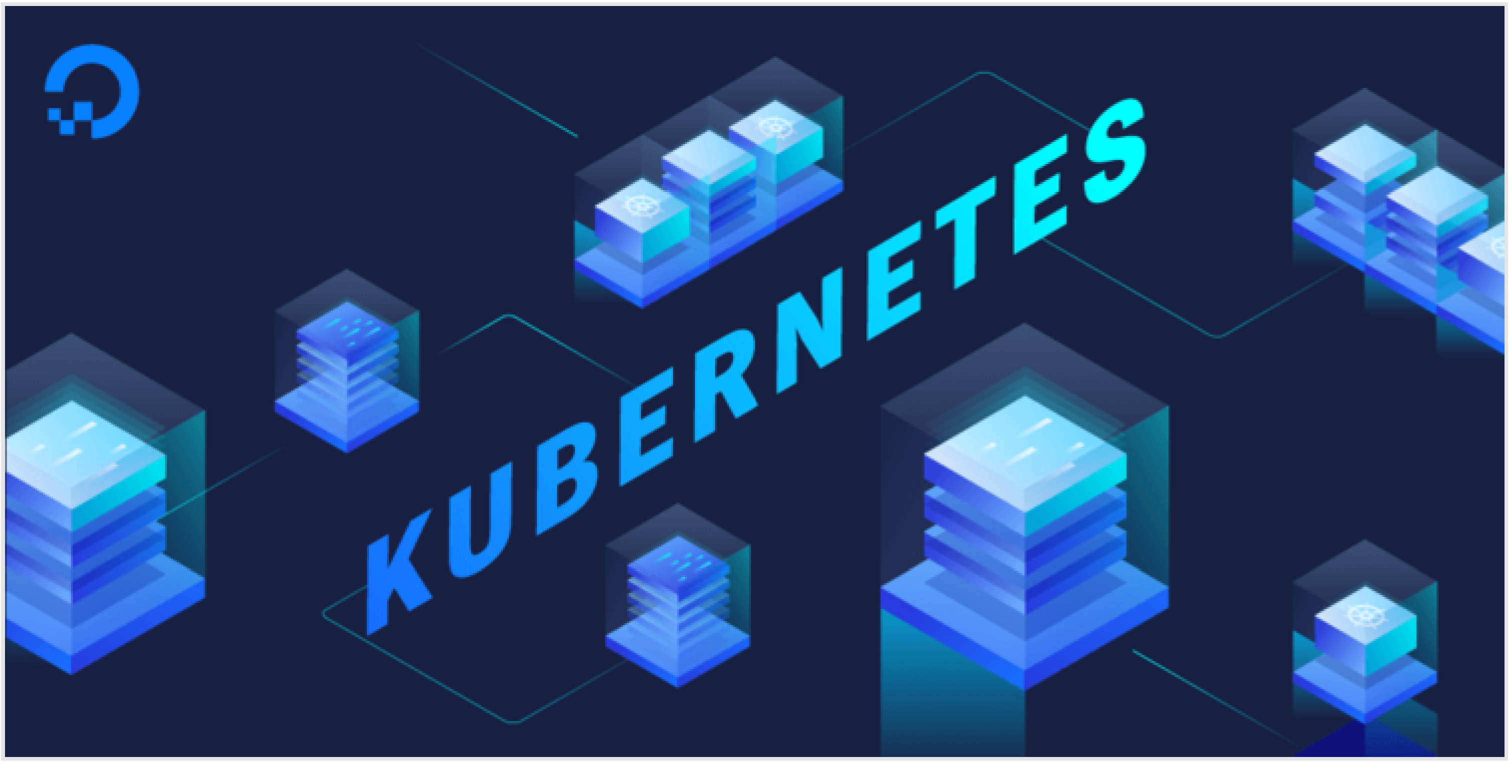

Building Optimized Containers for Kubernetes

Posted July 31, 2018 16.4k KUBERNETES CONCEPTUAL DOCKER

9

By: Justin Ellingwood

Introduction

Container images are the primary packaging format for defining applications within Kubernetes. Used as the basis for pods and other objects, images play an important role in leveraging Kubernetes' features to efficiently run applications on the platform. Well-designed images are secure, highly performant, and focused. They are able to react to configuration data or instructions provided by Kubernetes and also implement endpoints the orchestration system uses to understand internal application state.

