✨ Member-only story

Shell Script Best Practices

Neuro Bytes · Follow
Published in CodeCuriosity
5 min read · Oct 27, 2024

Listen Share More

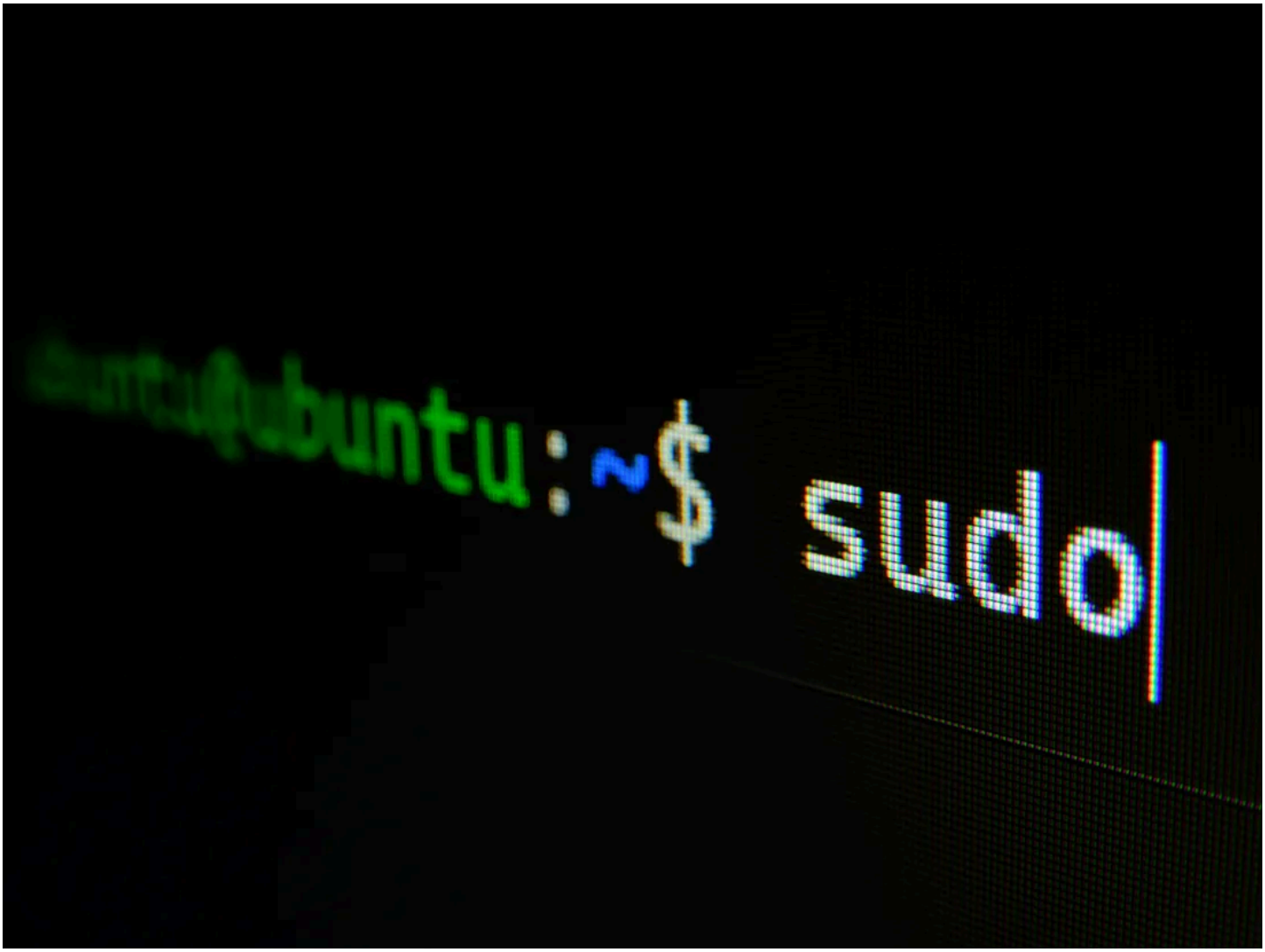

Photo by Gabriel Heinzer on Unsplash

Shell scripting is a powerful tool for automating tasks, managing system configurations, and processing data. But with power comes responsibility — shell scripts can quickly become complicated, error-prone, and difficult to maintain if they aren't written carefully. Here, we'll dive into best practices that can make your shell scripts not only functional but also efficient, reliable, and easy to understand. Whether you're a beginner or a seasoned pro, these principles will help you elevate your scripting game.

1. Choose the Right Shell

The first step in any shell scripting project is to select the right shell environment. For most scripts, Bash is a solid choice due to its robust features and wide compatibility. However, certain systems or use cases might require a different shell like Dash, Zsh, or even KornShell (ksh).

Use the shebang (`#!/bin/bash` or `#!/bin/sh`) to specify your shell at the top of each script, ensuring consistent behavior across environments. This can prevent surprises later on, especially when scripts are run in different operating systems.

```
#!/bin/bash
```

2. Comment Your Code

Good scripts are self-documenting, and comments are a key part of that. Start each script with a brief description of its purpose, parameters, and any dependencies. Use inline comments to explain non-obvious code sections, especially any complicated logic or external dependencies. Well-placed comments will save you (and others) time when revisiting the code.

```
#!/bin/bash
# This script backs up the specified directory to a remote server.
backup_directory="/path/to/directory"
```

3. Use Meaningful Variable Names

Readable code makes for maintainable code. Avoid cryptic one-letter variable names — use descriptive names that make the code self-explanatory. Instead of `f` or `i`, try `file` or `index` to make the purpose of each variable clear.

```
#!/bin/bash
source_dir="/home/user/documents"
backup_dir="/backup/documents"
```

4. Error Handling and Debugging

Errors happen. Anticipate them by using techniques like `set -e` to exit the script on any error. Also, consider `set -u` to catch uninitialized variables and `set -o pipefail` to detect errors in pipelines.

For debugging, `set -x` is incredibly useful. It will print each command before executing it, making it easy to identify where things go wrong. For production, turn off `set -x` to avoid verbose output.

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

5. Quoting Variables to Prevent Word Splitting

One common source of bugs in shell scripts is unquoted variables. Unquoted variables can lead to word splitting and unexpected behavior, especially with strings containing spaces or special characters. Make it a habit to enclose variables in double quotes.

```
#!/bin/bash
filename="my file.txt"
echo "$filename"   # Correct
echo $filename     # Incorrect - leads to word splitting
```

6. Test for File and Directory Existence

Checking for the existence of files or directories is critical, especially for scripts that rely on certain resources. Use conditional tests (`-f` for files, `-d` for directories) to verify that resources are available before attempting to use them.

```
#!/bin/bash
# Check if the file exists before attempting to copy it.
file_path="/path/to/file.txt"
if [[ -f "$file_path" ]]; then
    echo "File exists, proceeding with backup."
else
    echo "File does not exist."
fi
```

7. Avoid Using Hard-Coded Paths

Scripts often depend on files, directories, or executables, and hard-coding paths can cause issues when running scripts on different machines. Use variables for paths and, when possible, avoid absolute paths by using relative ones or environment variables.

```
#!/bin/bash
log_dir="${HOME}/logs"
mkdir -p "$log_dir"
```

8. Validate User Input

If your script accepts user input or arguments, validate them. For example, if an argument expects a directory, check if the path is valid. Validating inputs early helps prevent unexpected crashes and errors.

```
#!/bin/bash
# Check if the argument is a directory.
if [[ ! -d "$1" ]]; then
    echo "Error: $1 is not a directory."
    exit 1
fi
```

9. Use Functions to Structure Code

As scripts grow, keeping code organized becomes essential. Functions allow you to break down tasks into reusable, self-contained units. This not only improves readability but also allows you to reuse code and make updates more efficiently.

```
#!/bin/bash
backup_file() {
    local file="$1"
    echo "Backing up $file..."
    # Backup code here
}
```

10. Follow POSIX Standards When Possible

If your script might be used on different Unix-based systems, aim to follow POSIX standards. POSIX-compliant scripts are more portable and likely to work across different shells. Avoiding non-standard Bash features can ensure greater compatibility with other shells like `sh` or `dash`.

11. Use `getopts` for Command-Line Options

Scripts with multiple options can quickly become unmanageable if arguments are parsed manually. Using `getopts` enables you to handle command-line options cleanly and makes it easier for users to understand and use your script.

```
#!/bin/bash
while getopts ":u:p:" opt; do
    case $opt in
        u) user="$OPTARG" ;;
        p) password="$OPTARG" ;;
        \?) echo "Invalid option: -$OPTARG" >&2 ;;
    esac
done
```

12. Log Outputs and Errors

For long-running or critical scripts, logging progress and errors is essential for tracking performance and troubleshooting. Use `>>` to append outputs to a log file, and consider redirecting errors to a separate file or log them with `2>>`. Timestamped logs add an additional layer of context for diagnostics.

```
#!/bin/bash
log_file="/var/log/backup.log"
echo "$(date) - Starting backup process" >> "$log_file"
```

13. Optimize Performance for Long-Running Scripts

When writing scripts that handle large data sets or perform repetitive tasks, be mindful of performance. Use built-in shell commands instead of external commands whenever possible (e.g., using `[[ ]]` for string comparison instead of `test`). For loops, avoid `cat` where possible by redirecting files into `while` loops.

```
#!/bin/bash
# Avoid: cat largefile | while read -r line; do ...
while read -r line; do
    # Process line
done < largefile
```

14. Exit Gracefully

Make sure your script exits gracefully, particularly in error scenarios. Use meaningful exit codes to indicate success (`0`) or specific errors (`1` , `2` , etc.). Exiting gracefully helps downstream scripts or users understand whether the process completed successfully or encountered an issue.

```
#!/bin/bash
echo "Running script..."
# Exit with error code 1 if something goes wrong
exit 1
```

Conclusion

You can create scripts that are not only functional but also robust and reliable by keeping readability, structure, and error handling in mind of course.

Follow our publication: