KINGsNOTES

K19G

2024-12-14

Table of contents

Preface

Welcome to this book!

Structure

This book is organized into categories and subcategories to help you navigate the content effectively.

How to Use This Book

You can read this book in various formats:

- Online HTML version
- Downloadable PDF
- EPUB for e-readers

Prerequisites

List any prerequisites or requirements here.

Acknowledgments

Add acknowledgments here.

Commands

01_FileOps

chmod

Overview

The `chmod` command changes file and directory permissions. Essential for security and access control.

Syntax

```
chmod [options] mode file(s)
```

Permission Modes

Symbolic Mode

- `u`: User/owner
- `g`: Group
- `o`: Others
- `a`: All

Operations

- `+`: Add permission
- `-`: Remove permission
- `=`: Set exact permission

Permissions

- `r`: Read (4)
- `w`: Write (2)
- `x`: Execute (1)

Common Options

- `-R`: Recursive
- `-v`: Verbose
- `-f`: Force
- `-c`: Report changes

Examples

```
# Give execute permission to owner
chmod u+x script.sh

# Remove write permission from others
chmod o-w file.txt

# Set full permissions for owner only
chmod 700 private.key

# Add execute for all
chmod a+x program

# Set complex permissions
chmod u=rwx,g=rx,o= file

# Recursive change
chmod -R 755 directory/
```

Common Permission Patterns

- `chmod 777`: Full access for all (dangerous)
- `chmod 755`: Standard for executables
- `chmod 644`: Standard for regular files
- `chmod 600`: Private files
- `chmod 440`: Read-only for user and group

Tips

1. Use symbolic mode for specific changes
2. Use numeric mode for full permission sets
3. Be careful with recursive changes
4. Consider security implications

cp

Overview

The `cp` command copies files and directories. It can preserve attributes and handle recursive operations.

Syntax

```
cp [options] source destination
```

Common Options

- `-r`, `-R`: Recursive copy
- `-i`: Interactive (prompt before overwrite)
- `-p`: Preserve attributes
- `-v`: Verbose output
- `-u`: Update (copy only newer files)
- `-l`: Create hard links
- `-s`: Create symbolic links

Examples

```
# Basic file copy
cp file1.txt file2.txt

# Copy directory recursively
cp -r dir1/ dir2/

# Preserve attributes
cp -p source.txt dest.txt

# Interactive copy
cp -i *.txt /backup/
```

```
# Update only newer files
cp -u * /backup/

# Copy with progress
cp -v largefile.iso /media/
```

Tips

1. Use `-i` to prevent accidental overwrites
2. `-a` preserves all attributes
3. `-u` is useful for backups
4. Use wildcards carefully

ls

Overview

The `ls` command lists directory contents. It's one of the most frequently used commands in Linux.

Syntax

```
ls [options] [file/directory]
```

Common Options

- `-l`: Long listing format
- `-a`: Show all files (including hidden)
- `-h`: Human-readable sizes
- `-R`: Recursive listing
- `-t`: Sort by modification time
- `-S`: Sort by file size

Examples

```
# Basic listing
ls

# Detailed listing with human-readable sizes
ls -lh

# Show hidden files
ls -la

# Sort by size
ls -lS

# Recent files first
ls -lt
```

```
# Recursive listing
ls -R
```

Tips

1. Use `ls -lah` for complete detailed view
2. Combine with `grep` for filtering
3. Use color coding for better visibility
4. Sort options help find specific files quickly

mv

Overview

The `mv` command moves or renames files and directories. It's essential for file management.

Syntax

```
mv [options] source destination
```

Common Options

- `-i`: Interactive mode
- `-f`: Force move
- `-n`: No overwrite
- `-v`: Verbose
- `-u`: Update only
- `-b`: Create backup

Examples

```
# Rename a file
mv oldname.txt newname.txt

# Move file to directory
mv file.txt /path/to/dir/

# Move multiple files
mv file1.txt file2.txt dir/

# Interactive move
mv -i important.txt /backup/

# Move with backup
mv -b file.txt /path/
```

```
# Move newer files only
mv -u *.txt /path/
```

Tips

1. Use `-i` for important files
2. Create backups with `-b`
3. Test with `-v` first
4. Check destination space

rm

Overview

The `rm` command removes files and directories. Use with caution as recovery is often impossible.

Syntax

```
rm [options] file(s)
```

Common Options

- `-r`, `-R`: Recursive removal
- `-f`: Force removal
- `-i`: Interactive mode
- `-v`: Verbose mode
- `-d`: Remove empty directories
- `--preserve-root`: Prevent root directory deletion

Examples

```
# Remove single file
rm file.txt

# Remove interactively
rm -i important.txt

# Remove directory and contents
rm -r directory/

# Force remove without prompts
rm -rf old_directory/

# Remove empty directory
rm -d empty_dir/
```

```
# Verbose removal
rm -v *.tmp
```

Safety Tips

1. ALWAYS double-check wildcards
2. Use `-i` for important operations
3. Never use `rm -rf /`
4. Consider using trash instead
5. Make backups before bulk deletions

Alternative Methods

1. Using trash-cli:

```
trash file.txt
trash-list
trash-restore
```

2. Creating aliases:

```
alias rm='rm -i'
alias del='mv -t ~/.trash'
```

02_Info

apropos

Overview

The `apropos` command searches the manual page names and descriptions for a specific keyword. It's useful for finding commands when you don't remember their exact names.

Syntax

```
apropos [options] keyword
```

Common Options

- `-a`: Display only matches that satisfy all keywords
- `-r`: Use regular expressions for searching
- `-s sections`: Look only in given manual sections
- `-l`: List only page names

Examples

```
# Find commands related to passwords
apropos password
# Shows all commands with "password" in their description

# Search with multiple keywords
apropos -a user password
# Shows commands containing both "user" and "password"

# Use regex pattern
apropos -r '^find.*'
# Lists all commands starting with "find"
```

Tips

1. Use when you can't remember the exact command name
2. Combine with `man` to read full documentation
3. More detailed than `whatis`
4. Great for discovering new commands

info

Overview

The `info` command provides comprehensive documentation with a more structured format than man pages. It supports hyperlinks and a menu-based navigation system.