In this article, we'll introduce some strategies for creating high quality images and discuss a few general goals to help guide your decisions when containerizing applications. We will focus on building images intended to be run on Kubernetes, but many of the suggestions apply equally to running containers on other orchestration platforms or in other contexts.

Characteristics of Efficient Container Images

Before we go over specific actions to take when building container images, we will talk about what makes a good container image. What should your goals be when designing new images? Which characteristics and what behavior are most important?

Some qualities to aim for are:

A single, well-defined purpose

Container images should have a single discrete focus. Avoid thinking of container images as virtual machines, where it can make sense to package related functionality together. Instead, treat your container images like Unix utilities, maintaining a strict focus on doing one small thing well. Applications can be coordinated outside of the container scope to compose complex functionality.

Generic design with the ability to inject configuration at runtime

Container images should be designed with reuse in mind when possible. For instance, the ability to adjust configuration at runtime is often required to fulfill basic requirements like testing your images before deploying to production. Small, generic images can be combined in different configurations to modify behavior without creating new images.

Small image size

Smaller images have a number of benefits in clustered environments like Kubernetes. They download quickly to new nodes and often have a smaller set of installed packages, which can improve security. Pared down container images make it simpler to debug problems by minimizing the amount of software involved.

Externally managed state

Containers in clustered environments experience a very volatile life cycle including planned and unplanned shutdowns due to resource scarcity, scaling, or node failures. To maintain consistency, aid in recovery and availability of your services, and to avoid losing data, it is critical that you store application state in a stable location outside of the container.

Easy to understand

It is important to try to keep container images as simple and easy to understand as possible. When troubleshooting, being able to easily reason about the problem by viewing container image configuration or testing container behavior can help you reach a resolution faster. Thinking of container images as a packaging format for your application instead of a machine configuration can help you strike the right balance.

Follow containerized software best practices

Images should aim to work within the container model instead of acting against it. Avoid implementing conventional system administration practices, like including full init systems and daemonizing applications. Log to standard out so Kubernetes can expose the data to administrators instead of using an internal logging daemon. Each of these differs from best practices for full operating systems.

Fully leverage Kubernetes features

Beyond conforming to the container model, it's important to understand and reconcile with the environment and tooling that Kubernetes provides. For example, providing endpoints for liveness and readiness checks or adjusting operation based on changes in the configuration or environment can help your applications use Kubernetes' dynamic deployment environment to their advantage.

Now that we've established some of the qualities that define highly functional container images, we can dive deeper into strategies that help you achieve these goals.

Reuse Minimal, Shared Base Layers

We can start off by examining the resources that container images are built from: base images. Each container image is built either from a *parent image*, an image used as a starting point, or from the abstract `scratch` layer, an empty image layer with no filesystem. A *base image* is a container image that serves as a foundation for future images by defining the basic operating system and providing core functionality. Images are comprised of one or more image layers built on top of one another to form a final image.

No standard utilities or filesystem are available when working directly from `scratch`, which means that you only have access to extremely limited functionality. While images created directly from `scratch` can be very streamlined and minimal, their main purpose is in defining base images. Typically, you want to build your container images on top of a parent image that sets up a basic environment that your applications run in so that you do not have to construct a complete system for every image.

While there are base images for a variety of Linux distributions, it's best to be deliberate about which systems you choose. Each new machine will have to download the parent image and any additional layers you've added. For large images, this can consume a significant amount of bandwidth and noticeably lengthen the startup time of your containers on their first run. There is no way to pare down an image that's used as a parent downstream in the container build process, so starting with a minimal parent is a good idea.

