DOCKER 101/A – UCREL NLP SUMMER SCHOOL EDITION

This session is being in a *hybrid lecture format* - this means you can choose your own engagement! If you prefer to just listen along with the demonstration or dive in head-first with the worksheet instructions, feel free!

We encourage you to use the lab desktops for this, but if you do want to use your own laptop, please install one of the following:

- Podman Desktop - https://podman-desktop.io/
- Docker Desktop - https://www.docker.com/products/docker-desktop/

On the lab PCs and the instructions here, we will be using Podman, but if you have Docker installed simply replace any instance of 'podman' in the instructions with 'docker' and the commands should be equivalent. If you have any issues, please ask.

BASIC CONTAINER OPERATIONS

NOTE: the $> at the start of code steps indicates the bash terminal prompt and should not be typed!

Many of the commands we will be running are self-describing – try adding `--help` to the commands in this session to see what other options you have.

We can immediately run a container without any additional configuration by launching the following:

```
$> podman run -it alpine
```

This should present you with a command prompt… but a different one! Your terminal is now showing you the Busybox shell built-in to the Alpine Linux runtime. If you explore around inside this container with `cd` and `ls` you'll notice that you're not in the same directory as your home directory in the labs – the container runs in an entirely separate *namespace* for this, and is entirely isolated from the host.

If you open a new terminal window and run the following command, you should see your container running on the host:

```
$> podman ps
```

Try exiting from the alpine container (with `exit`) and re-run `podman ps` – maybe unsurprisingly the container has gone; but If we run `podman ps -a` you'll see that the container still exists in an 'exited' state; from this we could revive the container and reattach our terminal to continue, but in general its best to think of these as throw-away instances, and recreate the container anew if we want to use the environment again.

To clear out all old, exited instances, we can run:

```
$> podman rm -a
```

Using `--help` on the commands above, explore launching, running, stopping and removing containers. Here are a few commands you might want to explore; you may want to launch a handful of terminals to do this:

```
$> podman stats
$> podman run --rm -it ubuntu
$> podman stop --help
```

BUILDING YOUR OWN IMAGES

Containers are only useful if we can build our own software to run inside them! To do this we use `Dockerfile` files. For this session we have prepared some examples which can be found at:

https://github.com/UCREL/Session_7_Docker_101a

(Note there are no spaces in the URL, those are underscores 😊)

Dockerfiles describe a recipe to build a container image; like Make or CMake files, and these recipes are built upon one another in layers. We should always strive to work from the closest possible image to what we need, to minimise changes and overheads. It makes little sense to rebuild a whole Linux image from nothing every time for a Python-based project, when Python docker images already exist.

Try searching for your favourite programming language (or your least favourite!) over at **https://hub.docker.com/** and see if there is an official image for that language – most likely there is, and much of the work has been done for you already.

Download a copy of one or more of the Dockerfiles in the example repository and open them in VSCode (or other text editor). There is some nuance with some of the directives in this file, but most are self-explanatory; if you want to know more about any of them, we can look to the reference documentation:

https://docs.docker.com/reference/dockerfile/

When you're ready to build your first image, from the same directory as the Dockerfile, run the following command:

```
$> podman build -t my-cool-image .
```

Note the dot at the end! This is important – it's actually the path to where the build system should start looking for files and is known as the *built context*. In more complex configurations you may want this to be a specific directory, but as we're building in a single location, the dot simply means 'right here'.

Images are indexed by 'tag' which is essentially just a special name for a particular image, without which we would have to use the complete hash name of the image (which would be a pain!). To tag our own image, we use the `-t` flag with any name we choose after. In production systems this name might be dictated by your organisation or affiliations. Often there are also *versioned images* which include a version string after a colon. If we wanted a specific build of Ubuntu Linux we could specify `ubuntu:20.04` or if we wanted *the latest version no matter what* we could also use `ubuntu:latest`.

