Making and Using Docker Containers

This section should cover the steps and basic principles of how to use, make, and push Docker containers and images to Docker Hub, using the correct tagging.

- The Bits and Bobs of Docker
- Dockerfiles and Making Images
    - Choosing Your Base Image
    - Building Your Image
- Docker Hub & Uploading Containers
    - Pushing to your Repository
- Running and Maintaining Docker Deployments
    - Ensuring Containers Run at Startup

The Bits and Bobs of Docker

This page should serve as a basic introduction to the individual parts one would need to know to begin building a Docker image.

Docker Engine

Docker Desktop

Dockerfile

Docker Compose (YAML)

Dockerfiles and Making Images

Choosing Your Base Image

Base images are, in a sense, the preexisting foundation that you will most likely build your containers upon. Often, it's a stripped down operating system like Alpine or Chiseled Ubuntu, but it can be something much more lightweight if needed, or you can even make your container completely from scratch by, well, adding `FROM scratch` to your Dockerfile. In this case, "scratch" is your base image, essentially meaning that nothing extra is added. This is most useful when trying to make your image as small and secure as possible. You will have to add every single dependency into your container by hand, however, so it can often be significantly more work to maintain.

Alpine

If you want to use a base like Alpine (a very lightweight Linux distro designed for containerization) for your image, you must be aware of the limitations and advantages that your base gives you. For instance, while Alpine is very small and minimal, it has it's own unique package manager invoked with `pkg`. Packages are installed by adding `pkg install [package_name]` to your Dockerfile while in a RUN instruction (this might look like `RUN pkg install ffmpeg` for instance), thus executing the package installation on the base image when the docker image is being built. Another potential drawback of Alpine is that it uses the incredibly lightweight and secure `musl libc` as its standard C library rather than `glibc` (the GNU C Library), which can affect compatibility with a number of pre-compiled Linux binaries. These will have to be recompiled for `musl libc` to work properly on Alpine, otherwise you may get assorted errors, such as these executable file not being found, even when present on the file system.

Building Your Image

To build your Docker image, make sure you are building within the same directory as your Dockerfile.

```
docker build -t myusername/myimage:tag .
```

Multi-platform builds, with multiple tags, and auto-push:

```
docker buildx build --push \
--platform linux/arm/v7,linux/arm64/v8,linux/amd64 \
--tag your-username/packagename:versionNum \
--tag your-username/packagename:latest .
```

If you just need to re-build the project due to updated dependencies, and don't want Docker to use the pre-cached steps of the build process, use `--no-cache` in the build command.

Docker Hub & Uploading Containers

Pushing to your Repository

Once you have your repository created in hub.docker.com, you can begin tagging and uploading your created docker images.

For instance, assuming we have a container called `myrepo/myimage`, you may want to build a version of your image tagged with the correct version number you want to use (such as 1.1.2) using `docker build -t "myrepo/myimage:tagname" .`. Or, if you don't want to re-create the image with a tag, you can simply run `docker tag myrepo/myimage myrepo/myimage:tagname` to add a tag to the image.

Now that you have a finished image with a version number, you can run the same command as above, but this time have the source image be the tagged version number, and the created image be tagged as latest. For example, you may run `docker tag myrepo/myimage:versionnumber myrepo/myimage:latest` , replacing "versionnumber" with the version number of the image you created and intend to upload, and keeping the word "latest" the same.

Now that you have two identical docker images, one tagged with a version number, and one tagged with "latest", you can push both images to your repository in Docker Hub with the following two commands:

```
docker push myrepo/myimage:versionnumber
docker push myrepo/myimage:latest
```

Here's a real-world example:

```
docker push zeppelinsforever/music-downloader:0.0.2
docker push zeppelinsforever/music-downloader:latest
```

If this is your first time pushing an image to your repository, you may have to create an account in Docker Desktop, then generate a PGP key and pass it to Docker Desktop to ensure secure pushes.

More information can be found in the Docker docs: https://docs.docker.com/desktop/setup/sign-in/

Running and Maintaining Docker Deployments

Ensuring Containers Run at Startup

You can run a Docker container that is downloaded to your system with:

```
docker run --restart=always container-name-or-ID
```

However, to ensure that the Docker Engine *service* runs at startup (which is needed for containers to run), also ensure that said service is *enabled*. Without the Docker Engine service, specifying `--restart=always` is pointless. No docker containers can run without Docker Engine being active on a system, both at startup and during system usage.

On Linux distributions which rely on **systemd** for service management (that is, most of them), you can enable the Docker Engine service to run at startup with the following command:

```
sudo systemctl enable docker
```

If you're not using systemd... you're on your own. Read your boot/service manager's documentation on starting system services.

If Docker is installed properly and the system recognizes "docker.service" as a possible service present on the machine, your device will now run Docker Engine at startup, thus enabling your containers specified with `--restart=always` to run at startup too.