# BASH Scripting in Gitlab CICD and Projects


*starring Markus Geiger mg@evolution515.net*

Press ? for help!

*Chapter*

# BASH

# Defacto CLI Standard on *nix

```
GNU bash, version 5.2.15(1)-release (x86_64-pc-linux-gn
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.
```

# As Command Line Interface

```
$ aws ec2 describe-instances --filters Name=iam-instanc
  |      | jq -rce '.Reservations[].Instances[].InstanceI
  |      | tee /dev/tty \
  |      | xargs aws ec2 terminate-instances --instance-i
```

⇒ **`$` is the default SH prompt where BASH was derived from.**
Therefore we still see it today in code blocks.

# A Scripting Standard on *nix

As *scripting language*:

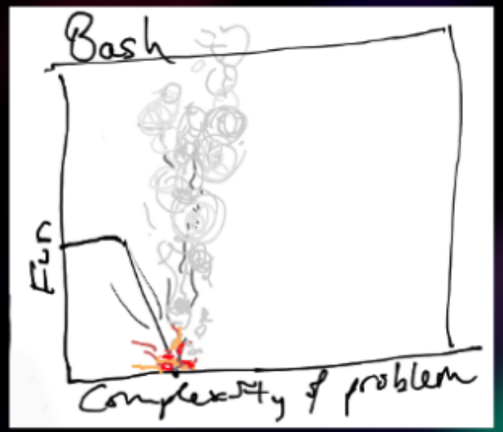
```
#!/usr/bin/env bash

set -eEuo pipefail

main() {
    local color="${1?color}"
    if [[ "$color" =~ ^(#?([a-fA-F0-9]{6}|[a-fA-F0-9]{3
        printf '%s\n' "${BASH_REMATCH[1]}"
    else
        printf '%s\n' "error: $color is an invalid hex
        return 1
    fi
}

main "$@"
```

# BASH isn’t meant for Complex Problems!

⇒ BASH lacks of data and control structures

⇒ BASH ancestor `sh` even doesn't know arrays!

⇒ BASH is slow and hard to learn!



# When to avoid BASH? [1]

**Performance matters?**
→ Use something other than shell!

**More than 100 lines long or non-straightforward control flow?**
→ rewrite it in a more structured language *now*.

[^1]: according to Google Shell Style Guide

*Chapter*

# Gitlab Runners

# Shell Election in a GitlabRunner Job

```
sh -c '
    if [ -x /usr/local/bin/bash ]; then
        exec /usr/local/bin/bash
    elif [ -x /usr/bin/bash ]; then
        exec /usr/bin/bash
    elif [ -x /bin/bash ]; then
        exec /bin/bash
    elif [ -x /usr/local/bin/sh ]; then
        exec /usr/local/bin/sh
    elif [ -x /usr/bin/sh ]; then
        exec /usr/bin/sh
    elif [ -x /bin/sh ]; then
        exec /bin/sh
    elif [ -x /busybox/sh ]; then
        exec /busybox/sh
    else
        echo shell not found
        exit 1
    fi
'
```

⇒ *`exec` does a process substitution*

⇒ The initial `sh` process is being replaced by the found shell binary.

**Gitlab Runner selects a suitable shell *which could be `sh` as well...***

# Trivia

What does that script execution behavior mean for *distroless* or *images without shell*?

# Trivia

What does that script execution behavior mean for *distroless* or *images without shell*?

They will fail because they require a shell!

# Trivia

What does that script execution behavior mean for *distroless* or *images without shell*?

They will fail because they require a shell!

Therefore Gitlab defaults to `:debug` version of the Kaniko image in their documentation.

# Trivia

What does that script execution behavior mean for *distroless* or *images without shell*?

They will fail because they require a shell!

Therefore Gitlab defaults to `:debug` version of the Kaniko image in their documentation.

Yes, you can *remove* `/bin/ba{sh}` for production if you don't need shell or that interpreter!

