Если вы видите что-то необычное, просто сообщите мне.

Готовые скрипты

- Ansible
  - Wait for connection if not available, timeout 600 seconds
- bash
  - Удаление ноды через cron
  - Template
  - Шпаргалка
- Установка openvpn(шаблончик превратить в скрипт)
- Пройтись по папкам и выполнить в каждой из них команду
- Шпаргалка go

Ansible

Ansible

Wait for connection if not available, timeout 600 seconds

```
- hosts: vivumlab
  gather_facts: no
  pre_tasks:
    - name: Wait for connection if not available, timeout 600 seconds
      wait_for_connection:
        timeout: 600
        delay: 0
      when: vlab_connection_wait == true

  tasks:
    - name: Restart services
      become: yes
      systemd:
        name: "{{ service_item }}"
        state: restarted
        enabled: true
      when: query | bool
      loop: "{{ services | flatten(1) }}"
      loop_control:
        loop_var: service_item
      ignore_errors: yes
      vars:
        query: "{{ vars[service_item]['enable'] }}"
```

bash

bash

Удаление ноды через cron

В крон помещаем запись вида:

```
*/5 * * * * /root/deletenode.sh
```

Каждые пять минут запускаем скрип `/root/deletenode.sh` следующего содержания:

```
#!/bin/bash
LOGGERTAG=nodeDeletion
kubectl  --kubeconfig /etc/kubernetes/admin.conf get nodes | grep NotReady | awk '{print $1;}'
> last
logger -t $LOGGERTAG "Deleting nodes:"
RESULT=$(kubectl --kubeconfig /etc/kubernetes/admin.conf delete node $(grep -f last old) 2>&1)
logger -t $LOGGERTAG "$RESULT"
mv last old
```

Результат работы скрипта будет записываться в syslog, чтобы посмотреть его, нужно выполнить следующую команду:

```
sudo journalctl -t nodeDeletion
```

bash

Template

Things Use `bash`. Using `zsh` or `fish` or any other, will make it hard for others to understand / collaborate. Among all shells, bash strikes a good balance between portability and DX.

Just make the first line be #!/usr/bin/env bash, even if you don’t give executable permission to the script file.

Use the .sh (or .bash) extension for your file. It may be fancy to not have an extension for your script, but unless your case explicitly depends on it, you’re probably just trying to do clever stuff. Clever stuff are hard to understand.

Use set -o errexit at the start of your script.

So that when a command fails, bash exits instead of continuing with the rest of the script. Prefer to use set -o nounset. You may have a good excuse to not do this, but, my opinion, it’s best to always set it.

This will make the script fail, when accessing an unset variable. Saves from horrible unintended consequences, with typos in variable names. When you want to access a variable that may or may not have been set, use "${VARNAME-}" instead of "$VARNAME", and you’re good. Use set -o pipefail. Again, you may have good reasons to not do this, but I’d recommend to always set it.

This will ensure that a pipeline command is treated as failed, even if one command in the pipeline fails. Use set -o xtrace, with a check on $TRACE env variable.

For copy-paste: if [["${TRACE-0}" == "1"]]; then set -o xtrace; fi. This helps in debugging your scripts, a lot. Like, really lot. People can now enable debug mode, by running your script as TRACE=1 ./script.sh instead of ./script.sh. Use [[ ]] for conditions in if / while statements, instead of [ ] or test.

[[ ]] is a bash builtin keyword, and is more powerful than [ ] or test. Always quote variable accesses with double-quotes.

One place where it’s okay not to is on the left-hand-side of an [[ ]] condition. But even there I’d recommend quoting. When you need the unquoted behaviour, using bash arrays will likely serve you much better. Use local variables in functions.

Accept multiple ways that users can ask for help and respond in kind.

Check if the first arg is -h or --help or help or just h or even -help, and in all these cases, print help text and exit. Please. For the sake of your future-self. When printing error messages, please redirect to stderr.

Use echo 'Something unexpected happened' >&2 for this. Use long options, where possible (like --silent instead of -s). These serve to document your commands explicitly.

Note though, that commands shipped on some systems like macOS don’t always have long options. If appropriate, change to the script’s directory close to the start of the script.

And it’s usually always appropriate. Use cd "$(dirname "$0")", which works in most cases. Use shellcheck. Heed its warnings

Template

```
#!/usr/bin/env bash

set -o errexit
set -o nounset
set -o pipefail

if [[ "${TRACE-0}" == "1" ]]; then
    set -o xtrace
fi

if [[ "${1-}" =~ ^-*h(elp)?$ ]]; then
    echo 'Usage: ./script.sh arg-one arg-two
This is an awesome bash script to make your life better.
'
    exit
fi
cd "$(dirname "$0")"
main() {
```

```
    echo do awesome stuff
}
main "$@"
```

Conclusion

I try to follow these rules in my scripts, and they’re known to have made at least my own life better. I’m still not consistent though, unfortunately, in following my own rules. So perhaps writing them down this way will help me improve there as well.