Syntax

```
info [options] [command_name]
```

Common Options

- `--help`: Display help information
- `--version`: Show version information
- `-f FILE`: Specify the Info file to read
- `-n NODE`: Specify the node to view

Navigation

- `n`: Next node
- `p`: Previous node
- `u`: Up one level
- `l`: Last node viewed
- `d`: Return to directory level
- `h`: Show help
- `q`: Quit

Examples

```
# View info documentation
info ls

# Go to specific node
info -n 'Copy/Paste' emacs

# View info directory
info dir
```

Tips

1. Use `h` to learn navigation commands
2. The menu structure makes finding information easier
3. Info pages often contain more detailed information than man pages
4. Use TAB to move between links

man

Overview

The `man` command is used to display the system's manual pages. It provides detailed documentation for commands, system calls, libraries, and more.

Syntax

```
man [section] command_name
```

Manual Sections

1. User Commands
2. System Calls
3. Library Functions
4. Special Files
5. File Formats
6. Games
7. Miscellaneous
8. System Administration

Common Options

- `-f`: Same as whatis
- `-k`: Same as apropos
- `-w`: Print manual page locations
- `-a`: Display all matching pages

Examples

```
# View manual for ls command
man ls

# View specific section
man 2 write

# Find all related pages
man -k password

# Show manual page location
man -w bash
```

Navigation

- `Space`: Next page
- `b`: Previous page
- `/pattern`: Search forward
- `n`: Next match
- `q`: Quit

Tips

1. Use `man man` to learn more about the man command
2. Section numbers help find specific documentation
3. The `-k` option helps find related commands

whatis

Overview

The `whatis` command displays one-line manual page descriptions. It's useful for quickly finding out what a command does without reading the full manual page.

Syntax

```
whatis [options] command_name
```

Common Options

- `-w`: Wildcard search
- `-r`: Regex search
- `-l`: List all matches

Examples

```
# Basic usage
whatis ls
# Output: ls (1) - list directory contents

# Multiple commands
whatis cp mv rm
# Output shows description for each command

# Wildcard search
whatis -w "ls*"
# Shows all commands starting with 'ls'
```

Tips

1. Use `whatis` for quick command reference
2. Combine with `apropos` for more detailed searches
3. Great for learning new commands

03_ProcessMgmt

Using the ls command

The `ls` command is one of the most commonly used commands in Linux and Unix systems. It is used to list the contents of a directory.

Basic Usage

The basic syntax is:

```
ls [options] [directory]
```

If no directory is specified, ls will show contents of the current directory.

Common Options

Some frequently used options include:

- `-l`: Long listing format showing detailed file information
- `-a`: Show all files including hidden ones (starting with .)

- `-h`: Human readable file sizes
- `-R`: Recursive listing of subdirectories
- `-t`: Sort by modification time
- `-S`: Sort by file size

Examples

List current directory contents:

```
ls
```

Show detailed file information:

```
ls -l
```

Show hidden files:

```
ls -la
```

List files recursively in subdirectories:

```
ls -R
```

Show files sorted by size in human readable format:

```
ls -lhS
```

Output Explanation

When using `ls -l`, the output shows:

- File permissions
- Number of hard links
- Owner name
- Group name
- File size
- Last modified date/time
- Filename

For example:

```
drwxr-xr-x  2 user group 4096 Jan 1 10:00 Documents
-rw-r--r--  1 user group  123 Jan 1 09:00 file.txt
```

The `ls` command is essential for navigating and managing files in Unix-like systems.

ps

Overview

The `ps` command provides information about active processes. Essential for system monitoring and troubleshooting.

Syntax

```
ps [options]
```

Common Options

- `-e`: Show all processes
- `-f`: Full format listing
- `-u username`: Show user's processes
- `-aux`: BSD style listing
- `--sort`: Sort by criteria

Examples

```
# Show all processes (BSD style)
ps aux

# Show process tree
ps -ejH

# Show specific user's processes
ps -u username

# Sort by memory usage
ps aux --sort=-%mem

# Sort by CPU usage
ps aux --sort=-%cpu
```

```
# Show process threads
ps -eLf
```

Output Fields

- `PID`: Process ID
- `TTY`: Terminal type
- `TIME`: CPU time
- `CMD`: Command name
- `%CPU`: CPU usage
- `%MEM`: Memory usage
- `VSZ`: Virtual memory size
- `RSS`: Resident set size

Tips

1. Combine with `grep` to find specific processes
2. Use `top` for real-time updates
3. Check both CPU and memory usage
4. Look for zombie processes

Monitoring Commands

Overview

Linux monitoring commands help track system resources, processes, and performance metrics in real-time.

Common Commands

top/htop

- Real-time system monitoring
- Shows CPU, memory, processes
- Interactive process viewer

vmstat

- Virtual memory statistics
- System performance data
- CPU/memory/IO metrics

iostat

- CPU and I/O statistics
- Disk activity monitoring
- System input/output stats

Resource Monitoring

Memory Commands

- `free`: Display memory usage
- `pmap`: Process memory map
- `smem`: Memory reporting tool

CPU Commands

- `mpstat`: CPU statistics
- `sar`: System activity reporter
- `uptime`: Load average info

Disk Commands

- `df`: Disk space usage
- `du`: Directory space usage
- `iotop`: I/O monitoring

Common Options

- `-h`: Human readable
- `-c`: Continuous monitoring
- `-d`: Delay between updates
- `-p`: Process specific monitoring

Examples

```
# Monitor memory every 2 seconds
free -h -s 2

# Check disk space in human readable format
df -h

# Monitor CPU usage
mpstat 1

# Watch disk I/O
iostat -xz 1

# Monitor specific process
top -p PID

# Check system load
uptime
```

Common Monitoring Patterns

- `top -u username`: Monitor user processes
- `free -m`: Memory in megabytes
- `df -i`: Check inode usage
- `du -sh */`: Directory sizes
- `sar -u 1 5`: CPU usage for 5 seconds

Tips

1. Use appropriate update intervals
2. Consider system impact
3. Filter output for relevant data
4. Save output for analysis
5. Use tools in combination

Pro Tips

1. Create aliases for common monitoring
2. Schedule regular checks
3. Set up monitoring thresholds
4. Keep historical data
5. Use graphical tools when needed


```
`<!-- quarto-file-metadata: eyJyZXNvdXJjZURpciI6ImNvbnRlbnQvMDEtQ29tbWFuZHMvMDNfUHJvY2Vzc01nbXQ

```{=html}
<!-- quarto-file-metadata: eyJyZXNvdXJjZURpciI6ImNvbnRlbnQvMDEtQ29tbWFuZHMvMDNfUHJvY2Vzc01nbXQi
```
```

top

Overview

The `top` command provides a dynamic real-time view of running processes. It's interactive and updates periodically.

