Deploying software: Docker and more

Arran Stewart

Overview

"Deployment" of software is everything we need to do to make it available to clients for use.

It can include:

- configuring software and hardware
- writing program "installers"
- making a website available
- providing updates to software (and a mechanism for getting them)

Overview

"Deployment" of software is everything we need to do to make it available to clients for use.

It can include:

- configuring software and hardware
- writing program "installers"
- making a website available
- providing updates to software (and a mechanism for getting them)

What steps would be part of "deployment" for your project?

Not just once

Some software engineering texts give the impression that "deployment" is something that happens once, at the end of the software development lifecycle.

But every time we need to make software available to a client, that counts as "deployment" – this includes

- providing prototypes or demonstrations to the client to get feedback
- providing a version of the software to the client for *acceptance testing*
- providing updates to the software

Not just once

Some software engineering texts give the impression that "deployment" is something that happens once, at the end of the software development lifecycle.

But every time we need to make software available to a client, that counts as "deployment" – this includes

- providing prototypes or demonstrations to the client to get feedback
- providing a version of the software to the client for *acceptance testing*
- providing updates to the software

And some sorts of software – for instance, websites – can be regarded as being deployed continuously.

Useful guides

Two good books that cover many issues that arise in deployment are:

- *Release It! – Design and Deploy Production-Ready Software*, by Michael Nygard (2nd ed, Pragmatic Programmers LLC)
- *Ship It! – A Practical Guide to Successful Software Projects*, by Jared Richardson and Will Gwaltney (2nd ed, Pragmatic Programmers LLC)

Problems that arise

If you're deploying an executable or app, you may find that the deployment environment – the systems on which the software will be running – differs from your development environment in unexpected ways.

You may find

- missing libraries
- missing programs that your project depends on (or which differ in behaviour to the versions you used)
- different operating system versions
- programs that interfere with the operation of your software (for instance, firewalls or malware/virus scanners)

Ways to tackle this

These problems may not arise for all projects (for instance, if you are deploying a website, and have a high level of control over your deployment environment).

But when they do, and if you have a client with some technical expertise, *containerization* can be a convenient way of tackling them.

What is “containerization”?

It means packaging up software *and all its dependencies* so that it can run with consistent behaviour on different infrastructure.

Docker[1] is an example of containerization software.

To a first approximation, a “container image” is just a collection of files – a program, the data it needs, and all its dependencies.

Together with some instructions on what sort of network environment it has access to, what programs should be run when the container is started, and so on.

[1] and similar technologies – we’ll see some later

Advantages of Docker

- Control over software installed – you can specify exactly what versions of software are installed in a Docker "container"
- Control over environment – you can specify what networking environment, files etc are visible to software running in a container

Limitations of Docker

- It's a Linux-only technology, in the sense that Docker *containers* must be running a version of Linux
  - However, virtualization software (like VirtualBox or VMWare) means Docker can still be run on Windows or MacOS computers.
- May not be suitable for end-users with little technical expertise
- Not suitable for embedded systems
- May be overkill if your software can be supplied as a statically-linked executable

How does it differ from a VM?

- If you run an instance of a virtual machine, you can emulate *any* sort of computer – you are not limited to the OS or hardware of the
  host system
  - However, programs will often run more slowly than they would if executed directly on the host hardware
- If you run a Docker container, you can provide any sort of *Linux* environment you want.

How does it differ from a VM?

	Container	Virtual machine
Speed	Fast	Often slower
Degree of isolation	Moderate	Very high
Size of artifacts	MB–GB	Often hundres of GB

So how do you use Docker

- Both you and the client need to have Docker installed – Docker Desktop for MacOS and Windows is often a good choice

Writing Dockerfiles

Dockerfile

```
FROM ubuntu:18.04

RUN sudo apt-get update && \
  apt-get install python3

WORKDIR /app
COPY hello.html .\

CMD ["python3", "-m", "http.server"]
```

Building images and containers

If we type

```
docker build -f Dockerfile -t myserver:0.1 .
```

then Docker will build a Docker "image" – rather like a VM image, it can be run or instantiated multiple times.

The image is normally stored on a directory on your computer as *layers* – each layer corresponding to a command in the Dockerfile, and consisting of a set of files that is "layered" over a previous set of files.

Distributing images

After an image (consisting of multiple layers) has been built, it can be "pushed" to a container *registry* – a server that allows images to be uploaded to and downloaded from it.

In general, you'll probably want to use a *private* registry, that only you and your client have access to.

Amazon, Google Cloud and Microsoft Azure all provide private container registries, which can be convenient if you're already using those cloud providers.

But if not, two services that provide free private registries are:

- Gitlab (https://gitlab.com)
- Treescale (https://treescale.com)

Distributing images

To send an image to a registry, we need to run the Docker "`login`" and "`push`" commands:

```
$ docker login registry.example.com -u myuserid
$ docker push myserver:0.1 registry.example.com/myproject/myserver:0.1
```

Using images

A client can then use the `docker run` command to download the layers in your image, and run a container:

```
$ docker run registry.example.com/myproject/myserver:0.1
```

How does Docker work?

If want to, say, look at the files in a directory on your computer, or see what programs are running, then on Linux, we'd use commands like "`ls`" or "`ps`".

We normally expect that the results the operating system gives us, are exactly what is there.

But they needn't be; the operating system could be set up (for instance) to not let some users see particular directories or processes.

How does Docker work?

An ancestor of Docker, the "`chroot`" program, was developed in 1979 – it ran programs in an environment where they could only see the files and directories under a particular parent directory.

Since then, Linux has added many other features (notably, Linux "namespaces") that allow us to precisely control the environment that a process sees.

The basic features of a containerization system can be implemented just using Linux shell commands – if interested, check out Bocker, an implementation of containers in 100 lines of Bash shell code.

Alternatives to Docker

- Podman (https://podman.io) – almost a drop-in replacement for Docker, but has some rough edges
- Full virtualization systems (for instance, libvirt, VirtualBox and VMWare)
- Containerization systems similar to docker – for instance LXC/LXD (developed by RedHat)

Exercise – Docker for beginners

We'll work through the Docker-provided "Docker for beginners" tutorial, at https://github.com/docker/labs/ (follow the "Docker for beginners" link).