05

A sintaxe da linha de comando

Linux Essentials 3: Bem-vindo!

Esse curso prepara para a certificação do *Linux Essentitials* do LPI. Nesse curso falaremos sobre o objetivo de encontrar o caminho no sistema *Linux*, em geral. Principalmente, na parte de linha de comanda básica.

Falaremos sobre o *shell*, sobre a sintaxe de uma linha de comando e as diversas maneiras de evocar os distintos comandos, sobre configurar o ambiente onde esse comando vai ser executados, os programas, sobre as variáveis, *couching* e etc.

Sempre conversando sobre isso para aplicar essas questões no dia a dia.

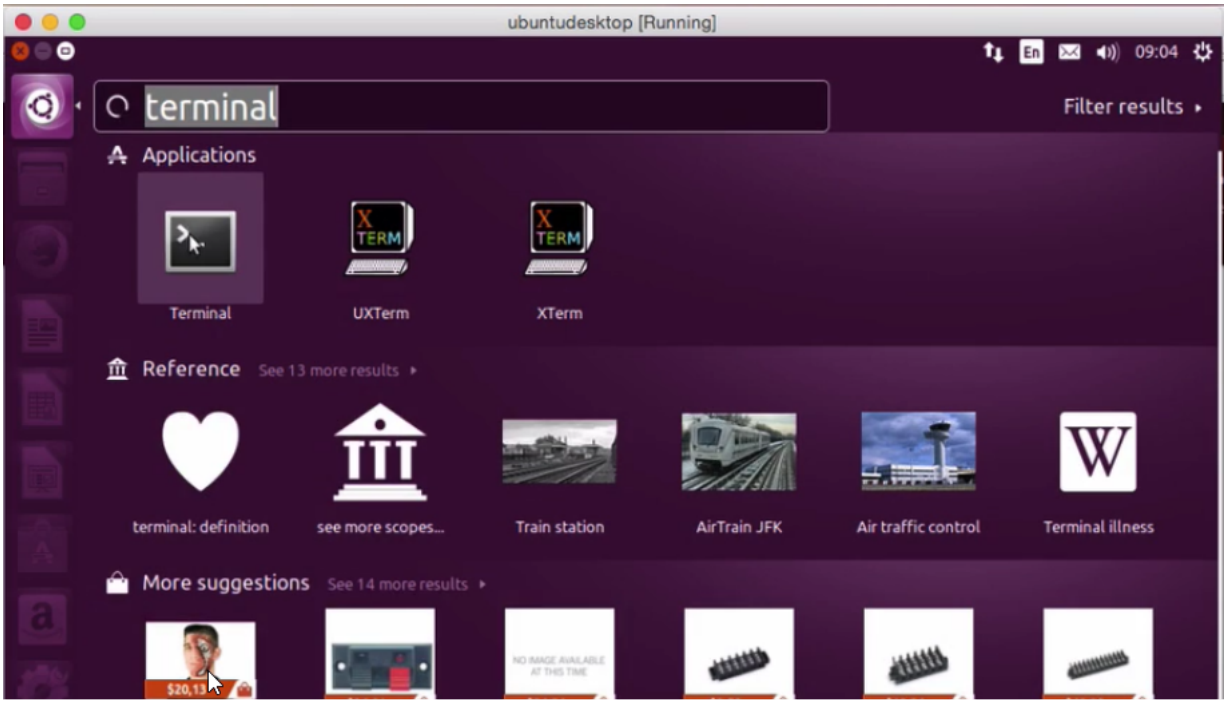
Então, vamos lá?!

Basic shell, bash e echo

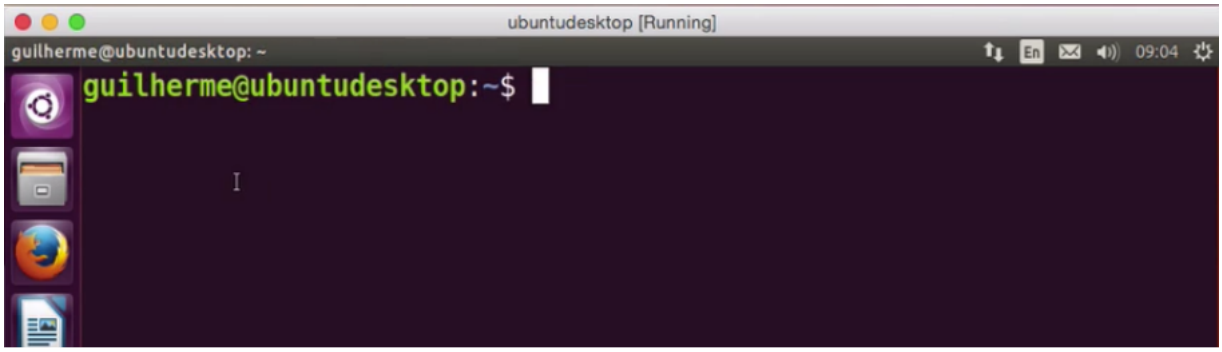
Tudo o que veremos a seguir é para a certificação e nos debruçaremos sobre todos esses aspectos aos poucos.