Feature rich environments like Ubuntu allow your application to run in an environment you're familiar with, but there are some tradeoffs to consider. Ubuntu images (and similar conventional distribution images) tend to be relatively large (over 100MB), meaning that any container images built from them will inherit that weight.

Alpine Linux is a popular alternative for base images because it successfully packages a lot of functionality into a very small base image (~ 5MB). It includes a package manager with sizable repositories and has most of the standard utilities you would expect from a minimal Linux environment.

When designing your applications, it's a good idea to try to reuse the same parent for each image. When your images share a parent, machines running your containers will download the parent layer only once. Afterwards, they will only need to download the layers that differ between your images. This means that if you have common features or functionality you'd like to embed in each image, creating a common parent image to inherit from might be a good idea. Images that share a lineage help minimize the amount of extra data you need to download on fresh servers.

Managing Container Layers

Once you've selected a parent image, you can define your container image by adding additional software, copying files, exposing ports, and choosing processes to run. Certain instructions in the image configuration file (a `Dockerfile` if you are using Docker) will add additional layers to your image.

For many of the same reasons mentioned in the previous section, it's important to be mindful of how you add layers to your images due to the resulting size, inheritance, and runtime complexity. To avoid building large, unwieldy images, it's important to develop a good understanding of how container layers interact, how the build engine caches layers, and how subtle differences in similar instructions can have a big impact on the images you create.

Understanding Image Layers and Build Cache

Docker creates a new image layer each time it executes a `RUN`, `COPY`, or `ADD` instruction. If you build the image again, the build engine will check each instruction to see if it has an image layer cached for the operation. If it finds a match in the cache, it uses the existing image layer rather than executing the instruction again and rebuilding the layer.

This process can significantly shorten build times, but it is important to understand the mechanism used to avoid potential problems. For file copying instructions like `COPY` and `ADD`, Docker compares the checksums of the files to see if the operation needs to be performed again. For `RUN` instructions, Docker checks to see if it has an existing image layer cached for that particular command string.

While it might not be immediately obvious, this behavior can cause unexpected results if you are not careful. A common example of this is updating the local package index and installing packages in two separate steps. We will be using Ubuntu for this example, but the basic premise applies equally well to base images for other distributions:

Package installation example Dockerfile

```
FROM ubuntu:18.04
RUN apt -y update
RUN apt -y install nginx
. . .
```

Here, the local package index is updated in one `RUN` instruction (`apt -y update`) and Nginx is installed in another operation. This works without issue when it is first used. However, if the Dockerfile is updated later to install an additional package, there may be problems:

Package installation example Dockerfile

```
FROM ubuntu:18.04
RUN apt -y update
RUN apt -y install nginx php-fpm
. . .
```

We've added a second package to the installation command run by the second instruction. If a significant amount of time has passed since the previous image build, the new build might fail. That's because the package index update instruction (`RUN apt -y update`) has *not* changed, so Docker reuses the image layer associated with that instruction. Since we are using an old package index, the version of the `php-fpm` package we have in our local records may no longer be in the repositories, resulting in an error when the second instruction is run.

To avoid this scenario, be sure to consolidate any steps that are interdependent into a single `RUN` instruction so that Docker will re-execute all of the necessary commands when a change occurs:

Package installation example Dockerfile

```
FROM ubuntu:18.04
RUN apt -y update && apt -y install nginx php-fpm
. . .
```

The instruction now updates the local package cache whenever the package list changes.

Reducing Image Layer Size by Tweaking RUN Instructions

The previous example demonstrates how Docker's caching behavior can subvert expectations, but there are some other things to keep in mind with how `RUN` instructions interact with Docker's layering system. As mentioned earlier, at the end of each `RUN` instruction, Docker commits the changes as an additional image layer. In order to exert control over the scope of the image layers produced, you can clean up unnecessary files in the final environment that will be committed by paying attention to the artifacts introduced by the commands you run.

In general, chaining commands together into a single `RUN` instruction offers a great deal of control over the layer that will be written. For each command, you can set up the state of the layer (`apt -y update`), perform the core command (`apt install -y nginx php-fpm`), and remove any unnecessary artifacts to clean up the environment before it's committed. For example, many Dockerfiles chain `rm -rf /var/lib/apt/lists/*` to the end of `apt` commands, removing the downloaded package indexes, to reduce the final layer size:

Package installation example Dockerfile

```
FROM ubuntu:18.04
RUN apt -y update && apt -y install nginx php-fpm && rm -rf /var/lib/apt/lists/*
. . .
```

To further reduce the size of the image layers you are creating, trying to limit other unintended side effects of the commands you're running can be helpful. For instance, in addition to the explicitly declared packages, `apt` also installs "recommended" packages by default. You can include `--no-install-recommends` to your `apt` commands to remove this behavior. You may have to experiment to find out if you rely on any of the functionality provided by recommended packages.

We've used package management commands in this section as an example, but these same principles apply to other scenarios. The general idea is to construct the prerequisite conditions, execute the minimum viable command, and then clean up any unnecessary artifacts in a single `RUN` command to reduce the overhead of the layer you'll be producing.

Using Multi-stage Builds

Multi-stage builds were introduced in Docker 17.05, allowing developers to more tightly control the final runtime images they produce. Multi-stage builds allow you to divide your Dockerfile into multiple sections representing distinct stages, each with a `FROM` statement to specify separate parent images.

Earlier sections define images that can be used to build your application and prepare assets. These often contain build tools and development files that are needed to produce the application, but are not necessary to run it. Each subsequent stage defined in the file will have access to artifacts produced by previous stages.

The last `FROM` statement defines the image that will be used to run the application. Typically, this is a pared down image that installs only the necessary runtime requirements and then copies the application artifacts produced by previous stages.

This system allows you worry less about optimizing `RUN` instructions in the build stages since those container layers will not be present in the final runtime image. You should still pay attention to how instructions interact with layer caching in the build stages, but your efforts can be directed towards minimizing build time rather than final image size. Paying attention to instructions in the final stage is still important in reducing image size, but by separating the different stages of your container build, it's easier to to obtain streamlined images without as much Dockerfile complexity.

Scoping Functionality at the Container and Pod Level

While the choices you make regarding container build instructions are important, broader decisions about how to containerize your services often have a more direct impact on your success. In this section, we'll talk a bit more about how to best transition your applications from a more conventional environment to running on a container platform.

Containerizing by Function

Generally, it is good practice to package each piece of independent functionality into a separate container image.

This differs from common strategies employed in virtual machine environments where applications are frequently grouped together within the same image to reduce the size and minimize the resources required to run the VM. Since containers are lightweight abstractions that don't virtualize the entire operating system stack, this tradeoff is less compelling on Kubernetes. So while a web stack virtual machine might bundle an Nginx web server with a Gunicorn application server on a single machine to serve a Django application, in Kubernetes these might be split into separate containers.

Designing containers that implement one discrete piece of functionality for your services offers a number of advantages. Each container can be developed independently if standard interfaces between services are established. For instance, the Nginx container could potentially be used to proxy to a number of different backends or could be used as a load balancer if given a different configuration.

Once deployed, each container image can be scaled independently to address varying resource and load constraints. By splitting your applications into multiple container images, you gain flexibility in development, organization, and deployment.

Combining Container Images in Pods

In Kubernetes, **pods** are the smallest unit that can be directly managed by the control plane. Pods consist of one or more containers along with additional configuration data to tell the platform how those components should be run. The containers within a pod are always scheduled on the same worker node in the cluster and the system automatically restarts failed containers. The pod abstraction is very useful, but it introduces another layer of decisions about how to bundle together the components of your applications.