bash

Шпаргалка

```
#!/bin/bash
###############################################################################
# SHORTCUTS and HISTORY
###############################################################################

CTRL+A  # move to beginning of line
CTRL+B  # moves backward one character
CTRL+C  # halts the current command
CTRL+D  # deletes one character backward or logs out of current session, similar to exit
CTRL+E  # moves to end of line
CTRL+F  # moves forward one character
CTRL+G  # aborts the current editing command and ring the terminal bell
CTRL+H  # deletes one character under cursor (same as DELETE)
CTRL+J  # same as RETURN
CTRL+K  # deletes (kill) forward to end of line
CTRL+L  # clears screen and redisplay the line
CTRL+M  # same as RETURN
CTRL+N  # next line in command history
CTRL+O  # same as RETURN, then displays next line in history file
CTRL+P  # previous line in command history
CTRL+Q  # resumes suspended shell output
CTRL+R  # searches backward
CTRL+S  # searches forward or suspends shell output
CTRL+T  # transposes two characters
CTRL+U  # kills backward from point to the beginning of line
CTRL+V  # makes the next character typed verbatim
CTRL+W  # kills the word behind the cursor
CTRL+X  # lists the possible filename completions of the current word
CTRL+Y  # retrieves (yank) last item killed
CTRL+Z  # stops the current command, resume with fg in the foreground or bg in the background

ALT+B   # moves backward one word
ALT+D   # deletes next word
```

```
ALT+F   # moves forward one word
ALT+H   # deletes one character backward
ALT+T   # transposes two words
ALT+.   # pastes last word from the last command. Pressing it repeatedly traverses through
command history.
ALT+U   # capitalizes every character from the current cursor position to the end of the word
ALT+L   # uncapitalizes every character from the current cursor position to the end of the
word
ALT+C   # capitalizes the letter under the cursor. The cursor then moves to the end of the
word.
ALT+R   # reverts any changes to a command you've pulled from your history if you've edited
it.
ALT+?   # list possible completions to what is typed
ALT+^   # expand line to most recent match from history

CTRL+X then (   # start recording a keyboard macro
CTRL+X then )   # finish recording keyboard macro
CTRL+X then E   # recall last recorded keyboard macro
CTRL+X then CTRL+E   # invoke text editor (specified by $EDITOR) on current command line then
execute resultes as shell commands
CTRL+A then D  # logout from screen but don't kill it, if any command exist, it will continue

BACKSPACE  # deletes one character backward
DELETE     # deletes one character under cursor

history   # shows command line history
!!        # repeats the last command
!<n>      # refers to command line 'n'
!<string> # refers to command starting with 'string'
esc :wq   # exits and saves script

exit      # logs out of current session


################################################################################
# BASH BASICS
################################################################################

env                 # displays all environment variables
```

```
echo $SHELL         # displays the shell you're using
echo $BASH_VERSION  # displays bash version

bash                # if you want to use bash (type exit to go back to your previously opened
shell)
whereis bash        # locates the binary, source and manual-page for a command
which bash          # finds out which program is executed as 'bash' (default: /bin/bash, can
change across environments)

clear               # clears content on window (hide displayed lines)


#############################################################################
# FILE COMMANDS
#############################################################################


ls                          # lists your files in current directory, ls <dir> to print files
in a specific directory
ls -l                       # lists your files in 'long format', which contains the exact
size of the file, who owns the file and who has the right to look at it, and when it was last
modified
ls -a                       # lists all files in 'long format', including hidden files (name
beginning with '.')
ln -s <filename> <link>     # creates symbolic link to file
readlink <filename>         # shows where a symbolic links points to
tree                        # show directories and subdirectories in easilly readable file
tree
mc                          # terminal file explorer (alternative to ncdu)
touch <filename>            # creates or updates (edit) your file
mktemp -t <filename>        # make a temp file in /tmp/ which is deleted at next boot (-d to
make directory)
cat <filename>              # displays file raw content (will not be interpreted)
cat -n <filename>           # shows number of lines
nl <file.sh>                # shows number of lines in file
cat filename1 > filename2   # Copy filename1 to filename2
cat filename1 >> filename2  # merge two files texts together
any_command > <filename>    # '>' is used to perform redirections, it will set any_command's
stdout to file instead of "real stdout" (generally /dev/stdout)
more <filename>             # shows the first part of a file (move with space and type q to
```

```
quit)
head <filename>               # outputs the first lines of file (default: 10 lines)
tail <filename>               # outputs the last lines of file (useful with -f option)
(default: 10 lines)
vim <filename>                # opens a file in VIM (VI iMproved) text editor, will create it
if it doesn't exist
mv <filename1> <dest>         # moves a file to destination, behavior will change based on
'dest' type (dir: file is placed into dir; file: file will replace dest (tip: useful for
renaming))
cp <filename1> <dest>         # copies a file
rm <filename>                 # removes a file
find . -name <name> <type>    # searches for a file or a directory in the current directory
and all its sub-directories by its name
diff <filename1> <filename2>  # compares files, and shows where they differ
wc <filename>                 # tells you how many lines, words and characters there are in a
file. Use -lwc (lines, word, character) to ouput only 1 of those informations
sort <filename>               # sorts the contents of a text file line by line in alphabetical
order, use -n for numeric sort and -r for reversing order.
sort -t -k <filename>         # sorts the contents on specific sort key field starting from 1,
using the field separator t.
rev                           # reverse string characters (hello becomes olleh)
chmod -options <filename>     # lets you change the read, write, and execute permissions on
your files (more infos: SUID, GUID)
gzip <filename>               # compresses files using gzip algorithm
gunzip <filename>             # uncompresses files compressed by gzip
gzcat <filename>              # lets you look at gzipped file without actually having to
gunzip it
lpr <filename>                # prints the file
lpq                           # checks out the printer queue
lprm <jobnumber>              # removes something from the printer queue
genscript                     # converts plain text files into postscript for printing and
gives you some options for formatting
dvips <filename>              # prints .dvi files (i.e. files produced by LaTeX)
grep <pattern> <filenames>    # looks for the string in the files
grep -r <pattern> <dir>       # search recursively for pattern in directory
head -n file_name | tail +n   # Print nth line from file.
head -y lines.txt | tail +x   # want to display all the lines from x to y. This includes the
xth and yth lines.

sed 's/<pattern>/<replacement>/g' <filename> # replace pattern in file with replacement value
```

```
to std output the character after s (/) is the delimeter
sed -i 's/<pattern>/<replacement>/g' <filename> # replace pattern in file with replacement
value in place
echo "this" | sed 's/is/at/g' # replace pattern from input stream with replacement value

###############################################################################
# DIRECTORY COMMANDS
###############################################################################


mkdir <dirname>               # makes a new directory
rmdir <dirname>               # remove an empty directory
rmdir -rf <dirname>           # remove a non-empty directory
mv <dir1> <dir2>              # rename a directory from <dir1> to <dir2>
cd                            # changes to home
cd ..                         # changes to the parent directory
cd <dirname>                  # changes directory
cp -r <dir1> <dir2>           # copy <dir1> into <dir2> including sub-directories
pwd                           # tells you where you currently are
cd ~                          # changes to home.
cd -                          # changes to previous working directory

###############################################################################
# SSH, SYSTEM INFO & NETWORK COMMANDS
###############################################################################


