

**EBOOK**

# **CONTAINERS: DOCKER E KUBERNETES**

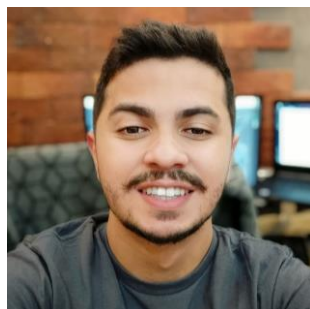


# Sumário

UNIDADE 1 – Entendendo o Docker .....	6
O que é o Docker? .....	6
Quais são os benefícios que o Docker pode oferecer? .....	7
O que são Containers? .....	7
O que são Imagens no contexto do Docker? .....	9
Como configurar um ambiente Docker?.....	10
• Linux (Ubuntu).....	10
• Windows .....	11
• MacOS.....	15
Considerações Finais.....	16
UNIDADE 2 - Utilizando o Docker.....	17
Executando nosso primeiro Container .....	17
Principais comandos do Docker .....	20
• docker run .....	20
• docker ps ou docker container ls.....	21
• docker images ou docker image ls .....	22
• docker stop e docker start.....	22
• docker attach .....	23
• docker logs.....	23
• docker inspect .....	23
• docker rm .....	24
• docker search.....	24
Considerações Finais.....	26
UNIDADE 3 - DockerFile e Docker Hub .....	27
O Docker Hub .....	27
O Dockerfile.....	27
A estrutura do Dockerfile .....	28
Os principais comandos do Dockerfile .....	28
• FROM.....	28
• CMD.....	28
• RUN .....	29

• EXPOSE .....	29
• COPY .....	29
• ENTRYPOINT .....	30
Como utilizar o Dockerfile? .....	30
Enviando a Imagem para o Docker Hub.....	33
Considerações Finais.....	35
UNIDADE 4 - Criando um projeto com o Docker .....	37
O Docker Compose .....	37
Criando a Aplicação .....	37
Considerações Finais.....	47
UNIDADE 5 - Kubernetes.....	48
Kubernetes .....	48
Minikube.....	49
Kubernetes na prática.....	51
Replication Controller.....	53
Services.....	57
Considerações Finais.....	62

## Apresentação



Olá, meu nome é Gabriel Kirsten Menezes.

Eu irei ministrar essa disciplina que irá abordar o funcionamento e utilização de Containers, passando por Docker e indo até a orquestração de containers através da utilização do Kubernetes.

Eu trabalho com desenvolvimento de software, sou formado em Engenharia de Computação pela Universidade Católica Dom Bosco, tenho mestrado em Ciências da Computação pela Universidade Federal de Mato Grosso do Sul. Estou envolvido estudando e trabalhando com a área de computação faz 8 anos. Tenho um pé na área acadêmica, mas também adoro desenvolver softwares profissionalmente. Sou entusiasta nas áreas de Computação Distribuída, Machine Learning, Inteligência Artificial e Visão Computacional. Sinta-se à vontade para entrar em contato comigo e bater um papo sobre tecnologia e carreira.

Essa disciplina é de extrema importância para quem almeja as melhores vagas nas melhores empresas. Apesar de o Docker ser uma tecnologia relativamente nova, ele é largamente empregado em empresas de pequeno a grande porte pois facilita MUITO o fluxo de trabalho para as equipes. O Docker não é importante somente para quem quer ser responsável pela infraestrutura onde o software é executado em produção, mas é importante também para os desenvolvedores de software que irão codificar uma solução que funciona em um ecossistema em que o Docker é empregado.

Eu espero que essa disciplina atinja as suas expectativas e estou aberto sempre a melhorar, por isso disponibilizo a você esse formulário onde é possível fornecer um feedback anonimamente dizendo o que você mais gostou e no que podemos melhorar:

<https://forms.gle/u4yo2qS7qtS9Eu8aA>

Fique à vontade também para me seguir em minhas redes sociais:

- Github: <https://github.com/gabrielkirsten>
- Instagram: <https://www.instagram.com/gabrielkirsten/>
- LinkedIn: <https://www.linkedin.com/in/gabrielkirsten>
- Website: <http://gabrielkirsten.me>
- Youtube: <https://www.youtube.com/gabrielkirsten>

É um prazer poder contribuir com o seu desenvolvimento em sua carreira.

Desejo a você um ótimo aprendizado.

Grande abraço.

## UNIDADE 1 – Entendendo o Docker

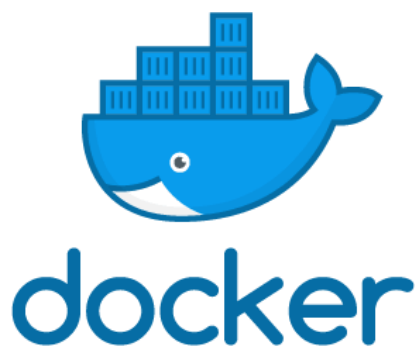
Esse capítulo fornecerá as informações necessárias para o entendimento do Docker, mesmo que já conheça a ferramenta peça que leia todo o conteúdo com atenção. Os fundamentos apresentados nesses capítulos são requisitos para o entendimento dos próximos capítulos.

Os objetivos deste capítulo são:

- Entender o que é o Docker.
- Entender o que são Containers.
- Aprender a como configurar o ambiente Docker.
- Entender o que são imagens.

### O que é o Docker?

Docker é uma ferramenta com o objetivo de tornar a tarefa de configurações de ambientes para a execução de aplicações mais simples, tudo isso através de **Containers** (será abordada a definição de Containers posteriormente). O Docker revolucionou a maneira em que um software é desenvolvido, testado, entregue e executado.



O Docker atingiu o seu sucesso rapidamente, oferecendo uma solução que facilita o fluxo de trabalho de diversas formas, ele tem como seu principal resultado a redução do tempo que uma aplicação leva desde a construção do código até a sua execução em produção.

O Docker não é somente uma ferramenta de virtualização (como muitos acreditam) e tão pouco é uma ferramenta de gerência de configurações, ele não resolve um problema específico, porém é uma ferramenta que resolve uma ampla variedade de desafios encontrados no dia-a-dia de uma empresa que trabalha com desenvolvimento de software, ele é muito mais do que aparenta ser em sua superfície.

## **Quais são os benefícios que o Docker pode oferecer?**

O Docker é uma das tecnologias mais promissoras que surgiram na última década. Muitas empresas encaram o Docker como um pré-requisito em seus processos de seleções para novos funcionários, e além disso, o Docker é encontrado tanto em empresas de pequeno porte como de grande porte.

Quando analisado de maneira mais profunda o Docker apresenta uma solução que reduz a complexibilidade na comunicação entre times que trabalham com desenvolvimento de software, isso foi o que levou a adoção do Docker em larga escala tão rapidamente. A complexidade de comunicação é minimizada devido a abstração da camada de infraestrutura onde o software é executado em produção para os desenvolvedores de software.

A infraestrutura de cada empresa que produz software possui uma parametrização complexa, sendo que isso leva a necessidade de os desenvolvedores entender muitos aspectos fora de seu escopo de trabalho, essas especificações não agregam valor para as equipes de desenvolvimento.

O Docker traz a agilidade da equipe de desenvolvimento poder escrever o seu código, testar e implantar uma nova versão sem a necessidade da intervenção do time de operação, reduzindo o tempo que leva desde uma aplicação ser escrita até entrar em produção. Da mesma forma o time de operações que gerencia a infraestrutura pode atualizar as configurações do host que roda o Docker diminuindo a necessidade de mobilizar a equipe de desenvolvimento. Neste ponto o Docker é capaz de isolar o ambiente de desenvolvimento e de operações reduzindo a tempo de trabalho para a execução das mesmas tarefas.

Os contêineres que o Docker oferece também são ótimos para equipes que trabalham com fluxos de integração contínua e entrega contínua (o famoso CI/CD). Pois facilitam a automatização e configuração da entrega de um software. Muitas ferramentas de CI/CD oferecem plugins para a fácil integração com o Docker.

## **O que são Containers?**

Eu inicio essa sessão dizendo: “Containers não são máquinas virtuais”. Essa é uma afirmação que o próprio Docker tenta desmistificar. Ambos são utilizados para facilitar a configuração e implantação de um ambiente de produção porém trabalham com conceitos diferentes.

A imagem a seguir apresenta as diferenças entre uma máquina virtual e um container. Uma máquina virtual possui a sua infraestrutura em uma camada de mais baixo nível,



no nível superior existe uma camada de *Hypervisor*<sup>1</sup>, responsável por hospedar diversos sistemas operacionais, cada sistema operacional é considerado uma máquina virtual, que possui sua camada de hardware virtualizada, porém compartilham o mesmo hardware físico. Você pode rodar uma ou mais aplicações em cada máquina virtual.

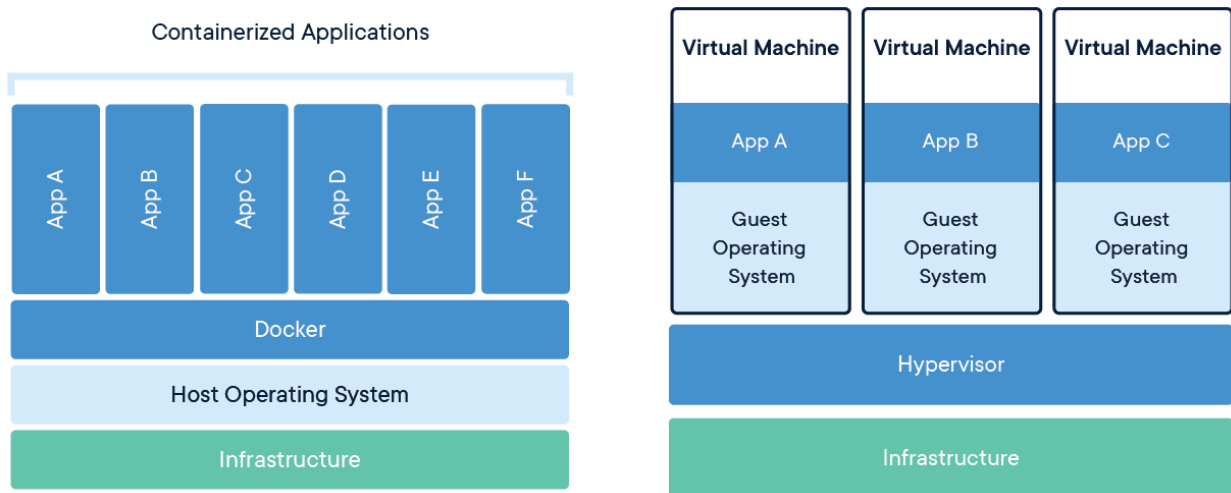


Figura que representa uma comparação entre a utilização do Docker (esquerda) e de Máquinas Virtuais (direita). Disponível em: <https://www.docker.com/resources/what-container>

Quando comparado às máquinas virtuais, o Docker também funciona sobre uma camada de infraestrutura, porém as aplicações compartilham de um mesmo sistema operacional, onde o Docker é executado fornecendo um ambiente no qual os containers são executados.

Máquinas Virtuais e Containers podem ser empregados para a mesma funcionalidade, na qual consiste em isolar uma aplicação da outra de maneira em que a aplicação não impacte no funcionamento das demais, sendo que suas dependências ficarão isoladas, reduzindo os problemas de incompatibilidade que um sistema possa ocasionar em outro.

“Na minha máquina funciona” - Disse um desenvolvedor em algum momento de sua vida. Frase comum de se ouvir em nosso ambiente, não é? Porém com a utilização dos Containers (juntamente com os gerenciadores de dependências isso está se tornando cada vez mais raro de se ouvir.

Resumindo, reforçando e apresentando novos pontos, a diferença entre Máquinas Virtuais e Containers são:

- Containers compartilham a camada de sistema operacional. Máquinas virtuais compartilham a camada de hardware.
- Containers são executados em frações de segundos (por compartilhar o mesmo sistema operacional), máquinas virtuais demoram mais por ser

<sup>1</sup> Um hipervisor é um software, firmware ou hardware de computador que cria e executa máquinas virtuais. O hipervisor apresenta os sistemas operacionais convidados com uma plataforma operacional virtual e gerencia a execução dos sistemas operacionais convidados.



necessário subir todo o sistema operacional.

- Por serem mais leves, os contêineres possibilitam a um desenvolvedor executar mais instâncias de um container quando comparados a instâncias inteiras de um sistema operacional executadas por Máquinas Virtuais.

Com o isolamento que os Containers podem oferecer, os desenvolvedores podem desenvolver o software localmente, sabendo que em produção ele será executado de maneira idêntica.

Containers e Máquinas Virtuais podem ser utilizados em conjunto. Você pode instalar e executar o Docker em Máquinas Virtuais, fornecendo assim o melhor dos dois mundos, o isolamento a nível de sistema operacional que uma máquina virtual pode oferecer e a simplicidade, agilidade e rapidez que o Docker pode oferecer.

A definição oficial feita pelo Docker é a seguinte:

*“Um contêiner é uma unidade padrão de software que empacota o código e todas as suas dependências para que o aplicativo seja executado de maneira rápida e confiável de um ambiente de computação para outro. Uma imagem de contêiner do Docker é um pacote de software leve, autônomo e executável que inclui tudo o que é necessário para executar um aplicativo: código, tempo de execução, ferramentas do sistema, bibliotecas do sistema e configurações.”*

## O que são Imagens no contexto do Docker?

Impossível falar de Containers do Docker sem falar em suas Imagens. Você precisa de uma Imagem para poder executar um Container do Docker. As imagens pode ser obtidas de diversas formas:

- Você pode transferir uma imagem de um Container do Docker através de seu arquivo.
- Você pode utilizar um repositório de imagem em nuvem, como por exemplo no repositório do **Docker Hub** (<https://hub.docker.com/>), onde é possível encontrar imagens de sistemas ou fazer o upload das suas próprias imagens. *Pense no Docker Hub como um Github para imagens Docker.*
- Você pode criar e compilar os seus próprios arquivos de imagens.

## Exemplo:

Para rodar uma aplicação em Java, você pode através do Docker Hub baixar uma imagem do JDK ([https://hub.docker.com/\\_/openjdk](https://hub.docker.com/_/openjdk)) que fornecerá o ambiente necessário para rodar sua aplicação em Java.

As imagens então, nada mais são do que arquivos utilizados para executar o Container. Uma imagem é essencialmente construída a partir das instruções para uma versão completa e executável de um aplicativo, se baseando no kernel do

sistema operacional host que está executando o Docker. Quando você executa uma imagem, ela se torna uma ou várias instâncias desse contêiner descrito pela imagem.

## Como configurar um ambiente Docker?

Esta sessão abordará como é realizada a instalação em três sistemas operacionais: Linux, Windows e MacOS. Apesar de apresentar como instalar em Windows e MacOS, eu recomendo fortemente que você utilize o Linux, mais precisamente a distribuição Ubuntu (<https://ubuntu.com/>) para esta disciplina. Teoricamente, o Docker irá funcionar da mesma maneira com os mesmos comandos em todas os três sistemas operacional, porém na prática Windows e MacOS costumam apresentar algumas inconsistências.

- **Linux (Ubuntu)**

Em ambientes Linux, existem mais de uma forma de instalar o Docker. Mas sem dúvidas, a forma mais simples é através de seu script de instalação fornecido pela própria Docker (caso queira verificar as outras formas de instalação, acesse <https://docs.docker.com/install/>).

### **Passo 1:** Obtenha o script de instalação do Docker.

Através do terminal do Ubuntu (pode ser acessado por ctrl+alt+t) execute o seguinte comando:

```
curl -fsSL https://get.docker.com -o get-docker.sh
```

Esse comando fará com que seja baixado o script de instalação do docker com o nome get-docker.sh disponível em <https://get.docker.com> no diretório atual.

