3.1. Build

3.3. Learning Objectives

By the end of this chapter, you should be able to:

- Learn about runtime and container options.
- Containerize an application.
- Host a local repository.
- Deploy a multi-container pod.
- Configure readinessProbes.
- Configure livenessProbes.

3.4. Container Options

There are many competing organizations working with containers. As an orchestration tool, Kubernetes is being developed to work with many of them, with the overall community moving toward open standards and easy interoperability. The early and strong presence of Docker meant that historically, this was not the focus. As Docker evolved, spreading their vendor-lock characteristics through the container creation and deployment lifecycle, new projects and features have become popular. As other projects become mature, the dependencies which had been developed into Kubernetes are being worked on and made independent.

A **container runtime** is the component which runs the containerized application upon request. Docker Engine remains the default for Kubernetes, though **cri-o**, **rkt**, and others are gaining community support.

The containerized image is moving from Docker to one that is not bound to higher-level tools and that is more portable across operating systems and environments. The Open Container Initiative (OCI) was formed to help with this. Docker donated their **libcontainer** project to form a new codebase called **runC** to support these goals. More information about runC can be found on GitHub.

Where Docker was once the only real choice for developers, the trend toward open specifications and flexibility indicates that building with vendor-neutral features is a wise choice.

3.5. Docker

Launched in 2013, Docker has become synonymous with running containerized applications. Docker made containerizing, deploying, and consuming applications easy. As a result, it became the default option in production. With an open registry of images, Docker Hub, you can download and deploy vendor or individual-created images on multiple architectures with a single and easy to use toolset. This ease meant it was the sensible default choice for any developer as well. Kubernetes defaults to using the Docker engine to run containers.

Over the past few years, Docker has continued to grow and add features, including orchestration, which they call Swarm. These added features addressed some of the pressing needs in production, but also increased the vendor-lock of the product. This has lead to an increase in open tools and specifications such as CRI-O. A developer looking toward the future would be wise to work with mostly open tools for containers and Kubernetes, but he or she should understand that Docker is still the production tool of choice for now.

3.6. Container Runtime Interface (CRI)

The goal of the Container Runtime Interface (CRI) is to allow easy integration of container runtimes with kubelet. By providing a protobuf method for API, specifications and libraries, new runtimes can easily be integrated without needing deep understanding of kubelet internals.

The project is in *alpha* stage, and has the goal of integrating Docker first. Once Docker-CRI integration is done, new runtimes can be easily added and swapped out. At the moment, **cri-o**, **rktlet** and **frakti** are listed as work-in-progress.

3.7. Rkt

The rkt runtime, pronounced rocket, provides a CLI for running containers. Announced by CoreOS in 2014, it is now part of the Cloud Native Computing Foundation family of projects. Learning from early Docker issues, it is focused on being more secure, open and interoperable. Many of its features have been met by Docker improvements. It is not quite an easy drop-in replacement for Docker, but progress has been made. **rkt** uses the **appc** specification, and can run Docker, **appc** and OCI images. It deploys immutable pods.

There has been a lot of attention to the project and it was expected to be the leading replacement for Docker until **cri-o** became part of the official Kubernetes Incubator.

3.8. CRI-O

This project is currently in incubation as part of Kubernetes. It uses the Kubernetes Container Runtime Interface with OCI-compatible runtimes, thus the name CRI-O. Currently, there is support for **runC** (default) and Clear Containers, but a stated goal of the project is to work with any OCI-compliant runtime.

While newer than Docker or rkt, this project has gained major vendor support due to its flexibility and compatibility.

3.9. Containerd

The intent of the **containerd** project is not to build a user-facing tool; instead, it is focused on exposing highly-decoupled low-level primitives:

- Defaults to runC to run containers according to the OCI Specifications
- Intended to be embedded into larger systems
- Minimal CLI, focused on debugging and development.

With a focus on supporting the low-level, or backend plumbing of containers, this project is better suited to integration and operation teams building specialized products, instead of typical build, ship, and run application.

3.10. Containerizing an Application

To containerize an application, you begin by creating your application. Not all applications do well with containerization. The more stateless and transient the application, the better. Also, remove any environmental configuration, as those can be provided using other tools, like ConfigMaps. Work on the application until you have a single build artifact which can be deployed to multiple environments without needing to be changed. Many legacy applications become a series of objects and artifacts, residing among multiple containers.

3.11. Creating the Dockerfile

Create a directory to hold application files. The **`docker build`** process will pull everything in the directory when it creates the image. Move the scripts and files for the containerized application into the directory.

Also, in the directory, create a *Dockerfile*. The name is important, as it must be those ten characters, beginning with a capital "D"; newer versions allow a different filename to be used after **`-f <filename>`**. This is the expected file for the **`docker build`** to know how to create the image.

