Bash Scripts - Detailed Code Explanation

Table of Contents

Basic Script Structure

bash

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

Shebang line - Tells the system to use bash to execute this script. Must be the first line.

Variables and Configuration

bash

```bash
LOG_DIR="/var/log"
DAYS_OLD=30
DATE=$(date +%Y%m%d_%H%M%S)
```

Variable assignment - No spaces around `=` sign

- `LOG_DIR` stores a string path
- `DAYS_OLD` stores a number
- `DATE` stores command output using `$(command)` syntax
- Access variables with `$VARIABLE_NAME` or `${VARIABLE_NAME}`

Date format breakdown:

- `%Y` = 4-digit year (2024)
- `%m` = month (01-12)
- `%d` = day (01-31)
- `%H` = hour (00-23)
- `%M` = minute (00-59)
- `%S` = second (00-59)

Conditionals

1. Checking User Permissions

```bash
if [ "$EUID" -ne 0 ]; then
    echo "Please run as root (use sudo)"
    return 1
fi
```

Breakdown:

- `if [ condition ]; then` - Start of conditional
- `$EUID` - Environment variable for user ID (0 = root)
- `-ne` - "not equal to" operator
- `return 1` - Exit function with error code (non-zero = failure)
- `fi` - End of if statement

Common comparison operators:

- `-eq` - equal to
- `-ne` - not equal to
- `-gt` - greater than
- `-lt` - less than
- `-ge` - greater than or equal

- `-le` - less than or equal

2. Checking Directory Existence

```bash
if [ ! -d "$BACKUP_DIR" ]; then
    mkdir -p "$BACKUP_DIR"
fi
```

Breakdown:

- `!` - NOT operator (negation)
- `-d` - Tests if path is a directory
- `mkdir -p` - Create directory and parent directories if needed

Common file test operators:

- `-f` - file exists and is regular file
- `-d` - directory exists
- `-e` - path exists (file or directory)
- `-r` - file is readable
- `-w` - file is writable
- `-x` - file is executable

3. String Comparison

```bash
if command -v apt-get &> /dev/null; then
    echo "apt-get found"
fi
```

Breakdown:

- `command -v` - Checks if command exists
- `&> /dev/null` - Redirects both stdout and stderr to null (suppress output)
- Returns 0 if command exists, non-zero otherwise

Loops

1. While Loop (Menu System)

```bash
while true; do
    show_menu
    read -p "Select option: " choice
    case $choice in
        1) cleanup_logs ;;
        *) echo "Invalid" ;;
    esac
done
```

Breakdown:

- `while true; do` - Infinite loop (true is always true)
- `read -p "prompt" variable` - Read user input with prompt
- `case` - Multi-way branch (like switch statement)
- `;;` - End of case branch
- `*` - Default case (matches anything)
- `esac` - End of case statement

2. For Loop (Array Iteration)

```bash
SERVICES=("ssh" "cron" "nginx")
for service in "${SERVICES[@]}"; do
    echo "Checking $service"
done
```

Breakdown:

- `("item1" "item2")` - Array syntax
- `"${SERVICES[@]}"` - Expands to all array elements
- `for item in list; do` - Iterate over each item

- `done` - End of loop

3. While Read Loop (Processing Lines)

bash

```bash
df -h | grep -vE '^Filesystem' | while read output; do
  usage=$(echo $output | awk '{print $1}')
  echo "Usage: $usage"
done
```

Breakdown:

- `|` - Pipe operator (pass output to next command)
- `grep -vE` - Exclude lines matching pattern
- `while read variable; do` - Read line by line
- `awk '{print $1}'` - Print first field/column

Functions

bash

```bash
backup_data() {
  echo "Starting backup..."
  local SOURCE_DIR="$HOME/Documents"

  if [ ! -d "$SOURCE_DIR" ]; then
    return 1
  fi

  tar -czf "backup.tar.gz" "$SOURCE_DIR"
  return 0
}
```

Breakdown:

- `function_name() { ... }` - Function definition
- `local` - Variable only exists within function
- `return 0` - Success (exit code 0)

- `return 1` - Failure (non-zero exit code)

Calling functions:

```bash
backup_data                   # Call function
if backup_data; then          # Call and check success
    echo "Backup succeeded"
fi
```

Command Execution

1. Find Command

```bash
find "$LOG_DIR" -name "*.log" -type f -mtime +30 -print -delete
```

Breakdown:

- `find path` - Search starting from path
- `-name "*.log"` - Match files ending in .log
- `-type f` - Only files (not directories)
- `-mtime +30` - Modified more than 30 days ago
- `-print` - Display found files
- `-delete` - Delete found files

