**How to containerize**

***Preparation***

Identify filesystem locations where persistent data is written

Since deploying a new version of the application is performed by replacing the Docker image, any persistent data must be stored *outside* of the container. If you're lucky, the application already writes all its data to a specific path, but many legacy applications spread their data all over the filesystem and intermingle it with the application itself. Either way, Docker's volume mounts let us expose the host's filesystem to specific locations in the container filesystem so that data survives between containers, so we must identify the locations to persist.

You may at this stage consider modifying the application to support writing all data within a single tree in the filesystem, as that will simplify deployment of the containerized version. However, this is not necessary if modifying the application is impractical.

Identify configuration files and values that will vary by environment

Since a single image should be usable in multiple environments (e.g. test and production) to ensure consistency, any configuration values that will vary by environment must be identified so that the container can be configured at startup time. These could take the form of environment variables, or of values within one or more configuration files.

You may at this stage want to consider modifying the application to support reading all configuration from environment variables, as that that will simplify containerizing it. However, this is not necessary if modifying the application is impractical.

Identify services that can be easily externalized

The application may use some services running on the local machine that are easy to externalize due to being highly independent and supporting communication by TCP/IP. For example, if you run a database such as MySQL or PostgreSQL or a cache such as Redis on the local system, that should be easy to run externally. You may need to adjust configuration to support specifying a hostname and port rather than assuming the service can be reached on localhost.

***Creating the image***

Create a Dockerfile that installs the application

If you already have the installation process automated via scripts or using a configuration management tool such as Chef or Puppet, this should be relatively easy. Start with an image of your preferred operating system, install any prerequisites, and then run the scripts.

If the current setup process is more manual, this will involve some new scripting. But since the exact state of the image is known, it's easier to script the process than it would be when you have to deal with the potentially inconsistent state of a raw system.

If you identified externalizable services earlier, you should modify the scripts to *not* install them.

A simple example Dockerfile:

# Start with an official Ubuntu 16.04 Docker image

FROM ubuntu:16.04

# Install prerequisite Ubuntu packages

RUN apt-get install -y <REQUIRED UBUNTU PACKAGES> \

&& apt-get clean \

&& rm -rf /var/lib/apt/lists/\*

# Copy the application into the image

ADD . /app

# Run the app setup script

RUN /app/setup.sh

# Switch to the application directory

WORKDIR /app

# Specify the application startup script

COMMAND /app/start.sh

Startup script for configuration

If the application takes all its configuration as environment variables already, then you don't need to do anything. However, if you have environment-dependent configuration values in configuration files, you will need to create an application startup script that reads these values from environment variables and then updates the configuration files.

An simple example startup script:

#!/usr/bin/env bash

set -e

# Append to the config file using $MYAPPCONFIG environment variable.

cat >>/app/config.txt <<END

my\_app\_config = "${MYAPPCONFIG}"

END

# Run the application using $MYAPPARG environment variable for an argument.

/app/bin/my-app --my-arg="${MYAPPARG}"

Push the image

After building the image (using docker build), it must be pushed to a Docker Registry so that it can be pulled on the machine where it will deployed (if you are running on the same machine as the image was built on, then this is not necessary).

You can use [Docker Hub](https://hub.docker.com/) for images (a paid account lets you create private image repositories), or most cloud providers also provide their own container registries (e.g. [Amazon ECR](https://aws.amazon.com/ecr/)).

Give the image a tag (e.g. docker tag myimage mycompany/myimage:mytag) and then push it (e.g. docker push mycompany/myimage:mytag). Each image for a version of the application should have a unique tag, so that you always know which version you're using and so that images for older versions are available to roll back to.

**How to deploy**

Deploying containers is a big topic, and this section just focuses on directly running containers using docker commands. Tools like [docker-compose](https://docs.docker.com/compose/) (for simple cases where all containers run on a single server) and [Kubernetes](http://kubernetes.io/) (for container orchestration across a cluster) should be considered in real-world usage.

Externalized services

Services you identified for externalization earlier can be run in separate Docker containers that will be linked to the main application. Alternatively, it is often easiest to outsource to managed services. For example, if you are using AWS, using RDS for a database or Elasticache for a cache significantly simplifies your life since they take care of maintenance, high availability, and backups for you.

