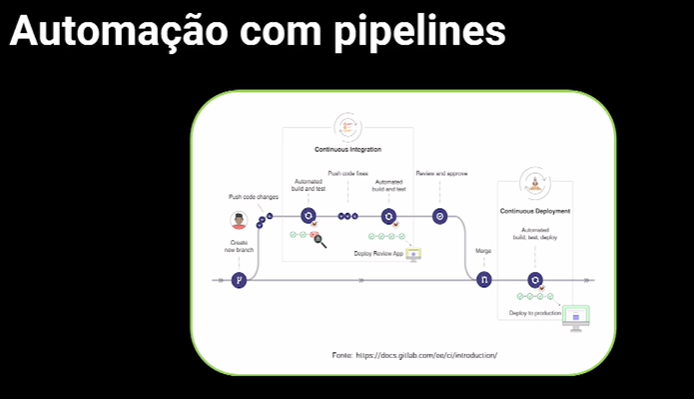
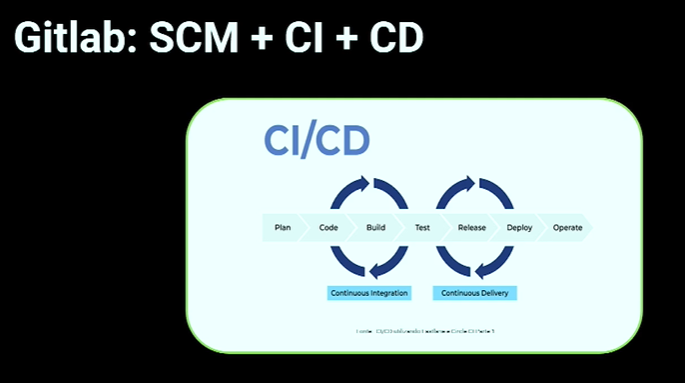
**Eficiência operacional em entrega de software:**

SCM: Source Control Management (SCM) é uma ferramenta essencial para controle de versão e colaboração em desenvolvimento de software, evitando problemas de sincronização e perdas de código.

As ferramentas de SCM permitem trabalho paralelo sem conflitos, histórico de versões para recuperação de código e exigem boas práticas, como manter o código atualizado e integrar código com maturidade, definindo um integrador responsável por lidar com conflitos.

Ferramentas como **Git** permitem colaboração em paralelo, merge de código e acompanhamento histórico, facilitando a atualização e manutenção do código de forma organizada e eficiente.

**Gitlab:**



Um **pipeline é um conjunto de etapas automatizadas que conduzem o código por diferentes estágios**, **desde a integração até a entrega**. Ele funciona como um fluxo, onde o código é verificado, testado e integrado continuamente, garantindo qualidade e feedback rápido.

As pipelines utilizam scripts para automatizar tarefas, reduzindo erros humanos e garantindo a execução consistente do processo. Através da execução automatizada de scripts, a pipeline elimina a necessidade de intervenção manual, minimizando o risco de falhas e garantindo a entrega eficiente do código.



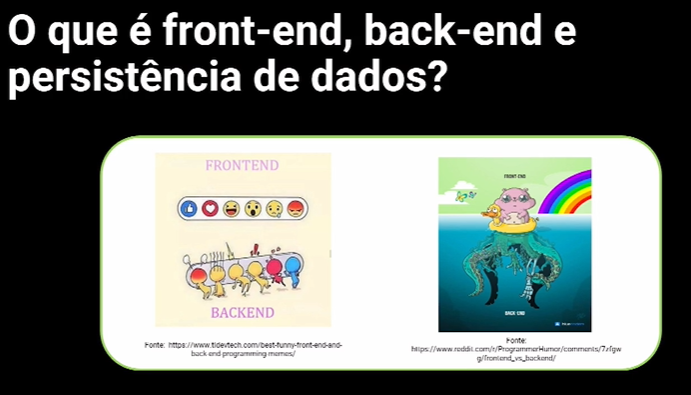
Docker é uma ferramenta de gerenciamento de containers que possibilita a execução de aplicações em ambientes isolados com os recursos mínimos necessários.

A pipeline do GitLab é composta por etapas definidas em um script, permitindo visualizar a execução de cada passo, como build, teste e deploy, com logs detalhados de cada comando.

O Docker oferece uma interface gráfica para gerenciar containers, enquanto o GitLab CI/CD, através de scripts, permite automatizar o processo de desenvolvimento e deploy, criando e gerenciando containers no Docker.

