UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

INSTITUTO DE INFORMÁTICA

CMP193/INF01015 – Gerência e Aplicações em Redes

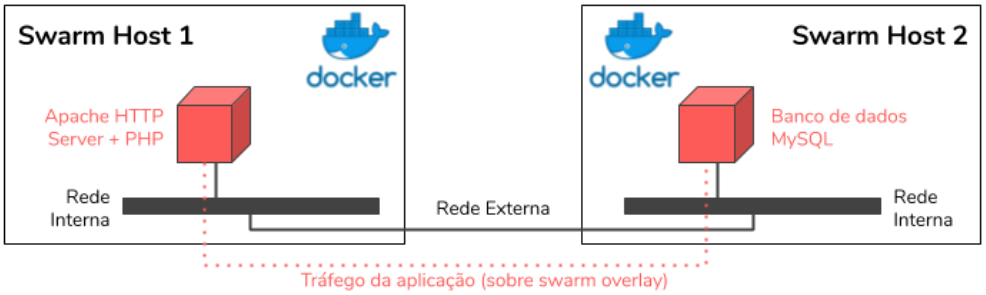
PROFESSORES: Juliano Wickboldt e Jéferson Campos Nobre

Segundo Trabalho Prático

Roteiro de Instalação e Configuração do Docker/Swarm

Cenário do trabalho

O objetivo geral do trabalho é implementar arquiteturas de gerenciamento de forma dinâmica através de micro-serviços (i.e. containers). Esse roteiro vai ajudar vocês a fazer a implantação básica de uma aplicação distribuída simples (i.e. Web Server + Banco de Dados), conforme a figura a seguir:



A partir desse cenário base você deve seguir os passos da apresentação contendo **"Orientações para o Segundo Trabalho"** para implementar uma arquitetura de gerenciamento em paralelo e obter dados sobre o comportamento da aplicação/serviço a fim de realizar o seu gerenciamento. Para configurar esse cenário utilizamos Docker em modo Swarm, instalado nas mesmas máquinas virtuais que utilizamos nas outras atividades práticas da disciplina (Linux Mint 19.2 XFCE). Vamos usar duas instâncias da máquina virtual para simular dois hosts Docker independentes. Essas duas máquinas virtuais podem rodar na mesma máquina física (se os recursos permitirem) ou em máquinas físicas diferentes, contanto que estejam conectadas em modo *bridge* à mesma rede local. Os detalhes de configuração do ambiente estão abaixo.

Antes de começar

Baixe a máquina virtual da disciplina (se você não fez isso ainda em atividades anteriores) fornecida pelo professor e importe ela no seu VirtualBox local. **Não esqueça de reinicializar o MAC address** das interfaces de rede e de colocá-las em modo bridge.

Depois de iniciar cada máquina virtual, atualize o sistema:

```
$ sudo apt update
$ sudo apt upgrade
```

Sugiro instalar o CD de adicionais para convidado do VirtualBox para facilitar o uso da interface gráfica (permitir copiar e colar, por exemplo).

No geral, basta inserir o CD pelo menu Dispositivos do VirtualBox. Ele deve montar um CD na pasta **/media/gerencia/VBOX...**

Dentro dessa pasta, basta executar o arquivo VBoxLinuxAdditions.run para instalar os adicionais.

```
$ sudo ./VBoxLinuxAdditions.run
```

Depois será necessário reiniciar a máquina virtual. No menu Dispositivos você poderá habilitar a área de transferência compartilhada que vai permitir copiar e colar texto para dentro da máquina virtual.

Docker crash course

O Docker é uma plataforma para criar serviços e aplicações baseadas em containers tornando sua implantação mais flexível, escalável, etc. Fundamentalmente, um container não passa de um processo em execução, com alguns recursos adicionais de encapsulamento aplicados a ele para mantê-lo isolado do host e de outros containers. Um dos aspectos mais importantes do isolamento de containers é que cada container interage com seu próprio sistema de arquivos privado; esse sistema de arquivos é fornecido por uma imagem do Docker. Uma imagem inclui tudo o necessário para executar uma aplicação - o código fonte ou binários, ambiente de execução, dependências e quaisquer outros objetos do sistema de arquivos necessários. Nesse tutorial vamos explorar apenas alguns aspectos do Docker para conseguir montar o setup básico acima. Leia mais sobre os principais conceitos envolvidos no Docker em https://docs.docker.com/get-started/.

Configuração básica do Docker

Primeiramente, vamos começar instalando os pacotes do Docker via apt:

```
$ sudo apt update
$ sudo apt upgrade
$ sudo apt install docker.io docker-compose
```

Para não precisar usar **`sudo`** sempre que formos rodar o comando **`docker`**, vamos adicionar o usuário atual (**`$USER`**) a um grupo especial chamado docker e usar o comando newgrp para ativar imediatamente o grupo docker para o usuário logado (sem precisar reiniciar a sessão do Linux):

```
$ sudo usermod -aG docker $USER
$ newgrp docker
```

Consideração de segurança: o docker utiliza alguns acessos privilegiados no host para gerenciar os containers, então procure incluir no grupo docker apenas usuário que realmente precisarem utilizar o sistema (de preferência aqueles que já estariam no grupo de administradores de qualquer forma).

Para ver mais informações sobre a versão do Docker que foi instalada você pode usar o **`apt show`**:

```
$ apt show docker.io
Package: docker.io
Version: 18.09.7-0ubuntu1~18.04.4
...
```

A versão do Docker que estamos usando nesse roteiro é 18.09.7. Além do Docker estar instalado, precisamos verificar que o serviços **`docker`** e **`containerd`** estão rodando. Podemos fazer isso via o comando **`service`**:

```
$ service docker status
```

```
● docker.service - Docker Application Container Engine
   Loaded: loaded (/lib/systemd/system/docker.service; disabled; vendor preset: enabled)
   Active: active (running) since Thu 2019-10-17 16:08:41 -03; 8min ago
     Docs: https://docs.docker.com
 Main PID: 2461 (dockerd)
    Tasks: 9
   CGroup: /system.slice/docker.service
           └─2461 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
...
$ service containerd status
● containerd.service - containerd container runtime
   Loaded: loaded (/lib/systemd/system/containerd.service; enabled; vendor preset: enabled)
   Active: active (running) since Thu 2019-10-17 16:08:37 -03; 8min ago
     Docs: https://containerd.io
 Main PID: 2268 (containerd)
    Tasks: 9
   CGroup: /system.slice/containerd.service
           └─2268 /usr/bin/containerd
...
```

Com o docker instalado podemos então começar a interagir com a ferramenta e lançar containers. Para lançar um container de teste podemos usar o seguinte comando:

```
$ docker run hello-world
```

O que esse comando faz na prática é:

1. O cliente docker entra em contato com o daemon docker.dockerd.
2. O daemon do docker baixa a imagem "hello-world" do Docker Hub (https://hub.docker.com/).
3. O daemon do docker cria um novo container a partir dessa imagem que apenas executa um binário que produz uma mensagem (parecida com essa).
4. O daemon do docker transmite essa mensagem de volta para o cliente docker, que a envia ao seu terminal.

Agora rodando o comando **`docker ps`** podemos verificar os containers existentes nessa máquina host:

```
$ docker ps --all
CONTAINER ID  IMAGE        COMMAND   CREATED         STATUS                     PORTS   NAMES
ba21d7d6a119  hello-world  "/hello"  13 seconds ago  Exited (0) 10 seconds ago          friendly_liskov
```

A opção **`--all`** é necessária apenas para listar todos os containers (rodando ou parados). No caso, devemos ver pelo menos o nosso container “hello-world” como mostrado acima. O container está parado (status exited) porque ele apenas executou o binário **`/hello`** e terminou.

Se você chegou até esse ponto e tudo funcionou, pode testar algo mais ousado e lançar um container com uma imagem de Ubuntu para ter mais opções e funcionalidades dentro do container para testar:

```
$ docker run -it ubuntu bash
```

O comando acima faz com que você baixe a imagem (deve demorar um pouco) do Ubuntu a partir do Docker Hub e lance um processo **`bash`** encapsulado dentro do container criado. A opção **`-it`** vai criar um console interativo para que você possa interagir com o novo processo por linha de comando como se estivesse em uma janela de terminal. Depois de rodar o comando seu terminal deve alterar o prompt para **`root@...`**, o que indica que agora você está usando um usuário diferente (root) em um outro host que no caso é o container. Você pode utilizar comandos normais do Linux como **`ps`** para listar processos, **`ls`** para listar arquivos e diretórios, **`apt`** para instalar pacotes, etc.

Por exemplo, como a imagem do Ubuntu para Docker é mínima (tem o mínimo de pacotes instalados por padrão), muitos dos utilitários para gerenciamento da rede não vêm instalados por padrão. Podemos instalar o comando **`ip`** disponível no pacote **`iproute2`** para verificar as configurações de rede.

```
$ apt update
$ apt install iproute2
```

Rodando agora o comando ip podemos listar as interfaces, endereços IP associados a elas e as rotas disponíveis dentro do contexto do container:

```
$ ip link
$ ip addr
$ ip route
```

Você pode testar os mesmo comandos acima diretamente no host e deve verificar que as informações e configurações de rede são de fato diferentes entre o host e o container. O resultado desses comandos deve dá a impressão que você está rodando dentro de um outro host com sistema operacional independente, mas na prática não é isso que acontece. Se você ficar curioso para saber como isso é possível, assista esse vídeo que explica o conceito de namespaces:

Containers unplugged: Linux namespaces - Michael Kerrisk:
https://www.youtube.com/watch?v=0kJPa-1Fuol

Containers unplugged: Linux namespaces - Michael Kerrisk
NDC Conferences • 567 visualizações • há 3 semanas
Linux namespaces are a resource isolation technique. Each namespace type wraps some global system resource in an ...

Esse outro vídeo também fala sobre o assunto e vai um pouco além, tratando especificamente de como o docker usa essas funcionalidades.

Cgroups, namespaces, and beyond: what are containers made from?
https://www.youtube.com/watch?v=sK5i-N34im8

Cgroups, namespaces, and beyond: what are containers made from?
Docker • 93 mil visualizações • há 3 anos
with Jérôme Petazzoni, Tinkerer Extraordinaire, Docker Linux containers are different from Solaris Zones or BSD Jails: they use ...

Bom, vamos em frente. Você pode usar o comando **`exit`** para sair do terminal do container e voltar ao terminal do host. Usando o comando **`docker ps --all`** você vai perceber que o container criado com a imagem de Ubuntu também está parado. Isso ocorre porque o comando exit termina o processo bash que havia sido iniciado pelo comando **`docker run`**, nesse caso era o único comando rodando no container. Para lançar um container em segundo plano (detached) você pode usar a opção **`--detach`** do comando **`docker run`**. Depois para conectar ao console do container você pode usar o comando **`docker attach`**. Mais detalhes sobre como conectar e desconectar ao console de containers em https://docs.docker.com/engine/reference/commandline/attach/.

As imagens baixadas ficam salvas localmente para uso futuro. Você pode ver as imagens armazenadas localmente com o comando:

```
$ docker images
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              latest              2ca708c1c9cc        3 weeks ago         64.2MB
hello-world         latest              fce289e99eb9        9 months ago        1.84kB
```

Você vai encontrar mais informações sobre manipulação de imagens em https://docs.docker.com/engine/reference/commandline/image/. Você pode criar suas próprias imagens a partir de arquivos *Dockerfile*. Falaremos disso depois, mas caso queira consultar a documentação completa está em https://docs.docker.com/engine/reference/builder/.

Compondo uma aplicação completa baseada em micro-serviços

Como forma de ilustrar o conceito de composição de uma aplicação em micro-serviços, vamos usar o Wordpress como exemplo nesse tutorial. O Wordpress é uma ferramenta para criação de sites ou blogs que pode ser utilizada como serviço diretamente no site (http://wordpress.com) ou baixada (de http://wordpress.org) e instalada localmente. O problema é que o Wordpress em si é uma aplicação escrita em PHP que precisa ser carregada a partir de um servidor Web, como o Apache HTTP Server (https://httpd.apache.org/). Além disso, o Wordpress quando instalado localmente também requer um banco de dados, por exemplo, MySQL (https://www.mysql.com/) para armazenar as informações do site. Todo esse ambiente pode ser bastante complexo de instalar, configurar e manter funcionando, mas quando composto como micro-serviços no Docker essa tarefa fica relativamente simples.

Vamos, primeiramente, criar um diretório **`wp`** (de WordPress) e um arquivo dentro desse diretório com nome **`docker-compose.yml`**. Esse arquivo será escrito em formato de um *Compose file* que descreve uma aplicação completa e todas as suas partes (micro-serviços, volumes, configurações de rede e outras coisas). Podemos colocar o seguinte conteúdo no arquivo **`docker-compose.yml`**:

```
version: '3.3'

services:
   db:
     image: mysql:5.7
     volumes:
       - db_data:/var/lib/mysql
     restart: always
     environment:
       MYSQL_ROOT_PASSWORD: somewordpress
       MYSQL_DATABASE: wordpress
       MYSQL_USER: wordpress
       MYSQL_PASSWORD: wordpress

   wordpress:
     depends_on:
       - db
     image: wordpress:latest
     ports:
       - "8000:80"
     restart: always
     environment:
       WORDPRESS_DB_HOST: db:3306
       WORDPRESS_DB_USER: wordpress
       WORDPRESS_DB_PASSWORD: wordpress
       WORDPRESS_DB_NAME: wordpress

volumes:
    db_data: {}
```

Com base neste arquivo, podemos levantar a aplicação completa. usando o comando **`docker-compose`**

up a partir da mesma pasta (**wp**) onde está salvo o arquivo YAML. O comando abaixo, pode demorar alguns minutos para executar (baixar as imagens é o que demora mais), então nesse meio tempo você pode seguir lendo o detalhamento do que está acontecendo logo abaixo:

```
$ cd wp/
$ docker-compose up -d
Creating network "wp_default" with the default driver
Creating volume "wp_db_data" with default driver
Pulling db (mysql:5.7)...
5.7: Pulling from library/mysql
...
Status: Downloaded newer image for mysql:5.7
Pulling wordpress (wordpress:latest)...
latest: Pulling from library/wordpress
...
Status: Downloaded newer image for wordpress:latest
Creating wp_db_1 ... done
Creating wp_wordpress_1 ... done
```

O arquivo **docker-compose.yml** segue um formato padrão YAML que é bastante auto-explicativo e legível, mas se tiver dúvidas você pode consultar detalhes da sintaxe em https://yaml.org/. No nosso arquivo **docker-compose.yml**, a seção **services** especifica dois micro-serviços, que na prática serão implantados como containers, um chamado **db** e outro **wordpress**. O serviço **db** utilizará a imagem do MySQL (**mysql:5.7**) que será o banco de dados da nossa aplicação. O serviço **wordpress** utilizará a imagem mais atual (**wordpress:latest**) do Wordpress disponível no Docker Hub que inclui o código da aplicação Wordpress em PHP e mais o servidor Web padrão que é o Apache. Outra informação relevante do serviço wordpress é a porta (TCP) que ele opera. Dentro da especificação do serviço, podemos ver na seção **ports** que ele mapeia a porta 8000 do host para a porta 80 do container. Isso significa que ao acessar a porta 8000 da máquina host (onde está instalado o Docker) a requisição será redirecionada para dentro do container na porta 80. As outras informações de ambiente (**environment**) são dependentes das imagens que estão sendo implantadas. Não confunda esse arquivo do docker compose em YAML com o *Dockerfile*. O *Dockerfile* define as imagens enquanto que o *Compose file* define os serviços implantados a partir das imagens. Mais informações sobre as opções das imagens do Wordpress (https://hub.docker.com/_/wordpress) e do MySQL (https://hub.docker.com/_/mysql) podem ser encontradas diretamente no Docker Hub.

Por fim, o arquivo ainda define um volume (seção **volumes**) chamado **db_data**. Esse volume será implantado juntamente com os containers para armazenar dados de forma persistente. Quando um container é destruído, todos os dados armazenados nele são também destruídos. Isso acontece comumente, por exemplo, quando as imagens são atualizadas e precisamos refazer o deploy da aplicação toda, por isso a importância do volume. Esse volume definido será utilizado para armazenar os arquivos do banco de dados MySQL de forma persistente mesmo que os containers precisem ser destruídos e reimplantados, conforme configurado no serviço **db** na seção **volumes**. Nesse primeiro exemplo não fazemos nenhuma definição de rede, mas isso é possível a partir da criação de uma seção **networks** dentro do mesmo arquivo. Mais detalhes sobre o que pode ser incluído em um *Docker Compose file* podem ser encontrados em https://docs.docker.com/compose/compose-file/.