Syntax

```
top [options]
```

Interactive Commands

- `q`: Quit
- `h`: Help
- `k`: Kill process
- `r`: Renice process
- `f`: Select fields
- `o`: Change sort field
- `W`: Save settings

Display Fields

1. System Summary
    - Uptime and load averages
    - Tasks and CPU states
    - Memory usage (RAM/Swap)
2. Process List
    - PID: Process ID
    - USER: Process owner
    - PR: Priority
    - NI: Nice value
    - VIRT: Virtual memory
    - RES: Physical memory

- SHR: Shared memory
- S: Process status
- %CPU: CPU usage
- %MEM: Memory usage
- TIME+: CPU time
- COMMAND: Command name

Common Options

```
# Update every 2 seconds
top -d 2

# Show specific user's processes
top -u username

# Batch mode (non-interactive)
top -b -n 1

# Sort by memory usage
top -o %MEM
```

Tips

1. Use `1` to show individual CPU cores
2. `M` sorts by memory usage
3. `P` sorts by CPU usage
4. `c` shows full command path

04_Monitoring

iostat

`iostat` (Input/Output Statistics) is a powerful Linux command that provides real-time statistics about disk I/O activity.

Basic Syntax

```
iostat [options] [delay [count]]
```

Common Options

- `-d` : Display disk statistics
- `-k` : Display statistics in kilobytes
- `-m` : Display statistics in megabytes
- `-n` : Display header only once
- `-x` : Display extended statistics

Example Output

```
$ iostat
Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_a
sda               0.00     0.00    0.00    0.00     0.00     0.00    0.00     0.00    0.00    0
sdb               0.00     0.00    0.00    0.00     0.00     0.00    0.00     0.00    0.00    0
```

Field Descriptions

- `rrqm/s`: Number of read requests per second
- `wrqm/s`: Number of write requests per second
- `r/s`: Number of reads per second
- `w/s`: Number of writes per second
- `rMB/s`: Number of read bytes per second in megabytes
- `wMB/s`: Number of write bytes per second in megabytes
- `avgrq-sz`: Average request size in bytes
- `avgqu-sz`: Average queue length

- `await`: Average wait time in milliseconds
- `r_await`: Average read wait time in milliseconds
- `w_await`: Average write wait time in milliseconds
- `svctm%`: Service time percentage (utilization)
- `util%`: Utilization percentage

Additional Examples

Display Disk Statistics

```
$ iostat -d
Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_a
sda               0.00     0.00    0.00    0.00     0.00     0.00    0.00     0.00    0.00    0
sdb               0.00     0.00    0.00    0.00     0.00     0.00    0.00     0.00    0.00    0
```

Display Extended Statistics

```
$ iostat -x
Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_a
sda               0.00     0.00    0.00    0.00     0.00     0.00    0.00     0.00    0.00    0


`<!-- quarto-file-metadata: eyJyZXNvdXJjZURpciI6ImNvbnRlbnQvMDEtQ29tbWFuZHMvMDRfTW9uaXRvcmluZyJ

```{=html}
<!-- quarto-file-metadata: eyJyZXNvdXJjZURpciI6ImNvbnRlbnQvMDEtQ29tbWFuZHMvMDRfTW9uaXRvcmluZyIs
```
```

iostat zed

The `iostat` command is used to monitor system input/output device loading by observing the time the devices are active in relation to their average transfer rates.

Syntax

```
iostat [options] [interval] [count]
```

Common Options

- `-c` : Display CPU utilization report
- `-d` : Display device utilization report
- `-k` : Display statistics in kilobytes per second
- `-m` : Display statistics in megabytes per second
- `-x` : Display extended statistics
- `-p` : Display statistics for block devices and partitions

Examples

1. Basic iostat output:

```
$ iostat
Linux 5.4.0-42-generic (hostname)     07/15/2023     _x86_64_    (4 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           2.50    0.00    1.50    0.50    0.00   95.50

Device             tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda               2.50        30.50        20.30     450568     300124
```

2. Display extended disk statistics:

```
$ iostat -x
Device   rrqm/s  wrqm/s  r/s   w/s   rsec/s  wsec/s  avgrq-sz avgqu-sz await svctm %util
sda        0.00    0.50   2.00  1.50   30.50   20.30    20.32     0.01  2.50  1.20  0.42
```

3. Display CPU statistics only:

```
$ iostat -c
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           2.50    0.00    1.50    0.50    0.00   95.50
```

4. Display disk statistics every 2 seconds for 5 times:

```
$ iostat -d 2 5
```

5. Display statistics in megabytes:

```
$ iostat -m
```

Output Fields Explained

- `tps`: Transfers per second (I/O requests)
- `kB_read/s`: Amount of data read per second
- `kB_wrtn/s`: Amount of data written per second
- `kB_read`: Total amount of data read
- `kB_wrtn`: Total amount of data written
- `%util`: Percentage of CPU time during which I/O requests were issued

vmstat

`vmstat` (Virtual Memory Statistics) is a powerful Linux command that provides information about system processes, memory, paging, block IO, traps, and CPU activity.

Basic Syntax

```
vmstat [options] [delay [count]]
```

Common Options

- `-a` : Display active and inactive memory
- `-f` : Display the number of forks since boot
- `-m` : Display memory information in MB rather than KB
- `-n` : Display header only once
- `-s` : Display memory statistics
- `-d` : Display disk statistics

Example Output

```
$ vmstat
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0      0 823012  66268 460644    0    0     1     1    1    2  1  0 98  0  0
```

Field Descriptions

Procs

- `r`: Number of processes waiting for runtime
- `b`: Number of processes in uninterruptible sleep

Memory

- `swpd`: Used virtual memory
- `free`: Idle memory
- `buff`: Memory used as buffers
- `cache`: Memory used as cache

Swap

- `si`: Memory swapped from disk
- `so`: Memory swapped to disk

IO

- `bi`: Blocks received from block device
- `bo`: Blocks sent to block device

System

- `in`: Number of interrupts per second
- `cs`: Number of context switches per second

CPU

- `us`: Time spent in user code
- `sy`: Time spent in system code
- `id`: Time spent idle
- `wa`: Time spent waiting for IO
- `st`: Time stolen from a virtual machine

