

BY BOBBY ILIEV

# Introduction to Bash Scripting

FOR DEVELOPERS



<b>About the book</b>	<b>7</b>
About the author	8
Sponsors	9
Ebook PDF Generation Tool	11
Ebook ePub Generation Tool	12
Book Cover	13
License	14
<b>Introduction to Bash scripting</b>	<b>15</b>
<b>Bash Structure</b>	<b>16</b>
<b>Bash Hello World</b>	<b>17</b>
<b>Bash Variables</b>	<b>19</b>
<b>Bash User Input</b>	<b>22</b>
<b>Bash Comments</b>	<b>24</b>
<b>Bash Arguments</b>	<b>25</b>
<b>Bash Arrays</b>	<b>27</b>
Substring in Bash :: Slicing	29
<b>Bash Conditional Expressions</b>	<b>31</b>

File expressions .....	32
String expressions .....	34
Arithmetic operators .....	36
Exit status operators .....	38
<b>Bash Conditionals .....</b>	<b>39</b>
If statement .....	40
If Else statement .....	41
Switch case statements .....	44
Conclusion .....	46
<b>Bash Loops .....</b>	<b>47</b>
For loops .....	48
While loops .....	50
Until Loops .....	52
Continue and Break .....	53
<b>Bash Functions .....</b>	<b>55</b>
<b>Debugging, testing and shortcuts .....</b>	<b>57</b>
<b>Creating custom bash commands .....</b>	<b>60</b>
Example .....	61
Making the change persistent .....	63
Listing all of the available aliases .....	64
Conclusion .....	65
<b>Write your first Bash script .....</b>	<b>66</b>
Planning the script .....	67
Writing the script .....	68

Adding comments .....	69
Adding your first variable .....	70
Adding your first function .....	71
Adding more functions challenge .....	73
The sample script .....	74
Conclusion .....	76
<b>Creating an interactive menu in Bash .....</b>	<b>77</b>
Planning the functionality .....	78
Adding some colors .....	80
<b>Adding the menu .....</b>	<b>81</b>
Testing the script .....	83
Conclusion .....	86
<b>Executing BASH scripts on Multiple Remote Servers .....</b>	<b>87</b>
Prerequisites .....	88
The BASH Script .....	89
Running the Script on all Servers .....	91
Conclusion .....	92
<b>Work with JSON in BASH using jq .....</b>	<b>93</b>
Planning the script .....	94
Installing jq .....	95
Parsing JSON with jq .....	97
Getting the first element with jq .....	99
Getting a value only for specific key .....	100
Using jq in a BASH script .....	101
Conclusion .....	104

<b>Working with Cloudflare API with Bash .....</b>	<b>105</b>
Prerequisites .....	106
Challenge - Script requirements .....	107
Example script .....	108
Conclusion .....	110
 <b>BASH Script parser to Summarize Your NGINX and Apache Access Logs .....</b>	<b>111</b>
 <b>Script requirements .....</b>	<b>112</b>
Example script .....	113
Running the script .....	114
Understanding the output .....	115
Conclusion .....	116
 <b>Sending emails with Bash and SSMTP .....</b>	<b>117</b>
Prerequisites .....	118
Installing SSMTP .....	119
Configuring SSMTP .....	120
Sending emails with SSMTP .....	121
Sending A File with SSMTP (optional) .....	122
Conclusion .....	123
 <b>Password Generator Bash Script .....</b>	<b>124</b>
:warning: Security .....	125
Script summary .....	126
Prerequisites .....	127
Generate a random password .....	128
The script .....	130
The full script: .....	131

Conclusion .....	132
Contributed by .....	133
<b>Redirection in Bash .....</b>	<b>134</b>
<b>Difference between Pipes and Redirections .....</b>	<b>135</b>
<b>Redirection in Bash .....</b>	<b>136</b>
STDIN (Standard Input) .....	137
STDOUT (Standard Output) .....	138
STDERR (Standard Error) .....	140
<b>Piping .....</b>	<b>142</b>
<b>HereDocument .....</b>	<b>144</b>
<b>HereString .....</b>	<b>146</b>
Summary .....	147
<b>Wrap Up .....</b>	<b>148</b>

- **This version was published on Feb 01 2021**

This is an open-source introduction to Bash scripting guide that will help you learn the basics of Bash scripting and start writing awesome Bash scripts that will help you automate your daily SysOps, DevOps, and Dev tasks. No matter if you are a DevOps/SysOps engineer, developer, or just a Linux enthusiast, you can use Bash scripts to combine different Linux commands and automate tedious and repetitive daily tasks so that you can focus on more productive and fun things.

The guide is suitable for anyone working as a developer, system administrator, or a DevOps engineer and wants to learn the basics of Bash scripting.

The first 13 chapters would be purely focused on getting some solid Bash scripting foundations, then the rest of the chapters would give you some real-life examples and scripts.

My name is Bobby Iliev, and I have been working as a Linux DevOps Engineer since 2014. I am an avid Linux lover and supporter of the open-source movement philosophy. I am always doing that which I cannot do in order that I may learn how to do it, and I believe in sharing knowledge.

I think it's essential always to keep professional and surround yourself with good people, work hard, and be nice to everyone. You have to perform at a consistently higher level than others. That's the mark of a true professional.

For more information, please visit my blog at <https://bobbyiliev.com>, follow me on Twitter [@bobbyiliev\\_](#) and [YouTube](#).



This book is made possible thanks to these fantastic companies!

The Streaming Database for Real-time Analytics.

[Materialize](#) is a reactive database that delivers incremental view updates. Materialize helps developers easily build with streaming data using standard SQL.

DigitalOcean is a cloud services platform delivering the simplicity developers love and businesses trust to run production applications at scale.

It provides highly available, secure, and scalable compute, storage, and networking solutions that help developers build great software faster.

Founded in 2012 with offices in New York and Cambridge, MA, DigitalOcean offers transparent and affordable pricing, an elegant user interface, and one of the largest libraries of open source resources available.

For more information, please visit <https://www.digitalocean.com> or follow [@digitalocean](#) on Twitter.

If you are new to DigitalOcean, you can get a free \$100 credit and spin up your own servers via this referral link here:

[Free \\$100 Credit For DigitalOcean](#)

The DevDojo is a resource to learn all things web development and web design. Learn on your lunch break or wake up and enjoy a cup of coffee with us to learn something new.

Join this developer community, and we can all learn together, build together, and grow together.

[Join DevDojo](#)

For more information, please visit <https://www.devdojo.com> or follow [@thedevdojo](https://twitter.com/thedevdojo) on Twitter.

This ebook was generated by [Ibis](#) developed by [Mohamed Said](#).

Ibis is a PHP tool that helps you write eBooks in markdown.

The ePub version was generated by [Pandoc](#).

The cover for this ebook was created with [Canva.com](https://www.canva.com).

If you ever need to create a graphic, poster, invitation, logo, presentation - or anything that looks good — give Canva a go.

## MIT License

Copyright (c) 2020 Bobby Iliev

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Welcome to this Bash basics training guide! In this **bash crash course**, you will learn the **Bash basics** so you could start writing your own Bash scripts and automate your daily tasks.

Bash is a Unix shell and command language. It is widely available on various operating systems, and it is also the default command interpreter on most Linux systems.

Bash stands for Bourne-Again SHell. As with other shells, you can use Bash interactively directly in your terminal, and also, you can use Bash like any other programming language to write scripts. This book will help you learn the basics of Bash scripting including Bash Variables, User Input, Comments, Arguments, Arrays, Conditional Expressions, Conditionals, Loops, Functions, Debugging, and testing.

In order to write Bash scripts, you just need a UNIX terminal and a text editor like Sublime Text, VS Code, or a terminal-based editor like vim or nano.

Let's start by creating a new file with a `.sh` extension. As an example, we could create a file called `devdojo.sh`.

To create that file, you can use the `touch` command:

```
touch devdojo.sh
```

Or you can use your text editor instead:

```
nano devdojo.sh
```

In order to execute/run a bash script file with the bash shell interpreter, the first line of a script file must indicate the absolute path to the bash executable:

```
#!/bin/bash
```

This is also called a [Shebang](#).

All that the shebang does is to instruct the operating system to run the script with the `/bin/bash` executable.



Once we have our `devdojo.sh` file created and we've specified the bash shebang on the very first line, we are ready to create our first `Hello World` bash script.

To do that, open the `devdojo.sh` file again and add the following after the `#!/bin/bash` line:

```
#!/bin/bash  
  
echo "Hello World!"
```

Save the file and exit.

After that make the script executable by running:

```
chmod +x devdojo.sh
```

After that execute the file:

```
./devdojo.sh
```

You will see a "Hello World" message on the screen.

Another way to run the script would be:

```
bash devdojo.sh
```

As bash can be used interactively, you could run the following command directly in your terminal and you would get the same result:



```
echo "Hello DevDojo!"
```

Putting a script together is useful once you have to combine multiple commands together.

