder image is a standard container image that contains a set of S2I scripts. It is very similar to Cloud Native Buildpacks but with a much simpler lifecycle that knows two mandatory commands:

`assemble`
: The script that gets called when the build starts. Its task is to take the source given by one of the configured inputs, compile it if necessary, and copy the final artifacts to the proper locations.

`run`
: Used as an entry point for this image. OpenShift calls this script when it deploys the image. This run script uses the generated artifacts to deliver the application services.

Optionally, you can also script to provide a usage message, saving the generated artifacts for so-called *incremental builds* that are accessible by the `assemble` script in a subsequent build run, or add some sanity checks.

Let's have a closer look at an S2I build in Figure 30-3. An S2I build has two ingredients: a builder image and a source input. Both are brought together by the S2I build system when a build is started—either because a trigger event was received or because we started it manually. When the build image has finished by, for example, compiling the source code, the container is committed to an image and pushed to the configured ImageStreamTag. This image contains the compiled and prepared artifacts, and the image's `run` script is set as the entry point.

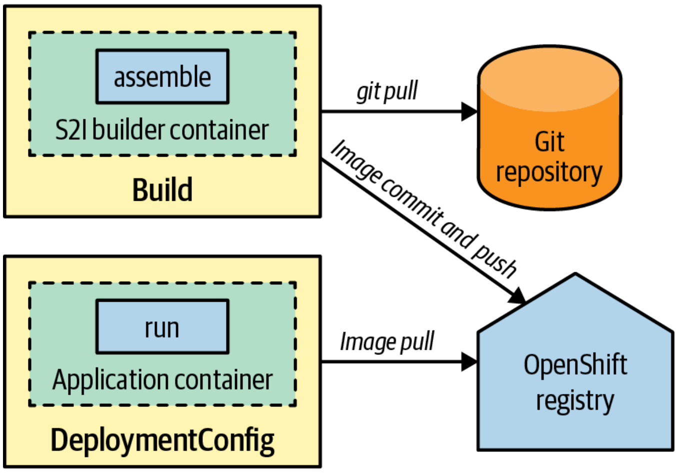

Figure 30-3. S2I build with Git source as input

Example 30-3 shows a simple Java S2I build with a Java S2I image. This build takes a source, the builder image, and produces an output image that is pushed to an ImageStreamTag. It can be started manually via `oc start-build` or automatically when the builder image changes.

Example 30-3. S2I Build using a Java builder image

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: random-generator-build
spec:
  source:     ❶
    git:
      uri: https://github.com/k8spatterns/random-generator
  strategy:  ❷
    sourceStrategy:
      from:
        kind: DockerImage
        name: fabric8/s2i-java
  output:     ❸
    to:
      kind: ImageStreamTag
      name: random-generator-build:latest
  triggers:  ❹
  - type: GitHub
    github:
      secretReference: my-secret
```

❶ Reference to the source code to fetch; in this case, pick it up from GitHub.

❷ `sourceStrategy` switches to S2I mode, and the builder image is picked up directly from Docker Hub.

❸ The ImageStreamTag to update with the generated image. It's the committed builder container after the `assemble` script has run.

❹ Rebuild automatically when the source code in the repository changes.

S2I is a robust mechanism for creating application images, and it is more secure than plain Docker builds because the build process is under full control of trusted builder images. However, this approach still has some drawbacks.

For complex applications, S2I can be slow, especially when the build needs to load many dependencies. Without any optimization, S2I loads all dependencies afresh for every build. In the case of a Java application built with Maven, there is no caching as when doing local builds. To avoid downloading half of the internet again and again, it is recommended that you set up a cluster-internal Maven repository that serves as a cache. The builder image then has to be configured to access this common repository instead of downloading the artifacts from remote repositories.

Another way to decrease the build time is to use *incremental builds* with S2I, which allows you to reuse artifacts created or downloaded in a previous S2I build. However, a lot of data is copied over from the previously generated image to the current build container, and the performance benefits are typically not much better than using a cluster-local proxy that holds the dependencies.

Another drawback of S2I is that the generated image also contains the whole build environment.[3] This fact increases not only the size of the application image but also the surface for a potential attack, as builder tools can become vulnerable too.

To get rid of unneeded builder tools like Maven, OpenShift offers *chained builds*, which take the result of an S2I build and create a slim runtime image. We look at chained builds in "Chained builds" on page 351.

Docker builds

OpenShift also supports Docker builds directly within the cluster. Docker builds work by mounting the Docker daemon's socket directly in the build container, which is then used for a `docker build`. The source for a Docker build is a Dockerfile and a directory holding the context. You can also use an `Image` source that refers an

3 This is different from Cloud Native Buildpacks, which use a separate runtime image in its stack for carrying the final artifact.

arbitrary image and from which files can be copied into the Docker build context directory. As mentioned in the next section, this technique, together with triggers, can be used for chained builds.

Alternatively, you can use a standard multistage Dockerfile to separate the build and runtime parts. Our example repository contains a fully working multistage Docker build example that results in the same image as the chained build described in the next section.

Chained builds

The mechanics of a chained build are shown in Figure 30-4. A chained build consists of an initial S2I build, which creates the runtime artifact such as a binary executable. This artifact is then picked up from the generated image by a second build, typically a Docker build.

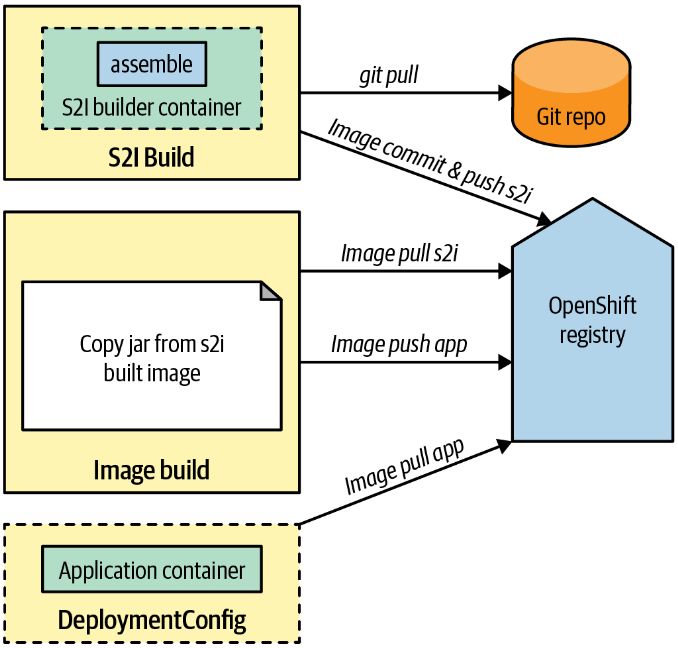

Figure 30-4. Chained build with S2I for compiling and Docker build for application image

Example 30-4 shows the setup of this second build config, which uses the JAR file generated in Example 30-3. The image that is eventually pushed to the Image-Stream `random-generator-runtime` can be used in a DeploymentConfig to run the application.

The trigger used in Example 30-4 monitors the result of the S2I build. This trigger causes a rebuild of this runtime image whenever we run an S2I build so that both ImageStreams are always in sync.

Example 30-4. Docker build for creating the application image

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: runtime
spec:
  source:
    images:
    - from:              ❶
        kind: ImageStreamTag
        name: random-generator-build:latest
      paths:
      - sourcePath: /deployments/.
        destinationDir: "."
    dockerfile: |-       ❷
      FROM openjdk:17
      COPY *.jar /
      CMD java -jar /*.jar
  strategy:              ❸
    type: Docker
  output:                ❹
    to:
      kind: ImageStreamTag
      name: random-generator:latest
  triggers:              ❺
  - imageChange:
      automatic: true
      from:
        kind: ImageStreamTag
        name: random-generator-build:latest
    type: ImageChange
```

❶ Image source references the ImageStream that contains the result of the S2I build run and selects a directory within the image that contains the compiled JAR archive.

❷ Dockerfile source for the Docker build that copies the JAR archive from the ImageStream generated by the S2I build.

❸ The `strategy` selects a Docker build.

❹ Rebuild automatically when the S2I result ImageStream changes—after a successful S2I run to compile the JAR archive.

❺ Register listener for image updates, and do a redeploy when a new image has been added to the ImageStream.

You can find the full example with installation instructions in our example repository.

As mentioned, OpenShift build, along with its most prominent S2I mode, is one of the oldest and most mature ways to safely build container images within an OpenShift cluster.

Discussion

You have seen two ways to build container images within a cluster. The plain build Pod illustrates the most crucial tasks that every build system needs to execute: fetching the source code, creating a runnable artifact from your source code, creating a container image containing the application's artifacts, pushing this image to an image registry, and finally updating any deployments so that it picks up the newly created image from that registry. This example is not meant for direct production use as it contains too many manual steps that existing build orchestrators cover more effectively.

The OpenShift build system nicely demonstrates one of the main benefits of building and running an application in the same cluster. With OpenShift's ImageStream triggers, you can connect multiple builds and redeploy your application if a build updates your application's container image. Better integration between build and deployment is a step forward to the holy grail of CD. OpenShift builds with S2I are a proven and established technology, but S2I is usable only when using the OpenShift distribution of Kubernetes.

