

TIP

Where to start with containers and microservices

Lessons from Java and virtualization show the way to microservices adoption



1

By Pete Johnson

InfoWorld | Oct 6, 2015 4:38 PM PT

Containers get a lot of headlines and it's clear why. For the daily lives of developers, containers are revolutionary. Much in the same way that interpreted languages enable greater productivity than compiled ones, containers save developers precious time by allowing them to create a functional environment in seconds instead of tens of minutes with virtual machines. Reducing that cycle time lets developers spend more time coding and less time waiting for something to happen.

Architecturally, the microservices that containers more easily enable are equally groundbreaking. The ability to break a problem into smaller pieces is always beneficial in unexpected ways, and containers offer a way of doing that on a scale not possible before.

However, in an enterprise IT ecosystem, it's rarely all about developer productivity. Similarly, architectural efficiency, while nice to have, is not always a top priority either. For large IT organizations with lots of brownfield applications and established operational processes, going all in on containers is not as quick and easy as it is for a born-on-the-cloud company with no legacy issues to deal with.

That doesn't mean that containers don't have a place in modernizing a huge IT shop. In fact, enterprise IT has seen this movie at least twice before, with two technologies that caused tectonic shifts, and found that mixed architectures were the way to bring in significant change at a pace that reduced risk. Those technologies were Java and virtualization.

Waiting for Java

Enterprise IT is typically focused on cost savings, and as such it's loath to change. If you were a developer in a big IT shop in the early '90s, the dominant language was C++. Client-server was slowly replacing the monolithic application to better take advantage of networking, and the big challenge of the day was writing your code in such a way that it could run on any flavor of Unix. That last

task wasn't easy because the base operating system varied among HP-UX, AIX, and other Unix variants with myriad different libraries. It was common to have elaborate branching in make files and header files so that the code would compile correctly for each target operating system.

Enter Java. This technology removed from developers the responsibility of understanding the complexity of each operating system and instead put that complexity in the Java virtual machine. Developers compiled their code into Java byte code, which got interpreted by JVMs written to translate those commands into OS-specific library calls.

It's hard to overstate how revolutionary that idea was or the impact it would have on developer productivity. The problem was, this was an era of bare-metal servers, and operations pros scoffed at the idea of introducing a layer of abstraction at runtime simply to improve the lives of developers. An application would live much more of its life in production than it would in development, so why cater to developer productivity? Eventually, Java would free developers to look at larger issues, including the services-oriented architecture improvements Java promised to bring, but not yet.

Countdown to virtualization

Amazingly, IT had a very similar argument when virtualization became available around 10 years after Java. Before VMs, applications ran on bare-metal servers that were sized to handle estimated peaks that often didn't materialize. However, there were bigger penalties imposed on the IT staff for guessing wrong on the low side of capacity needs (like getting fired) than there were for guessing high (hardware sat around underutilized).

Then IT budgets got tight after the dot-com-bubble years, and IT management noticed all these bare-metal servers sitting around running at 25 percent capacity or typically much less. "Why not run multiple applications on the same physical hardware so we can save costs?" they would ask.

Well, when Application A is running on the same bare-metal server as Application B and experiences a memory leak, suddenly Application B suffers performance issues -- not because it was doing anything wrong but because it was colocated with a noisy neighbor. If only there were a technology that allowed applications to share the same hardware, but with some sort of boundary that could reduce the noisy-neighbor phenomenon, then utilization could be improved.


Enter virtualization. This technology solved this problem, but like Java before it, it introduced a level of abstraction between the application and the hardware that made operations pros wince. Like Java, virtualization helped developer productivity when a VM could be made available in minutes instead of waiting weeks or months for new physical hardware. Like Java, virtualization

was initially a hard sell despite obvious benefits -- in this case, the ability to automate the creation of VMs to replicate environments or create more capacity to meet unexpected changes in demand.

A paradigm for containers

How did Java and virtualization eventually break through? The adoption problem for both was solved through mixed architectures. As a compromise, it was common for ops pros to allow a Web tier to run entirely in Java but have the physical load balancers under their control and run databases on bare metal hardware. Eventually, after understanding the huge flexibility that a Java-based model gave even them, ops pros relented on other layers of the architecture, and service-oriented architecture became a reality in most shops.

When virtualization came along, the same mixed architecture pattern emerged. First, development and test workloads became virtualized. Then, like Java before it, virtualization became attractive for Web tiers because users could rapidly add resources to what was typically the bottleneck at times of high demand. That solved operations' pressing problem on the Web layer but still allowed those same databases and load balancers to stay protected.

 Microservices means different things to different ops pros. A good definition comes from noted software development evangelist [Martin Fowler](#):

[Sign In](#) | [Register](#)

JAWORLD

Microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource. The services are built around business capabilities and independently deployable by fully automated deployment machinery.

The business capabilities and automation that microservices unlock represent the real payoff for containers, which are the vehicle for those independently deployable pieces. While the short-term productivity gains containers bring individual developers are nice, the long-term productivity improvements they promise to organizations are huge. Containers enable groups of teams to more rapidly create iterations to business problems by piecing together a set of smaller services.


In conclusion, the mixed architecture approach gets everyone involved with containers at a pace they are comfortable with. This paradigm has proven successful at least twice in the past: first with Java, then with virtualization. In mixed architectures, abstraction between the physical hardware and the applications gives us enormous flexibility, so there is no need to rehash old arguments about developer productivity priorities relative to other concerns or to fret over performance penalties. Mixed architectures allow an organization set its own pace -- and set itself up for the big microservices payoff sooner rather than later.

Pete Johnson is senior director of product evangelism at CliQr.

This story, "Where to start with containers and microservices" was originally published by [InfoWorld](#).



Follow everything from JavaWorld

 View 1 Comment

Copyright © 1994 - 2016 JavaWorld, Inc. All rights reserved.