Containerization and Docker

Supplement to the Video Presentation

The Problem

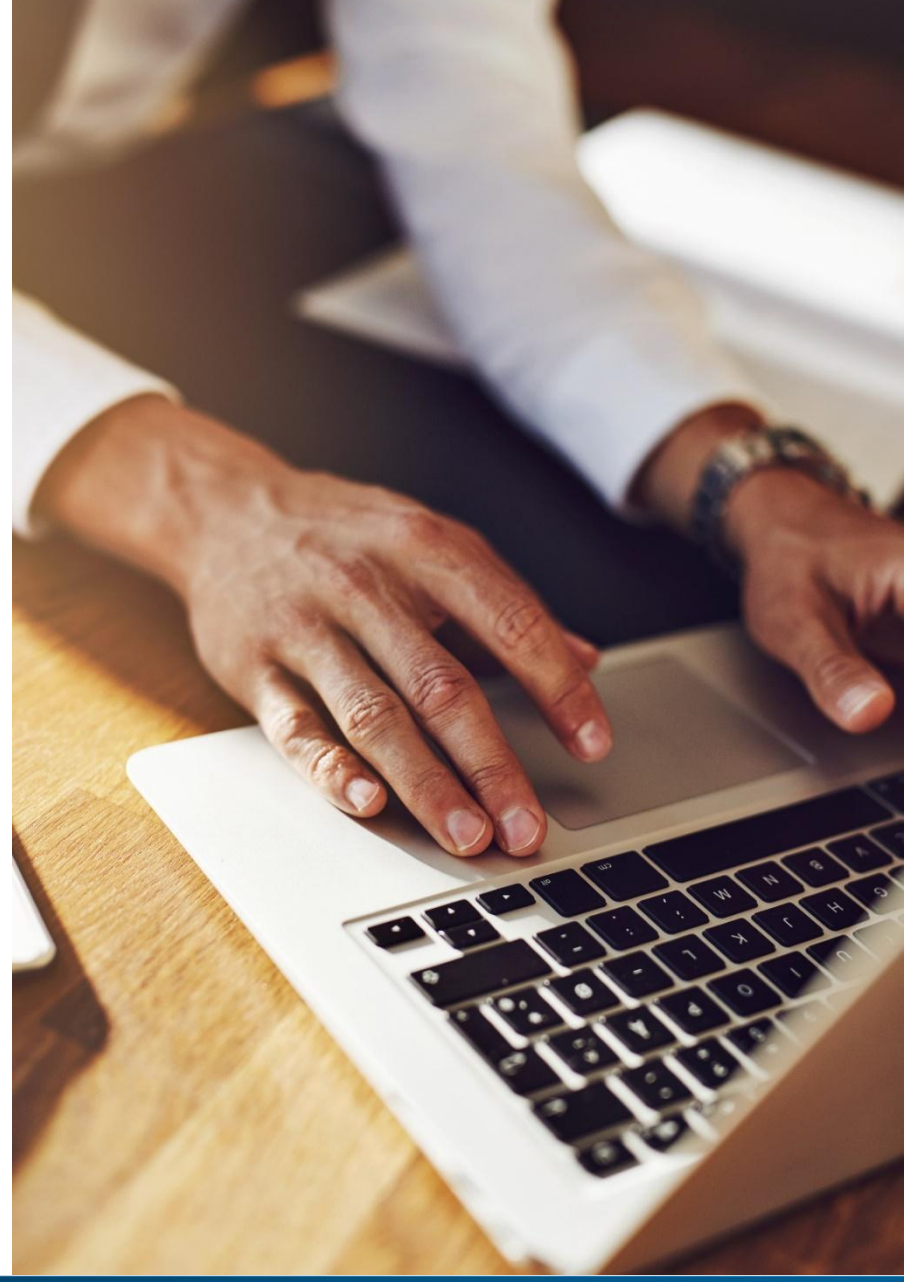
For complex development environments, getting code to work on multiple computers can be a challenge.

While it is possible to share development environments without containerization, there is no guarantee it will work across any OS – in fact, it almost never will.

Containerization

Containerization can be used to streamline the process of developing and sharing complicated applications.

Using Docker is an effective way to share an app you've built on your own computer – especially if you want to actively maintain it for all users without them needing to worry about it.



