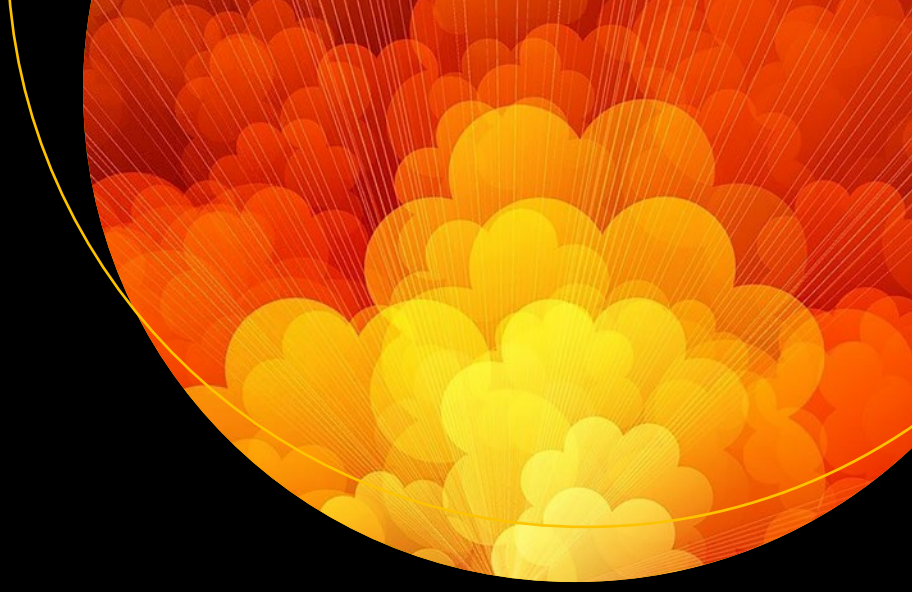

Cloud Native Applications with Docker and Kubernetes

Design and Build Cloud Architecture and Applications with Microservices, EMQ, and Multi-Site Configurations

Jonathan Bartlett

Apress®

Cloud Native Applications with Docker and Kubernetes

Design and Build Cloud Architecture and Applications with Microservices, EMQ, and Multi-Site Configurations

Jonathan Bartlett

Apress®

Cloud Native Applications with Docker and Kubernetes: Design and Build Cloud Architecture and Applications with Microservices, EMQ, and Multi-Site Configurations

Jonathan Bartlett
Tulsa, OK, USA

ISBN-13 (pbk): 978-1-4842-8875-7 ISBN-13 (electronic): 978-1-4842-8876-4
https://doi.org/10.1007/978-1-4842-8876-4

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Celestin Suresh John
Development Editor: James Markham
Coordinating Editor: Mark Powers

Cover designed by eStudio Calamar

Cover image by Fullvector on Freepik (www.freepik.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub (https://github.com/Apress). For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Jonathan Bartlett is a senior software developer at McElroy Manufacturing. His career has had a primary focus on developing data models and web APIs for both internally and externally facing business applications. He has worked with a large variety of languages, platforms, and development styles in his more than 20 years of experience in the industry.

Jonathan has been educating programmers for well over a decade. His first book, *Programming from the Ground Up*, is an Internet classic and was endorsed by Joel Spolsky, cofounder of StackExchange. It was one of the first open source books and has been used by a generation of programmers to learn how computers work from the inside out, using assembly language as a starting point. After more than 15 years, *Programming from the Group Up* was recently updated and rereleased as *Learn to Program with Assembly: Foundational Learning for New Programmers*.

Recently, Jonathan released *Programming for Absolute Beginners* which is focused on teaching brand new programmers about computers, the Internet, and JavaScript programming, based on his experience teaching programming to high-school and college students. Additionally, Jonathan has written several books on the interplay of philosophy, math, and science, including *Calculus from the Ground Up*, *Engineering and the Ultimate*, and *Naturalism and Its Alternatives in Scientific Methodologies*. For those interested in the more physical aspects of computing, he wrote *Electronics for Beginners*.

Jonathan also writes developer-focused articles for a number of technology websites. He is currently a writer for the technology blog *MindMatters*, and you can find his articles at `https://mindmatters.ai/author/jbartlett`. He has also written for IBM's DeveloperWorks, Linux.com, Medium.com, and Classical Conversations.

Jonathan also participates in a variety of academic work. He is an associate fellow of the *Walter Bradley Center for Natural and Artificial Intelligence*. There he does research

into fundamental mathematics and the mathematics of artificial intelligence. He also serves on the editorial board for the journal *Bio-Complexity*, focusing on reviewing information-theoretic papers for the journal. Jonathan also served as editor of the book *Controllability of Dynamic Systems: The Green's Function Approach*, which received the RA Presidential Award of the Republic of Armenia in the area of technical sciences and information technologies. Jonathan also spends time teaching at a homeschool co-op through Classical Conversations.

Jonathan is married to his wife Christa. They have been together for over 20 years and have had five children.

About the Technical Reviewer

Shivakumar R. Goniwada is a chief enterprise architect, technology leader, and inventor with more than 24 years of experience focusing on Cloud Native Elements, Enterprise Architecture Setup, and Big Transformation Projects. He currently works at Accenture and leads a highly experienced technology enterprise and cloud architects. In his 24 years of experience, he has led many highly complex projects across industries and geographic regions. He has ten software patents to his name in the areas of cloud, microservices architecture, software engineering, and IoT (a few yet to publish). He has been a speaker at multiple global and in-house conferences. He holds Master Technology Architecture Accenture, Google Professional, AWS, and data science certifications. His executive MBA is from the MIT Sloan School of Management.

Source Code

All source code used in this book is available to readers at github.com/apress/cloud-native-applications-docker-kubernetes.

Acknowledgments

There are a lot of people I would like to thank for this book. However, the most influential people that inspired this book were Clark Ritchie and Tyler Brown (and Chris Zenthoefer for introducing me to them). Both Clark and Tyler helped me to think more explicitly about clouds and cloud services and move my thinking from traditional forms of system administration and system architecture to more cloud native ways of thinking. I actually want to thank all my friends at Specialized Bicycles for the time I spent with them. You all were a great crew to work with, and I am extremely thankful for the time I spent with you all.

CHAPTER 1

Introduction

There has been a very large shift over the last several years in the way that web applications are built and deployed. The modern approach is termed "cloud native," and it is an approach that is geared toward maximizing the benefit and ease of development and scalability that modern cloud computing offers. Whether you have spent years building applications "the old way" or you are just starting out and want to start building applications the right way, this book will set you on the right track to become a cloud native master.

1.1 A Brief History of Web Service Hosting

Oftentimes it is hard to understand why something is the way that it is unless you understand its history. To start with, I want to present a quick overview of the history of web infrastructure hosting to give you a better feel for what sorts of problems cloud native development solves.

1.1.1 The Old Way

Way back in the early days of the Internet, web applications were hosted on specific server machines. That is, when you wanted to host a web application, you had to purchase a physical machine, install Linux or some other operating system on it, and then pay an Internet Service Provider to put your machine on their network. This process was both time-consuming and expensive, often costing hundreds of dollars a month just to rent the space where you install your own server.

If you needed more capacity, this process was equally painful. You had to buy another server, install another copy of Linux on it, make sure it was configured exactly like your other server, and then pay to host that one as well.

J. Bartlett, *Cloud Native Applications with Docker and Kubernetes*,
https://doi.org/10.1007/978-1-4842-8876-4_1

Additionally, you had to do quite a bit of work to share the load between servers. You either had to implement a DNS trick to get clients to split their time between machines, or you had to buy *another* piece of equipment, a load balancer, which took incoming traffic and balanced it between machines.

If one machine had faulty hardware, well, you are back to the drawing board. If you needed to upgrade the operating system, you had to log in to each machine, take it out of the lineup, and manually upgrade components. This was ridiculously hard work, and the system administrators who performed these tasks also cost a lot of money. Systems were eventually built to make the task of synchronizing such systems easier, but those had their own complexities to learn.

1.1.2 The Virtual Private Server

The first move to "the cloud" is known as a Virtual Private Server, or VPS. What happened is that advances in processors and operating systems allowed administrators to run virtual machines (VMs) underneath a host machine at essentially the same speed as a non-virtual machine. A virtual machine is simply a "fake" machine running under a real machine. For instance, you can run a Windows operating system within a Macintosh operating system using a virtual machine. The Windows operating system "thinks" it is a real computer, when actually it is just acting like a sub-computer of the Macintosh.

Virtual machines have been around since the 1970s. However, it was not until much later that virtual machines on standard processors were able to function at a speed that is close to "bare metal" and simultaneously be sufficiently secure that you don't have to worry about two VMs on the same host spying on each other. The way that a virtual machine works is that the actual hardware runs a slimmed-down operating system called a hypervisor. The hypervisor is in charge of creating and running virtual machines under it and making sure the resources of the computer (RAM, CPU, hard drives, and input/output operations) are properly distributed between the various VMs that are running under it.

Figure 1-1 shows a conceptual view of how physical machines were broken up into virtual machines (VMs). The hypervisor is in primary control of the machine and can partition the machine to run VMs underneath it. Each VM has the full functionality of a computer and runs its own operating system and applications. They even have their own IP addresses and disk drives (though likely these are virtual as well). The VMs have no idea that they are virtual machines, and, from the point of view of the VM itself, it

thinks that it is a full computer. Its operation and usage of system resources, however, is governed by the hypervisor.

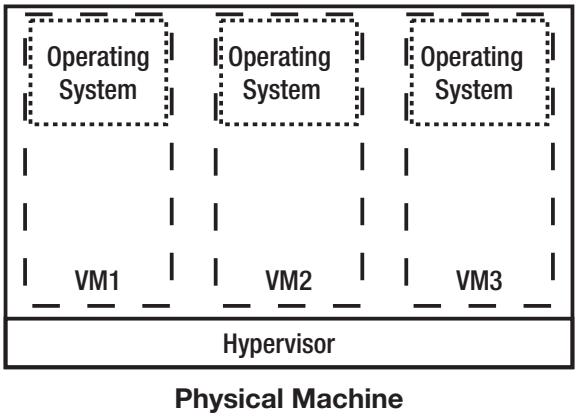

Figure 1-1. *A Physical Machine Hosting Multiple VMs*

Virtual machines paved the way for all sorts of innovations. To start with, system administrators no longer had to worry about getting the right size machine for the task. They could just purchase a high-powered machine and then later decide how to divide the computing power up. If they bought a machine with eight CPUs, 64GB of memory, and 10TB of drive space, they could decide to give two CPUs, 1GB of memory, and 3TB of drive space to one VM, three CPUs, 2GB of memory, and 1TB of drive space to another VM, and so forth. Individual CPUs can even be split—with a VM being given only a portion of a CPU's time.

Since the actual running operating system was virtualized, this made installation and maintenance a lot easier. If one physical host was having hardware problems, you could pause the VMs running on that physical machine, move them to another physical host, and start them back up. It is easy to replicate VMs, because their hard drives are all visible to the hypervisor. Thus, management of the servers was greatly simplified.

Hosting companies were able to take great advantage of these advances as well. They would offer users the ability to buy VMs with just a click of a button. You just select which system image you want to install and what size of computer you want and click a button, and the hosting platform would find a machine with enough capacity for your needs, allocate a VM, and send you the IP address. These are known as virtual private servers, or VPSs. Additionally, VPS providers usually allowed you to request the allocation of a load balancer to distribute traffic to all of your different VPSs.

1.1.3 From Virtual Private Servers to Containers

The problem with VPSs is that they actually have quite a bit of somewhat needless overhead:

- Each VPS has a complete copy of the operating system stored on-disk.
- Each VPS is running a complete copy of the operating system in its memory.
- Each VPS has to continually run general system management tasks.

This all seems like overkill, especially considering that what we usually want to run is just a single process, such as a web server or database. What if we didn't really need all of that overhead? What if there was another way to give developers something similar to full access to a server, but didn't have the overhead of a full machine?