An example of running a Postgres database container:

docker run \

-d \

--name db \

-v /usr/local/var/docker/volumes/postgresql/data:/var/lib/postgresql/data \

postgres

The application

To run the application in a Docker container, you use a command-line such as this:

docker run \

-d \

-p 8080:80 \

--name myapp \

-v /usr/local/var/docker/volumes/myappdata:/var/lib/myappdata \

-e MYAPPCONFIG=myvalue \

-e MYAPPARG=myarg \

--link db:db \

myappimage:mytag

The -p argument exposes the container's port 80 on the host's port 8080, -v argument sets up the volume mount for persistent data (in the hostpath:containerpath format), the -e argument sets a configuration environment variable (these may both be repeated for additional volumes and variables), and the --link argument links the database container so the application can communicate with it. The container will be started with the startup script you specified in the Dockerfile's COMMAND.

Upgrades

To upgrade to a new version of the application, stop the old container (e.g., docker rm -f myapp) and start a new one with the new image tag (this will require a brief down time). Rolling back is the similar, except that you use the old image tag.

**Additional considerations**

"init" process (PID 1)

Legacy applications often run multiple processes, and it's not uncommon for orphan processes to accumulate if there is no "init" (PID 1) daemon to clean them up. Docker does not, by default, provide such a daemon, so it's recommended to add one as the ENTRYPOINT in your Dockerfile. [dumb-init](https://engineeringblog.yelp.com/2016/01/dumb-init-an-init-for-docker.html) is an example lightweight init daemon, among others. [phusion/baseimage](http://phusion.github.io/baseimage-docker/) is a fully-featured base image that includes an init daemon in addition to other services.

See our blog post dedicated to this topic: [Docker demons: PID-1, orphans, zombies, and signals](https://www.fpcomplete.com/blog/2016/10/docker-demons-pid1-orphans-zombies-signals).

Daemons and cron jobs

The usual way to use Docker containers is to have a single process per container. Ideally, any cron jobs and daemons can be externalized into separate containers, but this is not always possible in legacy applications without re-architecting them. There is no intrinsic reason why containers cannot run many processes, but it does require some extra setup since standard base images do not include process managers and schedulers. Minimal process supervisors, such as [runit](http://smarden.org/runit/), are more appropriate to use in containers than full-fledged systems like systemd. [phusion/baseimage](http://phusion.github.io/baseimage-docker/) is a fully-featured base image that includes runit and cron, in addition to other services.

Volume-mount permissions

It's common (though not necessarily recommended) to run all processes in containers as the root user. Legacy applications often have more complex user requirements, and may need to run as a different user (or multiple processes as multiple users). This can present a challenge when using volume mounts, because Docker makes the mount points owned by root by default, which means non-root processes will not be able to write to them. There are two ways to deal with this.

The first approach is to create the directories on the host first, owned by the correct UID/GID, before starting the container. Note that since the container and host's users don't match up, you have to be careful to use the same UID/GID as the container, and not merely the same usernames.

The other approach is for the container itself to adjust the ownership of the mount points during its startup. This has to happen while running as root, before switching to a non-root user to start the application.

Database migrations

Database schema migrations always present a challenge for deployments, because the database schema can be very tightly coupled with the application, and that makes controlling the timing of the migration important, as well as making rolling back to an older version of the application more difficult since database migrations can't always be rolled back easily.

A way to mitigate this easily is to have a staged approach to migrations. You need to make an incompatible schema change, you split that change over two application deployments. For example, if you want to move a piece of data from one location to another, these would be the phases:

1. Write the data to both the old and new locations, and read it from the new location. This means that if you roll the application back to the previous version, any the new data is still where it expects to find it.
2. Stop writing it to the old location.

Note that if you want to have deployments with no downtime, that means running multiple versions of the application at the same time, which makes this even more of a challenge.

Backing up data

Backing up from a containerized application is usually easier than the non-containerized deployment. Data files can be backed up from the host and you don't risk any intermingling of data files with application files because they are strictly separated. If you've moved databases to managed services such as RDS, those can take care of backups for you (at least if your needs are relatively simple).

Migrating existing data

To transition the production application to the new containerized version, you will need to migrate the old deployment's data. How to do this will vary, but usually the simplest is to stop the old deployment, back up all the data, and restore it to the new deployment. This should be practiced in advance, and will necessitate some down time.

**Conclusion**

While it requires some up-front work, containerizing a legacy application will help you get control of, automate, and minimize the stress of deploying it. It sets you on a path toward modernizing your application and supporting no-downtime deployments, high availability, and horizontal scaling.

FP Complete has undertaken this process many times in addition to building containerized applications from the ground up. If you'd like to get on the path to modern and stress-free deployment of your applications, you can learn more about our [Devops](https://www.fpcomplete.com/devops) and [Consulting](https://www.fpcomplete.com/consulting) services, or contact us straight away!