2. Tar Command (Compression)

```bash
tar -czf "backup.tar.gz" -C "/path" "folder"
```

Breakdown:

- `-c` - Create archive
- `-z` - Compress with gzip

- `-f` - Specify filename
- `-C` - Change to directory first
- Result: compressed backup file

Extract:

```bash
tar -xzf "backup.tar.gz"
```

- `-x` - Extract archive

3. Disk Usage (df and du)

```bash
df -h                    # Show disk space (human readable)
du -h "file"             # Show file size
```

Breakdown:

- `-h` - Human-readable format (KB, MB, GB)
- `df` - Disk free space
- `du` - Disk usage

4. Package Management

```bash
apt-get update              # Update package lists
apt-get upgrade -y          # Upgrade packages (-y = yes to all)
apt-get autoremove -y       # Remove unused packages
apt-get clean               # Clear package cache
```

5. Service Management

```bash
```

```
systemctl is-active service    # Check if service running
systemctl start service        # Start service
systemctl stop service         # Stop service
systemctl restart service      # Restart service
systemctl status service       # Detailed status
```

Error Handling

1. Exit Codes

```bash
command
if [ $? -eq 0 ]; then
    echo "Success"
else
    echo "Failed"
fi
```

Breakdown:

- `$?` - Exit code of last command
- `0` - Success
- Non-zero - Error

2. Conditional Execution

```bash
command && echo "Success"    # Run if command succeeds
command || echo "Failed"     # Run if command fails
```

3. Error Suppression

```bash
command 2>/dev/null          # Suppress error messages
command &>/dev/null          # Suppress all output
command 2>&1                 # Redirect stderr to stdout
```

Breakdown:

- `2>` - Redirect stderr (error output)
- `&>` - Redirect both stdout and stderr
- `/dev/null` - Discard output

4. Set Error Options

bash

```bash
set -e              # Exit on any error
set -u              # Exit on undefined variable
set -x              # Print commands before executing
```

Practical Examples

Example 1: Simple Backup with Error Checking

bash

```
#!/bin/bash

backup_folder() {
    local src="$1"
    local dest="$2"

    # Check source exists
    if [ ! -d "$src" ]; then
        echo "Error: Source $src not found"
        return 1
    fi

    # Create destination
    mkdir -p "$dest" || return 1

    # Perform backup
    echo "Backing up $src to $dest..."
    tar -czf "$dest/backup_$(date +%Y%m%d).tar.gz" "$src"

    if [ $? -eq 0 ]; then
        echo "Backup completed successfully"
        return 0
    else
        echo "Backup failed"
        return 1
    fi
}

# Usage
backup_folder "$HOME/Documents" "$HOME/Backups"
```

Example 2: Loop Through Files

```
bash
```

```bash
#!/bin/bash

for file in /var/log/*.log; do
    if [ -f "$file" ]; then
        size=$(du -h "$file" | cut -f1)
        echo "File: $(basename "$file") - Size: $size"
    fi
done
```

Example 3: Interactive Script

bash

```bash
#!/bin/bash

read -p "Enter your name: " name
read -sp "Enter password: " password
echo ""

if [ -z "$name" ]; then
    echo "Name cannot be empty"
    exit 1
fi

echo "Hello, $name!"
```

Breakdown:

- `-p` - Prompt text
- `-s` - Silent mode (for passwords)
- `-z` - Test if string is empty

Best Practices

1. **Always quote variables:** `"$VAR"` prevents word splitting
2. **Use meaningful variable names:** `BACKUP_DIR` not `bd`
3. **Check for errors:** Test command success with `$?`
4. **Comment your code:** Explain complex logic

5. **Make scripts executable:** `chmod +x script.sh`
6. **Use functions:** Break code into reusable pieces
7. **Validate input:** Check user input and file existence
8. **Use absolute paths:** `/usr/bin/command` instead of `command`

Quick Reference

Task	Command
Make executable	`chmod +x script.sh`
Run script	`./script.sh` or `bash script.sh`
Run as root	`sudo ./script.sh`
Debug script	`bash -x script.sh`
Check syntax	`bash -n script.sh`

Common Pitfalls

1. **Spaces around** `=` ❌ `VAR = "value"` ✅ `VAR="value"`
2. **Unquoted variables** ❌ `if [ $var = "" ]` ✅ `if [ "$var" = "" ]`
3. **Forgetting shebang** ❌ Missing `#!/bin/bash`
4. **Wrong file permissions** ❌ File not executable
5. **Not checking errors** ❌ Assuming commands always succeed