Microservices in action, Part 2: Containers and microservices — a perfect pair

Why smaller, faster application components can be delivered more easily than ever

Rick E. Osowski
Technical Product Manager
IBM

13 November 2015

Discover how Linux containers are revolutionizing software development and powering microservices to shift an entire industry. Know the requirements that are critical to success in microservices adoption and how container-based infrastructures make it easier to meet those requirements.

View more content in this series

In Part 1 of this series, I talked about what exactly microservices are and how they differ from traditionally built systems (monoliths). This second installment is about the power of Linux containers — how they are revolutionizing software development and powering microservices to shift an entire industry. I'll touch on three concepts that are critical for you to focus on when adopting container-based infrastructure for your microservice-based applications:

- Logging and monitoring
- Zero-downtime continuous delivery
- Dynamic service registries

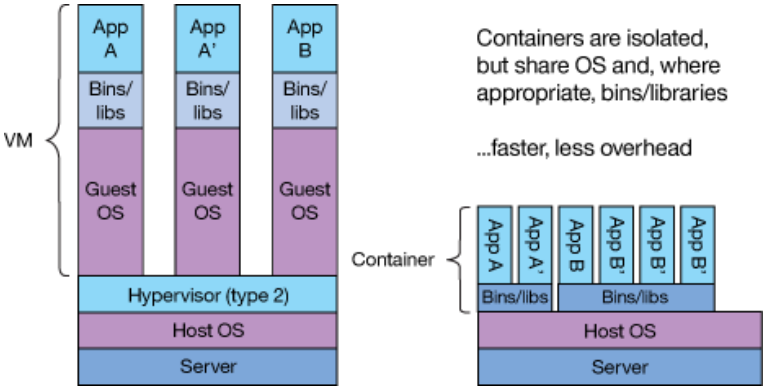
I'll start with an overview of containers, container managers, and how containers relate to microservices.

Containers and microservices: A perfect pair

Unless you're completely new to cloud technology and cloud-native application development, you've probably heard of Linux containers and the container-based projects that have caught fire over the past couple of years. But in case you haven't, think of Linux containers as lightweight virtual machines that can be used more flexibly, integrated more rapidly, and distributed much more easily. One of the projects leading this charge is Docker. Since its launch in 2012, the Docker

team (and now company) has provided a dead-simple way to build, package, and distribute cloud-native applications via Linux containers.

How do containers differ from virtual machines? Each virtual machine (illustrated on the left side of the following image) runs its own guest operating system instance and contains its own libraries and binaries. Containers (shown on the right) are isolated, sharing the underlying host OS and libraries, while packaging only the necessary application binaries.



> *“ Many industry leaders are moving to container-based infrastructure, both in the cloud and on-premises, for extreme gains. ”*

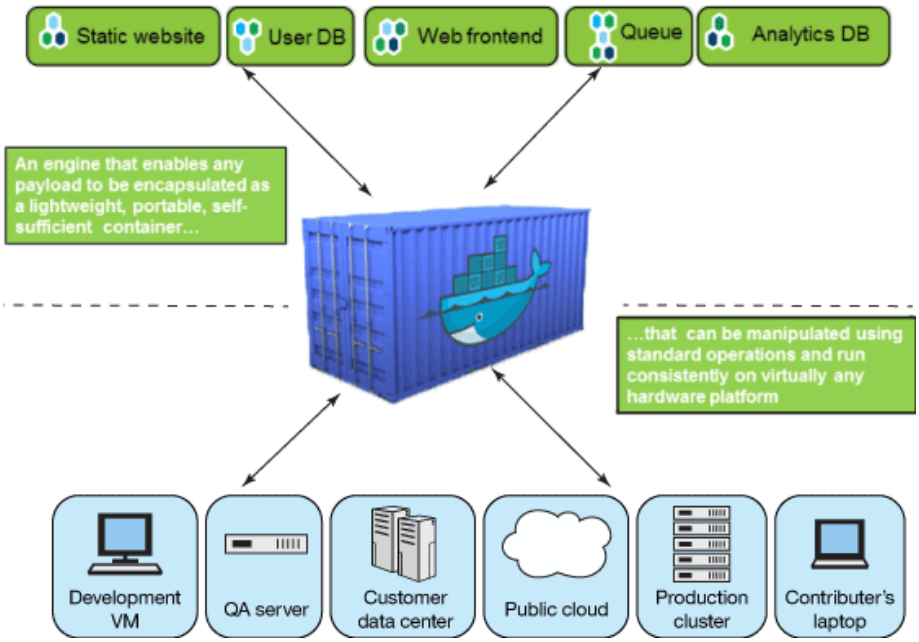
Containers run as a minimal set of resources on top of Linux systems, and often the packaged application is no more than a few hundred megabytes. Virtual machine-based applications are often at least three to four orders of magnitude larger (tens of gigabytes). You can easily see how containers fit into the microservice paradigm, being **smaller** and **faster**— two of the microservice tenets from Part 1.

