**Containerizing stateful applications**

**Can stateful applications be containerized? Or are containers only good for stateless applications?**

Application container technology, like Docker Engine, provides standards based packaging and runtime management of the underlying application components.

Containers are fast to deploy and make efficient use of system resources. Using containers, developers get application portability and programmable image management. The operations team get standard runtime units of deployment and management.

However, with all the known benefits of application containers there is a common misperception that containers are ephemeral and so are only good for stateless microservices-style applications. And that it’s not possible to containerize stateful applications. Let’s dive in and see if this holds up:

**Understanding application state**

Application state is simply data needed by application components to do their job i.e. perform a task. All applications have state Software programming architectural patterns, paradigms and languages, essentially describe how to manage application behaviors (tasks, operations, etc.) and state (data).

Even microservices style applications have state! In a microservices style architecture, each service can have multiple instances and each service instance is designed to be stateless. What that means, is that a service instance does not store any data across operations. Hence being stateless, simply means that any service instance can retrieve all application state required to execute a behavior, from somewhere else. This is an important [architectural constraint of microservices style applications](http://www.nirmata.com/2015/02/microservices-five-architectural-constraints/), as it enables resiliency, elasticity, and allows any available service instance to execute any task.

Typically, application state is stored in a database, a cache, a file, or some other form of storage. And, any change in application state that needs to be remembered across operations must be written back to storage.

So, all applications have state, but an application component can be stateless if it cleanly separates behaviors from data, and can fetch data required to perform any behavior. But this seems like simply passing the problem to something else - how does the other component manage the state? That depends on the type of state we’re discussing.

To answer that, let’s consider the five types of state an application may have, and how we can deal with each one of these to containerize the application:

1. Persistent state
2. Configuration state
3. Session state
4. Connection state
5. Cluster state

**Containerization and persistent state**

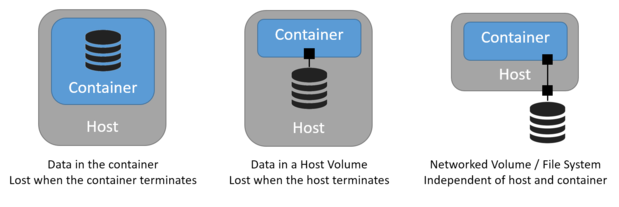
Persistent application state needs to survive application restarts and outages. This is the type of state is typically stored in a redundant database tier, and had periodic backups performed on it.

Although it’s possible to put applications and databases in the same container, it’s best to keep your database separate from your application as your application components will change far more frequently. And separating the database, also allows it to be shared across multiple application instances.

If your application already uses an external database, delivered as a service or installed on a different set of physical or virtual servers, you can keep that architecture and start simply by containerizing the application tier. Most container management systems will allow passing in the database access information, as configuration state, to the application-tier containers (see “Configuration State” below.)

Or, you can choose to containerize the database! This brings fast recovery and deployments, along with all other benefits of containers to the data tier. In this case, there are a few items to consider related to your database:

1. How does the database manage clustering and replication for availability and scale? Do the replicas have specific roles, or can new members join and get a dynamically assigned role?
2. How much data is there to be managed? Is it practical to do a full-sync when a new node the database cluster?
3. Based on the above, does your data, for a replica, need to live when the container running the database software terminates? How about when the host terminates?



To allow the data to exist when the container terminates,  you will need to use a storage mechanism that manages data outside of the container. This is easily done using host volumes and mapping them to containers.

Similarly, to allow the data to exist when the host terminates, you will need to use a storage mechanism that manages data outside of the host. Most cloud platforms support shared (networked) file systems or block storage (volumes) that can be independently managed and attached / detached to any host. So this too is fairly straightforward, assuming your container orchestrator provides lifecycle events to manage storage components.

But what if your data needs to stay attached to a specific container? There may be reasons why this may be necessary - for example one of our customers wanted to manage a large amount of video content which could not be replicated. If their container died and was restarted on another host, they wanted the same data to be available to that container.

If you have many such applications, volume plugins can simplify the orchestration of data. A volume plugin sits below the container engine and assists with storage orchestration. A number of volume plugins out there are simply thin wrappers around IaaS / CMP calls. But others aim to provide a rich set of features such as QoS and tiered storage and support for enterprise storage and may be worth a look.

Let’s summarize the options:

1. **Host volumes**: this works well for small datasets and if the database supports replicas which can join a cluster and dynamically sync-up with other members.
2. **Shared volumes or file systems**: when your data needs to survive independently of your host. This is a good option for large data sets where you don't want to perform a full data-sync when a new node joins the database cluster/
3. **Volume Plugins**: if you have applications where data needs to be attached to a container, or your orchestration does not allow managing external systems.

