```
$ docker image prune
$ docker container prune
$ docker volume prune
$ docker network prune
```

Try each of them out. If you are tired of answering y each time you can add the flag `-f` or `--force` to the command, e.g.:

```
$ docker system prune --force
```

This is especially useful if you want to run the above commands as part or an automation script.

11 Inspection Commands

11.1 System Information

1. We can find the `info` command under `system`. Execute:

   ```
   $ docker system info
   ```

2. From the output of the last command, identify:
   - how many images are cached on your machine?
   - how many containers are running or stopped?
   - what version of containerd are you running?
   - whether Docker is running in swarm mode?

11.2 System Events

1. There is another powerful system command that allows us to monitor what's happening on the Docker host. Execute the following command:

   ```
   $ docker system events
   ```

   Please note that it looks like the system is hanging, but that is not the case. The system is just waiting for some events to happen.

2. Open a second terminal and execute the following command:

   ```
   $ docker container run --rm alpine echo 'Hello World!'
   ```

   and observe the generated output in the first terminal. It should look similar to this:

   ```
   2017-01-25T16:57:48.553596179-06:00 container create 30eb63 ...
   2017-01-25T16:57:48.556718161-06:00 container attach 30eb63 ...
   2017-01-25T16:57:48.698190608-06:00 network connect de1b2b ...
   2017-01-25T16:57:49.062631155-06:00 container start 30eb63 ...
   2017-01-25T16:57:49.065552570-06:00 container resize 30eb63 ...
   2017-01-25T16:57:49.164526268-06:00 container die 30eb63 ...
   2017-01-25T16:57:49.613422740-06:00 network disconnect de1b2b ...
   2017-01-25T16:57:49.815845051-06:00 container destroy 30eb63 ...
   ```

3. If you don't like the format of the output then we can use the `--format` parameter to define our own format in the form of a Go template. Stop the events watch on your first terminal with CTRL+C, and try this:

   ```
   $ docker system events --format '--> {{.Type}}-{{.Action}}'
   ```

   now the output looks a little bit less cluttered when we run our alpine container on the second terminal as above.

4. Finally we can find out what the event structure looks like by outputting the events in `json` format (once again after killing the events watcher on the first terminal and restarting it with):

```
$ docker events --format '{{json .}}'
```

which should give us for the first event in the series after re-running our alpine container on the second node something like this (note, the output has been prettyfied for readability):


```
{
   "status":"create",
   "id":"95ddb6ed4c87d67fa98c3e63397e573a23786046e00c2c68a5bcb9df4c17635c",
   "from":"alpine",
   "Type":"container",
   "Action":"create",
   "Actor":{
      "ID":"95ddb6ed4c87d67fa98c3e63397e573a23786046e00c2c68a5bcb9df4c17635c",
      "Attributes":{
         "image":"alpine",
         "name":"sleepy_roentgen"
      }
   },
   "time":1485385702,
   "timeNano":1485385702748011034
}
```


11.3 Summary

In this exercise we have learned how to inspect system wide properties of our Docker host by usign the `docker system inspect` command. We have also used the `docker system events` command to further inspect the activity on the system when containers and other resources are created, used and destroyed.

12 Creating and Mounting Volumes

12.1 Creating a Volume

1. Create a volume called `test1`:

```
$ docker volume create --name test1
```

2. Run `docker volume ls` and verify that you can see your `test1` volume.
3. Execute a new `ubuntu` container and mount the `test1` volume. Map it to the path `/www/website` and run bash as your process:

```
$ docker container run -it -v test1:/www/website ubuntu:16.04 bash
```

4. Inside the container, verify that you can get to `/www/website`:

```
$ cd /www/website
```

5. Create a file called `test.txt` inside the `/www/website` folder:

```
$ touch test.txt
```

6. Exit the container without stopping it by hitting `CTRL + P + Q`.

12.2 Volumes During Image Creation

1. Commit the updated container as a new image called test and tag it as `1.0`:


```
$ docker container commit <container ID> test:1.0
```

2. Execute a new container with your test image and go into its bash shell:

```
$ docker container run -it test:1.0 bash
```

3. Verify that the `/www/website` folder exists. Are there any files inside it? Exit this container.
4. Run `docker container ls` to ensure that your first container is still running in preparation for the next step.