Primeiro, vamos começar a falar do *basic shell*, um `shell` que é básico.

Se formos na nossa máquina *Ubuntu*, podemos selecionar a opção para abrir um terminal:



Esse terminal está rodando um *console* que está conectado a uma entrada, o teclado, e a uma saída, que é essa janela que vemos na seguinte imagem:



Ele está rodando um *shell* dentro dele. O *shell* está encapsulando um comportamento de: "me diga qual o comportamento que você gostaria de executar".

O comando que nós vamos executar é "listar os arquivos", para isso, digitaremos `ls` e damos um "Enter". Quando digitarmos o comando e dermos um "Enter" o *shell* executará um o comando e devolverá o resultado.

```
ls
Desktop Downloads   Music   Public  Videos
Documents examples.desktop Pictures Templates
```

Existem dezenas de comandos que podem acompanhar o *shell*, mas isso depende também das características de cada *shell*. Ocorre um padrão para todos os *shells* e, inclusive, existe toda uma história de como eles foram desenvolvidos.

Um *shell* extremamente utilizado, atualmente, é o *Bash*, que é cobrado, inclusive, na prova. Podemos obter mais informações a repeito da história do *Bash* no link, (http://%20https://en.wikipedia.org/wiki/Bash_%28Unix_shell%29.)https://en.wikipedia.org/wiki/Bash_(Unix_shell (https://en.wikipedia.org/wiki/Bash_%28Unix_shell))

Bash (Unix shell)

From Wikipedia, the free encyclopedia

"Bash (software)" redirects here. For other software, see Bash (disambiguation).

Bash is a Unix shell and command language written by Brian Fox for the GNU Project as a free software replacement for the Bourne shell.[6][7] Released in 1989,[8] it has been distributed widely as the shell for the GNU operating system and as a default shell on Linux and OS X. It was announced during the 2016 Build Conference that Windows 10 has added a Linux subsystem which fully supports Bash and other Ubuntu binaries running natively in Windows.[9] In the past, and currently, it has also ported to Microsoft Windows and distributed with Cygwin and MinGW, to DOS by the DJGPP project, to Novell NetWare and to Android via various terminal emulation applications. In the late 1990s, Bash was a minor player among multiple commonly used shells; at present Bash has overwhelming favor.

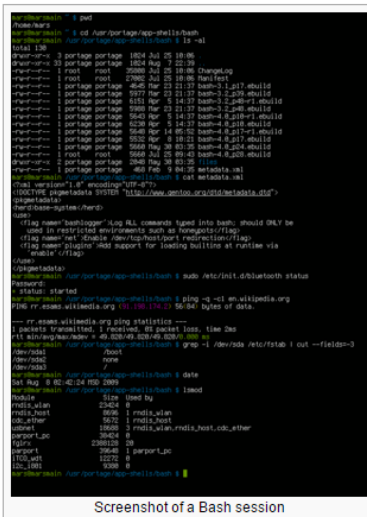
Bash is a command processor that typically runs in a text window, where the user types commands that cause actions. Bash can also read commands from a file, called a script. Like all Unix shells, it supports filename globbing (wildcard matching), piping, here documents, command substitution, variables and control structures for condition-testing and iteration. The keywords, syntax and other basic features of the language were all copied from sh. Other features, e.g., history, were copied from csh and ksh. Bash is a POSIX shell, but with a number of extensions.

