Packaging Software with Docker

Joseph Redfern

November 18, 2020

The Problem

Software is fast moving.

Scripts or tools that worked 2–3 years ago (or less) may not work now due to changes to the system running the software.

Software X may require library version A, software Y may require library version B.

Breakages can be called by operating system upgrades, updates to dependencies.

The Problem (cont.)

Some ecosystems try to solve this problem with virtual environments:

- `pip` and `requirements.txt`
- `anaconda` and `environment.yml`
- Tools like `npm`/`nvm` for node.js packages/version, `gem`/`rvm` for ruby packages/versions

These solutions solve some (but not all) of these issues.

System library requirements can be issue, even down to things like version of `glibc`.

Often require the end user be familiar with the ecosystem, not necessarily the case.

What is Docker?

Builds on standard features from Linux kernel, like `cgroups`, `netfilter`, `namespaces`.

Provides a set of tools that allows developers to package and distribute software as stand-alone containers.

A container is *like* a lightweight virtual machine – boot time can be in milliseconds.

However, shares Kernel of host machine (unlike more traditional virtualisation), but not userspace.

What is Docker? (cont.)

A container has its own filesystem, but you can map directories from the host to the container if desired.

Registry of base “images” on which to build – these images typically the userspace (libraries/system utilities) of existing operating systems, or preconfigured environments for existing software packages.

Docker (the company) provide an image registry (Docker hub). University’s GitLab install also features Docker registry.

Example usage

To distribute software through Docker, we need to create an image.

To create an image, we need to write a `Dockerfile`.

The `Dockerfile` contains the commands needed to assemble our image.

Includes reference to parent container, defines working directory, copies files and resources from build machine to container, runs commands in the container before being snapshotted.

Also defines commands to be run when container starts.

Example Dockerfile

```
# Specify the image on which this image will be based.
FROM python:3.8.3

# Sets the working directory for subsequent commands.
WORKDIR /usr/src/app

# Copy requirements from current directory on build machine
# to the working directory on the image.
COPY requirements.txt ./

# Copy the src/ directory on build machine to src on image.
COPY src ./src

# Install python dependencies from requirements.txt.
# --no-cache-dir stops pip from caching the packages, which
# saves some space.
RUN pip install --no-cache-dir -r requirements.txt

# CMD specifies the command (and it's arguments) to be run
# when the container starts
CMD [ "python", "./src/coolscript.py"]
```

Building an image

Once we've defined our `Dockerfile`, we need to build the image.

Use the command: `docker build . -t name:version`.

The `-t` arg lets us tag the image with a name & version – for instance, `-t awesomesoft:1.3.37`.

Demo

<demo goes here>

Parting comments

Docker makes it easy to distribute functional software, minimising barrier to entry. Helpful for reproducible research.

Containers can be really lightweight. Minimal performance hit, very little overhead.

Some security benefits – can limit access to filesystem, restrict network access, limit memory/CPU usage. (not without pitfalls)

Can use cuda within Docker using `nvidia-docker`.

Possible to pass USB devices to Docker containers with `--device` flag.

Compatible with Windows and macOS (through virtualisation).

Extremely popular in industry (cloud hosting providers support it, platforms like Kubernetes)