What is Docker?

Docker is a lightweight containerization software.

It creates isolated file systems, or containers, that your services run inside of.

Containers use a construction blueprint, called an “image”. Images are constructed in stepwise fashion by users.

Containers can be networked together.

https://www.docker.com/

What is Docker? (2)

Docker containers are meant to be built and destroyed quickly, not sustained indefinitely. This reduces the required resources and improves security.

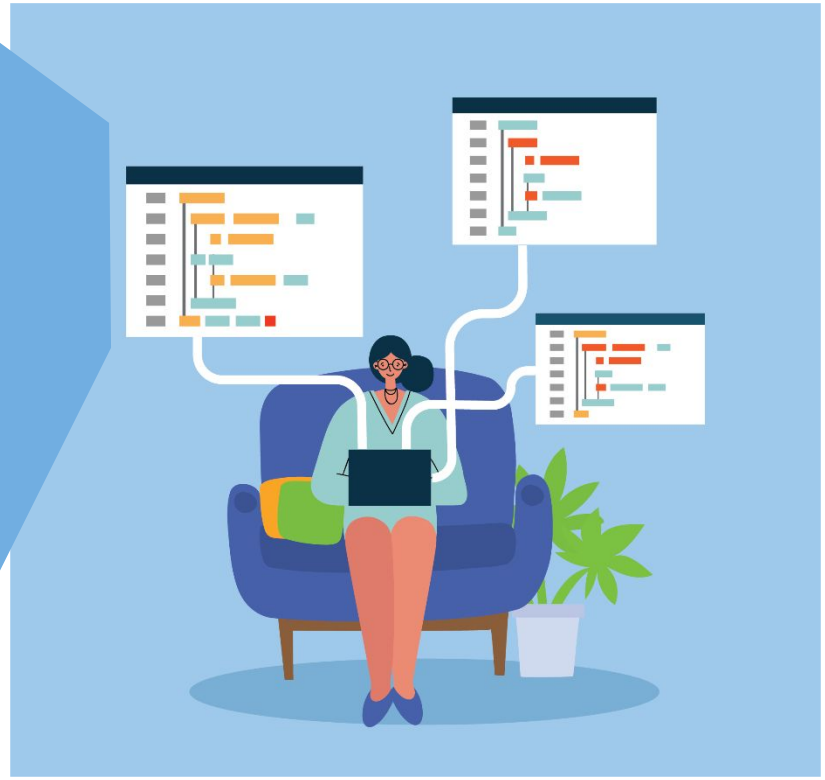
Docker is not virtualization software. Containers are not VMs and do not have their own OS – instead, they depend on the host's kernel. This means that there is a limit to Docker's flexibility – more on this at the end of the presentation.

https://www.docker.com/

Using Docker: Creating .dockerfiles

The .dockerfile is the recipe for an image. The main components of most .dockerfiles are the same:

1. Pull a base image from an online repository.
2. Copy local files (your code) into the new container.
3. Execute commands on files in the container.



.dockerfile Base Images

Base images are curated, lightweight versions of a normal OS.

There are base images available for every commonly used OS on public repositories.

Note

Advanced users can create their own base images and “go distroless” – not something to worry about right now.

.dockerfile Commands

There are many we won't cover – these commands are the ones we'll go over in the video example.

1. **FROM** – Used to specify a base image. When executing a .dockerfile, Docker assumes that you are pulling from its own repo. If you pull from somewhere else, you'll need to specify – but that's beyond the scope of this tutorial.

.dockerfile Commands (Cont.)

2. **COPY & WORKDIR** – Used to make content from your local filesystem available inside the container.

   Whatever folder you copy into is relative to that container - it is isolated from everything else, and therefore can have any name you'd like. This is relevant when considering Docker volumes (stay tuned).

   The .dockerignore text file (just like a .gitignore file) prevents unwanted files in the container. Very useful when the .dockerfile is in the parent folder.

.dockerfile Commands (Cont.)

3. **RUN** – Used to issue many types of commands to intermediate containers – installing packages like *curl*, for example.

   There are many “best practice” rules concerning security and deployment speed. Many are outside the scope of this tutorial, but a general rule: pipe together multiple related commands where possible to reduce the number of intermediate stages needed by your container.