The name itself is an acronym, a pun, and a description. As an acronym, it stands for *Bourne-again shell*, referring to its objective as a free replacement for the Bourne shell.[10] As a pun, it expressed that objective in a phrase that sounds similar to *born again*, a term for spiritual rebirth.[11][12] The name is also descriptive of what it did, *bashing together* the features of sh, csh, and ksh.[13]

A security hole in Bash dating from version 1.03 (August 1989),[14] dubbed Shellshock, was discovered in early September 2014.[15][16]

Contents [hide]

Bash



Screenshot of a Bash session

O *Bash* é um unix shell*, podemos visualizar nessa página, a história dos* shells*. Teremos a descrição da origem deles, com os* shells *antigos, como o primeiro thompson* shell*, que era chamado de* sh*. Hoje, o original do* sh *é difícil de ser utilizado e encontrado no dia a dia. Depois dele foram surgindo outros* shells*, como o* "bourne shell" *que por ter sido reescrito por Stephen Bourne recebeu esse nome. Esse "bourne shell" suportava diversas características. A partir desse momento surgiram diversos "shells", como o "C shell", que tentava suportar comandos baseados na linguagem de programação C, por exemplo, se o desejo era o de fazer um laço, este se assemelharia a linguagem C. O* C Shell* buscava, portanto, se assemelhar, mas não ser igual.

"Bash" é uma abreviação das iniciais de *"bourne again shell"*, isto é, b-a-sh. O "Bash" permite rodar diversos comandos, ele permite que você diga que gostaria de executar vários programas. O mais famoso dos *shells* que foram surgindo ao longo do tempo e o que nos interessa mais, é, justamente, o *"bourne shell again"*, o *bash*. Ele foi reescrito para a comunidade do GNU Project. Ele possui um conjunto de funcionalidades que eram disponíveis no *bourne shell* original. Ele é extremamente comum e aparece em uma quantidade enorme nas distribuições do *Linux* e também do *Mac OS X*.

Não precisamos decorar a história, mas é interessante saber que existiu, primeiro, um *shell* original e que depois foram surgindo o bourne shell e outras diversas variações, como o *C shell* e também o *Bourne Again Shell*, o *Bash*, que é o que utilizaremos aqui, pois, em uma maior parte das distribuições é com ele que iremos nos deparar.

Quando estamos no terminal e mandamos executar um comando, por exemplo, o `ls` novamente, o *shell* pega esse comando e manda executá-lo e pegando o resultado pergunta para nós: "Qual o próximo comando que você quer que eu execute?".

```
ls
Desktop Downloads   Music   Public  Videos
Documents examples.desktop Pictures Templates
ls
Desktop Downloads   Music   Public  Videos
Documents examples.desktop Pictures Templates
```

Este é o papel do *shell*: ele suporta diversos comandos e suporta também programação. Podemos programar para ele, criar laços, variáveis, condicionais e muito mais. O básico do *shell* é, portanto, poder executar comandos como, por exemplo, o `ls` , para listar arquivos. Já mencionamos que o *Bash* é uma implementação de um *shell* extremamente famoso e muito utilizado no mercado.

Vamos observar mais um comando, o *echo*. O *echo* devolve coisas. É como se falássemos algo para ele, uma palavra, por exemplo, "Guilherme", e ele irá nos responder devolvendo essa mesma palavra. Vamos testar ele para descobrir o que faz esse comando:

```
echo Guilherme
Guilherme
```

Vamos testar agora com uma frase:

```
echo Guilherme
Guilherme
echo Guilherme, bom dia.
Guilherme, bom dia.
```

O echo devolve, exatamente, o que falamos para ele. E como sei que o `echo` e `ls` são comandos do *shell* e não programas que são executáveis? Podemos perguntar isso utilizando o `type`. Perguntaremos sobre o `ls`:

```
type ls
ls is aliased to `ls --color=auto`
```

Aqui, vemos que o `ls` na verdade é uma *aliased*, isto é, ele possui um outro nome, uma espécie de apelido que estamos dando para o comando `ls -- color=auto`.

Bom, já falamos sobre o `echo` e sobre o `ls`. O comando `echo` foi implementado pelo *shell*. Mas, como podemos provar isso? Simples, vamos perguntar para o `shell` qual é o tipo de comando.

```
type echo
echo is a shell builtin
```

Ele nos respondeu que `echo` é um comando construído, *builtin*, internamente no *shell*, o que significa que ele é interno ao *shell*. Podemos pegar, ainda, outros comandos típicos do *shell* para descobrir outras coisas. Podemos perguntar sobre "Qual o diretório atual?". Para descobrir isso usamos o `pwd`.