**Front-end,back-end e persistência de dados:**

Front-end, back-end e persistência de dados são componentes essenciais no desenvolvimento web. **O front-end representa a interface com o usuário**, enquanto o **back-end lida com a lógica, regras de negócio e interação com o banco de dados**, onde os dados são armazenados de forma persistente.



A interação entre front-end, back-end e banco de dados é crucial. O front-end coleta dados do usuário e os envia para o back-end, que processa, aplica regras de negócio e interage com o banco de dados para armazenar ou recuperar informações. O back-end então retorna os resultados para o front-end, que os exibe ao usuário.

**Integração com API:**

**GET para consulta**, **POST para inclusão**, **PUT para alteração** e **DELETE para exclusão de dados**.

Através de uma API, é possível manipular os dados do banco de dados H2 sem a necessidade de interação direta com o banco. As requisições HTTP, como GET, POST, PUT e DELETE, enviadas para a API, são refletidas no banco de dados, permitindo a criação, leitura, atualização e exclusão de dados.

Aplicações back-end devem fornecer funcionalidades para manipular o banco de dados através de APIs, ocultando a complexidade da interação direta com o banco de dados. Essa abordagem facilita a manutenção e o desenvolvimento da aplicação, além de garantir a segurança dos dados.

O front-end, back-end e banco de dados foram integrados, demonstrando como o front-end acessa o back-end e como isso se reflete na camada de dados. O site em React utiliza uma API para se comunicar com o back-end, que por sua vez, interage com o banco de dados H2.

O fluxo de dados entre o front-end, back-end e banco de dados foi demonstrado, juntamente com a importância da segurança na arquitetura. A API atua como intermediária entre o front-end e o banco de dados, garantindo que o front-end não acesse o banco de dados diretamente.

No desenvolvimento de aplicações web, a arquitetura em camadas, também conhecida como n-tier architecture, é um modelo amplamente utilizado que organiza os componentes de um sistema em camadas distintas, cada uma com responsabilidades específicas. Essa abordagem oferece diversas vantagens, como a separação de responsabilidades, a reutilização de código e a maior facilidade de manutenção.

Em uma arquitetura em camadas típica, temos três camadas principais: **apresentação, domínio e dados**.

A camada de **apresentação,** representada pelo front-end, é responsável pela interface com o usuário, ou seja, é a parte visual da aplicação com a qual o usuário interage, como um site ou aplicativo mobile. **É nessa camada que são tratados os elementos visuais, a interação do usuário e a navegação**.

A camada de **domínio**, por sua vez, representa o back-end e abriga as regras de negócio, a lógica da aplicação e o processamento das informações**. É nessa camada que os dados recebidos do front-end são processados, validados e utilizados para realizar as operações necessárias.** O back-end atua como um intermediário entre o front-end e a camada de dados, garantindo a integridade e a segurança das informações.

A camada de **dados**, como o próprio nome sugere, é responsável por persistir os dados da aplicação, geralmente em um banco de dados. **É nessa camada que são realizadas as operações de leitura, escrita, atualização e exclusão de dados**. A camada de dados deve ser projetada para garantir a integridade, a consistência e a segurança dos dados armazenados.

A comunicação entre as camadas é fundamental para o funcionamento da aplicação. O front-end se comunica com o back-end por meio de APIs (Application Programming Interfaces), que definem um conjunto de regras e especificações que permitem a comunicação entre diferentes sistemas. O back-end, por sua vez, se comunica com a camada de dados utilizando bibliotecas e frameworks específicos para cada tipo de banco de dados.

Com a evolução da arquitetura de software, o modelo de microserviços surgiu como uma alternativa para lidar com a crescente complexidade das aplicações. Diferentemente da arquitetura monolítica, em que todos os componentes da aplicação estão interligados em um único bloco, os microserviços permitem dividir a aplicação em serviços menores, independentes e autônomos.

Essa abordagem traz diversas vantagens, como a possibilidade de utilizar diferentes tecnologias e linguagens de programação em cada serviço, a escalabilidade independente de cada serviço, a maior facilidade de manutenção e a tolerância a falhas. Cada microserviço é responsável por uma funcionalidade específica da aplicação e se comunica com outros microserviços por meio de APIs.

