# Docker/Containerized Reference Architecture

## Goals

* scalable
* resilient
* replicable

## Presentation Outline:

* players
* terms
* request flow diagram
* physical architecture diagram
* node diagram
* manage.containers
* consul
* dockerdash
  + docker -H ps
* registry
* cicd-21.containers
* proxy
  + show stop/start containers
* configuration
  + files vs keys
  + GetConfigurationFiles.ps1
  + UpdateAppSettings.ps1
* Enter application in online application
* View application in store portal
* Splunk

## Diagrams

Request Flow

Request Flow

Physical Layout

Physical Layout

Node

Node

## Players:

* **consul** - Consul is a highly available and distributed service discovery solution created by HashiCorp that provides a way for processes and services to register themselves and be aware of other components and services in a distributed environment via DNS or HTTP interfaces.
* In addition to service discovery, Consul provides several key features like offering a distributed Key/Value store which can be a powerful solution for shared configuration.
* In addition to the Key/Value store, Consul provides a way to define health checks for the registered services in the service catalog, the health check can take several forms like HTTP, script, or TTL health check.
* **consul-template** - Consul Template uses Consul to update files and execute commands when it detects the services in Consul have changed. For example, it can rewrite an nginx.conf file to include all the routing information of the services then reload the nginx configuration to load-balance many similar services or provide a single end-point to multiple services.
* **nginx** - NGINX is a free, open-source, high-performance HTTP server and reverse proxy. NGINX is known for its high performance, stability, rich feature set, simple configuration, and low resource consumption.
* **docker/docker-engine** - The Docker Engine is a lightweight container runtime and robust tooling that builds and runs your container. Docker allows you to package up application code and dependencies together in an isolated container that share the OS kernel on the host system. The in-host daemon communicates with the Docker Client to execute commands to build, ship and run containers.
* **registrator** - Service registry bridge for Docker. Registrator automatically registers and deregisters services for any Docker container by inspecting containers as they come online.
* **registry** - The Registry is a stateless, highly scalable server side application that stores and lets you distribute Docker images. This would be an on premise alternative to [Docker Hub](https://hub.docker.com/).
* **splunk/logging** - Log aggregation, search, analysis and visualization of machine generated streams of data.
* **application analytics** - Real-time application and performance monitoring. Popular in this space are tools like New Relic and AppDynamics.
* **kong** - Kong is a scalable, open source API Layer (also known as an API Gateway, or API Middleware). Kong runs in front of any RESTful API and is extended through Plugins, which provide extra functionality and services beyond the core platform. An API Gateway can support functions such as: reshaping/transformation, authentication, security, and traffic control.
* **git2consul** - git2consul takes one or many git repositories and mirrors them into Consul KVs. The goal is for organizations of any size to use git as the backing store, audit trail, and access control mechanism for configuration changes and Consul as the delivery mechanism.

## Terms:

* **image** - Docker images are the basis of containers. An image is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. An image typically contains a union of layered filesystems stacked on top of each other. An image does not have state and it never changes.
* **tag** - A tag is a label applied to a Docker image in a repository. tags are how various images in a repository are distinguished from each other.
* **container** - A container is a runtime instance of a docker image.
* **volume/data volume** - A data volume is a specially-designated directory within one or more containers that bypasses the Union File System. Data volumes are designed to persist data, independent of the container’s life cycle. Docker therefore never automatically delete volumes when you remove a container, nor will it “garbage collect” volumes that are no longer referenced by a container.
* **Dockerfile** - A Dockerfile is a text document that contains all the commands you would normally execute manually in order to build a Docker image. Docker can build images automatically by reading the instructions from a Dockerfile.
* **node** - A node is a physical or virtual machine running an instance of the Docker Engine in swarm mode. Manager nodes perform swarm management and orchestration duties. By default manager nodes are also worker nodes. Worker nodes execute tasks.
* **compose** - Compose is a tool for defining and running complex applications with Docker. With compose, you define a multi-container application in a single file, then spin your application up in a single command which does everything that needs to be done to get it running.
* **swarm** - A swarm is a cluster of one or more Docker Engines running in swarm mode.
* **service** - In terms of consul (to disambiguate from Docker Swarm), a service is your application that is running and exposed in a container. A container may have more than one exposed service.
* **service discovery** - Service discovery tools manage how processes and services in a cluster can find and talk to one another. It involves a directory of services, registering services in that directory, and then being able to lookup and connect to services in that directory.
* At its core, service discovery is about knowing when any process in the cluster is listening on a TCP or UDP port, and being able to look up and connect to that port by name.
* **reverse proxy** - A reverse proxy server is a type of proxy server that directs client requests to the appropriate backend server. A reverse proxy provides an additional level of abstraction and control to ensure the smooth flow of network traffic between clients and servers.
* Reverse proxies can also act as a load balancer, web accelerator (compression, cache), SSL endpoint and obscures the server and network infrastructure behind it. A reverse proxy can also do response rewriting.

## Unanswered:

* **networking/segregation** - Use of overlay networks with swarm to segregate containers. The networking side for windows has serveral known deficiencies and problems right now that are waiting on an upcoming windows update.
* **domain participation** - the containers do not belong to a domain. I am unsure if I think this is really a problem or not, but it would mean that the use of domain controlled resources and trusted connections would not be possible as it is now.
* **datacenter** - I have not done anything to prototype this across multiple data centers, but that is possible with the technologies currently in use, like Docker Swarm and Consul. Part of this would also be a responsibility of the orchestration tool that we choose. Use of a Global Traffic Manager would be required as well as active/active
* **scheduling/orchestration** - Orchestration is a broad term that refers to container scheduling, cluster management, and possibly the provisioning of additional hosts. In this environment, "scheduling" refers to the ability for an administrator to define a service that establishes how to run a specific container. While scheduling refers to the specific act of loading the service definition, in a more general sense, schedulers are responsible for hooking into a host to manage services in whatever capacity needed. One of the biggest responsibilities of schedulers is host selection based on defined contraints and host resource availability.
  + Possible choices include mesos/marathon, kubernetes and nomad among others.

## Docker Ideologies

Docker is rather unbiased in how it allows you to use it, giving you significant freedom. That said, there are several principles that are typically desirable that will guide how you set up containers.

* Keep containers small and cheap - this plays into the implementation details, but ideally your containers are very small, and thus easy to trash and re-create.
* One process per container - This is a good general rule for all containers. Later on we’ll see that this is not a hard and fast rule, but in general you want containers to have one main purpose, even if there are other smaller processes (eg. syslog) supporting it.
* Trash and rebuild, not fix and restart - This is another rule that we’ll find some exceptions to in the practical world, but ideally we treat containers like white boxes - if they’ve failed, throw it away and start some new ones.
* Store data in volumes, not containers - Volumes, as discussed above, are directories that live on the Host and are mounted into containers. They separate the data from the container itself, and allow it to persist beyond the container lifecycle, and even be shared by containers (eg. read only configuration). Almost all data you care about should be stored in volumes, including logs, configuration, user data, and interesting output.
* Don’t connect to containers directly - This is related to the above; you want to have access to everything you need in a container without having to connect to it directly, as this doesn’t scale well and is difficult in several situations. Ideally we won’t be running an ssh server on our container, and everything we’d want to see, like logs, should be exposed via volumes.
* Open as few ports as possible - While it may seem necessary to open ports for every application you want to connect to on your container, most of the time this is not true. Connecting containers with Docker Networks allows containers to intercommunicate, and if you are exposing the right data in volumes, you should have little need to open any ports. A common exception is a debug port for your database service, but even this can be replaced by a debug container that is connected to the docker network.