DEV TOOLING FOR YOU AND ME

Why BASH is sometimes your friend...
How can our **dev** script save you time...
What other scripts exist you may not know about...

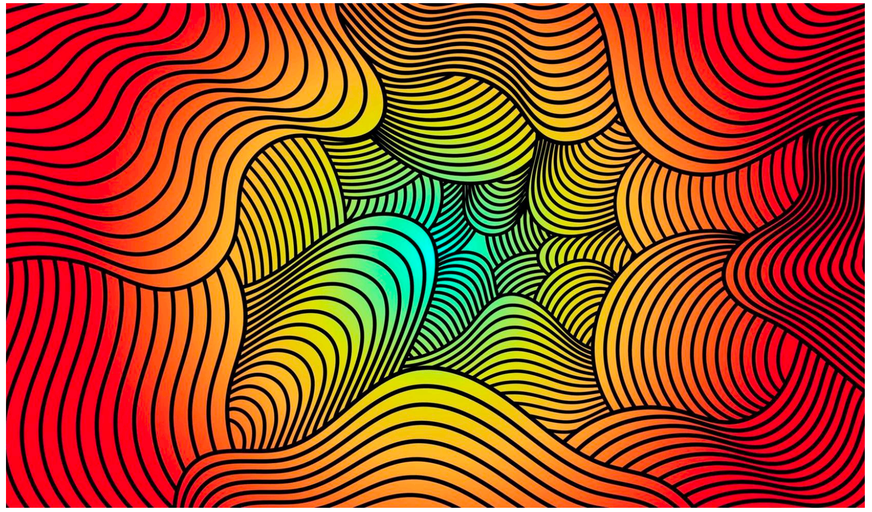
academia.edu • May 2025

AGENDA

- **Why shell scripting at all?**
  - When is it appropriate?
  - Why does it have such a bad rep?
  - Self-help and self-explanatory usage
- **dev script for simplifying your workflow**
  - What are its design goals?
  - What is it not?
  - Usage and demo
- **Other scripts demo:**
  - ecs-ssh
  - process-watch & process-list

WHEN IS SHELL SCRIPTING APPROPRIATE?

- Shell scripting is great when you need to need to automate a repeatable sequence of UNIX commands
- It's great when you want to install, run, verify, abort on error — all around running external commands and processes.
- It's not great when:
  - you need to use associative arrays (only supported in BASH v5, MacOS comes with v3 due to licensing issues, and switched ZSH as the default shell)
  - you feel like you need OOP — inheritance, polymorphism, internal state, etc
  - you need to parse complex data structures for which other languages have libraries you can use
  - the script is **NOT** well structured, functions well named, errors are unchecked, and nobody can understand it or fix any issues that may come up.



WHY THE BAD REP?

SEEING IS BELIEVING THAT IT'S WORKING

- Running shell scripts more often than not hides what's actually running, and only "barf" when they error out.
- If you know how to use "set", it will show too much output, or almost none at all.
  - `set -x`
- Style guides are few and far between, and most folks don't know how to write "good" shell code or how to lint it

WHY THE BAD REP?

ERROR HANDLING IS ... WAT?

- If you use this setting, any error interrupts the entire script
  - `set -e`
- Any good script must print help with -h or —help, but not all do or do it well
- And if the script is not self explanatory (i.e. if you run it with wrong options it doesn't tell you what you did wrong) most folks will abandon it, myself included.

THE DEV SCRIPT

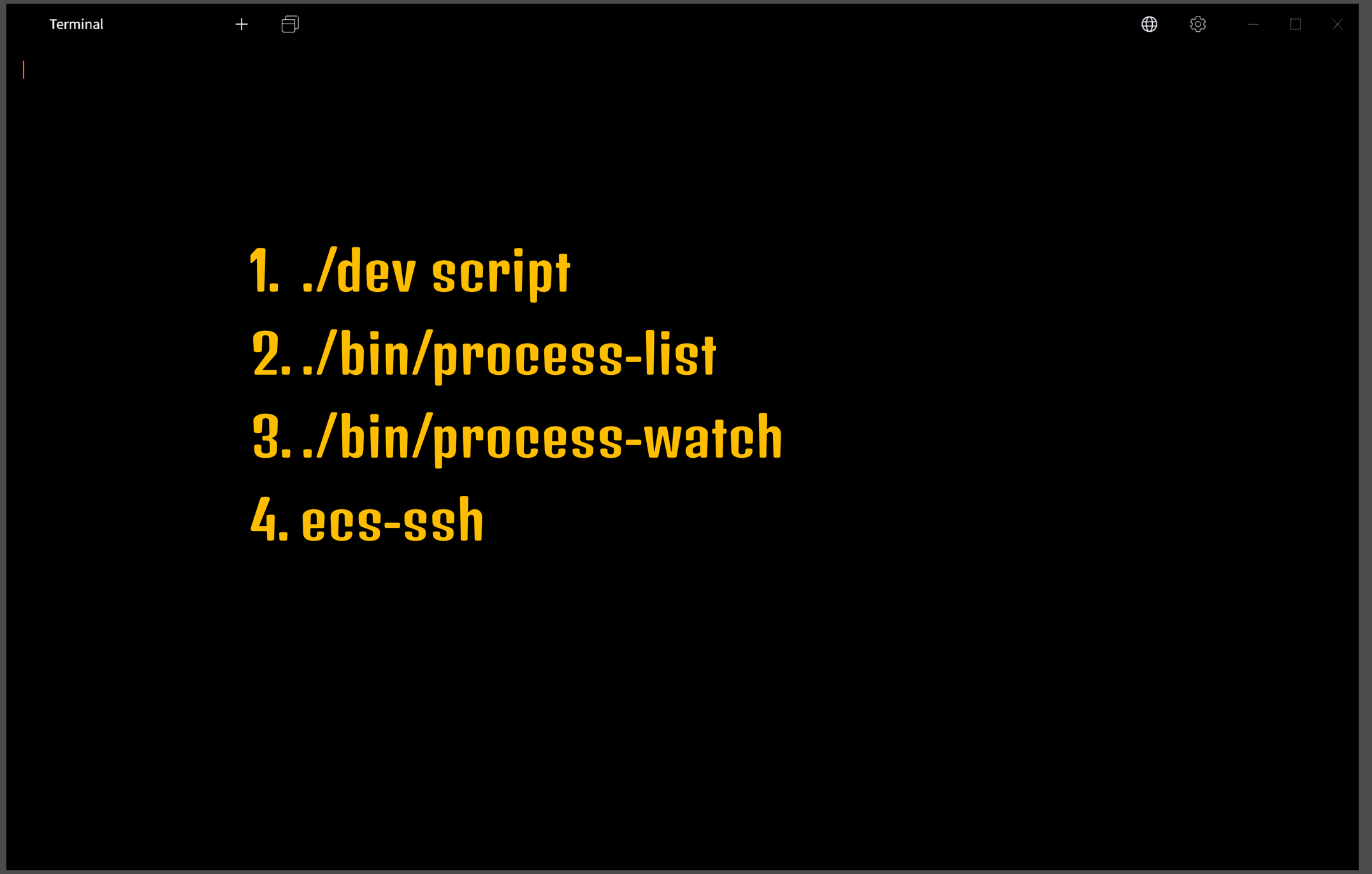
- It's a **Facade** to the Development Environment **most frequent operations and actions**
  - It's a way to automate things that can slow you down, such as:
    - db migrations
    - elasticsearch migrations
    - one-time rake tasks that need to run
    - bundle install & yarn install
    - node / nvm upgrades + ruby upgrades

WHY? WHY DO WE NEED DEV SCRIPT?

- We have conflicting methods of starting things and running tasks:
  - There are **RAKE** tasks
    - Being slowly replaced by the **RAILS** command
  - There is **YARN RUN**
    - There are **RAKE** tasks invoking **YARN**
  - There are **YARN** tasks invoking **RAKE & RAILS**
  - There are Ruby scripts (**scripts/dblab**)
  - How can anyone remember all of that?

HOW IT SAVES YOU TIME?

- dev's setup only runs migrations **if db/migrate folder has changed since the last time it ran**
- same about **elastic search migrations**
- upcoming features:
  - dev will read a configuration file **config/dev-run-once.yml**
  - for each task defined there, **it will run it and save task's SHA in a locally ignored file**
  - next time it will skip this task, but will run any new ones added to the YAML
  - no more announcing: run "**rake blah:blah:blah**" once


Terminal
1. ./dev script
2. ./bin/process-list
3. ./bin/process-watch
4. ecs-ssh

Foreman — Multi-process Ruby Launcher

Terminal

File: Procfile

```
# This is the Procfile used by the gem "foreman". The gem is not part
# of the bundle, and must be installed separately:
#
#       gem install foreman -N
#
# Alternatively, you can use the application boot script ./dev
# which installs foreman if not already installd.
#
#-------------------------------------------------------------------
#
# NOTE: we recommend that you start foreman via the ./dev script which also
#       ensures bundle install & yarn install + migrations are up to date.
#
#-------------------------------------------------------------------
#
# NOTE: Most commands below load shell environment from .env-procfile file.
#       The file contains reasonable defaults for running the application.
#       To override any of the variables, please place the overrides into
#       the file .env-procfile.local file, which is git-ignored.
#
# webpack (takes up to 4 minutes to build)
js:           bash -c "source .env-procfile; yarn run js-hot"
# web server, runs in a 0-worker mode (no clustering), but with 1-4 threads.
puma:         bash -c "source .env-procfile; bundle exec ${DATADOG_PROFILE_COMMAND} puma -C config/puma/development.rb --tag academia -v"
# sidekiq that uses a partial set of queues defined in config/sidekiq.yml
sidekiq:      bash -c "source .env-procfile; SIDEKIQ=y bundle exec sidekiq -C config/sidekiq.yml"
# tail the
log:          bash -c "source .env-procfile; print-academia-env; tail -f log/development.log"
```

Terminal

academia-app: **./dev -h**

```
EXAMPLES:
  dev                          # start everything in one window, and run setup
  dev sidekiq puma log         # start Ruby backend and show dev log
  dev js -s -i                 # start WebPack, skip setup, ignore existing processes
  dev -w js -i                 # start all but JS, but run setup

  dev show                     # Show application processes (if any)
  dev show -o js               # Show all JS processes (webpack/esbuild/etc)

  dev stop [ -o all ]          # kill application processes & exit
  dev stop -o ruby             # kill all Ruby processes (puma/sidekiq/etc)
  dev stop -o js               # kill all JS processes (webpack/esbuild/etc)
  dev stop -o spring           # kill Spring pre-loader and exit

  The two-terminal setup:

  Terminal 1: dev -w js        # same as ./dev sidekiq puma log
  Terminal 2: dev -i -s js     # This should be started second, as it skips
                               # the setup and ignores any running processes.
```

Terminal

academia-app: ./bin/process-list -h

```
USAGE:
  # general usage pattern
  bin/process-list [-][sort-column] [ -- [ ps args ] ]

  # sorting in ascending order, and descending order
  bin/process-list  [sort-column]
  bin/process-list -[sort-column]

  # print top 10 processes owned by ubuntu by CPU
  bin/process-list pcpu -- -u ubuntu | tail -10

SEE ALSO:
  bin/process-watch — like 'top', but uses process-list
                      and accepts the same arguments.

SORT COLUMNS:
  • rsz
  • pcpu
  • user
  • start_time
```

Terminal

academia-shell: **bin/ecs-ssh**

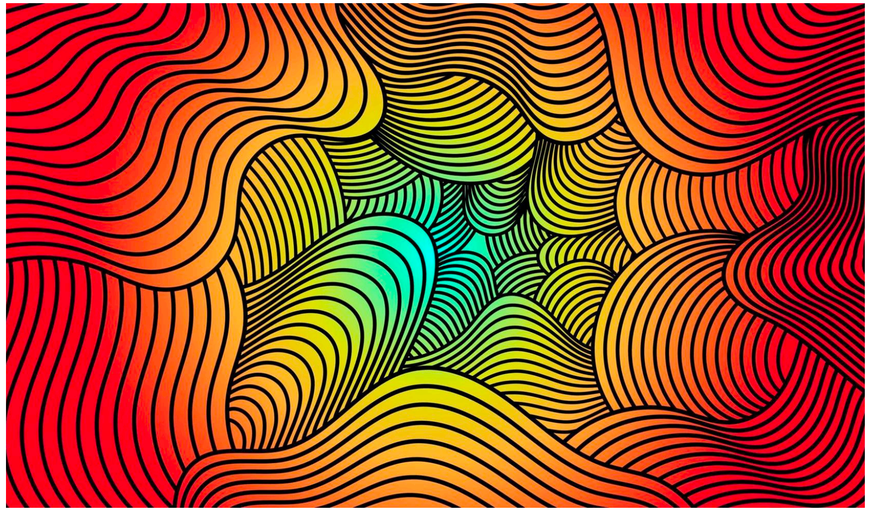
```
EXAMPLES:
    # Choose a production cluster and a service interactively
    ecs-ssh p

    # Choose production cluster that matches 'web', and a service
    # that matches 'admin' interactively
    ecs-ssh p -c web -s admin

    ecs-ssh p -c '^sidekiq' -s suppressed_ring_choose_treatment
    ecs-ssh q -c app-sidekiq -s free_ring

    # List all the sidekiq queues for the given environment
    ecs-ssh p queues
    ecs-ssh q queues
```


DEMO TIME

ASCII Cinema Recorded Demos for Academia

- Academia Shell: **bin/ecs-ssh**

  https://asciinema.org/a/DtoIxsX7p2xY3DukhM89F7t18



THANKS! QUESTIONS?