As in any other programming language, you can use variables in Bash Scripting as well. However, there are no data types, and a variable in Bash can contain numbers as well as characters.

To assign a value to a variable, all you need to do is use the `=` sign:

```
name="DevDojo"
```

**Notice:** as an important note, you can not have spaces before and after the `=` sign.

After that, to access the variable, you have to use the `$` and reference it as shown below:

```
echo $name
```

Wrapping the variable name between curly brackets is not required, but is considered a good practice, and I would advise you to use them whenever you can:

```
echo ${name}
```

The above code would output: `DevDojo` as this is the value of our `name` variable.

Next, let's update our `devdojo.sh` script and include a variable in it.

Again, you can open the file `devdojo.sh` with your favorite text editor, I'm using nano here to open the file:

```
nano devdojo.sh
```

Adding our `name` variable here in the file, with a welcome message. Our file now looks like this:

```
#!/bin/bash  
  
name="DevDojo"  
  
echo "Hi there $name"
```

Save it and run the file using the command below:

```
./devdojo.sh
```

You would see the following output on your screen:

```
Hi there DevDojo
```

Here is a rundown of the script written in the file:

- `#!/bin/bash` - At first, we specified our shebang.
- `name=DevDojo` - Then, we defined a variable called `name` and assigned a value to it.
- `echo "Hi there $name"` - Finally, we output the content of the variable on the screen as a welcome message by using `echo`

You can also add multiple variables in the file as shown below:

```
#!/bin/bash  
  
name="DevDojo"  
greeting="Hello"  
  
echo "$greeting $name"
```

Save the file and run it again:

```
./devdojo.sh
```

You would see the following output on your screen:

```
Hello DevDojo
```

Note that you don't necessarily need to add semicolon ; at the end of each line. It works both ways, a bit like other programming language such as JavaScript!

With the previous script, we defined a variable, and we output the value of the variable on the screen with the `echo $name`.

Now let's go ahead and ask the user for input instead. To do that again, open the file with your favorite text editor and update the script as follows:

```
#!/bin/bash

echo "What is your name?"
read name

echo "Hi there $name"
echo "Welcome to DevDojo!"
```

The above will prompt the user for input and then store that input as a string/text in a variable.

We can then use the variable and print a message back to them.

The output of the above script would be:

- First run the script:

```
./devdojo.sh
```

- Then, you would be prompted to enter your name:

```
What is your name?
Bobby
```

- Once you've typed your name, just hit enter, and you will get the following output:

```
Hi there Bobby  
Welcome to DevDojo!
```

To reduce the code, we could change the first `echo` statement with the `read -p`, the `read` command used with `-p` flag will print a message before prompting the user for their input:

```
#!/bin/bash  
  
read -p "What is your name? " name  
  
echo "Hi there $name"  
echo "Welcome to DevDojo!"
```

Make sure to test this out yourself as well!

As with any other programming language, you can add comments to your script. Comments are used to leave yourself notes through your code.

To do that in Bash, you need to add the `#` symbol at the beginning of the line. Comments will never be rendered on the screen.

Here is an example of a comment:

```
# This is a comment and will not be rendered on the screen
```

Let's go ahead and add some comments to our script:

```
#!/bin/bash

# Ask the user for their name

read -p "What is your name? " name

# Greet the user
echo "Hi there $name"
echo "Welcome to DevDojo!"
```

Comments are a great way to describe some of the more complex functionality directly in your scripts so that other people could find their way around your code with ease.



You can pass arguments to your shell script when you execute it. To pass an argument, you just need to write it right after the name of your script. For example:

```
./devdojo.com your_argument
```

In the script, we can then use `$1` in order to reference the first argument that we specified.

If we pass a second argument, it would be available as `$2` and so on.

Let's create a short script called `arguments.sh` as an example:

```
#!/bin/bash  
  
echo "Argument one is $1"  
echo "Argument two is $2"  
echo "Argument three is $3"
```

Save the file and make it executable:

```
chmod +x arguments.sh
```

Then run the file and pass **3** arguments:

```
./arguments.sh dog cat bird
```

The output that you would get would be:

```
Argument one is dog
Argument two is cat
Argument three is bird
```

To reference all arguments, you can use `$@`:

```
#!/bin/bash

echo "All arguments: $@"
```

If you run the script again:

```
./arguments.sh dog cat bird
```

You will get the following output:

```
All arguments: dog cat bird
```

Another thing that you need to keep in mind is that `$0` is used to reference the script itself.

This is an excellent way to create self destruct the file if you need to or just get the name of the script.

For example, let's create a script that prints out the name of the file and deletes the file after that:

```
#!/bin/bash

echo "The name of the file is: $0 and it is going to be self-
deleted."

rm -f $0
```

You need to be careful with the self deletion and ensure that you have your script backed up before you self-delete it.

If you have ever done any programming, you are probably already familiar with arrays.

But just in case you are not a developer, the main thing that you need to know is that unlike variables, arrays can hold several values under one name.

You can initialize an array by assigning values divided by space and enclosed in `()`.  
Example:

```
my_array=("value 1" "value 2" "value 3" "value 4")
```

To access the elements in the array, you need to reference them by their numeric index.

**Notice:** keep in mind that you need to use curly brackets.

- Access a single element, this would output: `value 2`

```
echo ${my_array[1]}
```

- This would return the last element: `value 4`

```
echo ${my_array[-1]}
```

- As with command line arguments using `@` will return all arguments in the array, as follows: `value 1 value 2 value 3 value 4`

```
echo ${my_array[@]}
```

- Prepending the array with a hash sign (#) would output the total number of elements in the array, in our case it is 4:

```
echo ${#my_array[@]}
```

Make sure to test this and practice it at your end with different values.

Let's review the following example of slicing in a string in Bash:

```
#!/bin/bash

letters=( "A""B""C""D""E" )
echo ${letters[@]}
```

This command will print all the elements of an array.

Output:

```
$ ABCDE
```

Let's see a few more examples:

- Example 1

```
#!/bin/bash

letters=( "A""B""C""D""E" )
b=${letters:0:2}
echo "${b}"
```

This command will print array from starting index 0 to 2 where 2 is exclusive.

```
$ AB
```

- Example 2

```
#!/bin/bash

letters=( "A" "B" "C" "D" "E" )
b=${letters::5}
echo "${b}"
```

This command will print from base index 0 to 5, where 5 is exclusive and starting index is default set to 0 .

```
$ ABCDE
```

- Example 3

```
#!/bin/bash

letters=( "A" "B" "C" "D" "E" )
b=${letters:3}
echo "${b}"
```

This command will print from starting index 3 to end of array inclusive .

```
$ DE
```

In computer science, conditional statements, conditional expressions, and conditional constructs are features of a programming language, which perform different computations or actions depending on whether a programmer-specified boolean condition evaluates to true or false.

In Bash, conditional expressions are used by the `[[` compound command and the `[]` built-in commands to test file attributes and perform string and arithmetic comparisons.

Here is a list of the most popular Bash conditional expressions. You do not have to memorize them by heart. You can simply refer back to this list whenever you need it!

- True if file exists.

```
[[ -a ${file} ]]
```

- True if file exists and is a block special file.

```
[[ -b ${file} ]]
```

- True if file exists and is a character special file.

```
[[ -c ${file} ]]
```

- True if file exists and is a directory.

```
[[ -d ${file} ]]
```

- True if file exists.

```
[[ -e ${file} ]]
```

- True if file exists and is a regular file.

```
[[ -f ${file} ]]
```

- True if file exists and is a symbolic link.

```
[[ -h ${file} ]]
```



- True if file exists and is readable.

```
[[ -r ${file} ]]
```

- True if file exists and has a size greater than zero.

```
[[ -s ${file} ]]
```

- True if file exists and is writable.

```
[[ -w ${file} ]]
```

- True if file exists and is executable.

```
[[ -x ${file} ]]
```

- True if file exists and is a symbolic link.

```
[[ -L ${file} ]]
```

- True if the shell variable varname is set (has been assigned a value).

```
[[ -v ${varname} ]]
```

True if the length of the string is zero.

```
[[ -z ${string} ]]
```

True if the length of the string is non-zero.

```
[[ -n ${string} ]]
```

- True if the strings are equal. = should be used with the test command for POSIX conformance. When used with the [[ command, this performs pattern matching as described above (Compound Commands).

```
[[ ${string1} == ${string2} ]]
```

- True if the strings are not equal.

```
[[ ${string1} != ${string2} ]]
```

- True if string1 sorts before string2 lexicographically.

```
[[ ${string1} < ${string2} ]]
```

- True if string1 sorts after string2 lexicographically.



```
[[ ${string1} > ${string2} ]]
```

- Returns true if the numbers are **equal**

```
[[ ${arg1} -eq ${arg2} ]]
```

- Returns true if the numbers are **not equal**

```
[[ ${arg1} -ne ${arg2} ]]
```

- Returns true if arg1 is **less than** arg2

```
[[ ${arg1} -lt ${arg2} ]]
```

- Returns true if arg1 is **less than or equal** arg2

```
[[ ${arg1} -le ${arg2} ]]
```

- Returns true if arg1 is **greater than** arg2

```
[[ ${arg1} -gt ${arg2} ]]
```

- Returns true if arg1 is **greater than or equal** arg2

```
[[ ${arg1} -ge ${arg2} ]]
```

As a side note, arg1 and arg2 may be positive or negative integers.