ssh user@host             # connects to host as user
ssh -p <port> user@host   # connects to host on specified port as user
ssh-copy-id user@host     # adds your ssh key to host for user to enable a keyed or
passwordless login

whoami                    # returns your username
su <user>                 # switch to a different user
su -                      # switch to root, likely needs to be sudo su -
sudo <command>            # execute command as the root user
passwd                    # lets you change your password
quota -v                  # shows what your disk quota is
date                      # shows the current date and time
cal                       # shows the month's calendar
```

```
uptime                  # shows current uptime
w                       # displays whois online
finger <user>           # displays information about user
uname -a                # shows kernel information
man <command>           # shows the manual for specified command
info <command>          # shows another documentation system for the specific command
help                    # shows documentation about built-in commands and functions
df                      # shows disk usage
du <filename>           # shows the disk usage of the files and directories in filename (du -
s give only a total)
resize2fs               # ext2/ext3/ext4 file system resizer
last <yourUsername>     # lists your last logins
ps -u yourusername      # lists your processes
kill <PID>              # kills the processes with the ID you gave
killall <processname>   # kill all processes with the name
top                     # displays your currently active processes
lsof                    # lists open files
bg                      # lists stopped or background jobs ; resume a stopped job in the
background
fg                      # brings the most recent job in the foreground
fg <job>                # brings job to the foreground

ping <host>             # pings host and outputs results
whois <domain>          # gets whois information for domain
dig <domain>            # gets DNS information for domain
dig -x <host>           # reverses lookup host
wget <file>             # downloads file
netstat                 # Print network connections, routing tables, interface statistics,
masquerade connections, and multicast memberships

time <command>          # report time consumed by command execution


###############################################################################
# VARIABLES
###############################################################################


varname=value               # defines a variable
varname=value command       # defines a variable to be in the environment of a particular
```

```
subprocess
echo $varname                # checks a variable's value
echo $$                      # prints process ID of the current shell
echo $!                      # prints process ID of the most recently invoked background job
echo $?                      # displays the exit status of the last command
read <varname>               # reads a string from the input and assigns it to a variable
read -p "prompt" <varname>   # same as above but outputs a prompt to ask user for value
column -t <filename>         # display info in pretty columns (often used with pipe)
let <varname> = <equation>   # performs mathematical calculation using operators like +, -, *,
/, %
export VARNAME=value         # defines an environment variable (will be available in
subprocesses)
export -f  <funcname>        # Exports function 'funcname'
export var1="var1 value"     # Export and assign in the same statement
export <varname>             # Copy Bash variable
declare -x <varname>         # Copy Bash variable

array[0]=valA                # how to define an array
array[1]=valB
array[2]=valC
array=([2]=valC [0]=valA [1]=valB)  # another way
array=(valA valB valC)              # and another

${array[i]}                  # displays array's value for this index. If no index is supplied,
array element 0 is assumed
${#array[i]}                 # to find out the length of any element in the array
${#array[@]}                 # to find out how many values there are in the array

declare -a                   # the variables are treated as arrays
declare -f                   # uses function names only
declare -F                   # displays function names without definitions
declare -i                   # the variables are treated as integers
declare -r                   # makes the variables read-only
declare -x                   # marks the variables for export via the environment
declare -l                   # uppercase values in the variable are converted to lowercase
declare -A                   # makes it an associative array

${varname:-word}             # if varname exists and isn't null, return its value; otherwise
return word
${varname:word}              # if varname exists and isn't null, return its value; otherwise
```

```
return word
${varname:=word}             # if varname exists and isn't null, return its value; otherwise
set it word and then return its value
${varname:?message}          # if varname exists and isn't null, return its value; otherwise
print varname, followed by message and abort the current command or script
${varname:+word}             # if varname exists and isn't null, return word; otherwise return
null
${varname:offset:length}     # performs substring expansion. It returns the substring of
$varname starting at offset and up to length characters

${variable#pattern}          # if the pattern matches the beginning of the variable's value,
delete the shortest part that matches and return the rest
${variable##pattern}         # if the pattern matches the beginning of the variable's value,
delete the longest part that matches and return the rest
${variable%pattern}          # if the pattern matches the end of the variable's value, delete
the shortest part that matches and return the rest
${variable%%pattern}         # if the pattern matches the end of the variable's value, delete
the longest part that matches and return the rest
${variable/pattern/string}   # the longest match to pattern in variable is replaced by string.
Only the first match is replaced
${variable//pattern/string}  # the longest match to pattern in variable is replaced by string.
All matches are replaced

${#varname}                  # returns the length of the value of the variable as a character
string

*(patternlist)               # matches zero or more occurrences of the given patterns
+(patternlist)               # matches one or more occurrences of the given patterns
?(patternlist)               # matches zero or one occurrence of the given patterns
@(patternlist)               # matches exactly one of the given patterns
!(patternlist)               # matches anything except one of the given patterns

$(UNIX command)              # command substitution: runs the command and returns standard
output

typeset -l <x>                 # makes variable local - <x> must be an interger

################################################################################
# FUNCTIONS
################################################################################
```

```
# The function refers to passed arguments by position (as if they were positional parameters),
that is, $1, $2, and so forth.
# $@ is equal to "$1" "$2"... "$N", where N is the number of positional parameters. $# holds
the number of positional parameters.


function functname() {
  shell commands
}

unset -f functname  # deletes a function definition
declare -f          # displays all defined functions in your login session


################################################################################
# FLOW CONTROLS
################################################################################


statement1 && statement2  # and operator
statement1 || statement2  # or operator

-a                        # and operator inside a test conditional expression
-o                        # or operator inside a test conditional expression

# STRINGS

str1 == str2               # str1 matches str2
str1 != str2               # str1 does not match str2
str1 < str2                # str1 is less than str2 (alphabetically)
str1 > str2                # str1 is greater than str2 (alphabetically)
str1 \> str2               # str1 is sorted after str2
str1 \< str2               # str1 is sorted before str2
-n str1                    # str1 is not null (has length greater than 0)
-z str1                    # str1 is null (has length 0)

# FILES
```

```
-a file                  # file exists or its compilation is successful
-d file                  # file exists and is a directory
-e file                  # file exists; same -a
-f file                  # file exists and is a regular file (i.e., not a directory or other
special type of file)
-r file                  # you have read permission
-s file                  # file exists and is not empty
-w file                  # your have write permission
-x file                  # you have execute permission on file, or directory search