A segurança é um aspecto crucial em qualquer aplicação, especialmente em arquiteturas em camadas e microserviços. É fundamental garantir que a comunicação entre as camadas seja segura, utilizando mecanismos como autenticação, autorização e criptografia. Além disso, é importante proteger a camada de dados contra acessos não autorizados, utilizando firewalls, senhas fortes e outros mecanismos de segurança.

Em resumo, a arquitetura em camadas e os microserviços são modelos de desenvolvimento de software que oferecem diversas vantagens em termos de organização, escalabilidade, manutenção e segurança. A escolha do modelo mais adequado depende das necessidades específicas de cada aplicação, mas é fundamental compreender os princípios e as características de cada abordagem para tomar a decisão mais acertada.

**Docker:**

Docker é uma ferramenta de **gerenciamento de containers** que permite a criação de ambientes isolados e portáteis para aplicações. Ele funciona como um hypervisor, gerenciando os recursos da máquina e permitindo a execução de containers com diferentes sistemas operacionais.

Docker permite o gerenciamento de versões de software através de imagens. Cada imagem representa um estado específico da aplicação, permitindo a fácil reversão para versões anteriores. A utilização de containers agiliza o processo de deploy, pois o sistema operacional já está carregado.

O Docker Hub é um repositório online de imagens Docker, contendo desde imagens oficiais de empresas até imagens personalizadas de outros usuários. Ele é usado para baixar imagens e subir containers, agilizando o processo de desenvolvimento e implantação.

Uma **imagem Docker contém tudo o que a aplicação precisa para ser executada, incluindo bibliotecas e dependências**. Ela é composta por camadas, com uma base (como OpenLiberty e OpenJ9 para aplicações Java), bibliotecas e a aplicação em si no topo.

Imagens Docker são construídas em camadas, com uma base imutável e camadas superiores que podem ser modificadas. A base fornece o ambiente de execução, como um sistema operacional e bibliotecas principais, enquanto as camadas superiores contêm a aplicação e suas dependências específicas.

As versões de imagem no Docker, chamadas de tags, representam diferentes estados de uma imagem. A tag "latest" indica a versão mais recente, mas é possível usar versões específicas para garantir estabilidade ou compatibilidade.

Para executar um container com uma versão específica, utilize a tag desejada após o nome da imagem no comando `docker run`. É possível baixar uma imagem sem iniciar um container usando o comando `docker pull`, o que pode ser útil para preparar o ambiente.

Criando o Dockerfile:

O Dockerfile é um arquivo utilizado para criar imagens Docker, definindo as instruções e configurações necessárias para construir o ambiente da aplicação. Através dele, é possível automatizar a criação de imagens personalizadas, garantindo a consistência e a reprodutibilidade do ambiente em diferentes máquinas e plataformas.



Começa sempre com o From indicando qual é a “base” da imagem a ser construída.

Ubuntu:18.04 é a imagem básica (sempre começa com uma imagem básica).

Toda imagem tem uma base à qual se sustentar, vamos dizer que essa base possui o mínimo necessário para o container rodar quando for criado um container à partir da imagem que estamos construindo. Neste caso a “base” usada é uma versão bem light do Ubuntu Linux, versão 21.10. Se estivéssemos montando uma imagem com uma aplicação em Java, por exemplo, poderíamos usar outra base como a “maven:3.8.5-openjdk-17-slim” que já vem com um mínimo necessário para rodar Java, mas veremos isso mais pra frente.

Então pensa assim: imagine que você esteja em uma versão mínima de um Ubuntu Linux, você poderia executar um comando para atualizar os pacotes que tem no Ubuntu através do comando “apt-get update” e instalar um pacote específico como o “nginx” por exemplo através do comando “apt-get install -y nginx”.

Bom, é exatamente isso que estamos fazendo nesse comando acima, só que está sendo feito de uma vez várias coisas, por enquanto não precisaremos entrar no detalhe, mas o que precisa ser conhecido é que o comando RUN no script do Dockerfile executa instruções dentro da “base”, é como se eu entrasse dentro do Ubuntu dessa base e digitasse esses comandos para fazer algo. O comando RUN só é executado enquanto estivermos construindo a imagem, lembre-se disso!