As with other programming languages you can use **AND** & **OR** conditions:



```
[[ test_case_1 ]] && [[ test_case_2 ]] # And  
[[ test_case_1 ]] || [[ test_case_2 ]] # Or
```

- returns true if the the command was successful without any errors

```
[[ $? -eq 0 ]]
```

- returns true if the the command was not successful or had errors

```
[[ $? -gt 0 ]]
```

In the last section, we covered some of the most popular conditional expressions. We can now use them with standard conditional statements like `if`, `if-else` and `switch case` statements.

The format of an `if` statement in Bash is as follows:

```
if [[ some_test ]]
then
    <commands>
fi
```

Here is a quick example which would ask you to enter your name in case that you've left it empty:

```
#!/bin/bash

# Bash if statement example

read -p "What is your name? " name

if [[ -z ${name} ]]
then
    echo "Please enter your name!"
fi
```



With an **if-else** statement, you can specify an action in case that the condition in the **if** statement does not match. We can combine this with the conditional expressions from the previous section as follows:

```
#!/bin/bash

# Bash if statement example

read -p "What is your name? " name

if [[ -z ${name} ]]
then
    echo "Please enter your name!"
else
    echo "Hi there ${name}"
fi
```

You can use the above if statement with all of the conditional expressions from the previous chapters:

```
#!/bin/bash

admin="devdojo"

read -p "Enter your username? " username

# Check if the username provided is the admin

if [[ "${username}" == "${admin}" ]] ; then
    echo "You are the admin user!"
else
    echo "You are NOT the admin user!"
fi
```

Here is another example of an **if** statement which would check your current **User ID** and would not allow you to run the script as the **root** user:

```
#!/bin/bash

if (( $EUID == 0 )); then
    echo "Please do not run as root"
    exit
fi
```

If you put this on top of your script it would exit in case that the EUID is 0 and would not execute the rest of the script. This was discussed on [the DigitalOcean community forum](#).

You can also test multiple conditions with an `if` statement. In this example we want to make sure that the user is neither the admin user or the root user to ensure the script is incapable of causing too much damage. We'll use the `or` operator in this example, noted by `||`. This means that either of the conditions needs to be true. If we used the `and` operator of `&&` then both conditions would need to be true.

```
#!/bin/bash

admin="devdojo"

read -p "Enter your username? " username

# Check if the username provided is the admin

if [[ "${username}" != "${admin}" ]] || [[ $EUID != 0 ]] ;
then
    echo "You are not the admin or root user, but please be
safe!"
else
    echo "You are the admin user! This could be very
destructive!"
fi
```

If you have multiple conditions and scenerios, then can use `elif` statement with `if` and `else` statements.

```
#!/bin/bash

read -p "Enter a number: " num

if [[ $num -gt 0 ]] ; then
    echo "The number is positive"
elif [[ $num -lt 0 ]] ; then
    echo "The number is negative"
else
    echo "The number is 0"
fi
```

As in other programming languages, you can use a **case** statement to simplify complex conditionals when there are multiple different choices. So rather than using a few **if**, and **if-else** statements, you could use a single **case** statement.

The Bash **case** statement syntax looks like this:

```
case $some_variable in
    pattern_1)
        commands
        ;;
    pattern_2| pattern_3)
        commands
        ;;
    *)
        default commands
        ;;
esac
```

A quick rundown of the structure:

- All **case** statements start with the **case** keyword.
- On the same line as the **case** keyword, you need to specify a variable or an expression followed by the **in** keyword.
- After that, you have your **case** patterns, where you need to use **)** to identify the end of the pattern.
- You can specify multiple patterns divided by a pipe: **|**.
- After the pattern, you specify the commands that you would like to be executed in case that the pattern matches the variable or the expression that you've specified.
- All clauses have to be terminated by adding **;;** at the end.
- You can have a default statement by adding a **\*** as the pattern.
- To close the **case** statement, use the **esac** (case typed backwards) keyword.

Here is an example of a Bash **case** statement:

```
#!/bin/bash

read -p "Enter the name of your car brand: " car

case $car in

    Tesla)
        echo -n "${car}'s car factory is in the USA."
        ;;

    BMW | Mercedes | Audi | Porsche)
        echo -n "${car}'s car factory is in Germany."
        ;;

    Toyota | Mazda | Mitsubishi | Subaru)
        echo -n "${car}'s car factory is in Japan."
        ;;

    *)
        echo -n "${car} is an unknown car brand"
        ;;

esac
```

With this script, we are asking the user to input a name of a car brand like Telsa, BMW, Mercedes and etc.

Then with a **case** statement, we check the brand name and if it matches any of our patterns, and if so, we print out the factory's location.

If the brand name does not match any of our **case** statements, we print out a default message: **an unknown car brand**.

I would advise you to try and modify the script and play with it a bit so that you could practice what you've just learned in the last two chapters!

For more examples of Bash `case` statements, make sure to check chapter 16, where we would create an interactive menu in Bash using a `cases` statement to process the user input.

As with any other language, loops are very convenient. With Bash you can use `for` loops, `while` loops, and `until` loops.

Here is the structure of a for loop:

```
for var in ${list}
do
    your_commands
done
```

Example:

```
#!/bin/bash

users="devdojo bobby tony"

for user in ${users}
do
    echo "${user}"
done
```

A quick rundown of the example:

- First, we specify a list of users and store the value in a variable called `$users`.
- After that, we start our `for` loop with the `for` keyword
- Then we define a new variable which would represent each item from the list that we give. In our case, we define a variable called `user`, which would represent each user from the `$users` variable.
- Then we specify the `in` keyword followed by our list that we will loop through
- On the next line, we use the `do` keyword, which indicates what we will do for each iteration of the loop
- Then we specify the commands that we want to run
- Finally, we close the loop with the `done` keyword

You can also use `for` to process a series of numbers. For example here is one way to loop through from 1 to 10:



```
#!/bin/bash

for num in {1..10}
do
    echo ${num}
done
```

The structure of a while loop is quite similar to the **for** loop:

```
while [ your_condition ]
do
    your_conditions
done
```

Here is an example of a **while** loop:

```
#!/bin/bash

counter=1
while [[ $counter -le 10 ]]
do
    echo $counter
    ((counter++))
done
```

First, we specified a counter variable and set it to **1**, then inside the loop, we added counter by using this statement here: **((counter++))**. That way, we make sure that the loop will run 10 times only and would not run forever. The loop will complete as soon as the counter becomes 10, as this is what we've set as the condition: **while [[ \$counter -le 10 ]]**.

Let's create a script that asks the user for their name and not allow an empty input:

```
#!/bin/bash

read -p "What is your name? " name

while [[ -z ${name} ]]
do
    echo "Your name can not be blank. Please enter a valid name!"
    read -p "Enter your name again? " name
done

echo "Hi there ${name}"
```

Now, if you run the above and just press enter without providing input, the loop would run again and ask you for your name again and again until you actually provide some input.

The difference between `until` and `while` loops is that the `until` loop will run the commands within the loop until the condition becomes true.

Structure:

```
until [ your_condition ]
do
    your_commands
done
```

Example:

```
#!/bin/bash

count=1
until [ $count -gt 10 ]
do
    echo $count
    ((count++))
done
```

As with other languages, you can use `continue` and `break` with your bash scripts as well:

- `continue` tells your bash script to stop the current iteration of the loop and start the next iteration.

The syntax of the continue statement is as follows:

```
continue [n]
```

The [n] argument is optional and can be greater than or equal to 1. When [n] is given, the n-th enclosing loop is resumed. `continue 1` is equivalent to `continue`.

```
#!/bin/bash

for i in 1 2 3 4 5
do
    if [ $i -eq 2 ]
    then
        echo "skipping number 2"
        continue
    fi
    echo "I is equal to $i"
done
```

We can also use `continue` command in similar way to `break` command for controlling multiple loops.

- `break` tells your bash script to end the loop straight away.

The syntax of the break statement takes the following form:

```
break [n]
```

[n] is an optional argument and must be greater than or equal to 1. When [n] is provided, the n-th enclosing loop is exited. `break 1` is equivalent to `break`.