.dockerfile Commands (Cont.)

4. **ENV** – Changes default environmental variable values within the container. **RUN** commands can do this too, but variables set in an intermediate stage do not persist in the final container.

   ENV variables can take whatever value you'd like and be named whatever you'd like.

.dockerfile Commands (Cont.)

5. **EXPOSE** – This simple command informs the container which port to get all its required information from during runtime. While running a service inside of a container on your local machine, you'll often navigate to the localhost port number in a browser.

   For example: **EXPOSE** 8080 □ localhost:8080

.dockerfile Commands (Cont.)

6. **CMD** & **ENTRYPOINT** – both are used to issue command(s) to a freshly created, final-stage container.

   Using **CMD** is akin to writing something at the command line – but only the final **CMD** in the .dockerfile is executed, so use only one.

   Alternatively, **ENTRYPOINT** points to another file that was copied into the container – all commands in that file are executed.

Tip
An ENTRYPOINT command can be used to build and activate a curated conda environment!

.dockerfile Commands (Cont.)

7. **VOLUME** – perhaps the trickiest Docker command. Used to "mount" a file-system to a container. **When files are "mounted", the container may see, use, or even delete them**.

   In addition, changes made to container-specific files persist even after the container is destroyed – when it is rebuilt, everything will be as it was*.

   You can even destroy and rebuild the container's image without making changes to the volume. This is possible because of the Docker Daemon's root privilege.

   *The name of the volume matters! To persist data, one of your volumes should have the same name as the WORKDIR – the container's version of changes made to those files (for instance, a .db file) will live with the Daemon while the container is down.

Increasing Complexity with Docker-Compose

If your application requires more than one service, it is best to network multiple containers together – each with their own .dockerfile – using Docker-Compose.

Don’t worry – these don’t look too different than a regular .dockerfile, with many of the same components.

In general, each container should run just one service and run it well.

> For example: a website where one container runs the web interface, and another container is responsible for the underlying database.

The Major Limitation of Docker

Cross-Platform Compatibility:

Remember that intro? Well, Docker can help to solve that problem … sort of.

In general, Docker is not cross-platform compatible – not without special tinkering and legwork. It can be made to work in many cases!

Docker is not a VM – it relies on the hosting filesystem's kernel.

It can work when the host is "compatible enough" with the development environment – for instance, a Docker service developed in an older Ubuntu environment may work in other Debian-based Linux environments.

"Distro-less" base environments can be made to ignore this rule. Not every service can work without a distro, though.

These are ultra-complicated use-cases. It's recommended to steer clear of this initially – intend for your containers to run on the same OS they were developed on.

Alternatives to Docker

Docker has slowly been buying many of its competitors and incorporating them into its own software, possibly in preparation for incorporating a new paywall. They still exist as open-source projects, but each individually isn't a feature-complete alternative.

Some of the competitors are: *runC*, *containerd*, and even *Compose*!

Alternatives to Docker (Cont.)

HyperV: Windows only, but each container has its own kernel. Improved security and reliability, but slower and with a bigger footprint.

rkt: Features cross-platform compatibility, high security (no daemon, no root access), and can use both Docker containers and Kubernetes. However, not being actively supported by parent company.

LXC: Linux-only, with emphasis on multi-service containers. Not as portable as Docker containers, more complexity can be packed into any given container.

Good-old-fashioned VMs.

Alternatives to Docker (Cont.)

Kubernetes-based (K8s) containerization: Not containerization software itself, but it “orchestrates” storage, load-balancing, and software rollouts between many compute resources. The K8 folks say: “if it runs in a container, it should run great in Kubernetes.”

This is more-so an alternative to Docker Swarm, than Docker itself. In fact, Docker containers can run on K8s, too! Docker Swarm is well beyond the scope of this intro – if you get into deploying containers as a swarm on an HPC or shared-server down the line, make sure to check out K8, as well as other systems designed to run on K8s.

Special Mention: **Ruckstack**

MRIGlobal

The science you expect.
The people you know.

THANK YOU

Follow us and learn more about our work

Facebook.com/MRIGlobalResearch

Youtube.com/user/MRIExternalComm

Linkedin.com/company/mriglobal

twitter.com/MRIGlobal