Many industry leaders are moving to container-based infrastructure, both in the cloud and on-premises, for extreme gains. One of the key gains is that Docker and other similar Linux container technologies are easy to integrate into continuous-integration and continuous-delivery pipelines: On average, Docker users ship software seven times more frequently, according to a recent self-funded Docker study. Companies like Gilt Groupe have embraced microservices and containerized infrastructure, shipping software sometimes as often as 100 times a day. The ability to push code changes quickly, automatically rebuild Docker images that are of minimal size, and manage a large number of these deployed images from a common code base results in impressive speed through a company's delivery pipeline.

One of the other benefits of Docker containers is the portability of these packaged applications, called *Docker images.* Docker images can be moved seamlessly across environments and through build pipelines. For example, BBC News (a division of the British Broadcasting Corp.) says that its continuous-integration jobs run more than 60 percent faster in a Docker-based infrastructure. The ability to move the same code throughout the delivery pipeline — minimizing the need for software configuration at each stage and having predictable hardware resource requirements

along the way — speeds applications through development, test, and production faster than ever before. Companies are able to see these gains in efficiency because their system components are modularized inside each Docker image. You don't need to configure the software each time you need it. You simply start a container instance, and it's ready to go.

Docker is a shipping container system for code that makes software development and delivery through Linux containers easy. Docker acts as an engine that enables any payload to be encapsulated as a lightweight, portable self-sufficient container. Such containers can be manipulated using standard operations and run consistently on virtually any hardware platform.



If you're new to containers and Docker, see Resources for links to great introductory material on Docker and Linux containers in general. If you're experienced with Docker and want to get hands-on with Docker in the cloud, IBM Containers for Bluemix is an enterprise-grade container service that you can get started on for free today. You get your own private registry to store all your images, access to the IBM public registry of supported middleware, hosted delivery-pipeline integration, and access to more than 150 Bluemix™ services. You can have your application running in Docker containers faster than ever.

Faster and smaller: Containers as the nanobots of software development

As you can begin to see why containers are so important to microservices — as one of the key enabling technologies for the architectural style — you can also begin to see that management of containers is equally important. As you know from Part 1, instead of scaling up, we scale out in microservices. Instead of adding more RAM to a microservice runtime, we simply get another microservice runtime of the same kind. Need even more RAM? Get a third instance. This approach is fine for only a couple services with one container instance each, but as anyone with computer skills and an extended family knows, it can get out of hand quickly when you're remotely managing dozens of servers.

Think about how quickly you will need to manage more than 100 individual instances. If you start out with a handful of microservices that make up your app — five or six, say — each of those should have at least three container instances supporting each microservice. So right off the bat, you're at 18 container instances. Say you add another microservice or your app is really successful and you need to scale to five to 10 container instances for certain services. You're easily approaching more than 100 container instances to manage — on a good day.

Thankfully, many open source projects can handle this exact need. For example, Kubernetes, Apache Mesos, and fleet make it easy to manage thousands of container instances from a single console or command line, using an infrastructure-based domain-specific language.