In this section, we saw how to create and mount a volume, and add data to it from within a container. We also saw that making an image out of a running container via `docker container commit` does *not* capture any information from volumes mounted inside the container; volumes and images are created and updated completely independently.

12.3 Finding the Host Mountpoint

1. Run `docker volume inspect` on the `test1` volume to find out where it is mounted on the host machine (see the 'Mountpoint' field):

```
$ docker volume inspect test1
```

2. Elevate your user privileges to root:

```
$ sudo su
```

3. Change directory into the volume path found in step 1:

```
$ cd /var/lib/docker/volumes/test1/_data
```

4. Run `ls` and verify you can see the `test.txt` file you created inside your original container.
5. Create another file called `test2.text` inside this directory:

```
$ touch test2.txt
```

6. Exit the superuser account:

```
$ exit
```

7. Use `docker container exec` to log back into the shell of your `ubuntu` container that is still running:

```
$ docker container exec -it <container name> bash
```

8. Change directory into the `/www/website` folder, and verify that you can see both the `test.txt` and `test2.txt` files. Files written to the volume on the host listed by `docker volume inspect` appear in the filesystem of every container that mounts it.

12.4 Deleting Volumes

1. After exiting yor container and returning to the host's bash prompt, attempt to delete the `test1` volume:

```
$ docker volume rm test1
```

2. Delete the remaining container without using any options:

```
$ docker container rm -f <container ID>
```

3. Run `docker volume ls` and check the result; notice our `test1` volume is still present, since removing containers doesn't affect mounted containers.
4. Check to see that the `test.txt` and `test2.txt` files are also still present in your volume on the host:

```
$ sudo ls /var/lib/docker/volumes/test1/_data
```


5. Delete the `test1` volume:

```
$ docker volume rm test1
```

6. Run `docker volume ls` and make sure the `test1` volume is in fact gone.

12.5 Conclusion

Volumes are the Docker-preferred tool for persisting data beyond the lifetime of a container. In this exercise, we saw how to create and destroy volumes; how to mount volumes when running a container; how to find their locations on the host (under `/var/lib/docker/volumes`) and in the container using `docker volume inspect` and `docker container inspect`; and how they interact with the container's union file system.

13 Volumes Usecase: Recording Logs

13.1 Set up an app with logging

1. Create a volume called `nginx_logs`:

```
host$ docker volume create --name nginx_logs
```

2. Run the custom `trainingteam/nginx` container and map your `public_html` host folder to a directory at `/usr/share/nginx/html`. Also mount your `nginx_logs` volume to the `/var/log/nginx` folder. Name the container `nginx_server`:

```
host$ docker container run -d -P --name nginx_server \
        -v ~/public_html:/usr/share/nginx/html \
        -v nginx_logs:/var/log/nginx \
        trainingteam/nginx
```

3. Get terminal access to your container:

```
host$ docker container exec -it nginx_server bash
```

4. Put some text into `/usr/share/nginx/html/index.html`, and then exit the terminal.
5. Run `docker container ls` to find the host port which is mapped to port 80 on the container. In your browser, access the URL and port nginx is exposed on.
6. Verify you can see the contents of your `index.html` file from your `public_html` folder on your host.

13.2 Inspect Logs on the Host

1. Get terminal access to your container again:

```
host$ docker container exec -it nginx_server bash
```

2. Change directory to `/var/log/nginx`:

```
container$ cd /var/log/nginx
```

3. Check that you can see the `access.log` and `error.log` files.
4. Run `tail -f access.log`, refresh your browser a few times and observe the log entries being written to the file. Exit the container terminal after seing a few live log entries.
5. Run `docker volume inspect nginx_logs` and copy the path indicated by the "Mountpoint" field; path should be `/var/lib/docker/volumes/nginx_logs/_data`.
6. Check for the presence of the `access.log` and `error.log` files, then follow the tail of `access.log`:


```
host$ sudo ls /var/lib/docker/volumes/nginx_logs/_data
host$ sudo tail -f /var/lib/docker/volumes/nginx_logs/_data/access.log
```

7. Refresh your browser a few times in order to make some requests to the NGINX server; observe log entries being written into the `access.log` file, available in the `nginx_logs` volume on your host machine.

13.3 Sharing Volumes

1. Run `docker container ls` and make sure that your `nginx_server` container from the last step is still running; if not, restart it.
2. Run a new `ubuntu` container and mount the `nginx_logs` volume to the folder `/data/mylogs` as read only, with bash as your process:

```
host$ docker container run -it \
        -v nginx_logs:/data/mylogs:ro \
        ubuntu:14.04 bash
```

3. In your new container's terminal, change directory to `/data/mylogs`
4. Confirm that you can see the `access.log` and `error.log` files.
5. Try and create a new file called `text.txt`

```
container$ touch test.txt
```

   Notice how it fails because we mounted the volume as read only.

13.4 Conclusion

In this exercise, you explored how mounting volumes makes live data being generated inside a container available to other containers and the outside world; the information nginx was writing to the logging volume can be consumed in real time by independent monitoring applications that would survive the failure or deletion of the nginx container.

We also used the `:ro` flag to mount a volume in read-only mode, so the corresponding container can't damage any data in the volume. This is also an important security best practice, since sharing volumes between containers breaks isolation between containers; by allowing read-only access, we prevent the container from injecting any malicious data into the volume that would then appear inside all other containers using that volume, as well as the host.

14 Demo: Docker Plugins

Plugins are used to extend the capabilities of the Docker Engine. Anyone, not just Docker, can implement plugins. Currently only volume and network driver plugins are supported, but in future support for more types will be added.

14.1 Installing a Plugin

1. Plugins can be hosted on Docker Store or any other (private) repository. Let's start with Docker Store. Browse to https://store.docker.com, enter `tiborvass/rexray-plugin` in the search box, and click on 'Community' under Filters->Type on the left to see plugins made by community members. The result should show you the plugin that we are going to work with.
2. Install a plugin that stores volumes on Amazon EBS into our Docker Engine:

```
$ docker plugin install tiborvass/rexray-plugin
```

   The system should ask us for permission to use privileges; answer with `y`.

3. Once we have successfully installed some plugins we can use the `ls` command to see the status of each of the installed plugins. Execute:

```
$ docker plugin ls
```

14.2 Inspecting, Enabling and Disabling a Plugin

1. We can inspect a plugin via:

```
$ docker plugin inspect tiborvass/rexray-plugin
```

   Note the access and secret keys are unset at the moment.

2. Once a plugin is installed it is `enabled` by default. We can disable it using this command:

```
$ docker plugin disable tiborvass/rexray-plugin
```

   only when a plugin is disabled can certain operations on it be executed, like changing settings.

3. Set the required access token for this plugin to be able to write to EBS:

```
$ AWS_ACCESSKEY = XXX
$ AWS_SECRETKEY = YYY
$ docker plugin set tiborvass/rexray-plugin \
  EBS_ACCESSKEY=$AWS_ACCESSKEY EBS_SECRETKEY=$AWS_SECRETKEY
```

4. The plugin can be (re-)enabled by using this command:

```
$ docker plugin enable tiborvass/rexray-plugin
```

14.3 Using the Plugin

1. To use the plugin we create a Docker volume:

```
$ docker volume create -d tiborvass/rexray-plugin my-ebs-volume
```

2. Now we can use that volume to access the remote folder and work with it as follows. Execute the following command to run an `alpine` container which has access to the remote volume:

```
$ docker container run -v my-ebs-volume:/volume-demo -it ubuntu:16.04 /bin/bash
```