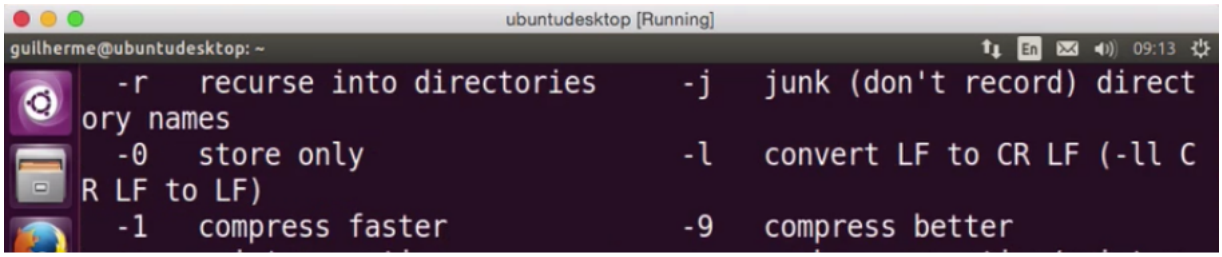
```
pwd
/home/guilherme
```

Ele respondeu que o diretório atual é `/home/guilherme`. Será que esse `pwd` é um comando implementado pelo `shell` ou ele é um programa que está instalado no *Linux*? Para descobrir isso, podemos usar o `type pwd`:

```
pwd
/home/guilherme
type pwd
pwd is a shell builtin
```

E aqui temos a reposta que ele também é interno ao *shell*. Tanto o `echo`, quanto o `pwd` são `builtings` do *shell*, isto é, comandos.

Vamos tentar com outra coisa, o `zip`. Se digitarmos `zip` e dermos "Enter" teremos a seguinte tela:



Como podemos observar, ele mostrará diversas mensagens na tela. O `zip` é um programa que está em *usr* em `zip`.

```
type zip
zip is hashed(/usr/bin/zip)
```

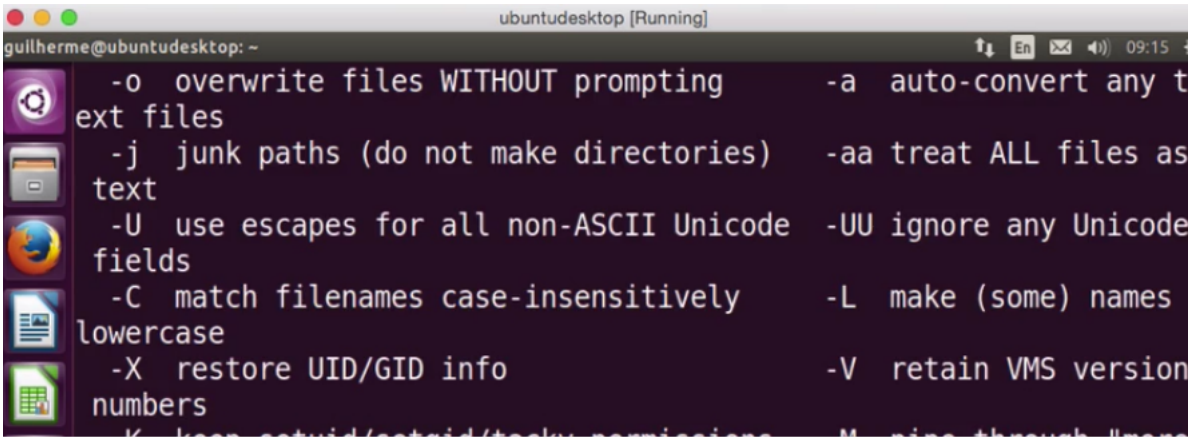
Toda vez que chamamos um `zip` ele também trará consido o `/bin` e o `/usr`. E se digitarmos o `type unzip`?

```
type zip
zip is hashed(/usr/bin/zip)
type unzip
unzip is /usr/bin/zip
```

Teremos o mesmo! Mas, qual a diferença entre os dois? O `zip` e o `unzip` são programas externos. Mas, por que o primeiro é *hashed* e o segundo não? Toda vez que executamos um comando no *shell* ele busca onde está o comando e quando ele acha ele mostra para nós. Toda busca que é realizada pode demorar um pouco. Porém, podemos ter uma resposta rápida em formato de *cache*. Depois que executamos o comando uma primeira vez, o *shell* esperto como é, descobre onde o comando está armazenado e guarda isso na memória, através de um *cache*. E isso é o *hashed*.