Enter the container. Containers are similar to virtual machines in that they give a developer something that feels like an isolated machine. However, a container actually shares its operating system and its filesystem with other containers on the same host *without affecting them*. That is, each container thinks and acts like it is an independent machine, but it is usually sharing both the operating system and the filesystem with other containers without the application even being aware of this. Figure 1-2 shows what this looks like. You can run many more containers on a single physical machine than you can run VMs, because the entire operating system is shared between all of the containers. We will describe the magic that manages to keep these containers from running over each other in Chapter 2. But, for now, just recognize that it works.

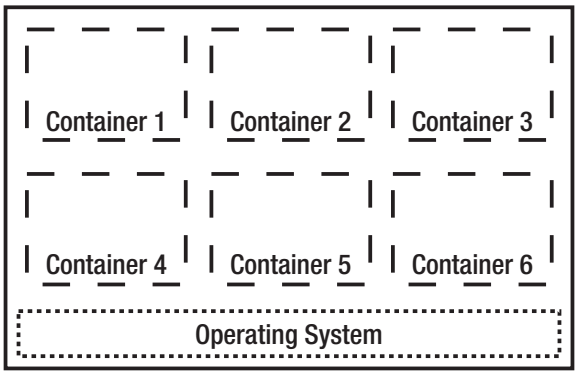

Figure 1-2. *A Physical Machine Hosting Containers*

VMs and VPSs, while flexible, had quite a bit of overhead, as each instance required all the overhead of a regular operating system. Now, with containers, all of this overhead is shared by all or most of the containers running on a physical machine. Each container only adds the memory and disk space used by the specific process or processes that it was launched to run, but, for the process, it has the same flexibility as if it had control of the whole machine.

So, a typical Linux server usually has an installed footprint of 10-20 gigabytes of disk space. Such a server is usually running a bunch of kernel (operating system) processes, the `init` program, a system logger, a terminal program, an SSH service, and possibly other system services. These services typically use up about a half of a gigabyte of working memory (including disk cache). With containers, this price only has to be paid once per physical machine rather than once per VPS. Even so, containers still allow developers the option of making any modification they want on their own running instance.

Additionally, a set of standards developed around containers and container images which allow them to be created, manipulated, stored, and retrieved by tools from a variety of different projects or vendors. For instance, there might be a standard container image for a base install of Debian, and I can pull down that container image and install my application on top of it, and create a new container image out of the result, and store it back on the service.

Additionally, because running containers utilizes so little overhead, it is possible for developers to run multiple containers on their own personal computers. In fact, they can set up containers that are exact replicas of the setup used on the servers, and it often has minimal impact on the performance of their own computers.

This almost eliminates an entire class of issues caused by developer's local development environments. Historically, the configuration of a developer's personal computer does not exactly match what is on the server. They may be running a different version of the programming language, database, or any other component of the environment. With containers, developers can specify, run, and validate the precise operating environment on their local computers as will be running on the server. The entire operating system image itself can be the subject of development and testing.

1.1.4 Cloud Native Infrastructure

The move to containerized computing then paved the way for what is known as cloud native infrastructure. In cloud native infrastructure, entire application infrastructures are specified, created, and deployed using code. That is, you don't have to go ask for a bunch

of servers, install the image you want on them, and then add them to a load balancer. Instead, you tell a separate control service that you want to build a load-balanced system using a particular image and run between three and ten copies of it depending on the load. The control service then launches everything you need and keeps everything in sync.[1]

In a cloud native infrastructure, the control system does most of the routine management tasks that were previously done by system administrators. If a container dies, it gets cleaned up and restarted by the control service. If you start to get a higher load, depending on how the cluster is configured, the control service may launch more copies of your container. The control service will handle registering each of these containers with the load balancer, as well as finding which machines it should run the containers on.

Cloud native applications essentially turn infrastructure into code. You write code which specifies what your network infrastructure should look like, and the cloud makes it happen. This allows the incorporation of source control tools into the development of infrastructure itself. The configuration of your infrastructure can then be version-controlled, allowing easier tracing of configuration changes for diagnosing problems.