permission if it is a directory
-N file                  # file was modified since it was last read
-O file                  # you own file
-G file                  # file's group ID matches yours (or one of yours, if you are in
multiple groups)
file1 -nt file2          # file1 is newer than file2
file1 -ot file2          # file1 is older than file2

# NUMBERS

-lt                      # less than
-le                      # less than or equal
-eq                      # equal
-ge                      # greater than or equal
-gt                      # greater than
-ne                      # not equal

if condition
then
  statements
[elif condition
  then statements...]
[else
  statements]
fi

for x in {1..10}
do
  statements
done
```

```
for name [in list]
do
  statements that can use $name
done

for (( initialisation ; ending condition ; update ))
do
  statements...
done

case expression in
  pattern1 )
    statements ;;
  pattern2 )
    statements ;;
esac

select name [in list]
do
  statements that can use $name
done

while condition; do
  statements
done

until condition; do
  statements
done

##############################################################################
# COMMAND-LINE PROCESSING CYCLE
##############################################################################


# The default order for command lookup is functions, followed by built-ins, with scripts and
executables last.
# There are three built-ins that you can use to override this order: `command`, `builtin` and
`enable`.
```

```
command  # removes alias and function lookup. Only built-ins and commands found in the search path are executed
builtin  # looks up only built-in commands, ignoring functions and commands found in PATH
enable   # enables and disables shell built-ins

eval     # takes arguments and run them through the command-line processing steps all over again


################################################################################
# INPUT/OUTPUT REDIRECTORS
################################################################################


cmd1|cmd2  # pipe; takes standard output of cmd1 as standard input to cmd2
< file     # takes standard input from file
> file     # directs standard output to file
>> file    # directs standard output to file; append to file if it already exists
>|file     # forces standard output to file even if noclobber is set
n>|file    # forces output to file from file descriptor n even if noclobber is set
<> file    # uses file as both standard input and standard output
n<>file    # uses file as both input and output for file descriptor n
n>file     # directs file descriptor n to file
n<file     # takes file descriptor n from file
n>>file    # directs file description n to file; append to file if it already exists
n>&        # duplicates standard output to file descriptor n
n<&        # duplicates standard input from file descriptor n
n>&m       # file descriptor n is made to be a copy of the output file descriptor
n<&m       # file descriptor n is made to be a copy of the input file descriptor
&>file     # directs standard output and standard error to file
<&-        # closes the standard input
>&-        # closes the standard output
n>&-       # closes the ouput from file descriptor n
n<&-       # closes the input from file descriptor n

|tee <file># output command to both terminal and a file (-a to append to file)


################################################################################
# PROCESS HANDLING
```

```
################################################################################


# To suspend a job, type CTRL+Z while it is running. You can also suspend a job with CTRL+Y.
# This is slightly different from CTRL+Z in that the process is only stopped when it attempts
to read input from terminal.
# Of course, to interrupt a job, type CTRL+C.

myCommand &  # runs job in the background and prompts back the shell

jobs         # lists all jobs (use with -l to see associated PID)

fg           # brings a background job into the foreground
fg %+        # brings most recently invoked background job
fg %-        # brings second most recently invoked background job
fg %N        # brings job number N
fg %string   # brings job whose command begins with string
fg %?string  # brings job whose command contains string

kill -l               # returns a list of all signals on the system, by name and number
kill PID              # terminates process with specified PID
kill -s SIGKILL 4500  # sends a signal to force or terminate the process
kill -15 913          # Ending PID 913 process with signal 15 (TERM)
kill %1               # Where %1 is the number of job as read from 'jobs' command.

ps           # prints a line of information about the current running login shell and any
processes running under it
ps -a        # selects all processes with a tty except session leaders

trap cmd sig1 sig2  # executes a command when a signal is received by the script
trap "" sig1 sig2   # ignores that signals
trap - sig1 sig2    # resets the action taken when the signal is received to the default

disown <PID|JID>    # removes the process from the list of jobs

wait                # waits until all background jobs have finished
sleep <number>      # wait # of seconds before continuing

pv                  # display progress bar for data handling commands. often used with pipe
like |pv
```

```
yes                 # give yes response everytime an input is requested from script/process


################################################################################
# TIPS & TRICKS
################################################################################


# set an alias
cd; nano .bash_profile
> alias gentlenode='ssh admin@gentlenode.com -p 3404'  # add your alias in .bash_profile

# to quickly go to a specific directory
cd; nano .bashrc
> shopt -s cdable_vars
> export websites="/Users/mac/Documents/websites"

source .bashrc
cd $websites


################################################################################
# DEBUGGING SHELL PROGRAMS
################################################################################


bash -n scriptname  # don't run commands; check for syntax errors only
set -o noexec       # alternative (set option in script)

bash -v scriptname  # echo commands before running them
set -o verbose      # alternative (set option in script)

bash -x scriptname  # echo commands after command-line processing
set -o xtrace       # alternative (set option in script)

trap 'echo $varname' EXIT  # useful when you want to print out the values of variables at the
point that your script exits

function errtrap {
  es=$?
```

```
  echo "ERROR line $1: Command exited with status $es."
}

trap 'errtrap $LINENO' ERR  # is run whenever a command in the surrounding script or function exits with non-zero status

function dbgtrap {
  echo "badvar is $badvar"
}

trap dbgtrap DEBUG  # causes the trap code to be executed before every statement in a function or script
# ...section of code in which the problem occurs...
trap - DEBUG  # turn off the DEBUG trap

function returntrap {
  echo "A return occurred"
}

trap returntrap RETURN  # is executed each time a shell function or a script executed with the . or source commands finishes executing

###############################################################################
# COLORS AND BACKGROUNDS
###############################################################################
# note: \e or \x1B also work instead of \033
# Reset
Color_Off='\033[0m' # Text Reset

# Regular Colors
Black='\033[0;30m'  # Black
Red='\033[0;31m'    # Red
Green='\033[0;32m'  # Green
Yellow='\033[0;33m' # Yellow
Blue='\033[0;34m'   # Blue
Purple='\033[0;35m' # Purple
Cyan='\033[0;36m'   # Cyan
White='\033[0;97m'  # White

# Additional colors
```

```
LGrey='\033[0;37m'  # Light Gray
DGrey='\033[0;90m'  # Dark Gray
LRed='\033[0;91m'   # Light Red
LGreen='\033[0;92m' # Light Green
LYellow='\033[0;93m'# Light Yellow
LBlue='\033[0;94m'  # Light Blue
LPurple='\033[0;95m'# Light Purple
LCyan='\033[0;96m'  # Light Cyan


# Bold
BBlack='\033[1;30m' # Black
BRed='\033[1;31m'   # Red
BGreen='\033[1;32m' # Green
BYellow='\033[1;33m'# Yellow
BBlue='\033[1;34m'  # Blue
BPurple='\033[1;35m'# Purple
BCyan='\033[1;36m'  # Cyan
BWhite='\033[1;37m' # White

# Underline
UBlack='\033[4;30m' # Black
URed='\033[4;31m'   # Red
UGreen='\033[4;32m' # Green
UYellow='\033[4;33m'# Yellow
UBlue='\033[4;34m'  # Blue
UPurple='\033[4;35m'# Purple
UCyan='\033[4;36m'  # Cyan
UWhite='\033[4;37m' # White

# Background
On_Black='\033[40m' # Black
On_Red='\033[41m'   # Red
On_Green='\033[42m' # Green
On_Yellow='\033[43m'# Yellow
On_Blue='\033[44m'  # Blue
On_Purple='\033[45m'# Purple
On_Cyan='\033[46m'  # Cyan
On_White='\033[47m' # White
```

```
# Example of usage
echo -e "${Green}This is GREEN text${Color_Off} and normal text"
echo -e "${Red}${On_White}This is Red test on White background${Color_Off}"
# option -e is mandatory, it enable interpretation of backslash escapes
printf "${Red} This is red \n"
```

Установка openvpn(шаблончик превратить в скрипт)

```
sudo mkdir /etc/openvpn/easy-rsa
sudo rm /etc/openvpn/easy-rsa/ -r
sudo mkdir /etc/openvpn/easy-rsa
sudo ./easyrsa init-pki
sudo ./easyrsa build-ca
sudo ./easyrsa gen-dh
sudo openvpn --genkey --secret /etc/openvpn/easy-rsa/pki/ta.key
sudo ./easyrsa gen-crl
sudo ./easyrsa build-server-full server nopass
sudo cp ./pki/ca.crt /etc/openvpn/ca.crt
sudo cp ./pki/dh.pem /etc/openvpn/dh.pem
sudo cp ./pki/crl.pem /etc/openvpn/crl.pem
sudo cp ./pki/ta.key /etc/openvpn/ta.key
sudo cp ./pki/issued/server.crt /etc/openvpn/server.crt
sudo cp ./pki/private/server.key /etc/openvpn/server.key
sudo vim /etc/openvpn/server.conf
cd ..
sudo openvpn /etc/openvpn/server.conf
sudo systemctl start openvpn@server
sudo sysctl -w net.ipv4.ip_forward=1
#заменить  eth0  на нужный интерфейс
sudo iptables -I FORWARD -i tun0 -o eth0 -j ACCEPT
sudo iptables -I FORWARD -i eth0 -o tun0 -j ACCEPT
sudo iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
cd easy-rsa/
sudo ./easyrsa build-client-full losst nopass
sudo mkdir -p /etc/openvpn/clients/losst
cd /etc/openvpn/clients/losst
sudo cp /etc/openvpn/easy-rsa/pki/ca.crt /etc/openvpn/clients/losst/
```

```
sudo cp /etc/openvpn/easy-rsa/pki/ta.key /etc/openvpn/clients/losst/
sudo cp /etc/openvpn/easy-rsa/pki/issued/losst.crt /etc/openvpn/clients/losst/
sudo cp /etc/openvpn/easy-rsa/pki/private/losst.key /etc/openvpn/clients/losst/
sudo cp /usr/share/doc/openvpn/examples/sample-config-files/client.conf ./losst.conf
ip -br a
sudo vi ./losst.conf
sudo cat losst.crt
cat /etc/openvpn/clients/losst/losst.conf
cat losst.conf
sudo cat ca.crt
sudo cat losst.key
sudo cat ta.key
sudo cat ca.crt
sudo cat losst.key
sudo cat ta.key
```