This management concept tightly reinforces the cattle-compared-to-pets concept from Part 1. The idea is that all of our containers that we deploy through our continuous-integration/continuous-delivery processes are immutable. Once they are deployed, you can't change them. Instead, if you need a change or an update, spin up a new cluster of containers with the correct updates applied and tear down the old ones. It's much easier to manage 1,000 cattle than 1,000 pets. Don't get me wrong: We all love our pets, but if you want to innovate at speed and deliver value to your customers, with revenue to your business, you can't afford to spend all the time necessary to give each pet all the tender loving care it needs. Projects like Kubernetes, Apache Mesos, and fleet—and hosted container services like IBM Containers — make it possible to integrate your delivery pipelines and image registries to quickly and easily manage all phases of your infrastructure as hyperefficient cattle.

As an aside, it should be noted that although certain container services still use virtual machines as Docker hosts, these should still be regarded as cattle. These VMs are a bit more robust in resources and integrated management capabilities, but they are still generally thought of as cattle since they are dynamically managed by the needs of the container-based workloads.

A microservices fabric: Your best excuse against playing Whack-A-Mole

So far, I've talked about why more containers is better and how to handle generic infrastructure at scale. Now that you're comfortable with the concept of developing for containers and saying goodbye to your pets, you need to start thinking about developing your applications and putting them into production.

This brings me to the three key elements that are crucial to microservice-based application development:

- Logging and monitoring
- Zero-downtime continuous delivery
- Dynamic service registries

You want to think about each of these capabilities from day one, but without necessarily solving for them immediately.

As I discuss these capabilities, you'll also see why an integrated microservices fabric — such as Bluemix and its relevant services — makes the management of your microservices architectures that much easier.

Logging and monitoring

If you provide production-level support for applications and services, your first question should always be, "What do I do when something goes wrong?" Notice that there's not even a hint of an "if" in that question. Components will fail, versions will change, third-party services will have outages. How can you maintain a level of sanity, along with a desired level of uptime for your users? That's question one.

As I said earlier, you want your containers to be immutable. For this reason, most IT organizations don't provide system-level access to container instances — no SSH, no console, no nothing. So how are you supposed to know what's going on inside a black box that you can't change? That's question two.

Thankfully, this question has been solved with the concept of an *ELK stack*—Elasticsearch, LogStash, and Kibana.

These three separate components provide the ability to aggregate logs, search free-form through aggregated logs, and create and share dashboards based on logs and monitored activity across the platform. This is a great capability — much better than logging into individual computers and running a sysadmin toolbox of `sed`, `grep`, and `awk`. You have full-featured access to a central repository of all of your logs. You can correlate events across systems and microservices because you'll usually see events and IDs traveling through your system and encountering similar issues.

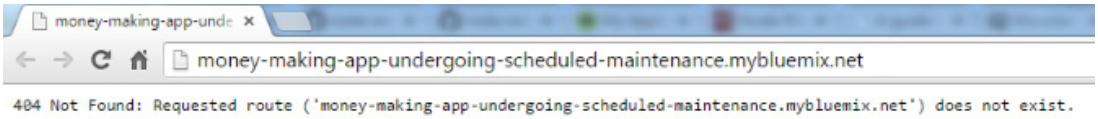
You can integrate with ELK stacks in many ways, whether you're running in the cloud or on-premises: via hosted services, open source variants, and often options built into platform-as-a-service offerings —IBM Containers, for example. Inside of the container runtime on Bluemix, you have access to a full-featured, multitenant ELK stack that automatically receives your logs from Docker container-based runtimes, giving you visibility and searchability into those runtime events while also giving you a preconfigured Kibana-based dashboard out of the box. If you're looking to get started with containers, this is one key capability that makes IBM Containers a preeminent choice for your container-based microservices.

