

way of the hax0r
n00b to console punk

Chapter 15

Earn Console Punk

Every chapter so far practiced one skill at a time. Real work does not arrive that way. It arrives as “the site was slow last week, find out why”, and the answer takes exploring a directory, reading files, building a pipeline, writing a script, fixing what breaks, and keeping the result. This chapter is that job, done end to end, in your own shell.

The project is a **log report tool**. You will generate three days of server logs, investigate them by hand, turn the investigation into a script, break the script three ways and repair it, and put the tool under version control. Both tracks are given in full; do the one for your main shell, then read the other.

The chapter is a checklist as much as a lesson. When you can do each step without looking back at earlier chapters, you have earned console punk.

Set Up the Project

Create a project directory, not inside any other repository, and start Git there:

```
$ mkdir console-punk
$ cd console-punk
$ git init
```

Every command that follows runs from this directory. Check with `pwd` or `Get-Location` before each section; Chapter 3’s habit applies to the whole project.

Generate the Logs

Rather than download sample data, write the generator yourself. It is a small script in the style of Chapter 11, and it gives you logs whose contents you understand.

Bash: save as `make-logs.sh`:

```
#!/usr/bin/env bash
set -euo pipefail
mkdir -p logs
for day in 18 19 20; do
```

```
f="logs/2026-09-$day.log"
: > "$f"
for i in $(seq 1 40); do
    stamp="2026-09-$day 10:$(printf '%02d' "$i"):00"
    case $((i % 7)) in
        0) echo "$stamp ERROR db: connection refused" >> "$f" ;;
        3) echo "$stamp WARN web: slow response 1200ms" >> "$f" ;;
        5) echo "$stamp ERROR web: 500 on /checkout" >> "$f" ;;
        *) echo "$stamp INFO web: 200 GET /index.html" >> "$f" ;;
    esac
done
done
```

: > "\$f" empties the file; : is a command that does nothing, and the redirection does the work. seq 1 40 prints the numbers, and \$((i % 7)) picks a line type by remainder.

```
$ chmod +x make-logs.sh
$ ./make-logs.sh
$ ls logs
2026-09-18.log 2026-09-19.log 2026-09-20.log
```

PowerShell: save as Make-Logs.ps1:

```
New-Item -ItemType Directory -Path logs -Force | Out-Null
foreach ($day in 18, 19, 20) {
    $lines = foreach ($i in 1..40) {
        $stamp = '2026-09-{0} 10:{1:d2}:00' -f $day, $i
        switch ($i % 7) {
            0 { "$stamp ERROR db: connection refused" }
            3 { "$stamp WARN web: slow response 1200ms" }
            5 { "$stamp ERROR web: 500 on /checkout" }
            default { "$stamp INFO web: 200 GET /index.html" }
        }
    }
    Set-Content -Path "logs/2026-09-$day.log" -Value $lines
}
```

The -f operator formats a string, with {1:d2} padding the minute to two digits. A foreach used as a value collects everything its body outputs, so \$lines holds forty strings.

```
$ ./Make-Logs.ps1
$ Get-ChildItem logs
```

Read one file with less or Get-Content before going on. Each line is a timestamp, a level, a component, and a message, separated by spaces.

Explore

Answer these questions with the tools from Chapters 3 and 6, and write down the command that answered each.

1. How many lines does each log have, and are they the same?
2. Which files contain the word ERROR?
3. What is the first error in the earliest log, with its line number?
4. Which components, db or web, appear in the logs?

Bash/zsh:

```
$ wc -l logs/*.log
$ grep -l ERROR logs/*.log
$ grep -n ERROR logs/2026-09-18.log | head -n 1
$ awk '{print $4}' logs/*.log | sort -u
```

PowerShell:

```
$ Get-ChildItem logs | ForEach-Object { "$($_.Name): $((Get-Content $_).Count)" }
$ Select-String -Path logs/*.log -Pattern ERROR -CaseSensitive -List | Select-Object Filename
$ Select-String -Path logs/2026-09-18.log -Pattern ERROR -CaseSensitive | Select-Object -First 1
$ Get-Content logs/*.log | ForEach-Object { ($_ -split '\s+')[3] } | Sort-Object -Unique
```

The fourth command uses `awk` and a regular-expression split rather than `cut -d ' '`, because the `WARN` and `INFO` lines have two spaces after the level and a single-space split would produce an empty field there. If a command did not do what you expected, that is the point of this section. Fix it before moving on, and note what was wrong.

Build the Pipeline

The report has two parts: errors per file, and the most common error messages across all files. Build each as a pipeline at the prompt, one stage at a time, as Chapter 7 showed.

Bash/zsh:

```
$ grep -c ERROR logs/*.log
logs/2026-09-18.log:11
logs/2026-09-19.log:11
logs/2026-09-20.log:11
$ grep -h ERROR logs/*.log | cut -d ' ' -f4- | sort | uniq -c | sort -rn
    18 web: 500 on /checkout
    15 db: connection refused
```

`grep -c` counts per file. The second pipeline drops filenames with `-h`, keeps the message from the fourth field onward with `cut -f4-`, and counts repeats with the `sort | uniq -c | sort -rn` pattern.

PowerShell:

```
$ Get-ChildItem logs | ForEach-Object { "$($_.Name): $((Select-String -Path $_ -Pattern ERROR -C))" }
$ Select-String -Path logs/*.log -Pattern ERROR -CaseSensitive | ForEach-Object { ($_.Line -split ' ', 4) }
```

`-split ' ', 4` splits into at most four pieces so that the message keeps its spaces, and `Group-Object` does what `sort | uniq -c` does, with objects.

Compare the two tracks' output. The numbers must match; the form differs because one pipeline carries text and the other carries objects.

Write the Script

Turn the pipelines into a tool that takes a directory, prints the report, and exits nonzero when there were errors, so that another script or a scheduler can act on it.

Bash: save as `report.sh`:

```
#!/usr/bin/env bash
set -euo pipefail

dir="${1:-logs}"
if [ ! -d "$dir" ]; then
    echo "usage: $0 [log-directory]" >&2
    exit 2
fi

total=0
for log in "$dir"/*.log; do
    count=$(grep -c ERROR "$log" || true)
    printf '%s: %s errors\n' "$(basename "$log")" "$count"
    total=$((total + count))
done

echo "most common errors:"
grep -h ERROR "$dir"/*.log | cut -d' ' -f4- | sort | uniq -c | sort -rn | head -n 3 || true

echo "total: $total"
[ "$total" -eq 0 ]
```

The `|| true` after `grep -c` matters: `grep` exits 1 when a file has no matches, and `set -e` would end the script there. The last line is the exit status: `[` succeeds when the total is zero and fails otherwise, and the script's status is that of its last command.

```
$ chmod +x report.sh
$ ./report.sh
2026-09-18.log: 11 errors
2026-09-19.log: 11 errors
2026-09-20.log: 11 errors
most common errors:
    18 web: 500 on /checkout
    15 db: connection refused
total: 33
$ echo $?
1
```

PowerShell: save as `Report.ps1`:

```

param(
    [string]$Directory = 'logs'
)

if (-not (Test-Path -LiteralPath $Directory -PathType Container)) {
    Write-Error "usage: Report.ps1 [-Directory log-directory]"
    exit 2
}
$ErrorActionPreference = 'Stop'

$logs = Get-ChildItem -LiteralPath $Directory -Filter *.log
$total = 0
foreach ($log in $logs) {
    $count = (Select-String -LiteralPath $log.FullName -Pattern 'ERROR' -CaseSensitive | Measure-
    '{0}: {1} errors' -f $log.Name, $count)
    $total += $count
}

'most common errors:'
$logs |
    Select-String -Pattern 'ERROR' -CaseSensitive |
    ForEach-Object { ($_.Line -split ' ', 4)[3] } |
    Group-Object |
    Sort-Object Count -Descending |
    Select-Object -First 3 Count, Name |
    Format-Table -HideTableHeaders

"total: $total"
if ($total -ne 0) { exit 1 }
exit 0

```

`$ErrorActionPreference = 'Stop'` comes after the usage check on purpose; the next section shows why. The final `exit 0` is also deliberate.

```

$ ./Report.ps1
$ $LASTEXITCODE
1

```

Break It and Fix It

A tool you have only seen working is a tool you do not understand. Each of these breaks is one you will see again; diagnose it with the chapter that covers it before reading the fix.

An Empty Directory

Make a directory with a log that has no errors and run the report on it:

```
$ mkdir quiet
$ printf '2026-09-21 10:00:00 INFO web: 200 GET /\n' > quiet/2026-09-21.log
$ ./report.sh quiet
```

Bash: without the `|| true` after the “most common errors” pipeline, the script prints the per-file line and then stops before `total`, with status 1. `grep -h` found nothing, exited 1, `pipefail` made the pipeline fail, and `set -e` ended the script. The version above already has the fix; remove the `|| true` to see the failure, then put it back. Decide for each command in a script whether “nothing found” is an error.

PowerShell: the script runs to the end, prints `total: 0`, and exits 0. Now delete the last line, `exit 0`, and run the report twice, first on logs and then on quiet:

```
$ ./Report.ps1
$ ./Report.ps1 -Directory quiet
$ $LASTEXITCODE
1
```

`$LASTEXITCODE` still says 1. A script that finishes without calling `exit` leaves `$LASTEXITCODE` as it was, so the value belongs to the previous run. Always end a script with an explicit `exit`, or check `$?` instead, which reports the last command’s success.

The Usage Error

Run the report on a directory that does not exist and check the status:

```
$ ./report.sh nowhere
usage: ./report.sh [log-directory]
$ echo $?
2
```

PowerShell: move `$ErrorActionPreference = 'Stop'` above the usage check and run `./Report.ps1 -Directory nowhere`. The message appears, but `$LASTEXITCODE` is 1, not 2, because `Write-Error` under a `Stop` preference is a terminating error and the script never reached `exit 2`. Chapter 11’s rule applies: decide which errors stop the script, and put the preference where it takes effect only after the checks you handle yourself.

The Wrong Line Endings

Copy the script through a Windows editor, or simulate it:

```
$ tr -d '\r' < report.sh | sed 's/$/\r/' > report-crlf.sh
$ chmod +x report-crlf.sh
$ ./report-crlf.sh
/usr/bin/env: 'bash\r': No such file or directory
```

The kernel read the shebang line up to the line feed and asked `env` for a program named `bash` followed by a carriage return. file `report-crlf.sh` confirms the CRLF endings, and `tr -d '\r'` from Chapter 5 repairs the file. On Windows, PowerShell reads both endings, which is why the PowerShell track has no version of this break.

The Missing Permission

```
$ chmod -x report.sh
$ ./report.sh
bash: ./report.sh: Permission denied
$ bash report.sh
```

Chapter 8: running a file directly needs the execute bit, and handing it to the interpreter does not. Restore the bit with `chmod +x report.sh`. The PowerShell equivalent on Windows is the execution policy; if `./Report.ps1` reports that scripts are disabled, `Set-ExecutionPolicy -Scope CurrentUser RemoteSigned` is the fix, and Chapter 8 says why.

Keep It

The tool works and you understand its failure modes. Record it.

```
$ printf 'logs/\nquiet/\n*.log\n' > .gitignore
$ git status --short
$ git add .gitignore make-logs.sh report.sh
$ git commit -m 'Add log report tool'
```

PowerShell users add `Make-Logs.ps1` and `Report.ps1` instead. The logs are generated, so they stay out of the repository; the generator is the thing worth keeping. `git status` before the commit shows that `report-crlf.sh` is untracked, and it should stay that way: delete it.

Now improve the tool, and commit the improvement separately. Add a count of WARN lines to the per-file report, in whichever track you are using, and run it. Then:

```
$ git diff
$ git add report.sh
$ git commit -m 'Report warnings alongside errors'
$ git log --oneline
```

Finally, break the script with a bad edit, confirm it fails, and use `git restore report.sh` to get the working version back. That round trip, change, check, commit, and recover, is the whole reason Chapter 14 exists.

Take It Further

Each of these uses one more chapter. Do any that interest you.

- **Across machines** (Chapter 12): copy the project to another machine with `scp` or `rsync`, run the report there over `ssh`, and bring the output back. Then run the generator inside a `tmux` session and detach while it works.
- **Across systems** (Chapter 13): run the Bash version in WSL on the logs the PowerShell version generated, and the other way round. Package the project with `tar` and publish a `SHA256SUMS` next to it.
- **Made yours** (Chapter 10): add an alias or function so that `report` runs the tool from anywhere, and a prompt that shows the Git branch.

- **Automated** (Chapter 9): run the report on a schedule with cron or a Windows scheduled task, in the background, writing to a log of its own.

You Are a Console Punk When

- You open a terminal on any of the three systems and know which shell answered.
- You read a command from a tutorial and can say what each part is, before running it.
- You know where you are, what is there, and how to move, create, copy, and remove without a file manager.
- You edit a file in the terminal and fix an encoding or line-ending problem when it appears.
- You find a file by name or by content in a tree of thousands.
- You connect commands with pipes and redirection, and read an exit status.
- You read “Permission denied” as information and fix the cause.
- You stop a program that will not stop, and keep one running when you leave.
- Your shell starts the way you want it to, and you can recover it when it does not.
- You turn a working pipeline into a script with arguments and proper failures.
- You reach another machine, move files both ways, and troubleshoot the connection in order.
- Your work is in Git, your secrets are not, and you can get last week’s version back.

That is the milestone. The next book in the series, *console punk to hacker & modder*, starts from here.

Exercises

1. **What does this do?** Predict the output of `grep -c ERROR logs/*.log | cut -d: -f2 | paste -sd+ | bc`, then run it and explain each stage.
2. **What does this do?** Predict what `./report.sh logs quiet` does, given the script as written, and decide whether that is a bug.
3. **What is the problem?** A teammate reports that `./report.sh` prints `total: 0` on a directory that clearly contains errors. Their logs use the word `Error`. Explain and decide whether to change the script or the logs.
4. **What is the problem?** After `git add .`, `git status` shows `logs/` as staged even though `.gitignore` lists it. Give two possible causes and the command that tells them apart.
5. **Why is it like this?** The Bash script needs `|| true` in two places and the PowerShell script needs none, yet the PowerShell script needs an explicit `exit 0`. Explain what each shell counts as failure.
6. **Why is it like this?** The PowerShell pipeline uses `-split ' ', 4` and the Bash pipeline uses `cut -f4-`. Explain why both need to say “four”.
7. **How do you** extend the report to show the busiest hour across all logs, in both tracks?
8. **How do you** confirm, without reading the script, that `report.sh` and `Report.ps1` produce the same counts on the same logs?

References