The landscape of in-cluster build tools as of 2023 is rich and contains many exciting techniques that partly overlap. As a result, you can expect some consolidation, but new tooling will arise over time, so we'll see more implementations of the *Image Builder* pattern emerge.

More Information

- Image Builder Example
- Image Builders:
  - Buildah
  - Kaniko

    — What Is BuildKit?
    — Building Multi-Architecture Images with Buildpacks
    — Jib
    — Pack
    — Kpack
    — Ko
    — Apko: A Better Way to Build Containers?
- Build Orchestrators:
    — OpenShift Builds
    — Kbld
- Multistage Build
- Chaining S2I Builds
- Build Triggers Overview
- Source-to-Image Specification
- Incremental S2I Builds
- Building Container Images in Kubernetes: It's Been a Journey!
- Build Multi-Architecture Container Images Using Kubernetes
- Best Practices for Running Buildah in a Container

Afterword

Kubernetes is the leading platform for deploying and managing containerized distributed applications at scale. However, these on-cluster applications rely on off-cluster resources, including databases, document stores, message queues, and other cloud services. Kubernetes is not limited to managing applications within a single cluster. Kubernetes can also orchestrate off-cluster resources through various cloud services' operators. This allows Kubernetes APIs to be the single "source of truth" for a resource's desired state, not only for on-cluster containers but also for off-cluster resources. If you are already familiar with Kubernetes patterns and practices for operating applications, you can leverage this knowledge for managing and using external resources too.

The physical boundaries of a Kubernetes cluster don't always conform to the desired application boundaries. Organizations often need to deploy applications across multiple data centers, clouds, and Kubernetes clusters for a variety of reasons, such as scaling, data locality, isolation, and more. Often, the same application or a fleet of applications has to be deployed into multiple clusters, which requires multicluster deployments and orchestration. Kubernetes is frequently embedded in various third-party services and used for operating applications across multiple clusters. These services utilize the Kubernetes API as the control plane, with each cluster serving as a data plane, allowing Kubernetes to extend its reach across multiple clusters.

Today, Kubernetes has evolved beyond just a container orchestrator. It is capable of managing on-cluster, off-cluster, and multicluster resources, making it a versatile and extensible operational model for managing many kinds of resources. Its declarative YAML API and asynchronous reconciliation process have become synonymous with the resource orchestration paradigm. Its CRDs and Operators have become common extension mechanisms for merging domain knowledge with distributed systems. We believe that the majority of modern applications will be running on platforms that are offering Kubernetes APIs, or on runtimes that are heavily influenced by Kubernetes abstractions and patterns. If you are a software developer creating such applications, you must be proficient in modern programming languages to implement business functionality, as well as cloud native technologies. Kubernetes patterns will become mandatory common knowledge for integrating applications with the runtime platform. Familiarizing yourself with the Kubernetes patterns will enable you to create and run applications in any environment.

What We Covered

In this book, we covered the most popular patterns from Kubernetes, grouped as the following:

- *Foundational patterns* represent the principles that containerized applications must comply with in order to become good cloud native citizens. Regardless of the application nature, and the constraints you may face, you should aim to follow these guidelines. Adhering to these principles will help ensure that your applications are suitable for automation on Kubernetes.
- *Behavioral patterns* describe the communication mechanisms and interactions between the Pods and the managing platform. Depending on the type of workload, a Pod may run until completion as a batch job or be scheduled to run periodically. It can run as a stateless or stateful service and as a daemon service or singleton. Picking the right management primitive will help you run a Pod with the desired guarantees.
- *Structural patterns* focus on structuring and organizing containers in a Pod to satisfy different use cases. Having good cloud native containers is the first step but is not enough. Reusing containers and combining them into Pods to achieve a desired outcome is the next step.
- *Configuration patterns* cover customizing and adapting applications for different configuration needs on the cloud. Every application needs to be configured, and no one way works for all. We explore patterns from the most common to the most specialized.
- *Security patterns* describe how to constrain an application while intersecting with Kubernetes. Containerized applications have security dimensions too, and we cover application interactions with the nodes, interactions with other Pods, the Kubernetes API server, and secure configurations.
- *Advanced patterns* explore more complex topics that do not fit in any of the other categories. Some of the patterns, such as *Controller*, are mature—Kubernetes itself is built on it—and some are still evolving and might change by the time you read this book. But these patterns cover fundamental ideas that cloud native developers should be familiar with.

Final Words

Like all good things, this book has come to an end. We hope you have enjoyed reading this book and that it has changed the way you think about Kubernetes. We truly believe Kubernetes and the concepts originating from it will be as fundamental as object-oriented programming concepts are. This book is our attempt to create the Gang of Four Design Patterns but for container orchestration. We hope this is not the end but the beginning of your Kubernetes journey; it has been so for us.

Happy kubectl-ing.

Index