Example:

```
#!/bin/bash

num=1
while [ $num -lt 10 ]
do
    if [ $num -eq 5 ]
    then
        break
    fi
    ((num++))
done
echo "Loop completed"
```

We can also use break command with multiple loops. If we want to exit out of current working loop whether inner or outer loop, we simply use break but if we are in inner loop & want to exit out of outer loop, we use break 2.

Example:

```
#!/bin/bash

for (( a = 1; a < 10; a++ ))
do
    echo "outer loop: $a"
    for (( b = 1; b < 100; b++ ))
    do
        if [ $b -gt 5 ]
        then
            break 2
        fi
        echo "Inner loop: $b "
    done
done
```

The bash script will begin with a=1 & will move to inner loop and when it reaches b=5, it will break the outer loop. We can use break only instead of break 2, to break inner loop & see how it affects the output.

Functions are a great way to reuse code. The structure of a function in bash is quite similar to most languages:

```
function function_name() {  
    your_commands  
}
```

You can also omit the `function` keyword at the beginning, which would also work:

```
function_name() {  
    your_commands  
}
```

I prefer putting it there for better readability. But it is a matter of personal preference.

Example of a "Hello World!" function:

```
#!/bin/bash  
  
function hello(){  
    echo "Hello World Function!"  
}  
  
hello
```

**Notice:** One thing to keep in mind is that you should not add the parenthesis when you call the function.

Passing arguments to a function work in the same way as passing arguments to a script:

```
#!/bin/bash

function hello(){
    echo "Hello $1!"
}

hello DevDojo
```

Functions should have comments mentioning description, global variables, arguments, outputs, and returned values, if applicable

```
#####
# Description: Hello function
# Globals:
#   None
# Arguments:
#   Single input argument
# Outputs:
#   Value of input argument
# Returns:
#   0 if successful, non-zero on error.
#####
function hello(){
    echo "Hello $1!"
}
```

In the next few chapters we will be using functions a lot!



In order to debug your bash scripts, you can use `-x` when executing your scripts:

```
bash -x ./your_script.sh
```

Or you can add `set -x` before the specific line that you want to debug, `set -x` enables a mode of the shell where all executed commands are printed to the terminal.

Another way to test your scripts is to use this fantastic tool here:

<https://www.shellcheck.net/>

Just copy and paste your code into the textbox, and the tool will give you some suggestions on how you can improve your script.

You can also run the tool directly in your terminal:

<https://github.com/koalaman/shellcheck>

If you like the tool, make sure to star it on GitHub and contribute!

As a SysAdmin/DevOps, I spend a lot of my day in the terminal. Here are my favorite shortcuts that help me do tasks quicker while writing Bash scripts or just while working in the terminal.

The below two are particularly useful if you have a very long command.

- Delete everything from the cursor to the end of the line:

```
Ctrl + k
```

- Delete everything from the cursor to the start of the line:

`Ctrl + u`

- Delete one word backward from cursor:

`Ctrl + w`

- Search your history backward. This is probably the one that I use the most. It is really handy and speeds up my work-flow a lot:

`Ctrl + r`

- Clear the screen, I use this instead of typing the `clear` command:

`Ctrl + l`

- Stops the output to the screen:

`Ctrl + s`

- Enable the output to the screen in case that previously stopped by `Ctrl + s`:

`Ctrl + q`

- Terminate the current command

`Ctrl + c`

- Throw the current command to background:



Ctrl + z

I use those regularly every day, and it saves me a lot of time.

If you think that I've missed any feel free to join the discussion on [the DigitalOcean community forum](#)!

As a developer or system administrator, you might have to spend a lot of time in your terminal. I always try to look for ways to optimize any repetitive tasks.

One way to do that is to either write short bash scripts or create custom commands also known as aliases. For example, rather than typing a really long command every time you could just create a short cut for it.

Let's start with the following scenario, as a system admin, you might have to check the connections to your web server quite often, so I will use the `netstat` command as an example.

What I would usually do when I access a server that is having issues with the connections to port 80 or 443 is to check if there are any services listening on those ports and the number of connections to the ports.

The following `netstat` command would show us how many TCP connections on port 80 and 443 we currently have:

```
netstat -plant | grep '80\|443' | grep -v LISTEN | wc -l
```

This is quite a lengthy command so typing it every time might be time-consuming in the long run especially when you want to get that information quickly.

To avoid that, we can create an alias, so rather than typing the whole command, we could just type a short command instead. For example, let's say that we wanted to be able to type `conn` (short for connections) and get the same information. All we need to do in this case is to run the following command:

```
alias conn="netstat -plant | grep '80\|443' | grep -v LISTEN | wc -l"
```

That way we are creating an alias called `conn` which would essentially be a 'short cut' for our long `netstat` command. Now if you run just `conn`:

```
conn
```

You would get the same output as the long `netstat` command. You can get even more creative and add some info messages like this one here:

```
alias conn="echo 'Total connections on port 80 and 443:' ;  
netstat -plant | grep '80\|443' | grep -v LISTEN | wc -l"
```

Now if you run `conn` you would get the following output:

```
Total connections on port 80 and 443:  
12
```

Now if you log out and log back in, your alias would be lost. In the next step you will see how to make this persistent.

In order to make the change persistent, we need to add the `alias` command in our shell profile file.

By default on Ubuntu this would be the `~/.bashrc` file, for other operating systems this might be the `~/.bash_profile`. With your favorite text editor open the file:

```
nano ~/.bashrc
```

Go to the bottom and add the following:

```
alias conn="echo 'Total connections on port 80 and 443:' ;  
netstat -plant | grep '80\|443' | grep -v LISTEN | wc -l"
```

Save and then exit.

That way now even if you log out and log back in again your change would be persisted and you would be able to run your custom bash command.

To list all of the available aliases for your current shell, you have to just run the following command:



```
alias
```

This would be handy in case that you are seeing some weird behavior with some commands.



This is one way of creating custom bash commands or bash aliases.

Of course, you could actually write a bash script and add the script inside your `/usr/bin` folder, but this would not work if you don't have root or sudo access, whereas with aliases you can do it without the need of root access.

**Notice:** This was initially posted on [DevDojo.com](https://devdojo.com)

Let's try to put together what we've learned so far and create our first Bash script!

As an example, we will write a script that would gather some useful information about our server like:

- Current Disk usage
- Current CPU usage
- Current RAM usage
- Check the exact Kernel version

Feel free to adjust the script by adding or removing functionality so that it matches your needs.

The first thing that you need to do is to create a new file with a `.sh` extension. I will create a file called `status.sh` as the script that we will create would give us the status of our server.

Once you've created the file, open it with your favorite text editor.

As we've learned in chapter 1, on the very first line of our Bash script we need to specify the so-called [Shebang](#):

```
#!/bin/bash
```

All that the shebang does is to instruct the operating system to run the script with the `/bin/bash` executable.

Next, as discussed in chapter 6, let's start by adding some comments so that people could easily figure out what the script is used for. To do that right after the shebang you can just add the following:

```
#!/bin/bash
```

```
# Script that returns the current server status
```

Then let's go ahead and apply what we've learned in chapter 4 and add some variables which we might want to use throughout the script.

To assign a value to a variable in bash, you just have to use the `=` sign. For example, let's store the hostname of our server in a variable so that we could use it later:

```
server_name=$(hostname)
```

By using `$()` we tell bash to actually interpret the command and then assign the value to our variable.

Now if we were to echo out the variable we would see the current hostname:

```
echo $server_name
```

As you already know after reading chapter 12, in order to create a function in bash you need to use the following structure:

```
function function_name() {  
    your_commands  
}
```

Let's create a function that returns the current memory usage on our server:

```
function memory_check() {  
    echo ""  
    echo "The current memory usage on ${server_name} is: "  
    free -h  
    echo ""  
}
```

Quick run down of the function:

- `function memory_check() {` - this is how we define the function
- `echo ""` - here we just print a new line
- `echo "The current memory usage on ${server_name} is: "` - here we print all a small message and the `$server_name` variable
- `}` - finally this is how we close the function

Then once the function has been defined, in order to call it, just use the name of the function:

```
# Define the function
function memory_check() {
    echo ""
    echo "The current memory usage on ${server_name} is: "
    free -h
    echo ""
}

# Call the function
memory_check
```



Before checking out the solution, I would challenge you to use the function from above and write a few functions by yourself.

The functions should do the following:

- Current Disk usage
- Current CPU usage
- Current RAM usage
- Check the exact Kernel version

Feel free to use google if you are not sure what commands you need to use in order to get that information.

Once you are ready, feel free to scroll down and check how we've done it and compare the results!

Note that there are multiple correct ways of doing it!