# Concenate Script Sections and Execute

Stored alike `/scripts-35992738-4805186176/step_script:`

```
 #!/usr/bin/env bash
set -eo pipefail set +o noclobber
: | eval $'\\\\
    export FF_CMD_DISABLE_DELAYED_ERROR_LEVEL_EXPANSION
    […]
    cd\\\\' "/builds/acme/users/markus.geiger/my-projec
    […]
    echo $\\\\'\\\\\\\\x1b[32;1m$ echo "Hello World!"\\
    echo "Hello World!"\\\\n'
    exit 0
    '
```

⇒ `before_script:` and `script:` sections are concenated *and*

⇒ escaped into a string to be evaluated

⇒ script path will be available as `$0` in script

# Takeaways

⇒ Dependend on container image you either use `SH` or `BASH`

⇒ `script:` is always executed by default shell options
`set -eo pipefail set +o noclobber`

⇒ `BA{SH}` therefore acts as *primary* control flow

⇒ In BASH we *primarily* work with *commands*

*Chapter*

# A Strange Language

*Section*

# Shebang!

# How do we execute commands?

Linux Kernel reads a *magic byte sequence* at the beginning of the file:

## *binfmt*

⇒ e.g. `0x7f454c46` → `ELF 64-bit LSB pie executable`

Program execution happens by the kernel.

## *Shebang!*

⇒ e.g. `#!/bin/bash` `#!/bin/python` `#!/usr/bin/env bash`

⇒ originally introduced as hack

Kernel finds `x2321` (`#!`) and delegates to userspace interpreter

# Where is your BASH located?

`#!/usr/bin/env bash` locates and executes `bash` depending on your `PATH` variable.

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

set -eEuo pipefail

SELF="${0}"
BASENAME="${0##*/}"

ROOT_PATH="$(realpath "${0%/*}/..")"
source "$ROOT_PATH/share/foo/lib.sh"

main "$@"
```

# Where is your Python located?

Use `#!usr/bin/env python3` and write portable python code

```
#!/usr/bin/env python

import sys
from foo import cli_main

if __name__ == '__main__':
    sys.exit(cli_main())
```

# How do we want to Execute our Scripts?

# Prepending Interpreter?

```
$ bash bin/myscript.sh
```

⇒ Now some people use `sh` – not compatible with `bash`

⇒ The "binary" can shall deceide what it needs to be executed!

⇒ Therefore we have a shebang like `#!/usr/bin/env {python,bash,..}``

# Better: Execute as Commands

```
# relative to current/project directory
$ bin/myscript
# if it's in the current directory
$ ./myscipt
```

```
# using env you don't need to specify location
$ export PATH="$PWD/bin:$PATH"
$ myscript
```

⇒ This is how *commands* are used in Linux!

⇒ By using commands we can chain input and outputs and use pipes.

# Takeaways

⇒ **Always start your scripts with `!#/usr/bin/env bash`**
(except you want to target the system's interpreter at `/bin/bash`)

⇒ **Make your script files executable by `chmod +x`**

⇒ **Avoid "binaries" having a `.sh` extension!**

⇒ **Use `.sh` only for libraries!**

*(Taken from primarly Google Shell Style Guide and other sources)*

*Section*

# SH, BASH and POSIX?

# How do SH and BASH relate to each other?

BASH can execute SH script but SH cannot execute BASH:

⇒ SH is POSIX.2 Standard

⇒ BASH is Extended POSIX Standard

# Little Shell History

## sh

⇒ **1979** first release of of the Bourne Shell

⇒ **1989** rewrite as `ash` by Kenneth Almquist (Almquist Shell)

⇒ **2002** ported to Debian as `dash` (DASH)

⇒ **2006** Ubuntu's system `sh` now defaults to `dash`

## bash

⇒ **1989** v.99 released as *better* `ash` under GPLv1

...

⇒ **2006** v4 released – license change to GPLv3

⇒ **2019** v5 released – Apple changes macOS default shell BASHv3 to `zsh`

**⚠ If you are targeting older systems make sure you test with BASH v4!**

# How does SH compare to BASH?

⇒ No arrays and `==` signs

⇒ No built-in Regular Expressions (BASH_REMATCH)

⇒ No `[[` or `source` `function` keywords

⇒ No `local` `shopts` (like pipefail) or `declare` keywords``

⇒ No `<<<'here strings'` or `$'...'`

⇒ No substrings (`${x/y/z}`), parameter substition (`${x:y:z}`) or expansion

⇒ Bashism provides a full list and a guide to port BASH to SH

*Section*

# On Container Shells

# Alpine or Embedded Linux

⇒ Uses `dash` (Debian ASH) but it's called `ash` and embedded into BusyBox

⇒ **It does *not* come with BASH except it was explicitly added**

⇒ `bash` can be installed as additional package

⇒ **but** since Gitlab only locates `sh` but not `ash` we will *only* have SH in Alpine.

# Forcing BASH in Alpine Images

You can rebuild or use this Alpine BASH Workaround