3. Inside the container navigate to the `/volume-demo` folder and create a new file:

```
$ cd /volume-demo
$ touch data.dat
```

4. Have a look on the Amazon EC2 console - your EBS volume named `my-ebs-volume` should be present.

14.4 Removing a Plugin

1. If we don't want or need this plugin anymore we can remove it using the command:

```
$ docker plugin disable tiborvass/rexray-plugin
$ docker plugin rm tiborvass/rexray-plugin
```

   Note how we first have to disable the plugin before we can remove it.

14.5 Summary

In this task we have learned how to install, inspect and remove plugins into or from a Docker engine. We have specifically installed a plugin that allows us to access a remote volume on Amazon EBS. We have then created a Docker volume that uses this plugin and have demonstrated its usage.

15 Introduction to Container Networking

15.1 The Default Bridge Network

1. First, let's investigate the linux bridge that Docker provides by default. Start by installing bridge utilities:

   ```
   $ apt-get install bridge-utils
   ```

2. Ask for information about the default Docker linux bridge, `Docker0`:

   ```
   $ brctl show docker0
   ```

3. Start some named containers and check again:

   ```
   $ docker container run --name=u1 -dt ubuntu:14.04
   $ docker container run --name=u2 -dt ubuntu:14.04
   $ brctl show docker0
   ```

   You should see two new some virtual ethernet (veth) connections to the bridge, one for each container. `veth` connections are a linux feature for creating an access point to a sandboxed network namespace.

4. The `docker network inspect` command yields network information about what containers are connected to the specified network; the default network is always called `bridge`, so run:

   ```
   $ docker network inspect bridge
   ```

   and find the IP of your container `u1`.

5. Connect to container `u2` of your containers using `docker container exec -it u2 /bin/bash`.

6. From inside `u2`, try pinging container `u1` by the IP address you found in the previous step; then try pinging `u1` by container name, `ping u1` - notice the lookup works with the IP, but not with the container name in this case.

7. Still inside container `u2`, run `ip a` to see some information about what the network connection looks like from inside the container. Find the `eth0` entry, and confirm that the MAC address and IP assigned are the same (Docker always assigns MAC and IP pairs in this way, to avoid collisions).

8. Finally, back on the host, run `docker container inspect u2`, and look for the `NetworkSettings` key to see what this connection looks like from outside the container's network namespace.

15.2 User Defined Bridge Networks

In the last step, we investigated the default bridge network; now let's try making our own. User defined bridge networks work exactly the same as the default one, but provide DNS lookup by container name, and are firewalled from other networks by default.

1. Create a bridge network by using the `bridge` driver with `docker network create`:

   ```
   $ docker network create --driver bridge my_bridge
   ```

2. Examine what networks are available on your host:

   ```
   $ docker network ls
   ```

   You should see `my_bridge` and `bridge`, the two bridge networks, as well as `none` and `host` - these are two other default networks that provide no network stack or connect to the host's network stack, respectively.

3. Launch a container connected to your new network via the `--network` flag:

```
$ docker container run --name=u3 --network=my_bridge -dt ubuntu:14.04
```

4. Use the `inspect` command to investigate the network settings of this container:

```
$ docker container inspect u3
```

`my_bridge` should be listed under the `Networks` key.

5. Launch another container, this time interactively:

```
$ docker container run --name=u4 --network=my_bridge -it ubuntu:14.04
```

6. From inside container u4, ping u3 by name: `ping u3`. Recall this didn't work on the default bridge network between u1 and u2; DNS lookup by container name is only enabled for explicitly created networks.

7. Finally, try pinging u1 by IP or container name as you did in the previous step, this time from container u4. u1 (and u2) are not reachable from u4 (or u3), since they reside on different networks; all Docker networks are firewalled from each other by default.