Like container images, pods also become less flexible when too much functionality is bundled into a single entity. Pods themselves can be scaled using other abstractions, but the containers within cannot be managed or scaled independently. So, to continue using our previous example, the separate Nginx and Gunicorn containers should probably not be bundled together into a single pod so that they can be controlled and deployed separately.

However, there are scenarios where it does make sense to combine functionally different containers as a unit. In general, these can be categorized as situations where an additional container supports or enhances the core functionality of the main container or helps it adapt to its deployment environment. Some common patterns are:

- **Sidecar**: The secondary container extends the main container's core functionality by acting in a supporting utility role. For example, the sidecar container might forward logs or update the filesystem when a remote repository changes. The primary container remains focused on its core responsibility, but is enhanced by the features provided by the sidecar.
- **Ambassador**: An ambassador container is responsible for discovering and connecting to (often complex) external resources. The primary container can connect to an ambassador container on well-known interfaces using the internal pod environment. The ambassador abstracts the backend resources and proxies traffic between the primary container and the resource pool.
- **Adaptor**: An adaptor container is responsible for normalizing the primary containers interfaces, data, and protocols to align with the properties expected by other components. The primary container can operate using native formats and the adaptor container translates and normalizes the data to communicate with the outside world.

As you might have noticed, each of these patterns support the strategy of building standard, generic primary container images that can then be deployed in a variety contexts and configurations. The secondary containers help bridge the gap between the primary container and the specific deployment environment being used. Some sidecar containers can also be reused to adapt multiple primary containers to the same environmental conditions. These patterns benefit from the shared filesystem and networking namespace provided by the pod abstraction while still allowing independent development and flexible deployment of standardized containers.

Designing for Runtime Configuration

There is some tension between the desire to build standardized, reusable components and the requirements involved in adapting applications to their runtime environment. Runtime configuration is one of the best methods to bridge the gap between these concerns. Components are built to be both general and flexible and the required behavior is outlined at runtime by providing the software with additional configuration information. This standard approach works for containers as well as it does for applications.

Building with runtime configuration in mind requires you to think ahead during both the application development and containerization steps. Applications should be designed to read values from command line parameters, configuration files, or environment variables when they are launched or restarted. This configuration parsing and injection logic must be implemented in code prior to containerization.

When writing a Dockerfile, the container must also be designed with runtime configuration in mind. Containers have a number of mechanisms for providing data at runtime. Users can mount files or directories from the host as volumes within the container to enable file-based configuration. Likewise, environment variables can be passed into the internal container runtime when the container is started. The `CMD` and `ENTRYPOINT` Dockerfile instructions can also be defined in a way that allows for runtime configuration information to be passed in as command parameters.

Since Kubernetes manipulates higher level objects like pods instead of managing containers directly, there are mechanisms available to define configuration and inject it into the container environment at runtime. Kubernetes **ConfigMaps** and **Secrets** allow you to define configuration data separately and then project the values into the container environment as environment variables or files at runtime. ConfigMaps are general purpose objects intended to store configuration data that might vary based on environment, testing stage, etc. Secrets offer a similar interface but are specifically designed for sensitive data, like account passwords or API credentials.

By understanding and correctly using the runtime configuration options available throughout each layer of abstraction, you can build flexible components that take their cues from environment-provided values. This makes it possible to reuse the same container images in very different scenarios, reducing development overhead by improving application flexibility.

Implementing Process Management with Containers

When transitioning to container-based environments, users often start by shifting existing workloads, with few or no changes, to the new system. They package applications in containers by wrapping the tools they are already using in the new abstraction. While it is helpful to use your usual patterns to get migrated applications up and running, dropping in previous implementations within containers can sometimes lead to ineffective design.

Treating Containers like Applications, Not Services

Problems frequently arise when developers implement significant service management functionality within containers. For example, running systemd services within the container or daemonizing web servers may be considered best practices in a normal computing environment, but they often conflict with assumptions inherent in the container model.