### **Passo 2:** Execute o script de instalação.

Execute o comando chmod para fornecer permissões de execução no script de instalação do Docker.

```
sudo sh get-docker.sh
```

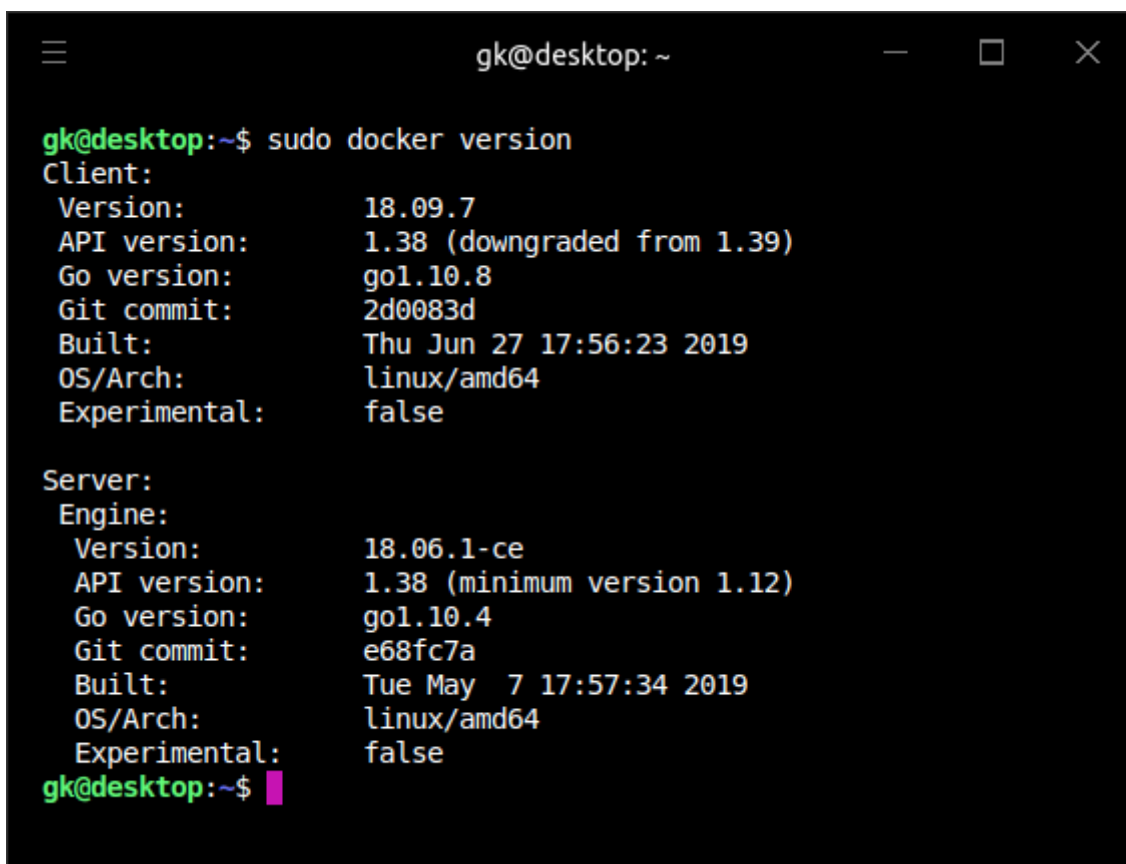
Aguarde o final da instalação.

### Passo 3: Verifique a instalação do Docker

Para conferir se está tudo OK com o Docker, abra o terminal execute o comando:

```
docker version
```

A saída do comando deverá apresentar a versão do Docker que foi instalada.

A terminal window titled 'gk@desktop: ~' with standard window controls. The command 'sudo docker version' has been executed. The output is divided into two sections: 'Client:' and 'Server:'. The 'Client' section lists version 18.09.7, API version 1.38 (downgraded from 1.39), Go version go1.10.8, Git commit 2d0083d, built on Thu Jun 27 17:56:23 2019, OS/Arch linux/amd64, and Experimental false. The 'Server' section lists Engine version 18.06.1-ce, API version 1.38 (minimum version 1.12), Go version go1.10.4, Git commit e68fc7a, built on Tue May 7 17:57:34 2019, OS/Arch linux/amd64, and Experimental false. The prompt 'gk@desktop:~\$' is visible at the bottom with a pink cursor.

```
gk@desktop:~$ sudo docker version
Client:
 Version:           18.09.7
 API version:       1.38 (downgraded from 1.39)
 Go version:        go1.10.8
 Git commit:        2d0083d
 Built:             Thu Jun 27 17:56:23 2019
 OS/Arch:           linux/amd64
 Experimental:      false

Server:
 Engine:
  Version:          18.06.1-ce
  API version:      1.38 (minimum version 1.12)
  Go version:       go1.10.4
  Git commit:       e68fc7a
  Built:            Tue May 7 17:57:34 2019
  OS/Arch:          linux/amd64
  Experimental:     false
gk@desktop:~$
```

- **Windows**

Em ambientes Windows, a instalação é realizada através do instalador oficial do Docker. A atual deste manual de instalação pode ser encontrada em: <https://docs.docker.com/docker-for-windows/install/>.

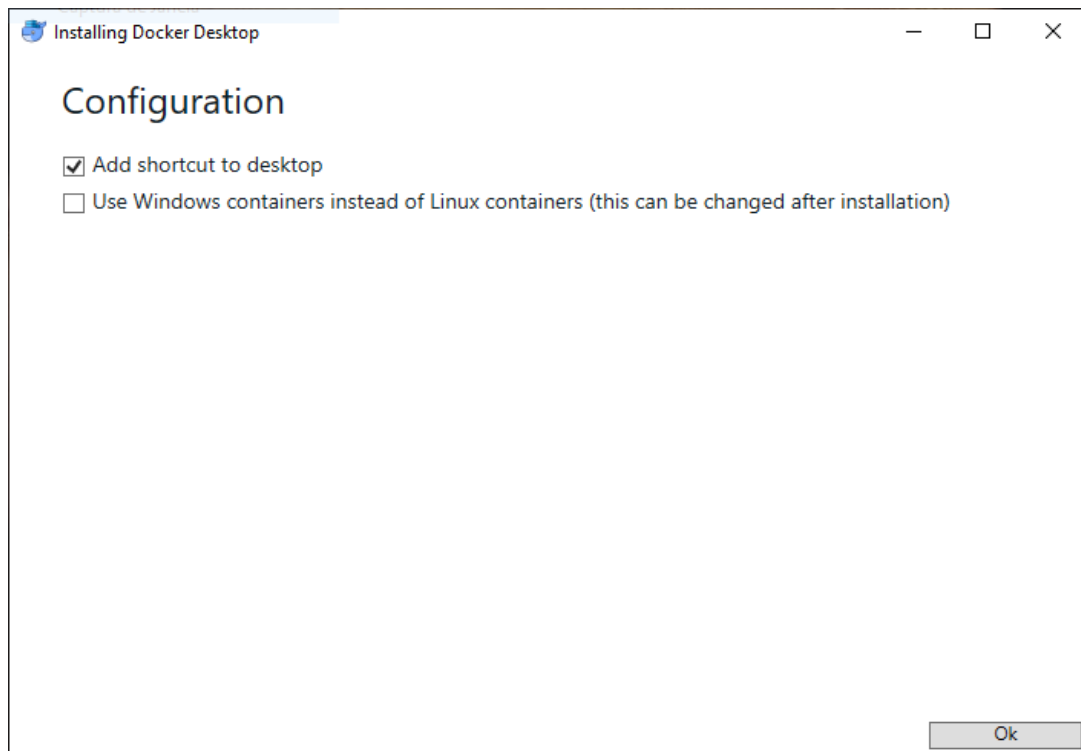
**Passo 1:** Obtenha o instalador do Docker para Windows.

Através do link:

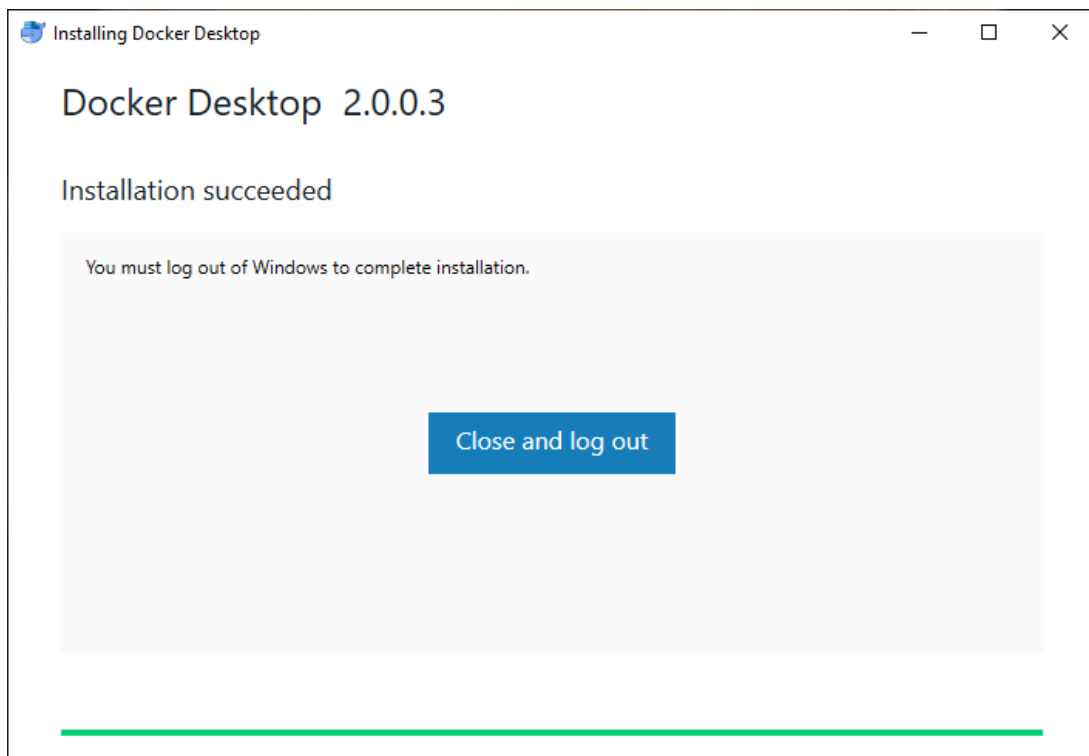
<https://download.docker.com/win/stable/Docker%20for%20Windows%20Installer.exe>

Baixe o instalador para sistemas windows do Docker.

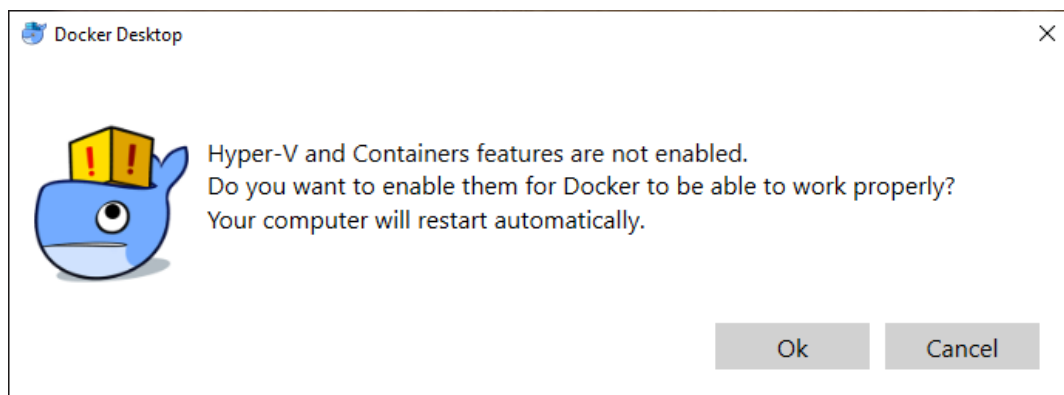
## **Passo 2:** Execute e instale o Docker para windows



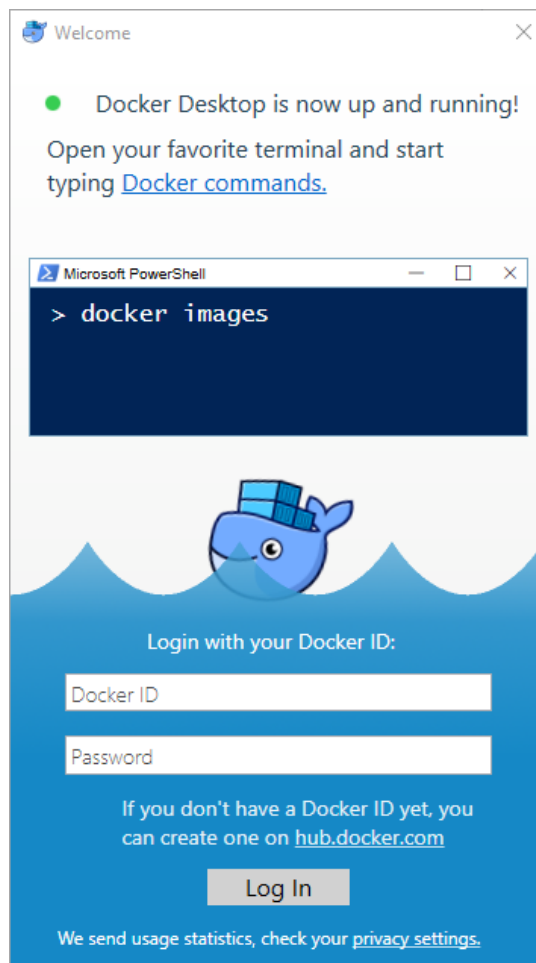
Aguarde o final da instalação.



Talvez seja necessário habilitar o Hyper-V para funcionar, pressione Ok e aguarde o computador reiniciar.



Após reiniciar, o Docker apresentará uma mensagem avisando que está sendo executado



### **Passo 3:** Verifique a instalação do Docker

Para conferir se está tudo OK com o Docker, abra o prompt de comando do windows e execute o comando:

```
docker version
```

A saída do comando deverá apresentar a versão do Docker que foi instalada.

```
Prompt de Comando
Microsoft Windows [versão 10.0.18362.239]
(c) 2019 Microsoft Corporation. Todos os direitos reservados.

C:\Users\gabri>docker version
Client: Docker Engine - Community
Version:      18.09.2
API version:  1.39
Go version:   go1.10.8
Git commit:   6247962
Built:        Sun Feb 10 04:12:31 2019
OS/Arch:      windows/amd64
Experimental: false

Server: Docker Engine - Community
Engine:
Version:      18.09.2
API version:  1.39 (minimum version 1.12)
Go version:   go1.10.6
Git commit:   6247962
Built:        Sun Feb 10 04:13:06 2019
OS/Arch:      linux/amd64
Experimental: false

C:\Users\gabri>
```

- **MacOS**

Em ambientes MacOS, as instruções mais atuais de instalação podem ser acessadas através do link:

<https://hub.docker.com/editions/community/docker-ce-desktop-mac>

**Passo 1:** Obtenha o arquivo de instalação do Docker para Mac

O arquivo de instalação pode ser obtido no link:

<https://hub.docker.com/editions/community/docker-ce-desktop-mac>

Você deverá criar uma conta (caso não tenha) no Docker Hub e clicar em “Get Docker”. O arquivo de instalação .dmg do Docker será baixado em seu computador.

**Passo 2:** Execute o arquivo de instalação

Clique duas vezes no arquivo Docker.dmg para iniciar o processo de instalação.

Aguarde o final da instalação.

Quando a instalação é concluída e o Docker é iniciado, o ícone do Docker na



barra de status superior mostra que o Docker está em execução e acessível a partir de um terminal.

### **Passo 3:** Verifique a instalação do Docker

Para conferir se está tudo OK com o Docker, abra o terminal execute o comando:

```
docker version
```

A saída do comando deverá apresentar a versão do Docker que foi instalada.

## **Considerações Finais**

Com esse capítulo, conseguimos compreender os principais aspectos que compõem a utilização do Docker, como Containers e Imagens, também foi possível abordar a sua instalação em três sistemas operacionais. Esse capítulo servirá como base para os próximos tópicos abordados.

## UNIDADE 2 - Utilizando o Docker

A utilização do Docker não é nada complicada, executar um container não leva mais do que uma linha de comando. Nós aprendemos os principais conceitos que constituem o Docker e neste capítulo aprenderemos o necessário para que possamos utiliza-lo.

Os objetivos deste capítulo são:

- Executar um Container do Docker
- Entender os principais comandos que o Docker possui.

