Containerization

Jeff Chapman

DMI Tech User Group Conference
October 2015

This document is for informational purposes only and is subject to change at any time without notice. The information in this document is proprietary to Actian and no part of this document may be reproduced, copied, or transmitted in any form or for any purpose without the express prior written permission of Actian.

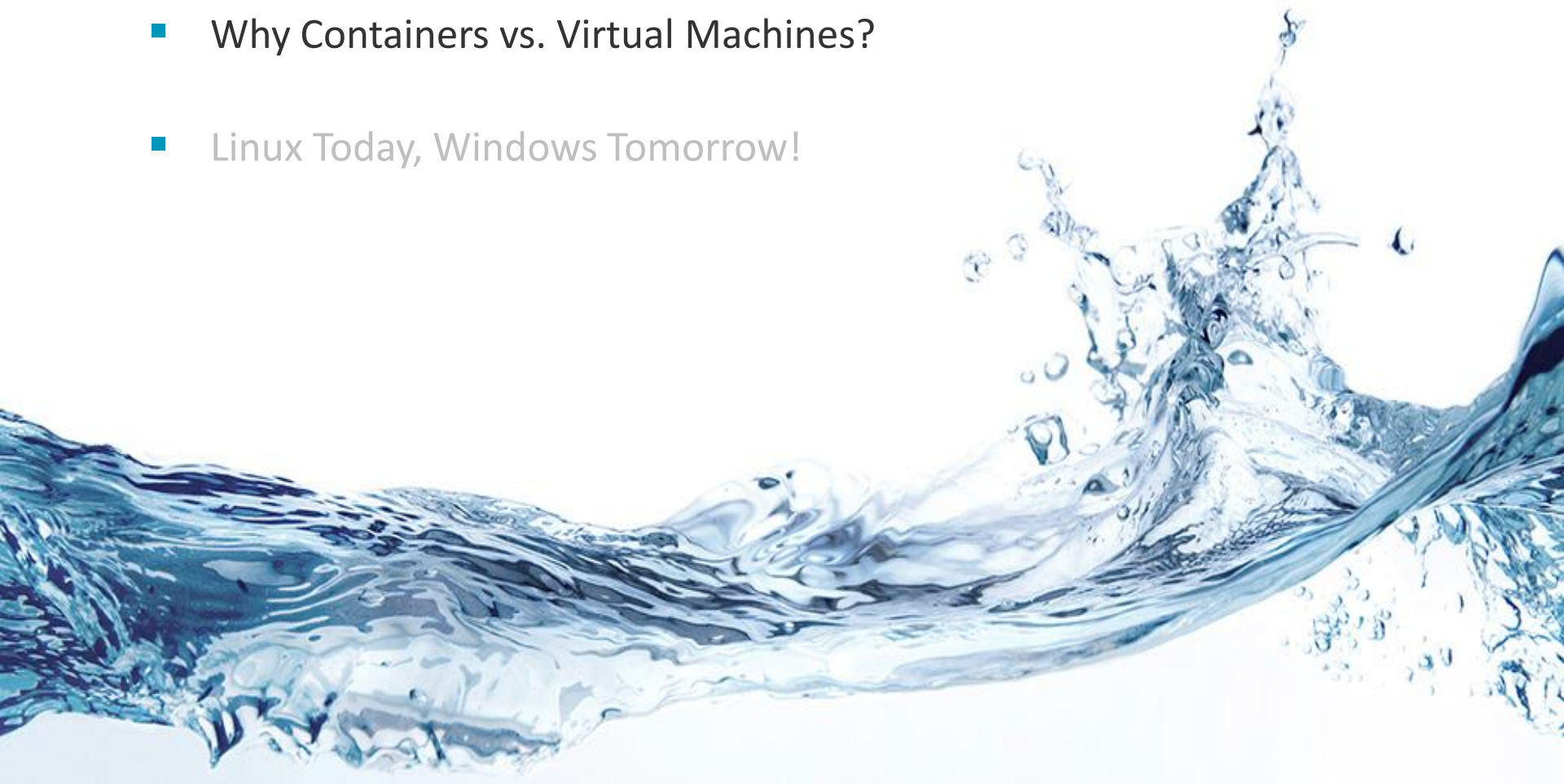
This document is not intended to be binding upon Actian to any particular course of business, pricing, product strategy, and/or development. Actian assumes no responsibility for errors or omissions in this document. Actian shall have no liability for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials. Actian does not warrant the accuracy or completeness of the information, text, graphics, links, or other items contained within this material. This document is provided without a warranty of any kind, either express or implied, including but not limited to the implied warranties of merchantability, fitness for a particular purpose, or non-infringement.