Hosts manage container life cycle events by sending signals to the process operating as PID (process ID) 1 inside the container. PID 1 is the first process started, which would be the init system in traditional computing environments. However, because the host can only manage PID 1, using a conventional init system to manage processes within the container sometimes means there is no way to control the primary application. The host can start, stop, or kill the internal init system, but can't manage the primary application directly. The signals sometimes propagate the intended behavior to the running application, but this adds complexity and isn't always necessary.

Most of the time, it is better to simplify the running environment within the container so that PID 1 is running the primary application in the foreground. In cases where multiple processes must be run, PID 1 is responsible for managing the life cycle of subsequent processes. Certain applications, like Apache, handle this natively by spawning and managing workers that handle connections. For other applications, a wrapper script or a very simple init system like dumb-init or the included tini init system can be used in some cases. Regardless of the implementation you choose, the process running as PID 1 within the container should respond appropriately to `TERM` signals sent by Kubernetes to behave as expected.

Managing Container Health in Kubernetes

Kubernetes deployments and services offer life cycle management for long-running processes and reliable, persistent access to applications, even when underlying containers need to be restarted or the implementations themselves change. By extracting the responsibility of monitoring and maintaining service health out of the container, you can leverage the platform's tools for managing healthy workloads.

In order for Kubernetes to manage containers properly, it has to understand whether the applications running within containers are healthy and capable of performing work. To enable this, containers can implement liveness probes: network endpoints or commands that can be used to report application health. Kubernetes will periodically check defined liveness probes to determine if the container is operating as expected. If the container does not respond appropriately, Kubernetes restarts the container in an attempt to reestablish functionality.

Kubernetes also provides readiness probes, a similar construct. Rather than indicating whether the application within a container is healthy, readiness probes determine whether the application is ready to receive traffic. This can be useful when a containerized application has an initialization routine that must complete before it is ready to receive connections. Kubernetes uses readiness probes to determine whether to add a pod to or remove a pod from a service.

Defining endpoints for these two probe types can help Kubernetes manage your containers efficiently and can prevent container life cycle problems from affecting service availability. The mechanisms to respond to these types of health requests must be built into the application itself and must be exposed in the Docker image configuration.

Conclusion

In this guide, we've covered some important considerations to keep in mind when running containerized applications in Kubernetes. To reiterate, some of the suggestions we went over were:

- Use minimal, shareable parent images to build images with minimal bloat and reduce startup time
- Use multi-stage builds to separate the container build and runtime environments
- Combine Dockerfile instructions to create clean image layers and avoid image caching mistakes
- Containerize by isolating discrete functionality to enable flexible scaling and management
- Design pods to have a single, focused responsibility
- Bundle helper containers to enhance the main container's functionality or to adapt it to the deployment environment
- Build applications and containers to respond to runtime configuration to allow greater flexibility when deploying
- Run applications as the primary processes in containers so Kubernetes can manage life cycle events
- Develop health and liveness endpoints within the application or container so that Kubernetes can monitor the health of the container

Throughout the development and implementation process, you will need to make decisions that can affect your service's robustness and effectiveness. Understanding the ways that containerized applications differ from conventional applications, and learning how they operate in a managed cluster environment can help you avoid some common pitfalls and allow you to take advantage of all of the capabilities Kubernetes provides.

By: Justin Ellingwood

Related Tutorials

An Introduction to Service Meshes

How To Deploy a PHP Application with Kubernetes on Ubuntu 16.04

How to Manually Set Up a Prisma Server on Ubuntu 18.04

How To Use Traefik as a Reverse Proxy for Docker Containers on Debian 9

How To Set Up a Private Docker Registry on Ubuntu 18.04

2 Comments

Leave a comment...

Log In to Comment

jbanety *August 9, 2018*

1 Hi Justin,

Thanks for your great articles about Kubernetes and containarized apps.
Very helpful to learn how to change my way of desiging my next app.

JB

zooloo2014 *December 7, 2018*

0 This is an excellent article! Thanks