Today, there are a lot of choices for building cloud native applications. However, there is one system which runs on practically every hosting platform—Kubernetes, also known as K8s (pronounced "Kates"). Kubernetes is based on a system originally developed by Google for managing their own infrastructure. However, Kubernetes is an open source product and an open standard which is used by numerous different cloud providers, including the big guys like Google, Microsoft, Amazon AWS, and IBM, as well as the smaller players such as Linode and Digital Ocean. Additionally, you can create your own private clouds using software such as Red Hat's OpenShift and VMWare's Tanzu. Kubernetes essentially gives you the ability to build a scalable cloud infrastructure which doesn't tie you down to any specific provider.

[1] Just to note, you can run cloud native systems using traditional VMs instead of containers. Infrastructure as code can specify full-blown VMs to image and deploy. However, most agree that containers allow for the maximum amount of benefit of the cloud native approach.

1.2 An Overview of This Book

The goal of this book is to provide you with an understanding of the tools for cloud native development and how they are put together into a scalable web application. To this end, the book is divided into three parts. The first part will cover containers, how they work, and how to work with them. Without a good understanding of containers, it is hard to really understand what is really going on under the hood. The second part will be an introduction to Kubernetes. This part will cover the basics of Kubernetes and will show you how to get a small Kubernetes cluster up and running. The third part will show how to think about application architecture from a cloud native perspective.

1.3 Prerequisites

This book has very few prerequisites. Even if you aren't familiar with the specific tools we are using, this book has enough step-by-step instructions that you should be able to follow it fairly easily.

Since the book focuses on web applications, it assumes that you are familiar with the absolute basics of HTML, CSS, and the basics of how the Internet works (i.e., domain names, IP addresses, etc.). If you are not familiar with these topics, you should put this book down and pick up a copy of my book, *Programming for Absolute Beginners*. *Programming for Absolute Beginners* doesn't hit every topic you need to know, but if you understand its concepts, you should be able to work through most of the examples in this book.

This book doesn't focus on programming languages, as the details provided apply no matter what programming language you are using. For the example applications, this book will focus on Ruby and Sinatra, simply to make the examples small, self-contained, and easy to follow. There is nothing Ruby- or Sinatra-specific that we will be focusing on.

This book also presumes that you have a basic familiarity with databases and SQL. This is not an absolute requirement, but the database code itself is left largely unexplained. However, the SQL code is fairly simple and should be self-evident to anyone with even a passing knowledge of SQL.

Finally, this book uses Linux as the underlying operating system for its containers. However, even if you don't know anything about Linux, this book gives you every command you need to type, so you don't actually have to know a whole lot about Linux to use this book. Appendix A has a list of common Linux commands for reference if

you want to know more or need a general introduction to command-line systems. The reason that Linux was chosen was because it is the most common operating system used in containerized environments. Additionally, most other operating systems can run Linux containers, but not the other way around (i.e., you can run Linux containers under Windows, but you can't run Windows containers under Linux). Also, because all of the components we will be using are all free and open source software, there are no licensing considerations for their use.

1.4 Typographical Conventions

Some commands in Kubernetes get kind of long. If a command goes longer than a single line, we will put a backslash (\) at the end of a line to indicate that it continues on the next line. The reason for this convention is that most command-line shells will interpret this the same way—if you add a backslash and go to a new line, then the shell treats that as if it were a continuation of the current line. However, you can put it all on the same line if you wish; just remove the backslashes.

For instance, we might ask you to type the following command:

```
echo this is a really long \
    command
```

That is identical to just typing the following:

```
echo this is a really long command
```

Also note that, when talking about Kubernetes, many things that you might speak about generically are actually proper names, so those names are capitalized. For instance, in Kubernetes, there is a resource called a Service, so, when referring specifically to a Kubernetes Service, the word "Service" will always be capitalized. Also, some Kubernetes resources combine two words together, like "ReplicaSet." This book follows the Kubernetes naming conventions and keeps these words together, capitalized in the same way that the Kubernetes documentation does.

PART I

An Introduction to Containers

CHAPTER 2

Containers Under the Hood

As businesses move more and more infrastructure off of their own premises to online services, finding the best way to manage that infrastructure becomes more and more important. As we will show throughout this book, containers enable development teams to have more reliable, repeatable, and testable services that can be easily deployed at any scale. Here, we are looking under the hood at containers, which will be used as the foundation for infrastructure management tools. Over the last decade, the popularity of containers for such tools has grown rapidly.