Session Outline

- Containerization
  - Why containers vs. virtual machines?
  - Linux Today, Windows Tomorrow!

Containerization – Thinking Out Loud

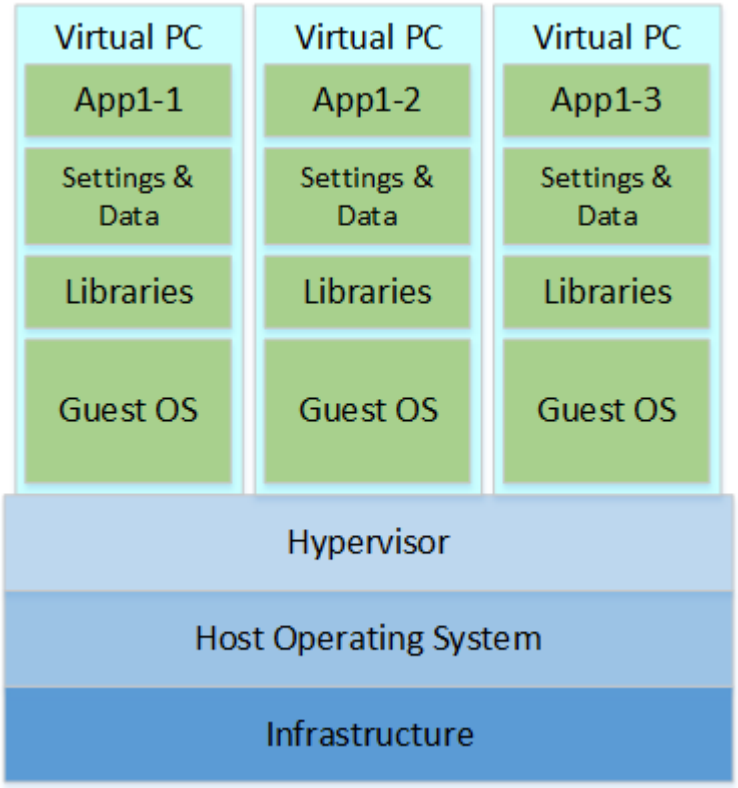
- Why Containers vs. Virtual Machines?
- Linux Today, Windows Tomorrow!



Containerization – Overview of Docker – Why?

Traditional hypervisors carry a lot of overhead for each App instance

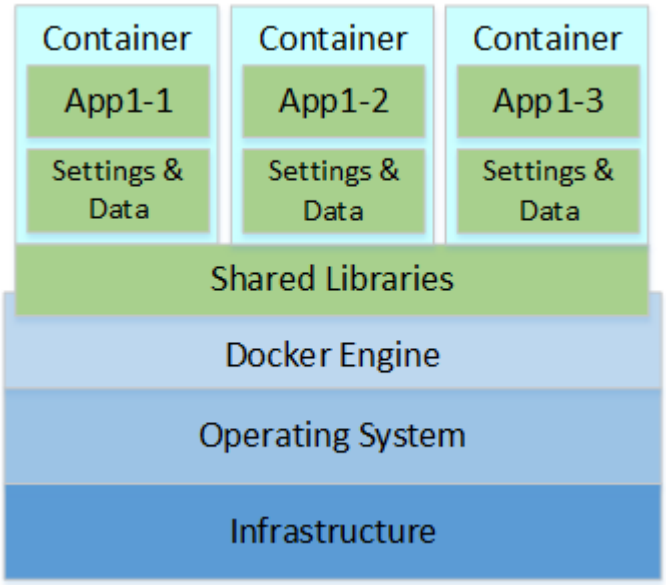
- Each VM has a separate copy of Guest OS, Libraries and the App
- Hypervisors try to optimize, but will never be as thin as Containers



With VMs, you must wait for the Guest OS to boot before you can launch your app.

The OS under your containers is already running so your apps are set to run.

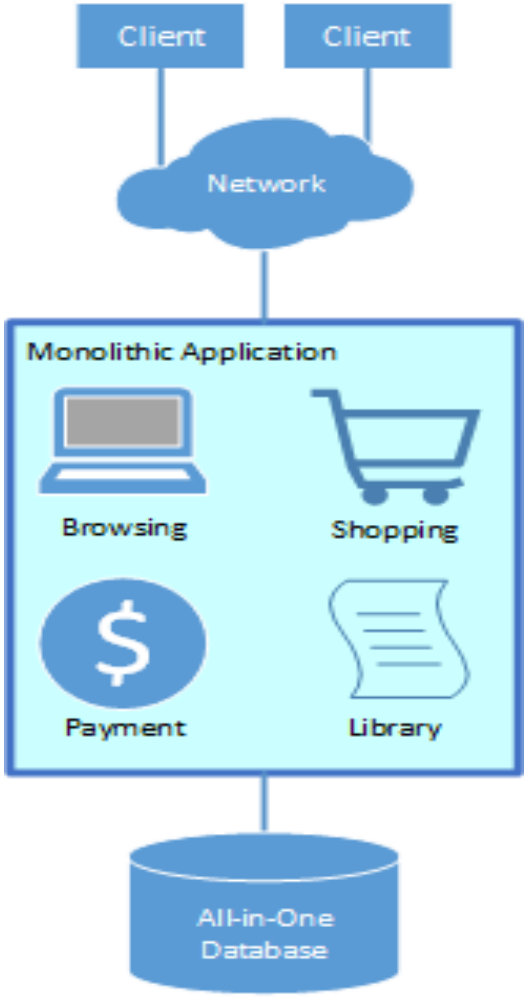
Containers inherently maximize sharing



Based on diagrams from https://www.docker.com/whatisdocker

Containerization – Overview of Docker – Traditional App

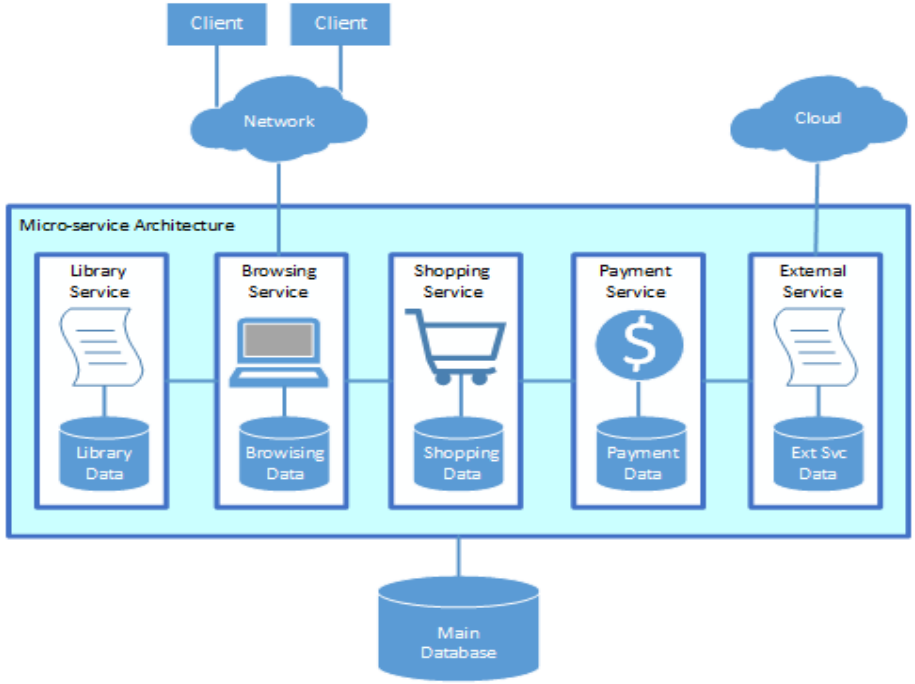
Traditional apps are written and delivered as Monolithic Apps



- All or nothing upgrades require full shutdown, and a leap of faith
- Global and local data is usually tightly entangled and increasingly hard to manage as data size and feature sets grow
- Schemas eventually become very fragile
- Very expensive to develop, test, deploy, support and replace

Containerization – Overview of Docker – Typical MSA

Same Monolithic App reimagined with a Micro-Service Architecture

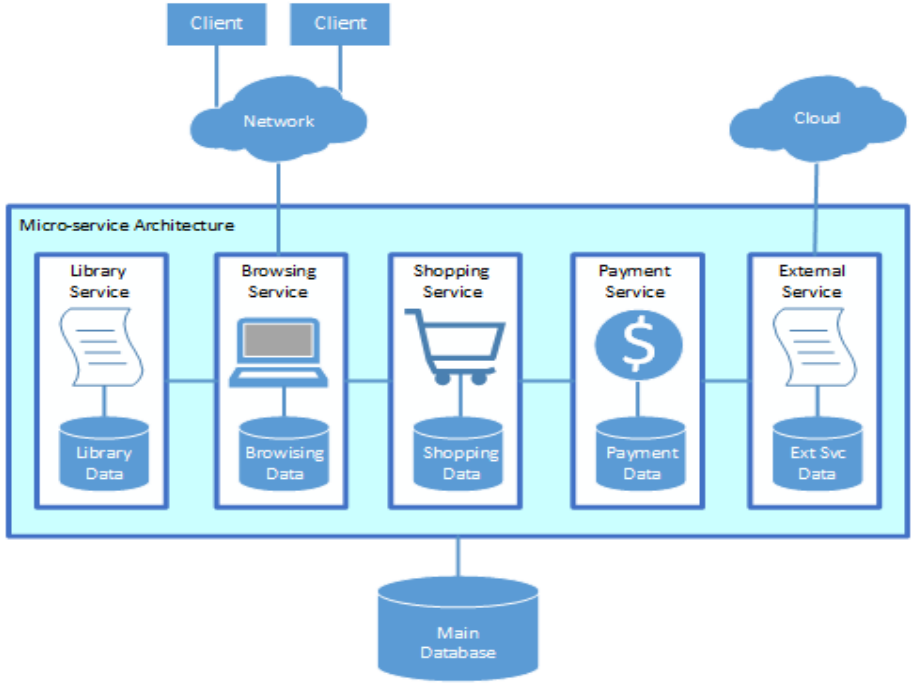


- Separation of concerns simplifies development and test
- Loosely coupled for flexibility
- Local data is kept local, usually in 'data volumes' passed between services
- Global data is kept global
- Local schemas can change without affecting or changing non-interested services or global data

- Services are isolated except the few places where it's truly needed
  - Usually via explicitly shared volumes and ports, secured with certificates and keys
- Services are developed, tested and rolled-out individually
  - Still have to do integration and system testing, but can be much more focused on changes

Containerization – Overview of Docker – Typical MSA

Same Monolithic App reimagined with a Micro-Service Architecture



Individual services are easily added or updated without stopping the entire app

- Simply add the updated service
- New service lives 'beside' the old
- Already connected users are unaffected—they continue to use the old service instance
- New connections are served by the updated service—new users can use new feature immediately
- When the last old instance goes away, the old service is deleted

In the above diagram, the payment service was augmented with an external billing service

- Only the payment service needed to be updated when the new external service was added
- All other services keep on working, unaffected and unchanged

Containerization – Overview of Docker - Security

- Container only shares ports/data with specified containers
  - Containers can share volumes and ports with the host and with each other
  - Shared ports/volumes must be explicitly defined and consumed
  - Shared ports can be easily and flexibly secured with cloud services like nginx
- CoreOS only accepts OpenSSL certificates for credentials
  - Uses the Config-Drive standard from OpenStack
    - Certificates can be pre-generated or on-the-fly for each customer, tenant or service
    - When launching the container, just the certificates required by this one activity/instance get copied into the Config-Drive and attached to the container
    - Container wakes with a pre-populated Config-Drive full of exactly the right certificates needed to talk only to the designated partners for just this particular activity/instance
- Short container lifecycles mean updates/fixes propagate quickly
  - Can respond nearly immediately to issues like the recent OpenSSL security flaw
  - CoreOS updates the kernel while it's running by using two memory images of the currently running kernel, the old one and the new one, and at exactly the right time flips execution to the fixed kernel and discards the old one

Containerization – Overview of Docker – Pets or Parts

Monolithic Applications become Pets

> We give them cute names. We hand raise them and nurse them back to health when sick. We grieve when they pass. Replacing the lost pet is never trivial.

Micro-service Components are simply Parts in a Machine

> We know them by what they do. If they have names, they're derived from an instance number, like SC0042. A part for a given purpose is interchangeable with other parts made for that purpose. When parts malfunction, we quickly replace them with little concern or effort.

Monolithic apps will continue to have a role to play but pressures to accommodate cloud and mobile scenarios will force an evolution toward a more nimble architecture.

Gavin McCance of CERN recently wrote that Micro-Service Architectures using fast, lightweight, disposable containers have emerged as the most flexible and scalable architecture for mobile back-ends and cloud native applications.

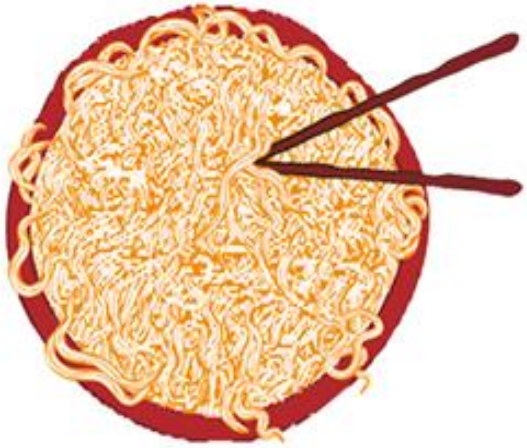
Borrowed from Gavin McCance, CERN, http://www.slideshare.net/gmccance/cern-data-centre-evolution

Containerization – Overview of Docker - Development

The Dec 2014 Tech Forecast from Price Waterhouse and Coopers offers this perspective:

1990s and earlier

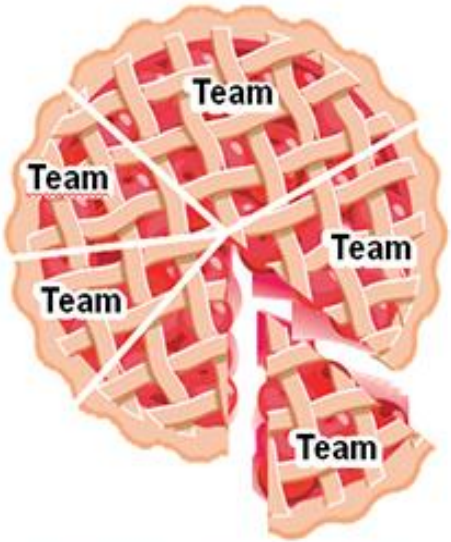
Pre-SOA (monolithic)
Tight coupling



For a monolith to change, all must agree on each change. Each change has unanticipated effects requiring careful testing beforehand.

2000s

Traditional SOA
Looser coupling



Elements in SOA are developed more autonomously but must be coordinated with others to fit into the overall design.

2010s

Microservices
Decoupled

Developers can create and activate new microservices without prior coordination with others. Their adherence to MSA principles makes continuous delivery of new or modified services possible.

http://www.pwc.com/us/en/technology-forecast/2014/cloud-computing/features/microservices.html

Containerization – Overview of Docker – No Free Lunch

Benjamin Wootton, CTO of Contino, offered this excellent summary of the pains:

- **Significant Operations Overhead** – More work needed for clustering, failover and resilience, load balancers, messaging layers, high quality monitoring and operations infrastructure
- **Distributed System Complexity** – Network latency, faulty hardware, message serialization, unreliable networks, asynchronicity, versioning, varying loads within our application tiers etc
- **Substantial DevOps Skills Required** - You simply can't throw applications built in this style over the wall to an operations team. They will need training and support until on their feet.
- **Implicit Interfaces** - Cross cutting behaviors may require changes to different components
- **Duplication Of Effort** - Functionality too small to be a 'service' winds up as duplicated code (more trouble to maintain) or shared libraries (goes against the 'loosely-coupled' mantra)
- **Full Asynchronicity Is Difficult** - When things have to happen synchronously or transactionally in an inherently Asynchronous architecture, things get complex quickly since we may need to manage things like correlation IDs and distributed transactions
- **Testability Challenges** - Add in asynchronicity and dynamic message loads and it becomes much harder to test systems built in this style and gain confidence in the set of services that we are about to release into production. Testability is as important as any aspect of the feature.

Source: http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html

Containerization – Overview of Docker – Perks and Pains

Benjamin Wootton, CTO of Contino, continued:

- Micro-service architectures have lots of very real and significant benefits:
  - The services themselves are simple, focusing on doing one thing well
  - Each service can be built using the best and most appropriate tool for the job
  - Systems built in this way are inherently loosely coupled
  - Multiple developers/teams deliver relatively independently of each other under this model
  - They are a great enabler for continuous delivery, allowing frequent releases whilst keeping the rest of the system available and stable
- Integration happens via data-driven activities rather than tightly coupled APIs
  - By keeping micro-services specific and separate, there's little to integrate
  - You typically deal with a handful of data
    - Rather than work through a complex API, a service generates the specific data you want in a RESTful way, writes it to the data volume, then passes that shared data volume directly to the next service
  - If you have to integrate anything other than shared data volumes, your coupling is too tight
  - Use late binding with shared data volumes to reduce dependencies

http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html

Containerization – Overview of Docker – Real Benefits

The Dec 2014 Tech Forecast from Price Waterhouse and Coopers offered these points:

- Mobile apps and web apps are natural venues for Micro-Service Architectures (MSA)
  - Fast is more important than elegant
  - Change in the application's functionality and usage is frequent
  - Change occurs at different rates within the application, so functional isolation and simple integration are more important than module cohesiveness
  - Functionality is easily separated into simple, isolatable components
- Goal is to separate concerns along these dimensions:
  - Each micro-service preforms one business activity
  - A small set of data and UI elements is involved, only what the business activity needs
  - One developer, or a small team, independently produces a micro-service with its own build to avoid trunk conflict
  - The business logic is stateless
  - The data access layer is statefully cached (e.g., with a container volume that can be passed between services, like when a shopping service hands a data volume to a payment service)
  - New functions are added swiftly, while old ones can be retired slowly

http://www.pwc.com/us/en/technology-forecast/2014/cloud-computing/features/microservices.html

Containerization – Thinking Out Loud

- Why Containers vs. Virtual Machines?
- Linux Today, Windows Tomorrow!

Containerization – Overview of Docker – Windows

- Nano Server image is 200MB
  - Bare minimum of things you take for granted
  - In comparison:
    - Ubuntu/CentOS base images are ~130MB
    - CoreOS base image is ~30MB
- Full support for usual features like PowerShell, C# and .Net
- Most automatable Windows apps can be Dockerized
- Some Windows apps make assumptions about interaction with the environment which a container might not be able to easily satisfy. For example…
  - Does your app assume the start menu or task bar exists? Might not work
  - Can we separate the PSQL part of the registry so it can live in a private volume?
- There's a lot to learn here…

Containerization – Overview of Docker – Windows?

- Microsoft recently released a Docker Client and tools so you can manage Docker containers from Windows
  - Until recently, this Microsoft release was only useful for Linux containers.
  - With Windows Server 2016 Tech Preview 3, Microsoft delivered a preview of Windows applications running in Docker-compatible Windows Containers on the new Nano Server
  - Can run in the same cluster as other Docker containers running on Linux
- Two types of container engines for the Nano Server
  - Windows Containers – Each container runs on the underlying Nano Server kernel, much like Linux containers
    - TP3 provides a preview of Windows Containers
  - Hyper-V containers – Each container runs in a separate/isolated Hyper-V VM where the VM is derived from the underlying Nano Server kernel
    - Preview coming soon

Containerization – Overview of Docker – Windows

Janakiram MSV offered these observations for Tech Republic:

- Windows Server Containers
  - Shares the underlying OS kernel so you get very fast application startup and maximum resource sharing, but offers less security due to the way isolation is implemented.
  - "According to Mark Russinovich, these containers are best for homogenous applications that don't require strong isolation and security constraints. Large microservices applications composed of multiple containers can use Windows Server Containers for performance and efficiency."
- Hyper-V Containers
  - Each container gets a dedicated copy of the underlying OS kernel and memory. Much better isolation, but…
    - Limited sharing of resources means Hyper-V Containers are not far removed from virtual machines with regard to startup time and resource utilization.
  - Why use it? This is what you use in multi-tenant and public cloud environments.

http://www.techrepublic.com/article/containers-are-high-on-microsofts-agenda-for-windows-server-2016

Questions?

Interest?

Thank You!