Some commands you may want to try:

```
$> podman image ls
$> podman image pull phpmyadmin
$> podman image tree phpMyAdmin:latest
```

The official Docker image registry is the Docker Hub, but others are available including the option to host your own. In no particular order, some examples of registries might be:

- https://quay.io/
- https://ghcr.io/
- https://goharbor.io/
- https://cloud.google.com/artifact-registry
- https://jfrog.com/container-registry/

Each has its own particular philosophy, and you should take these into account when downloading or publishing images to or from these registries (this beyond the scope of this workshop!)

RUNNING MORE – WORKING WITH MANY CONTAINERS

Having a single program running in isolation is increasingly rare, and we often need to run several services at once. Containers offer a way to do this via *compose* files. Examples 2 and 3 in the repository show these in practice.

```yaml
services:
  db:
    image: mariadb:10.6
    restart: unless-stopped
    environment:
      MYSQL_ROOT_PASSWORD: changeMe

  phpmyadmin:
    image: phpmyadmin
    restart: unless-stopped
    ports:
      - 8080:80
    environment:
      PMA_ARBITRARY: 1
    depends_on:
      - db
```

The contents of `docker-compose.yml` for Example 2

In Example 2, we define two services, each one which corresponds to a single container running a specified image; in this case, one for the 'MariaDB MySQL database engine' and the other for 'phpMyAdmin' which presents a web-based database management interface

We also specify a which `ports:` we want to be available to connect to. In this case, we're using port `8080` on our host machine to connect to port `80` in the `phpmyadmin` container.

Both containers require an environment variable to be set to function correctly, and we can also specify these in the `docker-compose.yml` file under the `environment` heading; the changes to the environments we specify for each container are independent of one another, so our `phpmyadmin` container will not have the `MYSQL_ROOT_PASSWORD` variable set, for example.

When we are ready to run our composition, we can launch both containers at once with:

```
$> podman compose up
```

This will run both containers and print their output on the terminal – once they are running you should be able to visit http://localhost:8080/ and view the web interface to phpMyAdmin.

Compose files can also be used to configure storage, networking and other capabilities, however learning all the options available via compose is far beyond the scope of this session, but the documentation is extremely good if you want to take this further and can be found here:

https://docs.docker.com/compose/compose-file/

Some commands you may want to explore:

```
$> podman stats
$> podman compose down
$> podman compose up -d
$> podman compose pull
$> podman compose --help
```

PUTTING EVERYTHING TOGETHER – BUILDING MANY CONTAINERS AT ONCE

In the previous example we saw Podman using pre-existing container images, but we can build our own images and use these as well. Compose files have a `build` directive, which should point to where a Dockerfile is stored describing how to build each image.

```
...
worker:
    image: demo-worker:latest
    build: worker
    restart: unless-stopped
    deploy:
      mode: replicated
      replicas: 5
...
```

Part of the contents of `docker-compose.yml` for Example 3

In the fragment above we include the `build` directive instructing compose to look in the `worker` folder for how to build the `demo-worker:latest` image.

We additionally include a new sub-section named `deploy` which describes how we want this container to be deployed when we `podman compose up` with this script. In this case we're asking compose to build us 5 identical replicas of the `worker` container, which it will automatically name `worker-0, worker-1, worker-2, etc.`

GOING FURTHER – INTO THE DOCKERVERSE

Additional resources:

- https://podman.io/
- https://www.docker.com/
- https://docs.docker.com/
- https://docs.docker.com/guides/use-case/ ← NLP and AI/ML Guides from the Docker folks directly 😊

There are also both original talks available via John's site:

- https://johnvidler.co.uk/blog/docker-101/
- https://johnvidler.co.uk/blog/docker-102/

Supercomputers and Multicomputers!

- https://ucrel-hex.scc.lancs.ac.uk/
- https://n8cir.org.uk/bede/
- https://www.lancaster.ac.uk/rse/