Each instruction is iterated by the Docker daemon in sequence. The instructions are not case-sensitive, but are often written in uppercase to easily distinguish from arguments. This file must have the **`FROM`** instruction first. This declares the base image for the container. Following can be a collection of **`ADD`**, **`RUN`** and a **`CMD`** to add resources and run commands to populate the image. More details about the Dockerfile reference can be found in the Docker documentation.

Test the image by verifying that you can see it listed among other images, and use **`docker run <app-name>`** to execute it. Once you find the application performs as expected, you can push the image to a local repository in Docker Hub, after creating an account.

- Create Dockerfile
- Build the container
  ```
  sudo docker build -t simpleapp
  ```
- Verify the image
  ```
  sudo docker images
  sudo docker run simpleapp
  ```
- Push to the repository
  ```
  sudo docker push
  ```

3.12. Hosting a Local Repository

While Docker has made it easy to upload images to their Hub, these images are then public and accessible to the world. A common alternative is to create a local repository and push images there. While it can add administrative overhead, it can save downloading bandwidth, while maintaining privacy and security.

Once the local repository has been configured, you can use `docker tag`, then `push` to populate the repository with local images. Consider setting up an insecure repository until you know it works, then configuring TLS access for greater security.

3.13. Creating a Deployment

Once you can push and pull images using the **`docker`** command, try to run a new deployment inside Kubernetes using the image. The string passed to the **`--image`** argument includes the repository to use, the name of the application, then the version.

Use **`kubectl run`** to test the image:

```
kubectl run <Deploy-Name> --image=<repo>/<app-name>:<version>
```

```
kubectl run time-date --image=10.110.186.162:5000/simpleapp:v2.2
```

Be aware that the string "latest" is just that - a string, and has no reference to actually being the latest. Only use this string if you have a process in place to name and rename versions of the application as they become available. If this is not understood, you could be using the "latest" release, which is several releases older then the most current.

Verify the Pod shows a running status and that all included containers are running as well.

```
kubectl get pods
```

Test that the application is performing as expected, running whichever tempest and QA testing the application has. Terminate a pod and test that the application is as transient as designed.

3.14. Running Commands in a Container

Part of the testing may be to execute commands within the Pod. What commands are available depend on what was included in the base environment when the image was created. In keeping to a decoupled and lean design, it's possible that there is no shell, or that the Bourne shell is available instead of bash. After testing, you may want to revisit the build and add resources necessary for testing and production.

Use the `-it` options for an interactive shell instead of the command running without interaction or access.

If you have more than one container, declare which container:

```
kubectl exec -it <Pod-Name> --/bin/bash
```

3.15. Multi-Container Pod

It may not make sense to recreate an entire image to add functionality like a shell or logging agent. Instead, you could add another container to the pod, which would provide the necessary tools.

Each container in the pod should be transient and decoupled. If adding another container limits the scalability or transient nature of the original application, then a new build may be warranted.

3.16. readinessProbe

Oftentimes, our application may have to initialize or be configured prior to being ready to accept traffic. As we scale up our application, we may have containers in various states of creation. Rather than communicate with a client prior to being fully ready, we can use a *readinessProbe*. The container will not accept traffic until the probe returns a healthy state.

With the **`exec`** statement, the container is not considered ready until a command returns a zero exit code. As long as the return is non-zero, the container is considered not ready and the probe will keep trying.

Another type of probe uses an HTTP GET request (*httpGet*). Using a defined header to a particular port and path, the container is not considered healthy until the web server returns a code **`200-399`**. Any other code indicates failure, and the probe will try again.

The TCP Socket probe (*tcpSocket*) will attempt to open a socket on a predetermined port, and keep trying based on *periodSeconds*. Once the port can be opened, the container is considered healthy.

3.17. livenessProbe

Just as we want to wait for a container to be ready for traffic, we also want to make sure it stays in a healthy state. Some applications may not have built-in checking, so we can use *livenessProbes* to continually check the health of a container. If the container is found to fail a probe, it is terminated. If under a controller, a replacement would be spawned.

Both probes are often configured into a deployment to ensure applications are ready for traffic and remain healthy.

3.18. Testing

With the decoupled and transient nature and great flexibility, there are many possible combinations of deployments. Each deployment would have its own method for testing. No matter which technology is implemented, the goal is the end user getting what is expected. Building a test suite for your newly deployed application will help speed up the development process and limit issues with the Kubernetes integration.

In addition to overall access, building tools which ensure the distributed application functions properly, especially in a transient environment, is a good idea.

3.18. Testing

With the decoupled and transient nature and great flexibility, there are many possible combinations of deployments. Each deployment would have its own method for testing. No matter which technology is implemented, the goal is the end user getting what is expected. Building a test suite for your newly deployed application will help speed up the development process and limit issues with the Kubernetes integration.

In addition to overall access, building tools which ensure the distributed application functions properly, especially in a transient environment, is a good idea.