O comando **docker-compose up** lê, por padrão, arquivo **docker-compose.yml** existente no diretório local. Alternativamente, você pode usar a opção **-f** para passar um ou mais arquivos YAML para o **docker-compose**. No caso de múltiplos arquivos de entrada o **docker-compose** é capaz de combinar as configurações em uma aplicação única. A opção **-d** (de *detach*) serve para levantar a aplicação em background, já que serão vários containers iniciados e na prática não precisaremos interagir com eles via console. Depois, se for necessário, sempre podemos usar o comando **docker attach** para conectar no console de um container qualquer rodando, seja ele parte de uma aplicação ou não.

Se tudo correu bem na implantação dos containers da aplicação, você deve conseguir ver eles rodando através do comando **`docker-compose ps`** ou mesmo com o comando **`docker ps`**:

```
$ docker-compose ps
        Name                      Command               State          Ports
-------------------------------------------------------------------------------------
wp_db_1          docker-entrypoint.sh mysqld      Up      3306/tcp, 33060/tcp
wp_wordpress_1   docker-entrypoint.sh apach ...   Up      0.0.0.0:8000->80/tcp
```

O comando **`docker-compose ps`** vai listar apenas os containers que fazem parte de uma aplicação implantada com **`docker-compose up`**. O comando **`docker ps`** lista todos os containers do host, independente de como foram criados.

Podemos ver pela saída acima que dois containers estão ativos (State: Up), um deles é chamado **`wp_db_1`** e o outro **`wp_wordpress_1`**. O nome da aplicação é **`wp`**, por padrão, esse é o mesmo nome do diretório onde foi salvo o arquivo **`docker-compose.yml`**. Esse nome é utilizado como prefixo (**`wp_`**) para todos os elementos associados à aplicação (containers, redes, volumes, etc.). Você pode usar a opção **`--project-name`** do **`docker-compose`** para especificar um nome diferente para a aplicação, isso pode ser útil para implantar várias aplicações a partir do mesmo arquivo YAML.

Além dos dois containers, também especificamos um volume no arquivo YAML. Para verificar se o volume foi criado, podemos utilizar o comando **`docker volume ls`**:

```
$ docker volume ls
DRIVER              VOLUME NAME
local               fb9afdf37c46067028d9e306a1fa94317610a8319652e50d6792a3d3da2e9ada
local               wp_db_data
```

Na saída desse comando podemos ver que um volume de nome **`wp_db_data`** foi criado. O outro volume da lista foi criado automaticamente (com nome aleatório) e associado ao container do Wordpress (**`wp_wordpress_1`**). A configuração desse outro volume não veio do nosso arquivo YAML, mas sim da configuração da imagem do Wordpress. É possível ver quais volumes estão associados a um container com o comando **`docker inspect`**:

```
$ docker inspect wp_db_1
...
        "Mounts": [
            {
                "Type": "volume",
                "Name": "wp_db_data",
                "Source": "/var/snap/docker/common/var-lib-docker/volumes/wp_db_data/_data",
                "Destination": "/var/lib/mysql",
                "Driver": "local",
                "Mode": "rw",
                "RW": true,
                "Propagation": ""
            }
        ],
...


$ docker inspect wp_wordpress_1
...
        "Mounts": [
            {
                "Type": "volume",
                "Name": "fb9afdf37c46067028d9e306a1fa94317610a8319652e50d6792a3d3da2e9ada",
```

```
                "Source":
"/var/snap/docker/common/var-lib-docker/volumes/fb9afdf37c46067028d9e306a1fa94317610a8319652e50d6792a3d
3da2e9ada/_data",
                "Destination": "/var/www/html",
                "Driver": "local",
                "Mode": "",
                "RW": true,
                "Propagation": ""
            }
        ],
...
```

A partir dessa saída, olhando a opção **"Source"**, é possível descobrir até mesmo o local no host onde estão armazenados os dados desse volume. Se você olhar dentro da pasta indicada, você verá o mesmo conteúdo que o container acessa a partir do ponto de montagem **"Destination"**. Também é possível ver detalhes da criação da imagem do Wordpress para verificar como o volume extra foi criado:

```
$ docker image history wordpress
IMAGE          CREATED        CREATED BY                                      SIZE         COMMENT
...
<missing>      12 days ago    /bin/sh -c #(nop)  VOLUME [/var/www/html]  0B
...
```

Os arquivos fonte do Wordpress em PHP são armazenados em um volume específico montados no caminho **/var/www/html** dentro do container. Quem determina isso é a imagem do Wordpress e não o arquivo YAML da aplicação.

Bom, se você chegou até esse ponto, você pode acessar via browser o endereço IP do host na porta 8000 (se você estiver conectado diretamente no host pode usar http://localhost:8000), você deve ser redirecionado para o serviço Web (porta 80) onde está rodando o Wordpress. Isso deve carregar a interface de configuração do Wordpress, como a seguinte:

Você pode interagir normalmente com a página Web carregada, só para testar, mas não precisamos fazer nada específico em relação a ela. O mais importante é entender que ao acessar essa página você é redirecionado para o serviço HTTP rodando no container **wp_wordpress_1** que por sua vez se

comunica via uma rede interna com o banco de dados rodando no container **wp_db_1** para fazer a aplicação funcionar. Essa comunicação ocorre usando uma rede específica criada junto com a aplicação. Podemos, por exemplo, ver as redes ativas com o comando **docker network ls**.

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
6f2b4ae2af42        bridge              bridge              local
6a1738ec7f76        host                host                local
133b0db07cbf        none                null                local
daccc89f2844        wp_default          bridge              local
```

Você pode perceber pela saída do comando acima que já existem algumas redes criadas no Docker do nosso host. A rede **wp_default** foi criada junto com a aplicação, onde, por padrão, todos os containers associados a nossa aplicação foram conectados. Outros containers, criados com **docker run**, por exemplo, são conectados por padrão todos à rede **bridge**. Você pode manipular esse comportamento e conectar containers de variadas formas editando a seção **networks** o *Compose file*, no caso de containers implantados por **docker-compose**, ou passando o argumento **--network** para o comando **docker run**.

Com o comando **docker network inspect** você consegue ver todos os detalhes de uma rede qualquer, como segue:

```
$ docker network inspect wp_default
...
"Containers": {
            "c073c91b59b1f353c5e07631a004df173c0e7c0f06d552cd2d491402a8b49be1": {
                "Name": "wp_db_1",
                "EndpointID": "1c821f6ed8ee1aadcbe1295b65d16645cbb75beb5671dca718288cc48e718c3c",
                "MacAddress": "02:42:ac:15:00:02",
                "IPv4Address": "172.21.0.2/16",
                "IPv6Address": ""
            },
            "cfb655bbc0a99c6eda3c9b3479dba5fe5c5eb64e947ced0e134ee35f6c573e52": {
                "Name": "wp_wordpress_1",
                "EndpointID": "8bf4378590aa30614cbc62bc960bde49b7b6f628c23b1f8fdfba317fedb87c1f",
                "MacAddress": "02:42:ac:15:00:03",
                "IPv4Address": "172.21.0.3/16",
                "IPv6Address": ""
            }
        },
...
```

Para saber, por exemplo, quais containers estão conectados em uma rede você pode olhar a seção **"Containers"** na saída do inspect. Outra informação relevante é o *driver* utilizado para implementar um determinado "tipo" de rede. Para ver detalhes sobre todos os *drivers* de redes disponíveis ou até mesmo como criar *drivers* customizados você pode acessar a documentação completa em https://docs.docker.com/network/.

Agora podemos derrubar a aplicação usando o comando **docker-compose down**:

```
$ docker-compose down --volume
Stopping wp_wordpress_1 ... done
Stopping wp_db_1        ... done
Removing wp_wordpress_1 ... done
Removing wp_db_1        ... done
Removing network wp_default
Removing volume wp_db_data
```

Por padrão, somente os containers e as redes são removidos pelo **`docker-compose down`**. A opção **`--volume`** vai remover também os volumes associados a aplicação.

Configurando um Docker Swarm

Até aqui vimos como criar uma aplicação baseada em micro-serviços localmente em um host Docker com **`docker-compose`**. Porém, podemos fazer algo ainda mais ousado e rodar essa aplicação de fato como um sistema distribuído, onde cada micro-serviço poderá rodar em um host Docker diferente. Para poder utilizar o Docker simultaneamente em mais de um host precisamos configurar um *swarm*. Resumidamente, um *swarm* (em português enxame) consiste em vários hosts Docker que são executados no modo *swarm* e atuam ou como ***managers*** (para gerenciar associação e delegação de micro-serviços) e ***workers*** (que executam micro-serviços em um *swarm*). Um determinado host Docker pode ser um *manager*, um *worker* ou executar as duas funções. Os principais conceitos associados ao Docker Swarm estão documentados em https://docs.docker.com/engine/swarm/key-concepts/.

Para iniciar um *swarm* podemos utilizar o **`docker swarm init`** a partir da máquina que será o *manager* inicialmente.