Zero-downtime continuous delivery

Now that you're comfortable that you'll have an idea of what to do when the sky starts falling (of course you hope it never does), you can move on to rapidly deploying all of your amazing application updates. Thinking about some of the larger companies that are deploying applications dozens, hundreds, or thousands of times a week, you have to wonder about downtime. Surely

those companies aren't having application outages every time they push a new version — if they did, they'd never be up.

Those companies have come to master *zero-downtime deployment.* This can be known as many things, often with colorful variants, ranging from Red-Black to Blue-Green and so on, but they all fall back to the same principle: Your application is always available, no matter what, even through abundant updates, because website outages caused by planned downtime aren't good for you or your users.



IBM Cloud Services: Active Deploy

Learn more about zero-downtime deployment with this demonstration of Active Deploy, a new IBM Cloud service. IBM Fellow Jason McGee talks with Daniel Berg, IBM Distinguished Engineer, Cloud Foundation Services and DevOps, covering the ins and outs of what you need to do to achieve zero downtime in your next rolling deployment or upgrade. Active Deploy is available on Bluemix today.

To view this video **Active Deploy | IBM Cloud Services: Video Demo Series** please access the online version of the article.

Now with the monoliths described in Part 1, avoiding planned downtime was dangerous, time-consuming, expensive, and exhausting for everyone involved. Needing multiple mirror images of a gargantuan set of resources was pretty unmanageable and led to late nights, in a supposed "off-hours" window when a lot of sweating was done while bringing the new version up and the old version down — let alone figuring out what to do with the old version if a rollback was needed or something went wrong outside the control of the immediately responsible team.

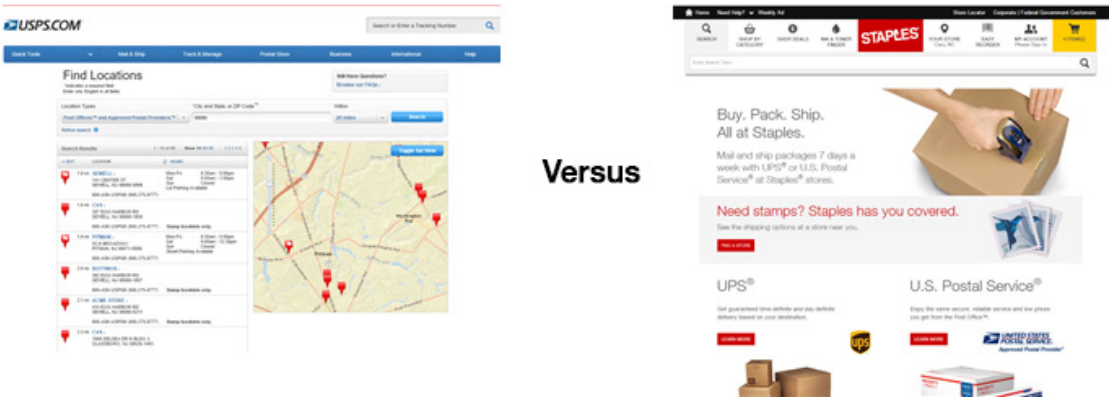
With microservices and container-based applications, these worries are minimized, if not completely removed, especially with some key services available on Bluemix today. By breaking your components down into much smaller pieces, you can deploy them with minimal impact to the overall system, while keeping more of them up at certain times to prevent outages. A new service available on Bluemix, Active Deploy, provides this capability and is preintegrated into the Delivery Pipeline. Smaller teams can manage more applications more efficiently because each version deployed has automatic oversight of maintaining its uptime, which costs less in both human attention and compute resources.

Dynamic service registries

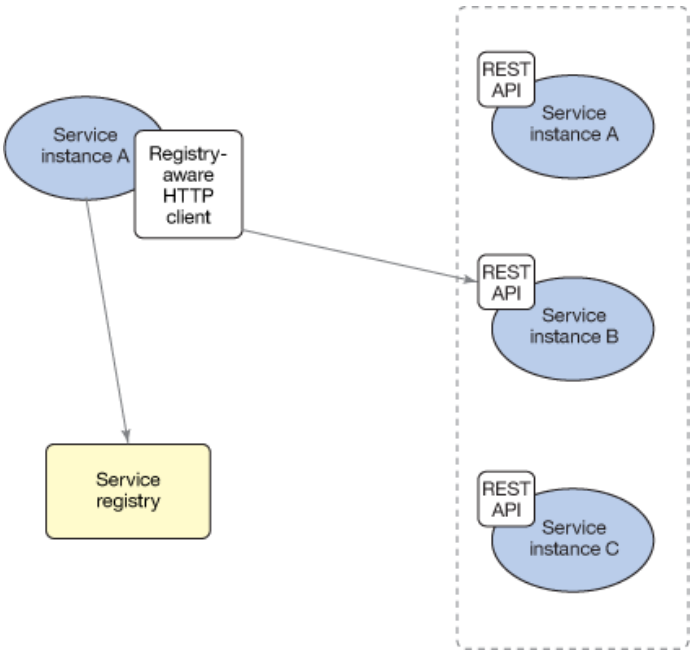
The last topic for this installment is the notion of dynamic service registries. This topic includes concepts that you might already know of as *service discovery* or *service proxy.* These two concepts are not the same by any means, but they're close enough to cover under this one approach.

Now that you'll be creating thousands of container instances that back your microservices across all of your applications, how will the other components in your application understand what's going on? How will they know what other microservices they have available to make service calls to? How will they respond to service calls being made to them?

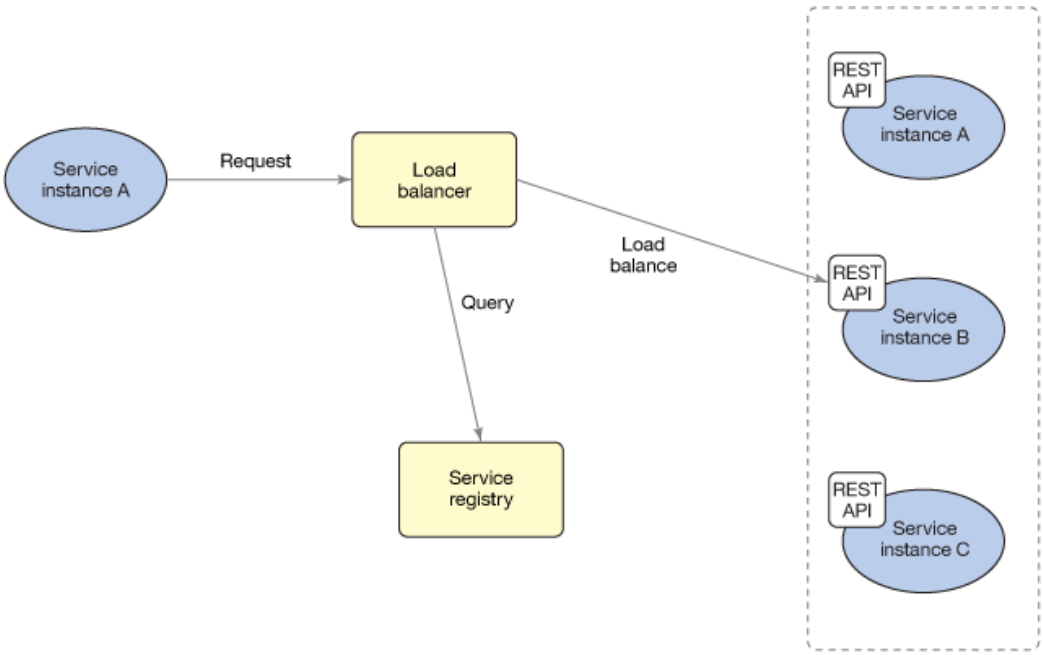
The difference between service discovery and service proxy comes down to whether you want to do the search yourself or let someone else handle it for you. As an analogy, suppose you need a shipping service. Do you want look up services from a single provider (the US Postal Service (USPS), for example) or from a multiservice clearing house (such as Staples).