15.3 Inter-Container Communication

1. Recall your container u2 is currently plugged in only to the default `bridge` network; confirm this using `docker container inspect u2`. Connect u2 to the `my_bridge` network:

```
$ docker network connect my_bridge u2
```

2. Check that you can ping the u3 and u4 containers from u2:

```
$ docker container exec u2 ping u3
$ docker container exec u2 ping u4
```

3. Check that you can ping the u2 and u4 container from u3

```
$ docker container exec u3 ping u2
$ docker container exec u3 ping u4
```

4. Note u1 still can't reach u3 and u4:

```
$ docker container exec u1 ping u3
$ docker container exec u1 ping u4
```

15.4 Conclusion

In this exercise, you explored the fundamentals of container networking. The key take away is that *containers on separate networks are firewalled from each other by default*. This should be leveraged as much as possible to harden your applications; if two containers don't need to talk to each other, put them on separate networks.

You also explored a number of API objects:

- `docker network ls` lists all networks on the host
- `docker network inspect <network name>` gives more detailed info about the named network
- `docker network create --driver <driver> <network name>` creates a new network using the specified driver; so far, we've only seen the `bridge` driver, for creating a linux bridge based network.
- `docker network connect <network name> <container name or id>` connects the specified container to the specified network after the container is running; the `--network` flag in `docker container run` achieves the same result at container launch.
- `docker container inspect <container name or id>` yields, among other things, information about the networks the specified container is connected to.

16 Container Port Mapping

16.1 Port Mapping at Runtime

1. Run an nginx container with no special port mappings:

```
$ docker container run -d nginx
```

   nginx stands up a landing page at <ip>:80; try to visit this at your host or container's IP, and it won't be visible; no external traffic can make it past the linux bridge's firewall to the nginx container.

2. Now run an nginx container and map port 80 on the container to port 5000 on your host using the -p flag; also map port 8080 on the container to port 9000 on the host:

```
$ docker container run -d -p 5000:80 -p 9000:8080 nginx
```

   Note that the syntax is: `-p [host-port]:[container-port]`.

3. Verify the port mappings with the `docker container port` command

```
$ docker container port <container id>
```

4. Visit your nginx landing page at `<host ip>:5000`.

16.2 Exposing Ports from the Dockerfile

1. In addition to manual port mapping, we can expose some ports in a Dockerfile for automatic port mapping on container startup. In a fresh directory, create a Dockerfile:

```
1 FROM nginx
2
3 EXPOSE 80 8080
```

2. Build your image with `docker image build -t my_nginx .`. Use the `-P` flag when running to map all ports mentioned in the `EXPOSE` directive:

```
$ docker container run -d -P my_nginx
```

3. Use `docker container ls` or `docker container port` to find out what host ports were used, and visit your nginx landing page at the appropriate ip/port.

16.3 Conclusion

In this exercise, we saw how to explicitly map ports from our container's network stack onto ports of our host at runtime with the `-p` option to `docker container run`, or more flexibly in our Dockerfile with `EXPOSE`, which will result in the listed ports inside our container being mapped to random available ports on our host.

17 Starting a Compose App

In a microservice-oriented design pattern, labor is divided among modular, independent services, many of which cooperate to form a full application. Docker images and containerization naturally enable this paradigm by using images to define services, and containers to correspond to instances of those services. In order to be successful, each running container will need to be able to interact; Docker Compose facilitates these interactions on a single host. In this example, we'll explore a toy example of such an application orchestrated by Docker Compose.

17.1 Prepare Service Images

1. Download the Dockercoins app from github:

```
$ git clone https://github.com/docker-training/orchestration-workshop.git
$ cd orchestration-workshop
$ git fetch origin 17.3
$ git checkout 17.3
```

   This app consists of 5 services: a random number generator `rng`, a hasher, a backend worker, a redis queue, and a web frontend. Each service has a corresponding image, which we will build and push to Docker Store for later use (if you don't have a Docker Store account, make a free one first at https://store.docker.com).