Bom, nós já tínhamos executado o `zip`. Lembra que digitamos o `zip` e demos um "enter" e várias mensagens surgiram em nossa tela? Ele jogou em um *cash* a informação de onde estava esse comando e nos informou que o `zip` foi encontrado pela última vez no `/usr/bin/zip`. Por isso, o `zip` é um `hashed`.

E o `unzip`? Bom, até esse momento ele não foi executado. Se executarmos o `unzip`, teremos também uma série de mensagens:



Podemos limpar nossa tela utilizando o `clear` e podemos perguntar qual o `type unzip`. Teremos a seguinte resposta:

```
type unzip
unzip is hashed (/usr/bin/unzip)
```

Vamos analisar o que o `type` nos informou, ele nos mostrou que o `pwd` é um `shell builtin` e mostrou que é um programa externo, pois ele vêm do `usr/bin/unzip/`. e observarmos apenas isso, pode significar que ele "cacheou", onde está esse programa, mas pode ser que ele ainda não tenha "cacheado". Se ele "cacheou", ele fala que o `unzip is hashed`.

Vamos ver também o *type* do *clear* e verificar o que ele é:

```
type clear
clear is hashed (/usr/bin/clear)
```

É clear é também um *hashed*, pois já o executamos várias vezes. O `clear` não é um comando do *Bash* é um programa que estamos executando.

```
type clear
clear is hashed (/usr/bin/clear)
```

Vamos ver um último exemplo de um comando, o `type` do `ls`. Já executamos o `ls`, mas ainda não testamos o `type` dele. Será que ele é um *hashed* externo ou será que ele é um *builtin*, ou, quem sabe, ele não seja nenhum dos dois?

```
type ls
ls is alliased to `ls --color=auto`
```

Repare que, no *Ubuntu Desktop*, vamos ter um resultado. Se você rodar em outros aplicativos o resultado pode ser diferente. Pode ser que esteja configurado de maneira distinta. Mas, aqui, o `ls` está configurado para que seja um apelido, um *aliased*, para `--color=auto`. Quando executamos o `ls`, na verdade, ele está executando o `ls --color=auto`. O `type` nos diz, então, que o `ls` é um apelido. Assim, se executarmos o `ls --color=auto` teremos o mesmo resultado se somente executássemos o `ls`:

```
ls --color=auto
Desktop Downloads   Music   Public  Videos
Documents examples.desktop Pictures Templates
ls
Desktop Downloads   Music   Public  Videos
Documents examples.desktop Pictures Templates
```

Podemos criar apelidos, e por padrão o *Ubuntu Desktop* já vêm configurado com alguns apelidos. Por exemplo, por que mostrar o `ls` em preto e branco? É o usuário final que está usando o *Linux* e para ele é feio usar em preto e branco, por isso, já mostra colorido, com o padrão colorido, o `--color=auto`.

Vimos diversas variações do `type` e os diversos resultados que ele oferece. Mas, será que esses são todos os resultados possíveis do type?

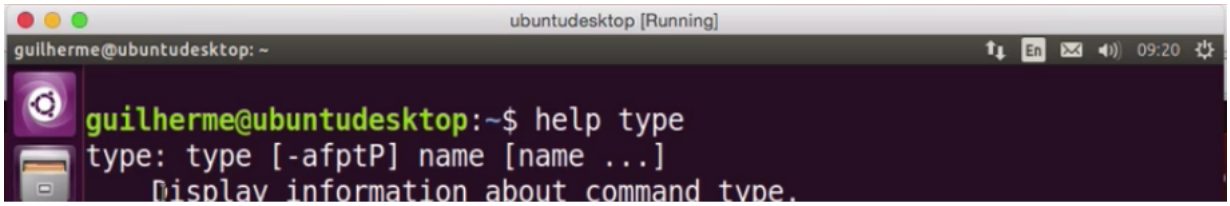
Para nós, é importante saber quais são os resultados do *builtin*, os que vêm de dentro do nosso *shell* e saber, também, quais são os externos. Isto é, se ele é *builtin* é *shell builtin* e se ele não é *builtin* ele é alguma outra coisa que o *type* irá nos informar. Vamos executar outro comando, o `date`, como mais um exemplo:

```
date
Ter Mar 8 09:18:54 BRT 2016
```

O `date` nos fornece a data e o horário, inclusive com o *time zone*, o BRT, *Brazilian Time*. E podemos executar também o `type` do `date`.

```
date
Ter Mar 8 09:18:54 BRT 2016
type date
date is hashed (/bin/date)
```

O *type* do date já está "hasheado" para `(/bin/date)`. Que é um outro comando. E como fazemos para descobrir um pouco mais a respeito do comando *type*? Vamos dar um `clear` na tela. Para saber isso podemos pedir ajuda, digitando `help type`.



