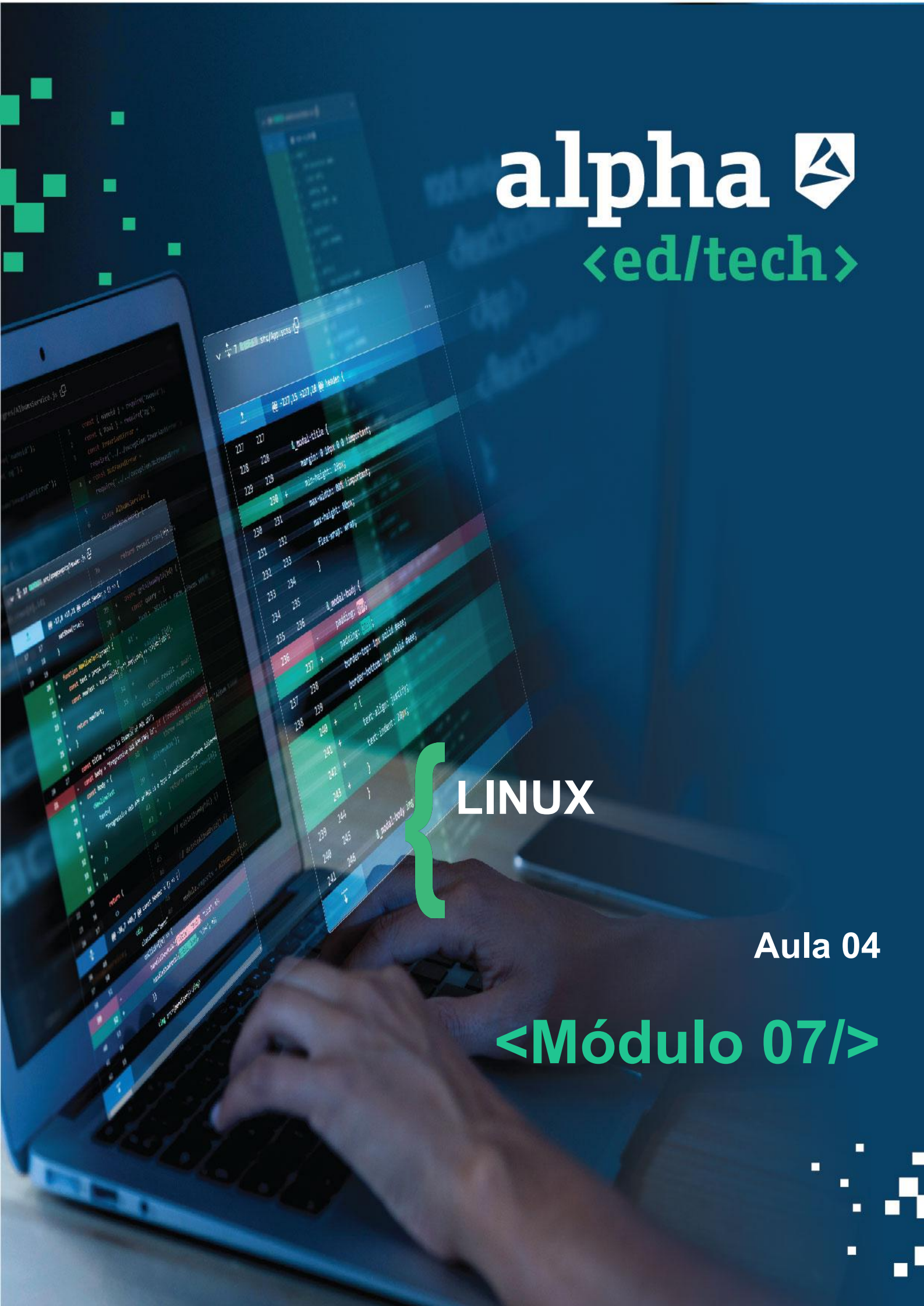

alpha
<ed/tech>
LINUX
Aula 04
<Módulo 07/>

Introdução

Na era digital, a gestão remota de servidores é uma prática essencial para administradores de sistemas, e o **Secure Shell** (SSH) emerge como uma ferramenta fundamental nesse contexto. Vamos explorar o universo do SSH, começando por compreender o que é esse protocolo de rede seguro. Ao longo desta jornada, mergulharemos no processo de acesso remoto via SSH no terminal Linux, abordando a instalação do servidor SSH e os passos necessários para estabelecer uma conexão segura. Além disso, exploraremos a criação de chaves SSH, um método robusto de autenticação que eleva a segurança das comunicações, proporcionando aos administradores uma abordagem eficaz e confiável para gerenciar sistemas remotamente.

Secure Shell

O SSH (**Secure Shell** ou **Secure Socket Shell**) é um protocolo de rede que oferece aos usuários, em particular aos administradores de sistemas, uma forma segura de acessar a um computador através de uma rede não segura. O SSH permite conectar-se a servidores remotos e realizar tarefas administrativas, como gerenciamento de arquivos e processos. Com ele, é possível acessar seus servidores de forma segura e privada, sem precisar se preocupar com invasões ou interceptações de dados.

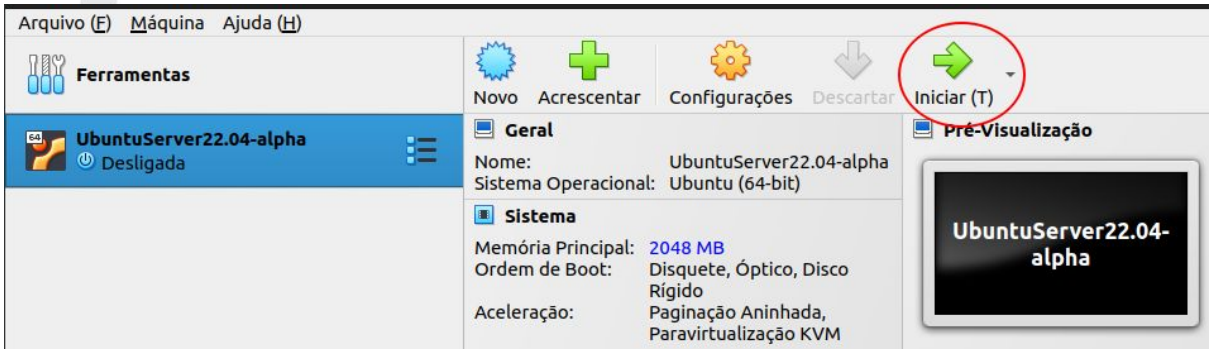
O Secure Shell fornece autenticação de senha e autenticação de chave pública, bem como comunicações de dados criptografadas entre dois computadores conectados em uma rede aberta, como a Internet. O protocolo SSH é dividido em três camadas: a camada de transporte, a camada de autenticação e a camada de sessão. Vamos falar sobre cada uma delas:

- **Camada de transporte** - A camada mais fundamental do SSH, incumbida de estabelecer e administrar uma conexão segura entre o cliente e o servidor. Sua função primordial consiste em criptografar todas as informações transmitidas, prevenindo interceptações não autorizadas e garantindo a confidencialidade dos dados.
- **Camada de autenticação** - Posicionada no nível intermediário do SSH, esta camada assume a responsabilidade de verificar a identidade do usuário conectando-se ao servidor. Esse processo de verificação é realizado por meio de autenticação de senha ou, alternativamente, por meio de uma chave SSH.
- **Camada de sessão** - A camada superior do SSH, encarregada de gerenciar as sessões estabelecidas entre o cliente e o servidor. Além de facultar a execução de comandos e a transferência de arquivos por parte do usuário, ela possibilita que o servidor envie informações de volta ao cliente.