One particular tool stands out among others—Docker. Docker is a tool that enables easy management of the entire container ecosystem. Docker has a command-line tool that makes running containers easy, as well as creating, downloading, and managing container images. Docker also comes with a desktop application, Docker Desktop, that makes container management even easier. Finally, the company that built the Docker tool also runs Docker Hub, which makes it easy to store and access container images in the cloud.

Docker has been at the forefront of container management throughout its technological growth. Many of the modern standards around containers arose from features that originated in Docker, and even those that arose elsewhere were popularized by having an easy-to-use tool to make them accessible to the average developer. Docker is known as a "container runtime," as it contains the tools to make the containers boot up and execute in their own provisioned space.

In this chapter, we are going to take a look at the technologies behind containers generally (and Docker specifically) and where they came from.

J. Bartlett, *Cloud Native Applications with Docker and Kubernetes*,
https://doi.org/10.1007/978-1-4842-8876-4_2

2.1 Answering Basic Questions About Containers

Let's start by looking at the basic questions that people have about containers themselves.

2.1.1 What Are Containers?

If you're not familiar with container technology, containers sit in a strange position in the technology toolchain. They are not something you develop with explicitly (applications in containers are rarely self-conscious of this fact). In fact, as a developer, you may never touch Docker or containers. A container isn't an operating system that you run things on. You still run on your normal server-side operating systems such as Linux. It isn't a hardware technology, either.

Essentially, containers wrap up both the application and the parts of the operating system's environment that the application uses into a single bundle that can be easily deployed anywhere. This is known as the container image. When it is deployed, it runs not as the primary system for the machine but as a process *within* your current operating system. That's right; you run a slimmed-down operating system inside your current operating system, which could either be your local development machine or a server. This running system is the container itself.

As far as the container is concerned, it thinks it is its own machine. It gets its own IP address, has its own filesystem, and even has its own isolated process tree. However, as we will see, it is actually sharing the core operating system with other containers on the system, but modern operating systems are able to sufficiently isolate processes from each other that they can't even see that they are sharing the operating system with others.

2.1.2 What Problems Do Containers Solve?

Containers solve two huge problems for deployments. The first is configuration management. Because Docker packages up both the operating system and the application into a single bundle, it allows for testing the app in the same configuration that it is going to run in when it is deployed on the server. The switch from staging to production can usually be handled entirely by changes to environment variables or runtime arguments.

Historically, developers have had the problem that they may be relying on a tool on their local machine that isn't available on the server or a version of something that is different on the server. With containers, both the operating system and the application are saved together before deployment, so the developers can be sure not only of their code but of the environment that it is running in.

The second problem is infrastructure management. Since containers run as a process under the main operating system, you can run multiple containers on a single machine. This means that, instead of choosing the exact right size for each machine, you can simply purchase or provision a large machine and run several containers on that machine. If you need more horsepower for one of your containers, you can simply move it to a bigger server.

Additionally, containers are becoming the standard for other infrastructure management tools. Amazon Web Services, which started out with their own infrastructure management tool (Elastic Beanstalk), has more recently been pushing containers instead, using their Elastic Container Service. Kubernetes, the open source cloud management system, also runs using containers and is available on Google Cloud, Linode, Azure, Amazon Web Services, and many other hosting services.

2.1.3 Does It Use a Lot of Disk Space?

It may sound like packaging up an entire operating system and shipping it to a server use a lot of disk space. Sometimes, in fact, it does. However, containers usually only contain the minimum of what is necessary to run the application, such as a minimal Linux distribution, the application runtime environment, and the application itself. Even further, some Docker applications don't need any of the operating system to run! For some programming environments (such as the Go programming language), the application and all its dependencies can be bundled together in a relatively small package that doesn't rely on any other operating system files.[1]

Many applications use the Alpine Linux distribution, whose base image currently uses less than ten megabytes of space. Most other distributions are more heavyweight, however, often being in the hundred megabyte range. Some container images, which

[1] These are known as "distroless" containers, since they don't contain an actual Linux distribution. There are, additionally, some distroless starter images which contain a few basic files, such as time zone listings and root SSL certificates, which may be needed even if the rest of the Linux distribution isn't needed.