2. Log into your Docker Store account from the command line:

```
$ docker login
```

3. Build and push the images corresponding to the `rng`, `hasher`, `worker` and `web` services. For `hasher1, this looks like (from theorchestration-workshop/dockercoins' folder you just cloned from GitHub):

```
$ docker image build -t <username>/dockercoins_hasher:1.0 hasher
$ docker image push <username>/dockercoins_hasher:1.0
```

4. Look in `docker-compose.yml`, and change all the `image` values to have your Docker Store id instead of `user`; now you'll be able to use this Compose file to set up your app on any machine that can reach Docker Store.

17.2 Start the App

1. Stand up the app (you may need to install Docker Compose first, if this didn't come pre-installed on your machine; see the instructions at https://docs.docker.com/compose/install/):

```
$ docker-compose up
```

2. Logs from all the running services are sent to STDOUT. Let's send this to the background instead; kill the app with `CTRL+C`, sending a `SIGTERM` to all running processes; some exit immediately, while others wait for a 10s timeout before being killed by a subsequent `SIGKILL`. Start the app again in the background:

```
$ docker-compose up -d
```

3. Check out which containers are running thanks to Compose:

```
$ docker-compose ps
```

4. Compare this to the usual `docker container ls`; at this point, it should look about the same. Start any other container with `docker container run`, and try both `ls` commands again. Do you notice any difference?

5. With all five containers running, visit Dockercoins' web UI at `<IP>:8000`. You should see a chart of mining speed, around 4 hashes per second.

17.3 Viewing Logs

1. See logs from a Compose-managed app via:

```
$ docker-compose logs
```

2. The logging API in Compose follows the main Docker logging API closely. For example, try following the tail of the logs just like you would for regular container logs:

```
$ docker-compose logs --tail 10 --follow
```

   Note that when following a log, `CTRL+S` and `CTRL+Q` pauses and resumes live following.

17.4 Conclusion

In this exercise, you saw how to start a pre-defined Compose app, and how to inspect its logs.

18 Scaling a Compose App

18.1 Scaling a Service

Any service defined in our `docker-compose.yml` can be scaled up from the Compose API; in this context, 'scaling' means launching multiple containers for the same service, which Docker Compose can route requests to and from.

1. Scale up the `worker` service in our Dockercoins app to have two workers generating coin candidates, while checking the list of running containers before and after:

```
$ docker-compose ps
$ docker-compose scale worker=2
$ docker-compose ps
```

2. Look at the performance graph provided by the web frontend; the coin mining rate should have doubled. Also check the logs using the logging API we learned in the last exercise; you should see a second `worker` instance reporting.

18.2 Scale Nonlinearity

1. Try running `top` to inspect the system resource usage; it should still be fairly neglegible. So, keep scaling up your workers:

```
$ docker-compose scale worker=10
$ docker-compose ps
```

2. Check your web frontend again; has going from 2 to 10 workers provided a 5x performance increase? It seems that something else is bottlenecking our application; any distributed application such as Dockercoins needs tooling to understand where the bottlenecks are, so that the application can be scaled intelligently.

3. Look in `docker-compose.yml` at the `rng` and `hasher` services; they're exposed on host ports 8001 and 8002, so we can use `httping` to probe their latency:

```
$ httping -c 10 localhost:8001
$ httping -c 10 localhost:8002
```

   `rng` on port 8001 has the much higher latency, suggesting that it might be our bottleneck. A random number generator based on entropy won't get any better by starting more instances on the same machine; we'll need a way to bring more nodes into our application to scale past this, which we'll explore in the next unit on Docker Swarm.

4. For now, shut your app down:

```
$ docker-compose down
```

18.3 Conclusion

In this exercise, we saw how to scale up a service defined in our Compose app using the `scale` API object. Also, we saw how crucial it is to have detailed monitoring and tooling in a microservices-oriented application, in order to correctly identify bottlenecks and take advantage of the simplicity of scaling with Docker.