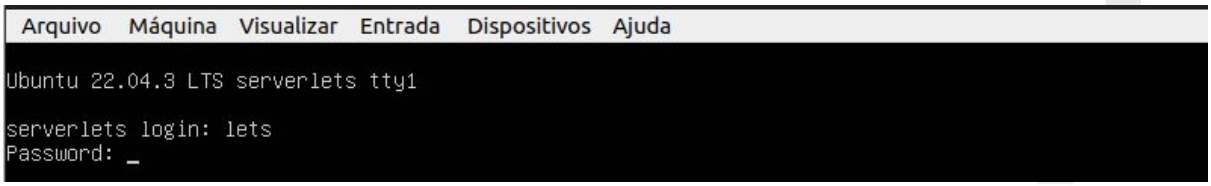
Dependendo de como você seguiu a trilha até agora, você pode ter usado SSH com autenticação login e senha. Ainda nesta aula, vamos aprender como gerar uma chave SSH. A principal vantagem é a praticidade de conseguir se conectar remotamente sem precisar fazer login.

Instalação SSH server

Ligue sua máquina virtual. Dessa vez, não vamos mantê-la ativa em background, nem fazer conexão ssh com login/senha.



Faça o login passando nome de usuário e senha.

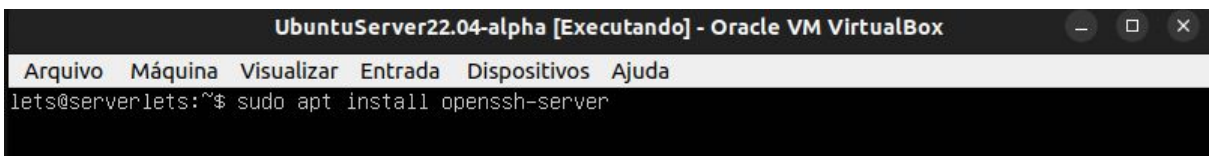


Conforme explicando em outras aula, é uma boa prática fazer a atualização de pacotes antes de instalar algum programa. Aplique esses comandos:

sudo apt update
sudo apt upgrade

OpenSSH é uma versão disponível gratuitamente da família de ferramentas do protocolo SSH para controlar remotamente ou transferir arquivos entre computadores. As ferramentas tradicionais utilizadas para realizar essas funções, como telnet ou rcp, são inseguras e transmitem a senha do usuário em texto não criptografado quando utilizadas. OpenSSH fornece um daemon de servidor e ferramentas de cliente para facilitar operações seguras e criptografadas de controle remoto e transferência de arquivos.
O próximo passo é instalar o openssh-server, através desse comando:

sudo apt install openssh-server



Obs: Dependendo de como você procedeu nas aulas anteriores, você já tem instalado o openssh-server.

Para iniciar o serviço, aplique:

sudo systemctl start ssh

Para verificar se o serviço está em execução:

sudo systemctl status ssh

```
lets@serverlets:~$ sudo systemctl start ssh
lets@serverlets:~$ sudo sytemctl status ssh
sudo: sytemctl: command not found
lets@serverlets:~$ sudo systemctl status ssh
● ssh.service - OpenBSD Secure Shell server
     Loaded: loaded (/lib/systemd/system/ssh.service; enabled; vendor preset: enabled)
     Active: active (running) since Mon 2024-02-12 13:43:18 UTC; 26min ago
       Docs: man:sshd(8)
             man:sshd_config(5)
    Process: 35216 ExecStartPre=/usr/sbin/sshd -t (code=exited, status=0/SUCCESS)
   Main PID: 35221 (sshd)
      Tasks: 1 (limit: 2221)
     Memory: 2.0M
        CPU: 10ms
     CGroup: /system.slice/ssh.service
             └─35221 "sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups"

Feb 12 13:43:18 serverlets systemd[1]: Stopping OpenBSD Secure Shell server...
Feb 12 13:43:18 serverlets systemd[1]: ssh.service: Deactivated successfully.
Feb 12 13:43:18 serverlets systemd[1]: Stopped OpenBSD Secure Shell server.
Feb 12 13:43:18 serverlets systemd[1]: Starting OpenBSD Secure Shell server...
Feb 12 13:43:18 serverlets sshd[35221]: Server listening on 0.0.0.0 port 22.
Feb 12 13:43:18 serverlets sshd[35221]: Server listening on :: port 22.
Feb 12 13:43:18 serverlets systemd[1]: Started OpenBSD Secure Shell server.
lets@serverlets:~$
```

A porta padrão do protocolo SSH é a 22. Caso essa porta não esteja habilitada ainda, aplique os seguintes comandos para que o **Uncomplicated Firewall** (ufw) permita tráfego na porta 22 e reiniciar o serviço do openssh-server:

sudo ufw allow 22
sudo systemctl restart ssh

Após esses passos, o servidor SSH está instalado e em execução na sua máquina virtual Ubuntu Server. Você pode se conectar a ela remotamente usando um cliente SSH, como o terminal no Linux ou macOS, terminal Windows para versões >=10, ou PuTTY no Windows para versões < 10.

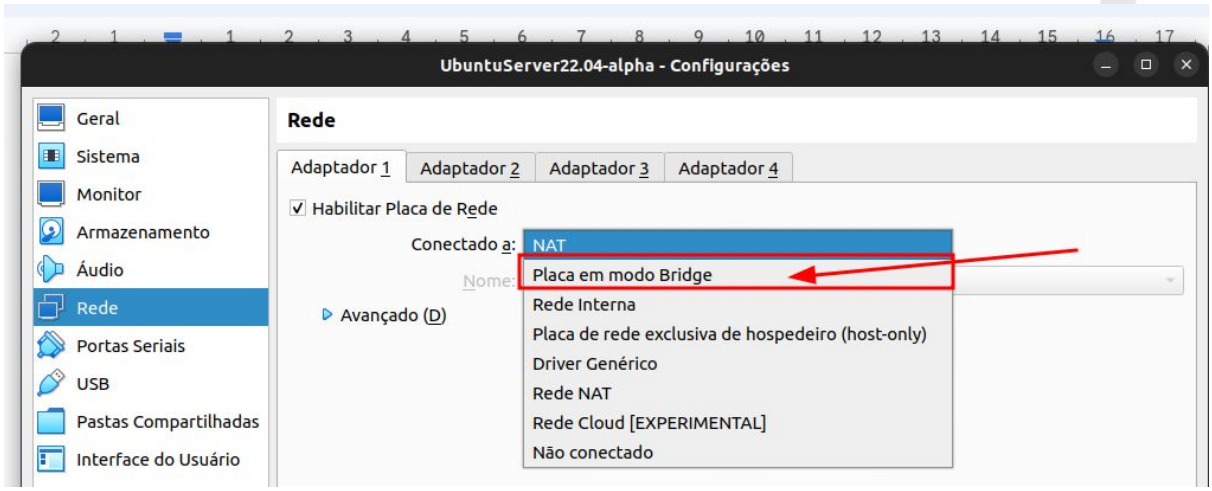
Ainda no servidor da sua máquina virtual, caso não saiba qual o endereço IP dela, aplique o comando a seguir:

ip a

```
Arquivo  Máquina  Visualizar  Entrada  Dispositivos  Ajuda
lets@serverlets:~$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 100
0
    link/ether 08:00:27:00:c5:86 brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.102/24 metric 100 brd 192.168.0.255 scope global dynamic enp0s3
       valid_lft 83362sec preferred_lft 83362sec
    inet6 fe80::a00:27ff:fe00:c586/64 scope link
       valid_lft forever preferred_lft forever
lets@serverlets:~$ _
```

Caso sua máquina virtual **não tenha endereço IP**, desligue-a; vá na interface da VirtualBox e, com a sua virtual machine (VM) selecionada, clique em configurações (settings).

Clique em Rede (Network) no menu à esquerda. Em seguida, em Adapter 1 (Adaptador 1), mude de NAT para Bridged Adapter (Placa em modo Bridge). Por fim, clique em Ok.



Finalmente, ligue sua máquina virtual novamente. Ela receberá um endereço IP na sua rede local. Consulte o endereço IP através do comando "**ip a**".

Acesso ao SSH no cliente

Caso sua máquina local seja **Linux** ou **MacOS**, você pode simplesmente utilizar o terminal para estabelecer uma conexão com o servidor remoto via SSH. Isso se deve ao fato de que ambos os sistemas operacionais já vêm com o cliente SSH instalado por padrão. Essa facilidade no acesso remoto via terminal torna a utilização do SSH uma escolha conveniente e eficiente para usuários dessas plataformas.

No caso do **Windows**, dependerá de qual versão você possui. Até pouco tempo, conectar-se via SSH usando o Windows, requeria a utilização de um cliente SSH, sendo que o mais comum, era utilizar o **Putty**, o mais conhecido e utilizado programa do gênero, que é muito bom, simples, leve e gratuito.

Desde a atualização do **Windows 10** (Windows 10 Fall Creators Update), utilizar o Putty não é mais necessário. Os Windows 10 e 11 contam com um cliente de SSH (OpenSSH) embutido no sistema operacional.

Acesso via login

Caso você nunca tenha feito acesso via SSH, passando login e senha, essa será a sua primeira vez. Na sua máquina local, abra o terminal. A sintaxe é a seguinte:

ssh usuario@endereco_ip

```
letonio.silva@BRRIOLN043879:~ $ ssh lets@192.168.0.102
lets@192.168.0.102's password:
Welcome to Ubuntu 22.04.3 LTS (GNU/Linux 5.15.0-91-generic x86_64)
```

Parabéns, você está acessando o seu servidor de forma remota usando SSH. Porém, existe uma abordagem onde não é necessário informar as credenciais na autenticação. Isso é feito através da criação de uma chave pública que é enviada para seu servidor. Assim, quando você desejar se conectar, basta passar ssh usuario@endereco_ip.

Criação de chave SSH para autenticação

Embora seja útil fazer login em um sistema remoto usando senhas, é uma ideia muito melhor configurar a autenticação baseada em chaves. A autenticação baseada em chaves funciona criando um par de chaves: uma **chave privada** e uma **chave pública**.

A **chave privada** está localizada na máquina cliente e é protegida e mantida em segredo.
A **chave pública** pode ser fornecida a qualquer um ou colocada em qualquer servidor que você queira acessar.

Quando você tentar se conectar usando um par de chaves, o servidor usará a chave pública para criar uma mensagem para o computador cliente que só pode ser lida com a chave privada. O computador cliente então envia a resposta adequada de volta ao servidor e o servidor saberá que o cliente é legítimo.Todo esse processo é feito automaticamente depois de você configurar as chaves.

Com o terminal da sua máquina local aberto, gere um par de chaves (uma pública e uma privada), através desse comando:

ssh-keygen -t rsa -b 2048

```
lets@serverlets:~$ ssh-keygen -t rsa -b 2048
Generating public/private rsa key pair.
Enter file in which to save the key (/home/lets/.ssh/id_rsa):
```

Pressione **Enter** para aceitar o local padrão para salvar a chave (geralmente ~/.ssh/id_rsa). Em seguida, você será perguntado se deseja inserir uma **passphrase**. Optei por não inserir nenhuma, pressionando Enter, mas sinta-se livre para proceder como desejar.

```
letonio.silva@BRRIOLN043879:~ $ ssh-keygen -t rsa -b 2048
Generating public/private rsa key pair.
Enter file in which to save the key (/home/letonio.silva/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/letonio.silva/.ssh/id_rsa
Your public key has been saved in /home/letonio.silva/.ssh/id_rsa.pub
```

A sua chave pública fica salva no arquivo **~./.ssh/id_rsa.pub**

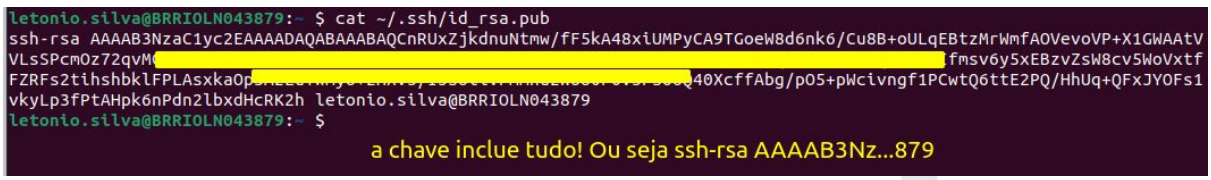
Você pode abrir o arquivo usando um editor de texto ou usar o comando cat para imprimir no terminal o conteúdo. Usando nano, temos que aplicar esse comando:

nano ~./.ssh/id_rsa.pub

Aplicando o comando cat, temos:

cat ~./.ssh/id_rsa.pub

É importante ressaltar que sua chave pública abrange o conteúdo todo: "**ssh-rsa AAAAB...879**"



O próximo passo é copiar a chave pública e mandar para o servidor remoto, para que ele guarde a informação de sua chave pública como uma "**chave autorizada**". Para fazer essa cópia, temos dois caminhos, mostrados nos próximos tópicos.

Enviar a chave pública - caminho 1

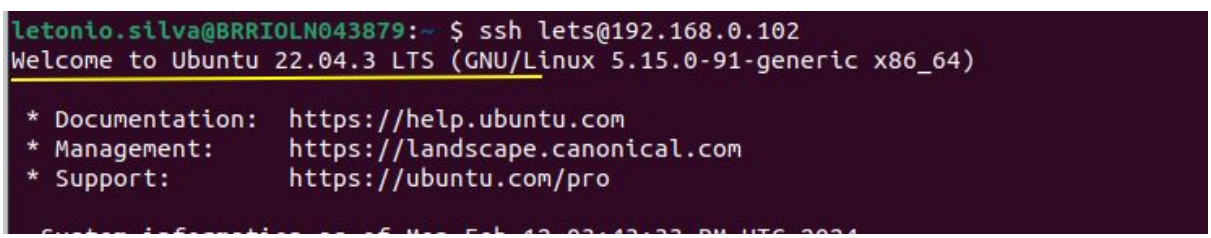
Usar o comando **ssh-copy-id**.

A sintaxe é:

ssh-copy-id usuario@ip_do_servidor

Na imagem abaixo, inserimos o comando. Em seguida, pede-se a senha do usuário para confirmar a adição da chave. Foi adicionado com sucesso.

Faça a conexão usando apenas **ssh usuario@ip_do_servidor**. Note que não será necessário fazer autenticação via senha:



Conexão realizada com sucesso!

Enviar a chave pública - caminho 2

Caso você não tenha acesso ao comando **ssh-copy-id**, você pode fazer uso do comando abaixo (lembre-se de substituir usuário e ip_do_servidor):

cat ~/.ssh/id_rsa.pub | ssh usuario@ip_do_servidor 'cat >> ~/.ssh/authorized_keys'

```
letonio.silva@BRRIOLN043879:~ $ cat ~/.ssh/id_rsa.pub | ssh lets@192.168.0.102 'cat >> ~/.ssh/authorized_keys'
letonio.silva@BRRIOLN043879:~ $
```

Verificar as permissões

Um ponto interessante é verificar as permissões associadas ao diretório (**~/.ssh**) e ao arquivo que armazena as chaves autorizadas (**~/.ssh/authorized_keys**) no seu servidor.

ls -la ~/.ssh

Na imagem abaixo, nota-se que apenas o **user owner** (lets) tem permissão de gravar no diretório ~/.ssh e apenas ele pode ler e gravar no arquivo "authorized_keys".

```
lets@serverlets:~$ ls -la ~/.ssh/
total 20
drwx------ 2 lets lets 4096 Feb 12 15:23 .
drwxr-x--- 7 lets lets 4096 Feb 11 22:45 ..
-rw------- 1 lets lets  919 Feb 12 15:57 authorized_keys
-rw------- 1 lets lets 1823 Feb 12 15:23 id_rsa
-rw-r--r-- 1 lets lets  397 Feb 12 15:23 id_rsa.pub
lets@serverlets:~$
```

Caso as duas permissões estejam diferentes disso, por exemplo, suponha que outros usuários (others) podem fazer alterações, você pode aplicar **chmod** para alterar as permissões. Por exemplo, você poderia aplicar os seguintes comandos:

sudo chmod 700 ~/.ssh
sudo chmod 600 ~/.ssh/authorized_keys

Comando SCP

Agora que você já aprendeu como se conectar a um servidor remoto usando o protocolo SSH, que tal transferir arquivos considerando essa forma segura de conexão. O comando **scp (secure copy)** permite transferir arquivos entre sua máquina local e o servidor remoto via SSH.

Supondo que você esteja na sua máquina local e deseja enviar um arquivo chamado **dados.txt** para um diretório no servidor remoto, a sintaxe é:

scp caminho/do/arquivo.txt usuario@is_do_servidor:/caminho/para/o/destino/

Primeiramente, vamos criar um arquivo com um conteúdo dentro, conforme o comando a seguir:

echo "alguma mensagem" >> dados.txt

```
etonio.silva@BRRIOLN043879:~ $ echo "Arquivo com dados" >> dados.txt
etonio.silva@BRRIOLN043879:~ $
```

O caminho absoluto até o arquivo dados.txt na minha máquina local é:

/home/letonio.silva/dados.txt

No servidor remoto, escolha um local onde deseja salvar esse arquivo. O caminho completo até o diretório escolhido é:

/home/lets/aula-linux

```
lets@serverlets:~$ mkdir aula-linux
lets@serverlets:~$ ls
aula-linux  ftp  page
lets@serverlets:~$ cd aula-linux/
lets@serverlets:~/aula-linux$ pwd
/home/lets/aula-linux
lets@serverlets:~/aula-linux$
```

Nesse caso, o comando para a transferência fica:

scp caminho/do/arquivo.txt usuario@is_do_servidor:/caminho/para/o/destino/

scp /home/letonio.silva/dados.txt lets@192.168.0.102:/home/lets/aula-linux

```
letonio.silva@BRRIOLN043879:~ $ scp /home/letonio.silva/dados.txt lets@192.168.0.102:/home/lets/aula-linux
dados.txt                                     100%   18    27.5KB/s   00:00
letonio.silva@BRRIOLN043879:~ $
```

No servidor remoto, podemos confirmar que a transferência foi um sucesso:

ls && cat dados.txt

```
lets@serverlets:~/aula-linux$ ls
dados.txt
lets@serverlets:~/aula-linux$ cat dados.txt
Arquivo com dados
lets@serverlets:~/aula-linux$
```

Para transferir um diretório inteiro use a opção "-r" junto com o comando scp. Na imagem abaixo, usei mkdir para criar um diretório na minha máquina local. Em seguida, aplicou-se o comando echo para criar uma mensagem e através do operador (>>) redirecionar a saída do primeiro comando para dentro do arquivo arq.txt. Por fim, aplicou-se o comando scp com a opção "-r" para enviar o diretório para o servidor remoto.

```
letonio.silva@BRRIOLN043879:~ $ mkdir z-diretorio && echo "Arquivo dentro do diretorio" >> ./z-diretorio/arq.txt
letonio.silva@BRRIOLN043879:~ $ ls z-diretorio/
arq.txt
letonio.silva@BRRIOLN043879:~ $ scp -r /home/letonio.silva/z-diretorio lets@192.168.0.102:/home/lets/aula-linux
arq.txt                                       100%   28    41.3KB/s   00:00
letonio.silva@BRRIOLN043879:~ $
```

Podemos confirmar que a transferência foi um sucesso, conforme ilustrado na imagem a seguir:

```
lets@serverlets:~/aula-linux$ cd z-diretorio/ && ls && cat arq.txt
arq.txt
Arquivo dentro do diretorio
lets@serverlets:~/aula-linux/z-diretorio$
```

Introdução Shell Script

Cada vez que um ator se prepara para uma cena, seja no cinema, televisão ou teatro, ele recebe um script. Esse script (roteiro) é um documento que contém detalhes essenciais sobre como o ator deve se comportar durante a cena e quais são os elementos cruciais para o desenvolvimento da narrativa. Os scripts no contexto computacional operam de forma bastante similar. Eles consistem em uma sequência de instruções que orientam o dispositivo sobre como executar tarefas específicas, seguindo uma programação predefinida.

Shell

Em computação, o termo **shell** refere-se a uma interface entre o usuário e o sistema operacional. Ele atua como um interpretador de comandos, permitindo que os usuários interajam com o sistema operacional enviando comandos. Ele tem funções seletivas e só permite a execução de ações de acordo com o nível de acesso de cada usuário. O shell pode ser acessado por meio de uma **Interface Gráfica de Usuário** (GUI - Graphical User Interface) ou de uma **Interface de Linha de Comando** (CLI - Command Line Interface). Na CLI, para que qualquer ação seja executada é preciso que um comando equivalente seja digitado.

Shell Script

O **Shell Script** é um tipo de linguagem de script utilizada para automatizar a execução de comandos em um ambiente de shell. O shell é a interface de linha de comando (CLI) que permite que os usuários interajam com o sistema operacional. O **Bash** (Bourne Again SHell) é um exemplo de shell e, atualmente, é o que vem instalado por padrão nas distribuições Linux.

Um Shell Script é um arquivo contendo uma sequência de comandos shell que podem ser executados de uma vez. Esses scripts são úteis para automatizar tarefas repetitivas, criar sequências de comandos complexas ou simplificar a execução de várias operações. Os scripts de shell podem conter variáveis, estruturas de controle de fluxo (como condicionais e loops), funções, e podem interagir com programas e utilitários do sistema operacional.

Algumas observações antes de começarmos a aprender sobre shell script:

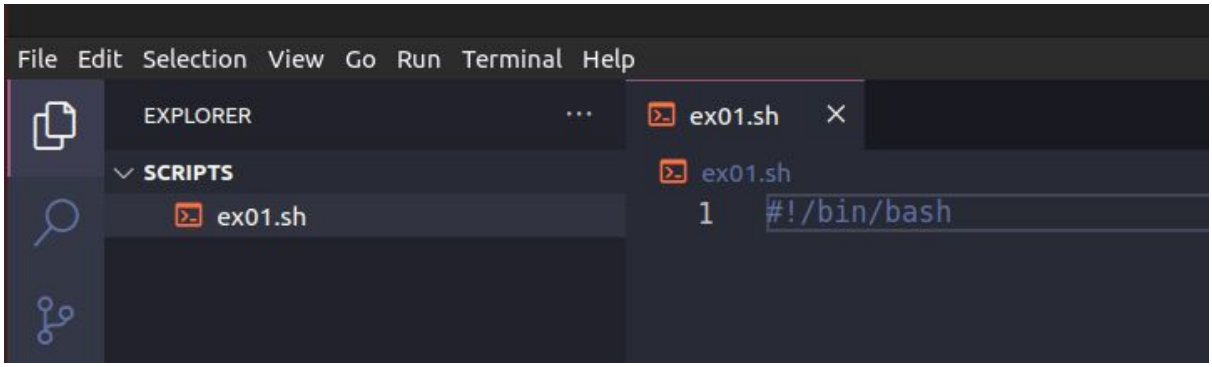
- Você pode usar um editor de texto de sua escolha, como o nano, vim, etc.
- Para facilitar a visualização do texto, optamos pelo Visual Studio Code.
- Nesse ponto da trilha, você é capaz de transferir arquivos tranquilamente de sua máquina local para a máquina virtual com Ubuntu server.
- Caso a sua máquina local (host) permite a execução de shell script, não é necessário usar a sua máquina virtual.

A partir do terminal, podemos abrir o Visual Studio Code através desse comando:

code .

```
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $ code .
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $
```

A extensão de um shell script é **.sh**. Na imagem a seguir, criamos um arquivo chamado **ex01.sh**. Na primeira linha, temos **#!/bin/bash**, que é conhecida como **shebang** ou **hashbang**, e é uma convenção usada em scripts shell para indicar qual interpretador deve ser usado para executar o script. Neste caso, estamos indicando que o interpretador de shell **Bash** deve ser utilizado para executar o script.



A seguir, vamos aprender sobre variáveis.

Variáveis

As variáveis são elementos fundamentais em shell scripts, proporcionando a capacidade de armazenar e manipular dados. Para atribuir um valor a uma variável, utiliza-se o operador de atribuição (=).
Por exemplo,

nome="Maria"

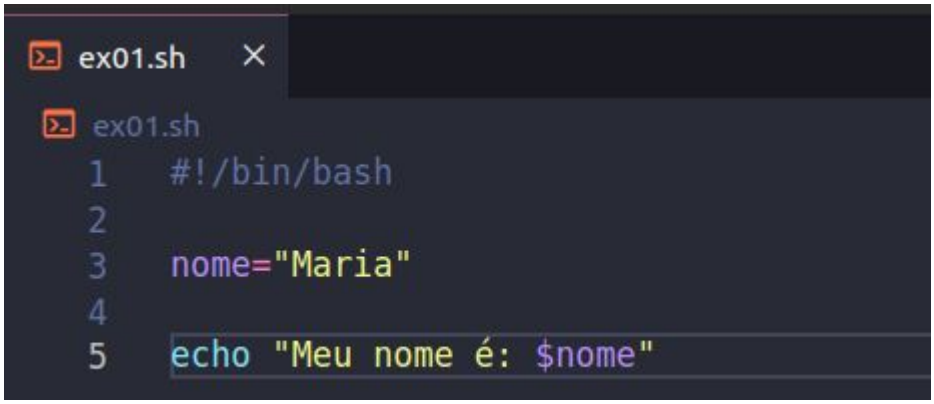
Quase todas as variáveis na linguagem de Shell Script são do tipo string. É possível, claro, fazer uma conversão do tipo de variável a depender do tipo de operação que se deseja executar.
Para utilizar o valor de uma variável, deve-se colocar o símbolo $ antes do nome da variável, conforme mostrado a seguir:

echo "Meu nome é: $nome"

Executar um shell script

Considere o script ex01.sh mostrado a seguir:

```
#!/bin/bash
nome="Maria"
echo "Meu nome é: $nome"
```



No terminal, use o comando **ls -l** para verificar se o arquivo tem permissão para ser executado. Se necessário, adicione a permissão de execução para o **user owner** e **group owner**, por meio desse comando:

chmod ug+x <seu_script.sh>
chmod ug+x ex01.sh

```
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $ ls -l
total 4
-rw-rw-r-- 1 letonio.silva letonio.silva 52 fev 12 17:47 ex01.sh
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $ chmod ug+x ex01.sh
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $ ls -l
total 4
-rwxrwxr-- 1 letonio.silva letonio.silva 52 fev 12 17:47 ex01.sh
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $
```

Para executar o script, escreva o caminho até o script e pressione **Enter**. Na imagem abaixo, foi necessário escrever apenas "./ex01.sh", pois já estava no diretório do script.

```
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $ ./ex01.sh
Meu nome é: Maria
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $
```

Escreva o caminho até o arquivo
pressione Enter

É possível armazenar o resultado de um comando dentro uma variável. Isso torna-se especialmente benéfico em situações nas quais planeja-se utilizar esse resultado em mais de uma parte do script. A seguir, apresenta-se um exemplo onde a saída do comando **echo** é armazenada na variável **saudacao**:

```
#!/bin/bash
nome="Fulano"
saudacao=$( echo "Boa tarde! $nome" )
echo "$saudacao"
```

```
1  #!/bin/bash
2
3  nome="Fulano"
4
5  saudacao=$( echo "Boa tarde! $nome" )
6  echo "$saudacao"
```

Operadores aritméticos

Os principais operadores aritméticos são:

- **+** Adição;
- **-** Subtração;
- ***** Multiplicação;
- **/** Divisão (O resultado é a parte inteira da divisão, por exemplo, 10/3 = 3);
- **%** Módulo (resto da divisão entre dois números, por exemplo, 10%3 = 1);
- ****** Exponenciação;

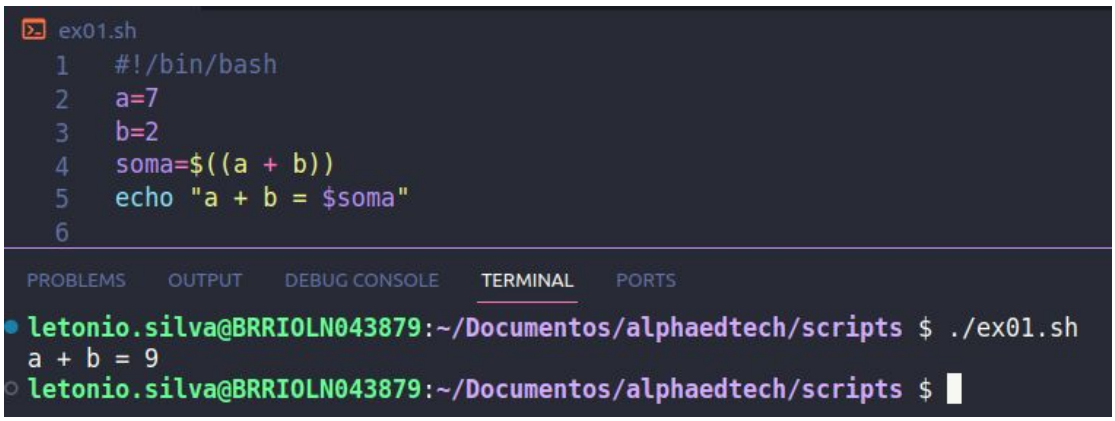
Para realizar uma operação aritmética entre duas variáveis, deve-se utilizar parênteses duplos **$(( ))**. Eles indicam ao shell que a expressão contida deve ser avaliada aritmeticamente. No exemplo abaixo, a variável soma recebe o resultado da operação entre as variáveis a e b:

soma=$((a + b))

Usar **$(())** converte string em números durante os cálculos aritméticos. Por exemplo, os dois scripts a seguir produzem o mesmo resultado:

```
#!/bin/bash
a="7"
b="2"
soma=$((a + b))
echo "a + b = $soma"
```


```
#!/bin/bash
a=7
b=2
soma=$((a + b))
echo "a + b = $soma"
```



O próximo shell script mostra todas as operações sendo aplicadas:

```
#!/bin/bash
a=7
b=2
# realiza operações
calc1=$((a + b * 3))
calc2=$(((a - b) * 4))
calc3=$(((a * 3) / b))
calc4=$((a % (b + 1)))
calc5=$(((a ** 2) - b))
echo "a = $a e b = $b"
echo "a + b * 3 = $calc1"
echo "(a - b) * 4 = $calc2"
echo "(a * 3) / b = $calc3"
echo "a % (b + 1) = $calc4"
echo "a^2 - b = $calc5"
```

ex01.sh

```
#!/bin/bash

# atribue valores às variáveis
a=7
b=2

# realiza operações
calc1=$((a + b * 3))
calc2=$(((a - b) * 4))
calc3=$(((a * 3) / b))
calc4=$((a % (b + 1)))
calc5=$(((a ** 2) - b))

# imprime os resultados na saída do terminal
echo "a = $a e b = $b"
echo "a + b * 3 = $calc1"
echo "(a - b) * 4 = $calc2"
echo "(a * 3) / b = $calc3"
echo "a % (b + 1) = $calc4"
echo "a^2 - b = $calc5"
```

Executar o script produz o seguinte resultado:

```
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $ ./ex01.sh
a = 7 e b = 2
a + b * 3 = 13
(a - b) * 4 = 20
(a * 3) / b = 10
a % (b + 1) = 1
a^2 - b = 47
```

Note que a operação de divisão resulta em um número inteiro. Caso deseje trabalhar com números decimais/ponto flutuante, recomenda-se usar o comando **bc**. No script a seguir, calcula-se a divisão entre dois números e armazena-se dentro da variável **result** considerando duas casas decimais (**scale=2**).

```
#!/bin/bash

numerador=10
denominador=3

# realiza a divisão e formata para duas casas decimais usando o bc
result=$( echo "scale=2; $numerador / $denominador" | bc )

echo "O resultado da divisão é: $result"
```

ex11.sh

```
#!/bin/bash

numerador=10
denominador=3

# realiza a divisão e formata para duas casas decimais usando o bc
result=$( echo "scale=2; $numerador / $denominador" | bc )

echo "O resultado da divisão é: $result"
```

Operadores comparativos

A seguir, apresenta-se a lista de operadores utilizados para fazer a comparação entre dois elementos é:

- **-eq** (equal): Igual à;
- **-ne** (not equal): Diferente de ou "Não igual à";
- **-lt** (less than): Menor que;
- **-gt** (greater than): Maior que;
- **-le** (less or equal): Menor ou igual à;
- **-ge** (greater or equal): Maior ou igual à;

No próximo item, vamos introduzir estruturas de decisão e fazer uso de alguns desses operadores.

Estruturas de decisão

Em shell scripts, as estruturas de decisão são usadas para controlar o fluxo de execução do script com base em condições específicas, isto é, condicionais. As estruturas mais comuns são o **if**, **else**, e **elif** (abreviação de "else if"), cujo uso é o mesmo presente em diversas linguagens de programação. A sintaxe básica é:

```
if [ condição ]; then
   # comandos se condição for verdadeira
else
   # comandos se condição for falsa
fi
```

O bloco de instruções dentro de um **if** só é executado se a condição definida no argumento é verdadeira. Todo **else** deve ser precedido de um **if** ou **elif**. Ele é executado quando o **if** ou **elif** associado a ele tem uma condição que não é atendida, ou seja, é falsa.
No shell script abaixo, se a pessoa é maior de idade uma mensagem é impressa na saída do terminal, caso contrário outra mensagem é mostrada.

```
#!/bin/bash
# atribue o nome  e a idade
nome="Felipe"
idade=20
# condicional
if [ $idade -ge 18 ]; then
   echo "$nome é maior de idade."
else
   echo "$nome é menor de idade."
fi
```

ex02.sh

```
#!/bin/bash

# atribue o nome  e a idade
nome="Felipe"
idade=20

# condicional
if [ $idade -ge 18 ]; then
    echo "$nome é maior de idade."
else
    echo "$nome é menor de idade."
fi
```

A imagem a seguir mostra o script sendo executado:

```
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $ ./ex02.sh
Felipe é maior de idade.
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $
```

No próximo exemplo, vemos a utilização do **elif**:

```
#!/bin/bash
# atribue o nome  e a idade
nome="Maria"
idade=25
# condicional
if [ $idade -lt 18 ]; then
   echo "$nome é menor de idade."
elif [ $idade -gt 70 ]; then
   echo "$nome é uma pessoa idosa."
else
   echo "$nome é uma pessoa adulta."
fi
```

ex02.sh

```
#!/bin/bash

# atribue o nome  e a idade
nome="Felipe"
idade=20

# condicional
if [ $idade -ge 18 ]; then
    echo "$nome é maior de idade."
else
    echo "$nome é menor de idade."
fi
```

Na imagem a seguir, mostra-se o script sendo executado:

```
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $ ./ex02.sh
Maria é uma pessoa adulta.
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $
```

Estruturas de repetição

While

A primeira estrutura de repetição permitida é o **while**. A sintaxe básica é a seguinte:

```
while [ condição ];
do
   # declarações
   # comandos
done
```

Tome cuidado ao utilizar uma estrutura de repetição, pois você pode acabar criando um loop infinito. A seguir, apresenta-se um exemplo de script que imprime números em ordem decrescente:

```
#!/bin/bash
contador=4

while [ $contador -ge 0 ]; do
   echo "contagem regressiva: $contador"
   contador=$((contador - 1))
done
```

ex03.sh

```
1  #!/bin/bash
2
3  contador=4
4
5  while [ $contador -ge 0 ]; do
6      echo "contagem regressiva: $contador"
7      contador=$((contador - 1))
8  done
```

Na próxima imagem, vemos o resultado da execução desse script:

```
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $ ./ex03.sh
contagem regressiva: 4
contagem regressiva: 3
contagem regressiva: 2
contagem regressiva: 1
contagem regressiva: 0
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $
```

For

A segunda estrutura de repetição é o **for**. A estrutura **for** é usada para iterar sobre uma sequência de valores, como elementos em uma lista ou sequência numérica. A sintaxe básica é a seguinte:

```
for elemento in lista; do
    # comandos
done
```

É possível atribuir a uma única variável uma lista de valores e, através do **for**, iterar sobre essa lista. A seguir, apresenta-se um exemplo onde declaramos uma variável que recebe um uma lista de frutas.

```
#!/bin/bash
frutas=("pera" "uva" "maçã" "morango")

for fruta in "${frutas[@]}"; do
    echo "Eu gosto de $fruta."
done

echo ""
echo "outras frutas:"
for outra in "maracujá" "limão" "acabate"; do
    echo "$outra também é bom."
done
```

ex04.sh

```
1   #!/bin/bash
2
3   frutas=("pera" "uva" "maçã" "morango")
4
5   for fruta in "${frutas[@]}"; do
6       echo "Eu gosto de $fruta."
7   done
8
9   echo ""
10  echo "outras frutas:"
11  for outra in "maracujá" "limão" "acabate"; do
12      echo "$outra também é bom."
13  done
```

Na próxima imagem, vemos o shell script sendo executado.

```
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $ ./ex04.sh
Eu gosto de pera.
Eu gosto de uva.
Eu gosto de maçã.
Eu gosto de morango.

outras frutas:
maracujá também é bom.
limão também é bom.
acabate também é bom.
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $
```

Ainda utilizando o **for**, é possível iterar sobre uma sequência numérica **{num1..num2}**. Se o número 1 é menor que o número 2, a variável é incrementada de 1 em 1 até atingir o número 2.

É possível mudar o tamanho do passo, através da adição de um terceiro número. Por exemplo, **elemento in {4..11..2}** vai atribuir a "**elemento**" os valores 4, 6, 8 e 10.

Caso o primeiro número seja maior que o segundo, "elemento" recebe os valores na ordem decrescente. Por exemplo, **elemento in {10..1}** vai atribuir a "**elemento**" os valores 10, 9, ..., 1.

A seguir, apresenta-se um script que faz uso da estrutura **for** para imprimir números na ordem crescente e decrescente.

```
#!/bin/bash
for i in {1..3}; do
   echo "Número $i"
done

echo ""
echo "Decrescente de 2 em 2"
for i in {10..1..2}; do
   echo "Número $i"
done
```

ex05.sh

```
1   #!/bin/bash
2
3   for i in {1..3}; do
4       echo "Número $i"
5   done
6
7   echo ""
8   echo "Decrescente de 2 em 2"
9   for i in {10..1..2}; do
10      echo "Número $i"
11  done
```

O resultado da execução desse shell script é mostrado a seguir:

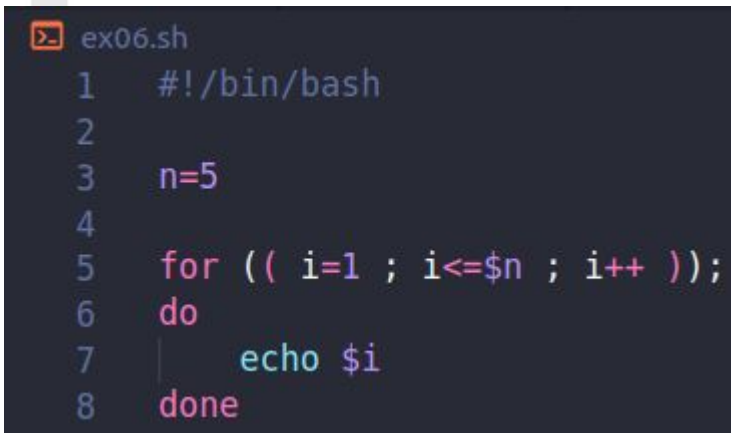
```
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $ ./ex05.sh
Número 1
Número 2
Número 3

Decrescente de 2 em 2
Número 10
Número 8
Número 6
Número 4
Número 2
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $
```

A estrutura **for** também permite uma sintaxe muito parecida com a implementada em linguagem C. A seguir, apresenta-se um exemplo prático dessa sintaxe:

```
#!/bin/bash

n=5
for (( i=1 ; i<=$n ; i++ ));
do
   echo $i
done
```



Entrada de dados

De acordo com o seu objetivo ao criar um shell script, pode ser necessário interagir com o usuário. Por exemplo, pedir que ele forneça algum dado de entrada para processamento. Neste caso, é necessário que se leia o que o usuário digitou. Isso é feito usando o comando **read**, de acordo com uma das sintaxes abaixo:

```
echo "instruções para o usuário:"
read <variavel_para_armazenar>
```

ou

```
read -p  "instruções para o usuário:"  <variavel_para_armazenar>
```

No script abaixo, mostra-se um exemplo que verifica se o número informado pelo usuário é par ou ímpar.

```
#!/bin/bash

echo "Insira um número inteiro positivo:"
read num

if [ $(($num % 2)) == 0 ]; then
        echo "$num é par"
else
        echo "$num é ímpar"
fi
```

```
ex07.sh
1   #!/bin/bash
2
3   echo "Insira um número inteiro positivo:"
4   read num
5
6   if [ $(($num % 2)) == 0 ]; then
7       echo "$num é par"
8   else
9       echo "$num é ímpar"
10  fi
```

A seguir, apresenta-se o resultado do script sendo executado, utilizando os valores 4 e 7, respectivamente.

```
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $ ./ex07.sh
Insira um número inteiro positivo:
4
4 é par
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $ ./ex07.sh
Insira um número inteiro positivo:
7
7 é ímpar
```

Argumentos

É possível passar argumentos no momento de execução do script. Por exemplo,

./calcula_soma.sh 4 7

executa um script passando dois argumentos. Dentro do script, os valores 4 e 7 ficam armazenados em **$1** e **$2**, respectivamente. Um resumo dos elementos mais importantes dos argumentos:
$0 – contém o nome do script que foi executado;
$1 , **$2**, …, **$n** – contêm os argumentos na ordem em que foram passados (1º argumento em $1, 2º argumento em $2, etc.);
$# - contém o número de argumentos que foi passado (obs:não considera o nome do script em $0);
$* - retorna todos os argumentos de uma vez só.
O exemplo a seguir exibe a quantidade de argumentos passados e lista todos eles:

```
#!/bin/bash

if [ $# -lt 1 ]; then
   echo "Precisa fornecer pelo menos 1 argumento!"
   exit 1 # use exit com valores !== 0 para indicar que houve algum erro
fi

echo "Número de argumentos passados: $#"
i=0
for argumento in $*; do
        i=$(($i+1))
        echo "Argumento $i passado: $argumento"
done
```

ex08.sh

```bash
#!/bin/bash

if [ $# -lt 1 ]; then
   echo "Precisa fornecer pelo menos 1 argumento!"
   exit 1 # use exit com valores !== 0 para indicar que houve algum erro
fi

echo "Número de argumentos passados: $#"

i=0
for argumento in $*; do
   i=$(($i+1))
   echo "Argumento $i passado: $argumento"
done
```

No exemplo a seguir, executa-se o script anterior usando três parâmetros: Brasil, China e EUA.

```
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $ ./ex08.sh Brasil China EUA
Número de argumentos passados: 3
Argumento 1 passado: Brasil
Argumento 2 passado: China
Argumento 3 passado: EUA
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $
```

O próximo exemplo recebe um número inteiro positivo como argumento e informa se ele é par ou ímpar.

```bash
#!/bin/bash

if [ $# -ne 1 ]; then
   echo "Precisa fornecer exatamente 1 argumento!"
   exit 1 # use exit com valores !== 0 para indicar que houve algum erro
fi

num=$1
# operador =~ verifica se dá match com o regex
if [[ ! "$num" =~ ^[0-9]+$ ]]; then
    echo "O argumento fornecido não é um número."
    exit 2 # usa código de erro 2 para indicar que o argumento não é um número
fi

if [ $(($num % 2)) == 0 ]; then
        echo "$num é par"
else
        echo "$num é ímpar"
fi
```

ex09.sh

```bash
#!/bin/bash

if [ $# -ne 1 ]; then
   echo "Precisa fornecer exatamente 1 argumento!"
   exit 1 # use exit com valores !== 0 para indicar que houve algum erro
fi

num=$1

# operador =~ verifica se dá match com o regex
if [[ ! "$num" =~ ^[0-9]+$ ]]; then
    echo "O argumento fornecido não é um número."
    exit 2 # usa código de erro 2 para indicar que o argumento não é um número
fi

if [ $(($num % 2)) == 0 ]; then
    echo "$num é par"
else
    echo "$num é ímpar"
fi
```

Na imagem a seguir, mostra-se a execução do script para algumas situações. A primeira, não informamos nenhum argumento. Nesse caso, o script emite uma mensagem de erro, explicando a quantidade exata de argumentos que devem ser passados. No segundo caso, não informamos um número, o que também ocasiona um aviso ao usuário. Por fim, na terceira execução, o script é executado com um argumento correto.

```
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $ ./ex09.sh
Precisa fornecer exatamente 1 argumento!
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $ ./ex09.sh oito
O argumento fornecido não é um número.
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $ ./ex09.sh 8
8 é par
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $
```

No script anterior, nota-se a presença do **regex** (regular expression) **^[0-9]+$**. Explicando-o:

- **^**: Indica o início da expressão.
- **[0-9]**: Representa uma classe de caracteres que corresponde a qualquer dígito de 0 a 9.
- **+**: Indica que o caractere ou grupo anterior (a classe de dígitos) deve ocorrer uma ou mais vezes.
- **$**: Indica o final da expressão.

Portanto, caso o argumento passado não seja um número, um erro é emitido e o script encerrado.

Funções

O emprego de funções é essencial para dividir, organizar e estruturar a lógica de qualquer algoritmo. Esse fato é válido para shell script ou em qualquer outra linguagem de programação.

Para declarar uma função utiliza-se a sintaxe a seguir:

```
nome_da_função()
{
    # comandos
}
```

Para invocar a função (parâmetros são opcionais):

```
nome_da_funcao  <param_1> <param_n>
```

A seguir, utilizamos funções para separar as responsabilidades do script que verifica se o número é par ou ímpar. Temos uma função que é responsável por validar a entrada de dados. Caso nenhum erro seja emitido, o script executará a função que verifica se o número é par ou ímpar.

```
#!/bin/bash
valida_entrada() {
  if [ $# -ne 1 ]; then
        echo "Precisa fornecer exatamente 1 argumento!"
        exit 1
  fi

  local num=$1

  if [[ ! "$num" =~ ^[0-9]+$ ]]; then
        echo "O argumento fornecido não é um número."
        exit 2
  fi
}
```

```
verifica_paridade() {
    local num=$1

    if [ $(($num % 2)) == 0 ]; then
            echo "$num é par."
    else
            echo "$num é ímpar."
    fi
}

# uso das funções
valida_entrada "$1"
verifica_paridade "$1"
```

ex10.sh

```bash
#!/bin/bash

valida_entrada() {
    if [ $# -ne 1 ]; then
        echo "Precisa fornecer exatamente 1 argumento!"
        exit 1
    fi

    local num=$1

    if [[ ! "$num" =~ ^[0-9]+$ ]]; then
        echo "O argumento fornecido não é um número."
        exit 2
    fi
}

verifica_paridade() {
    local num=$1

    if [ $(($num % 2)) == 0 ]; then
        echo "$num é par."
    else
        echo "$num é ímpar."
    fi
}

# chama as funções passando um parâmetro
valida_entrada "$1"
verifica_paridade "$1"
```

É válido ressaltar que, embora seja possível utilizar um argumento passado pelo usuário (**$1**) como parâmetro da função, isso não é obrigatório. O exemplo a seguir ilustra isso, onde uma função não recebe parâmetros, enquanto a outra recebe dois. Na função **apresenta_pessoa** "João" é o primeiro parâmetro e está disponível dentro da função através do **$1**; enquanto 25 é o segundo parâmetro e está disponível em **$2**. Não é obrigatório, mas podemos declarar variáveis internas com escopo local através da palavra reservada "local". Assim fica mais claro entender o que são os parâmetros recebidos.

```
#!/bin/bash

faz_saudacao() {
  echo "Bom dia!"
}

apresenta_pessoa() {
  local nome=$1
  local idade=$2
  echo "Olá, meu nome é $nome e eu tenho $idade anos."
}

# chama funcao sem parâmetros
faz_saudacao

# chama funcao que tem dois parâmetros
apresenta_pessoa "João" 25
```

ex13.sh

```
1   #!/bin/bash
2
3   faz_saudacao() {
4     echo "Bom dia!"
5   }
6
7   apresenta_pessoa() {
8     local nome=$1
9     local idade=$2
10    echo "Olá, meu nome é $nome e eu tenho $idade anos."
11  }
12
13  # chama funcao sem parâmetros
14  faz_saudacao
15
16  # chama funcao que tem dois parâmetros
17  apresenta_pessoa "João" 25
```

Por fim, o resultado da execução do script é mostrado a seguir:

```
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $ ./ex13.sh
Bom dia!
Olá, meu nome é João e eu tenho 25 anos.
letonio.silva@BRRIOLN043879:~/Documentos/alphaedtech/scripts $
```