For instance, if I want to ship a package, I can go to the USPS website, enter a couple parameters for the kind of post office I am looking for, get back a list of options, pick one, and go there to ship my package. This is the concept of service discovery: I use a well-known registry of available service endpoints that's updated when services come online and go offline. Through REST APIs, calling applications can query service-discovery services to determine which and how many types of services are available for a specific service call, down to a specific version of that requested service.



If I don't want to be the one to choose where I go to ship my package — I just want to say, "Get this package to what it says on the address label" — I can take it to a Staples. Staples then chooses the most cost-effective shipping option for my package based on all the information I've provided. I'm taking the package to a well-known service provider and letting it handle the routing of my package for me. This is the concept of service proxying: You make a call to a well-known endpoint, and that call is automatically forwarded, based on preestablished rules or metadata, to a backing service that provides the actual response to the service request.



There are good arguments for preferring service discovery over service proxy and vice versa, but it really comes down to preference or implementation experience/requirements. Several open source

service-discovery offerings, such as etcd and Consul, provide distributed service registries with which you can register, tag, and heartbeat all of your available service instances. There are even cool projects like Registrator that will automatically register your services to one of these endpoints as soon as the Docker container is created.

Some of the more popular service-proxy projects support the key argument that, in the long run, the service-proxy method requires fewer network hops to get to your eventual backing service. One of the largest and most popular service-proxy projects is the Netflix Hystrix project. Hystrix is a library based on Java™ technology that goes above and beyond simple service proxying, but provides a number of quality-of-service improvements in a service-proxy library.

Obviously, both of these patterns are important in automagically managing your service instances and making them part of your available microservices infrastructure. Bluemix will have its own multitenant service-discovery offering soon (if it doesn't already have it by the time you are reading this) — removing the need for you to manage the service-discovery service, which can be a handful. Imagine registering all your container instances automatically, whether you're spinning them up manually, pushing them through the Bluemix Delivery Pipeline, or integrating with an on-premises pipeline like IBM UrbanCode Deploy or Jenkins.

Conclusion

You've seen here how containers are accelerating the push to cloud-native application development. Smaller, faster application components can be delivered faster and more easily than ever, with more built-in management capability than ever. Deploying containers on a managed service, such as IBM Containers, along with all the supporting microservice capabilities — including logging and monitoring, Active Deploy, and service discovery — make it easy to take that next step away from your existing monoliths and into the cloud.

We are moving toward smarter, modular systems that are more automated and more integrated from start to finish. These will evolve at their own pace, self-aware of what is available in the infrastructure, make it known what is not, and make it known when something needs to change. All of these capabilities and more will be covered in some of the upcoming series installments in this series.

Resources

- Making Logs Awesome — Elasticsearch in the Cloud Using Docker: See how to easily manage your own ELK stack on Bluemix using IBM Containers.
- Zero Downtime, Instant Deployment and Rollback: Read about zero downtime deployment on eBay's Tech Blog.
- Implement a microservice-based architecture in Bluemix: Learn how to implement microservice-based applications using Cloud Foundry apps and using IBM Containers, in the IBM Bluemix platform.
- Bluemix Active Deploy — Zero-Downtime Deployment: Use Active Deploy on Bluemix to make system-wide outages a thing of the past.
- Service Discovery in a Microservices Architecture: Learn more about service discovery on the NGINX blog.
- Gilt Tech Blog: Get more information on how Gilt Groupe implements microservices and containers.
- Cloud Foundation Services: Follow Cloud Foundation Services on Twitter.

About the author

Rick E. Osowski

Rick Osowski is a technical product manager for IBM Cloud. With over 10 years of deep technical experience in IBM's middleware and business process management capabilities, Rick now focuses on next-generation cloud platforms, delivering hybrid cloud solutions to the company's expanding client base. He has a bachelor's degree in computer science from Pennsylvania State University and considers himself hopelessly addicted to architecting, scripting, and automating in work and everyday life. When not strategically placed in front of a computer monitor, Rick enjoys playing hockey, listening to music, and traveling.