Dalia’s gentle introduction to Docker

The friendly overview of containers and Docker that many Java developers have been looking for

by Dalia Abo Sheasha

July 9, 2021

You’ve certainly heard of containers and Docker by now. It seems like someone is always talking about them—just like my brother-in-law always talking about cryptocurrency and Bitcoin. What are containers? What is Docker? Why would you use them? Well, I’m going to attempt to answer those questions the way I wish someone explained them to me when I first started learning about this technology.

(Would you prefer watching instead of reading? This article goes over the same concepts covered in the first half of my Intro to Docker video.)

The problem

Imagine you’ve written an awesome application in Java. For simplicity’s sake, let’s say the application is running on a laptop, perhaps on Linux, macOS, or Windows. The application uses a database, such as Oracle Database, MySQL, or MongoDB. It also uses an application server, such as Tomcat or Open Liberty, as seen in Figure 1. You’ve finished writing your application and it runs great on your machine.
Now, you need to show the application to your coworker Alex to try out. A simple task, right? Let's see how it plays out.

You: “Hey Alex, I’d like you to try out my application. I’m sending you the application JAR file so you can run it on your machine.”

Alex: “Got it! I’ve already got Java installed. I’ll go ahead and run it.”

You: “Wait, you have to install MongoDB first.”

Alex: “Oh, I’ve never used MongoDB before. I’ll look up how to do that.”

You: “I’d be happy to help you! Let’s start a screen-sharing session.”

<You start a screen-sharing session.>

You (to yourself): Oh no, it’s a Linux machine. Linux… I have no idea how to do this on Linux.

<Several minutes (or hours) and steps later>

Alex: “Looks like MongoDB is installed! Okay, let’s run the application.”

You: “Wait. You have to install an application server too.”

Alex: “An application server?”

Alex (to himself): This had better be the best application I’ve ever seen.

You: “Yeah. Let me show you.”

<Several minutes, curses, and steps later>

Alex: “Okay, the app server is ready. Can I run the application now?”

You: “Yes!”

<The application server starts and shows an error: “The application failed to start.”>

You: “BUT IT WORKS ON MY MACHINE!”

<You and Alex debug for a while.>
You: “Oh, which version of Java are you using?”

Alex: “Java 8.”

You: “I think that’s the problem. I’m using Java records and that capability is available only since Java 16. Can you install Java 16 and try running the application with that?”

Alex: “Java 16? They’re on Java 16 now?”

<Alex goes off to install Java 16.>

Alex: “Okay. Java 16 is installed. Let me try to run the application again.”

<Another failure: Debug, fix, rinse, and repeat.>

Finally, Alex gets the Java application working (see Figure 2) and is amazed at how awesome it is.

![Figure 2](image)

You and Alex can run the Java application on your machines, because each machine contains the same stack.

A couple of hours later, you receive a message from Jordan.

Jordan: “Hey, Alex told me about the new application. I’d love to try it out.”

You (to yourself): Oh no, not again!

All this was to illustrate that if you have Java applications that use any external resources such as a database or application servers, you’ll find it painful to set up the applications on other machines. (That can be equally true for server-based, cloud-based, or hybrid applications.)

Every time you add a new developer to the project, set up a machine for testing, or configure a continuous integration and continuous deployment (CI/CD) pipeline, you could be spending hours or days doing so (see Figure 3). This is one of the many ways Docker and containers can help.
What is a container?

A container is a standard component that lets you package your application and its dependencies in an easy-to-share way. Some of the most important characteristics of a container are that they're portable, isolated, and lightweight (see Figure 4).

The term *container* comes from shipping containers, which became popular in the 1950s and 1960s. With shipping containers, you could transport a car or a bunch of apples without the ship's captain having to worry about the size or shape of its cargo (see Figure 5). As long as you provide the contents in a uniform container, they will be shipped. (These are also called *intermodal containers*, since the same container can be carried by ships, trains, and trucks without unloading or repacking its contents.)
Similarly, a containerized application can be a web application, a mobile application, or an enterprise application. It can be written in any language, such as Java. The application can use Spring Boot, Jakarta EE, or any other framework. The container can include databases, application servers, and other necessary dependencies. As long as you package your application in a container, Alex, Jordan, and everyone else can run the software with minimal work.

