Just use just

June 02, 2021 · 1364 words · 7 min

OMG, the blog is live! 😱 And this is the first article! 😱

This first article will be about Just a **command-line** tool I recently discovered that immediately became essential in many work projects. Since it's a tool written in **Rust**, it's fast, it's well designed and documented, it features colored output, and it's an essential step in your terminal's *hypsterization* process!

Let's suppose you've just deployed your application via `scp` (*sigh!*) on one of your work's machines. Maybe your application was already built using tools like Decline, so it's already capable of parsing command-line options and flags and printing a complete help like:

```
$ foo --help
Usage:
    foo schedule
    foo encrypt
    foo decrypt

foo tool, it can encrypt and decrypt files and schedule operations

Options and flags:
    --help
        Display this help text.

Subcommands:
    schedule
        schedules encryptions/decriptions
    encrypt
        encrypts files
    decrypt
        decrypts files
```

But let's add a **slow-changing configuration** to the scenario, which changes so often that it doesn't justify a refactor to add a library like Ciris to your code. Maybe some **non-**

power users need to change that configuration once a week or month *because of reasons*.

What's missing? Maybe there's a local MySql that needs to be queried for maintenance operations, or perhaps a remote database/storage/service/whatever that requires another command-line tool to be interacted with.

This is one of the times in which unmaintained, undocumented, faulty crap like **maintenance_script.sh** or **fix_for_prod.sh** begins to spread around. In no time, the situation will look similar to

```
/home/applicative_account/perform_operation.sh
/home/colleague1/perform_operation_copy.sh
/home/colleague1/old_version/perform_operation_as_root.sh
/home/sre_guy/this_should_fix_everything.sh
/home/random_data_scientist/do_not_run.sh #(ofc it was chmod +x)
```

90% of them will have the shebang `#!/bin/bash` while the 10% `#!/bin/sh`. Some of them will have `zsh` commands because there are people around that uses `zsh` (like me) that forgets that it doesn't share 100% of the syntax with `bash` (not like me, I swear).

Most of them will contain almost the same commands like

```
mysql prod_db < maintenance.sql > maintenance_output.dump
```

or templatized commands like

```
"/foo-${VERSION}/bin/foo"
```

that depend on environment variables defined in the `.profile` of a deleted user.

The last time you used ShellCheck to check the scripts, the linter exploded, and somewhere in the world, Stephen Bourne suddenly began crying without any apparent reason.

Just to the rescue

As its Github README states, Just *is a handy way to save and run project-specific commands* called **recipes**, stored in a file called `justfile` with a syntax inspired by **Make**.

Here's a tiny example:

```
build:
    cc *.c -o main

# test everything
test-all: build
    ./test --all

# run a specific test
test TEST: build
    ./test --test {{TEST}}
```

Just searches for a `justfile` in the current directory written in its particular syntax, so let's begin creating one with an hello world recipe and let's try to run it:

```
hello-world:
    echo "Hello World!"
```

▾ output

```
$ just hello-world
echo "Hello World!"
Hello World!
```

As you can see, just **shows the command** that is about to run before running it, while we can't say the same for global or user-defined `alias`es in various shells (unless using something like `set -x` for bash). If you want to suppress this behaviour, you can put a `@` in front of the command to hide.

```
hello-world:
    @echo "Hello World!"
```

▾ output

```
$ just hello-world
Hello World!
```

Let's try to create a second recipe with an argument.

```
hello-world:
    @echo "Hello World!"

salute guy:
    @echo "Hello {{guy}}!"
```

▾ output

```
$ just salute
error: Recipe `salute` got 0 arguments but takes 1
usage:
    just salute guy

$ just salute Tonio
Hello Tonio!

$ just --dry-run salute Tonio
echo "Hello Tonio"
```

The recipe cannot obviously run without an argument since that argument is referred to in the body of the recipe using just syntax `{{ argument_or_variable_name }}`. If you want to "debug" the recipe that will run with the provided arguments, you can use the `--dry-run` command-line flag. This can come in handy if a command is long and complex and you have, for example, to schedule it in your crontab file. Just copy it from there.