**Containerization and configuration state**

Applications typically need non-domain data to be configured correctly. This, configuration state, could be things like IP addresses for other external services, or credentials to connect to a database.

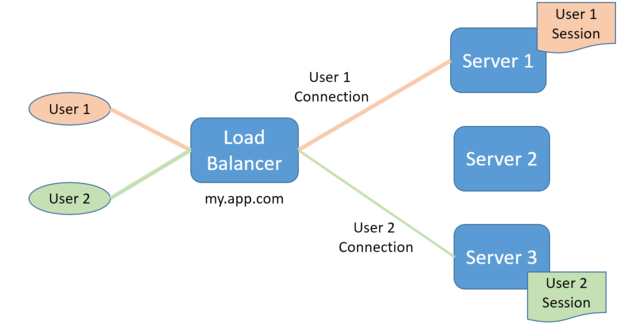
The [12-factor app](http://12factor.net/) guidelines popularized by Heroku and adopted by most PaaS solutions prescribe storing configuration data in the environment. In a containerized world, most configuration data is can be managed as environment variables that can be injected into the container.

However confidential information, like credentials, passwords, keys and other secret data, are best handled through other secure mechanisms that do a better job of not making the secret data visible and accessible on the host, network, or storage.  For this type of configuration state, credential management tools like KeyWhiz and Vault can be used from within a container with one-time access tokens. Other options combine use of volume plugins with a key store, to securely provide secret data to containerized applications.

**Containerization and session state**

When a user logs in, session data may be generated by the application. This could be an authentication key, or other temporary state for the user. In most modern applications the session state is stored in a distributed cache or a database that can be accessed by any service instance.

However, in traditional multi-page web applications each web page requires access to the session state managed by the server. Hence all user requests for that session must be directed to the same backend server, or the user will be forced to login again. Such applications are said to require “sticky sessions” where session state is stored in a specific server, and all requests for a client session are always routed to the same service.



This is not a containerization problem, as the same issue exists when requests are load-balanced across applications servers deployed in virtual or physical machines. And, most load balancers have an option to support sticky sessions.

In a containerized world, the IP addresses of your containers may be different than the IP addresses of your hosts. If you use a Layer 4-7 load balancing solution to front-end application containers with stateful session data, the load balancer will also need to handle sticky sessions.

Container-native solutions Nirmata’s [Service Gateway](http://nirmata-documentation.readthedocs.io/en/latest/ServiceNetworking.html#service-gateway) provide support for sticky sessions, and also dynamically update route information as containers get re-deployed across hosts.

**Containerization and connection state**

Some applications may communicate using protocols, like Websockets, which are considered stateful as the communicating entities can exchange a sequence of messages over a connection. In contrast, protocols like HTTP are considered stateless as the server does not remember any state across requests, allowing any other server to answer the next request.

If your applications uses stateful protocols, the container load-balancing solution will also need to support routing client requests to containers for stateful protocols. For example, if Websockets is used the load balancing solution will need to support TCP connections that persists across requests. Once again this feature is common in traditional load balancers, and can be found in most container-native load balancers.

**Containerization and cluster state**

Some applications run as multiple instances in a cluster, for availability and scale, and require shared knowledge of cluster membership and state. This state is not persistent but may need to be kept up to date, if cluster membership changes

In a clustered applications, each cluster member will need to know about other members and their roles. Most modern clustered applications require initial bootstrapping with a seed set of members, typically their IP addresses and ports, and then are able to dynamically manage membership and changes. However some clustered services may require a manual update and restart when membership information changes need to be propagated.

Container-native orchestration systems should be able to handle both of these cases. For example, Kubernetes recently introduced a feature called a [PetSet](http://kubernetes.io/docs/user-guide/petset/)for managing a stateful cluster.  Nirmata supports staged orchestration with reservations for clustered systems, where container placement is pre-calculated, and all cluster members are injected with unique identities and cluster state.

**Other types of state?**

As we work with customers to containerize their applications we’ve come across all sorts of interesting situations. For example, one application read the local MAC address and used it as a way to uniquely identify itself! Obviously, this scheme falls apart if the container gets restarted and gets a different MAC address.

Luckily, Docker now allows specifying the container MAC address. For such outliers, you will need to make sure your orchestration system has the flexibility to specify custom settings when running the container.

**Summary**

In this post we discussed what application state is, the different types of application states you are likely to encounter. We also covered how each type of state can be managed in a containerized environment. In most cases, several options are available to choose from. So, although containers are ephemeral, the application state does not need to be!

My goal for this post was to show that stateful applications can be indeed containerized. So, how did I do? Would love to hear about your experiences, or if you have any questions I can help answer.