```
$ docker swarm init
Swarm initialized: current node (...) is now a manager.

To add a worker to this swarm, run the following command:

       docker swarm join --token ... 143.54.13.NN:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

A saída do comando dará informações sobre como conectar os *workers* ou outros *managers* posteriormente. Anote esses dados para referência futura. Você sempre pode usar o comando **`docker swarm join-token (worker|manager)`** para verificar essas informações.

Para associar um *worker*, utilize uma segunda máquina (pode ser uma VM) e realize os passos básicos de instalação do Docker acima (não precisa fazer a parte de instanciar containers para testar nem a parte do compose), e depois execute o comando passando o seu token e o endereço IP do host onde foi iniciado o *swarm* no passo anterior.

```
$ docker swarm join --token ... 143.54.13.NN:2377
This node joined a swarm as a worker.
```

É importante que os dois hosts Docker, o *manager* e o *worker*, estejam conectados na mesma rede local para que o IP do *manager* seja alcançável a partir do *worker*. Se você estiver usando VMs, pode conectar as duas VMs em modo bridge na rede local ou criar uma rede isolada entre as duas VMs. Se tudo der certo você deve ver a mensagem acima como saída do comando **`docker swarm join`**. Voltando ao nodo *manager* onde você iniciou o *swarm* no passo anterior você agora pode utilizar o comando **`docker node ls`** para verificar o status dos nodos do seu Docker Swarm.

```
$ docker node ls
ID                            HOSTNAME    STATUS  AVAILABILITY    MANAGER STATUS      ENGINE VERSION
mqlxa0q41hh2e7g8m98e7fx13 *   manager     Ready   Active          Leader              18.09.7
5op25vgabb1xa3btraam0ye90     worker1     Ready   Active                              18.09.7
```

No exemplo acima, temos um host Docker como *manager* (hostname manager) e um host como *worker* (hostname worker1). No caso, o único manager também é o líder (Leader) atual do Swarm, responsável por delegar as tarefas aos demais. Quando mais de um manager está ativo, eles podem eleger o líder

dinamicamente ou um novo líder pode ser designado quando um manager falha, etc. Nós não vamos focar em configurar essas questões de resiliência aqui, mas se quiser entender mais profundamente o funcionamento desses mecanismos, a documentação completa do Docker Swarm está em https://docs.docker.com/engine/swarm/.

Implantando uma aplicação baseada em micro-serviços com Docker Swarm

Quando executamos o Docker no modo *swarm*, podemos usar o comando **`docker stack deploy`** para implantar uma aplicação completa de forma distribuída sobre o *swarm*. A partir do nodo *manager*, você pode acessar o mesmo diretório **`wp`** que usamos no exemplo anterior e fazer a implantação da aplicação (ou stack) sobre o swarm assim:

```
$ docker stack deploy --compose-file docker-compose.yml wpstack
Ignoring unsupported options: restart

Creating network wpstack_default
Creating service wpstack_db
Creating service wpstack_wordpress
```

Pela saída do comando vemos que foi criada uma rede **`wpstack_default`** e dois serviços/containers **`wpstack_db`** e **`wpstack_wordpress`**. Usando o comando **`docker stack ls`** podemos ver as aplicações implantadas no *swarm*.

```
$ docker stack ls
NAME                SERVICES            ORCHESTRATOR
wpstack             2                   Swarm
```

Nossa aplicação/stack, cujo nome é **`wpstack`**, possui dois serviços rodando sobre o *swarm*. Podemos ver o status de cada serviço usando o comando **`docker stack services`**:

```
$ docker stack services wpstack
ID            NAME                MODE        REPLICAS  IMAGE               PORTS
bajq9b8f4lix  wpstack_db          replicated  1/1       mysql:5.7
weqejn2cyf65  wpstack_wordpress   replicated  1/1       wordpress:latest    *:8000->80/tcp
```

Na lista de serviços podemos perceber que os dois serviços (**`wpstack_db`** e **`wpstack_wordpress`**) possuem apenas uma réplica cada e ambos já estão com as réplicas ativas (**`1/1`**). Pode ser que no seu caso alguma das réplicas ainda não esteja (**`0/1`**), isso pode levar algum tempo para que os serviços sejam levantados por conta do download das imagens. Se você seguiu os passos acima para levantar o serviço usando o docker-compose, as imagens provavelmente já estarão em cache no host onde isso foi feito. No entanto, o host worker que foi adicionado por último, provavelmente terá que baixar as imagens também para poder subir os serviços localmente e isso pode levar alguns minutos. Usando o comando **`docker stack ps`** você consegue ver informações mais detalhadas de cada container implantado:

```
$ docker stack ps wpstack
ID            NAME                  IMAGE             NODE     DESIRED STATE CURRENT STATE  ERROR  PORTS
x8gb3avqr0d4  wpstack_db.1          mysql:5.7         manager  Running       Running about an hour ago
hbuslv1l13pt  wpstack_wordpress.1   wordpress:latest  worker1  Running       Running 45 minutes ago
```

Nessa lista, vemos também dois containers, porque cada serviço possui apenas uma réplica. Podemos também ver que os serviços estão rodando em hosts diferentes, um deles (Wordpress) roda no host **`worker1`** e o outro (MySQL) no host **`manager`**. Independente de onde os serviços são posicionados no *swarm*, a comunicação entre eles deve sempre fluir graças ao mecanismo de configuração de redes do Docker. Além disso, também é possível acessar a porta externa (8000) do serviço do Wordpress a partir

de qualquer nodo do *swarm*, independente de onde o serviço for instanciado. Você pode ver os detalhes das redes atualmente existentes no Docker com o comando **`docker network ls`**:

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
7efaff6c360e        bridge              bridge              local
8acf750df0b2        docker_gwbridge     bridge              local
6635bed7ce33        host                host                local
j54t0sfci9uz        ingress             overlay             swarm
28c5a0e4fc99        none                null                local
jnrv91ipafqk        wpstack_default     overlay             swarm
```

Você vai perceber que existem agora duas redes no escopo do *swarm* (**`ingress`** e **`wpstack_default`**). A rede **`wpstack_default`** foi criada pelo Docker para conectar os serviços do nosso stack, enquanto que a rede **`ingress`** é utilizada para expor as portas dos serviços para o exterior. Existem várias opções de configurações de rede possíveis com o swarm, você pode ler mais sobre isso em https://docs.docker.com/network/overlay/.

O *swarm* também suporta múltiplas réplicas de um serviço assim como ele controla o status e tenta recuperar os serviços em caso de falhas. Nós não vamos cobrir esses aspectos (apesar de interessantes) nesse roteiro, mas a documentação completa para implantação de stacks sobre *swarms* Docker está em https://docs.docker.com/engine/swarm/stack-deploy/.

Por fim, se quiser remover seu stack você pode usar o comando **`docker stack rm`**:

```
$ docker stack rm wpstack
Removing service wpstack_db
Removing service wpstack_wordpress
Removing network wpstack_default
```

E agora?

Bom, agora você pode começar a “containerizar” a sua arquitetura de gerenciamento e planejar sua implantação junto com o serviço.

Estude sobre Dockerfile (https://docs.docker.com/engine/reference/builder/) para preparar as imagens dos componentes da arquitetura de gerenciamento que seu grupo precisará implantar. Procure por imagens base já pré-prontas no http://hub.docker.com.

Estude sobre como controlar namespaces para compartilhar as informações necessárias entre os containers de aplicação e de gerenciamento: https://docs.docker.com/engine/docker-overview/#namespaces.

Estude sobre configurações de redes no Docker (https://docs.docker.com/network/) para fazer o tráfego da aplicação fluir até os componentes da arquitetura de gerenciamento onde for necessário. É possível que vocês precisem compreender ou até mesmo criar drivers de rede.

Estude opções avançadas de placement e replicas em Compose files que podem ser úteis: https://docs.docker.com/compose/compose-file/#placement.

Boa sorte!