Repare na ajuda que ele vai dar, falando o que o programa faz. Ele mostra informações sobre o comando e como ele pode ser usado, por exemplo, type `-a` , `-f` , `-p` , `-t` . Ele fala as opções, por exemplo, vamos observar o `-t` , ele mostra uma palavra apenas, que pode ser `alias` , `keyword` , uma `function` , uma `builtin` , um `file` ou alguma outra coisa específica.

O que podemos fazer é executar o type com `-t` . Para isso vamos limpar a tela, damos um `clear` . E digitamos `type pwd` e `type -t pwd` . Teremos:

```
type pwd
pwd is a shell builtin
type -t pwd
builtin
```

No `type -t pwd` teremos apenas o `builtin` .Vamos executar outro `type` , o `type if` :

```
type pwd
pwd is a shell builtin
type -t pwd
builtin
type if
if is a shell keyword
```

O `type if` é uma palavra chave do *shell*. Podemos executar, ainda, um `type -t if` e ele mostrará apenas a palavra `keyword` .

```
type pwd
pwd is a shell builtin
type -t pwd
builtin
type if
if is a shell keyword
type -t if
keyword
```

Perceba que depende muito do que você está buscando, isto é, para quê você está utilizando o type?

Repare que o `help`, um espaço e o nome do comando, como em `help type` vai fazer com que tenhamos acesso as informações do `shell`. Veremos, mais adiante, diversas maneiras de pegar ajuda sobre comandos, sejam do *shell*, sejam de fora. Agora, o *type* é quem vai nos fornecer essas informações, uma vez que ele é quem cobrado.

Cuidado! A prova pode cobrar simplesmente qual a função do `type`. Retomando, o `type` serve para entendermos se aquilo que o acompanha é um comando, um programa, uma palavra chave do *shell*, uma função e etc. Isto é, ele nos ajuda a compreender o que é aquilo que a gente está perguntando?

E quais são os parâmetros, as opções que a prova pode cobrar da gente no *type*? Bom, qualquer coisa que está descrita no `help type` pode cair na prova. Na verdade, existe uma infinidade de comandos com suas respectivas infinidades de opções que a prova pode cobrar. Não coseguimos saber exatamente o que será cobrado, mas, é mais provável que seja o que é utilizado no dia a dia.

Como já foi citado, no caso do *type* o mais comum é perguntar direto:

```
type pwd
pwd is a shell builtin
```

Ou perguntar qual o tipo:

```
type -t pwd
builtin
```

Essas são as questões mais comuns a respeito do *type*, mas é interessante que se leia também sobre o restante das informações, é o que deixaremos recomendado aqui! Leiam a documentação de todos os comandos que a gente ver a partir de agora.

Não passaremos, aqui, por todos esses conteúdos detalhadamente. Veremos todos os comandos que são citados na prova, veremos as opções principais desses comandos e é interessante que se leia toda a documentação a respeito desse comando para sabermos o que mais ele é capaz e fazer. As vezes a prova pode cobrar uma variação que não é tão comum. Mas, em geral, o que será cobrado são as opções mais comuns, não aquelas que são raras. Antigamente, ela era famosa por cobrar as opções raras, porém, não funciona mais assim. Claro, existe a possibilidade, então, lendo a documentação você pode garantir um acerto em uma questão dessas.

Finalizando, vimos o comando *type* para saber o tipo de comando que estamos executando, vimos também o *dash* e o *echo*.

Command Line Syntax

Um dos tópicos que é cobrado na prova é sobre a sintaxe da linha de comando. Isto é, qual a sintaxe quando vamos executar um comando na linha de comando.

Vamos na linha de comando, no terminal:

Se queremos executar um comando chamamos o nome dele. Só o nome é o suficiente para que ele faça o que queremos, então poderíamos digitar apenas `ls` para listar os documentos, `pwd` para saber o diretório ou só `date` para que nos informasse a data.

Teríamos:

```
ls
Desktop     Downloads           Music       Public  Videos
Documents   examples.desktop    Pictures    Templates
pwd
/home/guilherme
date
Ter Mar 8 09:18:54 BRT 2016
```

Alguns outros comandos, entretanto, podem ficar estranhos quando chamamos apenas o nome deles, por exemplo, o caso do `echo`. Ele nos devolverá "nada", se apenas digitarmos `echo`. Não é que tenha algo de errado nesse comando, mas é estranho não passarmos nenhum argumento para ele. Ao executar esse comando é necessário que passemos, também, um argumento para ele. Por exemplo `echo Guilherme`. Aí sim, ele irá ecoar o argumento e teremos:

```
echo Guilherme
Guilherme
```

O comando `echo` é implementado ecoando os argumentos que o acompanham. Poderíamos passar, ainda, três argumentos diferentes, por exemplo, `echo Guilherme, bom dia` e ele iria ecoar da mesma maneira, repetindo os três argumentos.

```
echo Guilherme, bom dia.
Guilherme, bom dia.
```

O `echo` funciona de uma maneira diferente do `ls`. Para executar o `ls` não precisamos acrescentar nenhum comando.

Vimos também que algus outros comandos recebem coisas a mais, como o `type`. Se perguntarmos ao `type`, qual é o tipo do comando `echo` ele irá nos responder que é `a shell builtin`.

```
type echo
echo is a shell builtin
```

O type pode receber além do comando e argumento, também uma opção, como o `type -t echo`.

Em geral na sintaxe da linha de comando temos três partes distribuídas da seguinte maneira: (1) o comando, (2) as opções e (3) o argumento.

As opções em geral são passadas com o sinal negativo "-" na frente ou um duplo negativo "--". Ambas as formas são válidas, cada programa vai aceitar um padrão diferente, cada comando vai ter uma regra, pois, não existe uma regra fixa. Mas, em geral, primeiro temos o comando, em seguida as opções e no fim, os argumentos. Na maior parte das vezes é isso que ocorre.

```
type -t echo
builtin
```

Aqui, por exemplo, se inserirmos o `type echo -t` a resposta que teremos seria `not found`. Pois esse é um programa que não aceita essa ordem, ele interpreta o `-t` como um argumento e não como uma opção`.

```
type echo -t
echo is a shell builtin
bash: type: -t: not found
```

Repare, no geral, o padrão é, primeiro, comando, depois, opções e, finalmente, argumentos. O que não significa, por exemplo, que outros programas não aceitem o inverso. Vão ocorrer casos em que o programa aceitará o inverso. Também, não significa, necessariamente, que é apenas um hífen "-" que é utilizado, tem casos que são dois hifens "--".

Vamos lembrar do `type ls`, ele nós responde que é um `aliased to ls --color=auto`:

```
type ls
ls is aliased to `ls --color=auto`
```

A opção é passado com um duplo menos, "--", é comum que isso indique que na verdade trata-se de uma palavra inteira. Apenas um menos, "-", pode indicar uma abreviação. Isso, entretanto, também não quer dizer que é sempre assim. Temos cerca de trinta anos de história de *shell* e de lá para cá vários e vários programas foram criados, então, eles não obedecem, necessariamente, a esse padrão.

Existem aqueles que fogem do padrão: comando, opções e argumentos.

Através do `help` podemos saber um pouco mais a cerca dos programas que são `builtin` no nosso `shell`. Digitando o `help pwd` teremos o seguinte:

Aqui estão demostradas as ajudas e as opções que o comando `pwd` suporta, no caso o `-L` e o `-P`, ambos maiúsculos.

Repare que podemos usar o `type` e o `help` para nos ajudar a entender quais são os comandos internos e externos no nosso `bash`. Internos, são as funções, os comandos etc. E externos, são aqueles que são escritos de fora, adicionados, colocados, instalados dentro do nosso computador. Usamos o `help` para entender os comandos do nosso `shell`, isto é, compreender as suas coisas internas.

Por fim, relembrando, a ordem de execução que um comando costuma ter: comando + opção + argumento.

Lembrando, essa é a ordem que temos quando queremos executar um comando. Os comandos são de dois tipos: internos e externos. Através do `type` sabemos qual dos dois tipos nós temos.