### Executando nosso primeiro Container

Então vamos executar nosso primeiro container (lembre-se de você deve executar esse comando com sudo, ter privilégios de super usuário no usuário atual ou ter adicionado as permissões de execução do docker). O comando para executar um container é o seguinte:

```
docker run hello-world
```

Esse comando, caso tenha sucesso, terá uma saída semelhante a seguir:

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:6540fc08ee6e6b7b63468dc3317e3303aae178cb8a45ed3123180328bcc1d20f
Status: Downloaded newer image for hello-world:latest
```

```
Hello from Docker!
```

```
This message shows that your installation appears to be working correctly.
```

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.  
(amd64)
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

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

Share images, automate workflows, and more with a free Docker ID:

<https://hub.docker.com/>

For more examples and ideas, visit:

<https://docs.docker.com/get-started/>

Agora vamos descobrir o que aconteceu quando executamos esse comando. Na primeira linha podemos ver que o Docker tentou localizar uma imagem chamada hello-world localmente, porém não teve sucesso (devido a ser a primeira execução da imagem), em seguida podemos ver o Docker realizando os procedimentos necessários para baixar a imagem através do Docker Hub ([https://hub.docker.com/\\_/hello-world](https://hub.docker.com/_/hello-world)), na sequência é exibida saudações que fazem parte do Container de Hello world.

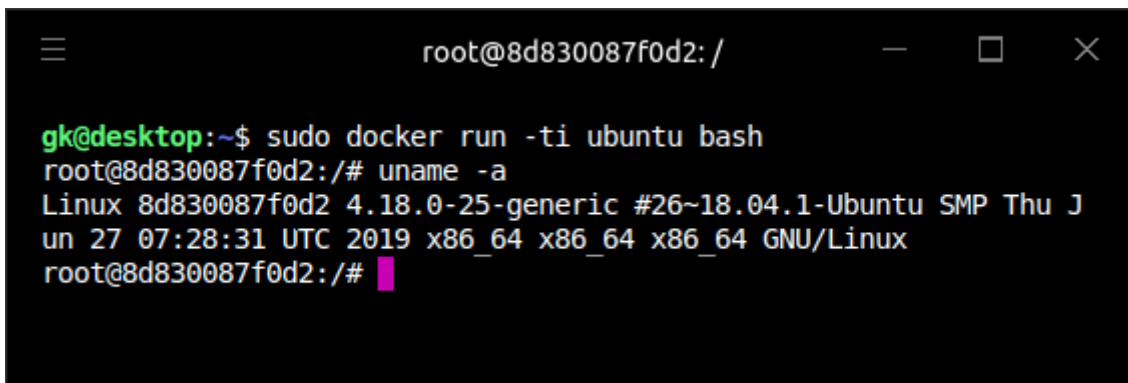
Agora vamos seguir o próprio conselho do docker e tentar algo mais ambicioso. Vamos executar um container com uma imagem do ubuntu. Através do comando a seguir iremos executar um container do ubuntu:

```
docker run -ti ubuntu bash
```

A imagem será baixada e saída esperada para esse comando é algo semelhante a:

```
root@c6d436aa253c:/#
```

Isso significa que estamos agora dentro da imagem do ubuntu que está sendo executada dentro do container. Sugiro que nesse momento, você execute alguns comandos dentro do ambiente do container, no meu caso, eu executei o comando `uname -a` para exibir as informações do sistema.

A terminal window with a black background and white text. The title bar shows 'root@8d830087f0d2: /'. The prompt is 'gk@desktop:~\$'. The user enters 'sudo docker run -ti ubuntu bash'. The prompt changes to 'root@8d830087f0d2:/#'. The user enters 'uname -a'. The output is 'Linux 8d830087f0d2 4.18.0-25-generic #26~18.04.1-Ubuntu SMP Thu Jun 27 07:28:31 UTC 2019 x86\_64 x86\_64 x86\_64 GNU/Linux'. The prompt returns to 'root@8d830087f0d2:/#'.

```
root@8d830087f0d2: /  
  
gk@desktop:~$ sudo docker run -ti ubuntu bash  
root@8d830087f0d2:/# uname -a  
Linux 8d830087f0d2 4.18.0-25-generic #26~18.04.1-Ubuntu SMP Thu J  
un 27 07:28:31 UTC 2019 x86_64 x86_64 x86_64 GNU/Linux  
root@8d830087f0d2:/#
```

Para fixar o conteúdo aprendido, você pode também criar um Container do Docker, recomendo que você execute os seguintes comandos:

```
docker run -ti ubuntu echo Hello World
```

```
gk@desktop: ~  
gk@desktop:~$ sudo docker run -ti ubuntu echo Hello World  
Hello World  
gk@desktop:~$
```

Esse comando irá subir um Container com a imagem do ubuntu no modo interativo e com um terminal anexado (parâmetro -ti, será explicado a seguir), quando o container subir, o Docker executará o comando do ubuntu echo exibindo a mensagem Hello World e o container será finalizado.

Agora vamos tentar outra coisa:

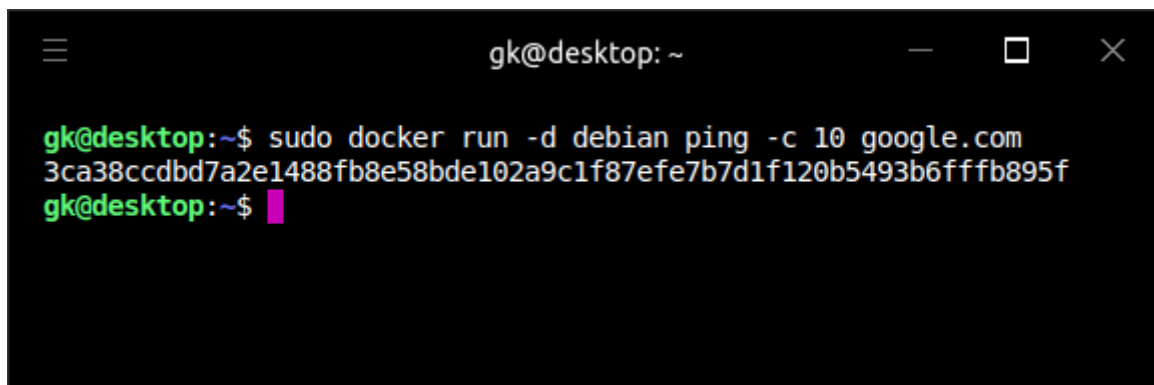
```
docker run -i debian ping -c 10 google.com
```

```
gk@desktop: ~  
gk@desktop:~$ sudo docker run -i debian ping -c 10 google.com  
PING google.com (172.217.30.110) 56(84) bytes of data:  
64 bytes from gru06s35-in-f14.1e100.net (172.217.30.110): icmp_seq=1 ttl=55 time=27.5 ms  
64 bytes from gru06s35-in-f14.1e100.net (172.217.30.110): icmp_seq=2 ttl=55 time=26.8 ms  
64 bytes from gru06s35-in-f14.1e100.net (172.217.30.110): icmp_seq=3 ttl=55 time=26.9 ms  
64 bytes from gru06s35-in-f14.1e100.net (172.217.30.110): icmp_seq=4 ttl=55 time=28.3 ms  
64 bytes from gru06s35-in-f14.1e100.net (172.217.30.110): icmp_seq=5 ttl=55 time=27.1 ms  
64 bytes from gru06s35-in-f14.1e100.net (172.217.30.110): icmp_seq=6 ttl=55 time=26.7 ms  
64 bytes from gru06s35-in-f14.1e100.net (172.217.30.110): icmp_seq=7 ttl=55 time=27.2 ms  
64 bytes from gru06s35-in-f14.1e100.net (172.217.30.110): icmp_seq=8 ttl=55 time=26.7 ms  
64 bytes from gru06s35-in-f14.1e100.net (172.217.30.110): icmp_seq=9 ttl=55 time=27.1 ms  
64 bytes from gru06s35-in-f14.1e100.net (172.217.30.110): icmp_seq=10 ttl=55 time=27.5 ms  
  
--- google.com ping statistics ---  
10 packets transmitted, 10 received, 0% packet loss, time 23ms  
rtt min/avg/max/mdev = 26.661/27.170/28.251/0.484 ms  
gk@desktop:~$
```

Esse comando executa uma imagem da distribuição do Linux chamada Debian, dentro dela executou um ping em google.com por dez vezes, após finalizar o ping o container foi encerrado.

Podemos também executar com o parâmetro -d:

```
docker run -d debian ping -c 10 google.com
```

A terminal window titled 'gk@desktop: ~' with standard window controls. It shows a command being executed: 'sudo docker run -d debian ping -c 10 google.com'. The output is a long alphanumeric string: '3ca38ccdbd7a2e1488fb8e58bde102a9c1f87efe7b7d1f120b5493b6fffb895f'. The prompt 'gk@desktop:~\$' is shown again on the next line.

```
gk@desktop:~$ sudo docker run -d debian ping -c 10 google.com
3ca38ccdbd7a2e1488fb8e58bde102a9c1f87efe7b7d1f120b5493b6fffb895f
gk@desktop:~$
```

Agora executamos o mesmo Container do exemplo passado, porém trocamos o parâmetro pelo parâmetro -d que executa o container de maneira desanexada (iremos abordar esse tipo de execução ainda neste capítulo), mas agora tivemos como saída somente o ID do Container criado. O Container só irá existir enquanto o comando ping estiver em execução, quando o ping finalizar o container deixará de existir, tudo isso em segundo plano.

## Principais comandos do Docker

Agora eu irei listar os principais comandos do Docker, as versões mais atuais apresentam a sua lista de comandos através do comando:

```
docker help
```

E cada comando possui também as informações específicas que podem ser acessadas através do comando:

```
docker COMMAND --help
```

- **docker run<sup>2</sup>**

É um dos principais comandos do Docker. O comando docker run é utilizado para executar um container utilizando como base uma imagem, ele que faz “nascer” um container. A sua sintaxe é a seguinte:

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

### **\*IMPORTANTE\* Modos de execução de um container**

Existem dois modos de execução para a execução de um container: Primeiro plano (Foreground) e desanexado (Detached). Quando você executa um container de maneira desanexada ele é executado em background e a única

---

<sup>2</sup> <https://docs.docker.com/engine/reference/commandline/run/>

saída que você terá é o ID do container que foi executado, para executar o container dessa maneira basta passar como argumento o parâmetro `-d` ao `docker run`, você pode reconectar ao container utilizando o comando `docker attach` (será explicado mais à frente) ou executar comandos através do comando `docker exec`. O modo em primeiro plano, é o modo padrão que o container é executado, ele apresenta os prompts do linux de maneira anexada ao console no qual o docker é executado.

### Principais parâmetros:

`-i` ou `--interactive`: Mantém o STDIN<sup>3</sup> aberto, para que possamos ver o fluxo de dados (a informação que entra e sai) que ocorre dentro do Container.

`-t` ou `--tty`: É a opção que aloca um pseudo terminal TTY para o Container, normalmente é utilizada juntamente com a opção `-i` vista anteriormente. Essas duas opções juntas foram vistas no exemplo anterior apresentado no livro, onde rodamos o `bash` do `ubuntu` associando um terminal a ele.

`-rm`: Essa opção remove um Container quando ele termina sua execução.

`-name`: Atribui um nome ao Container, caso você não defina esse parâmetro o Docker atribui um nome aleatório ao seu container, o que às vezes acaba sendo engraçado, pois aleatoriamente é escolhido um adjetivo e um nome de algum cientistas notável ou hacker.

- **docker ps<sup>4</sup> ou docker container ls<sup>5</sup>**

`docker ps [OPTIONS]`

`docker container ls [OPTIONS]`

Ambos são comandos tem a responsabilidade de listar os Containers, não existem diferenças entre eles. O comando `docker container ls` foi adicionado a partir da versão 1.13 para melhorar a experiência do usuário seguindo a nova estrutura de comandos.

---

<sup>3</sup> STDIN é a entrada padrão de dados de fluxo (o que geralmente consiste em texto) que entram em um programa.

<sup>4</sup> <https://docs.docker.com/engine/reference/commandline/ps/>

<sup>5</sup> [https://docs.docker.com/engine/reference/commandline/container\\_ls/](https://docs.docker.com/engine/reference/commandline/container_ls/)

- **docker images<sup>6</sup> ou docker image ls<sup>7</sup>**

```
docker images [OPTIONS] [REPOSITORY[:TAG]]
```

```
docker image ls [OPTIONS] [REPOSITORY[:TAG]]
```

Ambos são comandos responsáveis por listar as imagens presentes do Docker. A diferença entre os dois é a mesma diferença explicada no comando anterior.

- **docker stop<sup>8</sup> e docker start<sup>9</sup>**

```
docker stop [OPTIONS] CONTAINER [CONTAINER...]
```

```
docker start [OPTIONS] CONTAINER [CONTAINER...]
```

O comando docker stop serve para parar um Container em execução e o comando docker start serve para iniciar um Container parado.

### Exemplo:

Usando o comando docker container ls podemos listar os Containers em execução, em seguida podemos para um Container usando o seu nome ou ID.

```
gk@desktop: ~
gk@desktop:~$ sudo docker container ls
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
fb80f0d2ca65   ubuntu   "bash"    About a minute ago   Up About a minute   quirky_almeida
gk@desktop:~$ sudo docker stop quirky_almeida
quirky_almeida
gk@desktop:~$ sudo docker container ls
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
gk@desktop:~$ sudo docker start quirky_almeida
quirky_almeida
gk@desktop:~$ sudo docker container ls
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
fb80f0d2ca65   ubuntu   "bash"    About a minute ago   Up 3 seconds        quirky_almeida
gk@desktop:~$
```

Neste exemplo o container poderia ser interrompido por ambos dos comandos:

```
docker stop quirky_almeida
```

```
docker stop fb80f0d2ca65
```

<sup>6</sup> <https://docs.docker.com/engine/reference/commandline/images/>

<sup>7</sup> [https://docs.docker.com/engine/reference/commandline/image\\_ls/](https://docs.docker.com/engine/reference/commandline/image_ls/)

<sup>8</sup> <https://docs.docker.com/engine/reference/commandline/stop/>

<sup>9</sup> <https://docs.docker.com/engine/reference/commandline/start/>



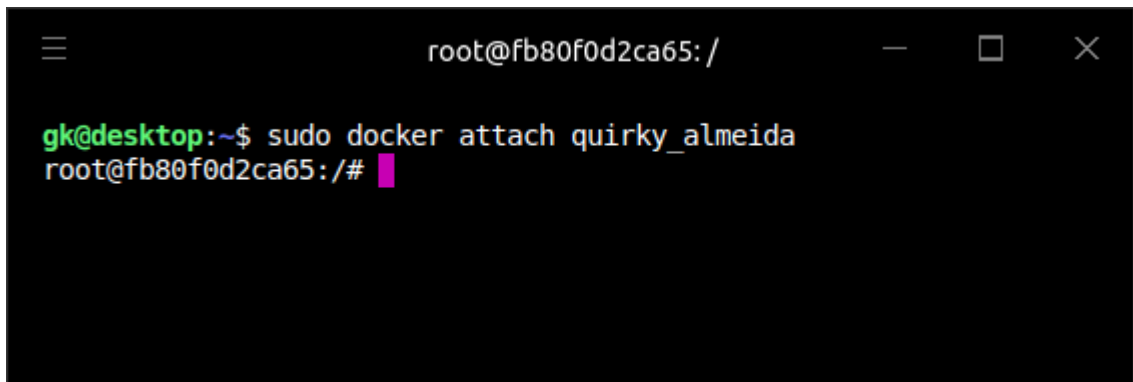
- **docker attach** <sup>10</sup>

`docker attach [OPTIONS] CONTAINER`

O `docker attach` é o comando responsável por conectar a um Container em execução para que possamos executar comandos sobre ele.

**Exemplo:**

Utilizando o exemplo anterior podemos conectar a um Container em execução através do `attach` especificando o ID ou nome do Container.



```
root@fb80f0d2ca65: /
gk@desktop:~$ sudo docker attach quirky_almeida
root@fb80f0d2ca65: /#
```

Neste exemplo o container poderia ser conectado por ambos dos comandos:

```
docker attach quirky_almeida
docker attach fb80f0d2ca65
```

- **docker logs** <sup>11</sup>

`docker logs [OPTIONS] CONTAINER`

O comando `docker logs` é utilizado para a verificação da saída de dados de um Container.

- **docker inspect** <sup>12</sup>

`docker inspect [OPTIONS] NAME|ID [NAME|ID...]`

O comando `docker inspect` é utilizado para visualizar as informações de baixo

---

<sup>10</sup> <https://docs.docker.com/engine/reference/commandline/attach/>

<sup>11</sup> <https://docs.docker.com/engine/reference/commandline/logs/>

<sup>12</sup> <https://docs.docker.com/engine/reference/commandline/inspect/>

nível de um Container ou de uma imagem.

- **docker rm**<sup>13</sup>

`docker rm [OPTIONS] CONTAINER [CONTAINER...]`

O comando `docker rm` pode ser utilizado quando for necessário remover um ou mais Containers, o Container a ser removido pode ser referenciado pelo seu ID ou nome do Container, o parâmetro `-f` força a remoção.

- **docker search**<sup>14</sup>

`docker search [OPTIONS] TERM`

O comando `docker search` é responsável por buscar Containers presentes no Docker Hub ou em algum repositório privado, caso configurado.

**\*\*EXEMPLO\*\***

Para buscar uma imagem do JDK no Docker hub você pode executar o seguinte comando

`docker search openjdk`



```
gk@desktop: ~  
gk@desktop:~$ sudo docker search openjdk  
NAME                DESCRIPTION                               STARS     OFFICIAL    AUTOMATED  
openjdk              OpenJDK is an open-source implementation of ... 1814      [OK]  
oracle/openjdk       Docker images containing OpenJDK Oracle Linux 56  
adoptopenjdk/openjdk11 Docker Images for OpenJDK Version 11 binarie... 49  
adoptopenjdk/openjdk8 Docker Images for OpenJDK Version 8 binaries... 39  
adoptopenjdk/openjdk8-openj9 Docker Images for Eclipse OpenJ9 Version 8 b... 23  
shipilev/openjdk     OpenJDK development builds                10  
arm32v7/openjdk      OpenJDK is an open-source implementation of ... 9  
adoptopenjdk/openjdk12 Docker Images for OpenJDK Version 12 binarie... 7  
arm64v8/openjdk      OpenJDK is an open-source implementation of ... 6  
circleci/openjdk     CircleCI images for OpenJDK                4  
adoptopenjdk/openjdk10 Docker Images for OpenJDK Version 10 binarie... 3  
opennms/openjdk      Base image providing OpenJDK for OpenNMS ser... 2  
adoptopenjdk/openjdk9 Docker Images for OpenJDK Version 9 binaries... 2  
symphonicsoft/openjdkbase openjdk base images with dumb-init and gette... 1  
i386/openjdk         OpenJDK is an open-source implementation of ... 1  
ccitest/openjdk      CircleCI test images for OpenJDK            0  
cfje/openjdk         OpenJDK Builder Image                     0  
classmethod/openjdk-with-git docker image for openjdk and git            0  
amd64/openjdk        OpenJDK is an open-source implementation of ... 0  
trollin/openjdk      OpenJDK is an open-source implementation of ... 0  
winamd64/openjdk     OpenJDK is an open-source implementation of ... 0  
ccistaging/openjdk   CircleCI images for OpenJDK                0  
s390x/openjdk        OpenJDK is an open-source implementation of ... 0  
vicamo/openjdk       Docker images for openjdk                  0  
ppc64le/openjdk      OpenJDK is an open-source implementation of ... 0  
gk@desktop:~$
```

<sup>13</sup> <https://docs.docker.com/engine/reference/commandline/rm/>

<sup>14</sup> <https://docs.docker.com/engine/reference/commandline/search/>

O comando exibe também a quantidade de stars que o projeto possui no Docker Hub, se é oficial e se possui build automatizado, esses parâmetros podem ter a sua busca filtrado pelo comando `--filter`.

## Exercitando

Agora você já possui o conhecimento para executar diversos comandos do Docker, isso quer dizer que somos capazes de manipular nossos Containers. Vamos então colocar em prática alguns comandos aprendidos, primeiro execute o seguinte comando:

```
docker run -dit --name teste ubuntu
```

Esse comando iniciará um Container do ubuntu de maneira iterativa, com um terminal anexado e com o nome de teste. Agora, liste os containers em execução através do comando:

```
docker container ls
```

Encontre os container em execução com o nome de teste. Conecte-se ao Container através do comando:

```
docker attach teste
```

Execute alguns comandos dentro do Container do ubuntu e saia do container. Para sair do container não utilize "exit" pois isso finaliza a execução do container, utilize os comandos `Ctrl-p` e `Ctrl-q`. Agora, pare a execução do container com o comando:

```
docker stop teste
```

Verifique se os o Container sumiu da lista de Containers em execução através do comando:

```
docker container ls
```

Execute novamente o Container através do comando:

```
docker start teste
```

Verifique se os o Container apareceu novamente na lista de Containers em execução através do comando:

```
docker container ls
```

Remova o container através do comando:

```
docker rm teste
```

**Dica:** Tente fazer o mesmo procedimento com a Imagem do Debian, sem consultar os comandos deste capítulo.

## Considerações Finais

Com esse capítulo, conseguimos compreender os principais comandos e seus principais parâmetros necessários para a manipulação do Docker, utilizamos exemplos que nos ajudaram a entender na prática a utilização destes comandos no dia a dia.

## UNIDADE 3 - DockerFile e Docker Hub

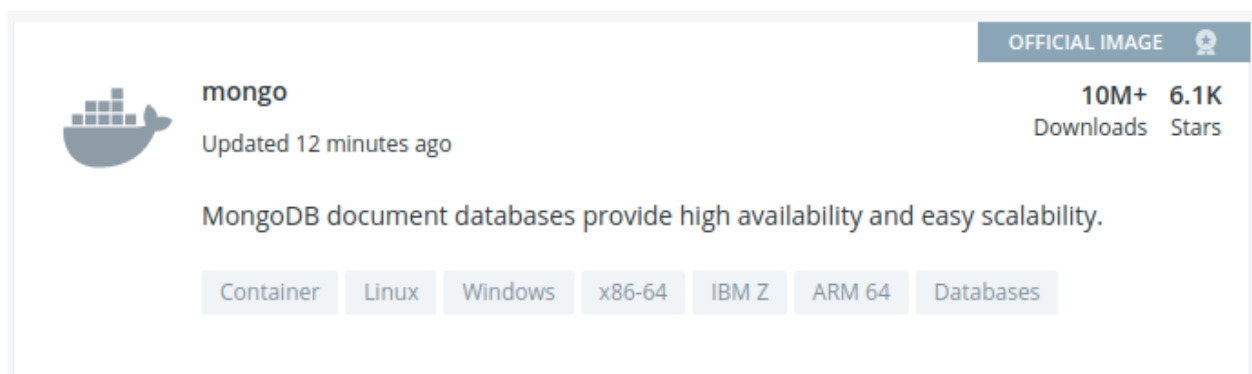
Chegamos a um ponto muito interessante com esse capítulo, vamos aprender a criar nossas próprias imagens do Docker para que possamos “contenizar” nossa aplicação. Aprenderemos também como salvar essa imagem na nuvem.

Os objetivos deste capítulo são:

- Entender a estrutura do Dockerfile
- Criar uma nova Imagem com um Dockerfile
- Fazer o upload da imagem para o Docker Hub

### O Docker Hub

O Docker Hub é uma importante ferramenta no ecossistema do Docker e consiste em um repositório de Imagens do Docker, as empresas podem fazer upload de sistemas, criando imagens considerada oficiais de muitas plataformas, por exemplo: podemos fazer uma breve busca por um determinado banco de dados (no caso, o mongoDB) e o Docker Hub retornará uma lista de imagens encontradas e entre elas temos a imagem oficial do MongoDB.



Mas o Docker Hub também permite que você envie e salve suas imagens em um ambiente de nuvem. Vamos então criar nossa primeira imagem e enviar ela para nuvem. Nesse momento você precisa criar uma conta no Docker Hub, recomendo que você vá para o link <https://hub.docker.com/signup> e crie uma nova conta. Pronto, com essa conta agora podemos seguir os próximos passos e criar nossa primeira imagem do Docker e salvar ela no Docker Hub.

### O Dockerfile

O Dockerfile consiste em um arquivo de texto contendo as instruções de como o Docker deve construir uma nova imagem, com o comando docker build você será

capaz de utilizar suas instruções dentro do seu Dockerfile para criar uma nova imagem do Docker.

## A estrutura do Dockerfile

A estrutura de um Dockerfile consiste em instruções, elas não são case sensitive porém é uma convenção que sempre utilizamos letras maiúsculas para distinguir dos argumentos da instrução. As instruções são executadas em ordem, de cima para baixo.

Uma das instruções mais importantes é a **FROM**, um Dockerfile sempre deve se iniciar com ela, ela especifica qual será a imagem base que iniciará a construção de nossa imagem. Exemplo: uma aplicação Java normalmente deverá ter como imagem base uma imagem do openjdk.

**\*\*IMPORTANTE\*\*** é recomendável sempre utilizar uma imagem oficial como base ao criar as suas próprias imagens.

É possível também construir comentários nos arquivos Dockerfile, todas as linhas iniciadas com # são considerados como comentário. O # em qualquer outro lugar da linha é tratado como argumento

## Os principais comandos do Dockerfile

Vamos estudar aqui os principais comandos do Dockerfile, a lista completa de comandos pode ser consultada em:

<https://docs.docker.com/engine/reference/builder/>

- **FROM<sup>15</sup>**

```
FROM <image> [AS <name>]
FROM <image>[:<tag>] [AS <name>]
FROM <image>[@<digest>] [AS <name>]
```

O Comando FROM como já foi explicado anteriormente consiste no comando que especifica a imagem base que fará parte da construção do nosso container.

- **CMD<sup>16</sup>**

---

<sup>15</sup> <https://docs.docker.com/engine/reference/builder/#from>

<sup>16</sup> <https://docs.docker.com/engine/reference/builder/#cmd>

CMD ["executable","param1","param2"] (forma executável, prefira usar essa forma)  
CMD ["param1","param2"] (como parâmetro padrão para o ENTRYPOINT)  
CMD command param1 param2 (forma shell)

Esse comando executa a instrução especificada toda vez que o container é iniciado. Você também pode especificar um ENTRYPOINT (será explicado mais à frente) para que sirva de parâmetro.

Qualquer parâmetro de execução especificado pelo docker run irá sobrepor a instrução CMD, somente a última instrução CMD tem efeito.

- **RUN<sup>17</sup>**

RUN <command> (forma shell, o comando é executado em um shell, que por padrão é /bin/sh -c no Linux ou cmd /S /C no Windows)

RUN ["executable", "param1", "param2"] (forma executável)

O comando RUN é responsável por executar um comando e confirmar o seu resultado. Quando um comando RUN é executado ele é processado e a imagem resultante é utilizada na próxima etapa no Dockerfile.

No shell, você pode usar uma \ (barra invertida) para continuar uma única instrução RUN na próxima linha.

- **EXPOSE<sup>18</sup>**

EXPOSE <port> [<port>/<protocol>...]

O comando EXPOSE é responsável por tornar uma porta exposta, é utilizado para conectar container ou publicar uma porta através do parâmetro -p no docker run.

- **COPY<sup>19</sup>**

COPY [--chown=<user>:<group>] <src>... <dest>

COPY [--chown=<user>:<group>] ["<src>",... "<dest>"] (essa forma é obrigatória para caminhos que

---

<sup>17</sup> <https://docs.docker.com/engine/reference/builder/#run>

<sup>18</sup> <https://docs.docker.com/engine/reference/builder/#expose>

<sup>19</sup> <https://docs.docker.com/engine/reference/builder/#copy>



possuam espaços em branco)

O comando COPY é utilizado para copiar arquivos do contexto de construção (máquina host) para o contexto da imagem

**\*\*EXEMPLO\*\*** : Podemos utilizar o COPY para copiar o nosso \*.jar que possui nossa aplicação em Java para dentro da imagem para que a aplicação possa ser executada.

- **ENTRYPOINT<sup>20</sup>**

ENTRYPOINT ["executable", "param1", "param2"] (formato de execução, preferido)

ENTRYPOINT command param1 param2 (formato shell)

O comando ENTRYPOINT é utilizado para configurar um comando que será executado quando o container for executado.

### **Como utilizar o Dockerfile?**

Vamos exercitar o que foi aprendido e criar uma Imagem do Docker simples, um servidor http com o Apache2, eu recomendo que você acompanhe cada passo dessa etapa em seu computador. A imagem a seguir apresenta um arquivo Dockerfile que cria um servidor http em uma imagem do Debian. Cada instrução do nosso Dockerfile está comentada para um melhor entendimento.

---

<sup>20</sup> <https://docs.docker.com/engine/reference/builder/#entrypoint>

```

1 # Considera como base a ultima versão da imagem do Debian
2 FROM debian:stretch
3
4 # Executa os comandos do debian para atualizar o repositório
5 # do apt-get, atualizar os pacotes instalados e instalar o
6 # apache2, respectivamente.
7 RUN apt-get update && \
8     apt-get upgrade && \
9     apt-get install -y apache2
10
11 # Executa o apache todas as vezes que o container é iniciado
12 CMD ["apachectl", "-D", "FOREGROUND"]
13
14 # Expomos a porta 80 para poder acessar externamente
15 # com o parametro -t do docker run
16 EXPOSE 80

```

Link para o código: <https://github.com/gabrielkirsten/containers-docker-kubernetes/blob/master/modulo-3/Dockerfile>

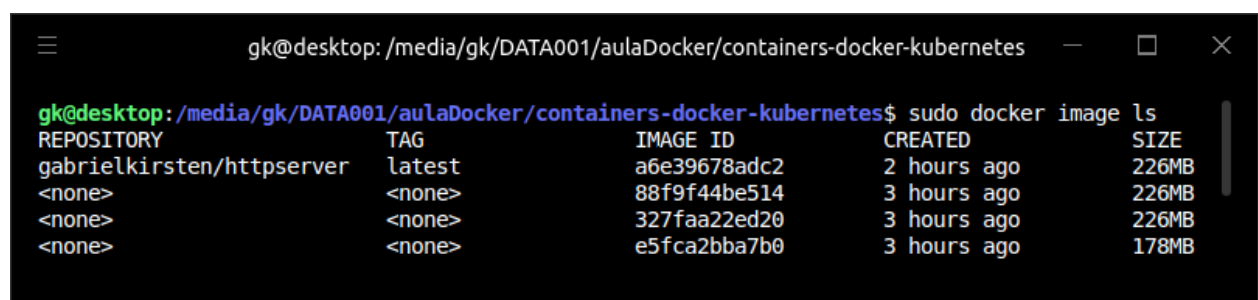
Crie esse arquivo em algum diretório de seu computador, com o nome de Dockerfile (sem extensão).

Agora vamos realizar a construção da nossa primeira imagem Docker, para isso, execute o seguinte comando (substitua o *dockerhub\_username* pelo seu nome de usuário no Docker Hub):

```
docker build -t dockerhub_username/httpserver .
```

Esse comando criará uma nova imagem em seu Docker, que pode ser visualizada pelo comando:

```
docker image ls
```



```

gk@desktop: /media/gk/DATA001/aulaDocker/containers-docker-kubernetes$ sudo docker image ls

```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
gabrielkirsten/httpserver	latest	a6e39678adc2	2 hours ago	226MB
<none>	<none>	88f9f44be514	3 hours ago	226MB
<none>	<none>	327faa22ed20	3 hours ago	226MB
<none>	<none>	e5fca2bba7b0	3 hours ago	178MB

Agora, podemos executar nosso container com o seguinte comando:

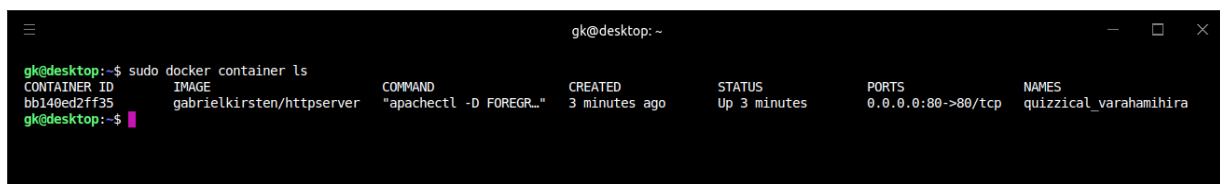
```
docker run -d -p 80:80 dockerhub_username/httpserver
```

Esse comando irá executar o Container criado com o mapeamento da porta exposta 80 para a porta do host (máquina que está executando o Docker) 80 definido pelo parâmetro -p. Também é possível ver o comando -d o que indica que o Container está sendo executado de maneira desanexada.

Agora podemos conferir a execução do container com o comando:

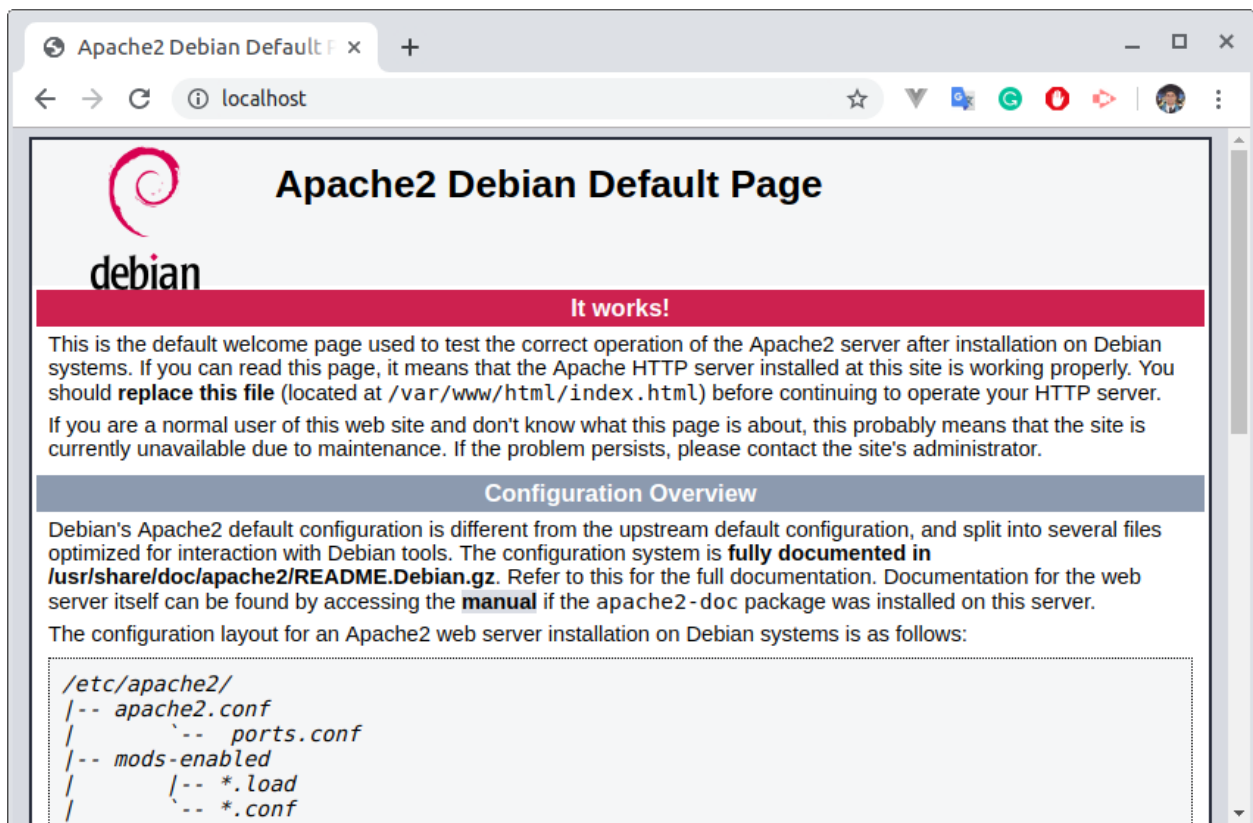
```
docker container ls
```

A saída do comando será algo semelhante a imagem a seguir, o que representa que o Container está em execução

A screenshot of a terminal window titled 'gk@desktop: ~'. The user has entered the command 'sudo docker container ls'. The output is a table with columns: CONTAINER ID, IMAGE, COMMAND, CREATED, STATUS, PORTS, and NAMES. The table shows one container with ID 'bb140ed2ff35', image 'gabrielkirsten/httpserver', command '"apachectl -D FOREGR..."', created '3 minutes ago', status 'Up 3 minutes', ports '0.0.0.0:80->80/tcp', and name 'quizzical\_varahamihira'.

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
bb140ed2ff35	gabrielkirsten/httpserver	"apachectl -D FOREGR..."	3 minutes ago	Up 3 minutes	0.0.0.0:80->80/tcp	quizzical_varahamihira

Agora podemos acessar um navegador e digitar na barra de endereços e digitar o endereço: <http://localhost>, a saída esperada é a página padrão do Apache para o Debian, nela dirá que o nosso Container que criar funcionou. A imagem a seguir representa a saída de exemplo:



Agora podemos executar o comando para finalizar a execução do container que está sendo executado de maneira desanexada, para isso, execute o seguinte comando:

```
docker stop id_do_container
```

Lembrando que o `id_do_container` pode ser obtido através do comando “`container ls`”, ou também pode ser interrompido pelo nome do Container.

## Enviando a Imagem para o Docker Hub

E se você quer acessar essa Imagem de outro computador, outro ambiente, outro contexto? Para solucionar esse problema, podemos agora fazer o upload de nossa Imagem para o Docker Hub. Para enviar a imagem execute o seguinte comando:

```
docker push dockerhub_username/httpserver
```

Caso você nunca tenha entrado no seu Docker Hub através do docker, a saída do seu comando será algo semelhante a imagem a seguir:

```
gk@desktop: ~  
gk@desktop:~$ sudo docker push gabrielkirsten/httpserver  
The push refers to repository [docker.io/gabrielkirsten/httpserver]  
332e4effcb5f: Layer already exists  
31b0e148310d: Layer already exists  
errors:  
denied: requested access to the resource is denied  
unauthorized: authentication required  
gk@desktop:~$
```

Isso quer dizer que você não tem acesso ao repositório que está tentando enviar, ou você não realizou o login, nesse caso, nós não realizamos o login. O Docker sugere um importante comando que resolve esse problema, o `docker login`, vamos executá-lo. Esse comando perguntará o seu login e sua senha e dará uma mensagem de sucesso no final.

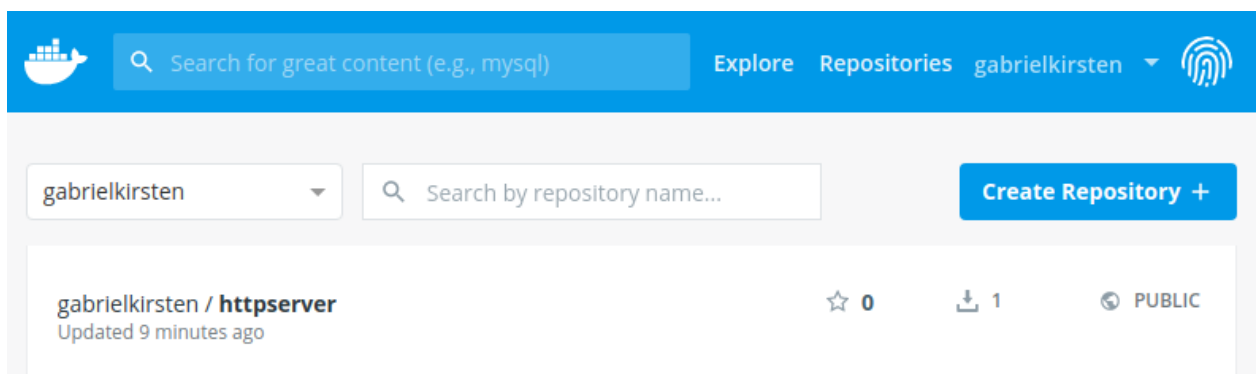
```
gk@desktop: ~  
gk@desktop:~$ sudo docker login  
Login with your Docker ID to push and pull images from Docker Hub. If you  
don't have a Docker ID, head over to https://hub.docker.com to create on  
e.  
Username: gabrielkirsten  
Password:  
WARNING! Your password will be stored unencrypted in /home/gk/.docker/conta  
fig.json.  
Configure a credential helper to remove this warning. See  
https://docs.docker.com/engine/reference/commandline/login/#credentials-s  
tore  
  
Login Succeeded  
gk@desktop:~$
```

Agora vamos tentar enviar nossa imagem novamente... O que aconteceu?

```
gk@desktop: ~  
gk@desktop:~$ sudo docker push gabrielkirsten/httpserver  
The push refers to repository [docker.io/gabrielkirsten/httpserver]  
332e4effcb5f: Pushed  
31b0e148310d: Mounted from library/debian  
latest: digest: sha256:d81274fc31e9981f427782d80f6a15132f4f044511e02923  
e50a00bf9cc17b98 size: 741  
gk@desktop:~$
```

Analizando a saída do comando podemos ver que agora não houve erros. Agora foi possível enviar nossa imagem ao repositório e caso você acesse sua conta do Docker Hub, lá estará ela.

A imagem a seguir apresenta a visão que temos ao acessar a nossa conta do Docker Hub. Nela que podemos visualizar a nossa nova Imagem. Aproveite esse tempo para navegar entre as telas do Docker Hub e descobrir tudo que essa ferramenta pode oferecer.



Para praticar, eu recomendo que você execute os passos anteriores executando outros servidores, como por exemplo o NGINX.

## Considerações Finais

Com esse capítulo, conseguimos apresentar os principais conceitos que compõe o Dockerfile que é o arquivo utilizado para construir novas imagens no Docker. Aprendemos a criar uma nova imagem no Docker e publicar no Docker Hub, o repositório de Imagens do Docker.

Como dito anteriormente, a utilização do Docker é simples, isso que levou a sua utilização em muitas empresas, com poucos comandos conseguimos preparar um ambiente simples com um servidor http e esse servidor será executado da mesma maneira em todos os lugares no qual ele for utilizado.



## UNIDADE 4 - Criando um projeto com o Docker

Chegamos a um capítulo bem interessante, já conseguimos aprender o suficiente para iniciarmos nossa primeira aplicação com o Docker. Esse é um capítulo prático, onde iremos construir um projeto que colocará em prática tudo o que aprendemos até o momento.

Os objetivos deste capítulo são:

- Aprender a utilizar o Docker Compose.
- Criar uma aplicação em Java e rodar no Docker.

### O Docker Compose

Quando temos um sistema composto por muitos Containers em uma mesmo host, subir todos os container é uma tarefa custosa, imagine o cenário onde temos 50 Containers, precisamos lembrar o comando de docker run de cada Container. podemos resolver isso fazendo a Orquestração dos Containers.

Em nossa aplicação teremos somente duas máquinas, um servidor do banco de dados e outro o servidor da API java, porém vamos utilizar o Docker Compose.

O Docker Compose utiliza um arquivo para sua configuração, ele é um arquivo no padrão YAML chamado docker-compose.yml, esse arquivo descreve como cada Container será executado. Cada container descrito no arquivo será baixado (via docker pull) ou construídos (via docker build).

Uma descrição mais extensa da documentação do Docker Compose pode ser encontrada em: <https://docs.docker.com/compose/>.

Os comandos e diretivas do Docker Compose utilizados neste capítulo serão explicados conforme são utilizados.

### Criando a Aplicação

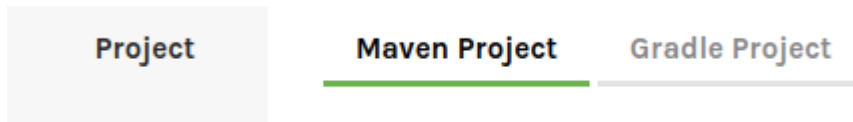
Primeiro precisamos criar a nossa aplicação, e a linguagem escolhida para esse exercício foi Java, iremos utilizar o framework Spring (<https://spring.io/>) para facilitar a criação do nosso app.

Vamos criar uma API com Spring que irá poder cadastrar e consultar dados em um banco de dados. Infelizmente, por ser um curso de Docker, não irei entrar em detalhes da implementação em Java, caso deseje. Você pode encontrar o código no repositório do curso:

<https://github.com/gabrielkirsten/containers-docker-kubernetes/tree/master/modulo-4>

Para facilitar o nosso setup, iremos utilizar o Spring Initializr. Então acesse o site <https://start.spring.io/> e vamos configurar nosso projeto.

Na página do Spring Initializr, vamos definir o tipo de projeto, selecione “Maven Project”:



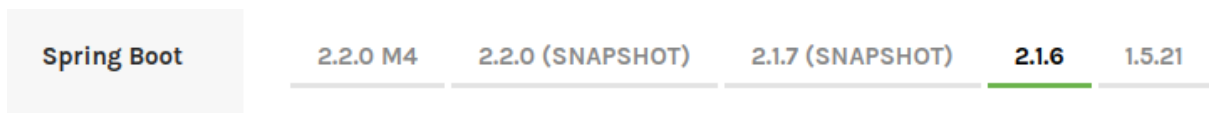
The image shows the 'Project' section of the Spring Initializr form. It has three tabs: 'Project', 'Maven Project', and 'Gradle Project'. The 'Maven Project' tab is selected, indicated by a green underline.

Defina a linguagem como “Java”:



The image shows the 'Language' section of the Spring Initializr form. It has three tabs: 'Language', 'Java', 'Kotlin', and 'Groovy'. The 'Java' tab is selected, indicated by a green underline.

Defina a versão do Spring Boot como “2.1.6”:



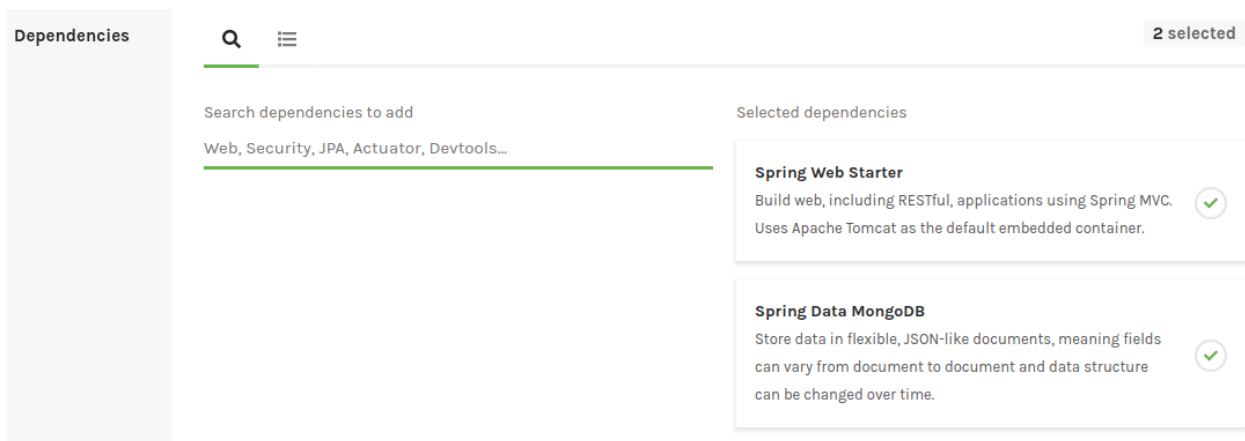
The image shows the 'Spring Boot' section of the Spring Initializr form. It has five tabs: 'Spring Boot', '2.2.0 M4', '2.2.0 (SNAPSHOT)', '2.1.7 (SNAPSHOT)', '2.1.6', and '1.5.21'. The '2.1.6' tab is selected, indicated by a green underline.

Defina os metadados do projeto, aqui definimos o Group como “br.com.uniciv” e o Artifact como “test-api”:

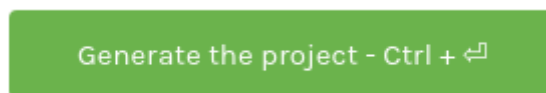


The image shows the 'Project Metadata' section of the Spring Initializr form. It has a sidebar with the title 'Project Metadata'. The main form has three fields: 'Group' with the value 'br.com.uniciv', 'Artifact' with the value 'test-api', and a section for 'Options' with a right-pointing arrow.

Adicione a dependência do “Spring Boot Web Starter” e “Spring Data MongoDB”:

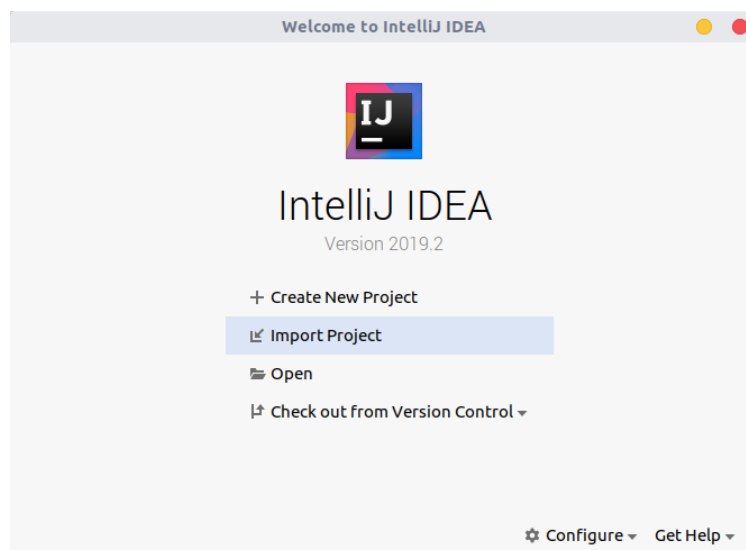


Clique em “Generate the project”:



Nesse momento o Spring Initializr irá baixar um arquivo test-api.zip que será nosso projeto configurado.

Vamos agora importar nosso projeto, no meu caso estou utilizando a versão community do IDE IntelliJ IDEA<sup>21</sup>, mas você pode abrir o seu projeto na IDE que você se sentir mais confortável.

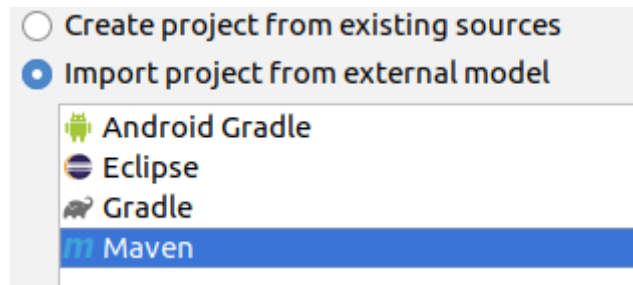


Para importar o projeto, descompacte o zip e clique em “Import Project” selecionado onde o projeto foi descompactado.

Informe que o arquivo é um projeto Maven:

---

<sup>21</sup> Você pode encontrar o IntelliJ IDEA aqui: <https://www.jetbrains.com/idea/download/>



E siga até o fim da importação. Pronto, teremos nosso projeto aberto. Talvez demore alguns minutinhos até o IntelliJ concluir a configuração do projeto e importar as dependências.

Vamos criar a classe que será nossa entidade do MongoDB, o nome dela será Message.java e ela tem como atributos o id e o campo message.

