Docker

Basic Commands
Container Lifecycle
    Executing Command in a Running Container
Creating Custom Docker Image
    Creating a Dockerfile
    Building the Image
        Specifying a Work Directory
        Container Port Mapping
        Rebuilds with Cache
Docker Compose
    Docker Compose Commands
    Restart Policies - Automatically Restarts Containers
Creating Production Grade Workflow
    Dockerfile.dev && Dockerfile.prod
    Docker Volume

Basic Commands

1. Creating and running a container using an image: `docker run <image_name>`
    a. Every image that is used to create and run a container comes with a default command that is executed when the container starts.
2. To override this default command: `docker run <image_name> command_to_be_executed`
3. List all running containers (only running/active): `docker ps`

```
CONTAINER ID   IMAGE     COMMAND             CREATED          STATUS          PORTS     NAMES
bbfa8419d60f   busybox   "ping google.com"   25 seconds ago   Up 23 seconds             sleepy_torvalds
```

To get a list of all the containers that were ever created/active, run command `docker ps --all`

```
CONTAINER ID   IMAGE         COMMAND             CREATED          STATUS                        PORTS     NAMES
580ff088a805   hello-world   "/hello"            8 minutes ago    Exited (0) 42 seconds ago               hungry_ganguly
bbfa8419d60f   busybox       "ping google.com"   21 minutes ago   Exited (137) 20 minutes ago             sleepy_torvalds
```

Container Lifecycle

1. The `docker run` command is a combination of `docker create <image_name>` and `docker start <container_id>`

    Creating a container refers to setting up the file system snapshot that will be used later. Starting up a container refers to actually executing the startup command.
2. On running `docker create <image_name>` returns the `CONTAINER_ID` (something like 580ff088a80519fcf2bba060ed961aae2af7d294ec15b336b2b4a7d10ae10150) for the newly created container. Now, the container can be started with:

    `docker start -a 580ff088a80519fcf2bba060ed961aae2af7d294ec15b336b2b4a7d10ae10150`
    a. The `-a` flag is used to attach the container to the terminal, to watch for any output that the container might provide. Starting a container without the `-a` flag will only start the container but now show any available outputs - `-a, --attach Attach STDOUT/STDERR and forward signals`

3. Restarting a stopped container: `docker start <container_id>`. This restarts the container and executes the startup command. When restarting a container, the startup command cannot be changed or re-issued.
4. Remove stopped / unnecessary containers: `docker system prune`. This deletes all the stopped containers and other data like cache, and downloaded images, and returns ids of containers that were deleted and space reclaimed.

```
WARNING! This will remove:
  - all stopped containers
  - all networks not used by at least one container
  - all dangling images
  - all dangling build cache


Deleted containers:
580ff088a80519fcf2bba060ed961aae2af7d294ec15b336b2b4a7d10ae10150
bbfa8419d60fcd115c1aabb3c09dfa3d0bd480bcca75736d52d12a3a12ab367e

Total reclaimed space: 0B
```

5. Get logs from a container using `docker logs <container_id>` which shows all the output that the container emitted, without restarting / running the container again.
6. To stop a container, issue `docker stop <container_id>` or `docker kill <container_id>`.
   a. Both commands stop a running container with just one difference - the `stop` command lets the container stop at its own terms, with any cleanup necessary while the `kill` command forces the process/container to stop immediately.
   b. If the container does not stop within 10 seconds after issuing the stop command, the `kill` command is automatically issued.

Executing Command in a Running Container

To execute an additional command in a running container - during the time of starting a container - run, `docker exec -it <container_id> <command_to_execute>`

a. The `-it` flag is used to attach / provide input to the container.
b. `-it` flag is equivalent to using `-i` `-t` that attaches our terminal with the container's `STDIN`, `STDOUT` ports.
c. Opening up a shell / terminal inside the running container: `docker exec -it <container_id> sh`
d. To open bash or powershell, use `bash` or `powershell` in place of `sh`.
e. To exit, `CTRL + C` (windows) / `CMD + C` (mac) or `exit`.

Creating Custom Docker Image

1. Create a Docker file
2. Docker client (docker-cli)
3. Docker server
4. Usable image

Creating a Dockerfile

1. Specify a base image.
2. Specify command(s) to download dependencies or additional programs
3. Specify startup command

```
// specify a base image
FROM <base_image_name>

 // copies files from current dir or context to the root of the container
```

```
COPY . .

// command to download additional programs / dependencies
RUN npm install

// stratup command
// specified as an array of strings
CMD ["npm", "start"]
```

Building the Image

To build an image using the docker-cli, run `docker build .` The dot at the end of the command is used to specify the build context - it tells docker to take everything from ***this*** directory to build the image.

To add a name / tag to the image, add the `-t` flag with the name as `<dockerhub-username>/<project_name>:<version>`. Only `<project_name>` is mandatory since rest values substitute to defaults if not provided.

`docker build -t ec2346aakashjha/crmnext:1 .`

Specifying a Work Directory

Nothing more than creating a new folder inside the fs of the container that will hold all the code. To create a working directory while creating an image using a Dockerfile, specify `WORKDIR /path/to/folder` before the `COPY` step. This will create a new folder and then copy files from `.` to `WORKDIR`.

```
// specify a base image
FROM <base_image_name>

// creates a new workspace folder inside app folder
WORKDIR ./app/workspace

 // copies files from current dir or context to the workdir of the container
COPY . .

// command to download additional programs / dependencies
RUN npm install

// stratup command
// specified as an array of strings
CMD ["npm", "start"]
```

Container Port Mapping

Refers to forwarding any incoming request made to a specific port on the local system to the port of the container. If we send request to port 8080 on our system, it will be forwarded to the specified port of the container.