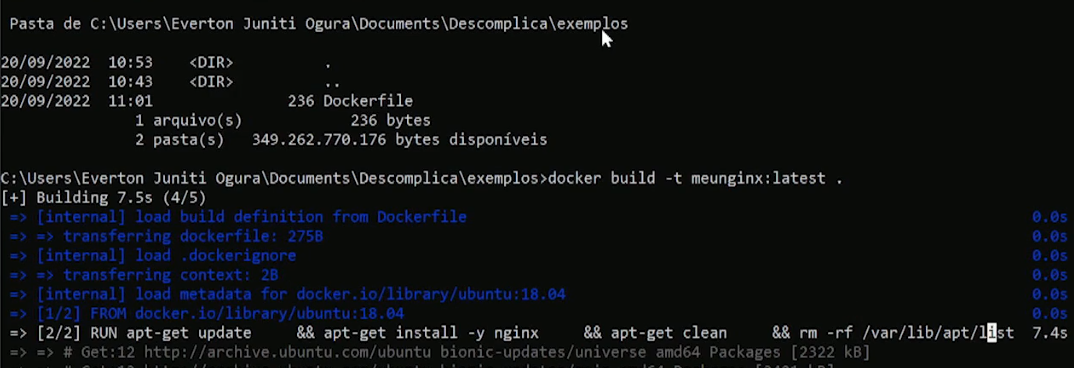
Run .... atualizar o sistema com os últimos pacotes;

Get install..... instalando o nginx

Echo “daemon off;” --- mensagem no arquivo

O comando CMD é parecido com o comando RUN, mas com uma diferença essencial: ele executa algo da “base” (nosso Ubuntu), só que somente quando um container é iniciado através dessa imagem que foi construída por nós, ou seja, enquanto o RUN só roda enquanto estamos construindo a imagem, o CMD só irá rodar quando um container iniciar, ele não é executado enquanto estivermos construindo nossa imagem.

No CMD:



Colocar Docker inicialmente e depois o build.

Para construir uma imagem Docker a partir de um Dockerfile, utiliza-se o comando `docker build -t <nome da imagem>:<tag> .`. O comando `docker image ls` lista as imagens Docker disponíveis, incluindo a recém-criada.

O parâmetro “-t meunginx:latest” indica o nome de “tag” que esta imagem terá, é através dela que indicamos ao Docker qual é a imagem que usaremos ao tentar criar um container. O formato é o nome do repositório antes dos dois pontos e a versão após os dois pontos. Lembrando que este parâmetro é opcional, mas altamente recomendável incluir!

A inspeção detalhada de uma imagem Docker pode ser feita com o comando `docker image inspect <nome da imagem>:<tag>`. Esse comando revela informações sobre o sistema de arquivos da imagem, comandos executados durante a construção, a porta exposta (se houver) e o comando que será executado ao iniciar um container a partir dessa imagem.

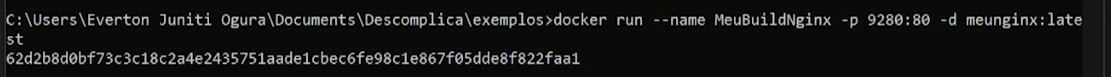
**Repositório local X Repositório remoto:**

O Docker Hub armazena imagens Docker em repositórios, sendo o local padrão para imagens públicas e privadas.

É crucial ter cuidado com imagens não oficiais do Docker Hub, pois seu conteúdo e segurança são desconhecidos.

**Subindo um container:**

Para subir um container Docker a partir de uma imagem, utilize o comando `docker run`. Defina um nome para o container com `--name`, especifique a porta de exposição com `-p` e utilize `-d` para rodar em background. Por fim, informe o nome da imagem e a tag.



É possível criar múltiplas imagens com diferentes tags a partir de um Dockerfile, permitindo gerenciar e subir versões específicas de uma aplicação.

O comando “docker run” instrui ao Docker que é para executar um container.

O parâmetro “–name MeuNginx” indica um apelido que estamos dando ao container, que poderá ser consultado mais tarde.

O parâmetro “-p 9080:80” indica que poderemos acessar “de fora” do container via porta 9080 e que o Docker fará a “tradução” de tudo que vier pela porta 9080 de fora, para a porta 80 de dentro do container. Haverá uma explicação mais detalhada sobre as portas, no momento não se preocupe.

