The purpose of this article is to walkthrough the setting up of infrastructure for containerized applications to communicate and access files. Containerizing an application gives great flexibility in how, where, and at what scale it’s deployed. A drawback to this versatility is that applications often rely on accessing and writing to remote files, this poses a problem if the app is being deployed in varying environments. The same problem accurse when multiple applications or tasks need to communicate with one-another but networking or hosts are variable. This is the problem of discoverability.

One way to resolve this is to collect all of these environment differences into a configuration file and ensure that the config. file matches the deployed application and host environment. This solution is straight forward but has a significant maintenance cost as the number of apps increases.

<Diagram direct file🡪Dir🡪app🡪ip🡪app >

A solution for generalized deployment scenarios is to deploy containerizes that can interface between apps and the environment they are deployed into. Interfaces deployed as containers are instantly discoverable to other apps in the same Docker “network”[add link to Docker Network Doc].can be natively discovered by the individual applications they will be serving.

The general concept is to utilize an sftp server and a messaging queue to provide applications with file access and intra-app communication, respectfully. This simplifies maintenance by focusing deployment specific configurations to a single interface.

<Diagram file🡪sftp🡪app🡪queue>

## SFTP Server

* What is a file server

An sftp file server allows for applications to securely access files from a directory addressed on the file server but stored physically somewhere else. The advantages of accessing files through a server is that extremely complex or changing network mappings can be statically addressed by an app.

The below example uses the <sftp\_image\_name> image to create a file server with 2 directories. The first directory points to <phys\_dir> on the host machine and is a addressed <srv\_dir> on the server. Likewise the second directory, though in much different source location, is mapped to the same parent directory as the last. In addition the ```ro``` suffix makes the first directory read only, very useful when safe guarding sourcing files from alteration.

* Describe the image being used
* Describe configuration options for volume
* Docker-compose code segment

## Messaging Queue

* What is a messaging queue
  + Pub/sub model

Messaging queues are standard design concepts for microservice applications. Queues allow for applications to be “loosely” connected. Meaning, that each app communicates with the queue instead of with each other. A message queue, in a way, is a running list of small messages. Messages are posted by one app (publishers) and then read by another (subscribers). This example uses an image created by RabbitMQ. RabbitMQ makes connectors for lots of different languages and is efficient for general purpose use cases.

Below is the docker-compose yaml for deploying the RabbitMQ broker.

<docker-compose-rabbitmq>

You can follow a detailed “Hello World” walkthrough in your language of choice from the RabbitMQ site [Include “Hello World” link]. Below are simple python scripts for a publisher and a subscriber.

<publisher-script>

<subscriber-script>

* What is RabbitMQ
* Configuration options
* Python pub example code segment
* Python sub example code segment
* Docker-compose code segment