**Docker** is a popular open source platform that allows you to use containers (see Figure 6).

Figure 6. Docker’s whale logo looks like a ship filled with standardized containers.

**The solution**

You (wisely) decide to install Docker and containerize your application. You put the application code, application server, and JVM into a single container since they’re all closely tied together. Then, in a separate container, you install the database. (See Figure 7.)

Figure 7. The application now lives in two containers: one for the application, application server, and JVM and the other for the database.

How is this any better? Well, if you had this setup earlier, the conversation with Alex might have gone like this.

**You:** “Hey Alex, I’d like you to try out my application. Since you already have Docker installed, you will need only a couple of simple commands to run it on your machine, and you don’t even need to upgrade from your old Java 8 to the Java 16 JVM that I’m using or figure out how to install MongoDB.”

**Alex:** “Done! Your application looks awesome!”

That sounds easy, but how do you create a container? First, you need an **image**.

**What is an image?**
A helpful analogy for understanding the relationship between images and containers is to consider classes and runtime objects. In Java, if I want to create an `Employee` object, I need to first create an `Employee` class. Once I have the `Employee` class, I can instantiate as many `Employee` objects as I want. Similarly, in Docker, if I want to create a container, I need to first create an image. Once I have the image, I can use it to instantiate as many containers as I’d like.

In other words, a container is the *running instance* of an image.

To create an image, you need a *Dockerfile*, which contains the list of instructions that need to be executed to create an image.

**Virtual machines versus containers**

A common question when developers are learning about containers is “What is the difference between containers and virtual machines?” Well, they’re both trying to solve the same problem, but containers and VMs approach it differently.

Let’s take a look at the VM scenario. If you wanted to run the applications in a virtual machine, the setup would look something like Figure 8, with a machine running a host operating system. That OS has a hypervisor that creates, executes, and manages virtual machines.

![Figure 8](image.png)

*Figure 8.* A machine running a hypervisor, which in turn hosts one or more VMs, each with its own operating system and application stack

When VMs are created, each VM includes its own guest operating system that the application runs on. Unfortunately, having a guest operating system is not cheap resource-wise. Lots of memory and CPU cycles are consumed by each VM’s guest operating system.

Now, let’s compare that to Docker. With Docker containers, you don’t have a guest operating system, making Docker containers much smaller and faster than virtual machines (see Figure 9).
Applications running in containers use fewer resources than those consumed by virtual machines, which lets you run more containers on a given physical machine. The startup time on a container is also much faster, especially since containers don’t need to load a guest operating system.

To be clear, there are pros and cons for containers versus virtual machines. I won’t dive into that here. The main takeaway is that they both solve the “but it works on my machine” problem but approach the solution in different ways. In many instances, however, such as for distributing applications, Docker is often the best solution.

Conclusion

Docker has many advantages over containerless application packaging. One of the main benefits is solving the “but it works on my machine” problem, because the container packages all the dependencies. Now that you understand the fundamentals of Docker, you’re ready to create your first Docker container. You can start up your favorite IDE and follow along with my 20-minute tutorial if you’d like.

Dig deeper

- Dalia’s 20-minute Docker tutorial video
- Getting started with Kubernetes
- GraalVM: Native images in containers

Dalia Abo Sheasha

Dalia Abo Sheasha (@DaliaShea) is a Java Developer Advocate at JetBrains. She has worked as a Java developer for many years developing Java and Jakarta EE applications, with expertise in Java enterprise technology. She has contributed to open source projects including Open Liberty and EclipseLink (JPA).

Figure 9. The stacks required by applications managed by Docker containers (left) and virtual machines (right)