```
1 public class Message implements Serializable {
2
3     @Id
4     private UUID id;
5
6     private String message;
7
8     public UUID getId() {
9         return id;
10    }
11
12    public void setId(UUID id) {
13        this.id = id;
14    }
15
16    public String getMessage() {
17        return message;
18    }
19
20    public void setMessage(String message) {
21        this.message = message;
22    }
23
24 }
```

Link para o código: <https://github.com/gabrielkirsten/containers-docker-kubernetes/blob/master/modulo-4/test-api/src/main/java/br/com/uniciv/testapi/Message.java>

Vamos criar o Repository responsável por acessar os dados no MongoDB. Crie a Interface chamada MessageRepository estendendo de MongoRepository conforme a imagem a seguir:

```
1 @Repository
2 public interface MessageRepository extends MongoRepository<Message, UUID> {
3 }
```

Link para o código: <https://github.com/gabrielkirsten/containers-docker-kubernetes/blob/master/modulo-4/test-api/src/main/java/br/com/uniciv/testapi/MessageRepository.java>

Vamos criar dois endpoints com o Spring. Abra a classe TestApiApplication e faça as seguintes modificações:

- Anote a classe como @RestController
- Importe a classe de repository e injete ela em seu construtor
- Crie o método getAll(), conforme a imagem a seguir.
- Crie o método post(), conforme a imagem a seguir.

(sabemos que esses não são os melhores padrões para a criação de uma aplicação, mas estamos fazendo tudo da maneira mais simples e dando o foco no Docker).

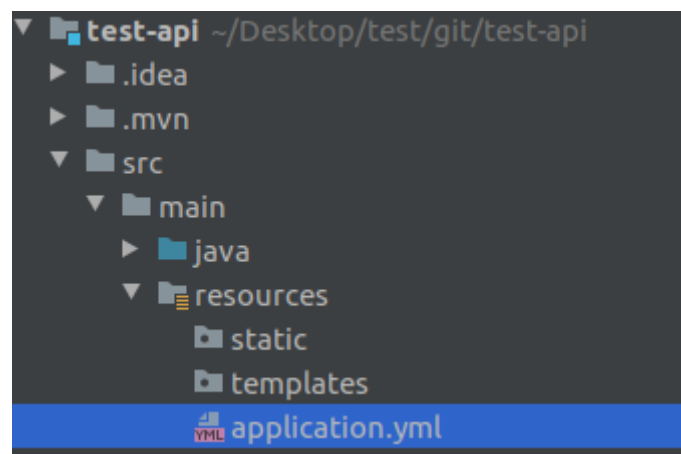
```

1 @RestController
2 @SpringBootApplication
3 public class TestApiApplication {
4
5     private final MessageRepository repository;
6
7     @Autowired
8     public TestApiApplication(MessageRepository repository) {
9         this.repository = repository;
10    }
11
12    public static void main(String[] args) {
13        SpringApplication.run(TestApiApplication.class, args);
14    }
15
16    @GetMapping
17    @ResponseStatus(HttpStatus.OK)
18    public List<Message> getAll(){
19        return repository.findAll();
20    }
21
22    @PostMapping
23    @ResponseStatus(HttpStatus.CREATED)
24    public void post(@RequestBody Message message) {
25        message.setId(UUID.randomUUID());
26        repository.save(message);
27    }
28
29

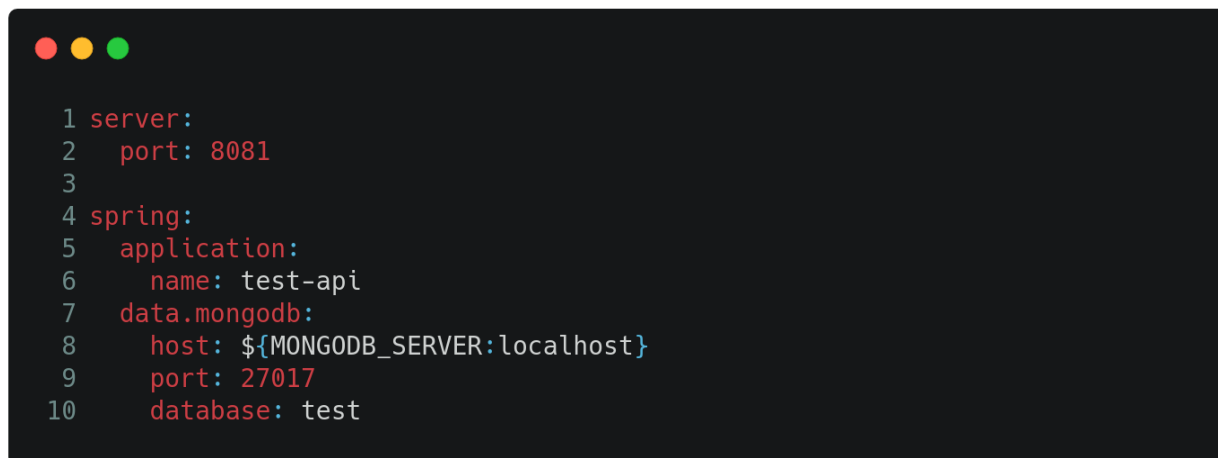
```

Link para o código: <https://github.com/gabrielkirsten/containers-docker-kubernetes/blob/master/modulo-4/test-api/src/main/java/br/com/uniciv/testapi/TestApiApplication.java>

Dentro do projeto, na pasta de resources, crie o arquivo application.yml



Configure o projeto conforme a imagem a seguir:



```
1 server:
2   port: 8081
3
4 spring:
5   application:
6     name: test-api
7   data.mongodb:
8     host: ${MONGODB_SERVER:localhost}
9     port: 27017
10    database: test
```

[08]

Link para o código: <https://github.com/gabrielkirsten/containers-docker-kubernetes/blob/master/modulo-4/test-api/src/main/resources/application.yml>

Esse arquivo está dizendo para subir o servidor na porta 8081, definindo o nome da aplicação como test-api e configurando o servidor do MongoDB como host como o parâmetro definido com o nome de MONGODB\_SERVER (esse parâmetro será passado posteriormente pelo arquivo docker-compose.yml) ou, caso não seja passado o parâmetro, o host será localhost. A porta do MongoDB será 27017 (padrão) e o banco de dados será com o nome de “test”.

Precisamos agora configurar o Docker para o nosso projeto, com uma novidade, vamos configurar o Docker com mais de um estágio. O primeiro estágio será o estágio de build e em seguida será o estágio de produção, usaremos então dois containers.

Crie na raiz do projeto test-api um arquivo chamado Dockerfile (sem extensão, como visto nos capítulos anteriores),

Agora vamos partir para a configuração do nosso Docker Compose. Configure o arquivo docker-compose.yml em um diretório acima do projeto, conforme a imagem a seguir:

```
1 version: '3'
2
3 services:
4
5   mongodb:
6     image: "mongo:3.0.4"
7     ports:
8       - "27017:27017"
9     command: mongod --smallfiles
10
11
12   test-api:
13     build: ./test-api
14     ports:
15       - "8081:8081"
16     depends_on:
17       - mongodb
18     environment:
19       MONGODB_SERVER: mongodb
20
```

Link para o código: <https://github.com/gabrielkirsten/containers-docker-kubernetes/blob/master/modulo-4/docker-compose.yml>

O arquivo do docker-compose, está especificando qual a versão que iremos trabalhar, no caso será a 3 e quais são os serviços (Containers) que serão executados, teremos dois serviços o “mongodb” que será o do banco de dados e o “test-api” que será nossa api em Java. O serviço mongodb utiliza a diretiva “image” que especifica qual imagem irá rodar no serviço, isso quer dizer que iremos utilizar uma imagem pronta e o serviço test-api utiliza a diretiva “build” que define o diretório onde está o Dockerfile que será construído. A diretiva ports é análoga ao parâmetro -p do docker run explicado anteriormente. A diretiva “environment” passa parâmetros para o Container. A diretiva command executa um comando no Container. A diretiva depends\_on especifica que o Container do test-api depende do mongodb.

Agora iremos criar nosso Dockerfile com dois estágios, que será responsável por realizar o build do nosso projeto através do Maven e subi-lo para podermos realizar requisições. Cada linha do nosso Dockerfile está comentada explicando cada comando. A imagem a seguir descreve o nosso arquivo do Dockerfile.



```

1 #####
2 # Estágio de build
3
4 # Utiliza a imagem do maven para fazer o estágio de build
5 FROM maven:3.6.1-jdk-11-slim as build-stage
6
7 # Cria um diretório para
8 RUN mkdir -p /usr/src/app
9 # Muda o diretório padrão de execução dos scripts para o diretório criado
10 WORKDIR /usr/src/app
11 # Adiciona os arquivos do projeto para dentro do Container
12 ADD . /usr/src/app
13 # Executa o maven com o goal de construção do projeto
14 RUN mvn install
15
16 #####
17 # Estágio de produção
18
19 # Utiliza a imagem do Open JDK para
20 FROM openjdk:8 as production-stage
21
22 # Copia os arquivos construídos na etapa anterior para ser executados
23 COPY --from=build-stage /usr/src/app/target/test-api-*.jar test-api.jar
24 # Expõe a porta 8081
25 EXPOSE 8081
26 # Executa nossa aplicação
27 ENTRYPOINT ["java", "-jar", "/test-api.jar"]

```

Link para o código: <https://github.com/gabrielkirsten/containers-docker-kubernetes/blob/master/modulo-4/test-api/Dockerfile>

Agora está tudo pronto para podermos executar o nosso projeto. Vamos executar o comando do Docker Compose que irá subir os dois Containers:

```
docker-compose up --build
```

ou

```
docker.compose up --build
```

Você verá em seu terminal, os Containers sendo construídos a partir das instruções do docker-compose.yml e ao final da construção, será exibida algumas saídas, umas com o início como:

```
test-api_1 |
```

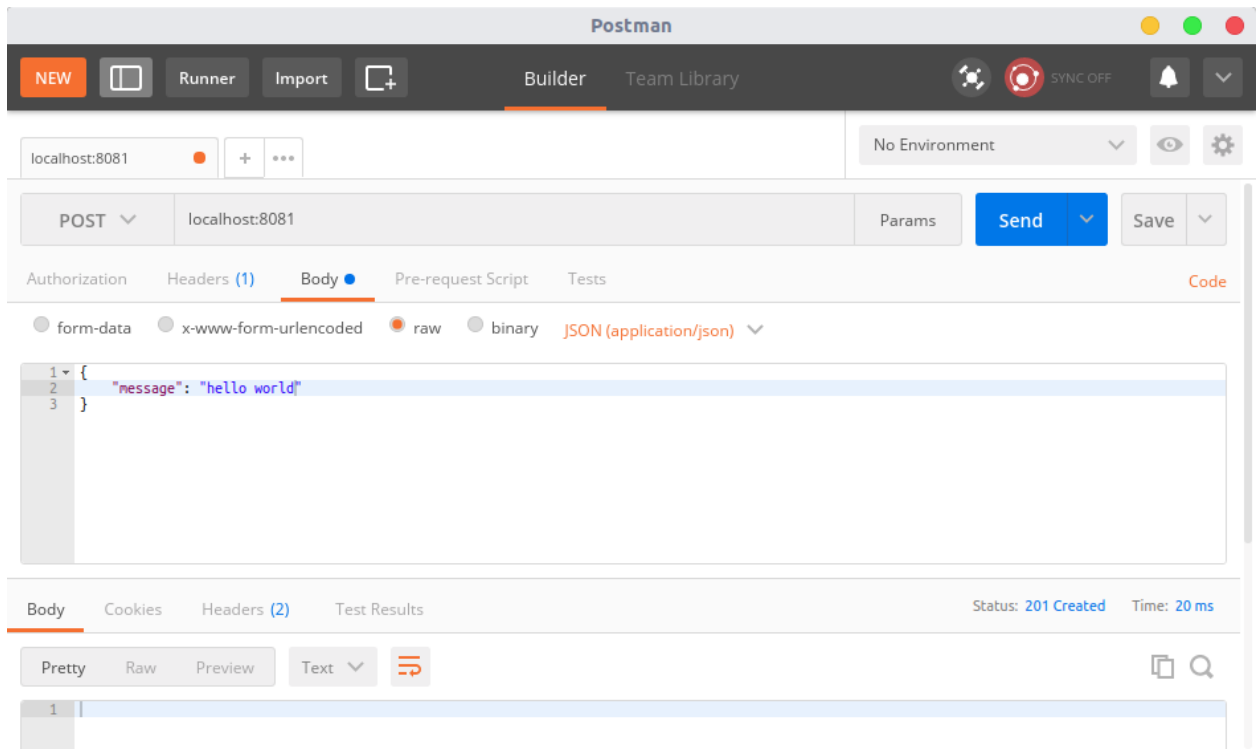
Outras com o início como:

```
mongodb_1 |
```

Isso serve para identificar os dois Containers, o do MongoDB e o outro da nossa aplicação test-api. O número após o nome é referente à instância da aplicação, podemos ter mais de uma instância rodando ao mesmo tempo.

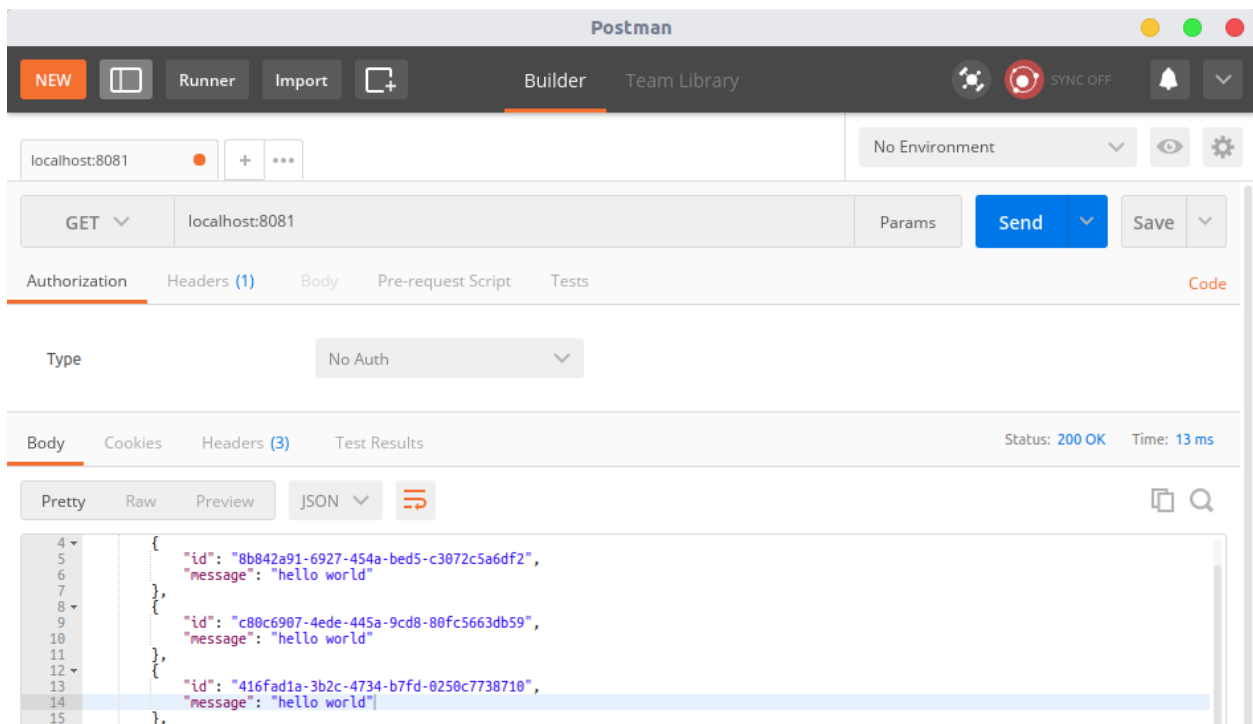
Vamos agora realizar alguns requests para a nossa API. Eu utilizei para criar os requests o Postman<sup>22</sup>, mas você pode utilizar qualquer cliente HTTP que você se sentir mais confortável, você pode até mesmo utilizar o comando curl.

Através do comando HTTP Post de nossa API, foi possível salvar alguns dados no MongoDB:



E quando executamos o comando HTTP Get em nossa API, o MongoDB retornou os dados:

<sup>22</sup> O Postman pode ser baixado aqui: <https://www.getpostman.com/downloads/>



Agora que aprendemos como executar um Container para aplicações Java. Você pode treinar com outras linguagens, tente executar o tutorial para criar uma aplicação do VueJS no Docker: <https://br.vuejs.org/v2/cookbook/dockerize-vuejs-app.html>

## Considerações Finais

Com esse capítulo, após termos estudado os capítulos anteriores, conseguimos criar um incrível projeto composto por dois Containers, um para o nosso banco de dados MongoDB e outro para a nossa aplicação que consiste em uma API Java. Aprendemos também o básico sobre Docker Compose que é um importante orquestrador de containers.

A importância da Orquestração de Containers se tornou ainda mais visível quando começamos a construir aplicações na arquitetura de microsserviços<sup>23</sup>, o Docker e a orquestração de Containers torna muito mais fácil a manipulação de aplicações distribuídas. Imagine a aplicação que criamos agora, porém se ela for composta de 10 serviços, o docker tornaria a execução desses Containers uma tarefa mais automatizada.

<sup>23</sup> Explicação da RedHat sobre o que são microsserviços: <https://www.redhat.com/pt-br/topics/microservices/what-are-microservices>

## UNIDADE 5 - Kubernetes

Agora, em nosso último capítulo, iremos aprender uma ferramenta avançada no Docker, a Kubernetes, executaremos além de tutoriais do próprio instalaremos uma versão do Kubernetes (Minikube) para executar o Kubernetes localmente.

Os objetivos deste capítulo são:

- Entender o funcionamento do Kubernetes.
- Aplicar o Kubernetes em um projeto.

### Kubernetes

A Kubernetes (<https://kubernetes.io/>), também chamada de k8s, é uma ferramenta avançada de orquestração de containers criada pelo Google, após a experiência de longa data que o Google utiliza Containers em ambiente de produção. A documentação completa do Kubernetes pode ser encontrada em: <https://kubernetes.io/docs/home/>

A orquestração de Containers é muito importante para que possamos organizar os vários Containers que podem construir a aplicação, de maneira em que um Container pode aumentar ou diminuir a quantidade de instâncias em execução conforme uma aplicação exija mais ou menos demanda.

A Kubernetes hoje está presente em diversos serviços de Cloud como AWS, Azure e Google Cloud Platform. Apesar de ser utilizado nesse curso em somente um computador, a Kubernetes vem para resolver o problema de gerenciar um cluster de servidores.

Os principais conceitos que temos que aprender para utilizar o Kubernetes são:

- **node**

O node (ou nó em português, também conhecido anteriormente por minion) consiste em a máquina de trabalho no cluster do Kubernetes. Um node pode ser uma máquina virtual ou física. Cada node contém os serviços necessários para executar pods.

- **cluster**

O cluster é um conjunto de nodes onde o Kubernetes é instalado.

- **pod**

O pod é a menor unidade para o contexto do Kubernetes. Ele está para o Kubernetes como o Container está para o Docker. O pod nada mais é que um

processo em execução dentro do cluster do Kubernetes. O pod pode encapsular um ou mais Containers de uma aplicação.

- **services**

Os services (ou serviços) são endpoints estáveis, diferente dos pods. Você pode em um cluster ter diversos pods de consulta de cep, você pode então criar um serviço chamado de consulta de cep para endereçar esses pods, para que os pods que precisam consultar o cep acesse o serviço de consulta de cep que irá retornar um pod disponível. O Kubernetes faz o balanceamento de carga entre os pods presentes em um mesmo serviço. Essa camada de serviços fornece uma abstração para fique a cargo do Kubernetes conhecer os detalhes internos de cada pod.

## **Minikube**

O Minikube consiste em uma implementação local em pequena escala do Kubernetes que pode ser executada em qualquer lugar.

**\*\*ATENÇÃO\*\*** O Minikube é interessante para estudos e NÃO DEVE SER UTILIZADO EM PRODUÇÃO. Caso deseje, acesse aqui um tutorial detalhado da instalação do Minikube: <https://kubernetes.io/docs/tasks/tools/install-minikube/>

A instalação do Minikube consiste basicamente em três passos:

1. Baixe o minikube

```
curl -Lo minikube https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64 &&  
chmod +x minikube
```

2. Instale o Minikube

```
sudo install minikube /usr/local/bin
```

3. Verifique a instalação (você precisa ter o VirtualBox instalado)

```
minikube start
```

```
gk@desktop: ~  
gk@desktop:~$ minikube start  
🐳 minikube v1.2.0 on linux (amd64)  
📥 Downloading Minikube ISO ...  
129.33 MB / 129.33 MB [=====] 100.00% 0s  
🔥 Creating virtualbox VM (CPUs=2, Memory=2048MB, Disk=20000MB) ...  
🐳 Configuring environment for Kubernetes v1.15.0 on Docker 18.09.6  
📥 Downloading kubeadm v1.15.0  
📥 Downloading kubelet v1.15.0  
📡 Pulling images ...  
🚀 Launching Kubernetes ...  
🔍 Verifying: apiserver proxy etcd scheduler controller dns  
🎉 Done! kubect! is now configured to use "minikube"  
💡 For best results, install kubectl: https://kubernetes.io/docs/tasks/tools/install-kubectl/  
gk@desktop:~$
```

Recomendo que instale também, conforme o indicado pela saída do comando minikube start, a instalação do kubectl, para isso, execute os seguintes comandos:

1. Baixe a última versão do kubectl

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/`curl -s  
https://storage.googleapis.com/kubernetes-release/release/stable.txt`/bin/linux/amd64/kubectl
```

2. Dê as permissões de execução para o script

```
chmod +x ./kubectl
```

3. Mova o binário para o seu PATH

```
sudo mv ./kubectl /usr/local/bin/kubectl
```

4. Teste a instalação

```
kubectl version
```

A saída desse comando deverá ser parecida com a imagem a seguir:

```
gk@desktop: ~  
gk@desktop:~$ kubectl version  
Client Version: version.Info{Major:"1", Minor:"15", GitVersion:"v1.15.1", GitComm  
it:"4485c6f18cee9a5d3c3b4e523bd27972b1b53892", GitTreeState:"clean", BuildDate:"2  
019-07-18T09:18:22Z", GoVersion:"go1.12.5", Compiler:"gc", Platform:"linux/amd64"  
}  
Server Version: version.Info{Major:"1", Minor:"15", GitVersion:"v1.15.0", GitComm  
it:"e8462b5b5dc2584fdcd18e6bcfe9f1e4d970a529", GitTreeState:"clean", BuildDate:"2  
019-06-19T16:32:14Z", GoVersion:"go1.12.5", Compiler:"gc", Platform:"linux/amd64"  
}  
gk@desktop:~$
```

## Kubernetes na prática

Vamos aprender na prática os conceitos do Kubernetes. O próprio site do Kubernetes oferece um incrível tutorial para aprendermos como utilizar o Kubernetes. Acesse <https://kubernetes.io/docs/tutorials/kubernetes-basics/deploy-app/deploy-interactive/> para realizar online ou utilize a sua máquina com o Minikube + kubectl para fazer o tutorial.

Vamos subir então um pod do Kubernetes com a imagem do ubuntu

```
kubectl run nginx --image=nginx --generator=run-pod/v1
```

E vamos conferir se o pod está sendo executado

```
kubectl get pods
```

## Criando um Pod com YAML

Os Pods do Kubernetes podem ser criados por um arquivo do tipo YAML.

Os elementos raízes deste YAML são:

- **apiVersion**  
versão do Kubernetes API.
- **kind:**  
Tipo do objeto. Aceita valores como: Pod, replicaset, deployment e service.
- **metadata:**  
Dados sobre o objeto como: nome, labels e namespace.
- **specs:**  
Especificações do Pod, provê informações adicionais pertencentes ao objeto. Cada tipo de objeto tem um formato.

Vamos criar então um pod para a imagem Docker do exercício anterior, crie um arquivo pod-definition.yml conforme a imagem a seguir:

```
1 apiVersion: v1
2
3 kind: Pod
4
5 metadata:
6   name: test-api
7   labels:
8     app: test-api
9
10 spec:
11   containers:
12     - name: test-api
13       image: gabrielkirsten/test-api
14
```

Link para o código: <https://github.com/gabrielkirsten/containers-docker-kubernetes/blob/master/modulo-5/kubernetes-definition/pod/pod-definition.yml>

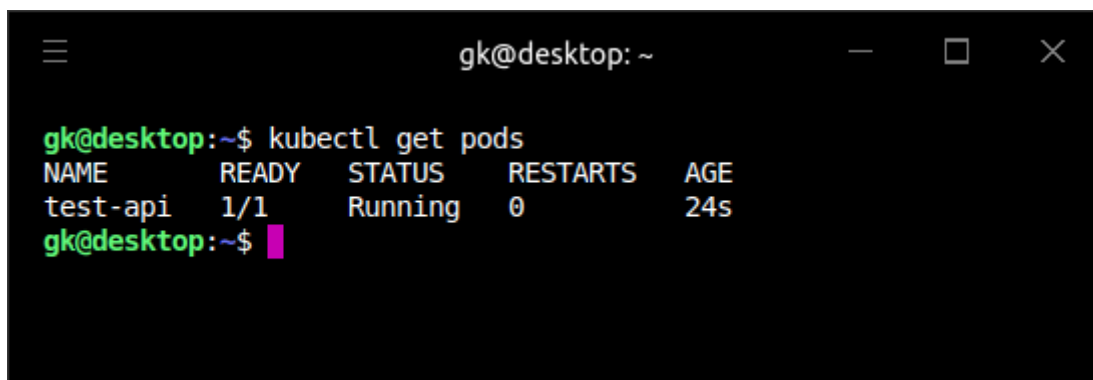
Definimos então a versão do formato de nosso arquivo YAML como v1, o tipo como Pod e em metadados definimos os nome e uma etiqueta para marcar nosso Pod. Ao final, nas especificações, definimos quais serão os containers e suas respectivas imagens.

Vamos agora dizer para o Kubernetes criar o nosso Pod a partir de nosso arquivo YAML:

```
kubectl create -f pod-definition.yml
```

Agora ao rodar o comando que lista os Pods:

```
kubectl get pods
```



A terminal window titled 'gk@desktop: ~' showing the command 'kubectl get pods' and its output. The output is a table with columns: NAME, READY, STATUS, RESTARTS, and AGE. The table shows one pod named 'test-api' with a READY status of '1/1', STATUS of 'Running', 0 RESTARTS, and an AGE of '24s'.

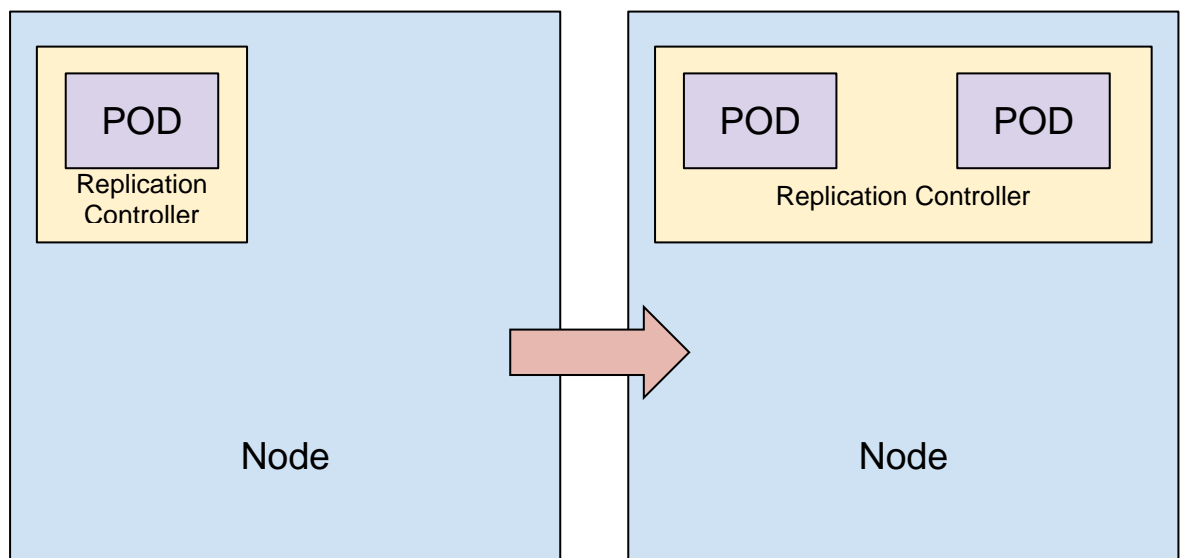
NAME	READY	STATUS	RESTARTS	AGE
test-api	1/1	Running	0	24s

Podemos ver que o nosso Pod está sendo listado.

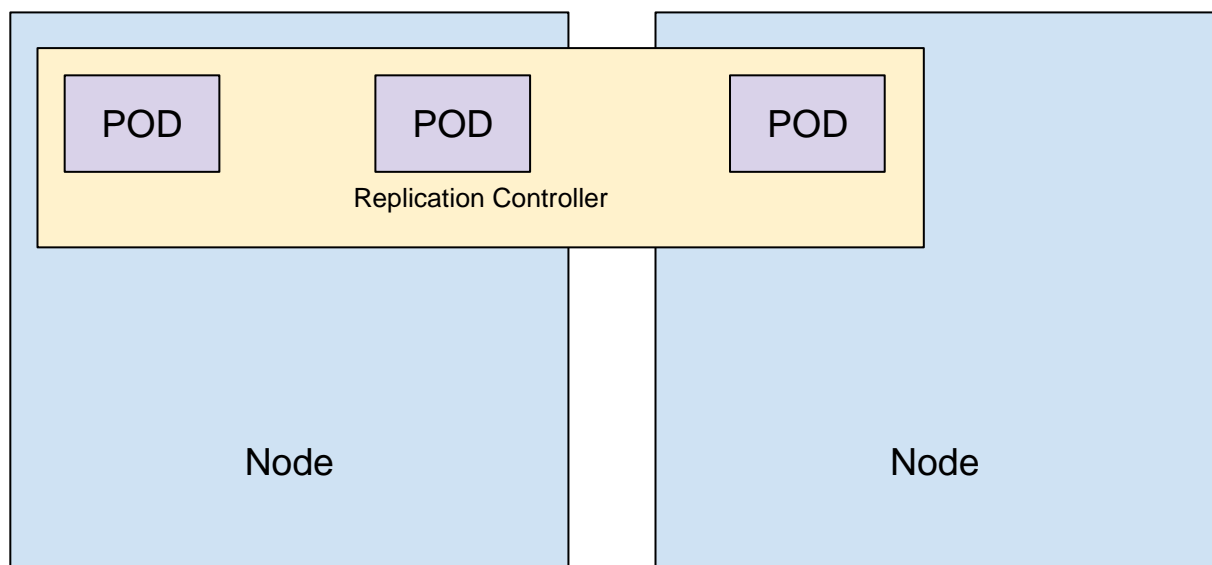


## Replication Controller

Um Replication Controller é responsável por garantir que um número especificado de réplicas de Pod seja executado a qualquer momento, garantindo que um Pod esteja sempre disponível. Como o nome já diz ele é responsável por fazer o controle da replicação de um determinado Pod. Quando uma demanda a um determinado recurso (descrito por um Pod) é aumentada ou quando um Pod parar de responder, o Replication Controller responsável por aquele Pod irá adicionar mais uma instância daquele Pod.



Caso não seja possível adicionar no mesmo nó o Replication Controller poderá adicionar em outro nó do mesmo cluster:



O Replication Controller também é responsável por fazer o balanceamento de cargas para os Pods, de maneira em que não fique sobrecarregado somente um dos Pods.

Além do Replication Controller existe o ReplicaSet e o Deployment, ambos são utilizados para configurar a replicação. A documentação do Kubernetes recomenda o seguinte:

*Nota: Um Deployment que configura um ReplicaSet agora é a maneira recomendada de configurar a replicação.*

Vamos falar então da maneira recomendada, vamos configurar um Deployment que configura um ReplicaSet. A documentação completa do Deployment pode ser encontrada em:

<https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>

Vamos seguir com a aplicação anterior (test-api) para funcionar com o Kubernetes, inicialmente é necessário configurar o Deployment para o banco de dados (database-deployment), a imagem a seguir mostra como esse arquivo será construído, ele segue basicamente o formato do arquivo do Pod, com os mesmos elementos raízes mas existem diferenças principalmente no elemento spec que serão abordadas a seguir.

```
1 apiVersion: apps/v1
2
3 kind: Deployment
4
5 metadata:
6   name: database-deployment
7   labels:
8     app: database
9
10 spec:
11   template:
12     metadata:
13       name: database
14       labels:
15         app: database
16     spec:
17       containers:
18         - name: mongo
19           image: mongo:3.0.4
20           ports:
21             - containerPort: 27017
22
23   replicas: 1
24   selector:
25     matchLabels:
26       app: database
27
```

Link para o código: <https://github.com/gabrielkirsten/containers-docker->

Vamos falar então sobre o conteúdo do spec, inicialmente temos a marcação “template” que descreve como será o conteúdo do pod a ser replicado, então podemos descrever o nosso pod no mesmo formato do Pod. Em seguida, temos a marcação “replicas” que descreve a quantidade de replicações do nosso Pod (por se tratar de um banco de dados com o sincronismo de dados desativado, iremos configurar apenas uma replicação) e por fim, temos a marcação selector que determina como o Deployment irá encontrar os Pods a ser gerenciados por ele.

Importante notar a marcação “selectors”, pois ela pode gerenciar Pods criados antes da criação do Deployment pois eles terão a mesma tag.

Vamos então executar o comando para a criação deste Deployment:

```
kubectll create -f nome_do_deployment.yml
```

Em seguida vamos configurar o Deployment da nossa aplicação conforme a imagem a seguir:

```

1 apiVersion: apps/v1
2
3 kind: Deployment
4
5 metadata:
6   name: test-api-deployment
7   labels:
8     app: test-api
9
10 spec:
11   template:
12     metadata:
13       name: test-api
14       labels:
15         app: test-api
16     spec:
17       containers:
18         - name: test-api
19           image: gabrielkirsten/test-api
20           env:
21             - name: MONGODB_SERVER
22               value: database-service
23
24   replicas: 5
25   selector:
26     matchLabels:
27       app: test-api
28

```

Link para o código: <https://github.com/gabrielkirsten/containers-docker-kubernetes/blob/master/modulo-5/kubernetes-definition/deployment/test-api-deployment-definition.yml>

Importante notar que fizemos o mapeamento do nosso banco de dados para o nome de database-service, isso será para mapear para o service que será criado em etapas futuras.

Esse deployment irá configurar cinco réplicas de nossa aplicação. Ao executar o comando:

```
kubectl get pods
```

```
gk@desktop: ~  
gk@desktop:~$ kubectl get pods  
NAME                                READY   STATUS    RESTARTS   AGE  
database-deployment-5ffcb54d88-j4jrk 1/1     Running   0           23h  
test-api-deployment-7cc46584bc-gfvrg 1/1     Running   0           21h  
test-api-deployment-7cc46584bc-k64gp 1/1     Running   0           21h  
test-api-deployment-7cc46584bc-srq22 1/1     Running   0           21h  
test-api-deployment-7cc46584bc-vlq2r 1/1     Running   0           21h  
test-api-deployment-7cc46584bc-zw2l8 1/1     Running   0           21h  
gk@desktop:~$
```

Podemos notar que agora temos o Pod criado pelo Deployment do banco de dados, e temos cinco Pods criados pelo Deployment da nossa aplicação.

Ao deletar um determinado pod com o seguinte comando:

```
kubectl delete pod nome_do_pod
```

Podemos notar que o Pod irá ser instanciado novamente pois o Deployment identifica que existe um Pod a menos que o especificado e criará uma nova instância:

```
gk@desktop: ~  
gk@desktop:~$ kubectl delete pod test-api-deployment-7cc46584bc-gfvrg  
pod "test-api-deployment-7cc46584bc-gfvrg" deleted  
gk@desktop:~$ kubectl get pods  
NAME                                READY   STATUS    RESTARTS   AGE  
database-deployment-5ffcb54d88-j4jrk 1/1     Running   0           23h  
test-api-deployment-7cc46584bc-k64gp 1/1     Running   0           21h  
test-api-deployment-7cc46584bc-srq22 1/1     Running   0           21h  
test-api-deployment-7cc46584bc-vlq2r 1/1     Running   0           21h  
test-api-deployment-7cc46584bc-wdl68 1/1     Running   0           11s  
test-api-deployment-7cc46584bc-zw2l8 1/1     Running   0           21h  
gk@desktop:~$
```

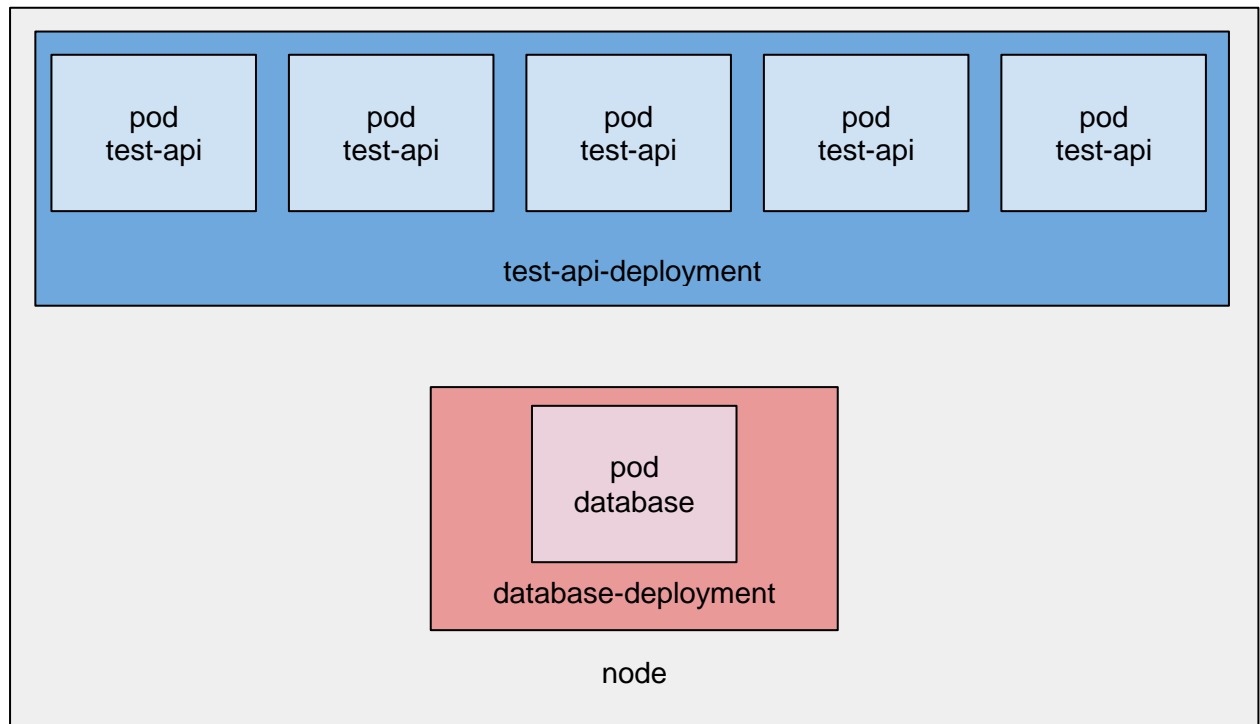
Podemos notar um dos Pods com o atributo AGE que identifica o tempo que o Pod está em execução com um valor menor do que os outros.

## Services

Quando trabalhamos com diversos Pods em um cluster de Kubernetes, muitas vezes um pod precisa se comunicar com o outro. No exemplo anterior, temos dois tipos de Pods, um para a aplicação e outro para o banco de dados. A documentação completa

dos Services pode ser vista em: <https://kubernetes.io/docs/concepts/services-networking/service/>

Até agora em nossa aplicação temos o seguinte cenário:

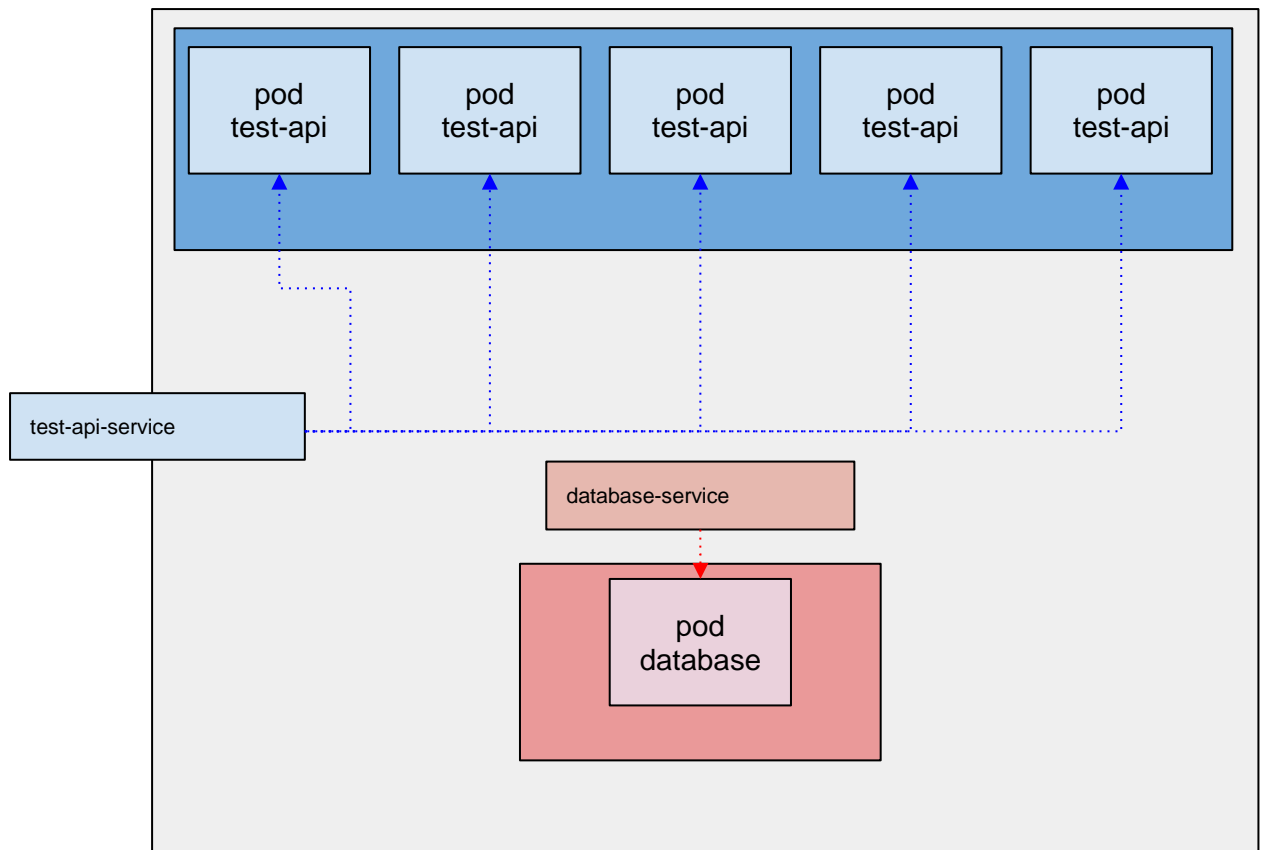


Porém tudo está acontecendo dentro de uma rede interna do Kubernetes, isso nos traz dois problemas, primeiro não temos acesso a aplicação test-api fora da rede do Kubernetes, segundo que a aplicação test-api não tem o IP do banco de dados, configurar o IP do banco de dados manualmente não é uma tarefa recomendada, pois o IP poderá ser alterado caso seja necessário colocar outra instância do banco de dados na rede.

Um service no Kubernetes vem para resolver esse problema: expor um serviço, tanto internamente para ser consumido por outros serviços dentro do Kubernetes quanto externamente, por exemplo um usuário acessar a aplicação. Vamos abordar aqui dois tipos de services:

- **ClusterIP:** Esse tipo de serviço serve para expor um serviço dentro do cluster do Kubernetes.
- **NodePort:** Esse tipo de serviço serve para expor um serviço para fora do cluster do Kubernetes.

Então agora teremos o seguinte cenário:



Adicionamos dois services, um para mapear o database e outro para mapear a nossa aplicação. Você consegue imaginar qual é o tipo de cada um deles? O test-api-service será o nosso NodePort que mapeará externamente a nossa aplicação. O database-service será responsável por mapear o nosso banco de dados para que esteja acessível dentro do cluster para a nossa aplicação.

Vamos agora criar os nossos dois arquivos YAML que descreverão os nossos dois services. Primeiramente vamos criar o service do banco de dados:

```
1 apiVersion: v1
2
3 kind: Service
4
5 metadata:
6   name: database-service
7
8 spec:
9   type: ClusterIP
10  ports:
11    - targetPort: 27017
12      port: 27017
13
14  selector:
15    app: database
16
```

Link para o arquivo: <https://github.com/gabrielkirsten/containers-docker-kubernetes/blob/master/modulo-5/kubernetes-definition/service/database-service-definition.yml>

E vamos criar o service para a aplicação test-api:

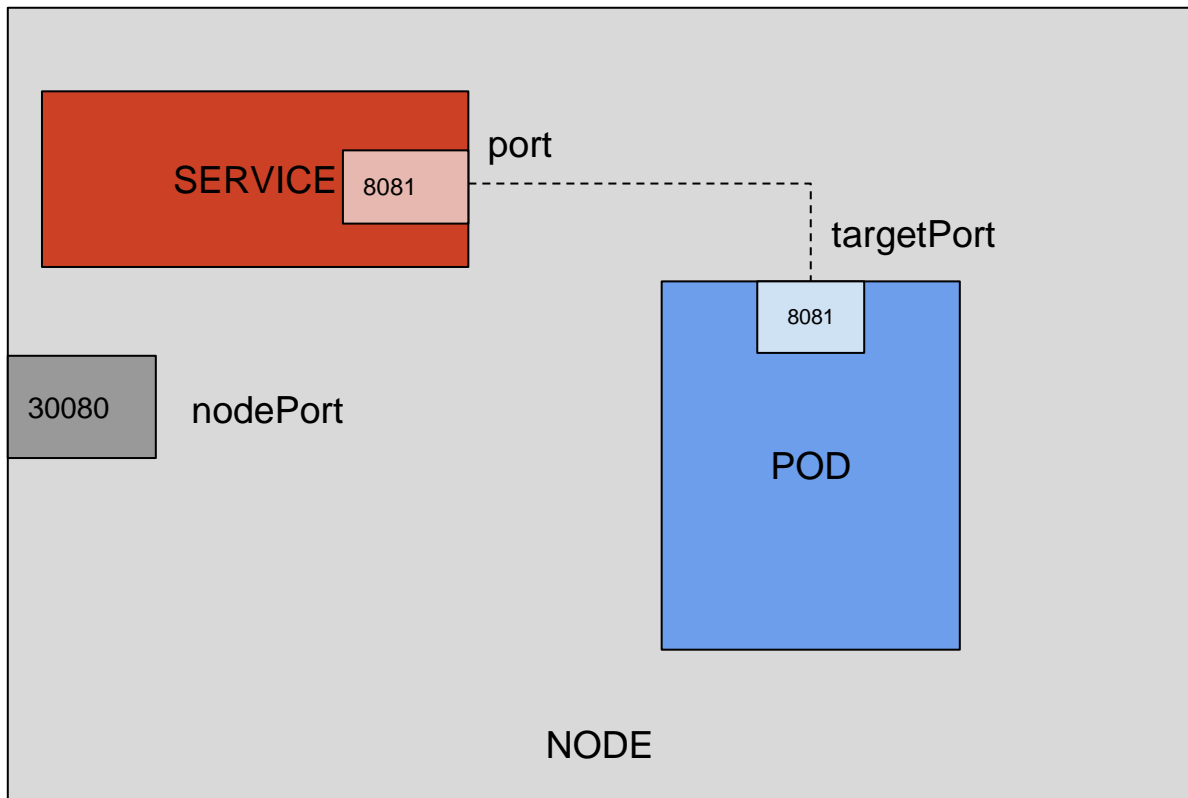
```
1 apiVersion: v1
2
3 kind: Service
4
5 metadata:
6   name: test-api-service
7
8 spec:
9   type: NodePort
10  ports:
11    - targetPort: 8081
12      port: 8081
13      nodePort: 30080
14
15  selector:
16    app: test-api
17
```

Link para o arquivo: <https://github.com/gabrielkirsten/containers-docker-kubernetes/blob/master/modulo-5/kubernetes-definition/service/test-api-service-definition.yml>



Podemos ver com esses dois arquivos, que a estrutura segue conforme vimos anteriormente com diferença em dois pontos, o “kind” que agora é Service e os “spec” que descreve o funcionamento de nosso service.

Dentro de “spec”, primeiramente temos o type que descreve o tipo do Service, como visto anteriormente. Em “ports” temos as configurações de mapeamento de portas, essa é a parte mais importante desse arquivo, “targetPort” é referente a porta do serviço que iremos conectar, no caso do test-api o serviço roda na porta 8081, port é a porta do Service que irá se conectar com o Pod e nodePort será a porta exposta no nó, conforme a imagem a seguir:



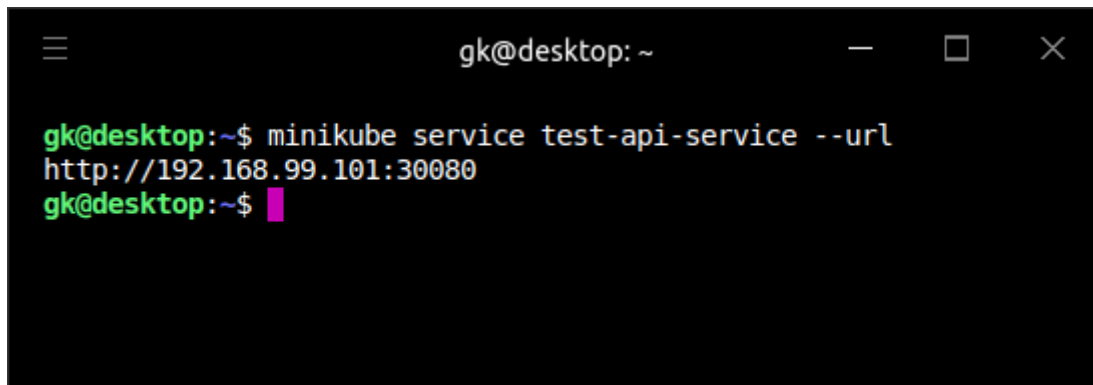
Vamos então executar o comando para a criação deste Service:

```
kubectl create -f nome_do_service.yml
```

Com essas configurações efetuadas de maneira corretas, podemos ter a nossa aplicação comunicando com o nosso banco de dados e podemos acessá-la de maneira externa, para conferir o IP de nosso Service, podemos rodar o seguinte comando:

```
minikube service test-api-service --url
```

Esse comando retornará o IP que deveremos acessar para ter acesso ao serviço:

A terminal window with a black background and green text. The window title is 'gk@desktop: ~'. The prompt is 'gk@desktop:~\$'. The command entered is 'minikube service test-api-service --url http://192.168.99.101:30080'. The prompt is now 'gk@desktop:~\$' with a red cursor.

```
gk@desktop:~$ minikube service test-api-service --url  
http://192.168.99.101:30080  
gk@desktop:~$
```

Pronto, temos uma aplicação completa rodando em nosso cluster Kubernetes.

## Considerações Finais

Primeiramente, parabéns por ter chegado até aqui, claramente você está engajado com o seu desenvolvimento e o desenvolvimento de sua carreira. Espero que essa disciplina te ajude na sua vida profissional.

Com esse capítulo conseguimos aprender uma importante ferramenta chamada Kubernetes, o gerenciamento de um cluster de containers se torna uma tarefa muito simples quando ele é aplicado. Foram ensinados os principais conceitos que fazem parte do seu funcionamento como Pods, Services e Deployments. Implementamos esses conceitos no projeto que abordamos no Docker e construímos uma arquitetura resiliente.

Espero que tenham gostado desse curso e até a próxima.