Here's what the end result would look like:

```
#!/bin/bash

##
# BASH script that checks:
#   - Memory usage
#   - CPU load
#   - Number of TCP connections
#   - Kernel version
##

server_name=$(hostname)

function memory_check() {
    echo ""
    echo "Memory usage on ${server_name} is: "
    free -h
    echo ""
}

function cpu_check() {
    echo ""
    echo "CPU load on ${server_name} is: "
    echo ""
    uptime
    echo ""
}

function tcp_check() {
    echo ""
    echo "TCP connections on ${server_name}: "
    echo ""
    cat /proc/net/tcp | wc -l
    echo ""
}

function kernel_check() {
    echo ""
    echo "Kernel version on ${server_name} is: "
    echo ""
}
```

```
        uname -r
    echo ""
}

function all_checks() {
    memory_check
    cpu_check
    tcp_check
    kernel_check
}

all_checks
```

Bash scripting is awesome! No matter if you are a DevOps/SysOps engineer, developer, or just a Linux enthusiast, you can use Bash scripts to combine different Linux commands and automate boring and repetitive daily tasks, so that you can focus on more productive and fun things!

**Notice:** This was initially posted on [DevDojo.com](https://devdojo.com)

In this tutorial, I will show you how to create a multiple-choice menu in Bash so that your users could choose between what action should be executed!

We would reuse some of the code from the previous chapter, so if you have not read it yet make sure to do so.

Let's start again by going over the main functionality of the script:

- Checks the current Disk usage
- Checks the current CPU usage
- Checks the current RAM usage
- Checks the check the exact Kernel version

In case that you don't have it on hand, here is the script itself:

```
#!/bin/bash

##
# BASH menu script that checks:
#   - Memory usage
#   - CPU load
#   - Number of TCP connections
#   - Kernel version
##

server_name=$(hostname)

function memory_check() {
    echo ""
    echo "Memory usage on ${server_name} is: "
    free -h
    echo ""
}

function cpu_check() {
    echo ""
    echo "CPU load on ${server_name} is: "
    echo ""
    uptime
    echo ""
}

function tcp_check() {
    echo ""
    echo "TCP connections on ${server_name}: "
    echo ""
    cat /proc/net/tcp | wc -l
}
```

```
    echo ""
}

function kernel_check() {
    echo ""
    echo "Kernel version on ${server_name} is: "
    echo ""
    uname -r
    echo ""
}

function all_checks() {
    memory_check
    cpu_check
    tcp_check
    kernel_check
}
```

We will then build a menu that allows the user to choose which function to be executed.

Of course, you can adjust the function or add new ones depending on your needs.

In order to make the menu a bit more 'readable' and easy to grasp at first glance, we will add some color functions.

At the beginning of your script add the following color functions:

```
##
# Color Variables
##
green='\e[32m'
blue='\e[34m'
clear='\e[0m'

##
# Color Functions
##

ColorGreen(){
    echo -ne $green$1$clear
}
ColorBlue(){
    echo -ne $blue$1$clear
}
```

You can use the color functions as follows:

```
echo -ne $(ColorBlue 'Some text here')
```

The above would output the **Some text here** string and it would be blue!



Finally, to add our menu, we will create a separate function with a case switch for our menu options:

```
menu(){
echo -ne "
My First Menu
$(ColorGreen '1') Memory usage
$(ColorGreen '2') CPU load
$(ColorGreen '3') Number of TCP connections
$(ColorGreen '4') Kernel version
$(ColorGreen '5') Check All
$(ColorGreen '0') Exit
$(ColorBlue 'Choose an option:') "
    read a
    case $a in
        1) memory_check ; menu ;;
        2) cpu_check ; menu ;;
        3) tcp_check ; menu ;;
        4) kernel_check ; menu ;;
        5) all_checks ; menu ;;
        0) exit 0 ;;
        *) echo -e $red"Wrong option."$clear;
WrongCommand;;
    esac
}
```

First we just echo out the menu options with some color:

```

echo -ne "
My First Menu
$(ColorGreen '1') Memory usage
$(ColorGreen '2') CPU load
$(ColorGreen '3') Number of TCP connections
$(ColorGreen '4') Kernel version
$(ColorGreen '5') Check All
$(ColorGreen '0') Exit
$(ColorBlue 'Choose an option:') "

```

Then we read the answer of the user and store it in a variable called `$a`:

```

read a

```

Finally, we have a switch case which triggers a different function depending on the value of `$a`:

```

case $a in
    1) memory_check ; menu ;;
    2) cpu_check ; menu ;;
    3) tcp_check ; menu ;;
    4) kernel_check ; menu ;;
    5) all_checks ; menu ;;
    0) exit 0 ;;
    *) echo -e $red"Wrong option."$clear;
WrongCommand;;
esac

```

At the end we need to call the menu function to actually print out the menu:

```

# Call the menu function
menu

```

In the end, your script will look like this:

```
#!/bin/bash

##
# BASH menu script that checks:
#   - Memory usage
#   - CPU load
#   - Number of TCP connections
#   - Kernel version
##

server_name=$(hostname)

function memory_check() {
    echo ""
    echo "Memory usage on ${server_name} is: "
    free -h
    echo ""
}

function cpu_check() {
    echo ""
    echo "CPU load on ${server_name} is: "
    echo ""
    uptime
    echo ""
}

function tcp_check() {
    echo ""
    echo "TCP connections on ${server_name}: "
    echo ""
    cat /proc/net/tcp | wc -l
    echo ""
}

function kernel_check() {
    echo ""
    echo "Kernel version on ${server_name} is: "
    echo ""
}
```

```

        uname -r
    echo ""
}

function all_checks() {
    memory_check
    cpu_check
    tcp_check
    kernel_check
}

##
# Color Variables
##
green='\e[32m'
blue='\e[34m'
clear='\e[0m'

##
# Color Functions
##

ColorGreen(){
    echo -ne $green$1$clear
}
ColorBlue(){
    echo -ne $blue$1$clear
}

menu(){
    echo -ne "
My First Menu
$(ColorGreen '1') Memory usage
$(ColorGreen '2') CPU load
$(ColorGreen '3') Number of TCP connections
$(ColorGreen '4') Kernel version
$(ColorGreen '5') Check All
$(ColorGreen '0') Exit
$(ColorBlue 'Choose an option:') "
    read a
    case $a in
        1) memory_check ; menu ;;
        2) cpu_check ; menu ;;
        3) tcp_check ; menu ;;
        4) kernel_check ; menu ;;
        5) all_checks ; menu ;;
    esac
}

```

```
0) exit 0 ;;
*) echo -e $red"Wrong option."$clear;
WrongCommand;;
    esac
}

# Call the menu function
menu
```

To test the script, create a new file with a `.sh` extension, for example: `menu.sh` and then run it:

```
bash menu.sh
```

The output that you would get will look like this:

```
My First Menu
1) Memory usage
2) CPU load
3) Number of TCP connections
4) Kernel version
5) Check All
0) Exit
Choose an option:
```

You will be able to choose a different option from the list and each number will call a different function from the script:



You now know how to create a Bash menu and implement it in your scripts so that users could select different values!

**Notice:** This content was initially posted on [DevDojo.com](https://devdojo.com)

Any command that you can run from the command line can be used in a bash script. Scripts are used to run a series of commands. Bash is available by default on Linux and macOS operating systems.

Let's have a hypothetical scenario where you need to execute a BASH script on multiple remote servers, but you don't want to manually copy the script to each server, then again login to each server individually and only then execute the script.

Of course you could use a tool like Ansible but lets learn how to do that with Bash!

For this example I will use 3 remote Ubuntu servers deployed on DigitalOcean. If you don't have a Digital Ocean account yet, you can sign up for DigitalOcean and get \$100 free credit via this referral link here:

<https://m.do.co/c/2a9bba940f39>

Once you have your Digital Ocean account ready go ahead and deploy 3 droplets.

I've gone ahead and created 3 Ubuntu servers:



I'll put a those servers IP's in a `servers.txt` file which I would use to loop though with our Bash script.

If you are new to DigitalOcean you can follow the steps on how to create a Droplet here:

- [How to Create a Droplet from the DigitalOcean Control Panel](#)

You can also follow the steps from this video here on how to do your initial server setul:

- [How to do your Initial Server Setup with Ubuntu](#)

Or even better, you can follow this article here on how to automate your initial server setup with Bash:

[Automating Initial Server Setup with Ubuntu 18.04 with Bash](#)

With the 3 new servers in place, we can go ahead and focus on running our Bash script on all of them with a single command!



I will reuse the demo script from the previous chapter with some slight changes. It simply executes a few checks like the current memory usage, the current CPU usage, the number of TCP connections and the version of the kernel.

```
#!/bin/bash

##
# BASH script that checks the following:
#   - Memory usage
#   - CPU load
#   - Number of TCP connections
#   - Kernel version
##

##
# Memory check
##
server_name=$(hostname)

function memory_check() {
    echo "#####"
    echo "The current memory usage on ${server_name} is: "
    free -h
    echo "#####"
}

function cpu_check() {
    echo "#####"
    echo "The current CPU load on ${server_name} is: "
    echo ""
    uptime
    echo "#####"
}

function tcp_check() {
    echo "#####"
    echo "Total TCP connections on ${server_name}: "
    echo ""
    cat /proc/net/tcp | wc -l
    echo "#####"
}
```

```
}

function kernel_check() {
    echo "#####"
    echo "The exact Kernel version on ${server_name} is: "
    echo ""
    uname -r
    echo "#####"
}

function all_checks() {
    memory_check
    cpu_check
    tcp_check
    kernel_check
}

all_checks
```

Copy the code bellow and add this in a file called `remote_check.sh`. You can also get the script from [here](#).