Пройтись по папкам и выполнить в каждой из них команду

```
#!/bin/bash

# Перебираем все папки в текущем каталоге
for dir in */; do
    # Переходим в каждую папку
    cd "$dir"

    # Проверяем наличие артефакта
    marker_file="processed_marker.txt"
    if [ -f "$marker_file" ]; then
        echo "Skipping folder $dir as it's already processed"
        cd ..
        continue
    fi

    # Выводим название папки
    echo "Contents of folder: $dir"

    # Выполняем команду
    tofu init && tofu apply && touch "$marker_file"

    # Возвращаемся обратно в исходную папку
    cd ..
done
```

Шпаргалка go

Golang

Summary

Hello World

```
package main

import "fmt"

func main() {
    fmt.Println("Hello Gophers!")
}
```

Return to Summary

Go CLI Commands

```
# Compile & Run code
$ go run [file.go]

# Compile
$ go build [file.go]
# Running compiled file
$ ./hello

# Test packages
$ go test [folder]

# Install packages/modules
$ go install [package]

# List installed packages/modules
$ go list

# Update packages/modules
$ go fix

# Format package sources
$ go fmt
```

```
# See package documentation
$ go doc [package]

# Add dependencies and install
$ go get [module]

# See Go environment variables
$ go env

# See version
$ go version
```

Return to Summary

Go Modules

- Go projects are called **modules**
- Each module has multiple **packages**
- Each package should has a scoped functionality. Packages talk to each other to compose the code
- A module needs at least one package, the **main**
- The package main needs a entry function called **main**

```
# Create Module
$ go mod init [name]
```

Tip: By convention, modules names has the follow structure:

domain.com/user/module/package

Example: github.com/spf13/cobra

Return to Summary

Basic Types

Type	Set of Values	Values
bool	boolean	true/false
string	array of characters	needs to be inside ""
int	integers	32 or 64 bit integer
int8	8-bit integers	[-128, 128]
int16	16-bit integers	[-32768, 32767]
int32	32-bit integers	[-2147483648, 2147483647]
int64	64-bit integers	[-9223372036854775808, 9223372036854775807]
uint8	8-bit unsigned integers	[0, 255]
uint16	16-bit unsigned integers	[0, 65535]
uint32	32-bit unsigned integers	[0, 4294967295]
uint64	64-bit unsigned integers	[0, 18446744073709551615]
float32	32-bit float	
float64	64-bit float	
complex64	32-bit float with real and imaginary parts	
complex128	64-bit float with real and imaginary parts	
byte	sets of bits	alias for uint8
rune	Unicode characters	alias for int32

Return to Summary

Variables

```
// Declaration
var value int

// Initialization
value = 10

// Declaration + Initialization + Type inference
var isActive = true

// Short declaration (only inside functions)
text := "Hello"

// Multi declaration
var i, j, k = 1, 2, 3

// Variable not initialized = Zero values
// Numeric: 0
// Boolean: false
// String: ""
// Special value: nil (same as null)

var number int // 0
var text string // ""
var boolean bool // false

// Type conversions
// T(v) converts v to type T

i := 1.234 // float
int(i) // 1

// Constants
const pi = 3.1415
```

Operators

Return to Summary

Arithmetic Operators

Symbol	Operation	Valid Types
`+`	Sum	integers, floats, complex values, strings
`-`	Difference	integers, floats, complex values
`*`	Product	integers, floats, complex values
`/`	Quotient	integers, floats, complex values
`%`	Remainder	integers
`&`	Bitwise AND	integers
`` ` ``	`` ` ``	Bitwise OR
`^`	Bitwise XOR	integers
`&^`	Bit clear (AND NOT)	integers
`<<`	Left shift	integer << unsigned integer
`>>`	Right shift	integer >> unsigned integer

Comparison Operators

Symbol	Operation
`==`	Equal
`!=`	Not equal
`<`	Less
`<=`	Less or equal
`>`	Greater
`>=`	Greater or equal

Logical Operators

Symbol	Operation
`&&`	Conditional AND
`` ` ``	
`!`	NOT

Return to Summary

Conditional Statements

```
// If / Else
i := 1

if i > 0 {
    // Condition is True! i is greater than zero
} else {
    // Condition is False! i is lower or equal to zero
}

// Else if
i := 1

if i > 0 {
    // Condition is True! i is greater than zero
} else if i > 0 && i < 2 {
    // Condition is True! i greater than zero and lower than two
} else if i > 1 && i < 4 {
    // Condition is True! i greater than one and lower than four
} else {
    // None of the above conditions is True, so it falls here
}

// If with short statements
i := 2.567

if j := int(i); j == 2 {
    // Condition is True! j, the integer value of i, is equal to two
} else {
    // Condition is False! j, the integer value of i, is not equal to two
}

// Switch
text := 'hey'
```

```
switch text {
    case 'hey':
        // 'Hello!'
    case 'bye':
        // 'Byee'
    default:
        // 'Ok'
}

// Switch without condition
value := 5

switch {
    case value < 2:
        // 'Hello!'
    case value >= 2 && value < 6:
        // 'Byee'
    default:
        // 'Ok'
}
```

Return to Summary

Loops

```
// Golang only has the for loop
for i := 0; i < 10; i++ {
    // i
}

// The first and third parameters are ommitable
// For as a while
i := 0;

for i < 10 {
    i++
```

```
}

// Forever loop
for {

}
```

Return to Summary

Arrays

```
// Declaration with specified size
var array [3]string
array[0] = "Hello"
array[1] = "Golang"
array[2] = "World"

// Declaration and Initialization
values := [5]int{1, 2, 3, 4, 5}

// Slices: A subarray that acts as a reference of an array
// Determining min and max
values[1:3] // {2, 3, 4}

// Determining only max will use min = 0
values[:2] // {1, 2, 3}

// Determining only min will use max = last element
values[3:] // {3, 4}

// Length: number of elements that a slice contains
len(values) // 5

// Capacity: number of elements that a slice can contain
values = values[:1]
len(values) // 2
cap(values) // 5
```

```
// Slice literal
slice := []bool{true, true, false}

// make function: create a slice with length and capacity
slice := make([]int, 5, 6) // make(type, len, cap)

// Append new element to slice
slice := []int{ 1, 2 }
slice = append(slice, 3)
slice // { 1, 2, 3 }
slice = append(slice, 3, 2, 1)
slice // { 1, 2, 3, 3, 2, 1 }

// For range: iterate over a slice
slice := string["W", "o", "w"]

for i, value := range slice {
    i // 0, then 1, then 2
    value // "W", then "o", then "w"
}

// Skip index or value

for i := range slice {
    i // 0, then 1, then 2
}

for _, value := range slice {
   value // "W", then "o", then "w"
}
```

Return to Summary

Functions

```
// Functions acts as a scoped block of code
func sayHello() {
    // Hello World!
}
sayHello() // Hello World!

// Functions can take zero or more parameters, as so return zero or more parameters
func sum(x int, y int) int {
    return x + y
}
sum(3, 7) // 10

// Returned values can be named and be used inside the function
func doubleAndTriple(x int) (double, triple int) {
    double = x * 2
    triple = x * 3
    return
}
d, t := doubleAndTriple(5)
// d = 10
// t = 15

// Skipping one of the returned values
_, t := doubleAndTriple(3)
// t = 9

// Functions can defer commands. Deferred commands are
// ran in a stack order after the execution and
// returning of a function
var aux = 0

func switchValuesAndDouble(x, y int) {
    aux = x
    defer aux = 0 // cleaning variable to post use
    x = y * 2
    y = aux * 2
}

a, b = 2, 5
switchValuesAndDouble(2, 5)
```

```
// a = 10
// b = 4
// aux = 0

// Functions can be handled as values and be anonymous functions
func calc(fn func(int, int) int) int {
    return fn(2, 6)
}

func sum(x, y int) int {
    return x + y
}

func mult(x, y int) int {
    return x * y
}

calc(sum) // 8
calc(mult) // 12
calc(
    func(x, y int) int {
□□return x / y
    }
) // 3

// Function closures: a function that returns a function
// that remembers the original context
func calc() func(int) int {
    value := 0
    return func(x int) int {
        value += x
        return value
    }
}

calculator := calc()
calculator(3) // 3
calculator(45) // 48
calculator(12) // 60
```

Return to Summary

Structs

Structs are a way to arrange data in specific formats.

```
// Declaring a struct
type Person struct {
    Name string
    Age int
}

// Initializing
person := Person{"John", 34}
person.Name // "John"
person.Age // 34

person2 := Person{Age: 20}
person2.Name // ""
person2.Age // 20

person3 := Person{}
person3.Name // ""
person3.Age // 0
```

Return to Summary

Maps

Maps are data structures that holds values assigned to a key.

```
// Declaring a map
var cities map[string]string
```

```
// Initializing
cities = make(map[string]string)
cities // nil

// Insert
cities["NY"] = "EUA"

// Retrieve
newYork = cities["NY"]
newYork // "EUA"

// Delete
delete(cities, "NY")

// Check if a key is set
value, ok := cities["NY"]
ok // false
value // ""
```

Return to Summary

Pointers

Pointers are a direct reference to a memory address that some variable or value is being stored.

```
// Pointers has *T type
var value int
var pointer *int

// Point to a variable memory address with &
value = 3
pointer = &value

pointer // 3
pointer = 20
pointer // 20
pointer += 5
```

```
pointer // 25

// Pointers to structs can access the attributes
type Struct struct {
    X int
}

s := Struct{3}
pointer := &s

s.X // 3
```

Obs: Unlike C, Go doesn't have pointer arithmetics.

Return to Summary

Methods and Interfaces

Go doesn't have classes. But you can implement methods, interfaces and almost everything contained in OOP, but in what gophers call "Go Way"

```
type Dog struct {
    Name string
}

func (dog *Dog) bark() string {
    return dog.Name + " is barking!"
}

dog := Dog{"Rex"}
dog.bark() // Rex is barking!
```

Interfaces are implicitly implemented. You don't need to inform that your struct are correctly implementing a interface if it already has all methods with the same name of the interface. All structs implement the `interface{}` interface. This empty interface means the same as `any`.

```
// Car implements Vehicle interface
type Vehicle interface {
    Accelerate()
}

type Car struct {

}

func (car *Car) Accelerate() {
    return "Car is moving on ground"
}
```

Return to Summary

Errors

Go doesn't support `throw`, `try`, `catch` and other common error handling structures. Here, we use `error` package to build possible errors as a returning parameter in functions

```
import "errors"

// Function that contain a logic that can cause a possible exception flow
func firstLetter(text string) (string, error) {
    if len(text) < 1 {
        return nil, errors.New("Parameter text is empty")
    }
    return string(text[0]), nil
}

a, errorA := firstLetter("Wow")
a // "W"
errorA // nil

b, errorB := firstLetter("")
b // nil
errorB // Error("Parameter text is empty")
```

Return to Summary

Testing

Go has a built-in library to unit testing. In a separate file you insert tests for functionalities of a file and run `go test package` to run all tests of the actual package or `go test path` to run a specific test file.

```
// main.go
func Sum(x, y int) int {
    return x + y
}

// main_test.go
import (
    "testing"
    "reflect"
)

func TestSum(t *testing.T) {
    x, y := 2, 4
    expected := 2 + 4

    if !reflect.DeepEqual(Sum(x, y), expected) {
        t.Fatalf("Function Sum not working as expected")
    }
}
```

Return to Summary

Concurrency

One of the main parts that make Go attractive is its form to handle with concurrency. Different than parallelism, where tasks can be separated in many cores that the machine processor have, in

concurrency we have routines that are more lightweight than threads and can run asynchronously, with memory sharing and in a single core.

```
// Consider a common function, but that function can delay itself because some processing
func show(from string) {
	for i := 0; i < 3; i++ {
		fmt.Printf("%s : %d\n", from, i)
	}
}

// In a blocking way...
func main() {
	show("blocking1")
	show("blocking2")

	fmt.Println("done")
}
/*  blocking1: 0
    blocking1: 1
    blocking1: 2
    blocking2: 0
    blocking2: 1
    blocking2: 2
    done
*/

// Go routines are a function (either declared previously or anonymous) called with the
keyword go
func main() {
	go show("routine1")
	go show("routine2")

	go func() {
		fmt.Println("going")
	}()

	time.Sleep(time.Second)

	fmt.Println("done")
}
```

```
/*  Obs: The result will depends of what processes first
    routine2: 0
    routine2: 1
    routine2: 2
    going
    routine1: 0
    routine1: 1
    routine1: 2
    done
*/

// Routines can share data with channels
// Channels are queues that store data between multiple routines
msgs := make(chan string)

go func(channel chan string) {
    channel <- "ping"
}(msgs)

go func(channel chan string) {
    channel <- "pong"
}(msgs)

fmt.Println(<-msgs) // pong
fmt.Println(<-msgs) // ping

// Channels can be bufferized. Buffered channels will accept a limited number of values and
when someone try to put belong their limit, it will throw and error
numbers := make(chan int, 2)

msgs<-0
msgs<-1
msgs<-2

// fatal error: all goroutines are asleep - deadlock!

// Channels can be passed as parameter where the routine can only send or receive
numbers := make(chan int)
```

```
go func(sender chan<- int) {
    sender <- 10
}(numbers)

go func(receiver <-chan int) {
    fmt.Println(<-receiver) // 10
}(numbers)

time.Sleep(time.Second)

// When working with multiple channels, the select can provide a control to execute code
accordingly of what channel has bring a message
c1 := make(chan string)
c2 := make(chan string)

select {
case msg1 := <-c1:
    fmt.Println("received", msg1)
case msg2 := <-c2:
    fmt.Println("received", msg2)
default:
    fmt.Println("no messages")
}

go func() {
    time.Sleep(1 * time.Second)
    c1 <- "channel1 : one"
}()
go func() {
    time.Sleep(2 * time.Second)
    c2 <- "channel2 : one"
}()

for i := 0; i < 2; i++ {
    select {
    case msg1 := <-c1:
        fmt.Println("received", msg1)
    case msg2 := <-c2:
        fmt.Println("received", msg2)
    }
```

```
}

/*
    no messages
    received channel1: one
    received channel2: one
*/

// Channels can be closed and iterated
channel := make(chan int, 5)

for i := 0; i < 5; i++ {
    channel <- i
}

close(channel)

for value := range channel {
    fmt.Println(value)
}

/*
    0
    1
    2
    3
    4
*/
```

Return to Summary

Package `fmt`

```
import "fmt"

fmt.Print("Hello World") // Print in console
fmt.Println("Hello World") // Print and add a new line in end
```

```
fmt.Printf("%s is %d years old", "John", 32) // Print with formatting
fmt.Errorf("User %d not found", 123) // Print a formatted error
```

Return to Summary