```
alpine-default-sh:
image: alpine:latest
script:
  - echo "$SHELL"
  # you are limited to SH even if alpine has ASH!!!

alpine-w-ash:
image:
  name: alpine:latest
script:
  - test -z "${SHEBANG}" && export SHEBANG=ash && exec
  # now we have ash and can continue scripting
  - echo "$SHELL"

alpine-w-bash:
image:
  name: alpine:latest
script:
  - test -z "${SHEBANG}" && apk add bash && export SHEB
   # now we have real bash and can continue scripting
  - echo "$SHELL"
```

# Installing Alpine Linux from `rootfs`

```
# we just got busybox. a good start to download alpine
busybox wget https://dl-cdn.alpinelinux.org/alpine/v3.1
  |  tar  -C / -zxv --exclude "./etc/hosts" --exclude "
# let's add bash
/sbin/apk add bash

# if you are interactive you could exec `/bin/bash -il`

# for gitlab we wanna do a process substitution
test -z "${SHEBANG}" && apk add bash && export SHEBANG=
```

⇒ This will install Alpine Linux from rootfs on top of a `busybox` image like `gcr.io/kaniko-project/executor:debug`

# Takeaways

⇒ A Kernel Hack made it possible to run scripts as commands

⇒ `SH` scripts are often run by `dash` or `bash` indeed

⇒ POSIX is basically *Korn Shell* and modern *sh*

⇒ BASH is “Extended POSIX” syntax

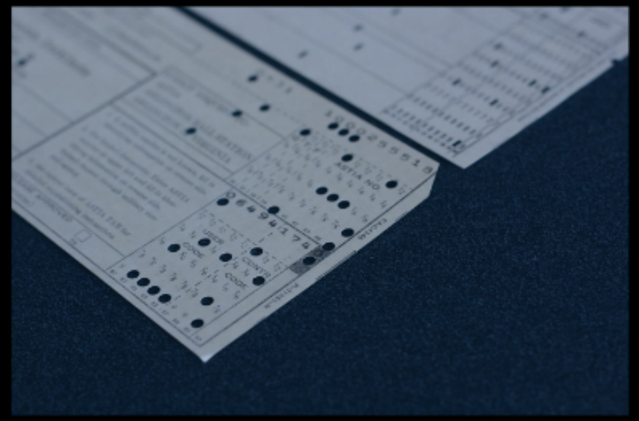
⇒ Alpine Linux does not come with `bash`

⇒ Gitlab could select `sh` for alpine based images (but we can work around)

*Chapter*

# A Strange Language

# BASH as *Command* Language



# Commands vs. Functions

**Commands do have Input / Output / Process IDs**

⇒ think of them as command cards

**Functions are used inside programming language runtimes**

⇒ think of them mathematically

# Error Codes

Bash has reserved error codes for exiting a process

⇒ `0` - successful termination (EX_OK) → *also your BASH functions should return 0 on success!*

⇒ `1` - Catchall for general errors

⇒ `2` - Misuse of shell builtins (according to Bash documentation)

⇒ `126` - Command invoked cannot execute

⇒ `127` - “command not found”

⇒ `128` - Invalid argument to exit

⇒ `128+n` - Fatal error signal “n”

⇒ `130` - Script terminated by Control-C

⇒ `255\*` - Exit status out of range

The rest is on you or the comands you are using!

# A POSIX Process

# Did anyone ask *Terminal?*


**ADM-3A is not a computer!**

*It's a remote input and output device!*

# You could still use a dumb terminal today!

```
$ TERM=dumb less /etc/hosts
WARNING: terminal is not fully functional
Press RETURN to continue
```

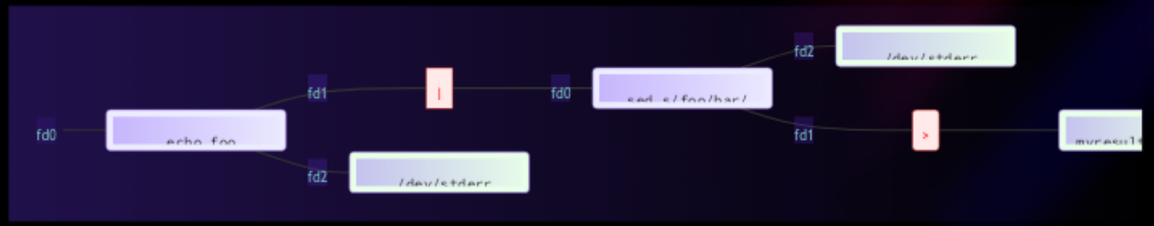
`TERM="xterm"` first defined 256-color, resizing and a mouse cursor

➡ *If you are interested in more: 8bit Guy – What are Dumb Terminals?*

# A Process in the Terminal (detailed with symlinks)

# Processes in a POSIX Pipe

```
$ echo foo | sed s/foo/bar/ > myresult.txt
```




Here we replace the input foo with the string bar using `sed`(search and displace)

The output STDOUT becomes the input STDIN of the next command.

**Note: STDERR is not piped to the next command.**

# STDERR is not piped thru!

```
# first part: Output to both stdout and stderr
# second part: Redirect to stderr to null
$ (echo foo >&1; echo bar >&2) | cat 2>/dev/null
bar
foo
```

```
# first part: Output to both stdout and stderr
# second part: Redirect to stdout to null
$ (echo foo >&1; echo bar >&2) | cat 1>/dev/null
bar
```

# Is there a way to catch STDERR of both?

```
#!/usr/bin/env bash
(echo foo >&1; echo bar >&2 | sed 's/foo/bar/g' ) > /tm
EXIT_CODE=$!
if "$EXIT_CODE" -ne 0 && grep -q "TooManyRequests" /tmp
    echo "Do something special on this error"
fi
```

*Chapter*

# Debugging

*Section*

# Strict Mode

# Strict Mode

It's better to know errors before they happen:

```
# (U # "Gitlab Script-Mode"
set  set -eo pipefail set +o noclobber
```

⇒ `-e` *exit on an error*

⇒ `-u` *error on undefined variables (otherwise BASH takes them as empty)*

⇒ `-E, -o errtrace` *if you set an error trap it is inherited to subshells*

⇒ `-o pipefail` *fail pipe if any command in pipe exits with <> `0`

⇒ `+o noclobber` *allow overwriting of files by `>` redirectionoperator*

**If these options are not set, BASH will just continue running a script that had an error including *all* consequences!**

It's also a good idea to use the first one in Gitlab `script:`!

Don’t just put `set -ueEo pipefail` on existing scripts unless you want pain!





## Please mind the `+` to switch options "off"

⇒ `+o ${option}` or `+x` – *switch "on"*

⇒ `-o` activated an option! – *switch "off"*

# BASH Option: Pipefail

```
$ echo foo | grep bar | cat
→ exit code 0

$ set -o pipefail
$ echo foo | grep bar | cat
→ exit code 1
```

Note: `sh` doesn't have this option!

# BASH Option: Exit on Error

```
$ set -e
$ rm -f nonexistent
→ exit script with code 1

$ rm -f nonexistent || true
→ exit code 0 continue script
```





# BASH Option: Exit on Undefined Variable

```
$ set -u
$ echo "${UNDEFINED}"
→ exit script with code 1

$ echo "${UNDEFINED:-unset}"
→ echos unset with exit code 0

# usage in if if nonzero
if [[ -n "${UNDEFINED:-}" ]]; then command; fi;

# will exit with 1 since you should add || true or set
[[ -n "${UNDEFINED:-}" ]] && command
```

*Section*

# Linting

# BASH Linting aka. “noexec” mode

Bash brings a built-in “linter”:

```
$ bash -n myscript
```

You could also put it at the start of a script – *including Gitlab CI ;)*

```
$ bash -n "$0" || exit 1
```

```
OUTPUT
/scripts-35992738-4805186176/step_script: line 66: synt
```

# Shellcheck

```
$ shellcheck myscript
```

```
In myscript line 6:
echo "hello world
      ^-- SC1078 (warning): Did you forget to close thi

In /home/ctang/.local/bin/ec2-log line 62:
        if [[ "$EC2_STATE_CURRENT" != $EC2_STATE_TARGET
              ^----------------^ SC2250 (style): Prefe
                                 ^---------------^
                                 ^---------------^


Did you mean:
        if [[ "${EC2_STATE_CURRENT}" != ${EC2_STATE_TAR
```

Ignore errors by annotation:

```
# shellcheck disable=SC2116,SC2086
```

**Website:** https://www.shellcheck.net/ (git |install|docs on ignore)

*Section*

# Non-interactive Debugging

# xtrace

```
# Enable xtrace and enable function tracing
set -xo functrace

function foo {
  echo "bar"
}
foo "$@"

# Disable xtrace here
set -x
```

OUTPUT

```
+ foo
+ echo bar
bar
+ set -x
```

# Using DEBUG trap

```
unset foo
# Set the trap - it's active fromon this point
trap 'echo "→ current foo=${foo:unset} → next $BASH_SUB
foo=1
let foo++
# Unset the trap
trap - DEBUG
```

```
→ current foo= → next 26:foo=1
→ current foo=1 → next 27:let foo++
→ current foo=2 → next 28:trap - DEBUG
```

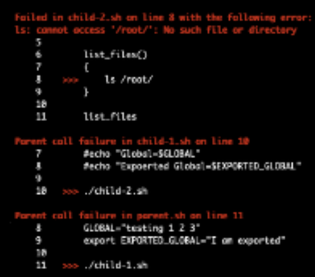
# Convinience Solution: Trapper

```
curl -sSfL https://raw.githubusercontent.com/Developers
    | install -m 0700 /dev/stdin -D $HOME/.local/share/
```

```
set -o functrace
source "$HOME/.local/share/trapper/trapper.sh"
```

Mirror on my Gitlab profile with `-o functrace` fix:

```
eval $(curl -sSfL 'https://gitlab.com/-/snippets/258846
```




https://github.com/DevelopersToolbox/trapper

*Section*

# *(Interactive)* Debugging

# “Simple” Bash Debugger

Using bashdb – not to be mistaken by “official” bashdb https://bashdb.sourceforge.net/

```
curl -sSfL https://gitlab.com/-/snippets/2588469/raw/ma
    | install -m 0700 /dev/stdin -D $HOME/.local/share/
```

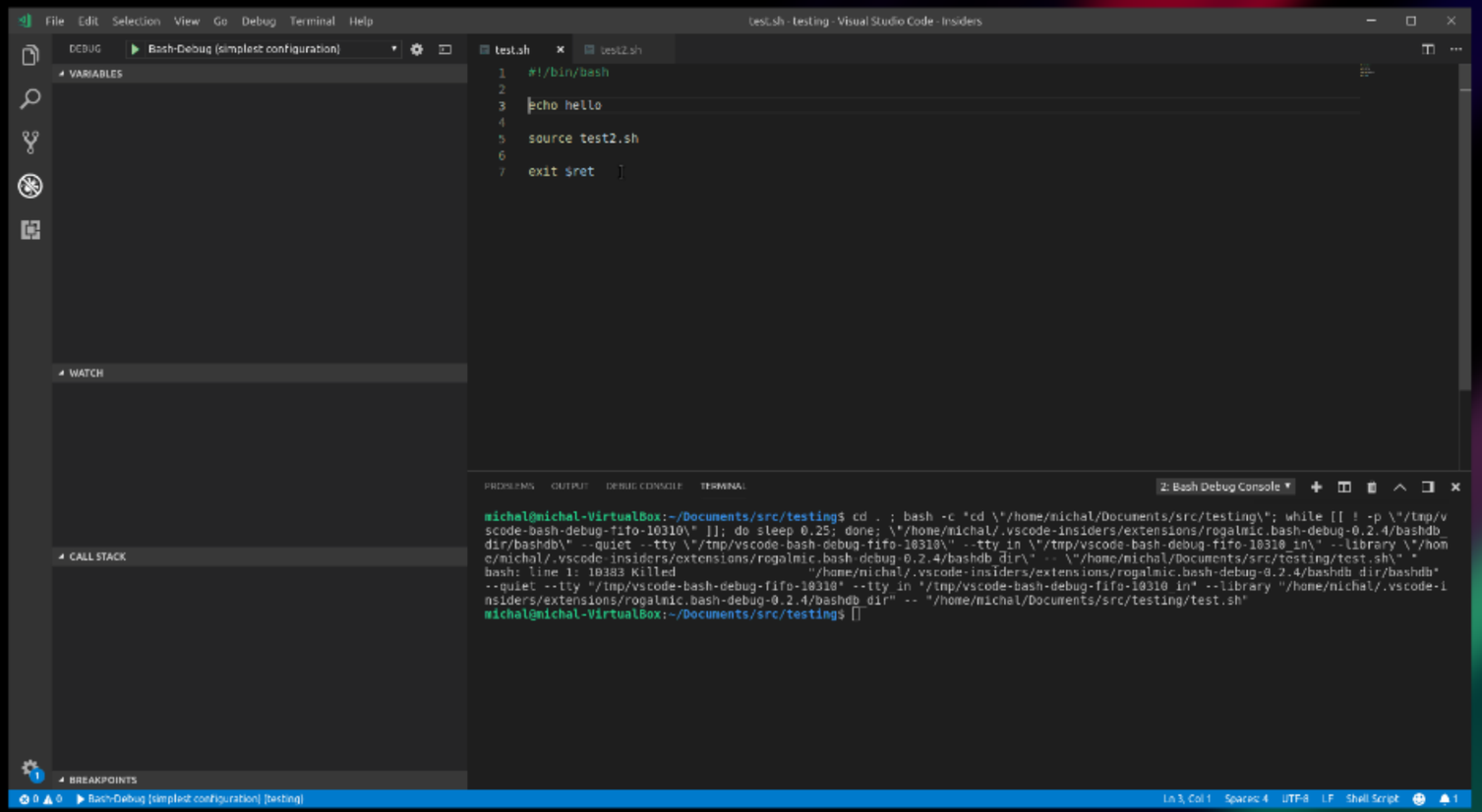
Then include after shebang or where debuggign shall begin and enable function tracing

```
set -o functrace
source "$HOME/.local/share/bashdb/bashdb.sh"
```

```
n=next  s=step into  r=run until  C=continue  p=print
(dbg) $?=0 /home/ctang/.local/bin/ec2-log:473 [main] ma
/home/ctang/.local/bin/ec2-log: line 453: 1: instance-i
```

It has also a nyce output for non-interactive bash execution tracing.

# VSCode or Jetbrains Plugin




https://marketplace.visualstudio.com/items?itemName=rogalmic.bash-debug

**Limitations and known problems**

⇒ `$0` variable shows path to bashdb

⇒ Older `bash` versions ( `4.0` - `4.2` ) are not tested, but might work™

⇒ `BASH_REMATCH` gets overwritten when stepping through code

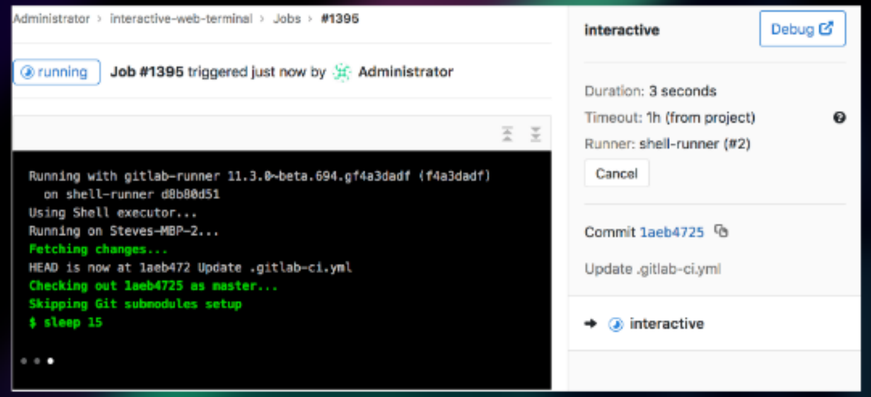
*Section*

# Interactive Remote Terminal(s)

# Gitlab Interactive Web Terminal

It needs a *session server* to be configured in the Gitlab Runner:

```
[session_server]
   session_timeout = 1800
   # actual server listen address
   listen_address = "0.0.0.0:8093"
   # advertise_address
   advertise_address = "runnerid-subdomain.domain:8093"
```




⇒ `advertise_address` is what the Gitlab API get's to connect to the runner.

⇒ Add K8s Ingress and a NetworkPolicy to allow GitLab CIDR only.

⇒ Probably it needs a `sleep 3600` in your job script.

# Use `sleep` in job and `kubectl exec`

Note: You may run into a Gitlab Runner timeout!

# Takeaways

⇒ It's better to know errors before they happen: use strict mode!
`set -ueEo pipefail`

⇒ `trapper` is nice solution for debugging

⇒ Interactive Web Terminals are a good Gitlab Solution but need to be setup

⇒ You need to place `sleep 3600` into you job script when debugging

*Chapter*

# Addendum

*Section*

# Understanding BASH

# ExplainShell.com

# ExplainShell.com


# AI based tools

The command you've provided is a playful example that demonstrates the use of exit status codes and logical operators in Bash. However, it's important to note that the command `destroy-the-world` is not a real command and is used here for illustrative purposes only.

Let's break down the command step by step:

```bash
{echo foo >&1; echo bar >&2) | cat 2>/dev/null && destroy-the-world
```

1. `(echo foo >&1; echo bar >&2)`: This is a subshell enclosed within parentheses. Inside the subshell:
   - `echo foo >&1`: The `echo` command prints "foo" to the standard output (file descriptor 1). `>&1` redirects the standard output to file descriptor 1, which is effectively writing "foo" to the standard output.
   - `echo bar >&2`: The `echo` command prints "bar" to the standard error (file descriptor 2). `>&2` redirects the standard output to file descriptor 2, which is effectively writing "bar" to the standard error.

   So, within the subshell, "foo" is sent to standard output, and "bar" is sent to standard error.
2. `| cat 2>/dev/null`: This part of the command pipes the output of the subshell to the `cat` command, and then redirects standard error to `/dev/null` to discard it.
   - `|`: This pipe symbol connects the standard output of the preceding command (the subshell) to the standard input of the following command (`cat`).
   - `cat`: The `cat` command reads from its standard input and outputs the received content to its standard output. In this case, it will receive "foo" from the subshell's standard output.
   - `2>/dev/null`: This part redirects standard error (file descriptor 2) to `/dev/null`, which effectively discards any error messages that might have been generated by the `cat` command. Since `cat` doesn't write any error messages, this redirection has no practical effect in this context.
3. `&& destroy-the-world`: The `&&` operator is a logical AND operator in Bash. It's used to execute the command on the right side of the operator only if the command on the left side succeeds (returns an exit status of 0).
   In this case, `destroy-the-world` is a fictional command, and it's used here to represent a catastrophic action that should only be executed if everything before it succeeds. Since the `&&` operator checks the exit status of the preceding commands, and the command before it (`(echo foo >&1; echo bar >&2) | cat 2>/dev/null`) doesn't actually fail (it produces "foo" as output and no errors), the fictional `destroy-the-world` command will not be executed.

In summary, the command demonstrates how exit status codes and logical operators work in Bash. It uses fictional commands to illustrate the concept of only executing a subsequent command if the preceding command succeeds. It's important to understand that the `destroy-the-world` command is not a real command and doesn't have any actual impact.

# ShellAI


https://github.com/ibigio/shell-ai

*Section*

# Learning Bash





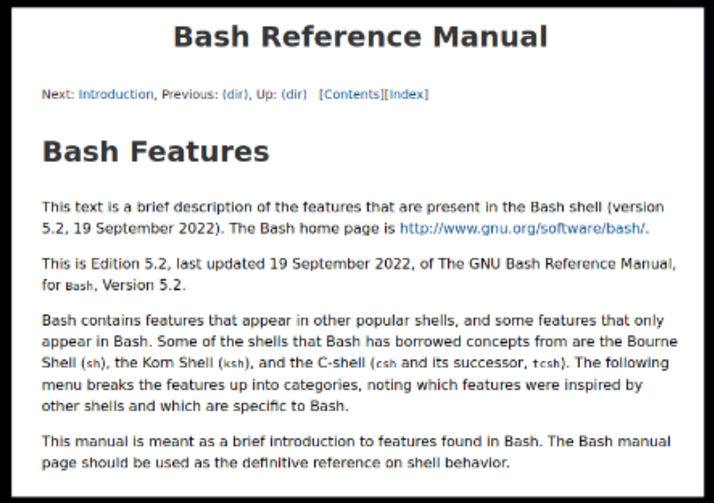
# Shell Style Guide

Revision 2.02

Authored, revised and maintained by many Googlers.

## Table of Contents

# Bash Reference Manual



Bash Reference Manual

Next: Introduction, Previous: (dir), Up: (dir) [Contents][Index]

Bash Features

This text is a brief description of the features that are present in the Bash shell (version 5.2, 19 September 2022). The Bash home page is http://www.gnu.org/software/bash/.

This is Edition 5.2, last updated 19 September 2022, of The GNU Bash Reference Manual, for Bash, Version 5.2.

Bash contains features that appear in other popular shells, and some features that only appear in Bash. Some of the shells that Bash has borrowed concepts from are the Bourne Shell (sh), the Korn Shell (ksh), and the C-shell (csh and its successor, tcsh). The following menu breaks the features up into categories, noting which features were inspired by other shells and which are specific to Bash.

This manual is meant as a brief introduction to features found in Bash. The Bash manual page should be used as the definitive reference on shell behavior.

⇒ Web: https://www.gnu.org/software/bash/manual/bash.html

⇒ Manpages: `man bash` `man ${COMMAND}`

⇒ Built-ins: `help ${COMMAND}` or when using other shell `bash -c 'help ${COMMAND}'`

# Cheat Sheets

# Cheat.sh

```
$ curl cheat.sh/
      _                    _          _    __
  ___| |__   ___  __ _| |_   ___| |__ \ \      The only
 / __| '_ \ / _ \/ _` | __| / __| '_ \ \ \     Unified
| (__| | | |  __/ (_| | |_ _\__ \ | | |/ /     communit
 \___|_| |_|\___|\__,_|\__(_)___/_| |_/_/      reposito

+------------------------+ +------------------------+ +
| $ curl cheat.sh/ls     | | $ cht.sh btrfs         | |
| $ curl cht.sh/btrfs    | | $ cht.sh tar~list      | |
| $ curl cht.sh/tar~list | |                        | |
| $ curl https://cht.sh  | |                        | |
|                        | |                        | |
|                        | |                        | |
+-- queries with curl ---+ +- own optional client --+ +
+------------------------+ +------------------------+ +
| $ cht.sh go/f<tab><tab>| | $ cht.sh --shell       | |
| go/for   go/func       | | cht.sh> help           | |
| $ cht.sh go/for        | | ...                    | |
| ...                    | |                        | |
|                        | |                        | |
|                        | |                        | |
```

# TL;DR Pages

> *The tldr pages are a community effort to simplify the beloved man pages with practical examples.*

```
$ tldr bash
  Bourne-Again SHell, an `sh`-compatible command-line i
  See also: `zsh`, `histexpand` (history expansion).
  More information: <https://gnu.org/software/bash/>.

  Start an interactive shell session:
      bash
 …
```

https://tldr.sh/ | https://github.com/tldr-pages/tldr 💔 TypeScript | https://github.com/dbrgn/tealdeer 💔 Rust

# Command Line Fu

⇒ Web: https://www.commandlinefu.com/commands/browse

⇒ CLI: https://gitlab.com/-/snippets/2588469/raw/main/clfu

*Section*

# Raw Links

# Reference

⇒ Bash Reference Manual

⇒ BASH Guide





# Learning BASH

⇒ A guide to learn BASH

⇒ The Art of Command Line

⇒ Linux Shell Scripting Tutorial (LSST) v2.0 by Vivek Gite (nixcraft)

⇒ The Linux Documentation Project: Bash Programming - Intro/How-to

⇒ The Linux Documentation Project: Advanced Bash Scripting Guide

# Efficient BASH (!)

⇒ Pure BASH Bible

⇒ Pure SH Bible

Can I do it in pure BASH or SH programming?!

Excellent examples that run with **strict mode** and to learn from!

# Dos and Don’ts

⇒ Google's Shell Style Guide

⇒ Shell Field Guide

# Awesome BASH

⇒ Awesome Shell – A curated list of awesome command-line frameworks, toolkits, guides and gizmos. Inspired by awesome-php.

⇒ Awesome BASH – A curated list of delightful Bash scripts and resources





# Useful References

⇒ Gist: ANSI Escape Sequences