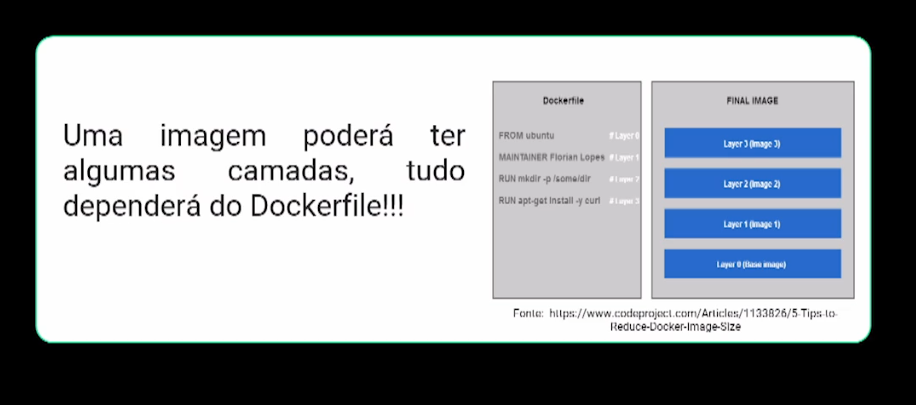
O parâmetro “-d” indica que o container subirá e ficará rodando sem precisarmos estar conectados nele, é o oposto do “-it” em que o container sobe no modo interativo, em que ficamos conectados dentro do container assim que ele sobe.

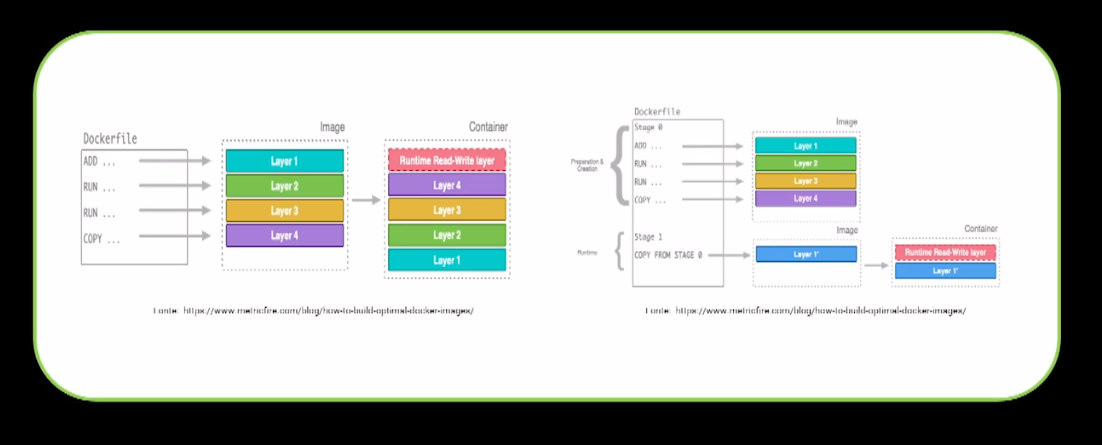
O último parâmetro “meunginx:latest” é o nome da imagem e a versão que iremos usar para subir o container, que neste exemplo é a imagem recém criada.

**Layered System:**

As camadas são componentes essenciais das imagens Docker, representando instruções do Dockerfile. Cada instrução, como FROM, RUN, e outras, forma uma camada. A camada 0 é sempre a imagem base, definida pela instrução FROM.

A otimização de camadas é crucial para a eficiência das imagens Docker. Menos camadas resultam em imagens menores e builds mais rápidos. Combinar múltiplos comandos em um único RUN, por exemplo, reduz o número de camadas e melhora o desempenho.





**Tageamento:**

Para criar diferentes versões de uma imagem Docker, utilize o comando `docker build -t <nome da imagem>:<tag> .`. A tag diferencia as versões, permitindo gerenciar e executar containers específicos.

Há 2 abordagens: ou você literalmente substitui a imagem atual, tendo que destruir todos os containers atrelados a esta imagem, remover a imagem e criar a imagem novamente, ou você pode usar tags de versão de imagem.

Utilize `docker run -d -p <porta externa>:<porta interna> <nome da imagem>:<tag>` para executar um container de uma versão específica da imagem, substituindo `<tag>` pela tag desejada.

O latest indica a versão da imagem, se tivermos um fluxo em que possamos construir várias imagens, podemos utilizar também o seguinte:

docker build -t meunginx:1.0 .

docker build -t meunginx:2.0 .

Geralmente a palavra “latest” significa ser “a última versão”, ou “a versão mais recente” e tudo bem usar ela para indicar que é a última versão.

