# Shell Scripting Profissional

terça-feira, 16 de dezembro de 2025 20:15

📘 **— Shell Scripting Profissional**

***Automação, lógica e resolução de problemas no mundo real***

> ⚠ **não é "bash para iniciantes".**
> Ele foi escrito para alguém que já usa Linux, mas quer **pensar como DevOps** e **automatizar com elegância e segurança**.

**3.1 — O papel do Shell Script no DevOps**

Antes de escrever código, precisamos alinhar o **porquê**.

Shell scripts são usados para:

- automatizar tarefas repetitivas
- padronizar operações
- reagir a falhas
- integrar ferramentas
- criar "cola" entre sistemas
- reduzir erro humano
- garantir previsibilidade

📌 **Em DevOps, tudo que é feito duas vezes deveria ser automatizado.**

**3.2 — Estrutura profissional de um script**

Todo script profissional começa assim:

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

Isso diz ao sistema:

> "Execute este script usando o interpretador bash."

Depois disso, seguem **boas práticas essenciais**:

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

Vamos entender isso com calma:

- set -e → o script **para** se ocorrer erro
- set -u → erro se variável não existir
- set -o pipefail → falha se qualquer comando em um pipe falhar

☞ Isso evita scripts "mentirosos", que parecem rodar mas fazem besteira por trás.

**3.3 — Variáveis: base da automação**

Variáveis armazenam valores reutilizáveis.

```
SERVICO="nginx"
LOG="/var/log/monitor.log"
```

Uso correto:

```
echo "Monitorando serviço: $SERVICO"
```

❌ **Erro comum:**

```
echo 'Monitorando serviço: $SERVICO'
```

Aspas simples **não expandem variáveis**.

**3.4 — Entrada de parâmetros**

Scripts profissionais aceitam argumentos:

```
./monitor.sh nginx
```

Dentro do script:

```
SERVICO="$1"
```

Verificação defensiva:

```
if [ -z "$SERVICO" ]; then   # -z → string vazia, usuário não passou argumento
  echo "Uso: $0 <serviço>"    # $0 → nome do script e a forma correta de uso
  exit 1   # Encerra o script imediatamente, 1 → indica erro
fi
```

☞ Isso é maturidade técnica. Ele garante que o script **só continue se o usuário informar o nome de um serviço** ao

executar o script.

**3.5 — Condições (if / else / case)**
**If simples:**

```
if systemctl is-active --quiet nginx; then
 echo "Serviço ativo"
else
 echo "Serviço parado"
fi
```

**Case (mais limpo para múltiplas opções):**

```
case "$1" in
 start) systemctl start nginx ;;
 stop) systemctl stop nginx ;;
 restart) systemctl restart nginx ;;
 *) echo "Uso: $0 {start|stop|restart}" ;;
esac
```

📌 Use case quando houver múltiplas ações.

**3.6 — Loops: automatizando em escala**
**For:**

```
for usuario in $(cut -d: -f1 /etc/passwd); do
 echo "Usuário: $usuario"
Done
```

**While (muito usado para monitoramento):**

```
while true; do
 date
 sleep 5
Done
```

**Exemplo prático simples (monitoramento)**
**Ver se um serviço está ativo**

```
while true; do
    systemctl is-active nginx sleep 5
done
```

☞ Loops mal-usados geram **consumo de CPU**. Sempre controle com sleep.

**3.7 — Funções: scripts organizados**
Scripts grandes **sem funções** viram bagunça.

```
verificar_servico() {
 systemctl is-active --quiet "$1"
}
```

Uso:

```
if verificar_servico nginx; then
 echo "OK"
else
 echo "FALHA"
fi
```

📌 Funções tornam scripts:

- reutilizáveis
- legíveis
- testáveis

**3.8 — Redirecionamento e pipes (nível profissional)**
**Redirecionar saída:**

```
echo "Erro" >> erro.log
```

**Redirecionar erro:**

```
comando 2>> erro.log
```

**Redirecionar tudo:**

```
comando &>> tudo.log
```

| Operador | O que faz |
|---|---|
| Comando **&>>** arquivo | Redireciona **stdout + stderr** e **faz append** |
| Comando **2>&1** >> arquivo | Redireciona **stderr para stdout**, depois stdout para arquivo, erro aparece na tela. |
| Comando >> arquivo **2>&1** | Redireciona **stdout para arquivo** e depois **stderr para o mesmo lugar** |

```
root@dbn13:~# ls arquivo_inexistente >> log.txt 2>&1
root@dbn13:~# ls arquivo_inexistente 2>&1 >> log.txt
ls: não foi possível acessar 'arquivo_inexistente': Arquivo ou diretório inexistente
root@dbn13:~#
```

**Pipes:**

```
ps aux | grep nginx | grep -v grep
```

☞ Em DevOps, **pipes são essenciais** para diagnóstico.

**3.9 — Script Profissional: Monitor de Serviço (completo)**

Agora vamos unir tudo.

**Objetivo:**

- verificar serviço
- registrar log
- reiniciar automaticamente
- ser reutilizável

**monitor.sh**

```
#!/bin/bash
set -euo pipefail
SERVICO="$1"
LOG="/var/log/monitor_${SERVICO}.log"
if [ -z "$SERVICO" ]; then
 echo "Uso: $0 <serviço>"
 exit 1
fi
if systemctl is-active --quiet "$SERVICO"; then
 echo "$(date) - $SERVICO ativo" >> "$LOG"
else
 echo "$(date) - $SERVICO INATIVO. Reiniciando..." >> "$LOG"
 systemctl restart "$SERVICO"
 echo "$(date) - $SERVICO reiniciado" >> "$LOG"
fi
```

Teste:

```
chmod +x monitor.sh
sudo ./monitor.sh nginx
```

**3.10 — Debugging de scripts (habilidade rara)**

Modo debug:

bash -x script.sh

Isso mostra:

- comandos executados
- valores de variáveis
- onde o script quebra

**3.11 — Erros comuns (e como resolver)**

| Erro | Causa |
|---|---|
| Permission denied | falta chmod +x |
| command not found | PATH errado |
| variável vazia | falta validação |
| loop infinito | while sem sleep |

**3.12 — Exercícios obrigatórios**

1. Adaptar o script para múltiplos serviços
2. Criar log por serviço
3. Criar modo --check sem reiniciar
4. Criar documentação README

**3.13 —** 🎁 Repositório GitHub:

```
shell-devops/
 ├── monitor.sh
 ├── log-cleaner.sh
 ├── README.md
 └── docs/
```

README deve explicar:

- objetivo
- funcionamento
- exemplos
- erros resolvidos

**3.14 — O que você acabou de aprender**

✓ Pensar em automação
✓ Escrever scripts defensivos
✓ Diagnosticar falhas
✓ Criar código reutilizável
✓ Documentar soluções

Isso é **DevOps de verdade**.

📘 **Exemplos de Shell Script em Situações REAIS de DevOps** ***Onde isso é usado no mundo profissional***

🔥 **CENÁRIO 1 — SERVIÇO CAI INTERMITENTEMENTE**

📌 **Problema real**

- Aplicação cai aleatoriamente
- Não há monitoramento sofisticado ainda
- Time precisa reagir rápido

🎯 **Objetivo do script**

- Verificar serviço
- Reiniciar automaticamente
- Gerar histórico de falhas

📄 **Script: Monitoramento com contagem de falhas**

```
#!/bin/bash
set -euo pipefail
SERVICO="nginx"
LOG="/var/log/${SERVICO}_monitor.log"
if ! systemctl is-active --quiet "$SERVICO"; then
   echo "$(date) - $SERVICO caiu" >> "$LOG"
   systemctl restart "$SERVICO"
   echo "$(date) - $SERVICO reiniciado" >> "$LOG"
else
   echo "$(date) - $SERVICO OK" >> "$LOG"
fi
```

💡 **O que isso mostra em DevOps**

✓ Automação reativa
✓ Conhecimento de systemd
✓ Logging
✓ Capacidade de resposta rápida

## 🔥 CENÁRIO 2 — DISCO LOTANDO (problema CLÁSSICO)

### 📌 Problema real

- Servidor cai porque /var enche
- Logs gigantes
- Ninguém percebeu a tempo

### 🎯 Objetivo do script

- Verificar uso de disco
- Alertar se ultrapassar limite
- Limpar logs antigos

### 📄 Script: Alerta de uso de disco

```
#!/bin/bash
LIMITE=80
USO=$(df / | tail -1 | awk '{print $5}' | sed 's/%//')
        if [ "$USO" -gt "$LIMITE" ]; then
          echo "ALERTA: Disco acima de ${LIMITE}% (${USO}%)"
        fi
```

### 💡 O que isso mostra

✓ Monitoramento básico
✓ Prevenção de incidentes
✓ Uso de find, awk, sed

### 📄 Script: Limpeza de logs antigos

```
find /var/log -type f -mtime +7 -exec gzip {} \;
```

## 🔥 CENÁRIO 3 — BACKUP NÃO CONFIÁVEL

### 📌 Problema real

- Backup “rodava”, mas ninguém testava
- Quando precisou, estava quebrado

### 🎯 Objetivo

- Criar backup
- Verificar sucesso
- Manter histórico

### 📄 Script: Backup defensivo

```
#!/bin/bash
set -e
ORIGEM="/etc"
DESTINO="/backup/etc_$(date +%F).tar.gz"
tar -czf "$DESTINO" "$ORIGEM"
if [ $? -eq 0 ]; then
  echo "$(date) - Backup OK: $DESTINO"
else
  echo "$(date) - ERRO no backup"
fi
```

### 💡 Em DevOps

✓ Automação crítica
✓ Validação de resultado
✓ Segurança operacional

## 🔥 CENÁRIO 4 — VARIÁVEIS DE AMBIENTE QUEBRADAS

### 📌 Problema real

- Aplicação não sobe
- Falta variável de ambiente
- Erro confuso

### 🎯 Objetivo

- Validar ambiente antes de rodar app

### 📄 Script: Validador de ambiente

```
#!/bin/bash
VARIAVEIS=("DB_HOST" "DB_USER" "DB_PASS")
        for var in "${VARIAVEIS[@]}"; do
          if [ -z "${!var:-}" ]; then
            echo "Erro: variável $var não definida"
            exit 1
          fi
        done
echo "Ambiente OK"
```

### 💡 Isso é MUITO valorizado

✓ Script defensivo
✓ Prevenção de falhas
✓ Maturidade técnica

## 🔥 CENÁRIO 5 — CONTAINER NÃO SOBE

### 📌 Problema real

- Docker container em loop
- Logs não claros
- Pipeline quebrada

### 🎯 Objetivo

- Inspecionar container
- Coletar logs automaticamente

**Script: Debug Docker**

```
#!/bin/bash
CONTAINER="$1"
    docker ps -a | grep "$CONTAINER"
    docker logs --tail 50 "$CONTAINER"
```

**Em DevOps**

✓ Integração shell + Docker
✓ Troubleshooting rápido
✓ Diagnóstico automatizado

**CENÁRIO 6 — SERVIÇOS EM VÁRIOS SERVIDORES**

**Problema real**

- Ambiente com múltiplas VMs
- Verificar status manualmente é inviável

**Objetivo**

- Checar serviços via SSH

**Script: Verificação remota**

```
#!/bin/bash
SERVIDORES=("srv1" "srv2" "srv3")
    for srv in "${SERVIDORES[@]}"; do
     echo "Verificando $srv"
     ssh "$srv" systemctl is-active nginx
    done
```

**Em DevOps**

✓ Automação distribuída
✓ Conhecimento de SSH
✓ Escalabilidade operacional

**CENÁRIO 7 — DEPLOY MANUAL DÁ ERRO**

**Problema real**

- Deploy manual quebra ambiente
- Erros humanos frequentes

**Objetivo**

- Padronizar deploy

**Script: Deploy simples**

```
#!/bin/bash
set -e
    git pull
    docker compose down
    docker compose up -d --build
```

**Isso é CI/CD "raiz"**

✓ Padronização
✓ Redução de erro humano
✓ Automação de deploy

**O QUE ESSES EXEMPLOS TÊM EM COMUM**

Todos resolvem:

- falhas reais
- dores comuns
- problemas recorrentes
- tarefas manuais perigosas

E todos mostram:

✓ raciocínio
✓ maturidade
✓ visão de produção

**PORTFÓLIO (muito importante)**

Crie repositórios como:

```
devops-shell-labs/
 ├── disk-monitor/
 ├── service-monitor/
 ├── backup/
 ├── docker-debug/
 └── README.md
```

Cada pasta:

- problema
- solução
- script
- lições aprendidas