`docker run -p <source_port> : <container_port> <image_name>`

Example: `docker run 3000:3100 <image_name>`

1. Port number the from and to port number can be same or different.
2. `<container_port>` specified while executing the `run` command and the server port for the application should be same.

Rebuilds with Cache

When the docker build . command is executed for the very first time, and an image is created, a cache is also generated that is later used when some changes are made in the Dockerfile.

If there are 5 steps in the Dockerfile and a change is made in step 4, the build process will use the cache for the first 3 steps and rebuild the image from step 4 onwards.

```
FROM <base_image_name>

WORKDIR ./app/workspace

COPY package.json .
RUN npm install
```

```
// if nothing changes in package.json, build process will use the cache upto that process
// and then execute a fresh build process to include changes

COPY . .

RUN npm install

CMD ["npm", "start"]
```

Docker Compose

1. Automate long arguments that are passed with docker run
2. Start multiple containers and connect them together
3. Needs a `.yaml` file which will then parsed by the `docker-compose cli`
4. By using docker-compose, multiple containers can be created in the same network and all that should be done is port mapping to forward traffic from our local machine to the container.

Sample usecase

An application needs a databse to work along with it (say MongoDB). The application and MongoDB instance will run on separate containers which needs to communicate with each other to perform operations.

```
docker-compose.yaml

// specify the version of docker-compose to be used
version: '3'

// services refers to containers
services:

  // create a container for MongoDB
  mongo-db:
    image: 'mongo'

  // create a container for the application
  application:
    // instead of directly using an image, build the image for the application using the Dockerfile in the curre
ct directory
    build: .
    ports:
      // a dash in a yaml file is used to specify an array
      // mapping port 3000 of our local machine with port 4000 of the application container
      - "3000:4000"
```

Docker Compose Commands

1. To create an instance of all the services listed in the docker-compose.yaml file, run: `docker-compose up` command - similar to `docker run <image_name>`
2. To first build / rebuild an image using a Dockerfile and then starting up the instance using that image, run: `docker-compose up --build` - a combination of `docker build .` and `docker run <image_name>`
3. Launch an instance in the background, run: `docker-compose up -d`
4. Stop all running container, run: `docker-compose down`
5. To list status of all the containers inside the docker compose file, run: `docker-compose ps` . And this only works when ran from the directory that contains the `docker-compose.yaml` file

Restart Policies - Automatically Restarts Containers

In case the server running in our container crashes / is stopped, we can automatically restart it based on `STATUS CODE` emitted by the process. Any code except `0` is considered to be un-natural process exit.

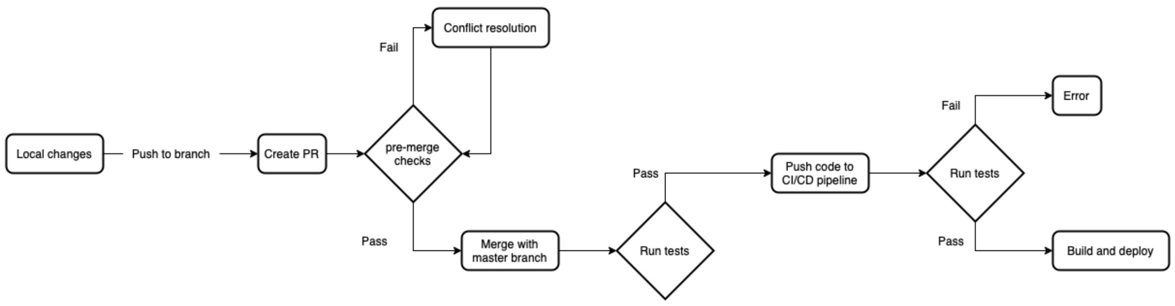
1. `"no"` - should not restart if container is stops or crashes. A no without quotes is interpreted as a `boolean` value, hence quotes are used.
2. `always` - always restart if the container stops or crashes
3. `on-failure` - only restart the container if it stops with an error code (any code except 0)
4. `unless-stopped` - only restart the container if it is stopped by the user (stops with a code 0)

```
docker-compose.yaml

version: '3'
services:
  mongo-db:
    image: 'mongo'
  application:
    restart: "no" / always / on-failure / unless-stopped
    build: .
    ports:
      - "3000:4000"
```

Creating Production Grade Workflow

A normal production workflow may look like



Dockerfile.dev && Dockerfile.prod

1. By default, `docker build .` looks for a `Dockerfile`. To use a custom Dockerfile - like Dockerfile.dev or Dockerfile.prod - run: `docker build -f <filename> .`

Docker Volume

When running in the dev env, any change made in the source code on the local machine should be reflected by the dev container. One way to incorporate those changes is to stop the container, rebuild the image using the Dockerfile.dev and restart the container using the new image. To avoid this, Docker Volume is used which is more like setting up a reference that points to the files and folders on the local machine (something similar to port mapping): `docker run -p 3000:3001 -v /workdir/node_modules -v $(pwd):/app <image_id>`

1. `-v` flag to create a Docker volume or mark a folder/directory as a docker volume
2. `pwd` works only on Linux/MacOS, or on windows if using gitbash
3. `-v $(pwd):/<workdir>` maps everything from the current directory (present working directory) to the working directory of the container (same working directory specified in the Dockerfile)
4. `-v /<workdir>/node_modules` does not actually map the `node_modules` folder from our local machine to the one inside the dev container, but **only acts a placeholder for node_modules but is not copied or mapped**.

5. Once docker volume is created, any changes to the local files / file system will automatically propagate to the running container.