Além da “latest”, você pode ir armazenado várias versões como no exemplo acima, supondo que haja alguma diferença em cada versão.

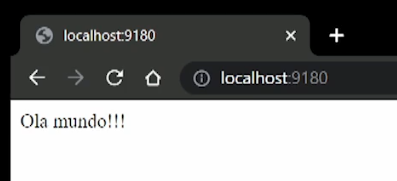
Assim ao subir um container, podemos escolher uma das versões e não obrigatoriamente a última.

Isso é útil quando você tem algo em produção e aí surgem versões mais novas, mas você não quer ir direto para a última versão (que pode ter bugs desconhecidos, justamente por ser a última), então você pode optar por uma versão “mais estável”.

**Subindo o “olá mundo” com o NGINX:**

Subir uma aplicação "Olá Mundo" com NGINX é simples. Após verificar se o Docker está em execução, utilize o comando 'docker run' com o nome desejado para o container, defina a porta e a imagem 'NGINX latest'. Após a execução, acesse a aplicação através do navegador na porta definida.





Para utilizar volumes no Docker com NGINX, utilize o comando 'docker run' com a flag '-v' seguida do caminho da pasta local e o caminho interno do container, separados por dois pontos. O conteúdo da pasta local será acessível dentro do container no caminho especificado.

**Volumes:**

Volumes no Docker são como gavetas externas que armazenam arquivos usados por containers. Sem volumes, os arquivos são excluídos quando o container é destruído. Ao usar volumes, os arquivos persistem, permitindo acesso contínuo mesmo após a reinicialização do container.

Volumes no Docker funcionam como links entre uma pasta externa e um diretório interno do container. Isso permite que arquivos externos sejam acessados como se estivessem dentro do container, garantindo persistência de dados e facilitando a atualização de arquivos sem precisar reconstruir a imagem.

Volumes permitem compartilhar arquivos e diretórios entre o sistema host e os containers Docker. Isso é útil para persistir dados, compartilhar dados entre containers e manter a imagem do container enxuta.

A flag `--mount` oferece uma sintaxe mais detalhada para configurar volumes, permitindo especificar o tipo de volume (`bind`, `volume`, `tmpfs`), a origem (`src`) e o destino (`dst`) do volume.

**Subindo um banco de dados:**

Para utilizar o MySQL é necessário instalar o software MySQL Workbench, que permite acessar e gerenciar bancos de dados MySQL de forma mais fácil.

Através do Docker, podemos criar um container com o MySQL, definindo variáveis de ambiente como senha do root e criar um novo usuário com privilégios específicos. Após a criação, podemos estabelecer uma conexão com o container utilizando o MySQL Workbench.

Dentro do esquema criado no banco de dados, podemos criar tabelas com suas respectivas colunas e tipos de dados.

Após a criação, podemos inserir dados utilizando comandos SQL.Ao finalizar a utilização do container, é fundamental entender que os dados inseridos não são persistidos. Ao remover o container, os dados são perdidos, o que demonstra a importância de mecanismos de persistência de dados em ambientes Docker.

CUIDADO: Se um container com um volume associado for excluído e um novo container for iniciado com a mesma imagem, o Docker criará um novo volume para o novo container, resultando na perda de acesso aos dados do volume anterior.

**Criando um volume para o nosso banco de dados:**

Para utilizar volumes com Docker e MySQL, primeiro crie um volume com "docker volume create <nome\_do\_volume>". Em seguida, ao iniciar o container MySQL, utilize a flag "-v" para mapear o volume à pasta desejada no container, por exemplo, "-v <nome\_do\_volume>:/var/lib/mysql".

A persistência de dados com volumes no MySQL permite que os dados sejam mantidos mesmo após o container ser reiniciado ou removido. Isso é possível porque os dados são armazenados no volume, que é independente do container.

Após a inicialização do MySQL com um volume, as variáveis de ambiente não são mais necessárias para inicializações subsequentes. O container utilizará as configurações armazenadas no volume.

**Acessando de dentro e de fora:**

O host é um sistema operacional que serve como base para os containers, gerenciando e distribuindo recursos da máquina local, como CPU, memória e rede. É ele quem permite que os containers acessem e utilizem esses recursos de forma isolada.

**Qual a diferença entre sistemas operacionais?**

Utilizar Windows ou Linux como host para Docker depende da necessidade. Softwares para Windows exigem host Windows, enquanto Linux é popular para servidores e empresas.

No Mac, o host utilizado é o Linux. Já no Windows, é possível escolher entre usar o próprio Windows ou o Linux como host. A escolha depende do software a ser entregue.

Linux é o host mais popular no Docker Hub, com mais imagens disponíveis. Suas imagens costumam ser menores e sua estrutura mais simples, utilizando Ubuntu como base.

**E o melhor?**

A escolha do host ideal, seja Windows ou Linux, depende das necessidades da empresa e da equipe de desenvolvimento. Se a empresa utiliza predominantemente Windows, um host Windows faz sentido. Se há uma mescla de sistemas, é possível ter servidores com hosts diferentes, mas a gestão de desenvolvimento se torna crucial para garantir a compatibilidade das aplicações com cada host.

**Porta:**

Portas virtuais são interfaces de software que permitem a comunicação entre processos e aplicações em um sistema operacional, diferenciando-as por meio de números de porta específicos.

As portas virtuais são essenciais para identificar e direcionar o tráfego de rede para a aplicação correta em um servidor, evitando conflitos e garantindo que os dados cheguem ao destino desejado.

Cada porta virtual é única em um sistema e não pode ser utilizada por duas aplicações simultaneamente, semelhante à restrição de conectar dois cabos em uma única porta física.



**GitLab CI/CD:**

O GitLab CI-CD é uma ferramenta integrada ao GitLab que permite a criação de pipelines, que são fluxos de trabalho automatizados definidos em um arquivo de script chamado `.gitlab-ci.yml`.

O script `.gitlab-ci.yml` define os estágios da pipeline, que são os passos a serem executados. Esses estágios podem incluir testes, build, implantação e outras tarefas personalizadas.

A pipeline é executada automaticamente a cada alteração no código-fonte, disparando o fluxo de CI/CD. O script permite personalização, como variáveis de ambiente e fluxos condicionais, para controlar o comportamento da pipeline.

**GitLab Runner:**

O GitLab Runner é um executor que viabiliza o funcionamento das pipelines no GitLab CI/CD, atuando como um motor que executa os scripts de teste, build e implantação.

Runners públicos, embora disponíveis no GitLab, não permitem interação com a máquina local. Para usar o Docker localmente, é necessário instalar um Runner na própria máquina, permitindo que ele acesse os recursos locais e execute as etapas da pipeline.

**Job:**

Um job no GitLab CI/CD é uma tarefa executada pelo GitLab Runner dentro de um container temporário. Cada job representa um estágio (stage) na pipeline, como testes unitários ou build da aplicação.

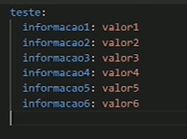
Na prática, o GitLab Runner, atuando como orquestrador, cria containers temporários para executar cada job definido na pipeline. Cada container representa um trabalho específico, como executar testes unitários ou construir a imagem da aplicação.

Jobs são trabalhos realizados pelo GitLab Runner e definidos no script de pipeline através das stages. Uma pipeline pode ter vários jobs, sendo cada um responsável por uma etapa do processo de CI/CD.

**YAML:**

YAML é um formato de arquivo legível por humanos e máquinas, ideal para pipelines. Diferente de XML e JSON, YAML utiliza indentação para estrutura e dois pontos para separar atributos de seus valores.

Para usar YAML em pipelines do GitLab, o script deve estar em um arquivo chamado ".gitlab-ci.yml" no repositório. Utilize apenas um arquivo YAML por repositório para gerenciar as pipelines.



**Lint:**

Para garantir a qualidade do código e evitar erros na pipeline, podemos utilizar o Lint. O Lint é uma ferramenta que analisa o código em busca de erros de sintaxe, estilo e boas práticas.

O GitLab oferece uma ferramenta de validação de scripts YAML chamada CILint. O CILint verifica se a sintaxe do arquivo .gitlab-ci.yml está correta e se as stages e jobs estão configurados corretamente, evitando assim que pipelines sejam executadas com erros.

**Comandos de Script:**

É possível executar comandos na pipeline que não resultem em um container, como comandos para listar containers, imagens e redes Docker. Esses comandos podem ser úteis para depurar ou obter informações sobre o ambiente de execução.

A diretiva 'concurrency' no arquivo de configuração do GitLab Runner controla quantos jobs podem ser executados simultaneamente. Aumentar esse valor permite a execução paralela de jobs, acelerando a pipeline.

**Usando Imagens:**