Arguments are really powerful since they can have **default values** and can be **variadic** (both in the form `zero or more` or `one or more`):

```
hello target="World":
    @echo "Hello {{target}}!"

hello-all +targets="Tim": # One or more plus a default value
    @echo "Hello to everyone: {{targets}}!"

hello-any *targets: # Zero or more
    @echo "Hello {{targets}}!"
```

▾ output

```
$ just hello
Hello World!

$ just hello-all
Hello to everyone: Tim!

$ just hello-all "Tim" "Martha" "Lisa"
Hello to everyone: Tim Martha Lisa!

$ just hello-any
Hello !

$ just hello-any "Bob" "Lucas"
```

```
Hello Bob Lucas!
```

We know enough syntax. Let's try to build a meaningful example for our **messed-up work machine** and let's try new features **just** if we need them (no pun intended 😄).

An almost working example

If we inspect the history of our machine, we'll notice that most of the commands are `foo` invocations with `nohup` and stdin and stderr redirection into a `.log` file. We should consider refactoring the application, removing all the `println`s to replace them with a `logger.info`, maybe using a logging framework that automatically handles log rotation and similar.

In the meantime, we can standardize how `foo` is called, how the outputs are redirected, and its execution detached to avoid interactive sessions that might early terminate if you close a terminal session.

```
foo_version    := "0.3.0"
foo_executable := "/home/power_user/foo-" + foo_version + "/bin/foo"
conf_file      := "/home/power_user/foo.conf"
log_file       := "/home/power_user/foo.log"

# encrypts 'target' and detaches
encrypt target:
    nohup {{foo_executable}} "encrypt" {{target}} {{conf_file}} &>> {{log_file}} &

# decrypts 'target' and detaches
decrypt target:
    nohup {{foo_executable}} "decrypt" {{target}} {{conf_file}} &>> {{log_file}} &

# schedules operations formatted like '<cron_expression> <decrypt|encrypt> <target>'
schedule operation:
    nohup {{foo_executable}} "schedule" "{{operation}}" {{conf_file}} &>> {{log_file}} &
```

(Probably `nohup` + `&` is overkilling, but who cares 😇?)

That's better. We've used **variables** to avoid repetitions, templatized every recipe and added comments. It would be nice, though, to directly tail the `log_file` once a recipe is launched and avoid repetitions even more.

```
foo_version    := "0.3.0"
foo_executable := "/home/power_user/foo-" + foo_version + "/bin/foo"
conf_file      := "/home/power_user/foo.conf"
log_file       := "/home/power_user/foo.log"
```

```
_default:
    @just --list --unsorted

# encrypts 'target' and detaches
encrypt target:
    @just _run_detached "schedule" "{{target}}"
    @just tail

# decrypts 'target' and detaches
decrypt target:
    @just _run_detached "schedule" "{{target}}"
    @just tail

# schedules operations formatted like '<cron_expression> <decrypt|encrypt> <target>'
schedule operation:
    @just _run_detached "schedule" "{{operation}}"
    @just tail 20

# Follows the log file
tail n="200":
    tail -{{n}}f {{log_file}}

_run_detached command argument:
    nohup {{foo_executable}} {{command}} {{argument}} {{conf_file}} &>> {{log_file}} &
```

Nice, we've used many features of just, in particular recipes whose name begins with an underscore are called *hidden recipes*. Hidden means that if you run `just --list`, they won't get printed since they're meant to be used internally. A special recipe was used, the `default` one, that gets called if you prompt `just` without any recipe name. [EDIT] (Since the name is not precisely `default`, just runs the first recipe in the justfile, that has to be a recipe without arguments)

```
$ just
Available recipes:
    encrypt target     # encrypts 'target' and detaches
    decrypt target     # decrypts 'target' and detaches
    schedule operation # schedules operations formatted like '<cron_expression> <decrypt|encry
    tail n="200"       # Follows the log file
```

Oh nice, the **comments** we wrote previously just became documentation! Plus, we called the `tail` recipe from others, letting `just encrypt "something"` resemble an interactive command.

Let's now set the same interpreter for all the recipes choosing from the available ones:

`set shell := ["bash", "-uc"]`. This way, every recipe line will run in a newly spawned sub`shell`, `bash` in this case. If it feels like the way the shebang `#!/bin/bash` works, you're right.

In fact, it's possible to define shebang recipes to be able to use local variables in recipes but remember to add `set -euxo pipefail` like the documentation suggests if you're using Bash to maintain the fail-fast behaviour.