Example with Delay

```
$ vmstat 2 5  # Display stats every 2 seconds, 5 times
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0      0 823012  66268 460644    0    0     1     1    1    2  1  0 98  0  0
 0  0      0 823012  66268 460644    0    0     0     0   88  156  1  0 99  0  0
 0  0      0 823012  66268 460644    0    0     0     0   85  152  0  0 100 0  0
 0  0      0 823012  66268 460644    0    0     0     0   86  154  0  0 100 0  0
 0  0      0 823012  66268 460644    0    0     0     0   84  150  0  0 100 0  0
```

Additional Examples

Display Active/Inactive Memory

```
$ vmstat -a
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free  inact active   si   so    bi    bo   in   cs us sy id wa st
 1  0      0 823012 264888 460644    0    0     1     1    1    2  1  0 98  0  0
```

Show Memory Statistics

```
$ vmstat -s
      2048000 K total memory
       823012 K used memory
       660644 K active memory
       264888 K inactive memory
        66268 K free memory
            0 K buffer memory
       458756 K swap cache
```

Display Disk Statistics

```
$ vmstat -d
disk- ------------reads------------ ------------writes----------- -----IO------
       total merged sectors      ms  total merged sectors      ms    cur    sec
sda    40307   2109  714418    8564   23012   14028  298010   22534   0      42
sdb    35292   1819  684290    7845   21320   13822  287999   20145   0      38
```

Show Fork Statistics

```
$ vmstat -f
     386281 forks
```

This command is particularly useful for: - System performance monitoring - Troubleshooting memory issues - Identifying system bottlenecks - Real-time system statistics

Networking

01_TCPIP

Intro

TCP IP is a network protocol that enables communication between devices on a network. It is a set of protocols that define how data is transferred over the network.

TCP

TCP is a reliable, connection-based protocol that provides a way for two devices to exchange data over a network. It ensures that data is delivered in order and without errors.

IP

IP is a network protocol that assigns unique addresses to devices on a network. It is used to identify and communicate with devices on a network.

OSI Model

The OSI model is a framework for understanding how data is transferred over a network. It divides the network into layers, each with a specific purpose and responsibilities.

OSI Layers

The OSI model has 7 layers, each with a specific purpose and responsibilities: - Physical Layer: Handles the physical connection between devices - Data Link Layer: Handles the transmission of data between devices - Network Layer: Handles the routing and delivery of data between devices - Transport Layer: Handles the delivery of data between devices - Session Layer: Handles the creation and management of sessions between devices - Presentation Layer: Handles the presentation of data to the user - Application Layer: Handles the application-specific logic of the data

TCP/IP Stack

The TCP/IP stack is a set of protocols that provide the underlying network infrastructure for TCP/IP. It includes protocols such as IP, TCP, and UDP.

Summary

02_CCNA

Intro

CCNA is a certification exam for network professionals. It is a series of exams that test the knowledge and skills of network administrators and network engineers.

CCNA 200-301

CCNA 200-301 is a certification exam for network professionals. It is a series of exams that test the knowledge and skills of network administrators and network engineers.

CCNA 300-301

CCNA 300-301 is a certification exam for network professionals. It is a series of exams that test the knowledge and skills of network administrators and network engineers.

CCNA 400-501

CCNA 400-501 is a certification exam for network professionals. It is a series of exams that test the knowledge and skills of network administrators and network engineers.

CCNA 500-501

CCNA 500-501 is a certification exam for network professionals. It is a series of exams that test the knowledge and skills of network administrators and network engineers.

Summary

CCNA is a certification exam for network professionals. It is a series of exams that test the knowledge and skills of network administrators and network engineers.

Linux

01-Commands

GitOps

Git Commands

1. Stashing Changes

Stashing allows you to save your current changes temporarily so you can work on something else and come back to them later.

- `git stash` -> Temporarily saves your uncommitted changes.
- `git stash list` -> Lists all stashed changes.
- `git stash apply stash@{0}` -> Applies the specified stash without removing it.
- `git stash pop` -> Applies the most recent stash and removes it from the stash list.
- `git stash drop stash@{0}` -> Deletes a specific stash.

2. Working with Tags

Tags are used to mark specific points in the repository's history as being important, typically for releases.

- `git tag` -> Lists all tags in the repository.
- `git tag -a v1.0 -m "Version 1.0"` -> Creates an annotated tag with a message.
- `git push origin v1.0` -> Pushes a specific tag to the remote repository.
- `git push --tags` -> Pushes all tags to the remote repository.
- `git tag -d v1.0` -> Deletes a tag locally.
- `git push origin --delete v1.0` -> Deletes a tag from the remote repository.

3. Interactive Rebase

Interactive rebase allows you to modify the commit history by reordering, editing, or squashing commits.

- `git rebase -i HEAD~3` -> Opens an interactive rebase for the last 3 commits.

Actions during interactive rebase:

- `pick`: Keep the commit as is.
- `squash`: Combine this commit with the previous one.
- `reword`: Modify the commit message.

- `edit`: Modify the commit.
- `git rebase --abort` -> Aborts the rebase process and restores the original branch.
- `git rebase --continue` -> Continues the rebase process after resolving conflicts.

4. Cherry-Picking

Cherry-picking allows you to apply specific commits from one branch to another.

- `git cherry-pick commit-hash` -> Applies a specific commit from another branch.
- `git cherry-pick --continue` -> Continues the cherry-pick process after resolving conflicts.
- `git cherry-pick --abort` -> Aborts the cherry-pick process.

5. Working with Remotes

Managing remote repositories involves adding, removing, fetching from, and pushing to remote repositories.

- `git remote -v` -> Lists all remote repositories and their URLs.
- `git remote add origin https://github.com/user/repo.git` -> Adds a new remote repository.
- `git remote remove origin` -> Removes a remote repository.
- `git fetch origin` -> Fetches all branches and updates from the remote.
- `git pull origin branch-name --rebase` -> Rebases the local branch with the remote branch.

6. Git Aliases

Aliases are shortcuts for longer Git commands, making it easier to use frequently used commands.

- `git config --global alias.st status` -> Creates an alias for 'git status'.
- `git config --global alias.co checkout` -> Creates an alias for 'git checkout'.
- `git config --global alias.br branch` -> Creates an alias for 'git branch'.

7. Bisecting to Find Bugs

Bisecting is a process to find the commit that introduced a bug by performing a binary search through the commit history.

- `git bisect start` -> Starts the bisect process.
- `git bisect bad` -> Marks the current commit as bad.
- `git bisect good commit-hash` -> Marks a specific commit as good.
- `git bisect reset` -> Ends the bisect session and returns to the original branch.

8. Cleaning Untracked Files

Cleaning removes untracked files and directories from your working directory, which can be useful for removing build artifacts or temporary files.

- `git clean -n` -> Displays the untracked files and directories that would be removed.
- `git clean -f` -> Removes untracked files.
- `git clean -fd` -> Removes untracked files and directories.

Git

Git and GitHub: A Comprehensive Learning Path

1. Git Fundamentals

Understanding Version Control

- ☐ Learn what version control is and its importance
- ☐ Compare Git with other version control systems
- ☐ Install Git on your local machine
- ☐ Configure Git (local and global settings)

Basic Git Operations

- ☐ Initialize a repository (`git init`)
- ☐ Understand the three states:
  - Working Directory
  - Staging Area
  - Repository
- ☐ Create and manage `.gitignore` files
- ☐ Make your first commit
- ☐ View commit history (`git log`)

Branching and Merging

- ☐ Create new branches (`git branch`)
- ☐ Switch between branches (`git checkout` or `git switch`)
- ☐ Merge branches
- ☐ Handle merge conflicts
- ☐ Learn branch naming conventions

2. GitHub Essentials

Getting Started

- ☐ Create a GitHub account
- ☐ Set up your profile
- ☐ Create your profile README
- ☐ Understand public vs private repositories

Remote Repository Management

- ☐ Add and manage remotes
- ☐ Push and pull changes
- ☐ Fetch changes without merging
- ☐ Clone repositories

Collaboration Basics

- ☐ Fork repositories
- ☐ Create pull requests
- ☐ Review pull requests
- ☐ Manage issues
- ☐ Use mentions and reactions
- ☐ Write effective commit messages

3. Advanced Git Operations

History Management

- ☐ Use `git stash`
- ☐ Understand HEAD and detached HEAD
- ☐ View diffs between:
  - Commits
  - Branches
  - Staged/unstaged changes

Undoing Changes

- ☐ Use `git revert`
- ☐ Reset changes (`--soft`, `--hard`, `--mixed`)
- ☐ Amend commits
- ☐ Rebase branches
- ☐ Force push safely

Tags and Releases

- ☐ Create and manage tags
- ☐ Push tags to remote
- ☐ Create GitHub releases
- ☐ Use semantic versioning

4. Git Hooks and Automation

Git Hooks

- ☐ Set up client-side hooks
- ☐ Configure server-side hooks
- ☐ Implement common hooks:
  - pre-commit
  - post-checkout
  - commit-msg

GitHub Actions

- ☐ Write YAML workflows
- ☐ Set up triggers
- ☐ Use workflow runners
- ☐ Manage secrets
- ☐ Cache dependencies
- ☐ Store artifacts

5. Team Collaboration

Organization Management

- ☐ Set up GitHub organizations
- ☐ Manage teams and members
- ☐ Set up permissions

Project Management

- ☐ Use GitHub Projects
- ☐ Create Kanban boards
- ☐ Plan roadmaps
- ☐ Set up automations
- ☐ Use GitHub Discussions

6. Advanced Features

GitHub Developer Tools

- ☐ Use GitHub CLI
- ☐ Work with GitHub API
  - REST API
  - GraphQL API
- ☐ Create GitHub Apps
- ☐ Set up webhooks

Additional GitHub Features

- ☐ Deploy with GitHub Pages
- ☐ Use GitHub Codespaces
- ☐ Work with GitHub Packages
- ☐ Explore GitHub Marketplace
- ☐ Implement GitHub Security features

Best Practices

- ☐ Write clear commit messages
- ☐ Follow branch naming conventions
- ☐ Create comprehensive documentation
- ☐ Maintain clean Git history
- ☐ Review code effectively
- ☐ Write contribution guidelines

Resources

- Official Git documentation: https://git-scm.com/doc
- GitHub documentation: https://docs.github.com
- GitHub Skills: https://skills.github.com
- Interactive Git learning: https://learngitbranching.js.org

Note: Check off items as you complete them to track your progress.

Category1

subcategory1

Introduction to Category 1

This is the main page for Category 1's first subcategory.

Section 1.1

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Section 1.2

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Subsection 1.2.1

Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Tables

Column 1	Column 2	Column 3
Value 1	Value 2	Value 3
Item A	Item B	Item C

Cross-references

You can reference other sections using Chapter .

subcategory2

Advanced Topics

Document Organization

Good document organization is crucial for readability and maintainability.

Sections and Subsections

Breaking down content into logical sections helps readers navigate the material effectively.

Callouts

> **Note**
>
> This is a note callout block for important information.

> **Warning**
>
> This is a warning callout block for critical warnings.

> **Pro Tip**
>
> You can add titles to callouts to make them more descriptive!

Lists and Formatting

1. **Ordered Lists**: Like this one
2. With multiple items
    - And sub-items
    - That can be nested
3. For structured content

Emphasis and Highlighting

You can use *italics* and **bold** text for emphasis. Also `inline code` for technical terms.

> Important quotes or excerpts can be blockquoted like this for emphasis and visual distinction.

subcategory3

Writing Best Practices

Style Guidelines

Consistency in Writing

Maintaining a consistent writing style throughout your document helps readers follow your content more easily.

- Use consistent terminology
- Maintain consistent formatting
- Keep a consistent tone

Voice and Tone

Choose an appropriate voice for your audience:

1. Technical audience
    - Be precise and direct
    - Use industry-standard terminology
    - Provide relevant examples
2. General audience
    - Use clear, simple language
    - Explain technical terms
    - Include more context

Document Structure

Headers and Sections

> Tip: Header Hierarchy
>
> Use headers to create a clear content hierarchy: - H1 for main titles - H2 for major sections - H3 for subsections

Lists and Bullets

Effective use of lists:

- Main points
  - Supporting details
  - Additional information
- Next main point
  - Related subtopics
  - More details

Review Process

1. Self-review
2. Peer review
3. Technical review
4. Final editorial review

subcategory4

Document Planning

Content Strategy

Audience Analysis

Before writing, consider your audience:

Aspect	Technical Audience	General Audience
Language	Technical terms	Simple explanations
Detail Level	In-depth	Overview
Examples	Code/Technical	Real-world
Assumptions	Domain knowledge	Basic understanding

Content Organization

> **Note**
>
> Good organization is key to effective documentation

1. Start with an outline
2. Group related topics
3. Create logical flow
4. Review and adjust

Research Methods

Primary Sources

- Direct observations
- Original research
- First-hand accounts

Secondary Sources

- Literature reviews
- Expert analyses
- Case studies

Timeline Planning

Phase 1

- Initial research
- Outline creation
- Stakeholder review

Phase 2

- Content development
- Technical review
- Revisions

Phase 3

- Final editing
- Publication
- Distribution

Quality Metrics

> "Measure twice, cut once" applies to documentation too.

Quality checkpoints: 1. Technical accuracy 2. Completeness 3. Clarity 4. Consistency 5. Accessibility

Category2

Category 2 Content

Document Features

This category demonstrates different document formatting options and features.

Images and Figures

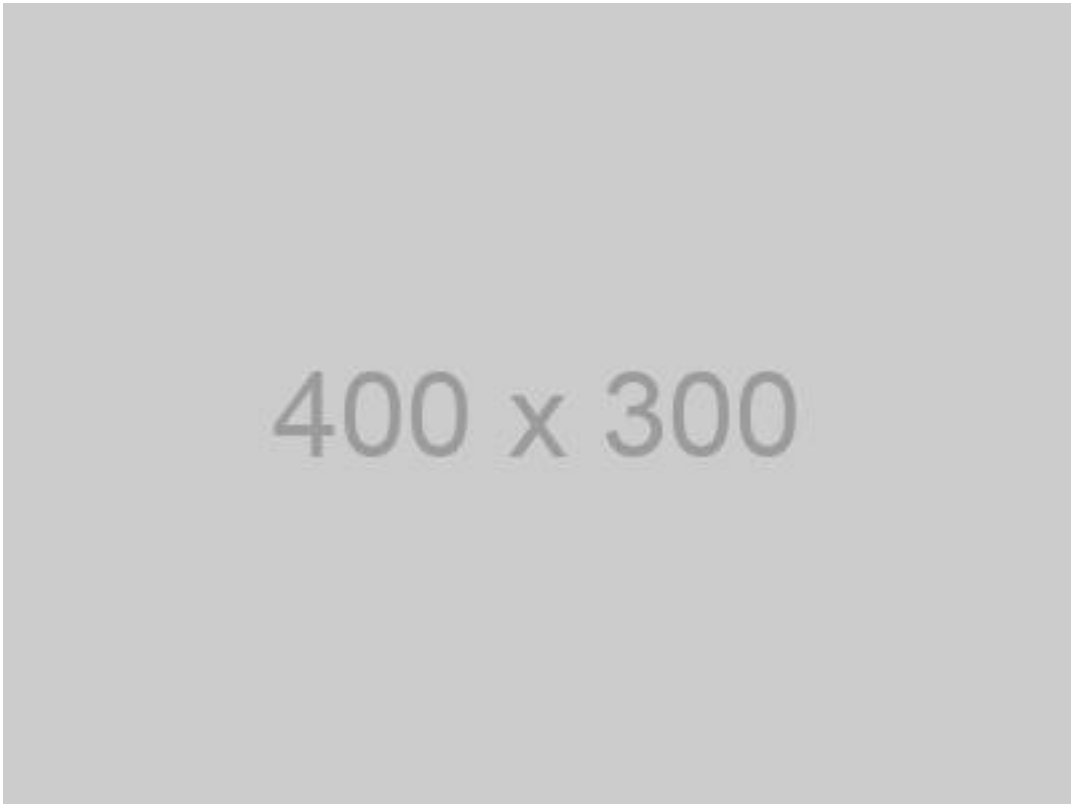

Figure 1: Sample Image

Tabbed Content

First Tab

Content for the first tab, discussing main points.

Second Tab

Additional information and details in the second tab.

Third Tab

Supplementary content and references.

Citations and References

Here's a sentence with a footnote[1].

Definition Lists

Term 1 Definition of the first term
Additional details about the first term
Term 2 Explanation of the second term

Margin Content

This is the main text of the document.

This is supplementary content that appears in the margin in HTML output.

Special Formatting

1. **Bold text** for emphasis
2. *Italic text* for subtle emphasis
3. `Monospace text` for technical terms
4. ~~Strikethrough~~ for removed content
5. Links for references

> This is a blockquote that can be used to highlight important quotes or excerpts from other sources.

[1] This is the footnote content explaining the reference.

subcategory1

Visual Elements

Effective Use of Images

Image Guidelines

1. Use relevant images
2. Maintain consistent style
3. Provide clear captions
4. Optimize for different formats

Example Figures

Figure 1: Placeholder for diagram

Tables and Charts

Data Presentation

Category	Print	Digital	Notes
Resolution	300dpi	72dpi	Adjust per medium
Color	CMYK	RGB	Consider output
Size	Fixed	Responsive	Plan accordingly

Layout Considerations

! Layout Tips

- Consider page breaks
- Mind the margins
- Balance text and visuals

Interactive Elements

Static Version

Content for print format

Interactive Version

Content for digital format

Hybrid Approach

Best of both worlds

Accessibility Guidelines

1. Alt text for images
2. Color contrast
3. Text alternatives
4. Screen reader support

Remember to test accessibility with different tools and users.

subcategory2

Publication Formats

Print Considerations

Paper Selection

Different paper types and their uses:

1. Bond paper
    - Standard documents
    - Internal drafts
2. Coated paper
    - High-quality images
    - Professional publications

Binding Options

> **Note**
>
> Choose binding based on: - Document length - Usage patterns - Budget constraints

Digital Formats

Format Comparison

Format	Advantages	Disadvantages
PDF	Universal support	Limited interactivity
HTML	Highly interactive	Requires internet
EPUB	Good for ebooks	Device dependent

Digital Enhancements

1. Hyperlinks
2. Bookmarks
3. Search functionality
4. Interactive elements

Cross-Format Compatibility

Layout

- Responsive design
- Fluid typography
- Flexible images

Content

- Format-neutral writing
- Alternative text
- Fallback options

Navigation

- Consistent structure
- Clear hierarchy
- Multiple access points

Best Practices

Focus on content first, then optimize for each format.

Key Considerations

1. Test on multiple devices
2. Verify all features work
3. Check accessibility
4. Validate links and references

Category3

subcategory1

Review and Feedback

Review Process

Types of Reviews

1. Content Review
    - Accuracy check
    - Completeness
    - Consistency
2. Technical Review
    - Factual accuracy
    - Technical correctness
    - Implementation feasibility
3. Editorial Review
    - Grammar and style
    - Flow and readability
    - Format consistency

Feedback Management

Collecting Feedback

> **!** Feedback Guidelines
>
> - Be specific
> - Provide examples
> - Suggest improvements
> - Stay constructive

Feedback Categories

Type	Focus	Reviewers
Technical	Accuracy	Subject experts
Editorial	Style	Editors

Type	Focus	Reviewers
User	Usability	Target audience

Implementation

Planning

- Review schedule
- Reviewer selection
- Tools and methods

Execution

- Feedback collection
- Analysis
- Prioritization

Follow-up

- Implementation
- Verification
- Documentation

Best Practices

1. Set clear expectations
2. Use structured forms
3. Track changes
4. Document decisions

Regular reviews improve quality and maintain consistency.

subcategory2

Distribution and Maintenance

Distribution Channels

Digital Distribution

1. Web Platforms
    - Documentation sites
    - Knowledge bases
    - Learning platforms
2. File Sharing
    - Cloud storage
    - Content management systems
    - Version control systems

Maintenance Strategy

Update Cycle

> **Note**
>
> Regular updates ensure content stays relevant and accurate

Update Type	Frequency	Scope
Minor	Monthly	Typos, small changes
Major	Quarterly	Content updates
Complete	Yearly	Full revision

Version Control

Tracking

- Version numbers
- Change logs
- Release notes

Management

- Branch strategy
- Review process
- Merge policies

Documentation

- Update history
- Migration guides
- Deprecation notices

Quality Assurance

Automated Checks

1. Link validation
2. Format verification
3. Style compliance
4. Accessibility tests

Manual Review

> Review Checklist
>
> - Content accuracy
> - Format consistency
> - Link functionality
> - Image quality

Archival

Proper archival ensures historical versions remain accessible when needed.

1. Archive strategy
2. Storage solutions
3. Retrieval process
4. Retention policy

Testing

Software Testing Overview

This chapter introduces fundamental concepts in software testing and quality assurance.

Why Testing Matters

Testing is a crucial part of software development that helps ensure:

1. Reliability
2. Performance
3. Security
4. User satisfaction

Types of Testing

We'll cover various testing approaches:

- Unit Testing
- Integration Testing
- System Testing
- Performance Testing

Choose the right testing strategy based on your project needs.

advanced

Testing Strategies

Advanced Testing Strategies

Learn about different testing strategies:

1. TDD (Test Driven Development)
2. BDD (Behavior Driven Development)
3. ATDD (Acceptance Test Driven Development)

Testing Patterns

Common testing patterns include:

1. Arrange-Act-Assert
2. Given-When-Then
3. Setup-Exercise-Verify-Teardown

basics

Introduction to Testing Basics

This is an introduction to testing basics.

What is Testing?

Testing is the process of evaluating a system or its components to find whether it satisfies the specified requirements.

Types of Testing

Types of Testing

There are several types of testing:

1. Unit Testing
2. Integration Testing
3. System Testing
4. Acceptance Testing

code-examples

Code Examples Across Languages

This chapter showcases code examples from various programming languages, demonstrating syntax highlighting and formatting.

Fibonacci Sequence

Here's how to implement the Fibonacci sequence in different languages:

Python

```python
def fibonacci(n: int) -> list[int]:
    if n <= 0:
        return []
    elif n == 1:
        return [0]

    fib = [0, 1]
    for i in range(2, n):
        fib.append(fib[i-1] + fib[i-2])
    return fib

# Example usage with type hints
result: list[int] = fibonacci(10)
print(f"First 10 Fibonacci numbers: {result}")
```

Go

```go
package main

import "fmt"

func fibonacci(n int) []int {
    if n <= 0 {
        return []int{}
    }
```

```
    fib := make([]int, n)
    if n > 0 {
        fib[0] = 0
    }
    if n > 1 {
        fib[1] = 1
    }

    for i := 2; i < n; i++ {
        fib[i] = fib[i-1] + fib[i-2]
    }

    return fib
}

func main() {
    result := fibonacci(10)
    fmt.Printf("First 10 Fibonacci numbers: %v\n", result)
}
```

Rust

```
fn fibonacci(n: usize) -> Vec<u64> {
    if n == 0 {
        return vec![];
    }

    let mut fib = Vec::with_capacity(n);
    fib.push(0);

    if n > 1 {
        fib.push(1);
        for i in 2..n {
            let next = fib[i-1] + fib[i-2];
            fib.push(next);
        }
    }

    fib
}

fn main() {
    let result = fibonacci(10);
    println!("First 10 Fibonacci numbers: {:?}", result);
}
```

Zig

```zig
const std = @import("std");

fn fibonacci(n: usize, allocator: std.mem.Allocator) ![]u64 {
    if (n == 0) return &[_]u64{};

    var fib = try allocator.alloc(u64, n);
    fib[0] = 0;

    if (n > 1) {
        fib[1] = 1;
        var i: usize = 2;
        while (i < n) : (i += 1) {
            fib[i] = fib[i-1] + fib[i-2];
        }
    }

    return fib;
}

pub fn main() !void {
    var gpa = std.heap.GeneralPurposeAllocator(.{}){};
    defer _ = gpa.deinit();
    const allocator = gpa.allocator();

    const n = 10;
    const result = try fibonacci(n, allocator);
    defer allocator.free(result);

    std.debug.print("First {d} Fibonacci numbers: {any}\n", .{ n, result });
}
```

Odin

```odin
package main

import "core:fmt"

fibonacci :: proc(n: int) -> []int {
    if n <= 0 {
        return []int{}
    }

    fib := make([]int, n)
```

```
    defer delete(fib)

    if n > 0 {
        fib[0] = 0
    }
    if n > 1 {
        fib[1] = 1
    }

    for i := 2; i < n; i += 1 {
        fib[i] = fib[i-1] + fib[i-2]
    }

    return fib
}

main :: proc() {
    result := fibonacci(10)
    fmt.printf("First 10 Fibonacci numbers: %v\n", result)
}
```

Error Handling Examples

Here's how different languages handle errors:

Rust Error Handling

```
use std::fs::File;
use std::io::{self, Read};
use std::path::Path;

fn read_file_contents(path: &Path) -> Result<String, io::Error> {
    let mut file = File::open(path)?;
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    Ok(contents)
}

fn process_file() -> Result<(), io::Error> {
    let path = Path::new("example.txt");
    let contents = read_file_contents(path)?;
    println!("File contents: {}", contents);
    Ok(())
}
```

Go Error Handling

```go
package main

import (
    "fmt"
    "io/ioutil"
    "os"
)

func readFileContents(path string) (string, error) {
    data, err := ioutil.ReadFile(path)
    if err != nil {
        return "", fmt.Errorf("reading file: %w", err)
    }
    return string(data), nil
}

func processFile() error {
    contents, err := readFileContents("example.txt")
    if err != nil {
        return fmt.Errorf("processing file: %w", err)
    }
    fmt.Printf("File contents: %s\n", contents)
    return nil
}
```

Zig Error Handling

```zig
const std = @import("std");
const fs = std.fs;

fn readFileContents(path: []const u8, allocator: std.mem.Allocator) ![]u8 {
    const file = try fs.cwd().openFile(path, .{});
    defer file.close();

    return try file.readToEndAlloc(allocator, std.math.maxInt(usize));
}

fn processFile() !void {
    var gpa = std.heap.GeneralPurposeAllocator(.{}){};
    defer _ = gpa.deinit();
    const allocator = gpa.allocator();

    const contents = try readFileContents("example.txt", allocator);
```

```
    defer allocator.free(contents);

    std.debug.print("File contents: {s}\n", .{contents});
}
```

Memory Management Examples

Rust Memory Management

```rust
struct Buffer {
    data: Vec<u8>,
}

impl Buffer {
    fn new(size: usize) -> Self {
        Buffer {
            data: vec![0; size],
        }
    }

    fn process(&mut self) {
        // Data is automatically cleaned up when Buffer is dropped
        for i in 0..self.data.len() {
            self.data[i] = (i % 256) as u8;
        }
    }
}

fn main() {
    let mut buf = Buffer::new(1024);
    buf.process();
    // Buffer is automatically freed here
}
```

Zig Memory Management

```zig
const std = @import("std");

const Buffer = struct {
    data: []u8,
    allocator: std.mem.Allocator,

    pub fn init(size: usize, allocator: std.mem.Allocator) !Buffer {
```

```
        const data = try allocator.alloc(u8, size);
        return Buffer{ .data = data, .allocator = allocator };
    }

    pub fn deinit(self: *Buffer) void {
        self.allocator.free(self.data);
    }

    pub fn process(self: *Buffer) void {
        for (self.data, 0..) |*byte, i| {
            byte.* = @intCast(u8, i % 256);
        }
    }
};

pub fn main() !void {
    var gpa = std.heap.GeneralPurposeAllocator(.{}){};
    defer _ = gpa.deinit();
    const allocator = gpa.allocator();

    var buf = try Buffer.init(1024, allocator);
    defer buf.deinit();

    buf.process();
}
```

Concurrency Examples

Go Concurrency

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func worker(id int, jobs <-chan int, results chan<- int, wg *sync.WaitGroup) {
    defer wg.Done()
    for j := range jobs {
        fmt.Printf("worker %d processing job %d\n", id, j)
        time.Sleep(time.Millisecond * 100)
        results <- j * 2
    }
```

```
}

func main() {
    jobs := make(chan int, 100)
    results := make(chan int, 100)
    var wg sync.WaitGroup

    // Start workers
    for w := 1; w <= 3; w++ {
        wg.Add(1)
        go worker(w, jobs, results, &wg)
    }

    // Send jobs
    for j := 1; j <= 9; j++ {
        jobs <- j
    }
    close(jobs)

    // Wait for workers
    wg.Wait()
    close(results)

    // Collect results
    for r := range results {
        fmt.Printf("Result: %d\n", r)
    }
}
```

Rust Concurrency

```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn worker(id: u32, receiver: mpsc::Receiver<u32>, sender: mpsc::Sender<u32>) {
    for job in receiver {
        println!("worker {} processing job {}", id, job);
        thread::sleep(Duration::from_millis(100));
        sender.send(job * 2).unwrap();
    }
}

fn main() {
    let (job_tx, job_rx) = mpsc::channel();
```

```
    let (result_tx, result_rx) = mpsc::channel();
    let job_rx = std::sync::Arc::new(std::sync::Mutex::new(job_rx));

    // Start workers
    let mut handles = vec![];
    for id in 1..=3 {
        let job_rx = job_rx.clone();
        let result_tx = result_tx.clone();
        handles.push(thread::spawn(move || {
            worker(id, job_rx.lock().unwrap(), result_tx);
        }));
    }

    // Send jobs
    for j in 1..=9 {
        job_tx.send(j).unwrap();
    }
    drop(job_tx);

    // Collect results
    for _ in 1..=9 {
        println!("Result: {}", result_rx.recv().unwrap());
    }

    // Wait for workers
    for handle in handles {
        handle.join().unwrap();
    }
}
```

This demonstrates various code styling features:

1. Syntax highlighting for multiple languages
2. Different programming paradigms
3. Complex code structures
4. Error handling patterns
5. Memory management
6. Concurrency patterns
7. Type systems
8. Modern language features

integration-testing

Integration Testing Strategies

Integration testing ensures that different components of your system work together correctly.

Understanding Integration Tests

Integration tests verify the interaction between:

- Multiple functions or classes
- Different modules or services
- External dependencies
- Database interactions

Testing Approaches

Top-Down Integration

Start with high-level components and gradually test lower-level modules:

```
graph TD
    A[UI Layer] --> B[Business Logic]
    B --> C[Data Access]
    C --> D[Database]
```

Bottom-Up Integration

Begin with low-level components and progressively test higher layers.

Best Practices

1. Use test doubles when needed:
    - Mocks
    - Stubs
    - Fakes
2. Focus on component interfaces
3. Test realistic scenarios
4. Consider error conditions

unit-testing

Unit Testing Best Practices

Unit testing is the foundation of a solid testing strategy. This chapter covers essential unit testing concepts and practices.

What is Unit Testing?

Unit testing involves testing individual components or functions in isolation. A unit test typically follows this pattern:

1. Arrange - Set up the test conditions
2. Act - Execute the code being tested
3. Assert - Verify the results

Writing Good Unit Tests

Here are some key principles:

- Test one thing at a time
- Use descriptive test names
- Follow the FIRST principles:
  - Fast
  - Independent
  - Repeatable
  - Self-validating
  - Timely

Example Test Case

```
def test_add_numbers():
    # Arrange
    a = 5
    b = 3
    expected = 8

    # Act
    result = add_numbers(a, b)
```

```
    # Assert
    assert result == expected
```

Common Pitfalls

- Testing implementation details
- Brittle tests
- Poor test isolation
- Missing edge cases