Now that we have the script and the servers ready and that we've added those servers in our servers.txt file we can run the following command to loop through all servers and execute the script remotely without having to copy the script to each server and individually connect to each server.

```
for server in $(cat servers.txt) ; do ssh your_user@${server}  
'bash -s' < ./remote_check.sh ; done
```

What this for loop does is, it goes through each server in the servers.txt file and then it runs the following command for each item in the list:

```
ssh your_user@the_server_ip 'bash -s' < ./remote_check.sh
```

You would get the following output:



This is just a really simple example on how to execute a simple script on multiple servers without having to copy the script to each server and without having to access the servers individually.

Of course you could run a much more complex script and on many more servers.

If you are interested in automation, I would recommend checking out the Ansible resources page on the DigitalOcean website:

[Ansible Resources](#)

**Notice:** This content was initially posted on [DevDojo](#)

The **jq** command-line tool is a lightweight and flexible command-line **JSON** processor. It is great for parsing JSON output in BASH.

One of the great things about **jq** is that it is written in portable C, and it has zero runtime dependencies. All you need to do is to download a single binary or use a package manager like apt and install it with a single command.

For the demo in this tutorial, I would use an external REST API that returns a simple JSON output called the [QuizAPI](#):

<https://quizapi.io/>

If you want to follow along make sure to get a free API key here:

<https://quizapi.io/clientarea/settings/token>

The QuizAPI is free for developers.

There are many ways to install **jq** on your system. One of the most straight forward ways to do so is to use the package manager depending on your OS.

Here is a list of the commands that you would need to use depending on your OS:

- Install jq on Ubuntu/Debian:

```
sudo apt-get install jq
```

- Install jq on Fedora:

```
sudo dnf install jq
```

- Install jq on openSUSE:

```
sudo zypper install jq
```

- Install jq on Arch:

```
sudo pacman -S jq
```

- Installing on Mac with Homebrew:

```
brew install jq
```

- Install on Mac with MacPort:

```
port install jq
```

If you are using other OS, I would recommend taking a look at the official

documentation here for more information:

<https://stedolan.github.io/jq/download/>

Once you have jq installed you can check your current version by running this command:

```
jq --version
```



Once you have `jq` installed and your QuizAPI API Key, you can parse the JSON output of the QuizAPI directly in your terminal.

First, create a variable that stores your API Key:

```
API_KEY=YOUR_API_KEY_HERE
```

In order to get some output from one of the endpoints of the QuizAPI you can use the `curl` command:

```
curl  
"https://quizapi.io/api/v1/questions?apiKey=${API_KEY}&limit=1  
0"
```

For a more specific output, you can use the QuizAPI URL Generator here:

<https://quizapi.io/api-config>

After running the `curl` command, the output which you would get would look like this:



This could be quite hard to read, but thanks to the `jq` command-line tool, all we need to do is pipe the `curl` command to `jq` and we would see a nice formatted JSON output:

```
curl  
"https://quizapi.io/api/v1/questions?apiKey=${API_KEY}&limit=1  
0" | jq
```

Note the `| jq` at the end.

In this case the output that you would get would look something like this:



Now, this looks much nicer! The `jq` command-line tool formatted the output for us and

added some nice coloring!

Let's say that we only wanted to get the first element from the JSON output, in order to do that we have to just specify the index that we want to see with the following syntax:

```
jq .[0]
```

Now, if we run the curl command again and pipe the output to jq .[0] like this:

```
curl  
"https://quizapi.io/api/v1/questions?apiKey=${API_KEY}&limit=1  
0" | jq.[0]
```

You will only get the first element and the output will look like this:



Sometimes you might want to get only the value of a specific key only, let's say in our example the QuizAPI returns a list of questions along with the answers, description and etc. but what if you wanted to get the Questions only without the additional information?

This is going to be quite straight forward with `jq`, all you need to do is add the key after `jq` command, so it would look something like this:

```
jq .[].question
```

We have to add the `.[]` as the QuizAPI returns an array and by specifying `.[]` we tell `jq` that we want to get the `.question` value for all of the elements in the array.

The output that you would get would look like this:



As you can see we now only get the questions without the rest of the values.

Let's go ahead and create a small bash script which should output the following information for us:

- Get only the first question from the output
- Get all of the answers for that question
- Assign the answers to variables
- Print the question and the answers
- To do that I've put together the following script:

**Notice:** make sure to change the `API_KEY` part with your actual QuizAPI key:

```
#!/bin/bash

##
# Make an API call to QuizAPI and store the output in a
variable
##
output=$(curl
'https://quizapi.io/api/v1/questions?apiKey=API_KEY&limit=10'
2>/dev/null)

##
# Get only the first question
##
output=$(echo $output | jq .[0])

##
# Get the question
##
question=$(echo $output | jq .question)

##
# Get the answers
##

answer_a=$(echo $output | jq .answers.answer_a)
answer_b=$(echo $output | jq .answers.answer_b)
answer_c=$(echo $output | jq .answers.answer_c)
answer_d=$(echo $output | jq .answers.answer_d)

##
# Output the question
##

echo "
Question: ${question}

A) ${answer_a}
B) ${answer_b}
C) ${answer_c}
D) ${answer_d}

"
```

If you run the script you would get the following output:



We can even go further by making this interactive so that we could actually choose the answer directly in our terminal.

There is already a bash script that does this by using the QuizAPI and `jq`:

You can take a look at that script here:

- <https://github.com/QuizApi/QuizAPI-BASH/blob/master/quiz.sh>

The **jq** command-line tool is an amazing tool that gives you the power to work with JSON directly in your BASH terminal.

That way you can easily interact with all kinds of different REST APIs with BASH.

For more information, you could take a look at the official documentation here:

- <https://stedolan.github.io/jq/manual/>

And for more information on the **QuizAPI**, you could take a look at the official documentation here:

- <https://quizapi.io/docs/1.0/overview>

**Notice:** This content was initially posted on [DevDojo.com](https://devdojo.com)



I host all of my websites on **DigitalOcean** Droplets and I also use Cloudflare as my CDN provider. One of the benefits of using Cloudflare is that it reduces the overall traffic to your user and also hides your actual server IP address behind their CDN.

My personal favorite Cloudflare feature is their free DDoS protection. It has saved my servers multiple times from different DDoS attacks. They have a cool API that you could use to enable and disable their DDoS protection easily.

This chapter is going to be an exercise! I challenge you to go ahead and write a short bash script that would enable and disable the Cloudflare DDoS protection for your server automatically if needed!

Before following this guide here, please set up your Cloudflare account and get your website ready. If you are not sure how to do that you can follow these steps here:

[Create a Cloudflare account and add a website.](#)

Once you have your Cloudflare account, make sure to obtain the following information:

- A Cloudflare account
- Cloudflare API key
- Cloudflare Zone ID

Also, Make sure curl is installed on your server:

```
curl --version
```

If curl is not installed you need to run the following:

- For RedHat/CentOs:

```
yum install curl
```

- For Debian/Ubuntu

```
apt-get install curl
```

The script needs to monitor the CPU usage on your server and if the CPU usage gets high based on the number vCPU it would enable the Cloudflare DDoS protection automatically via the Cloudflare API.

The main features of the script should be:

- Checks the script CPU load on the server
- In case of a CPU spike the script triggers an API call to Cloudflare and enables the DDoS protection feature for the specified zone
- After the CPU load is back to normal the script would disable the "I'm under attack" option and set it back to normal

I already have prepared a demo script which you could use as a reference. But I encourage you to try and write the script yourself first and only then take a look at my script!

To download the script just run the following command:

```
wget  
https://raw.githubusercontent.com/bobbyiliev/cloudflare-ddos-p  
rotection/main/protection.sh
```

Open the script with your favorite text editor:

```
nano protection.sh
```

And update the following details with your Cloudflare details:

```
CF_ZONE_ID=YOUR_CF_ZONE_ID  
CF_EMAIL_ADDRESS=YOUR_CF_EMAIL_ADDRESS  
CF_API_KEY=YOUR_CF_API_KEY
```

After that make the script executable:

```
chmod +x ~/protection.sh
```

Finally, set up 2 Cron jobs to run every 30 seconds. To edit your crontab run:

```
crontab -e
```

And add the following content:

```
* * * * * /path-to-the-script/cloudflare/protection.sh
* * * * * ( sleep 30 ; /path-to-the-
script/cloudflare/protection.sh )
```

Note that you need to change the path to the script with the actual path where you've stored the script at.

This is quite straight forward and budget solution, one of the downsides of the script is that if your server gets unresponsive due to an attack, the script might not be triggered at all.

Of course, a better approach would be to use a monitoring system like Nagios and based on the statistics from the monitoring system then you can trigger the script, but this script challenge could be a good learning experience!

Here is another great resource on how to use the Discord API and send notifications to your Discord Channel with a Bash script:

[How To Use Discord Webhooks to Get Notifications for Your Website Status on Ubuntu 18.04](#)

**Notice:** This content was initially posted on [DevDojo](#)

One of the first things that I would usually do in case I notice a high CPU usage on some of my Linux servers would be to check the process list with either `top` or `htop` and in case that I notice a lot of Apache or Nginx process I would quickly check my access logs to determine what has caused or is causing the CPU spike on my server or to figure out if anything malicious is going on.

Sometimes reading the logs could be quite intimidating as the log might be huge and going through it manually could take a lot of time. Also, the raw log format could be confusing for people with less experience.

Just like the previous chapter, this chapter is going to be a challenge! You need to write a short bash script that would summarize the whole access log for you without the need of installing any additional software.

This BASH script needs to parse and summarize your access logs and provide you with very useful information like:

- The 20 top pages with the most POST requests
- The 20 top pages with the most GET requests
- Top 20 IP addresses and their geo-location



I already have prepared a demo script which you could use as a reference. But I encourage you to try and write the script yourself first and only then take a look at my script!

In order to download the script, you can either clone the repository with the following command:

```
git clone  
https://github.com/bobbyiliev/quick\_access\_logs\_summary.git
```

Or run the following command which would download the script in your current directory:

```
wget  
https://raw.githubusercontent.com/bobbyiliev/quick\_access\_logs\_summary/master/spike\_check
```

The script does not make any changes to your system, it only reads the content of your access log and summarizes it for you, however, once you've downloaded the file, make sure to review the content yourself.

All that you have to do once the script has been downloaded is to make it executable and run it.

To do that run the following command to make the script executable:

```
chmod +x spike_check
```

Then run the script:

```
./spike_check /path/to/your/access_log
```

Make sure to change the path to the file with the actual path to your access log. For example if you are using Apache on an Ubuntu server, the exact command would look like this:

```
./spike_check /var/log/apache2/access.log
```

If you are using Nginx the exact command would be almost the same, but with the path to the Nginx access log:

```
./spike_check /var/log/nginx/access.log
```

Once you run the script, it might take a while depending on the size of the log.

The output that you would see should look like this:



Essentially what we can tell in this case is that we've received 16 POST requests to our `xmlrpc.php` file which is often used by attackers to try and exploit WordPress websites by using various username and password combinations.

In this specific case, this was not a huge brute force attack, but it gives us an early indication and we can take action to prevent a larger attack in the future.

We can also see that there were a couple of Russian IP addresses accessing our site, so in case that you do not expect any traffic from Russia, you might want to block those IP addresses as well.

This is an example of a simple BASH script that allows you to quickly summarize your access logs and determine if anything malicious is going on.

Of course, you might want to also manually go through the logs as well but it is a good challenge to try and automate this with Bash!

**Notice:** This content was initially posted on [DevDojo](#)

SSMTP is a tool that delivers emails from a computer or a server to a configured mail host.

SSMTP is not an email server itself and does not receive emails or manage a queue.

One of its primary uses is for forwarding automated email (like system alerts) off your machine and to an external email address.

You would need the following things in order to be able to complete this tutorial successfully:

- Access to an Ubuntu 18.04 server as a non-root user with sudo privileges and an active firewall installed on your server. To set these up, please refer to our [Initial Server Setup Guide for Ubuntu 18.04](#)
- An SMTP server along with SMTP username and password, this would also work with Gmail's SMTP server, or you could set up your own SMTP server by following the steps from this tutorial on [\[https://www.digitalocean.com/community/tutorials/how-to-install-and-configure-postfix-as-a-send-only-smtp-server-on-ubuntu-16-04\]](https://www.digitalocean.com/community/tutorials/how-to-install-and-configure-postfix-as-a-send-only-smtp-server-on-ubuntu-16-04)(How to Install and Configure Postfix as a Send-Only SMTP Server on Ubuntu 16.04)

In order to install SSMTP, you'll need to first update your apt cache with:

```
sudo apt update
```

Then run the following command to install SSMTP:

```
sudo apt install ssmtp
```

Another thing that you would need to install is **mailutils**, to do that run the following command:

```
sudo apt install mailutils
```

Now that you have **ssmtp** installed, in order to configure it to use your SMTP server when sending emails, you need to edit the SSMTP configuration file.

Using your favourite text editor open the **/etc/ssmtp/ssmtp.conf** file:

```
sudo nano /etc/ssmtp/ssmtp.conf
```

You need to include the your SMTP configuration:

```
root=postmaster
mailhub=<^>your_smtp_host.com<^>:587
hostname=<^>your_hostname<^>
AuthUser=<^>your_gmail_username@your_smtp_host.com<^>
AuthPass=<^>your_gmail_password<^>
FromLineOverride=YES
UseSTARTTLS=YES
```

Save the file and exit.



Once your configuration is done, in order to send an email just run the following command:

```
echo "<^>Here add your email body<^>" | mail -s "<^>Here  
specify your email subject<^>"  
<^>your_receipient_email@yourdomain.com<^>
```

You can run this directly in your terminal or include it in your bash scripts.

If you need to send files as attachments, you can use **mpack**.

To install **mpack** run the following command:

```
sudo apt install mpack
```

Next, in order to send an email with a file attached, run the following command.

```
mpack -s "<^>Your Subject here<^>" your_file.zip  
<^>your_receipient_email@yourdomain.com<^>
```

The above command would send an email to  
<^>your\_receipient\_email@yourdomain.com<^> with the  
<^>your\_file.zip<^> attached.

SSMTP is a great and reliable way to implement SMTP email functionality directly in bash scripts.

For more information about SSMTP I would recommend checking the official documentation [here](#).

**Notice:** This content was initially posted on the [DigitalOcean community forum](#).

It's not uncommon situation where you will need to generate a random password that you can use for any software installation or when you sign-up to any website.

There are a lot of options in order to achieve this. You can use a password manager/vault where you often have the option to randomly generate a password or to use a website that can generate the password on your behalf.

You can also use Bash in your terminal (command-line) to generate a password that you can quickly use. There are a lot of ways to achieve that and I will make sure to cover few of them and will leave up to you to choose which option is most suitable with your needs.

**This script is intended to practice your bash scripting skills. You can have fun while doing simple projects with BASH, but security is not a joke, so please make sure you do not save your passwords in plain text in a local file or write them down by hand on a piece of paper.**

**I will highly recommend everyone to use secure and trusted providers to generate and save the passwords.**

Let me first do a quick summary of what our script is going to do.:

1. We will have to option to choose the password characters length when the script is executed.
2. The script will then generate 5 random passwords with the length that was specified in step 1

You would need a bash terminal and a text editor. You can use any text editor like vi, vim, nano or Visual Studio Code.

I'm running the script locally on my Linux laptop but if you're using Windows PC you can ssh to any server of your choice and execute the script there.

Although the script is pretty simple, having some basic BASH scripting knowledge will help you to better understand the script and how it's working.

One of the great benefits of Linux is that you can do a lot of things using different methods. When it comes to generating a random string of characters it's not different as well.

You can use several commands in order to generate a random string of characters. I will cover few of them and will provide some examples.

- Using the `date` command. The date command will output the current date and time. However we also further manipulate the output in order to use it as randomly generated password. We can hash the date using md5, sha or just run it through base64. These are few examples:

```
date | md5sum
94cb1cdecfed0699e2d98acd9a7b8f6d -
```

using sha256sum:

```
date | sha256sum
30a0c6091e194c8c7785f0d7bb6e1eac9b76c0528f02213d1b6a5fbcc76cef
f4 -
```

using base64:

```
date | base64
0YHQsSDRj9C90YMgMzAgMTk6NTE6NDggRUVUIDIwMjEK
```

- We can also use openssl in order to generate pseudo-random bytes and run the output through base64. An example output will be:

```
openssl rand -base64 10
9+soM9bt8mhdcw==
```

Keep in mind that openssl might not be installed on your system so it's likely that you will need to install it first in order to use it.



- The most preferred way is to use the pseudorandom number generator - `/dev/urandom` since it is intended for most cryptographic purposes. We would also need to manipulate the output using `tr` in order to translate it. An example command is:

```
tr -cd '[:alnum:]' < /dev/urandom | fold -w10 | head -n 1
```

With this command we take the output from `/dev/urandom` and translate it with `tr` while using all letters and digits and print the desired number of characters.

First we begin the script with the shebang. We use it to tell the operating system which interpreter to use to parse the rest of the file.

```
#!/bin/bash
```

We can then continue and ask the user for some input. In this case we would like to know how many characters the password needs to be:

```
# Ask user for password length
clear
printf "\n"
read -p "How many characters you would like the password to
have? " pass_length
printf "\n"
```

Generate the passwords and then print it so the user can use it.

```
# This is where the magic happens!
# Generate a list of 10 strings and cut it to the desired
value provided from the user

for i in {1..10}; do (tr -cd '[:a1num:]' < /dev/urandom | fold
-w${pass_lenght} | head -n 1); done

# Print the strings
printf "$pass_output\n"
printf "Goodbye, ${USER}\n"
```

```
#!/bin/bash
#=====
# Password generator with login option
#=====

# Ask user for the string length
clear
printf "\n"
read -p "How many characters you would like the password to
have? " pass_lenght
printf "\n"

# This is where the magic happens!
# Generate a list of 10 strings and cut it to the desired
value provided from the user

for i in {1..10}; do (tr -cd '[:a\lnum:]' < /dev/urandom | fold
-w${pass_lenght} | head -n 1); done

# Print the strings
printf "$pass_output\n"
printf "Goodbye, ${USER}\n"
```

This is pretty much how you can use simple bash script to generate random passwords.

**:warning: As already mentioned, please make sure to use strong passwords in order to make sure your account is protected. Also whenever is possible use 2 factor authentication as this will provide additional layer of security for your account.**

While the script is working fine, it expects that the user will provide the requested input. In order to prevent any issues you would need to do some more advance checks on the user input in order to make sure the script will continue to work fine even if the provided input does not match our needs.

[Alex Georgiev](#)

A Linux superuser must have a good knowledge of pipes and redirection in Bash. It is an essential component of the system and is often helpful in the field of Linux System Administration.

When you run a command like `ls`, `cat`, etc, you get some output on the terminal. If you write a wrong command, pass a wrong flag or a wrong command-line argument, you get error output on the terminal. In both the cases, you are given some text. It may seem like "just text" to you, but the system treats this text differently. This identifier is known as a File Descriptor (fd).

In Linux, there are 3 File Descriptors, **STDIN** (0); **STDOUT** (1) and **STDERR** (2).

- **STDIN** (fd: 0): Manages the input in the terminal.
- **STDOUT** (fd: 1): Manages the output text in the terminal.
- **STDERR** (fd: 2): Manages the error text in the terminal.

Both *pipes* and *redidertions* redirect streams (**file descriptor**) of process being executed. The main diffrence is that *redirections* deal with **files stream**, sending the output stream to a file or sending the content of a given file to the input stream of the process.

On the otherhand a pipe connects two commands by sending the output stream of the first one to the input stream of the second one. without any redidertions specified.





When you enter some input text for a command that asks for it, you are actually entering the text to the **STDIN** file descriptor. Run the `cat` command without any command-line arguments. It may seem that the process has paused but in fact it's `cat` asking for **STDIN**. `cat` is a simple program and will print the text passed to **STDIN**. However, you can extend the use case by redirecting the input to the commands that take **STDIN**.

Example with `cat`:

```
cat << EOF
Hello World!
How are you?
EOF
```

This will simply print the provided text on the terminal screen:

```
Hello World!
How are you?
```

The same can be done with other commands that take input via STDIN. Like, `wc`:

```
wc -l << EOF
Hello World!
How are you?
EOF
```

The `-l` flag with `wc` counts the number of lines. This block of bash code will print the number of lines to the terminal screen:

```
2
```

The normal non-error text on your terminal screen is printed via the **STDOUT** file descriptor. The **STDOUT** of a command can be redirected into a file, in such a way that the output of the command is written to a file instead of being printed on the terminal screen. This is done simply with the help of `>` and `>>` operators.

Example:

```
echo "Hello World!" > file.txt
```

The following command will not print "Hello World" on the terminal screen, it will instead create a file called `file.txt` and will write the "Hello World" string to it. This can be verified by running the `cat` command on the `file.txt` file.

```
cat file.txt
```

However, everytime you redirect the **STDOUT** of any command multiple times to the same file, it will remove the existing contents of the file to write the new ones.

Example:

```
echo "Hello World!" > file.txt  
echo "How are you?" > file.txt
```

On running `cat` on `file.txt` file:

```
cat file.txt
```

You will only get the "How are you?" string printed.

```
How are you?
```

This is because the "Hello World" string has been overwritten. This behaviour can be

avoided using the `>>` operator.

The above example can be written as:

```
echo "Hello World!" > file.txt  
echo "How are you?" >> file.txt
```

On running `cat` on the `file.txt` file, you will get the desired result.

```
Hello World!  
How are you?
```

Alternatively, the redirection operator for **STDOUT** can also be written as `1>`. Like,

```
echo "Hello World!" 1> file.txt
```

The error text on the terminal screen is printed via the **STDERR** of the the command. For example:

```
ls --hello
```

would give an error messages. This error message is the **STDERR** of the command.

**STDERR** can be redirected using the **2>** operator.

```
ls --hello 2> error.txt
```

This command will redirect the error message to the **error.txt** file and write it to it. This can be verified by running the **cat** command on the **error.txt** file.

You can also use the **2>>** operator for **STDERR** just like you used **>>** for **STDOUT**.

Error messages in Bash Scripts can be undesirable sometimes. You can choose to ignore them by redirecting the error message to the **/dev/null** file. **/dev/null** is pseudo-device that takes in text and then immediately discards it.

The above example can be written follows to ignore the error text completely:

```
ls --hello 2> /dev/null
```

Of course, you can redirect both **STDOUT** and **STDERR** for the same command or script.

```
./install_package.sh > output.txt 2> error.txt
```

Both of them can be redirected to the same file as well.

```
./install_package.sh > file.txt 2> file.txt
```

There is also a shorter way to do this.

```
./install_package.sh > file.txt 2>&1
```

So far we have seen how to redirect the **STDOUT**, **STDIN** and **STDERR** to and from a file. To concatenate the output of program (*command*) as the input of another program (*command*) you can use a vertical bar `|`.

Example:

```
ls | grep ".txt"
```

This command will list the files in the current directory and pass output to *grep* command which then filter the output to only show the files that contain the string ".txt".

Syntax:

```
[time [-p]] [!] command1 [ | or |& command2 ] ...
```

You can also build arbitrary chains of commands by piping them together to achieve a powerful result.

This example create a listing of every user which owns a file in a given directory as well as how many files and directories they own:

```
ls -l /projects/bash_scripts | tail -n +2 | sed 's/\s\s*/ /g'  
| cut -d ' ' -f 3 | sort | uniq -c
```

Output:



8 anne  
34 harry  
37 tina  
18 ryan

The symbol `<<` can be used to create a temporary file [heredoc] and redirect from it at the command line.

```
COMMAND << EOF
    ContentOfDocument
    ...
    ...
EOF
```

Note here that `EOF` represents the delimiter (end of file) of the heredoc. In fact, we can use any alphanumeric word in its place to signify the start and the end of the file. For instance, this is a valid heredoc:

```
cat << randomword1
    This script will print these lines on the terminal.
    Note that cat can read from standard input. Using this
heredoc, we can
    create a temporary file with these lines as it's
content and pipe that
    into cat.
randomword1
```

Effectively it will appear as if the contents of the heredoc are piped into the command. This can make the script very clean if multiple lines need to be piped into a program.

Further, we can attach more pipes as shown:



```
cat << randomword1 | wc
    This script will print these lines on the terminal.
    Note that cat can read from standard input. Using this
heredoc, we can
    create a temporary file with these lines as it's
content and pipe that
    into cat.
randomword1
```

Also, pre-defined variables can be used inside the heredocs.

Herestrings are quite similar to heredocs but use `<<<`. These are used for single line strings that have to be piped into some program. This looks cleaner than heredocs as we don't have to specify the delimiter.

```
wc <<<"this is an easy way of passing strings to the stdin of  
a program (here wc)"
```

Just like heredocs, herestrings can contain variables.

**Operator Description**

>	Save output to a file
>>	Append output to a file
<	Read input from a file
2>	Redirect error messages
	Send the output from one program as input to another program
<<	Pipe multiple lines into a program cleanly
<<<	Pipe a single line into a program cleanly

Congratulations! You have just completed the Bash basics guide!

If you found this useful, be sure to star the project on [GitHub](#)!

If you have any suggestions for improvements, make sure to contribute pull requests or open issues.

In this introduction to Bash scripting book, we just covered the basics, but you still have enough under your belt to start wringing some awesome scripts and automating daily tasks!

As a next step try writing your own script and share it with the world! This is the best way to learn any new programming or scripting language!

In case that this book inspired you to write some cool Bash scripts, make sure to tweet about it and tag [@bobbyiliev](#) so that we could check it out!

Congrats again on completing this book!