Mixing and stirring *commands*, *recipes*, *just features* you'll probably come up with something similar to this **prod-like example**:

○○○ Justfile

```
set shell := ["bash", "-uc"]

# Foo
foo_version    := "0.3.0"
foo_executable := "/home/power_user/foo-" + foo_version + "/bin/foo"
conf_file      := "/home/power_user/foo.conf"
log_file       := "/home/power_user/foo.log"

# Bar
bar_executable := "/home/power_user/bar"
sre_victim     := "baz@sre.com"

# MySql
my_sql_default_user := "random_guy"
dump_query          := "select 'I have no intention to write queries in this example';"
now                 := `date -u +"%Y-%m-%dT%H:%M:%SZ"`
mysql_output_file   := "/home/power_user/mysql_dumps/" + now + ".dump"

# Colors
RED    := "\\u001b[31m"
GREEN  := "\\u001b[32m"
YELLOW := "\\u001b[33m"
BOLD   := "\\u001b[1m"
RESET  := "\\u001b[0m"

## Foo Recipes

_default:
  @just --list --unsorted

# encrypts 'target' and detaches
encrypt target:
    @just _run_detached "schedule" "{{target}}"
    @just tail
```

```
# decrypts 'target' and detaches
decrypt target:
    @just _run_detached "schedule" "{{target}}"
    @just tail

# schedules operations formatted like '<cron_expression> <decrypt|encrypt> <target>'
schedule operation:
    @just _run_detached "schedule" "{{operation}}"
    @just tail 20

# Follows the log file
tail n="200":
    tail -{{n}}f {{log_file}}

# Unsurprisingly kills foo
kill:
    pgrep -f {{foo_executable}}

## Bar Recipes

# Will notify an SRE with a boring mail.
notify:
    @just _bold_squares "{{YELLOW}}WARNING"
    @echo -e "{{BOLD}} A SRE will be notified with an e-mail!{{RESET}}"
    {{bar_executable}} notify {{sre_victim}}

## MySql Recipes

# runs the dump query
dump username password:
    @just kill
    @just _mysql_command_to {{username}} {{password}} {{dump_query}} > {{mysql_output_file}}

# runs the dump query with default user
dump-with-default-user password:
    @just kill
    @just _mysql_command_to {{my_sql_default_user}} {{password}} {{dump_query}} > {{mysql_out

## Hidden Recipes

_bold_squares message:
          @echo -e "{{BOLD}}[{{RESET}}{{message}}{{RESET}}{{BOLD}}]{{RESET}}"

_mysql_command username password query:
    mysql -u {{username}} -p {{password}} -e {{query}}

_mysql_command_to username password query output_file:
    _mysql_command {{username}} {{password}} {{query}} > {{output_file}}
```

```
_run_detached command argument:
    nohup {{foo_executable}} {{command}} {{argument}} {{conf_file}} &>> {{log_file}} &
```

"It's not enough to enforce people to not mess up production machines with crappy shell scripts!"

Obviously, just doesn't automatically solve every problem you might encounter in **heavily unmaintained machines** with a lot of conflicting shell scripts, mostly because of people, but at least:

- It lets you concentrate every **project-related** commands in a **single file** that can be easily tracked by a VCS to become part of the deployment
- It **declaratively** sets the interpreter
- It lets you you write a multi-command script without relying on super-verbose and tricky `match-case` bash syntax with the addition of:
  - **default arguments**
  - **easy string templating**
  - command evaluation using backticks (see the `now` variable in the previous example)
  - conditional expressions that are evaluated before the command execution
  - `get_or_else` syntax for environment variables
- It integrates with `fzf` to choose argument-less recipes interactively
- Recipes can depend on other recipes, like `tests` on `build` as in the first example
- It can generate its own shell completion scripts using `just --completions <shell_name>`
- It can be used as an interpreter, turning `justfile`s in runnable just script simply prepending `#!/usr/bin/env just --justfile` (This can be handy if you maybe want to use it with `crontab`)

and **HIPSTER ALERT**:

- It has its own Github Action
- Syntax Highlight for Vim, Emacs and Visual Studio Code is already available

Creating practical recipes, installing the prebuilt binaries, and the command-line completion scripts can probably convince people to use it. If not, try documenting your software, using examples in the `justfile` that's sitting in the home of the repo, or try harder using

```
*********************************
* Run `just` for a complete list *
* of available commands          *
*********************************
```

as the `/etc/motd` for the prod machines.

More Tools!

▾ SPOILER: next tool
Zola : the templating engine I'm using for this blog :)

1 reaction

0 comments *– powered by giscus*