Retomando, acabamos observando, mais a cima, o `type echo` e vimos que ele era um `builtin`. Então, se ele é algo interno ao `shell`, podemos usar o `help` para saber mais informações a seu respeito.

```
type echo
echo is a shell builtin
help echo
```

No help, encontraremos o seguinte:

Observando essa tela encontramos a opção `-n`, que não adiciona uma quebra de linha no final. Vamos testar essa opção. Primeiro, vamos inserir um `echo Guilherme` para comparação e, logo em seguida, um `echo -n Guilherme`. Repare que não ocorrerá, nesse segundo caso, uma quebra de texto.

```
echo Guilherme
Guilherme
echo -n Guilherme Guilherme
```

Além disso, repare que vimos o comando do `ls` e que ele recebia a opção `--color=auto`:

```
ls --color=auto
Desktop     Downloads           Music       Public  Videos
Documents   examples.desktop    Pictures    Templates
```

Ele utiliza essa opção para mostrar uma cor automática. Além do `--color=auto`, podemos também ter outras opções no `ls`. Por exemplo, o `-l`, que veremos com mais precisão mais adiante.

Se inserirmos o `ls -l`, ele mostrará uma espécie de listagem. Um em baixo do outro com uma série de informações sobre as quais nos debruçaremos mais adiante quando falarmos de arquivos, permissões e etc. Teremos o seguinte:

```
ls -l
total 44
drwr-xr-x 2 guilherme guilherme 4096 Mar 2 09:52 Desktop
drwr-xr-x 3 guilherme guilherme 4096 Mar 3 12:36 Documents
drwr-xr-x 2 guilherme guilherme 4096 Mar 4 13:43 Downloads
-rw-r-r-- 1 guilherme guilherme 8980 Mar 2 09:44 examples.desktop
drwr-xr-x 2 guilherme guilherme 4096 Mar 2 09:52 Music
drwr-xr-x 2 guilherme guilherme 4096 Mar 2 09:52 Pictures
drwr-xr-x 2 guilherme guilherme 4096 Mar 2 09:52 Public
drwr-xr-x 2 guilherme guilherme 4096 Mar 2 09:52 Templates
drwr-xr-x 2 guilherme guilherme 4096 Mar 2 09:52 Videos
```

Repare que no `ls` temos opções tanto de um único hífen, `-`, isto é, abreviadas, como as opções mais longas, por extenso, como o `--color`. Esse é um padrão que costuma aparecer nos comandos.

Temos também o `ls -a`, vamos ver o que aparece: ´´´ ls -a . Dowloads .profile .. examples.desktop Public .bash_history .gconf .sudo_as_admin_successful .bash_logout .gip-2.8 Templates .bashrc .ICEauthority .thunderbird .cache .local Videos .config .mozilla .Xauthority Desktop Music x.session-errors .dmrc Pictures .xsession-errors.old Documents .pki ```

Repare que temos diversos arquivos que aparecem com um ponto, esses arquivos, no sistema operacional do *Unix* e do *Linux*, são considerados "invisíveis". Essa característica de não serem visíveis deve-se ao fato de que, se dermos um `ls` normal, eles não irão aparecer. É preciso dar um `ls -a` para que eles se façam visíveis. O "-a" vêm de *all*, ou seja, todos, assim, usando ele podemos ver todos os arquivos.

Retomando

Foram passadas três opções. O `-a` que é para visualizar todos, o `-l` que é para exibir uma listagem, e o `--color` que é para exibir um colorido.

E se quisermos combinar todas essas opções?

Vamos, primeiro, dar uma limpada na tela com o `clear` e "Enter".

Em geral as opções pequenas, por exemplo, o `-l` e o `-a` podem aparecer separados. Se digitarmos o `ls -l -a` e dermos um "enter" será mostrada uma listagem de tudo que eles trazem. Teremos:

Mas também podemos fazer eles juntos, isto é, `ls -la`. Isso é possível quando a opção é apenas uma letra e ela possui um único hífen na frente, é comum que os comandos aceitem os dois de uma vez só, que as opções se concatenem. Isso é comum, o que não quer dizer que é algo obrigatório.

Mas, em relação ao `--color`, que está escrito por extenso, o padrão é que não se possa concatená-lo. Isto é, tem que ser separado. Por exemplo, `ls -la --color=auto` e aí poderemos executar o comando e teremos: