Git-Github

K19G

2025-08-07

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

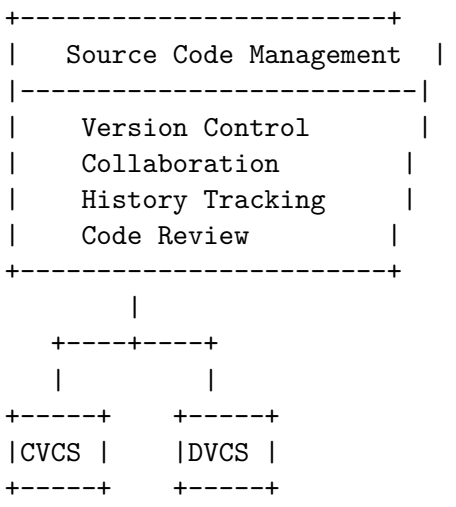
List any prerequisites or requirements here.

Acknowledgments

Add acknowledgments here.

Intro

Comprehensive Overview



Introduction to Source Code Management (SCM)

Source Code Management (SCM), also known as Version Control System (VCS), is a fundamental tool in modern software development. It helps developers track and manage changes to software code over time, enabling multiple developers to work together efficiently on complex projects.

Key Benefits of SCM

1. **Version Tracking**
    - Complete history of changes
    - Ability to revert to previous versions
    - Detailed audit trail of modifications
2. **Collaboration**
    - Multiple developers working simultaneously
    - Conflict resolution mechanisms
    - Code review capabilities
3. **Backup and Recovery**
    - Distributed copies of code
    - Disaster recovery
    - Branch-based development

Historical Evolution

Early Days (1970s-1980s)

1. **Source Code Control System (SCCS)**
    - Developed in 1972 at Bell Labs
    - Created by Marc Rochkind
    - Features:
        - Single-file version control
        - Delta compression
        - Access control lists
    - Limitations:
        - No networking capabilities
        - Single user at a time
        - Platform-dependent
2. **Revision Control System (RCS)**
    - Created in 1982 by Walter F. Tichy
    - Improvements over SCCS:
        - Better storage efficiency
        - Improved locking mechanisms
        - Enhanced merge capabilities
    - Still used in some Unix systems today

The Rise of Centralized VCS (1990s-2000s)

Concurrent Versions System (CVS)

- Released in 1990
- Key Features:
    - Client-server architecture
    - Multiple user support
    - Branch management
- Technical Details:
    - Written in C
    - Uses RCS file format
    - Network protocol: pserver

Apache Subversion (SVN)

- Released in 2000 by CollabNet
- Major Improvements:
  - Atomic commits
  - Directory versioning
  - Property support
- Architecture:
  - Repository storage options
    - FSFS (File System)
    - Berkeley DB
  - Network protocols
    - svn:// (custom protocol)
    - http:// (WebDAV)

Popular Version Control Tools

Centralized Version Control Systems

1. Apache Subversion (SVN)

- **Enterprise Features**
  - Path-based authorization
  - Integration with LDAP
  - Hook scripts for automation
- **Use Cases**
  - Document management
  - Configuration management
  - Digital asset management
- **Performance Characteristics**
  - Repository size: Unlimited
  - Number of users: Thousands
  - Network dependency: Required

2. Perforce

- **Enterprise Features**
  - Advanced access control
  - Built-in code review
  - Binary file management

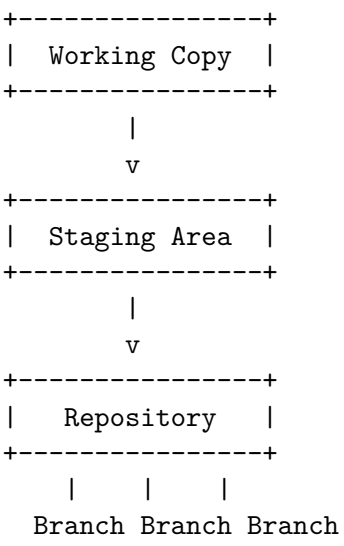
- **Specialized Features**
  - Graphic asset management
  - Large file handling
  - Stream-based development
- **Industry Usage**
  - Gaming companies
  - Hardware development
  - Film and animation

Distributed Version Control Systems

1. Git

- **Core Features**
  - Distributed architecture
  - SHA-1 hash integrity
  - Content-addressable storage
- **Performance**
  - Compression techniques
  - Delta storage
  - Local operations
- **Branching Model**
  - Lightweight branches
  - Multiple workflows support
  - Branch protection rules

Git's Architecture



2. Mercurial

- **Key Features**
  - Simple command set
  - Extension system
  - Built-in web interface
- **Technical Details**
  - Written in Python
  - Revlog storage format
  - Efficient delta compression

3. Fossil

- **Integrated Features**
  - Bug tracking
  - Wiki documentation
  - Built-in web interface
- **Unique Aspects**
  - Single-file repository
  - Built-in sync protocol
  - Automatic branching

The Git Story

Birth of Git

- **Historical Context**
  - Linux kernel development needs
  - BitKeeper controversy
  - Community requirements
- **Technical Design**
  - Content-addressable filesystem
  - Directed acyclic graph
  - Distributed architecture
- **Initial Development**
  - First commit: April 7, 2005
  - Written in C and Shell
  - Self-hosted within days

Git's Rise to Dominance

GitHub's Impact (2008)

- **Social Features**
  - Pull requests
  - Issues and wikis
  - Social coding
- **Platform Statistics**
  - 100+ million users
  - 330+ million repositories
  - 3+ billion contributions

Key Milestones

1. **2011**: GitHub surpasses Sourceforge
   - 2 million users
   - 3 million repositories
2. **2018**: Microsoft Acquisition
   - $7.5 billion deal
   - Enterprise focus
3. **2023**: Platform Growth
   - AI-powered features
   - Advanced security
   - Improved collaboration tools

Current Status

Industry Standard

- **Market Share**
  - 90%+ developer adoption
  - Used by 90 of Fortune 100
  - Powers most open source

Modern Features

1. **Development Tools**
   - Advanced IDE integration
   - GitHub Copilot
   - Actions for CI/CD
2. **Security Features**
   - Dependabot
   - Code scanning
   - Secret scanning
3. **Collaboration Tools**
   - Codespaces
   - Projects
   - Discussions

Conclusion

Source code management has evolved from simple file versioning to sophisticated distributed systems. Git has emerged as the de facto standard, transforming how software is developed and maintained. Its impact continues to grow with new tools and practices in modern software development.

Additional Resources

1. Official Git Documentation
2. Pro Git Book
3. GitHub Guides
4. Git Cheat Sheet

Git Cheatsheet

Basic Commands

- `git init`: Initialize a new Git repository.
- `git clone <repo>`: Clone a repository into a new directory.
- `git add <file>`: Stage changes for commit.
- `git commit -m "<message>"`: Commit changes with a message.
- `git status`: Check the status of the working directory.
- `git push`: Push changes to a remote repository.
- `git pull`: Fetch and integrate changes from a remote repository.

Branching

- `git branch`: List all branches.
- `git checkout <branch>`: Switch to a specified branch.
- `git merge <branch>`: Merge a specified branch into the current branch.

Viewing History

- `git log`: Show the commit history.
- `git diff`: Show changes between commits.

Undoing Changes

- `git reset <file>`: Unstage a file.
- `git revert <commit>`: Create a new commit that undoes changes from a specified commit.

The Evolution of Source Code Control Systems

Introduction

Source code control systems (SCCS), also known as version control systems (VCS), have evolved dramatically over the past five decades. From simple file locking mechanisms to sophisticated distributed platforms, these tools have fundamentally transformed how software is developed, shared, and maintained. This chapter explores the historical progression of source code control, highlighting key innovations, paradigm shifts, and the impact on modern software development practices.

The Need for Source Code Control

Before diving into the history, it's important to understand why source code control became necessary:

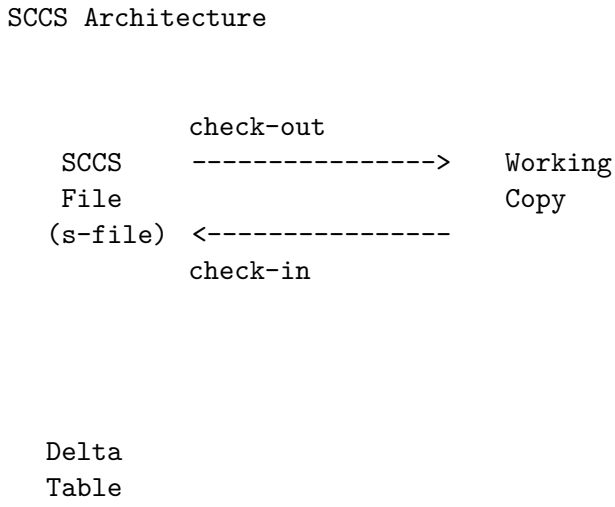
```
Problems Without Version Control

  File overwrites
  No change history
  Difficult collaboration
  No rollback capability
  Manual backup processes
  No audit trail
```

Early programmers relied on manual processes like dated backups, commented code, and strict access controls to manage source code. As software projects grew in complexity and team sizes increased, these approaches became unsustainable.

First Generation: Local VCS (1970s-1980s)

Source Code Control System (SCCS) - 1972

Developed by Marc Rochkind at Bell Labs, SCCS was the first formal version control system. It introduced the concept of storing changes as "deltas" rather than complete file copies.

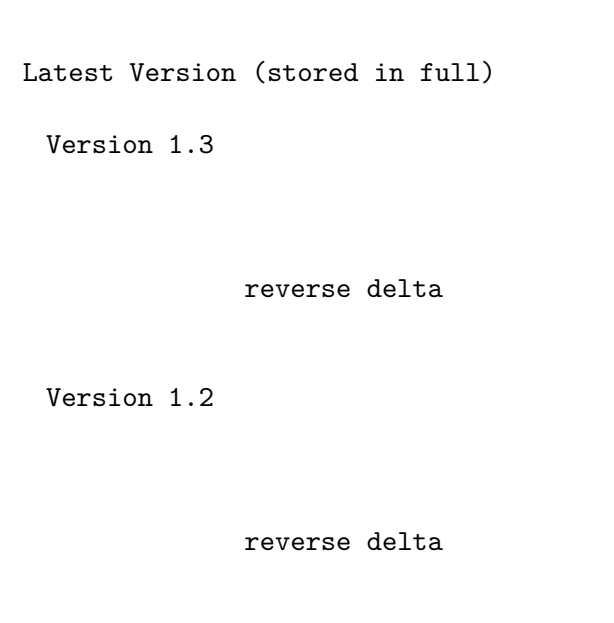


Key Features: - Single-file version control - Delta compression (storing only changes) - File locking mechanism (check-out/check-in) - Access control lists - Revision numbering

Limitations: - Single-user access at a time - No networking capabilities - Platform-dependent (primarily UNIX) - No directory structure versioning - Complex command syntax

Revision Control System (RCS) - 1982

Walter F. Tichy developed RCS at Purdue University as an improved alternative to SCCS. It gained widespread adoption in academic and research environments.



```
Version 1.1
```

Improvements over SCCS: - Reverse delta storage (more efficient for recent versions) - Better merge algorithms - Symbolic revision names - Enhanced keyword substitution - Simplified command interface - Improved performance

Technical Details: - File format: Text-based RCS files (.v extension) - Storage: Reverse deltas (latest version stored in full) - Locking: Exclusive file locking with optional unlocking - Branching: Basic branch support with numeric branch identifiers

Second Generation: Centralized VCS (1990s-2000s)

The next evolution addressed the limitations of single-file, single-user systems by introducing client-server architectures that enabled team collaboration.

Concurrent Versions System (CVS) - 1990

Originally developed by Dick Grune in 1986 and later rewritten by Brian Berliner, CVS was the first system to enable concurrent development through a copy-modify-merge approach rather than strict locking.

```
CVS Client-Server Architecture


    CVS Server

     Repository


            Network Protocol (pserver, ssh, etc.)



  Developer 1       Developer 2          Developer 3
  Working Copy      Working Copy         Working Copy
```

Revolutionary Features: - Client-server architecture - Concurrent development (copy-modify-merge) - Network protocols for remote access - Directory structure versioning - Team collaboration support - Cross-platform compatibility

CVS Workflow: 1. **Checkout**: Get a working copy from the repository 2. **Update**: Sync with latest changes from others 3. **Edit**: Make changes locally 4. **Commit**: Send changes back to the repository

Limitations: - No atomic commits (file-by-file commits) - No support for file/directory renames or moves - No changesets or commit integrity - Weak branching and merging capabilities - No binary file versioning

Apache Subversion (SVN) - 2000

Developed by CollabNet (with key developers including Jim Blandy, Karl Fogel, and Brian Behlendorf), SVN was designed specifically to address CVS's limitations while maintaining a similar workflow.

```
SVN Repository Structure


   Repository

      trunk/      branches/    tags/


      Revision History (r1, r2, r3...)



   Working Copy

      Files

      .svn/     (metadata)
```

Major Improvements: - Atomic commits (all-or-nothing transactions) - True version history for directories - Efficient binary file handling - File/directory renames and moves - Property metadata support - Improved branching and tagging model - Path-based authorization

Technical Architecture: - **Storage Backends**: FSFS (file system) or Berkeley DB - **Network Protocols**: HTTP/WebDAV, custom svn:// protocol - **Branching**: Copy-on-write mechanism (cheap copies) - **Merging**: Enhanced merge tracking (post-1.5)

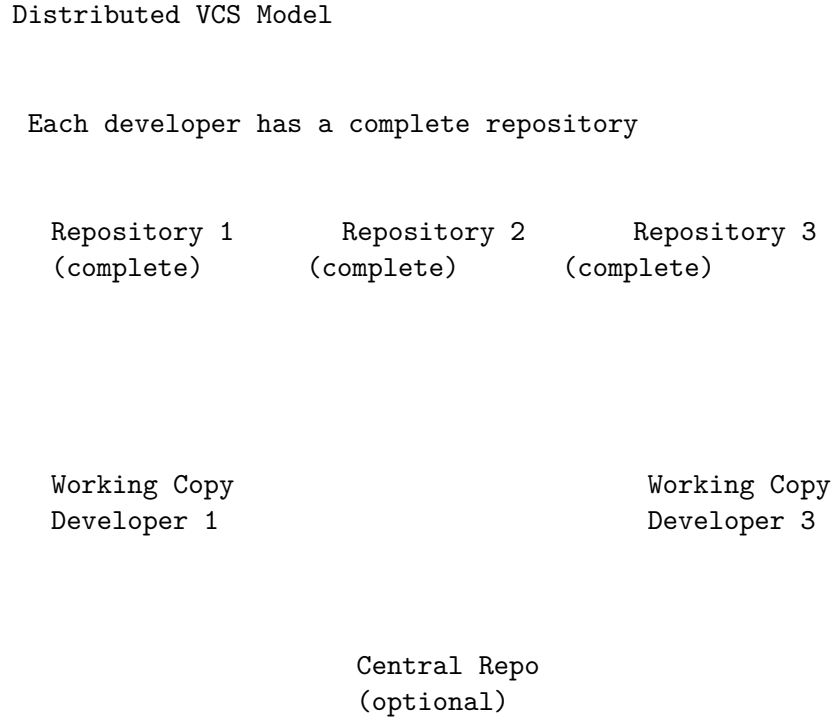
Enterprise Features: - Integration with LDAP and Active Directory - Path-based access control - Hook scripts for automation - Robust Windows support

Third Generation: Distributed VCS (2000s-Present)

The third major paradigm shift came with distributed version control systems, which eliminated the dependency on a central server and enabled offline work.

BitKeeper - 2000

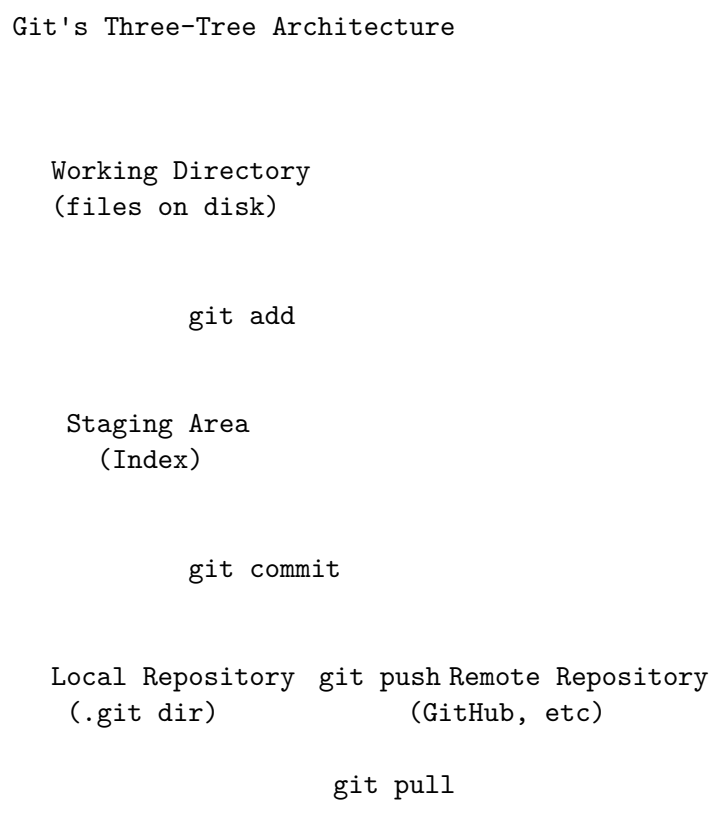
BitKeeper, developed by BitMover (founded by Larry McVoy), was one of the first commercially successful distributed VCS tools. It was temporarily used for Linux kernel development before licensing issues led to the creation of Git.



Key Innovations: - Full repository distribution - Offline operations - Peer-to-peer synchronization - Changeset-based history - Enhanced branching and merging

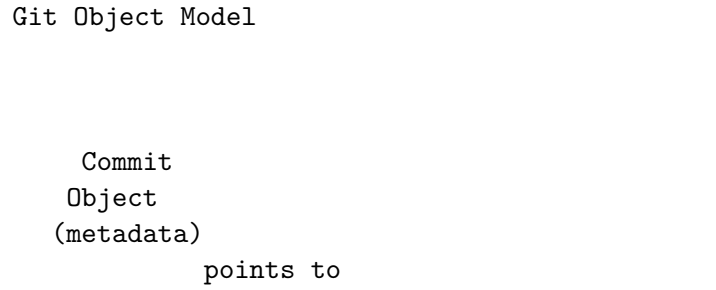
Git - 2005

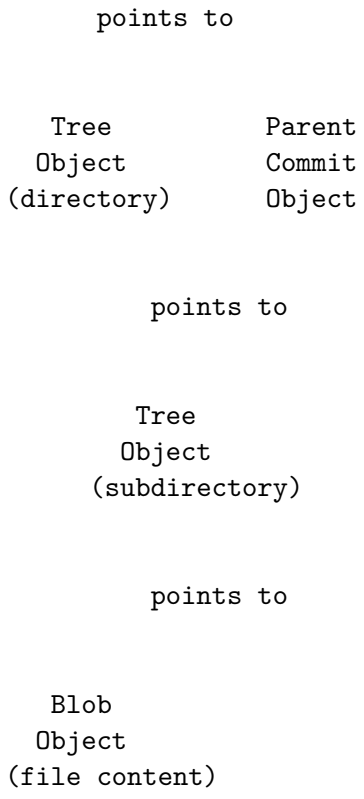
Created by Linus Torvalds for Linux kernel development, Git has become the dominant version control system worldwide. It was designed for speed, data integrity, and support for distributed, non-linear workflows.



Core Design Principles: 1. **Distributed**: Every clone is a full repository with complete history 2. **Performance**: Optimized for speed, even with large repositories 3. **Integrity**: SHA-1 content addressing ensures data integrity 4. **Non-linear**: First-class support for branching and merging

Git's Object Model:





Technical Innovations: - Content-addressable storage - Directed acyclic graph (DAG) history - Lightweight branching - Flexible staging area - Powerful merging capabilities - Cryptographic integrity verification

Mercurial - 2005

Developed by Matt Mackall around the same time as Git, Mercurial offered a more user-friendly alternative with similar distributed capabilities.

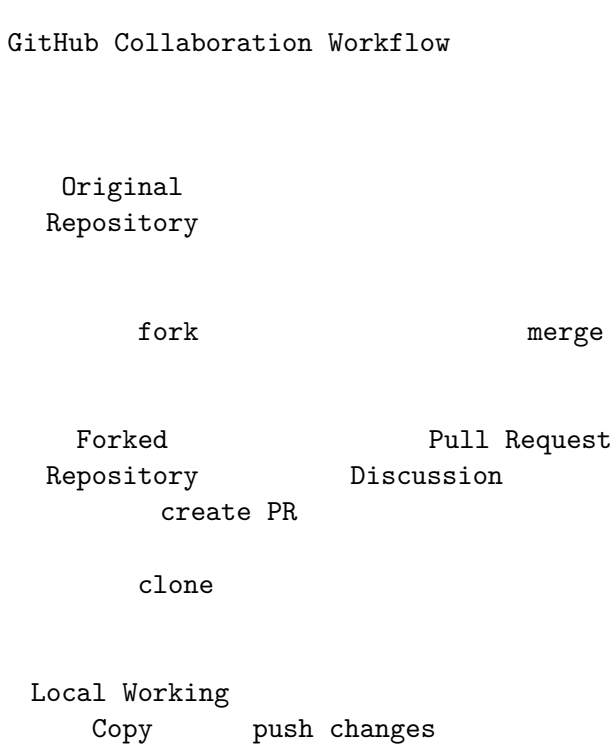
Key Features: - Python-based implementation - Simpler command interface - Extension system - Efficient handling of large files - Built-in web interface - Cross-platform consistency

The Platform Era (2008-Present)

The most recent evolution has been the transformation of version control systems into collaborative development platforms.

GitHub - 2008

Founded by Tom Preston-Werner, Chris Wanstrath, and PJ Hyett, GitHub transformed Git from a command-line tool into a social coding platform.



Revolutionary Features: - Pull requests for code review - Issue tracking integration - Wiki documentation - Social networking for developers - Web-based code browsing and editing - Continuous integration hooks

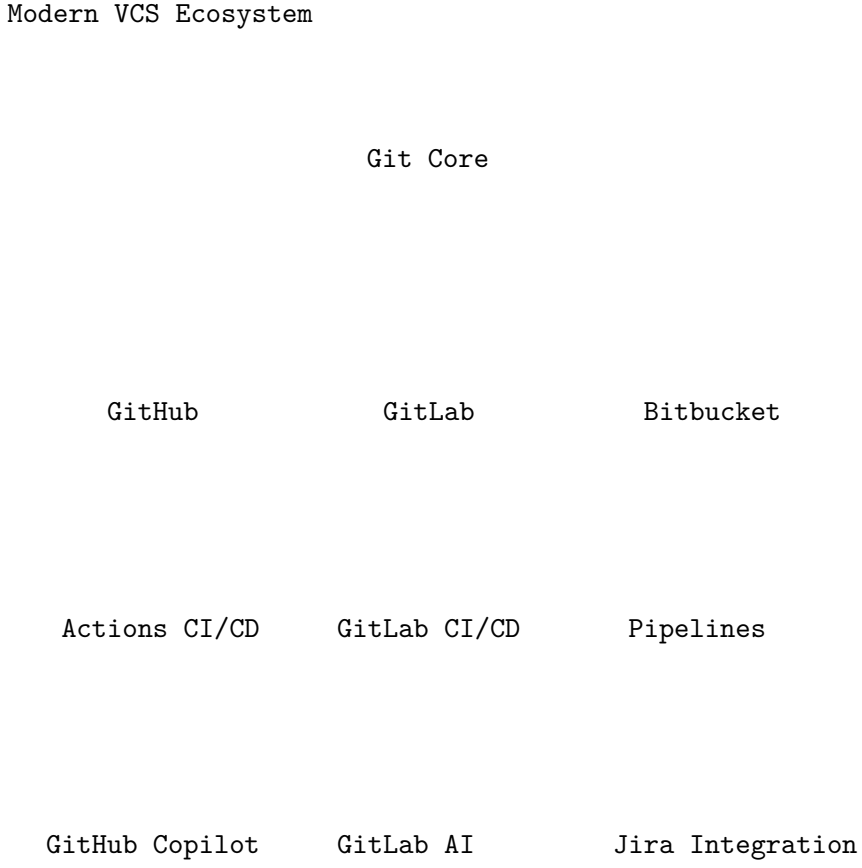
GitLab, Bitbucket, and Others

Following GitHub's success, other platforms emerged with different focuses:

- **GitLab**: DevOps platform with built-in CI/CD
- **Bitbucket**: Atlassian ecosystem integration
- **Azure DevOps**: Microsoft's enterprise development solution

Modern VCS Landscape (2020s)

Today's version control ecosystem is dominated by Git-based platforms, each offering unique features:



Current Market Statistics

System	Market Share	Primary Users	Key Strength
Git	~90%	All sectors	Flexibility, ecosystem
SVN	~5%	Enterprise	Simplicity, access control
Mercurial	~2%	Mozilla, Facebook	Performance with large repos
Perforce	~2%	Gaming, hardware	Binary file handling
Others	~1%	Specialized niches	Domain-specific features

Emerging Trends

AI-Powered Development

Modern version control platforms are integrating AI capabilities:

- **GitHub Copilot**: AI pair programming

- **Code Review Automation**: Intelligent suggestions
- **Security Scanning**: Automated vulnerability detection
- **Dependency Management**: Smart updates and alerts

Cloud-Native Development

The integration of version control with cloud development environments:

- **GitHub Codespaces**: Cloud-based development environments
- **GitPod**: Ephemeral development environments
- **Cloud-based CI/CD**: Integrated testing and deployment
- **Infrastructure as Code**: Version-controlled infrastructure

Security and Compliance

Enhanced security features in modern VCS:

- **Branch Protection Rules**: Enforce code review
- **Signed Commits**: Cryptographic verification
- **Secret Scanning**: Prevent credential leaks
- **Dependency Scanning**: Vulnerability detection
- **Compliance Automation**: Audit trails and reporting

Conclusion

The evolution of source code control systems reflects the broader transformation of software development from individual craft to collaborative engineering discipline. From SCCS's simple file tracking to Git's distributed architecture and GitHub's social coding platform, each generation has built upon previous innovations while addressing new challenges.

Today's developers benefit from: - **Distributed workflows** enabling global collaboration - **Integrated platforms** combining code, issues, and CI/CD - **AI assistance** for code review and generation - **Security features** protecting intellectual property - **Cloud-native tools** supporting modern development practices

As we look toward the future, version control systems will continue evolving to support emerging paradigms like AI-assisted development, quantum computing, and edge-native applications. The fundamental principles of tracking changes, enabling collaboration, and maintaining code integrity remain constant, but their implementation continues to advance with technology.

Further Reading

1. Pro Git Book
2. Version Control by Example
3. The Architecture of Open Source Applications - Git
4. Git Internals - Plumbing and Porcelain
5. A Visual Guide to Version Control

Introduction

Chapter 1: Introduction to Version Control

What is Version Control?

Version control is a system that records changes to files over time so that you can recall specific versions later. It's like having a time machine for your code and documents, allowing you to:

- Track changes made to files
- Revert files back to a previous state
- Compare changes over time
- See who last modified something that might be causing problems
- Collaborate with others without conflicts

Why Do We Need Version Control?

The Problem Without Version Control

Imagine working on a project without version control:

```
my-project/
   index.html
   index_backup.html
   index_final.html
   index_final_v2.html
   index_really_final.html
   index_really_final_fixed.html
```

This approach leads to: - **Confusion**: Which file is the actual current version? - **Lost work**: Accidentally overwriting important changes - **No history**: Can't see what changed and when - **Collaboration nightmares**: Multiple people editing the same files

The Solution: Version Control Systems

Version control systems solve these problems by: - Maintaining a complete history of changes - Allowing multiple people to work on the same project - Providing mechanisms to merge changes - Enabling you to revert to previous versions - Tracking who made what changes and when

History of Version Control Systems

First Generation: Local Version Control

- **Examples**: RCS (Revision Control System)
- **How it works**: Keeps patch sets (differences between files) on local disk
- **Limitations**: No collaboration, single point of failure

Second Generation: Centralized Version Control

- **Examples**: CVS (Concurrent Versions System), Subversion (SVN), Perforce
- **How it works**: Single server contains all versioned files, clients check out files
- **Advantages**: Team collaboration, administrators have fine-grained control
- **Limitations**: Single point of failure, requires network connection

Third Generation: Distributed Version Control

- **Examples**: Git, Mercurial, Bazaar
- **How it works**: Every client has a complete copy of the project history
- **Advantages**: No single point of failure, works offline, flexible workflows

Why Git?

Git has become the de facto standard for version control due to several key advantages:

1. Distributed Nature

- Every developer has a complete copy of the project history
- Work offline without any limitations
- No single point of failure

2. Performance

- Nearly all operations are local and therefore fast
- Branching and merging are lightweight operations
- Handles large projects efficiently

3. Data Integrity

- Everything is checksummed using SHA-1 hash
- Impossible to change file contents without Git knowing
- Complete data integrity

4. Flexible Workflows

- Supports various development workflows
- Non-linear development through branching
- Distributed development models

5. Strong Community and Ecosystem

- Widely adopted in the industry
- Extensive tooling and integrations
- Large community support

Git vs Other Version Control Systems

Git vs Subversion (SVN)

Feature	Git	SVN
Architecture	Distributed	Centralized
Offline work	Full functionality	Limited
Branching	Lightweight, fast	Heavy, slow
Merging	Advanced algorithms	Basic merging
Storage	Snapshots	Differences
Learning curve	Steeper	Gentler

Git vs Mercurial

Feature	Git	Mercurial
Performance	Faster	Good
Complexity	More complex	Simpler
Flexibility	More flexible	More opinionated
Windows support	Good (improved)	Better
Adoption	Wider	Smaller

Key Concepts to Remember

1. **Version control is essential** for any serious development work
2. **Git is distributed**, meaning every copy is a full backup
3. **Git tracks snapshots**, not differences
4. **Git is fast** because most operations are local
5. **Git ensures data integrity** through checksumming

What's Next?

In the next chapter, we'll dive into Git fundamentals, including: - Installing and configuring Git - Understanding the three states of Git - Creating your first repository - Basic Git workflow

Exercises

1. **Research Exercise**: Look up a project you use regularly (like a web browser or text editor) and find their version control system. What do they use and why?
2. **Reflection**: Think about a time when you lost work or had trouble collaborating on a project. How could version control have helped?
3. **Comparison**: Create a simple text file and manually create "versions" by copying and renaming it. Make changes to each version. Now imagine doing this with hundreds of files and multiple people. What problems do you foresee?

Summary

Version control systems are fundamental tools for managing changes in software development and beyond. Git's distributed nature, performance, and flexibility have made it the industry standard. Understanding these foundational concepts prepares you for learning Git's practical applications in the following chapters.

The journey from manual file copying to sophisticated distributed version control represents one of the most important advances in software development practices. As we move forward, you'll see how Git's design principles solve real-world collaboration and code management challenges.

Git Fundamentals

Chapter 2: Git Fundamentals

Installing Git

macOS

```
# Using Homebrew (recommended)
brew install git

# Using Xcode Command Line Tools
xcode-select --install

# Verify installation
git --version
```

Linux (Ubuntu/Debian)

```
# Update package list
sudo apt update

# Install Git
sudo apt install git

# Verify installation
git --version
```

Windows

1. Download from git-scm.com
2. Run the installer with default settings
3. Open Git Bash or Command Prompt
4. Verify: `git --version`

Initial Configuration

Before using Git, you need to configure your identity:

```
# Set your name and email (required)
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"

# Set default branch name
git config --global init.defaultBranch main

# Set default editor (optional)
git config --global core.editor "code --wait"  # VS Code
git config --global core.editor "vim"          # Vim
git config --global core.editor "nano"         # Nano

# View your configuration
git config --list
git config user.name
git config user.email
```

Configuration Levels

Git has three levels of configuration:

1. **System** (`--system`): Applies to all users on the system
2. **Global** (`--global`): Applies to all repositories for the current user
3. **Local** (default): Applies only to the current repository

The Three States of Git

Understanding Git's three states is crucial:

1. Working Directory

- Your current project files
- Where you make changes
- Untracked or modified files

2. Staging Area (Index)

- Files prepared for the next commit
- A snapshot of what will be committed
- Use `git add` to stage files

3. Git Repository

- Committed snapshots of your project
- Permanent record of changes
- Use `git commit` to save to repository

```
Working Directory → Staging Area → Git Repository
     (modify)          (stage)        (commit)
```

Repository Initialization

Creating a New Repository

```
# Create a new directory
mkdir my-project
cd my-project

# Initialize Git repository
git init

# Check status
git status
```

Cloning an Existing Repository

```
# Clone from GitHub
git clone https://github.com/user/repository.git

# Clone to specific directory
git clone https://github.com/user/repository.git my-folder

# Clone specific branch
git clone -b branch-name https://github.com/user/repository.git
```

Basic Git Workflow

1. Check Status

```
git status
```

2. Add Files to Staging Area

```
# Add specific file
git add filename.txt

# Add all files in current directory
git add .

# Add all files with specific extension
git add *.js

# Add all files (including deleted ones)
git add -A
```

3. Commit Changes

```
# Commit with message
git commit -m "Add initial project files"

# Commit with detailed message
git commit -m "Add user authentication

- Implement login functionality
- Add password validation
- Create user session management"

# Add and commit in one step (tracked files only)
git commit -am "Update existing files"
```

4. View History

```
# View commit history
git log

# Compact view
git log --oneline

# Graphical view
git log --graph --oneline --all

# Show specific number of commits
git log -5
```

Understanding Git Objects

Git stores data as snapshots, not differences. Every commit contains:

Commit Object

- Points to a tree object
- Contains author and committer info
- Contains commit message
- Points to parent commit(s)

Tree Object

- Represents directory structure
- Points to blob objects (files)
- Points to other tree objects (subdirectories)

Blob Object

- Contains file content
- Identified by SHA-1 hash of content

File States in Git

Files in your working directory can be in one of these states:

Untracked

- New files not yet added to Git
- Git doesn't know about them

Tracked

Files that Git knows about, which can be:

Unmodified

- File hasn't changed since last commit
- Clean working directory

Modified

- File has been changed but not staged
- Changes exist in working directory

Staged

- File has been added to staging area
- Ready for next commit

The .git Directory

When you run `git init`, Git creates a `.git` directory containing:

```
.git/
  HEAD              # Points to current branch
  config            # Repository configuration
  description       # Repository description
  hooks/            # Client/server-side hook scripts
  info/             # Global exclude file
  objects/          # Git objects (commits, trees, blobs)
  refs/             # References (branches, tags)
  index             # Staging area information
```

Common Git Commands Summary

```
# Repository setup
git init                     # Initialize repository
git clone <url>             # Clone repository

# Configuration
git config --global user.name "Name"
git config --global user.email "email"

# Basic workflow
git status                   # Check repository status
git add <file>              # Stage file
git add .                   # Stage all files
git commit -m "message"     # Commit changes
git log                     # View history

# Information
git show                    # Show last commit
```

```
git diff                    # Show unstaged changes
git diff --staged           # Show staged changes
```

Best Practices

Commit Messages

- Use present tense ("Add feature" not "Added feature")
- Keep first line under 50 characters
- Use imperative mood ("Fix bug" not "Fixes bug")
- Provide context in the body if needed

When to Commit

- Commit early and often
- Each commit should represent a logical change
- Don't commit broken code
- Commit related changes together

Repository Organization

- Use meaningful directory structure
- Keep repositories focused on single projects
- Use .gitignore for files that shouldn't be tracked

Exercises

Exercise 1: First Repository

1. Create a new directory called `git-practice`
2. Initialize it as a Git repository
3. Create a `README.md` file with project description
4. Add and commit the file
5. Check the repository status and history

Exercise 2: Multiple Commits

1. Create three files: `index.html`, `style.css`, `script.js`
2. Add some basic content to each
3. Stage and commit each file separately
4. View the commit history
5. Make changes to one file and commit again

Exercise 3: Configuration Practice

1. Check your current Git configuration
2. Set up a different editor if you haven't already
3. Create a repository-specific configuration
4. Compare global vs local settings

Troubleshooting Common Issues

"Author identity unknown"

```
# Solution: Configure user name and email
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

"Not a git repository"

```
# Solution: Initialize repository or navigate to existing one
git init
# or
cd /path/to/git/repository
```

Accidentally committed wrong files

```
# Solution: Amend the last commit (if not pushed)
git add correct-file.txt
git commit --amend
```

Summary

This chapter covered Git fundamentals including: - Installation and configuration - The three states of Git - Basic workflow (add, commit, status, log) - Understanding Git objects and file states - Best practices for commits and repository management

These fundamentals form the foundation for all Git operations. In the next chapter, we'll explore core Git operations in more detail, including working with files, viewing changes, and managing your repository effectively.

Understanding these concepts is crucial because they underpin everything else you'll do with Git. The three-state model, in particular, is key to understanding how Git manages your files and changes.

Core Operations

Chapter 3: Core Git Operations

Working with Files

Adding Files to Git

Git tracks files through explicit commands. Understanding how to add files effectively is crucial for good Git workflow.

```
# Add a specific file
git add filename.txt

# Add multiple specific files
git add file1.txt file2.txt file3.txt

# Add all files in current directory
git add .

# Add all files in the repository
git add -A

# Add all files with specific pattern
git add *.js
git add src/*.py
git add "*.txt"

# Add parts of a file interactively
git add -p filename.txt
```

Interactive Staging

Interactive staging allows you to stage parts of files:

```
# Interactive staging
git add -i

# Patch mode (stage hunks)
git add -p

# Edit mode (manually edit hunks)
```

```
git add -e
```

When using `git add -p`, you'll see options: - `y` - stage this hunk - `n` - do not stage this hunk - `s` - split the hunk into smaller hunks - `e` - manually edit the hunk - `q` - quit

Viewing Repository Status and Changes

Git Status

```
# Basic status
git status

# Short format
git status -s
git status --short

# Show ignored files too
git status --ignored
```

Status output meanings: - `??` - Untracked files - `A` - Added (staged) - `M` - Modified - `D` - Deleted - `R` - Renamed - `C` - Copied

Viewing Differences

```
# Show unstaged changes
git diff

# Show staged changes
git diff --staged
git diff --cached

# Show changes between commits
git diff HEAD~1 HEAD

# Show changes for specific file
git diff filename.txt

# Show word-level differences
git diff --word-diff

# Show statistics
git diff --stat
```

Viewing File History

```
# Show commits that modified a file
git log filename.txt

# Show changes to a file over time
git log -p filename.txt

# Show who changed what line
git blame filename.txt

# Show file content at specific commit
git show commit-hash:filename.txt
```

Committing Changes

Basic Commits

```
# Commit staged changes
git commit -m "Commit message"

# Commit with detailed message
git commit -m "Short description

Longer explanation of what this commit does and why.
Can include multiple paragraphs and bullet points:
- Feature A added
- Bug B fixed
- Performance improved"

# Add and commit tracked files in one step
git commit -am "Update existing files"

# Commit with editor for longer message
git commit
```

Amending Commits

```
# Change the last commit message
git commit --amend -m "New commit message"

# Add files to the last commit
git add forgotten-file.txt
```

```
git commit --amend --no-edit

# Amend with new message and files
git add new-file.txt
git commit --amend -m "Updated commit message"
```

Commit Message Best Practices

Structure

```
Short summary (50 chars or less)

More detailed explanation if needed. Wrap at 72 characters.
Explain what and why, not how.

- Use bullet points if helpful
- Reference issue numbers: Fixes #123
- Use imperative mood: "Add feature" not "Added feature"
```

Good Examples

```
git commit -m "Add user authentication system"

git commit -m "Fix memory leak in image processing

The image cache wasn't being cleared properly after processing,
causing memory usage to grow continuously. This fix ensures
the cache is cleared after each batch of images.

Fixes #456"
```

Working with .gitignore

Creating .gitignore

```
# Create .gitignore file
touch .gitignore

# Add patterns to ignore
echo "*.log" >> .gitignore
echo "node_modules/" >> .gitignore
echo ".env" >> .gitignore
```

Common .gitignore Patterns

```
# Compiled files
*.class
*.o
*.pyc
__pycache__/

# Logs
*.log
logs/

# Dependencies
node_modules/
vendor/

# Environment files
.env
.env.local

# IDE files
.vscode/
.idea/
*.swp
*.swo

# OS files
.DS_Store
Thumbs.db

# Build directories
build/
dist/
target/

# Temporary files
*.tmp
*.temp
~*
```

Advanced .gitignore Patterns

```
# Ignore all .txt files except important.txt
*.txt
!important.txt

# Ignore files in any directory named temp
temp/

# Ignore .log files in root directory only
/*.log

# Ignore all files in logs directory but keep the directory
logs/*
!logs/.gitkeep

# Use double asterisk for recursive matching
**/node_modules/
```

Managing Already Tracked Files

```
# Remove file from Git but keep in working directory
git rm --cached filename.txt

# Remove directory from Git but keep locally
git rm -r --cached directory/

# After adding to .gitignore, remove from tracking
git rm --cached *.log
git commit -m "Remove log files from tracking"
```

Viewing History and Logs

Basic Log Commands

```
# Show commit history
git log

# One line per commit
git log --oneline

# Show last 5 commits
git log -5
```

```
# Show commits with file changes
git log --stat

# Show commits with actual changes
git log -p
```

Advanced Log Formatting

```
# Custom format
git log --pretty=format:"%h - %an, %ar : %s"

# Graph view
git log --graph --oneline --all

# Show commits by author
git log --author="John Doe"

# Show commits in date range
git log --since="2023-01-01" --until="2023-12-31"

# Show commits that modified specific file
git log -- filename.txt

# Show commits with specific message
git log --grep="bug fix"
```

Useful Log Aliases

Add these to your Git configuration:

```
git config --global alias.lg "log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Cres

git config --global alias.hist "log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=sho
```

Undoing Changes

Unstaging Files

```
# Unstage specific file
git reset HEAD filename.txt

# Unstage all files
git reset HEAD

# In newer Git versions
git restore --staged filename.txt
```

Discarding Changes

```
# Discard changes in working directory
git checkout -- filename.txt

# Discard all changes in working directory
git checkout -- .

# In newer Git versions
git restore filename.txt
git restore .
```

Reverting Commits

```
# Create new commit that undoes a previous commit
git revert commit-hash

# Revert without creating commit (stage changes)
git revert --no-commit commit-hash

# Revert merge commit
git revert -m 1 merge-commit-hash
```

File Operations

Renaming and Moving Files

```
# Rename file
git mv old-name.txt new-name.txt

# Move file to directory
git mv file.txt directory/

# Equivalent manual process
mv old-name.txt new-name.txt
git add new-name.txt
git rm old-name.txt
```

Removing Files

```
# Remove file from Git and working directory
git rm filename.txt

# Remove file from Git but keep in working directory
git rm --cached filename.txt

# Remove directory
git rm -r directory/

# Force removal (if file is modified)
git rm -f filename.txt
```

Practical Exercises

Exercise 1: File Management

1. Create a new repository
2. Add several files with different content
3. Practice staging files individually and in groups
4. Create commits with meaningful messages
5. View the history in different formats

Exercise 2: Working with Changes

1. Modify several files
2. Use `git diff` to see changes

3. Stage some changes but not others
4. Use `git diff --staged` to see staged changes
5. Commit and view the result

Exercise 3: .gitignore Practice

1. Create files that should be ignored (logs, temp files)
2. Create and configure .gitignore
3. Test that ignored files don't appear in status
4. Practice ignoring already-tracked files

Exercise 4: History Exploration

1. Create several commits with different types of changes
2. Use various `git log` commands to explore history
3. Practice finding specific commits by author, date, or message
4. Use `git blame` to see who changed what

Common Workflows

Daily Development Workflow

```
# Start of day
git status                    # Check current state
git pull                      # Get latest changes

# During development
git add .                     # Stage changes
git commit -m "Description"   # Commit changes
git push                      # Share changes

# End of day
git status                    # Ensure clean state
git log --oneline -5          # Review recent work
```

Feature Development Workflow

```
# Start feature
git checkout -b feature-name
git add .
git commit -m "Start feature implementation"
```

```
# During development
git add modified-files
git commit -m "Implement part of feature"

# Complete feature
git add .
git commit -m "Complete feature implementation"
git checkout main
git merge feature-name
```

Summary

This chapter covered core Git operations including: - Adding and staging files effectively - Viewing repository status and changes - Creating meaningful commits - Managing ignored files with .gitignore - Exploring repository history - Undoing changes safely - Basic file operations

These operations form the daily toolkit for Git users. Mastering them enables efficient version control and sets the foundation for more advanced Git features covered in subsequent chapters.

The key to becoming proficient with Git is practicing these core operations until they become second nature. Each operation serves a specific purpose in the Git workflow, and understanding when and how to use each one is crucial for effective version control.

Git Internals

Chapter 4: Understanding Git Internals

Git's Data Model

Understanding how Git stores and manages data internally is crucial for mastering advanced Git operations. Git's design is elegant and simple once you understand the underlying concepts.

Git as a Content-Addressable Filesystem

Git is fundamentally a content-addressable filesystem with a VCS user interface written on top. This means: - Every piece of content is stored based on its hash - The hash serves as both identifier and integrity check - Content is immutable once stored

Git Objects

Git stores all data as objects in the `.git/objects` directory. There are four types of objects:

1. Blob Objects

Blobs store file content without any metadata.

```
# Create a blob object manually
echo "Hello, World!" | git hash-object -w --stdin

# View blob content
git cat-file -p <blob-hash>

# Check object type
git cat-file -t <blob-hash>
```

Example:

```
$ echo "Hello, Git!" | git hash-object -w --stdin
8d0e41234f24b6da002d962a26c2495ea16a425f

$ git cat-file -p 8d0e41234f24b6da002d962a26c2495ea16a425f
Hello, Git!
```

```
$ git cat-file -t 8d0e41234f24b6da002d962a26c2495ea16a425f
blob
```

2. Tree Objects

Trees store directory structure and point to blobs and other trees.

```
# View tree object
git cat-file -p <tree-hash>

# Create tree object manually
git write-tree
```

Tree object format:

```
100644 blob a906cb2a4a904a152e80877d4088654daad0c859    README.md
100644 blob 8d1c8b69c050f2424c26d2073fac4b4f2c47c4f8    index.html
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0    src
```

3. Commit Objects

Commits point to trees and contain metadata.

```
# View commit object
git cat-file -p <commit-hash>

# Show commit structure
git show --format=raw <commit-hash>
```

Commit object format:

```
tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
parent 1a410efbd13591db07496601ebc7a059dd55cfe9
author John Doe <john@example.com> 1234567890 +0000
committer John Doe <john@example.com> 1234567890 +0000

Initial commit message
```

4. Tag Objects

Tags create permanent references to specific commits.

```
# Create annotated tag
git tag -a v1.0 -m "Version 1.0"
```

```
# View tag object
git cat-file -p v1.0
```

SHA-1 Hashing

Git uses SHA-1 hashing to identify objects:

How Hashes are Generated

```
# For blobs: hash of "blob <size>\0<content>"
echo -n "Hello, Git!" | git hash-object --stdin

# Manual calculation
echo -n "blob 11\0Hello, Git!" | sha1sum
```

Hash Properties

- **Deterministic**: Same content always produces same hash
- **Unique**: Extremely unlikely for different content to have same hash
- **Integrity**: Any change in content changes the hash

References and HEAD

Understanding References

References (refs) are pointers to commits stored in `.git/refs/`:

```
# View all references
git show-ref

# View specific reference
cat .git/refs/heads/main

# View HEAD
cat .git/HEAD
```

Types of References

Branch References

```
# Branch refs are in .git/refs/heads/
ls .git/refs/heads/

# Each file contains a commit hash
cat .git/refs/heads/main
```

Tag References

```
# Tag refs are in .git/refs/tags/
ls .git/refs/tags/

# Lightweight tags point directly to commits
# Annotated tags point to tag objects
```

Remote References

```
# Remote refs are in .git/refs/remotes/
ls .git/refs/remotes/origin/
```

HEAD Reference

HEAD is a symbolic reference pointing to the current branch:

```
# View HEAD
cat .git/HEAD
# Output: ref: refs/heads/main

# In detached HEAD state
git checkout <commit-hash>
cat .git/HEAD
# Output: <commit-hash>
```

The Git Directory Structure

Complete .git Directory Layout

```
.git/
```

```
HEAD                    # Current branch reference
config                  # Repository configuration
description             # Repository description
index                   # Staging area (binary file)
hooks/                  # Hook scripts
    pre-commit
    post-commit
    ...
info/                   # Additional repository info
    exclude             # Local ignore patterns
objects/                # Object database
    01/
    02/
    ...
    info/
    pack/               # Packed objects
refs/                   # References
    heads/              # Branch references
    tags/               # Tag references
    remotes/            # Remote references
logs/                   # Reference logs (reflog)
    HEAD
    refs/
```

The Index (Staging Area)

The index is a binary file that stores staging area information:

```bash
# View index contents
git ls-files --stage

# View index in detail
git ls-files --debug
```

Index entry format:

```
100644 a906cb2a4a904a152e80877d4088654daad0c859 0    README.md
100644 8d1c8b69c050f2424c26d2073fac4b4f2c47c4f8 0    index.html
```

How Git Stores Data

Object Storage

Objects are stored in `.git/objects/` using the first two characters of the hash as directory name:

```
# Hash: a906cb2a4a904a152e80877d4088654daad0c859
# Stored as: .git/objects/a9/06cb2a4a904a152e80877d4088654daad0c859

# View object directly (compressed)
cat .git/objects/a9/06cb2a4a904a152e80877d4088654daad0c859

# Decompress and view
git cat-file -p a906cb2a4a904a152e80877d4088654daad0c859
```

Pack Files

Git optimizes storage using pack files:

```
# Trigger garbage collection and packing
git gc

# View pack files
ls .git/objects/pack/

# View pack contents
git verify-pack -v .git/objects/pack/pack-*.idx
```

Plumbing vs Porcelain Commands

Git commands are divided into two categories:

Porcelain Commands (User-Friendly)

- `git add`, `git commit`, `git push`
- High-level commands for daily use
- Hide internal complexity

Plumbing Commands (Low-Level)

- `git hash-object`, `git cat-file`, `git write-tree`
- Direct access to Git internals
- Used for scripting and understanding

Useful Plumbing Commands

```
# Object manipulation
git hash-object -w <file>      # Create blob object
git cat-file -p <hash>         # View object content
git cat-file -t <hash>         # View object type
git cat-file -s <hash>         # View object size

# Tree manipulation
git write-tree                 # Create tree from index
git read-tree <tree-hash>      # Read tree into index

# Reference manipulation
git update-ref refs/heads/branch <commit-hash>
git symbolic-ref HEAD refs/heads/branch

# Index manipulation
git update-index --add <file>
git ls-files --stage
```

Practical Examples

Example 1: Creating Objects Manually

```
# Create a blob
echo "Hello, Git internals!" | git hash-object -w --stdin
# Output: 7c4a013e52c76442ab80ee5572399a5a4c3f4e5f

# Create a tree
git update-index --add --cacheinfo 100644 7c4a013e52c76442ab80ee5572399a5a4c3f4e5f hello.txt
git write-tree
# Output: 68aba62e560c0ebc3396e8ae9335232cd93a3f60

# Create a commit
echo "First commit" | git commit-tree 68aba62e560c0ebc3396e8ae9335232cd93a3f60
# Output: 166ae0c4d3f420721acbb115cc33848dfcc2121a
```

Example 2: Exploring Object Relationships

```
# Start with a commit
git cat-file -p HEAD

# Follow the tree
```

```
git cat-file -p <tree-hash>

# View a blob
git cat-file -p <blob-hash>

# Trace the parent chain
git cat-file -p HEAD^
git cat-file -p HEAD^^
```

Example 3: Understanding Branches

```
# Create branch manually
git update-ref refs/heads/new-branch HEAD

# Verify branch creation
git branch

# Switch to branch
git symbolic-ref HEAD refs/heads/new-branch
```

Git's Efficiency

Delta Compression

Git uses delta compression for efficiency: - Similar objects are stored as deltas - Pack files contain base objects and deltas - Reduces storage space significantly

Deduplication

Git automatically deduplicates content: - Identical files share the same blob object - Moving/copying files doesn't duplicate content - Only metadata changes

Debugging with Internals Knowledge

Finding Corrupted Objects

```
# Check repository integrity
git fsck

# Find dangling objects
git fsck --unreachable
```

```
# Recover lost commits
git reflog
git fsck --lost-found
```

Understanding Performance Issues

```
# Check repository size
du -sh .git

# Analyze pack files
git count-objects -v

# Find large objects
git rev-list --objects --all | git cat-file --batch-check='%(objecttype) %(objectname) %(obj
```

Exercises

Exercise 1: Object Exploration

1. Create a simple file and add it to Git
2. Find the blob hash and examine the object
3. Make a commit and explore the commit object
4. Trace the relationships between commit, tree, and blob

Exercise 2: Manual Object Creation

1. Use plumbing commands to create a blob object
2. Create a tree object containing the blob
3. Create a commit object pointing to the tree
4. Update a branch reference to point to your commit

Exercise 3: Repository Analysis

1. Analyze your repository's object database
2. Find the largest objects
3. Understand the pack file structure
4. Use `git fsck` to verify integrity

Advanced Topics

Object Packing

```
# Force packing
git repack -ad

# Analyze pack efficiency
git gc --aggressive

# Unpack objects for inspection
git unpack-objects < .git/objects/pack/pack-*.pack
```

Custom Hash Functions

Git is transitioning from SHA-1 to SHA-256:

```
# Create repository with SHA-256
git init --object-format=sha256

# Check hash function
git config core.repositoryformatversion
```

Summary

Understanding Git internals provides: - **Deeper comprehension** of Git operations - **Better debugging** capabilities - **Confidence** in advanced operations - **Optimization** knowledge for large repositories

Key concepts covered: - Git's object model (blob, tree, commit, tag) - SHA-1 hashing and content addressing - References and HEAD - The .git directory structure - Plumbing vs porcelain commands - Storage optimization techniques

This internal knowledge forms the foundation for understanding advanced Git features like rebasing, cherry-picking, and complex merge scenarios covered in later chapters. When Git behaves unexpectedly, understanding these internals helps you diagnose and fix issues effectively.

Branching

Chapter 5: Branching Fundamentals

What are Branches?

A branch in Git is simply a movable pointer to a specific commit. Unlike many other version control systems, Git branches are lightweight and fast to create, making branching and merging a core part of the Git workflow.

Key Concepts

- **Branch**: A movable pointer to a commit
- **HEAD**: A pointer to the current branch
- **Default Branch**: Usually `main` or `master`
- **Working Directory**: Reflects the content of the current branch

Why Use Branches?

Branches enable: - **Parallel Development**: Work on multiple features simultaneously - **Experimentation**: Try new ideas without affecting main code - **Collaboration**: Multiple developers working independently - **Release Management**: Maintain different versions - **Bug Fixes**: Isolate fixes from ongoing development

Creating and Switching Branches

Creating Branches

```
# Create a new branch
git branch feature-login

# Create and switch to new branch
git checkout -b feature-login

# Modern syntax (Git 2.23+)
git switch -c feature-login

# Create branch from specific commit
git branch feature-login abc1234
```

```
# Create branch from another branch
git branch feature-login develop
```

Switching Branches

```
# Switch to existing branch
git checkout feature-login

# Modern syntax
git switch feature-login

# Switch to previous branch
git checkout -
git switch -

# Switch and create if doesn't exist
git checkout -b feature-login
git switch -c feature-login
```

Branch Information

```
# List all branches
git branch

# List all branches with last commit
git branch -v

# List remote branches
git branch -r

# List all branches (local and remote)
git branch -a

# Show current branch
git branch --show-current

# Show branches merged into current branch
git branch --merged

# Show branches not merged into current branch
git branch --no-merged
```

Understanding Branch Pointers

Visualizing Branches

```
# View branch history graphically
git log --oneline --graph --all

# More detailed graph
git log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold
```

Example output:

```
* 2f8a1b3 - (HEAD -> feature-login) Add login form (2 hours ago) <John Doe>
* 1a2b3c4 - (main) Initial commit (1 day ago) <John Doe>
```

How Branches Work Internally

When you create a branch, Git creates a new pointer:

```
# View branch references
cat .git/refs/heads/main
cat .git/refs/heads/feature-login

# View HEAD
cat .git/HEAD
```

Branch Management

Renaming Branches

```
# Rename current branch
git branch -m new-name

# Rename specific branch
git branch -m old-name new-name

# Rename and update upstream
git branch -m old-name new-name
git push origin -u new-name
git push origin --delete old-name
```

Deleting Branches

```
# Delete merged branch
git branch -d feature-login

# Force delete unmerged branch
git branch -D feature-login

# Delete remote branch
git push origin --delete feature-login

# Delete local tracking branch
git branch -dr origin/feature-login
```

Cleaning Up Branches

```
# Remove remote-tracking branches that no longer exist
git remote prune origin

# Remove local branches that have been merged
git branch --merged | grep -v "\*\|main\|develop" | xargs -n 1 git branch -d
```

Working with Branches

Basic Branch Workflow

```
# 1. Create and switch to feature branch
git checkout -b feature-user-profile

# 2. Make changes and commit
echo "User profile code" > profile.js
git add profile.js
git commit -m "Add user profile functionality"

# 3. Switch back to main
git checkout main

# 4. Merge feature branch
git merge feature-user-profile

# 5. Delete feature branch
git branch -d feature-user-profile
```

Tracking Remote Branches

```
# Create local branch tracking remote branch
git checkout -b feature-login origin/feature-login

# Set upstream for existing branch
git branch -u origin/feature-login

# Push and set upstream
git push -u origin feature-login

# View tracking relationships
git branch -vv
```

Branch Naming Conventions

Common Patterns

```
# Feature branches
feature/user-authentication
feature/shopping-cart
feat/payment-integration

# Bug fix branches
bugfix/login-error
fix/memory-leak
hotfix/security-patch

# Release branches
release/v1.2.0
release/2023-q4

# Development branches
develop
staging
integration
```

Best Practices

- Use descriptive names
- Include issue numbers: `feature/123-user-login`
- Use consistent prefixes
- Keep names short but meaningful
- Use lowercase and hyphens

Branch Strategies

Feature Branch Strategy

```
# Start feature
git checkout main
git pull origin main
git checkout -b feature/new-dashboard

# Develop feature
# ... make changes and commits ...

# Finish feature
git checkout main
git pull origin main
git merge feature/new-dashboard
git push origin main
git branch -d feature/new-dashboard
```

Topic Branches

Short-lived branches for specific topics:

```
# Quick fix
git checkout -b fix-typo
echo "Fixed typo" > README.md
git commit -am "Fix typo in README"
git checkout main
git merge fix-typo
git branch -d fix-typo
```

Advanced Branch Operations

Branch Comparison

```
# Compare branches
git diff main..feature-branch

# Show commits in feature branch not in main
git log main..feature-branch

# Show commits in main not in feature branch
git log feature-branch..main
```

```
# Show commits in either branch but not both
git log main...feature-branch --left-right
```

Branch Statistics

```
# Show branch commit count
git rev-list --count feature-branch

# Show branch age
git log -1 --format="%cr" feature-branch

# Show branch author
git log -1 --format="%an" feature-branch
```

Stashing Changes

When switching branches with uncommitted changes:

```
# Stash current changes
git stash

# Switch branch
git checkout other-branch

# Switch back and restore changes
git checkout original-branch
git stash pop

# Stash with message
git stash save "Work in progress on feature X"

# List stashes
git stash list

# Apply specific stash
git stash apply stash@{1}
```

Practical Examples

Example 1: Feature Development

```
# Start new feature
git checkout main
git pull origin main
git checkout -b feature/user-notifications

# Implement feature
echo "Notification system" > notifications.js
git add notifications.js
git commit -m "Add notification system"

echo "Email notifications" > email.js
git add email.js
git commit -m "Add email notification support"

# Push feature branch
git push -u origin feature/user-notifications

# Create pull request (on GitHub)
# After review and approval, merge
git checkout main
git pull origin main
git branch -d feature/user-notifications
```

Example 2: Hotfix Workflow

```
# Critical bug found in production
git checkout main
git pull origin main
git checkout -b hotfix/security-vulnerability

# Fix the issue
echo "Security fix" > security.patch
git add security.patch
git commit -m "Fix security vulnerability"

# Deploy hotfix
git checkout main
git merge hotfix/security-vulnerability
git push origin main
git tag v1.0.1
git push origin v1.0.1
```

```
# Clean up
git branch -d hotfix/security-vulnerability
```

Troubleshooting Branch Issues

Common Problems

Cannot Switch Branch (Uncommitted Changes)

```
# Problem: You have uncommitted changes
git checkout other-branch
# error: Your local changes would be overwritten

# Solutions:
# 1. Commit changes
git add .
git commit -m "WIP: Save current work"

# 2. Stash changes
git stash
git checkout other-branch

# 3. Force checkout (loses changes)
git checkout -f other-branch
```

Branch Diverged from Remote

```
# Problem: Local and remote branches have diverged
git push
# error: Updates were rejected because the tip of your current branch is behind

# Solutions:
# 1. Pull and merge
git pull origin feature-branch

# 2. Pull and rebase
git pull --rebase origin feature-branch

# 3. Force push (dangerous)
git push --force-with-lease origin feature-branch
```

Branch Aliases

Useful Git aliases for branch operations:

```
# Add to ~/.gitconfig or use git config --global
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.sw switch
git config --global alias.nb "checkout -b"
git config --global alias.bd "branch -d"
git config --global alias.bD "branch -D"
```

Exercises

Exercise 1: Basic Branching

1. Create a new repository
2. Make initial commit on main branch
3. Create a feature branch
4. Make commits on both branches
5. Visualize the branch structure

Exercise 2: Branch Management

1. Create multiple feature branches
2. Practice switching between branches
3. Rename a branch
4. Delete merged and unmerged branches
5. Clean up old branches

Exercise 3: Collaborative Workflow

1. Simulate working with remote branches
2. Create local tracking branches
3. Push branches to remote
4. Practice upstream relationships

Best Practices Summary

1. **Create branches frequently** for features and fixes
2. **Use descriptive names** that explain the purpose
3. **Keep branches focused** on single features or fixes
4. **Merge or delete** branches when done

5. **Pull before creating** new branches from main
6. **Test before merging** to ensure quality
7. **Use consistent naming** conventions
8. **Document branch purpose** in commit messages

Summary

Branching is one of Git's most powerful features, enabling: - Parallel development workflows - Safe experimentation - Organized feature development - Effective collaboration

Key concepts covered: - Branch creation and management - Switching between branches - Branch naming conventions - Basic branching workflows - Troubleshooting common issues

Understanding branching fundamentals is essential for effective Git usage. In the next chapter, we'll explore merging strategies and conflict resolution, which are crucial for combining work from different branches.

Merging

Chapter 6: Merging and Conflict Resolution

Understanding Merging

Merging is the process of combining changes from different branches. Git provides several merge strategies to handle different scenarios effectively.

Types of Merges

1. **Fast-Forward Merge**: When target branch hasn't diverged
2. **Three-Way Merge**: When branches have diverged
3. **Squash Merge**: Combines all commits into a single commit
4. **Rebase Merge**: Replays commits on top of target branch

Fast-Forward Merges

When Fast-Forward Occurs

Fast-forward happens when the target branch hasn't moved since the feature branch was created:

```
main:     A---B
               \
feature:        C---D
```

After fast-forward merge:

```
main:     A---B---C---D
```

Performing Fast-Forward Merge

```
# Create and work on feature branch
git checkout -b feature-branch
echo "New feature" > feature.txt
git add feature.txt
git commit -m "Add new feature"

# Switch to main and merge
```

```
git checkout main
git merge feature-branch

# Output: Fast-forward merge
```

Controlling Fast-Forward Behavior

```
# Force fast-forward (fails if not possible)
git merge --ff-only feature-branch

# Prevent fast-forward (always create merge commit)
git merge --no-ff feature-branch

# Default behavior (fast-forward when possible)
git merge feature-branch
```

Three-Way Merges

When Three-Way Merge Occurs

When both branches have new commits since they diverged:

```
main:     A---B---E---F
               \
feature:        C---D
```

Git finds the common ancestor (B) and creates a merge commit:

```
main:     A---B---E---F---M
               \         /
feature:        C---D-----
```

Performing Three-Way Merge

```
# Scenario: Both branches have new commits
git checkout main
git merge feature-branch

# Git automatically creates merge commit
# Opens editor for merge commit message
```

Merge Commit Messages

Default merge commit message format:

```
Merge branch 'feature-branch' into main

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
```

Custom merge commit message:

```
git merge feature-branch -m "Merge feature: Add user authentication"
```

Merge Conflicts

What Causes Conflicts

Conflicts occur when: - Same lines modified in both branches - File deleted in one branch, modified in another - File renamed differently in both branches

Conflict Example

File content in main branch:

```
function greet(name) {
    return "Hello, " + name + "!";
}
```

File content in feature branch:

```
function greet(name) {
    return "Hi there, " + name + "!";
}
```

Conflict Markers

When conflict occurs, Git adds conflict markers:

```
function greet(name) {
<<<<<<< HEAD
    return "Hello, " + name + "!";
=======
    return "Hi there, " + name + "!";
>>>>>>> feature-branch
```

```
}
```

Conflict markers explained: - `<<<<<<< HEAD`: Start of current branch content - `=======`: Separator between conflicting content - `>>>>>>> branch-name`: End of merging branch content

Resolving Conflicts

Manual Resolution Process

1. **Identify conflicts**:

```
git status
# Shows files with conflicts
```

2. **Edit conflicted files**:

```
// Choose one version or combine both
function greet(name) {
    return "Hello there, " + name + "!";
}
```

3. **Stage resolved files**:

```
git add conflicted-file.js
```

4. **Complete the merge**:

```
git commit
# Or use: git merge --continue
```

Conflict Resolution Strategies

Strategy 1: Accept One Side

```
# Accept current branch (HEAD)
git checkout --ours conflicted-file.js

# Accept incoming branch
git checkout --theirs conflicted-file.js

# Stage the resolution
git add conflicted-file.js
```

Strategy 2: Manual Edit

```
# Edit file to resolve conflicts manually
vim conflicted-file.js

# Remove conflict markers and choose/combine content
# Stage the resolved file
git add conflicted-file.js
```

Strategy 3: Use Merge Tool

```
# Configure merge tool (one-time setup)
git config --global merge.tool vimdiff

# Launch merge tool
git mergetool

# Popular merge tools: vimdiff, meld, kdiff3, p4merge
```

Advanced Merge Strategies

Merge Strategies

Git provides several merge strategies:

```
# Recursive (default for two branches)
git merge -s recursive feature-branch

# Ours (always prefer our version)
git merge -s ours feature-branch

# Theirs (always prefer their version)
git merge -s theirs feature-branch

# Octopus (for multiple branches)
git merge -s octopus branch1 branch2 branch3
```

Merge Strategy Options

```
# Prefer our version for conflicts
git merge -X ours feature-branch

# Prefer their version for conflicts
git merge -X theirs feature-branch

# Ignore whitespace changes
git merge -X ignore-space-change feature-branch

# Ignore all whitespace
git merge -X ignore-all-space feature-branch
```

Squash Merging

What is Squash Merge

Squash merge combines all commits from feature branch into a single commit:

```
# Squash merge
git merge --squash feature-branch
git commit -m "Add complete user authentication feature"
```

When to Use Squash Merge

- Clean up messy commit history
- Combine related commits
- Maintain linear history
- Before merging to main branch

Squash Merge Example

```
# Feature branch has multiple commits
git log --oneline feature-auth
# a1b2c3d Fix typo in login form
# e4f5g6h Add password validation
# i7j8k9l Add login form
# m0n1o2p Start authentication feature

# Squash merge
git checkout main
git merge --squash feature-auth
```

```
git commit -m "Add user authentication system

- Login form with validation
- Password strength requirements
- Session management
- Error handling"
```

Aborting Merges

When to Abort

- Conflicts are too complex
- Wrong branch merged
- Need to prepare better

How to Abort

```
# During merge conflict resolution
git merge --abort

# Reset to state before merge
git reset --hard HEAD

# If merge commit already created
git reset --hard HEAD~1
```

Merge Tools

Popular Merge Tools

Command Line Tools

- **vimdiff**: Built into Vim
- **emacs**: Built into Emacs
- **kdiff3**: Cross-platform GUI tool

GUI Tools

- **Meld**: Linux/Windows GUI tool
- **P4Merge**: Perforce visual merge tool
- **Beyond Compare**: Commercial tool
- **VS Code**: Built-in merge conflict resolution

Configuring Merge Tools

```
# Configure VS Code as merge tool
git config --global merge.tool vscode
git config --global mergetool.vscode.cmd 'code --wait $MERGED'

# Configure Meld
git config --global merge.tool meld

# Configure P4Merge
git config --global merge.tool p4merge
git config --global mergetool.p4merge.path "/Applications/p4merge.app/Contents/MacOS/p4merge
```

Using Merge Tools

```
# Launch configured merge tool
git mergetool

# Use specific tool
git mergetool --tool=meld

# Don't prompt for each file
git mergetool --no-prompt
```

Best Practices for Merging

Before Merging

1. **Update target branch**:

```
git checkout main
git pull origin main
```

2. **Test feature branch**:

```
git checkout feature-branch
# Run tests, verify functionality
```

3. **Rebase if needed**:

```
git rebase main
```

During Merging

1. **Read conflict carefully**
2. **Understand both changes**
3. **Test after resolution**
4. **Write meaningful merge commit messages**

After Merging

1. **Test merged code**
2. **Delete feature branch**:

```
git branch -d feature-branch
```

3. **Push changes**:

```
git push origin main
```

Common Merge Scenarios

Scenario 1: Simple Feature Merge

```
# Feature completed, ready to merge
git checkout main
git pull origin main
git merge feature-user-profile
git push origin main
git branch -d feature-user-profile
```

Scenario 2: Conflict Resolution

```
# Merge with conflicts
git checkout main
git merge feature-branch
# CONFLICT: Merge conflict in file.js

# Resolve conflicts
vim file.js  # Edit and resolve
git add file.js
git commit

# Push merged changes
git push origin main
```

Scenario 3: Multiple Branch Merge

```
# Merge multiple related branches
git checkout main
git merge feature-frontend feature-backend
# Resolve any conflicts
git commit -m "Merge frontend and backend features"
```

Merge vs Rebase

When to Use Merge

- **Preserving context**: Keep branch history
- **Collaborative branches**: Multiple developers
- **Feature completion**: Clear feature boundaries
- **Public branches**: Already pushed branches

When to Use Rebase

- **Linear history**: Clean, straight-line history
- **Private branches**: Not yet pushed
- **Cleanup**: Before merging to main
- **Synchronization**: Update feature branch with main

Troubleshooting Merge Issues

Common Problems

Problem: Merge Conflicts Too Complex

```
# Solution: Abort and try different approach
git merge --abort

# Try rebasing first
git rebase main
# Resolve conflicts incrementally
git checkout main
git merge feature-branch
```

Problem: Wrong Files Merged

```
# Solution: Reset to before merge
git reset --hard HEAD~1

# Or use reflog to find previous state
git reflog
git reset --hard HEAD@{1}
```

Problem: Merge Tool Not Working

```
# Solution: Configure different tool
git config --global merge.tool vimdiff

# Or resolve manually
git status  # See conflicted files
vim conflicted-file.js  # Edit manually
git add conflicted-file.js
git commit
```

Exercises

Exercise 1: Fast-Forward Merge

1. Create a repository with initial commit
2. Create feature branch and add commits
3. Merge back to main (should be fast-forward)
4. Verify the history

Exercise 2: Three-Way Merge

1. Create feature branch from main
2. Add commits to both main and feature branch
3. Merge feature branch into main
4. Examine the merge commit

Exercise 3: Conflict Resolution

1. Create conflicting changes in same file
2. Attempt to merge and encounter conflicts
3. Resolve conflicts manually
4. Complete the merge

Exercise 4: Merge Tools

1. Install and configure a merge tool
2. Create merge conflicts
3. Use the merge tool to resolve conflicts
4. Compare with manual resolution

Summary

Merging is a fundamental Git operation that enables: - Combining work from different branches - Integrating features into main codebase - Collaborative development workflows - Maintaining project history

Key concepts covered: - Fast-forward vs three-way merges - Conflict identification and resolution - Merge strategies and options - Merge tools and their configuration - Best practices for successful merging

Understanding merging and conflict resolution is crucial for effective Git collaboration. The next chapter will explore advanced branching strategies that build upon these merging concepts to create robust development workflows.

Advanced Branching

Chapter 7: Advanced Branching Strategies

Introduction to Branching Strategies

Branching strategies are systematic approaches to organizing branches in a project. They define how teams create, name, merge, and delete branches to support different development workflows.

Why Branching Strategies Matter

- **Consistency**: Team follows same patterns
- **Collaboration**: Clear rules for working together
- **Quality**: Systematic testing and review processes
- **Deployment**: Organized release management
- **Maintenance**: Easier bug fixes and hotfixes

Git Flow

Git Flow is a branching model designed around project releases, created by Vincent Driessen.

Git Flow Branch Types

Main Branches

- **main/master**: Production-ready code
- **develop**: Integration branch for features

Supporting Branches

- **feature/**: New features
- **release/**: Prepare new releases
- **hotfix/**: Critical production fixes

Git Flow Workflow

1. Feature Development

```
# Start feature from develop
git checkout develop
git pull origin develop
git checkout -b feature/user-authentication

# Develop feature
# ... make commits ...

# Finish feature
git checkout develop
git merge --no-ff feature/user-authentication
git branch -d feature/user-authentication
git push origin develop
```

2. Release Process

```
# Start release branch
git checkout develop
git checkout -b release/1.2.0

# Prepare release (version bumps, documentation)
echo "1.2.0" > VERSION
git commit -am "Bump version to 1.2.0"

# Finish release
git checkout main
git merge --no-ff release/1.2.0
git tag -a v1.2.0 -m "Release version 1.2.0"

git checkout develop
git merge --no-ff release/1.2.0
git branch -d release/1.2.0

git push origin main develop --tags
```

3. Hotfix Process

```
# Start hotfix from main
git checkout main
git checkout -b hotfix/1.2.1

# Fix critical issue
echo "Critical fix" > hotfix.patch
git commit -am "Fix critical security issue"

# Finish hotfix
git checkout main
git merge --no-ff hotfix/1.2.1
git tag -a v1.2.1 -m "Hotfix version 1.2.1"

git checkout develop
git merge --no-ff hotfix/1.2.1
git branch -d hotfix/1.2.1

git push origin main develop --tags
```

Git Flow Tools

Git Flow Extension

```
# Install git-flow
# macOS: brew install git-flow
# Ubuntu: sudo apt-get install git-flow

# Initialize git-flow
git flow init

# Feature workflow
git flow feature start user-authentication
# ... develop feature ...
git flow feature finish user-authentication

# Release workflow
git flow release start 1.2.0
# ... prepare release ...
git flow release finish 1.2.0

# Hotfix workflow
git flow hotfix start 1.2.1
# ... fix issue ...
```

```
git flow hotfix finish 1.2.1
```

Git Flow Pros and Cons

Advantages

- Clear separation of concerns
- Systematic release process
- Good for scheduled releases
- Well-documented workflow

Disadvantages

- Complex for simple projects
- Overhead for continuous deployment
- Long-lived branches can cause conflicts
- Not suitable for rapid iterations

GitHub Flow

GitHub Flow is a simpler, more streamlined workflow designed for continuous deployment.

GitHub Flow Principles

1. **main branch is always deployable**
2. **Create descriptive branches**
3. **Push to named branches frequently**
4. **Open pull request when ready**
5. **Deploy after review**
6. **Merge after deployment**

GitHub Flow Workflow

```
# 1. Create feature branch from main
git checkout main
git pull origin main
git checkout -b feature/add-payment-processing

# 2. Make changes and push frequently
echo "Payment code" > payment.js
git add payment.js
git commit -m "Add payment processing logic"
```

```
git push -u origin feature/add-payment-processing

# 3. Open pull request (on GitHub)
# 4. Discuss and review code
# 5. Deploy to staging/production for testing
# 6. Merge pull request after approval

# 7. Clean up
git checkout main
git pull origin main
git branch -d feature/add-payment-processing
```

GitHub Flow Best Practices

```
# Use descriptive branch names
git checkout -b feature/improve-search-performance
git checkout -b fix/login-redirect-issue
git checkout -b docs/update-api-documentation

# Push early and often
git push origin feature-branch

# Keep branches focused and small
# One feature/fix per branch

# Write good commit messages
git commit -m "Add search indexing for better performance

- Implement Elasticsearch integration
- Add search result caching
- Update search API endpoints"
```

GitHub Flow Advantages

- **Simple**: Easy to understand and follow
- **Fast**: Quick iterations and deployments
- **Flexible**: Adapts to different project needs
- **Continuous**: Supports continuous deployment
- **Collaborative**: Built around pull requests

GitLab Flow

GitLab Flow combines Git Flow and GitHub Flow concepts with environment-specific branches.

GitLab Flow with Environment Branches

```
# Branch structure
main                    # Development
pre-production          # Staging environment
production             # Production environment

# Workflow
git checkout main
git checkout -b feature/new-dashboard

# Develop feature
git push origin feature/new-dashboard
# Create merge request to main

# After merge to main, promote to staging
git checkout pre-production
git merge main
git push origin pre-production

# After testing, promote to production
git checkout production
git merge pre-production
git push origin production
```

GitLab Flow with Release Branches

```
# For projects with scheduled releases
main                    # Development
2-3-stable             # Release branch for version 2.3
2-4-stable             # Release branch for version 2.4

# Create release branch
git checkout main
git checkout -b 2-4-stable
git push origin 2-4-stable

# Cherry-pick fixes to release branch
git checkout 2-4-stable
git cherry-pick <commit-hash>
git push origin 2-4-stable
```

Feature Branch Workflow

A simple workflow focusing on feature isolation.

Basic Feature Branch Workflow

```
# 1. Update main branch
git checkout main
git pull origin main

# 2. Create feature branch
git checkout -b feature/user-notifications

# 3. Develop feature
echo "Notification system" > notifications.js
git add notifications.js
git commit -m "Add notification system"

echo "Email notifications" > email.js
git add email.js
git commit -m "Add email notification support"

# 4. Push feature branch
git push -u origin feature/user-notifications

# 5. Create pull request
# 6. Code review and approval
# 7. Merge to main
git checkout main
git pull origin main

# 8. Clean up
git branch -d feature/user-notifications
git push origin --delete feature/user-notifications
```

Feature Branch Best Practices

```
# Keep branches up to date
git checkout feature-branch
git rebase main

# Or merge main into feature
git merge main
```

```
# Squash commits before merging
git rebase -i HEAD~3

# Use meaningful branch names
feature/user-authentication
feature/payment-integration
bugfix/login-error
hotfix/security-vulnerability
```

Release Branching

Release Branch Strategy

```
# Create release branch
git checkout main
git checkout -b release/v2.1.0

# Prepare release
echo "2.1.0" > VERSION
git commit -am "Bump version to 2.1.0"

# Bug fixes during release preparation
git commit -am "Fix release blocker bug"

# Merge back to main and develop
git checkout main
git merge release/v2.1.0
git tag v2.1.0

git checkout develop
git merge release/v2.1.0

# Clean up
git branch -d release/v2.1.0
```

Release Branch Benefits

- **Stabilization**: Fix bugs without blocking new features
- **Parallel development**: Continue feature work on develop
- **Quality assurance**: Dedicated testing phase
- **Documentation**: Update docs and changelogs

Choosing the Right Strategy

Project Characteristics

Small Teams / Simple Projects

- **GitHub Flow**: Simple, fast iterations
- **Feature Branch Workflow**: Basic isolation

Large Teams / Complex Projects

- **Git Flow**: Structured, systematic
- **GitLab Flow**: Environment-aware

Continuous Deployment

- **GitHub Flow**: Built for CD
- **GitLab Flow**: Environment branches

Scheduled Releases

- **Git Flow**: Release branches
- **GitLab Flow**: Release branches

Decision Matrix

Factor	Git Flow	GitHub Flow	GitLab Flow	Feature Branch
Complexity	High	Low	Medium	Low
Release Schedule	Scheduled	Continuous	Flexible	Flexible
Team Size	Large	Any	Any	Small-Medium
Deployment	Traditional	Continuous	Environment-based	Manual
Learning Curve	Steep	Gentle	Medium	Gentle

Implementing Branching Strategies

Team Guidelines

```
# Branching Strategy Guidelines

## Branch Naming
- feature/description-of-feature
- bugfix/description-of-bug
```

```
- hotfix/critical-issue-description
- release/version-number

## Workflow Rules
1. Always branch from main/develop
2. Keep branches focused and small
3. Write descriptive commit messages
4. Test before merging
5. Delete branches after merging

## Code Review
- All changes require pull request
- At least one approval required
- Automated tests must pass
- Documentation updated if needed
```

Automation and Tools

```
# Branch protection rules (GitHub)
# - Require pull request reviews
# - Require status checks
# - Restrict pushes to main

# Git hooks for enforcement
# pre-commit: Run tests
# pre-push: Prevent direct pushes to main
# commit-msg: Enforce commit message format

# CI/CD integration
# - Automated testing on feature branches
# - Deployment pipelines
# - Quality gates
```

Advanced Techniques

Branch Policies

```
# Prevent direct pushes to main
git config branch.main.pushRemote no_push

# Require merge commits
git config branch.main.mergeoptions "--no-ff"
```

```
# Set up branch protection
# (Usually done through Git hosting platform)
```

Automated Branch Management

```
# Delete merged branches automatically
git branch --merged | grep -v "\*\|main\|develop" | xargs -n 1 git branch -d

# Prune remote tracking branches
git remote prune origin

# Clean up old branches (script)
#!/bin/bash
git for-each-ref --format='%(refname:short) %(committerdate)' refs/heads | \
awk '$2 <= "'$(date -d '30 days ago' '+%Y-%m-%d')'"' | \
cut -d' ' -f1 | \
xargs -r git branch -D
```

Exercises

Exercise 1: Git Flow Implementation

1. Set up a repository with Git Flow
2. Create a feature branch and implement a feature
3. Create a release branch and prepare a release
4. Simulate a hotfix scenario

Exercise 2: GitHub Flow Practice

1. Set up a repository for GitHub Flow
2. Create feature branches for different features
3. Practice pull request workflow
4. Implement branch protection rules

Exercise 3: Strategy Comparison

1. Implement the same feature using different strategies
2. Compare the complexity and overhead
3. Analyze which strategy fits different scenarios
4. Document pros and cons for your team

Summary

Branching strategies provide structure and consistency for team development:

- **Git Flow**: Comprehensive strategy for scheduled releases
- **GitHub Flow**: Simple strategy for continuous deployment
- **GitLab Flow**: Flexible strategy with environment awareness
- **Feature Branch Workflow**: Basic isolation strategy

Key considerations: - Team size and experience - Release schedule and deployment model - Project complexity and requirements - Tool integration and automation

The right branching strategy depends on your specific context. Start simple and evolve as your team and project grow. The next chapter will explore working with remote repositories, which is essential for implementing these collaborative branching strategies effectively.

Remotes

Chapter 8: Working with Remotes

Understanding Remote Repositories

A remote repository is a version of your project hosted on a network (internet, intranet, or local network). Remote repositories enable collaboration by providing a central location where team members can share changes.

Key Concepts

- **Remote**: A reference to a repository on another machine
- **Origin**: Default name for the primary remote repository
- **Upstream**: The original repository you forked from
- **Clone**: Local copy of a remote repository
- **Push**: Send local changes to remote repository
- **Pull**: Fetch and merge changes from remote repository
- **Fetch**: Download changes without merging

Adding and Managing Remotes

Viewing Remotes

```
# List all remotes
git remote

# List remotes with URLs
git remote -v

# Show detailed information about a remote
git remote show origin
```

Adding Remotes

```
# Add a new remote
git remote add origin https://github.com/username/repository.git

# Add upstream remote (for forks)
```

```
git remote add upstream https://github.com/original-owner/repository.git

# Add remote with custom name
git remote add backup https://github.com/username/backup-repo.git
```

Modifying Remotes

```
# Change remote URL
git remote set-url origin https://github.com/username/new-repository.git

# Rename remote
git remote rename origin old-origin

# Remove remote
git remote remove backup
```

Remote URL Formats

```
# HTTPS (requires authentication for push)
https://github.com/username/repository.git

# SSH (requires SSH key setup)
git@github.com:username/repository.git

# Local path
/path/to/local/repository.git
file:///path/to/local/repository.git
```

Cloning Repositories

Basic Cloning

```
# Clone repository
git clone https://github.com/username/repository.git

# Clone to specific directory
git clone https://github.com/username/repository.git my-project

# Clone specific branch
git clone -b develop https://github.com/username/repository.git
```

```
# Shallow clone (limited history)
git clone --depth 1 https://github.com/username/repository.git
```

Clone Options

```
# Clone without checkout (bare repository)
git clone --bare https://github.com/username/repository.git

# Clone with specific origin name
git clone -o upstream https://github.com/username/repository.git

# Clone recursively with submodules
git clone --recursive https://github.com/username/repository.git

# Clone single branch only
git clone --single-branch --branch main https://github.com/username/repository.git
```

Fetching Changes

Understanding Fetch

Fetch downloads changes from remote repository without merging them into your working directory.

```
# Fetch from default remote (origin)
git fetch

# Fetch from specific remote
git fetch upstream

# Fetch all remotes
git fetch --all

# Fetch specific branch
git fetch origin main

# Fetch with pruning (remove deleted remote branches)
git fetch --prune
```

Viewing Fetched Changes

```
# View remote branches
git branch -r

# View all branches (local and remote)
git branch -a

# Compare local branch with remote
git diff main origin/main

# View commits in remote branch
git log origin/main

# View commits not in local branch
git log main..origin/main
```

Pulling Changes

Basic Pull Operations

```
# Pull from default remote and branch
git pull

# Pull from specific remote and branch
git pull origin main

# Pull with rebase instead of merge
git pull --rebase

# Pull from upstream (for forks)
git pull upstream main
```

Pull Strategies

```
# Configure default pull strategy
git config --global pull.rebase false  # merge (default)
git config --global pull.rebase true   # rebase
git config --global pull.ff only       # fast-forward only

# One-time pull strategies
git pull --no-rebase    # Force merge
git pull --rebase       # Force rebase
```

```
git pull --ff-only      # Fast-forward only
```

Handling Pull Conflicts

```
# If pull results in conflicts during merge
git pull origin main
# CONFLICT: Merge conflict in file.txt

# Resolve conflicts and complete merge
vim file.txt  # Resolve conflicts
git add file.txt
git commit

# If pull results in conflicts during rebase
git pull --rebase origin main
# CONFLICT: Rebase conflict in file.txt

# Resolve conflicts and continue rebase
vim file.txt  # Resolve conflicts
git add file.txt
git rebase --continue
```

Pushing Changes

Basic Push Operations

```
# Push to default remote and branch
git push

# Push to specific remote and branch
git push origin main

# Push and set upstream tracking
git push -u origin feature-branch

# Push all branches
git push --all origin

# Push tags
git push --tags origin
```

Push Options

```
# Force push (dangerous - rewrites history)
git push --force origin main

# Safer force push (fails if remote has new commits)
git push --force-with-lease origin main

# Push new branch and set upstream
git push -u origin new-feature

# Push to different branch name
git push origin local-branch:remote-branch

# Delete remote branch
git push origin --delete feature-branch
```

Push Conflicts

```
# When push is rejected
git push origin main
# error: Updates were rejected because the remote contains work

# Solutions:
# 1. Pull first, then push
git pull origin main
git push origin main

# 2. Pull with rebase, then push
git pull --rebase origin main
git push origin main

# 3. Force push (only if you're sure)
git push --force-with-lease origin main
```

Tracking Branches

Understanding Tracking

Tracking branches are local branches that have a direct relationship with a remote branch.

```
# View tracking relationships
git branch -vv
```

```
# Set upstream for current branch
git branch -u origin/main

# Set upstream when pushing
git push -u origin feature-branch

# Unset upstream
git branch --unset-upstream
```

Working with Tracking Branches

```
# Create local branch tracking remote branch
git checkout -b feature-branch origin/feature-branch

# Shorter syntax (if branch doesn't exist locally)
git checkout feature-branch

# Push to upstream branch
git push

# Pull from upstream branch
git pull
```

Remote Branch Management

Viewing Remote Branches

```
# List remote branches
git branch -r

# List all branches with tracking info
git branch -vv

# Show remote branch details
git remote show origin
```

Creating Remote Branches

```
# Create and push new branch
git checkout -b new-feature
git push -u origin new-feature

# Push existing branch to remote
git push origin existing-branch
```

Deleting Remote Branches

```
# Delete remote branch
git push origin --delete feature-branch

# Alternative syntax
git push origin :feature-branch

# Clean up local tracking references
git remote prune origin
```

Working with Forks

Fork Workflow

```
# 1. Fork repository on GitHub
# 2. Clone your fork
git clone https://github.com/yourusername/repository.git

# 3. Add upstream remote
git remote add upstream https://github.com/original-owner/repository.git

# 4. Create feature branch
git checkout -b feature-branch

# 5. Make changes and commit
git add .
git commit -m "Add new feature"

# 6. Push to your fork
git push origin feature-branch

# 7. Create pull request on GitHub
```

Keeping Fork Updated

```
# Fetch upstream changes
git fetch upstream

# Merge upstream changes into main
git checkout main
git merge upstream/main

# Push updated main to your fork
git push origin main

# Update feature branch with latest main
git checkout feature-branch
git rebase main
```

Multiple Remotes Workflow

Working with Multiple Remotes

```
# Add multiple remotes
git remote add origin https://github.com/yourusername/repo.git
git remote add upstream https://github.com/original/repo.git
git remote add backup https://github.com/yourusername/backup.git

# Push to multiple remotes
git push origin main
git push backup main

# Pull from different remotes
git pull upstream main
git pull origin main
```

Remote Management Scripts

```
#!/bin/bash
# Script to push to multiple remotes
for remote in origin backup; do
    echo "Pushing to $remote..."
    git push $remote main
done
```

Authentication

HTTPS Authentication

```
# Configure credentials (one-time setup)
git config --global credential.helper store

# Or use credential manager
git config --global credential.helper manager

# Personal Access Token (GitHub)
# Use token instead of password when prompted
```

SSH Authentication

```
# Generate SSH key
ssh-keygen -t ed25519 -C "your.email@example.com"

# Add key to SSH agent
ssh-add ~/.ssh/id_ed25519

# Add public key to GitHub/GitLab
cat ~/.ssh/id_ed25519.pub

# Test SSH connection
ssh -T git@github.com
```

Switching Between HTTPS and SSH

```
# Change from HTTPS to SSH
git remote set-url origin git@github.com:username/repository.git

# Change from SSH to HTTPS
git remote set-url origin https://github.com/username/repository.git
```

Troubleshooting Remote Issues

Common Problems

Problem: Permission Denied

```
# Check remote URL
git remote -v

# Verify authentication
ssh -T git@github.com  # For SSH
git ls-remote origin   # For HTTPS

# Fix: Update credentials or SSH keys
```

Problem: Remote Branch Doesn't Exist

```
# Fetch latest remote information
git fetch origin

# Check if branch exists remotely
git branch -r | grep feature-branch

# Create remote branch if needed
git push -u origin feature-branch
```

Problem: Diverged Branches

```
# Check status
git status

# View divergence
git log --oneline --graph origin/main main

# Solutions:
git pull --rebase origin main  # Rebase approach
git pull origin main           # Merge approach
```

Best Practices

Remote Management

1. **Use descriptive remote names**

```
git remote add upstream https://github.com/original/repo.git
git remote add fork https://github.com/yourusername/repo.git
```

2. **Keep remotes organized**

```
# Regular cleanup
git remote prune origin
git fetch --prune
```

3. **Use SSH for frequent access**

```
# More secure and convenient
git remote set-url origin git@github.com:username/repo.git
```

Collaboration Workflow

1. **Always pull before pushing**

```
git pull origin main
git push origin main
```

2. **Use feature branches for collaboration**

```
git checkout -b feature/new-functionality
git push -u origin feature/new-functionality
```

3. **Keep main branch clean**

```
# Don't push directly to main
# Use pull requests/merge requests
```

Exercises

Exercise 1: Basic Remote Operations

1. Create a repository on GitHub
2. Clone it locally
3. Make changes and push them
4. Verify changes appear on GitHub

Exercise 2: Multiple Remotes

1. Fork a repository on GitHub
2. Clone your fork locally
3. Add upstream remote
4. Fetch changes from upstream
5. Push changes to your fork

Exercise 3: Branch Synchronization

1. Create a feature branch locally
2. Push it to remote
3. Make changes on both local and remote
4. Practice resolving conflicts

Exercise 4: Authentication Setup

1. Set up SSH keys for GitHub
2. Switch repository from HTTPS to SSH
3. Test authentication
4. Configure credential helper for HTTPS

Summary

Working with remotes is essential for collaboration:

- **Remotes** provide centralized repositories for sharing code
- **Cloning** creates local copies of remote repositories
- **Fetching** downloads changes without merging
- **Pulling** fetches and merges changes
- **Pushing** uploads local changes to remote
- **Tracking branches** maintain relationships between local and remote branches

Key skills developed: - Managing multiple remotes - Handling authentication - Resolving push/pull conflicts - Working with forks - Maintaining synchronized repositories

Understanding remote operations is crucial for effective team collaboration. The next chapter will introduce GitHub, which builds upon these remote repository concepts to provide a complete collaboration platform.

GitHub Intro

Chapter 9: Introduction to GitHub

What is GitHub?

GitHub is a web-based platform that provides Git repository hosting along with additional collaboration features. It's built on top of Git and adds a user-friendly interface, project management tools, and social coding features.

GitHub vs Git

Git	GitHub
Version control system	Hosting platform for Git repositories
Command-line tool	Web-based interface
Local and distributed	Cloud-based with collaboration features
Open source	Proprietary platform (owned by Microsoft)
Works offline	Requires internet connection

Key GitHub Features

- **Repository hosting**: Store Git repositories in the cloud
- **Collaboration tools**: Pull requests, issues, project boards
- **Social features**: Following, starring, forking
- **CI/CD**: GitHub Actions for automation
- **Documentation**: Wiki, GitHub Pages
- **Security**: Vulnerability scanning, secret management
- **Package management**: GitHub Packages

Creating Your GitHub Account

Account Setup

1. **Visit GitHub**: Go to github.com
2. **Sign up**: Click "Sign up" and provide:
    - Username (choose carefully - hard to change)
    - Email address
    - Password

3. **Verify email**: Check your email and verify account
4. **Choose plan**: Free plan includes unlimited public/private repos

Profile Setup

```
# Complete your profile
- Profile picture
- Bio description
- Location
- Website/blog URL
- Company/organization
- Social media links
```

Account Security

```
# Enable two-factor authentication (2FA)
# 1. Go to Settings > Security
# 2. Enable 2FA with authenticator app or SMS
# 3. Download recovery codes

# Add SSH keys for secure authentication
ssh-keygen -t ed25519 -C "your.email@example.com"
# Add public key to GitHub Settings > SSH and GPG keys
```

Creating Repositories

Repository Creation Options

Option 1: Create on GitHub First

```
# 1. Create repository on GitHub web interface
# 2. Clone to local machine
git clone https://github.com/username/repository-name.git
cd repository-name

# 3. Start working
echo "# My Project" > README.md
git add README.md
git commit -m "Initial commit"
git push origin main
```

Option 2: Create Locally First

```
# 1. Create local repository
mkdir my-project
cd my-project
git init
echo "# My Project" > README.md
git add README.md
git commit -m "Initial commit"

# 2. Create repository on GitHub (web interface)
# 3. Add remote and push
git remote add origin https://github.com/username/my-project.git
git branch -M main
git push -u origin main
```

Repository Settings

Basic Settings

- **Repository name**: Descriptive and clear
- **Description**: Brief explanation of the project
- **Visibility**: Public (anyone can see) or Private (restricted access)
- **Initialize with**: README, .gitignore, license

Advanced Settings

- **Features**: Issues, Wiki, Projects, Discussions
- **Pull Requests**: Allow merge commits, squash merging, rebase merging
- **Branches**: Default branch, branch protection rules
- **Webhooks**: Integration with external services

GitHub Interface Overview

Repository Structure

```
Repository Page
  Code tab (default)
     File browser
     README display
     Clone/download options
     Branch selector
  Issues tab
  Pull requests tab
```

```
Actions tab (CI/CD)
Projects tab
Wiki tab
Security tab
Insights tab
Settings tab (if you have access)
```

Navigation Elements

Repository Header

- Repository name and owner
- Star, watch, and fork buttons
- Clone/download dropdown
- Branch/tag selector

File Browser

- Directory structure
- File preview
- Commit information for each file
- Last modified date and author

README Display

- Automatically renders README.md
- Supports Markdown formatting
- First impression for visitors

Working with Files on GitHub

Web Editor

```
# Edit files directly on GitHub
1. Navigate to file
2. Click pencil icon (Edit)
3. Make changes in web editor
4. Add commit message
5. Choose commit directly or create pull request
```

Creating Files

```
# Create new file
1. Click "Create new file" button
2. Enter filename (use / for directories)
3. Add content
4. Commit with message
```

Uploading Files

```
# Upload files via web interface
1. Click "Upload files" button
2. Drag and drop or choose files
3. Add commit message
4. Commit changes
```

File History

```
# View file history
1. Navigate to file
2. Click "History" button
3. View all commits that modified the file
4. Click commit to see changes
```

README Files

README Importance

README files are crucial for: - **First impressions**: What visitors see first - **Project explanation**: What the project does - **Usage instructions**: How to use the project - **Contribution guidelines**: How to contribute - **Documentation**: Basic project documentation

README Best Practices

```
# Project Title

Brief description of what the project does.

## Features

- Feature 1
```

```
- Feature 2
- Feature 3

## Installation

```bash
git clone https://github.com/username/project.git
cd project
npm install
```

Usage

```
npm start
```

Contributing

1. Fork the repository
2. Create feature branch
3. Make changes
4. Submit pull request

License

This project is licensed under the MIT License - see the LICENSE file for details.

Markdown Formatting

```markdown
# Headers
## Subheaders
### Sub-subheaders

**Bold text**
*Italic text*
`Code snippets`

```code blocks```

[Links](https://example.com)
![Images](image.png)

- Bullet points
```

```
1. Numbered lists

> Blockquotes

| Tables | Are | Supported |
|--------|-----|-----------|
| Cell 1 | Cell 2 | Cell 3 |
```

Licenses

Why Licenses Matter

- **Legal protection**: Define how others can use your code
- **Clarity**: Clear terms for contributors and users
- **Open source compliance**: Required for open source projects
- **Professional appearance**: Shows project maturity

Common Licenses

MIT License

```
Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction...
```

- **Permissive**: Allows almost anything
- **Popular**: Widely used and understood
- **Simple**: Easy to understand terms

Apache License 2.0

- **Patent protection**: Includes patent grant
- **Attribution required**: Must include license notice
- **Popular**: Used by many large projects

GNU GPL v3

- **Copyleft**: Derivative works must be open source
- **Strong protection**: Ensures code remains free
- **Viral**: Affects entire project if used

Adding License

```
# Add license file
# 1. Create LICENSE file in repository root
# 2. Copy license text
# 3. Update copyright year and name
# 4. Commit to repository

# GitHub license picker
# 1. Create repository with license option
# 2. Or add LICENSE file via web interface
# 3. GitHub provides license templates
```

.gitignore Files

Purpose of .gitignore

Prevent tracking of: - **Build artifacts**: Compiled files, build directories - **Dependencies**: node_modules, vendor directories - **Secrets**: API keys, passwords, certificates - **IDE files**: Editor-specific configuration - **OS files**: .DS_Store, Thumbs.db - **Logs**: Application and system logs

GitHub .gitignore Templates

```
# GitHub provides templates for different languages/frameworks
# Available when creating repository or adding .gitignore file

# Popular templates:
- Node.js
- Python
- Java
- C++
- Swift
- Go
- Ruby
- PHP
```

Custom .gitignore Example

```
# Dependencies
node_modules/
npm-debug.log*

# Build outputs
build/
dist/
*.min.js

# Environment files
.env
.env.local
.env.production

# IDE files
.vscode/
.idea/
*.swp

# OS files
.DS_Store
Thumbs.db

# Logs
logs/
*.log

# Runtime data
pids/
*.pid
*.seed
```

Repository Management

Repository Settings

General Settings

```
# Repository name and description
- Clear, descriptive names
- Helpful descriptions
- Appropriate topics/tags
```

```
# Features
- Enable/disable Issues
- Enable/disable Wiki
- Enable/disable Projects
- Enable/disable Discussions
```

Collaboration Settings

```
# Manage access
- Add collaborators
- Set permissions (read, write, admin)
- Create teams (for organizations)

# Branch protection
- Protect main branch
- Require pull request reviews
- Require status checks
- Restrict pushes
```

Repository Visibility

Public Repositories

- **Visible to everyone**: Anyone can view and clone
- **Search indexing**: Appears in GitHub search
- **Free**: No cost for public repositories
- **Open source**: Suitable for open source projects

Private Repositories

- **Restricted access**: Only you and collaborators can see
- **Not searchable**: Doesn't appear in public search
- **Paid feature**: Free tier includes limited private repos
- **Proprietary**: Suitable for private/commercial projects

Repository Templates

```
# Create template repository
# 1. Go to repository settings
# 2. Check "Template repository"
# 3. Others can use "Use this template" button
```

```
# Benefits of templates:
- Standardized project structure
- Pre-configured settings
- Boilerplate code
- Consistent setup across projects
```

GitHub Desktop

Installation and Setup

```
# Download GitHub Desktop
# 1. Visit desktop.github.com
# 2. Download for your OS
# 3. Install and sign in with GitHub account

# Features:
- Visual Git interface
- Repository management
- Commit history visualization
- Branch management
- Conflict resolution tools
```

Basic Operations

```
# Clone repository
1. File > Clone repository
2. Choose from GitHub.com or URL
3. Select local path

# Make commits
1. View changes in left panel
2. Add commit message
3. Click "Commit to main"

# Push/pull changes
1. Click "Push origin" to upload
2. Click "Fetch origin" to download
3. Automatic sync notifications
```

Exercises

Exercise 1: Account Setup

1. Create GitHub account if you don't have one
2. Complete your profile information
3. Set up two-factor authentication
4. Add SSH key for secure access

Exercise 2: Repository Creation

1. Create a new public repository on GitHub
2. Initialize with README, .gitignore, and license
3. Clone repository locally
4. Make changes and push back to GitHub

Exercise 3: README and Documentation

1. Create a comprehensive README for a project
2. Use various Markdown formatting features
3. Include code examples and images
4. Add proper license information

Exercise 4: Repository Management

1. Create both public and private repositories
2. Practice uploading files via web interface
3. Edit files directly on GitHub
4. Explore repository settings and features

Best Practices

Repository Organization

1. **Clear naming**: Use descriptive repository names
2. **Good documentation**: Comprehensive README files
3. **Proper licensing**: Choose appropriate license
4. **Organized structure**: Logical file and directory layout
5. **Regular updates**: Keep repositories current

Collaboration Preparation

1. **Enable features**: Turn on Issues, Wiki, Projects as needed
2. **Set guidelines**: Create CONTRIBUTING.md file
3. **Branch protection**: Protect main branch from direct pushes
4. **Templates**: Use issue and pull request templates

Security Considerations

1. **Never commit secrets**: Use .gitignore for sensitive files
2. **Review permissions**: Regularly audit collaborator access
3. **Enable security features**: Vulnerability alerts, dependency scanning
4. **Use SSH keys**: More secure than password authentication

Summary

GitHub provides a comprehensive platform for Git repository hosting and collaboration:

- **Repository hosting**: Secure, reliable Git repository storage
- **Web interface**: User-friendly way to interact with repositories
- **Collaboration tools**: Issues, pull requests, project management
- **Documentation**: README files, wikis, GitHub Pages
- **Security**: Access control, vulnerability scanning

Key skills developed: - Creating and managing repositories - Using GitHub's web interface effectively - Writing good documentation - Understanding licensing - Setting up secure authentication

GitHub transforms Git from a local tool into a collaborative platform. The next chapter will explore collaboration workflows that leverage GitHub's features for effective team development.

Collaboration

Chapter 10: Collaboration Workflows

Introduction to Collaborative Development

Collaborative development involves multiple developers working on the same project simultaneously. GitHub provides tools and workflows that make this collaboration efficient, organized, and conflict-free.

Collaboration Challenges

- **Concurrent changes**: Multiple people editing the same code
- **Code quality**: Maintaining standards across contributors
- **Communication**: Coordinating work and decisions
- **Integration**: Combining different features and fixes
- **Review process**: Ensuring code quality before merging

GitHub's Collaboration Solutions

- **Pull Requests**: Structured code review process
- **Issues**: Bug tracking and feature requests
- **Project Boards**: Task management and planning
- **Discussions**: Community conversations
- **Teams**: Organized access control

Forking Workflow

Understanding Forks

A fork is a personal copy of someone else's repository. It allows you to: - Experiment without affecting the original - Contribute to projects you don't have write access to - Maintain your own version of a project

Fork Workflow Steps

1. Fork the Repository

```
# On GitHub web interface:
1. Navigate to the repository you want to contribute to
2. Click the "Fork" button in the top-right corner
3. Choose where to fork (your account or organization)
4. GitHub creates a copy in your account
```

2. Clone Your Fork

```
# Clone your fork to local machine
git clone https://github.com/yourusername/repository-name.git
cd repository-name

# Add upstream remote (original repository)
git remote add upstream https://github.com/original-owner/repository-name.git

# Verify remotes
git remote -v
# origin    https://github.com/yourusername/repository-name.git (fetch)
# origin    https://github.com/yourusername/repository-name.git (push)
# upstream  https://github.com/original-owner/repository-name.git (fetch)
# upstream  https://github.com/original-owner/repository-name.git (push)
```

3. Create Feature Branch

```
# Always create a new branch for your changes
git checkout -b feature/add-user-authentication

# Make your changes
echo "Authentication code" > auth.js
git add auth.js
git commit -m "Add user authentication system"
```

4. Keep Fork Updated

```
# Fetch latest changes from upstream
git fetch upstream

# Switch to main branch
```

```
git checkout main

# Merge upstream changes
git merge upstream/main

# Push updated main to your fork
git push origin main

# Update your feature branch (optional but recommended)
git checkout feature/add-user-authentication
git rebase main
```

5. Push Changes to Your Fork

```
# Push feature branch to your fork
git push -u origin feature/add-user-authentication
```

6. Create Pull Request

```
# On GitHub web interface:
1. Navigate to your fork
2. Click "Compare & pull request" button
3. Fill out pull request form:
   - Title: Clear, descriptive summary
   - Description: Detailed explanation of changes
   - Link to related issues
4. Click "Create pull request"
```

Pull Requests

What are Pull Requests?

Pull requests (PRs) are a way to propose changes to a repository. They provide: - **Code review**: Team members can review changes before merging - **Discussion**: Comments and feedback on specific lines - **Testing**: Automated tests can run on proposed changes - **Documentation**: Record of what changed and why

Creating Effective Pull Requests

PR Title Best Practices

```
# Good titles:
- "Add user authentication system"
- "Fix memory leak in image processing"
- "Update API documentation for v2.0"

# Poor titles:
- "Fix bug"
- "Update code"
- "Changes"
```

PR Description Template

```
## Description
Brief summary of changes made.

## Type of Change
- [ ] Bug fix (non-breaking change which fixes an issue)
- [ ] New feature (non-breaking change which adds functionality)
- [ ] Breaking change (fix or feature that would cause existing functionality to not work as
- [ ] Documentation update

## How Has This Been Tested?
- [ ] Unit tests
- [ ] Integration tests
- [ ] Manual testing

## Related Issues
Fixes #123
Closes #456

## Screenshots (if applicable)
[Add screenshots here]

## Checklist
- [ ] My code follows the style guidelines of this project
- [ ] I have performed a self-review of my own code
- [ ] I have commented my code, particularly in hard-to-understand areas
- [ ] I have made corresponding changes to the documentation
- [ ] My changes generate no new warnings
- [ ] I have added tests that prove my fix is effective or that my feature works
```

Pull Request Workflow

1. Review Process

```
# Reviewers can:
- Comment on specific lines of code
- Suggest changes
- Approve the pull request
- Request changes before approval
- Dismiss reviews if needed
```

2. Addressing Feedback

```
# Make requested changes
git checkout feature/add-user-authentication
# Edit files based on feedback
git add modified-files
git commit -m "Address review feedback: improve error handling"
git push origin feature/add-user-authentication

# Changes automatically appear in the pull request
```

3. Merging Pull Requests

```
# Merge options:
1. **Merge commit**: Creates merge commit preserving branch history
2. **Squash and merge**: Combines all commits into single commit
3. **Rebase and merge**: Replays commits without merge commit

# After merging:
- Feature branch can be deleted
- Close related issues automatically
- Notify contributors
```

Code Review Process

Why Code Review Matters

- **Quality assurance**: Catch bugs before they reach production
- **Knowledge sharing**: Team learns from each other's code
- **Consistency**: Maintain coding standards
- **Mentoring**: Help junior developers improve
- **Documentation**: Record decisions and reasoning

Effective Code Review Practices

For Authors (PR Creators)

```
# Before creating PR:
- [ ] Test your changes thoroughly
- [ ] Write clear commit messages
- [ ] Update documentation if needed
- [ ] Keep changes focused and small
- [ ] Self-review your code first

# PR description should include:
- What changed and why
- How to test the changes
- Any breaking changes
- Screenshots for UI changes
```

For Reviewers

```
# Review checklist:
- [ ] Does the code solve the stated problem?
- [ ] Is the code readable and maintainable?
- [ ] Are there any obvious bugs or edge cases?
- [ ] Does it follow project conventions?
- [ ] Are tests adequate?
- [ ] Is documentation updated?

# Review etiquette:
- Be constructive and specific
- Explain the "why" behind suggestions
- Acknowledge good code
- Ask questions instead of making demands
- Focus on the code, not the person
```

Review Comments Examples

Good Review Comments

```
# Specific and constructive:
"Consider using a Map instead of an object here for better performance with large datasets."

# Asking questions:
"What happens if userId is null here? Should we add a guard clause?"
```

```
# Suggesting improvements:
"This function is getting complex. Could we extract the validation logic into a separate fun

# Acknowledging good work:
"Nice use of the builder pattern here! This makes the code much more readable."
```

Poor Review Comments

```
# Too vague:
"This is wrong."

# Not constructive:
"I don't like this approach."

# Personal attacks:
"You always write code like this."
```

Issue Tracking

Understanding GitHub Issues

Issues are used for: - **Bug reports**: Document problems in the code - **Feature requests**: Propose new functionality - **Tasks**: Track work items - **Questions**: Ask for help or clarification - **Discussions**: Broader project conversations

Creating Effective Issues

Bug Report Template

```
## Bug Description
A clear and concise description of what the bug is.

## Steps to Reproduce
1. Go to '...'
2. Click on '....'
3. Scroll down to '....'
4. See error

## Expected Behavior
A clear description of what you expected to happen.

## Actual Behavior
```

```
A clear description of what actually happened.

## Screenshots
If applicable, add screenshots to help explain your problem.

## Environment
- OS: [e.g. iOS]
- Browser: [e.g. chrome, safari]
- Version: [e.g. 22]

## Additional Context
Add any other context about the problem here.
```

Feature Request Template

```
## Feature Description
A clear and concise description of what you want to happen.

## Problem Statement
What problem does this feature solve?

## Proposed Solution
Describe the solution you'd like.

## Alternatives Considered
Describe any alternative solutions you've considered.

## Additional Context
Add any other context or screenshots about the feature request here.
```

Issue Management

Labels

```
# Common label categories:
- Type: bug, enhancement, documentation
- Priority: high, medium, low
- Status: in-progress, blocked, needs-review
- Difficulty: beginner, intermediate, advanced
- Area: frontend, backend, database, testing
```

Milestones

```
# Use milestones for:
- Release versions (v1.0, v1.1, v2.0)
- Sprint planning (Sprint 1, Sprint 2)
- Project phases (Alpha, Beta, Release)
```

Assignees

```
# Assign issues to:
- Person responsible for fixing
- Team lead for triage
- Subject matter expert
```

Project Management

GitHub Projects

GitHub Projects provide Kanban-style project management:

Setting Up Projects

```
# Create project:
1. Go to repository or organization
2. Click "Projects" tab
3. Click "New project"
4. Choose template or start from scratch
5. Configure columns (To Do, In Progress, Done)
```

Project Automation

```
# Automated workflows:
- Move issues to "In Progress" when assigned
- Move pull requests to "Review" when opened
- Move items to "Done" when closed
- Add new issues to "To Do" column
```

Project Board Workflow

Column Structure

```
# Typical columns:
1. **Backlog**: All planned work
2. **To Do**: Ready to start
3. **In Progress**: Currently being worked on
4. **Review**: Waiting for code review
5. **Testing**: Being tested
6. **Done**: Completed work
```

Card Management

```
# Cards can be:
- Issues from the repository
- Pull requests
- Notes (simple text cards)
- External items (linked)

# Card actions:
- Move between columns
- Add labels and assignees
- Convert notes to issues
- Archive completed cards
```

Team Collaboration

Repository Permissions

Permission Levels

```
# Repository access levels:
- **Read**: Can view and clone repository
- **Triage**: Can manage issues and pull requests
- **Write**: Can push to repository
- **Maintain**: Can manage repository settings
- **Admin**: Full access including deletion
```

Branch Protection Rules

```
# Protect main branch:
# 1. Go to Settings > Branches
# 2. Add rule for main branch
# 3. Configure protections:
#    - Require pull request reviews
#    - Require status checks
#    - Restrict pushes
#    - Require linear history
```

Team Workflows

Feature Development Workflow

```
# 1. Create issue for feature
# 2. Assign to developer
# 3. Developer creates feature branch
git checkout -b feature/issue-123-user-profile

# 4. Develop feature with regular commits
git add .
git commit -m "Add user profile form"
git push -u origin feature/issue-123-user-profile

# 5. Create pull request
# 6. Code review process
# 7. Merge after approval
# 8. Close issue automatically
```

Bug Fix Workflow

```
# 1. Bug reported via issue
# 2. Triage and prioritize
# 3. Assign to developer
# 4. Create hotfix branch if critical
git checkout main
git checkout -b hotfix/critical-security-fix

# 5. Fix bug and test
# 6. Create pull request
# 7. Fast-track review for critical fixes
# 8. Merge and deploy
```

Advanced Collaboration Features

GitHub Discussions

```
# Use discussions for:
- General questions
- Ideas and brainstorming
- Announcements
- Show and tell
- Community building

# Discussion categories:
- General
- Ideas
- Q&A
- Show and tell
- Announcements
```

Draft Pull Requests

```
# Create draft PR for work in progress:
# 1. Create pull request as usual
# 2. Select "Create draft pull request"
# 3. Continue working and pushing commits
# 4. Mark as "Ready for review" when complete

# Benefits:
- Early feedback on approach
- Continuous integration testing
- Visibility of work in progress
- Collaboration on complex features
```

Co-authored Commits

```
# When pair programming or collaborating:
git commit -m "Implement user authentication

Co-authored-by: Jane Doe <jane@example.com>
Co-authored-by: John Smith <john@example.com>"

# GitHub recognizes co-authors and gives credit
```

Conflict Resolution in Collaboration

Preventing Conflicts

```
# Best practices:
# 1. Keep branches up to date
git fetch upstream
git rebase upstream/main

# 2. Communicate about overlapping work
# 3. Make small, focused changes
# 4. Merge frequently
```

Resolving Conflicts

```
# When conflicts occur in PR:
# 1. Fetch latest changes
git fetch upstream

# 2. Rebase your branch
git rebase upstream/main

# 3. Resolve conflicts
# Edit conflicted files
git add resolved-files
git rebase --continue

# 4. Force push to update PR
git push --force-with-lease origin feature-branch
```

Exercises

Exercise 1: Fork and Contribute

1. Find an open source project on GitHub
2. Fork the repository
3. Create a small improvement (documentation, bug fix)
4. Submit a pull request
5. Engage with maintainer feedback

Exercise 2: Team Collaboration Setup

1. Create a repository for a team project
2. Add collaborators with appropriate permissions
3. Set up branch protection rules
4. Create issue templates
5. Set up a project board

Exercise 3: Code Review Practice

1. Create pull requests with different types of changes
2. Practice giving constructive code reviews
3. Address review feedback effectively
4. Use different merge strategies

Exercise 4: Issue Management

1. Create various types of issues (bugs, features, tasks)
2. Use labels, milestones, and assignments
3. Link issues to pull requests
4. Practice issue triage and prioritization

Best Practices Summary

For Contributors

1. **Fork and branch**: Always work in feature branches
2. **Small changes**: Keep pull requests focused and manageable
3. **Clear communication**: Write descriptive titles and descriptions
4. **Test thoroughly**: Ensure changes work as expected
5. **Respond promptly**: Address review feedback quickly

For Maintainers

1. **Clear guidelines**: Provide contribution guidelines
2. **Prompt reviews**: Review pull requests in timely manner
3. **Constructive feedback**: Help contributors improve
4. **Consistent standards**: Apply rules fairly to all contributors
5. **Recognize contributions**: Acknowledge good work

For Teams

1. **Establish workflows**: Define clear processes for everyone
2. **Use templates**: Standardize issues and pull requests
3. **Protect branches**: Prevent direct pushes to main branches
4. **Automate testing**: Use CI/CD for quality assurance
5. **Regular communication**: Keep team informed of changes

Summary

Effective collaboration on GitHub requires:

- **Structured workflows**: Fork, branch, pull request patterns
- **Quality processes**: Code review and testing
- **Clear communication**: Issues, discussions, and documentation
- **Project management**: Boards, milestones, and labels
- **Team coordination**: Permissions, protection rules, and guidelines

Key skills developed: - Creating and managing pull requests - Conducting effective code reviews - Using issues for project management - Collaborating in team environments - Resolving conflicts constructively

These collaboration workflows form the foundation of modern software development. The next chapters will explore more advanced Git techniques that support these collaborative processes.

Rewriting History

Chapter 11: Rewriting History

Introduction to History Rewriting

Git allows you to modify commit history, which can be powerful for cleaning up your work before sharing it with others. However, rewriting history should be done carefully, especially on shared branches.

When to Rewrite History

Safe scenarios (private branches): - Clean up messy commit history before merging - Fix commit messages with typos or unclear descriptions - Combine related commits into logical units - Remove sensitive information accidentally committed - Reorganize commits for better logical flow

Dangerous scenarios (shared branches): - Never rewrite history on public branches (main, develop) - Avoid rewriting commits that others have based work on - Don't rewrite history on collaborative feature branches

The Golden Rule

Never rewrite history that has been pushed and shared with others unless you have explicit agreement from all collaborators.

Interactive Rebasing

Interactive rebase is the most powerful tool for rewriting history. It allows you to modify, reorder, combine, and delete commits.

Starting Interactive Rebase

```
# Rebase last 3 commits
git rebase -i HEAD~3

# Rebase from specific commit
git rebase -i abc1234

# Rebase from branch point
git rebase -i main
```

Interactive Rebase Commands

When you start interactive rebase, Git opens an editor with commands:

```
pick f7f3f6d Add feature A
pick 310154e Fix typo in feature A
pick a5f4a0d Add feature B

# Rebase abc1234..def5678 onto abc1234 (3 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
```

Common Rebase Operations

1. Squashing Commits

Combine multiple related commits into one:

```
# Original commits:
pick f7f3f6d Add user login form
pick 310154e Fix login form validation
pick a5f4a0d Add login form styling

# Change to:
pick f7f3f6d Add user login form
squash 310154e Fix login form validation
squash a5f4a0d Add login form styling

# Result: Single commit with combined changes
```

2. Reordering Commits

Change the order of commits:

```
# Original order:
pick f7f3f6d Add feature A
pick 310154e Add feature B
pick a5f4a0d Fix feature A

# Reorder to:
pick f7f3f6d Add feature A
pick a5f4a0d Fix feature A
pick 310154e Add feature B
```

3. Editing Commits

Stop at a commit to make changes:

```
# Mark commit for editing:
edit f7f3f6d Add user authentication
pick 310154e Add password validation

# Git stops at the commit, allowing you to:
# Make changes to files
git add modified-file.js
git commit --amend

# Continue rebase
git rebase --continue
```

4. Dropping Commits

Remove commits entirely:

```
# Remove unwanted commit:
pick f7f3f6d Add feature A
drop 310154e Add debug logging  # This commit will be removed
pick a5f4a0d Add feature B
```

5. Rewording Commit Messages

Change commit messages:

```
# Change commit message:
reword f7f3f6d Add user authentication  # Will prompt for new message
pick 310154e Add password validation
```

Amending Commits

Amending the Last Commit

```
# Change the last commit message
git commit --amend -m "New commit message"

# Add files to the last commit
git add forgotten-file.txt
git commit --amend --no-edit

# Amend with new message and files
git add new-file.txt
git commit --amend -m "Updated commit message"
```

Amending Author Information

```
# Change author of last commit
git commit --amend --author="New Author <new.email@example.com>"

# Change author and committer
git commit --amend --author="New Author <new.email@example.com>" --reset-author
```

Squashing Commits

Manual Squashing with Reset

```
# Reset to 3 commits ago (keeping changes)
git reset --soft HEAD~3

# Create new commit with all changes
git commit -m "Combined commit message"
```

Squashing During Merge

```
# Squash merge (combines all commits into one)
git checkout main
git merge --squash feature-branch
git commit -m "Add complete feature X"
```

Cherry-Picking

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

Basic Cherry-Pick

```
# Apply specific commit to current branch
git cherry-pick abc1234

# Cherry-pick multiple commits
git cherry-pick abc1234 def5678 ghi9012

# Cherry-pick range of commits
git cherry-pick abc1234..def5678
```

Cherry-Pick Options

```
# Cherry-pick without committing (stage changes only)
git cherry-pick --no-commit abc1234

# Cherry-pick and edit commit message
git cherry-pick --edit abc1234

# Cherry-pick and sign off
git cherry-pick --signoff abc1234

# Continue after resolving conflicts
git cherry-pick --continue

# Abort cherry-pick
git cherry-pick --abort
```

Cherry-Pick Use Cases

Hotfix to Multiple Branches

```
# Fix applied to main branch
git checkout main
git commit -m "Fix critical security issue"

# Apply same fix to release branch
git checkout release/v1.2
```

```
git cherry-pick main

# Apply to another release branch
git checkout release/v1.1
git cherry-pick main
```

Selective Feature Porting

```
# Pick specific improvements from feature branch
git checkout main
git cherry-pick feature-branch~2  # Pick specific commit
git cherry-pick feature-branch~1  # Pick another commit
```

Handling Conflicts During Rebase

Conflict Resolution Process

```
# Start rebase
git rebase -i HEAD~3

# If conflicts occur:
# CONFLICT (content): Merge conflict in file.txt
# error: could not apply abc1234... commit message

# 1. View conflicted files
git status

# 2. Resolve conflicts manually
vim file.txt  # Edit and resolve conflicts

# 3. Stage resolved files
git add file.txt

# 4. Continue rebase
git rebase --continue

# Or abort if needed
git rebase --abort
```

Rebase Conflict Strategies

```
# Use merge strategy during rebase
git rebase -X ours main        # Prefer current branch
git rebase -X theirs main      # Prefer target branch

# Skip problematic commit
git rebase --skip

# Edit commit during conflict
git rebase --edit-todo
```

Advanced History Rewriting

Filter-Branch (Legacy)

```
# Remove file from entire history (use with caution)
git filter-branch --force --index-filter \
  'git rm --cached --ignore-unmatch secrets.txt' \
  --prune-empty --tag-name-filter cat -- --all
```

Git Filter-Repo (Modern Alternative)

```
# Install git-filter-repo
pip install git-filter-repo

# Remove file from history
git filter-repo --path secrets.txt --invert-paths

# Remove directory from history
git filter-repo --path sensitive-dir/ --invert-paths

# Change author information
git filter-repo --mailmap mailmap.txt
```

BFG Repo-Cleaner

```
# Install BFG
# Download from: https://rtyley.github.io/bfg-repo-cleaner/

# Remove large files
```

```
java -jar bfg.jar --strip-blobs-bigger-than 100M my-repo.git

# Remove sensitive data
java -jar bfg.jar --delete-files passwords.txt my-repo.git

# Clean up after BFG
cd my-repo.git
git reflog expire --expire=now --all && git gc --prune=now --aggressive
```

Rewriting Shared History

When Rewriting Shared History is Necessary

Sometimes you must rewrite shared history: - Removing sensitive information (passwords, keys) - Legal requirements (removing copyrighted content) - Fixing critical issues in commit history

Safe Shared History Rewriting Process

```
# 1. Communicate with all team members
# 2. Ensure everyone has pushed their work
# 3. Coordinate timing for the rewrite

# 4. Perform the rewrite
git filter-repo --path secrets.txt --invert-paths

# 5. Force push to all branches
git push --force-with-lease --all origin
git push --force-with-lease --tags origin

# 6. Team members must re-clone or reset their repositories
# Each team member:
git fetch origin
git reset --hard origin/main
```

Communicating History Changes

```
# Team notification template:
Subject: URGENT: Repository history rewrite required

Team,

We need to rewrite the repository history to remove sensitive information
```

```
that was accidentally committed.

REQUIRED ACTIONS:
1. Push all your current work before [DATE/TIME]
2. After the rewrite, you must:
   - Delete your local repository
   - Clone fresh from origin
   - OR reset your branches: git reset --hard origin/main

Timeline:
- [DATE/TIME]: Deadline to push work
- [DATE/TIME]: History rewrite performed
- [DATE/TIME]: Repository available with new history

Please confirm receipt of this message.
```

Best Practices for History Rewriting

Before Rewriting

1. **Backup your work**: Create backup branches

   ```
   git branch backup-before-rebase
   ```

2. **Ensure clean working directory**:

   ```
   git status  # Should show "working tree clean"
   ```

3. **Communicate with team**: Inform others about planned changes

During Rewriting

1. **Work on feature branches**: Never rewrite main/develop directly
2. **Test after rewriting**: Ensure code still works
3. **Review changes carefully**: Double-check what you're changing

After Rewriting

1. **Test thoroughly**: Run all tests
2. **Review history**: Ensure changes are correct

   ```
   git log --oneline --graph
   ```

3. **Force push carefully**: Use `--force-with-lease`

```
git push --force-with-lease origin feature-branch
```

Recovery from Mistakes

Using Reflog

```
# View reflog to find lost commits
git reflog

# Recover lost commit
git checkout abc1234  # From reflog
git branch recovery-branch  # Create branch to save it

# Reset to previous state
git reset --hard HEAD@{2}  # From reflog entry
```

Recovering from Bad Rebase

```
# Find the commit before rebase started
git reflog
# Find entry like: abc1234 HEAD@{5}: rebase -i (start)

# Reset to before rebase
git reset --hard HEAD@{5}

# Or create recovery branch
git branch recovery-branch HEAD@{5}
```

Exercises

Exercise 1: Interactive Rebase Practice

1. Create a repository with messy commit history
2. Use interactive rebase to clean it up:
    - Squash related commits
    - Reword unclear messages
    - Reorder commits logically
3. Compare before and after history

Exercise 2: Cherry-Pick Scenarios

1. Create multiple branches with different features
2. Practice cherry-picking specific commits between branches
3. Handle conflicts during cherry-picking
4. Use cherry-pick for hotfix scenarios

Exercise 3: Amending Commits

1. Practice amending commit messages
2. Add forgotten files to commits
3. Change author information
4. Understand when amending is appropriate

Exercise 4: Recovery Practice

1. Intentionally mess up history with bad rebase
2. Use reflog to understand what happened
3. Recover lost commits
4. Reset to previous state

Common Pitfalls and Solutions

Pitfall 1: Rewriting Public History

```
# Problem: Rewrote history on shared branch
# Solution: Communicate with team and coordinate reset

# For team members:
git fetch origin
git reset --hard origin/main
```

Pitfall 2: Lost Commits During Rebase

```
# Problem: Commits disappeared during rebase
# Solution: Use reflog to recover

git reflog
git branch recovery HEAD@{n}  # Where n is the reflog entry
```

Pitfall 3: Conflicts During Interactive Rebase

```
# Problem: Complex conflicts during rebase
# Solution: Abort and try different approach

git rebase --abort
# Try smaller chunks or different strategy
```

Summary

History rewriting is a powerful Git feature that enables:

- **Clean commit history**: Organize commits logically
- **Professional presentation**: Clean up before sharing
- **Error correction**: Fix mistakes in commit messages or content
- **Selective changes**: Apply specific commits across branches

Key tools covered: - Interactive rebase for comprehensive history editing - Commit amending for quick fixes - Cherry-picking for selective commit application - Recovery techniques for when things go wrong

Remember the golden rule: **Never rewrite shared history without team coordination**. Use these tools responsibly to maintain clean, professional commit histories while preserving collaboration integrity.

The next chapter will explore advanced Git commands that complement these history rewriting techniques for comprehensive repository management.

Advanced Commands

Chapter 12: Advanced Git Commands

Git Stash

Git stash temporarily saves your uncommitted changes, allowing you to switch branches or pull updates without committing incomplete work.

Basic Stash Operations

```
# Stash current changes
git stash

# Stash with a message
git stash save "Work in progress on feature X"

# List all stashes
git stash list

# Apply most recent stash
git stash apply

# Apply and remove stash
git stash pop

# Apply specific stash
git stash apply stash@{2}

# Drop specific stash
git stash drop stash@{1}

# Clear all stashes
git stash clear
```

Advanced Stash Options

```
# Stash including untracked files
git stash -u
git stash --include-untracked

# Stash only staged changes
git stash --staged

# Stash with patch mode (interactive)
git stash -p

# Create branch from stash
git stash branch new-feature-branch stash@{1}

# Show stash contents
git stash show
git stash show -p stash@{0}
```

Stash Use Cases

Quick Branch Switch

```
# Working on feature, need to fix urgent bug
git stash save "Feature X in progress"
git checkout main
git checkout -b hotfix/urgent-bug
# Fix bug and commit
git checkout feature-branch
git stash pop
```

Pulling Updates

```
# Have uncommitted changes, need to pull
git stash
git pull origin main
git stash pop
# Resolve any conflicts if they occur
```

Experimenting with Changes

```
# Try different approach
git stash save "Current approach"
# Make experimental changes
# If experiment fails:
git reset --hard HEAD
git stash pop
```

Git Bisect

Git bisect helps you find the commit that introduced a bug using binary search.

Basic Bisect Workflow

```
# Start bisect session
git bisect start

# Mark current commit as bad
git bisect bad

# Mark known good commit
git bisect good v1.0

# Git checks out middle commit
# Test the commit, then mark it:
git bisect good    # if test passes
git bisect bad     # if test fails

# Continue until Git finds the problematic commit
# Git will show: "X is the first bad commit"

# End bisect session
git bisect reset
```

Automated Bisect

```
# Automated bisect with test script
git bisect start HEAD v1.0
git bisect run ./test-script.sh

# Test script should exit with:
```

```
# 0 for good commit
# 1-127 for bad commit (except 125)
# 125 to skip commit
```

Example Test Script

```
#!/bin/bash
# test-script.sh

# Build the project
make clean && make

# Run tests
if ./run-tests.sh; then
    exit 0  # Good commit
else
    exit 1  # Bad commit
fi
```

Bisect with Specific Path

```
# Bisect only commits that changed specific files
git bisect start -- src/main.c include/header.h
git bisect bad
git bisect good v1.0
```

Git Blame and Annotate

Git blame shows who last modified each line of a file, helping track down the origin of code changes.

Basic Blame Usage

```
# Show blame for entire file
git blame filename.txt

# Show blame for specific lines
git blame -L 10,20 filename.txt

# Show blame for function (if language supports it)
git blame -L :function_name filename.c
```

Blame Options

```
# Show email addresses instead of names
git blame -e filename.txt

# Show commit hash in short format
git blame -s filename.txt

# Ignore whitespace changes
git blame -w filename.txt

# Show line numbers
git blame -n filename.txt

# Show original line numbers
git blame -f filename.txt
```

Advanced Blame Features

```
# Follow file renames
git blame -C filename.txt

# Detect moved/copied lines within file
git blame -M filename.txt

# Detect moved/copied lines from other files
git blame -C -C filename.txt

# Show blame with commit dates
git blame --date=short filename.txt
```

Blame with Time Range

```
# Blame since specific date
git blame --since="2023-01-01" filename.txt

# Blame until specific date
git blame --until="2023-12-31" filename.txt

# Blame for specific revision
git blame v1.0 -- filename.txt
```

Git Grep

Git grep searches for patterns in tracked files, offering more features than regular grep for Git repositories.

Basic Grep Usage

```
# Search for pattern in all tracked files
git grep "function_name"

# Search in specific files
git grep "pattern" -- "*.js"

# Search in specific directory
git grep "pattern" src/

# Case-insensitive search
git grep -i "pattern"
```

Advanced Grep Options

```
# Show line numbers
git grep -n "pattern"

# Show only filenames
git grep -l "pattern"

# Show count of matches per file
git grep -c "pattern"

# Show context lines
git grep -A 3 -B 3 "pattern"  # 3 lines after and before

# Word boundary search
git grep -w "word"

# Regular expressions
git grep -E "pattern1|pattern2"
git grep -P "perl_regex_pattern"
```

Grep in Specific Commits

```
# Search in specific commit
git grep "pattern" HEAD~5

# Search in all commits
git rev-list --all | xargs git grep "pattern"

# Search in commit range
git grep "pattern" v1.0..v2.0
```

Complex Grep Queries

```
# Search for pattern and exclude another
git grep "function" --and --not -e "test"

# Search for multiple patterns in same file
git grep -e "pattern1" --and -e "pattern2"

# Search for pattern in specific file types
git grep "TODO" -- "*.js" "*.ts"

# Search with custom pathspec
git grep "pattern" -- ':!test/' ':!docs/'
```

Git Log Advanced Features

Custom Log Formatting

```
# Custom format
git log --pretty=format:"%h - %an, %ar : %s"

# Available format options:
# %H  - commit hash
# %h  - abbreviated commit hash
# %T  - tree hash
# %t  - abbreviated tree hash
# %P  - parent hashes
# %p  - abbreviated parent hashes
# %an - author name
# %ae - author email
# %ad - author date
# %ar - author date, relative
```

```
# %cn - committer name
# %ce - committer email
# %cd - committer date
# %cr - committer date, relative
# %s  - subject
# %b  - body
```

Log Filtering

```
# Filter by author
git log --author="John Doe"
git log --author="john@example.com"

# Filter by committer
git log --committer="Jane Smith"

# Filter by date
git log --since="2023-01-01"
git log --until="2023-12-31"
git log --since="2 weeks ago"

# Filter by message
git log --grep="bug fix"
git log --grep="feature" --grep="enhancement" --all-match

# Filter by file changes
git log -- filename.txt
git log --follow -- filename.txt  # Follow renames
```

Log Statistics and Analysis

```
# Show file change statistics
git log --stat

# Show patch (actual changes)
git log -p

# Show short stat
git log --shortstat

# Show name status (A/M/D)
git log --name-status
```

```
# Show only names of changed files
git log --name-only

# Show merge commits only
git log --merges

# Show non-merge commits only
git log --no-merges
```

Advanced Log Queries

```
# Show commits that changed specific lines
git log -L 10,20:filename.txt

# Show commits that added or removed specific string
git log -S "function_name"

# Show commits that changed regex pattern
git log -G "regex_pattern"

# Show commits in date order (not topological)
git log --date-order

# Show first parent only (useful for merge commits)
git log --first-parent

# Show commits reachable from one ref but not another
git log main..feature-branch
git log feature-branch..main
```

Git Reflog

Reflog records updates to branch tips and other references, providing a safety net for recovery.

Basic Reflog Usage

```
# Show reflog for current branch
git reflog

# Show reflog for specific branch
git reflog feature-branch
```

```
# Show reflog for all references
git reflog --all

# Show reflog with dates
git reflog --date=iso
```

Reflog Recovery Scenarios

Recover Deleted Branch

```
# Find the branch in reflog
git reflog

# Look for entry like: abc1234 HEAD@{5}: checkout: moving from deleted-branch to main
# Recreate branch
git branch recovered-branch abc1234
```

Recover from Hard Reset

```
# Find commit before reset
git reflog

# Look for entry before reset
# Reset to that commit
git reset --hard HEAD@{3}
```

Recover Lost Commits

```
# Find lost commits in reflog
git reflog

# Create branch to save lost commits
git branch recovery-branch HEAD@{2}
```

Reflog Maintenance

```
# Expire reflog entries older than 30 days
git reflog expire --expire=30.days

# Expire unreachable entries immediately
git reflog expire --expire-unreachable=now --all
```

```
# Garbage collect after expiring
git gc --prune=now
```

Git Worktree

Git worktree allows you to have multiple working directories for the same repository.

Basic Worktree Operations

```
# List existing worktrees
git worktree list

# Add new worktree
git worktree add ../feature-work feature-branch

# Add worktree with new branch
git worktree add -b new-feature ../new-feature-work

# Remove worktree
git worktree remove ../feature-work

# Prune stale worktree references
git worktree prune
```

Worktree Use Cases

Parallel Development

```
# Main development in current directory
# Feature development in separate directory
git worktree add ../feature-auth feature/authentication

# Work on feature in separate directory
cd ../feature-auth
# Make changes and commits

# Switch back to main work
cd ../main-repo
```

Testing Different Branches

```
# Create worktree for testing
git worktree add ../test-branch test-branch

# Run tests in separate environment
cd ../test-branch
npm test

# Clean up when done
cd ../main-repo
git worktree remove ../test-branch
```

Git Submodules

Submodules allow you to include other Git repositories as subdirectories.

Basic Submodule Operations

```
# Add submodule
git submodule add https://github.com/user/library.git libs/library

# Initialize submodules after cloning
git submodule init
git submodule update

# Or combine both commands
git submodule update --init

# Update submodules to latest commits
git submodule update --remote

# Update specific submodule
git submodule update --remote libs/library
```

Working with Submodules

```
# Clone repository with submodules
git clone --recursive https://github.com/user/project.git

# Update submodule to specific commit
cd libs/library
```

```
git checkout v2.0
cd ../..
git add libs/library
git commit -m "Update library to v2.0"

# Push submodule changes
git push --recurse-submodules=on-demand
```

Submodule Configuration

```
# Configure submodule to track specific branch
git config -f .gitmodules submodule.libs/library.branch main

# Update submodule configuration
git submodule sync

# Remove submodule
git submodule deinit libs/library
git rm libs/library
rm -rf .git/modules/libs/library
```

Git Subtree

Git subtree provides an alternative to submodules for including external repositories.

Basic Subtree Operations

```
# Add subtree
git subtree add --prefix=libs/library https://github.com/user/library.git main --squash

# Pull updates from subtree
git subtree pull --prefix=libs/library https://github.com/user/library.git main --squash

# Push changes back to subtree
git subtree push --prefix=libs/library https://github.com/user/library.git main

# Split subtree into separate branch
git subtree split --prefix=libs/library -b library-branch
```

Subtree vs Submodule

Feature	Submodule	Subtree
Complexity	More complex	Simpler
History	Separate	Merged
Cloning	Requires –recursive	Normal clone
Updates	Manual	Integrated
Size	Smaller	Larger

Exercises

Exercise 1: Stash Mastery

1. Create a repository with ongoing work
2. Practice different stash operations
3. Use stash to switch between tasks
4. Create branches from stashes

Exercise 2: Bug Hunting with Bisect

1. Create a repository with a known bug introduction
2. Use git bisect to find the problematic commit
3. Create an automated test script for bisect
4. Practice bisect with different scenarios

Exercise 3: Code Investigation

1. Use git blame to understand code history
2. Use git grep to find patterns across codebase
3. Combine blame and grep for comprehensive analysis
4. Create aliases for common investigation tasks

Exercise 4: Advanced Repository Management

1. Set up worktrees for parallel development
2. Experiment with submodules and subtrees
3. Use reflog for recovery scenarios
4. Practice advanced log filtering and formatting

Best Practices

Stash Management

1. **Use descriptive messages** for stashes
2. **Don't let stashes accumulate** - apply or drop them regularly
3. **Consider branches** for longer-term work instead of stashes
4. **Test after applying** stashes to ensure no conflicts

Debugging Workflow

1. **Use bisect systematically** for complex bugs
2. **Combine blame with log** for complete context
3. **Create reproducible test cases** for bisect automation
4. **Document findings** for future reference

Repository Organization

1. **Use worktrees** for parallel development needs
2. **Choose submodules vs subtrees** based on project requirements
3. **Regular maintenance** of reflog and garbage collection
4. **Monitor repository size** and performance

Summary

Advanced Git commands provide powerful tools for:

- **Temporary storage**: Stash for work-in-progress management
- **Bug hunting**: Bisect for systematic problem identification
- **Code investigation**: Blame and grep for understanding code history
- **Advanced logging**: Custom formatting and filtering for analysis
- **Recovery**: Reflog for safety net and mistake recovery
- **Parallel work**: Worktrees for multiple working directories
- **Code reuse**: Submodules and subtrees for external dependencies

These commands form the advanced toolkit for Git power users, enabling sophisticated workflows and efficient problem-solving. Mastering them significantly improves productivity and confidence when working with complex Git repositories.

The next chapter will explore Git hooks and automation, building upon these advanced commands to create automated workflows and quality assurance processes.

Hooks Automation

Chapter 13: Git Hooks and Automation

Introduction to Git Hooks

Git hooks are scripts that run automatically at specific points in the Git workflow. They allow you to automate tasks, enforce policies, and integrate with external systems.

Types of Git Hooks

Client-Side Hooks

- **pre-commit**: Runs before commit is created
- **prepare-commit-msg**: Runs before commit message editor opens
- **commit-msg**: Runs after commit message is entered
- **post-commit**: Runs after commit is completed
- **pre-rebase**: Runs before rebase operation
- **post-checkout**: Runs after checkout operation
- **post-merge**: Runs after merge operation
- **pre-push**: Runs before push operation

Server-Side Hooks

- **pre-receive**: Runs before any references are updated
- **update**: Runs for each reference being updated
- **post-receive**: Runs after all references are updated
- **post-update**: Runs after post-receive (for compatibility)

Hook Location and Setup

```
# Hooks are stored in .git/hooks/
ls .git/hooks/

# Sample hooks are provided (with .sample extension)
# To activate, remove .sample extension and make executable
mv .git/hooks/pre-commit.sample .git/hooks/pre-commit
chmod +x .git/hooks/pre-commit
```

Client-Side Hooks

Pre-Commit Hook

The pre-commit hook runs before a commit is created, allowing you to validate changes.

Basic Pre-Commit Hook

```
#!/bin/sh
# .git/hooks/pre-commit

# Check for debugging statements
if git diff --cached --name-only | xargs grep -l "console.log\|debugger\|pdb.set_trace"; the
    echo "Error: Debugging statements found in staged files"
    echo "Please remove debugging code before committing"
    exit 1
fi

# Check for TODO comments in staged files
if git diff --cached | grep -q "TODO\|FIXME\|XXX"; then
    echo "Warning: TODO/FIXME comments found in changes"
    echo "Consider addressing these before committing"
    # Don't exit 1 here - just warn
fi

echo "Pre-commit checks passed"
exit 0
```

Advanced Pre-Commit Hook

```
#!/bin/sh
# .git/hooks/pre-commit

# Colors for output
RED='\033[0;31m'
GREEN='\033[0;32m'
YELLOW='\033[1;33m'
NC='\033[0m' # No Color

echo "${GREEN}Running pre-commit checks...${NC}"

# Check if this is an initial commit
if git rev-parse --verify HEAD >/dev/null 2>&1; then
    against=HEAD
else
```

```
    # Initial commit: diff against an empty tree object
    against=$(git hash-object -t tree /dev/null)
fi

# Check for whitespace errors
if ! git diff-index --check --cached $against --; then
    echo "${RED}Error: Whitespace errors found${NC}"
    exit 1
fi

# Check for large files
large_files=$(git diff --cached --name-only | xargs ls -la 2>/dev/null | awk '$5 > 1048576 {
if [ -n "$large_files" ]; then
    echo "${YELLOW}Warning: Large files detected:${NC}"
    echo "$large_files"
    echo "${YELLOW}Consider using Git LFS for large files${NC}"
fi

# Run linting for JavaScript files
js_files=$(git diff --cached --name-only --diff-filter=ACM | grep '\.js$')
if [ -n "$js_files" ]; then
    echo "Linting JavaScript files..."
    if ! npx eslint $js_files; then
        echo "${RED}ESLint failed. Please fix the issues above.${NC}"
        exit 1
    fi
fi

# Run tests
echo "Running tests..."
if ! npm test; then
    echo "${RED}Tests failed. Please fix failing tests before committing.${NC}"
    exit 1
fi

echo "${GREEN}All pre-commit checks passed!${NC}"
exit 0
```

Commit-Msg Hook

The commit-msg hook validates commit messages.

```
#!/bin/sh
# .git/hooks/commit-msg

commit_regex='^(feat|fix|docs|style|refactor|test|chore)(\(.+\))?: .{1,50}'
```

```
if ! grep -qE "$commit_regex" "$1"; then
    echo "Invalid commit message format!"
    echo "Format: type(scope): description"
    echo "Types: feat, fix, docs, style, refactor, test, chore"
    echo "Example: feat(auth): add user login functionality"
    exit 1
fi

# Check commit message length
if [ $(head -n1 "$1" | wc -c) -gt 72 ]; then
    echo "Commit message first line too long (max 72 characters)"
    exit 1
fi

exit 0
```

Pre-Push Hook

The pre-push hook runs before pushing to a remote repository.

```
#!/bin/sh
# .git/hooks/pre-push

protected_branch='main'
current_branch=$(git symbolic-ref HEAD | sed -e 's,.*/\(.*\),\1,')

# Prevent direct push to protected branch
if [ "$current_branch" = "$protected_branch" ]; then
    echo "Direct push to $protected_branch branch is not allowed"
    echo "Please create a pull request instead"
    exit 1
fi

# Run full test suite before push
echo "Running full test suite before push..."
if ! npm run test:full; then
    echo "Full test suite failed. Push aborted."
    exit 1
fi

# Check if branch is up to date with remote
remote_ref=$(git rev-parse origin/$current_branch 2>/dev/null)
local_ref=$(git rev-parse $current_branch)

if [ "$remote_ref" != "$local_ref" ] && [ -n "$remote_ref" ]; then
```

```
    echo "Your branch is not up to date with origin/$current_branch"
    echo "Please pull the latest changes first"
    exit 1
fi

exit 0
```

Post-Commit Hook

The post-commit hook runs after a commit is completed.

```
#!/bin/sh
# .git/hooks/post-commit

# Send notification
commit_hash=$(git rev-parse HEAD)
commit_message=$(git log -1 --pretty=%B)
author=$(git log -1 --pretty=%an)

# Log commit to file
echo "$(date): $author committed $commit_hash" >> .git/commit.log

# Send Slack notification (if webhook configured)
if [ -n "$SLACK_WEBHOOK_URL" ]; then
    curl -X POST -H 'Content-type: application/json' \
        --data "{\"text\":\"New commit by $author: $commit_message\"}" \
        "$SLACK_WEBHOOK_URL"
fi

# Update documentation if needed
if git diff HEAD~1 --name-only | grep -q "README\|docs/"; then
    echo "Documentation updated, consider regenerating docs"
fi
```

Server-Side Hooks

Pre-Receive Hook

The pre-receive hook runs on the server before any references are updated.

```
#!/bin/sh
# hooks/pre-receive

# Read all references being updated
```

```
while read oldrev newrev refname; do
    # Extract branch name
    branch=$(echo $refname | cut -d/ -f3)

    # Protect main branch
    if [ "$branch" = "main" ]; then
        # Only allow fast-forward merges to main
        if [ "$oldrev" != "0000000000000000000000000000000000000000" ]; then
            # Check if this is a fast-forward
            if ! git merge-base --is-ancestor $oldrev $newrev; then
                echo "Error: Non-fast-forward updates to main branch are not allowed"
                echo "Please rebase your changes or use a pull request"
                exit 1
            fi
        fi
    fi

    # Check commit messages in the push
    for commit in $(git rev-list $oldrev..$newrev); do
        message=$(git log -1 --pretty=%s $commit)
        if ! echo "$message" | grep -qE '^(feat|fix|docs|style|refactor|test|chore)(\(.+\))?
            echo "Error: Invalid commit message format in $commit"
            echo "Message: $message"
            exit 1
        fi
    done
done

exit 0
```

Post-Receive Hook

The post-receive hook runs after all references are updated.

```
#!/bin/sh
# hooks/post-receive

# Read all references that were updated
while read oldrev newrev refname; do
    branch=$(echo $refname | cut -d/ -f3)

    # Deploy if main branch was updated
    if [ "$branch" = "main" ]; then
        echo "Deploying to production..."

        # Change to deployment directory
```

```
        cd /var/www/myapp

        # Pull latest changes
        git pull origin main

        # Install dependencies
        npm install --production

        # Restart application
        sudo systemctl restart myapp

        echo "Deployment completed"

        # Send notification
        curl -X POST -H 'Content-type: application/json' \
            --data '{"text":"Production deployment completed"}' \
            "$SLACK_WEBHOOK_URL"
    fi

    # Update staging if develop branch was updated
    if [ "$branch" = "develop" ]; then
        echo "Updating staging environment..."
        # Similar deployment process for staging
    fi
done
```

Hook Management and Distribution

Sharing Hooks with Team

Since hooks are not tracked by Git, you need alternative methods to share them:

Method 1: Hooks Directory in Repository

```
# Create hooks directory in repository
mkdir .githooks

# Create hooks in .githooks/
cat > .githooks/pre-commit << 'EOF'
#!/bin/sh
echo "Running pre-commit hook..."
# Hook content here
EOF
```

```
# Make executable
chmod +x .githooks/pre-commit

# Configure Git to use hooks from .githooks/
git config core.hooksPath .githooks

# Team members run:
git config core.hooksPath .githooks
```

Method 2: Installation Script

```
#!/bin/bash
# install-hooks.sh

HOOK_DIR=".git/hooks"
HOOKS_SOURCE="scripts/hooks"

# Copy hooks
for hook in pre-commit commit-msg pre-push; do
    if [ -f "$HOOKS_SOURCE/$hook" ]; then
        cp "$HOOKS_SOURCE/$hook" "$HOOK_DIR/$hook"
        chmod +x "$HOOK_DIR/$hook"
        echo "Installed $hook hook"
    fi
done

echo "Git hooks installation completed"
```

Hook Templates

Create reusable hook templates:

```
# templates/pre-commit-template
#!/bin/sh
# Pre-commit hook template

# Configuration
ENABLE_LINTING=true
ENABLE_TESTING=false
ENABLE_SECURITY_SCAN=true

# Source common functions
. "$(dirname "$0")/hook-functions"
```

```
# Run checks based on configuration
if [ "$ENABLE_LINTING" = true ]; then
    run_linting || exit 1
fi

if [ "$ENABLE_TESTING" = true ]; then
    run_tests || exit 1
fi

if [ "$ENABLE_SECURITY_SCAN" = true ]; then
    run_security_scan || exit 1
fi

exit 0
```

Advanced Hook Patterns

Language-Specific Hooks

Python Project Hook

```
#!/bin/sh
# .git/hooks/pre-commit

# Check Python syntax
python_files=$(git diff --cached --name-only --diff-filter=ACM | grep '\.py$')
if [ -n "$python_files" ]; then
    echo "Checking Python syntax..."
    for file in $python_files; do
        python -m py_compile "$file"
        if [ $? -ne 0 ]; then
            echo "Python syntax error in $file"
            exit 1
        fi
    done

    # Run Black formatter check
    if ! black --check $python_files; then
        echo "Code formatting issues found. Run 'black .' to fix."
        exit 1
    fi

    # Run flake8 linting
    if ! flake8 $python_files; then
```

```
        echo "Linting issues found. Please fix before committing."
        exit 1
    fi

    # Run mypy type checking
    if ! mypy $python_files; then
        echo "Type checking failed. Please fix type issues."
        exit 1
    fi
fi
```

Node.js Project Hook

```
#!/bin/sh
# .git/hooks/pre-commit

# Check for package.json changes
if git diff --cached --name-only | grep -q "package\.json"; then
    echo "package.json changed, updating package-lock.json..."
    npm install
    git add package-lock.json
fi

# Lint JavaScript/TypeScript files
js_files=$(git diff --cached --name-only --diff-filter=ACM | grep -E '\.(js|ts|jsx|tsx)$')
if [ -n "$js_files" ]; then
    echo "Linting JavaScript/TypeScript files..."
    npx eslint $js_files
    if [ $? -ne 0 ]; then
        echo "ESLint failed. Please fix the issues."
        exit 1
    fi

    # Type checking for TypeScript
    if echo "$js_files" | grep -q '\.tsx\?$'; then
        echo "Running TypeScript type checking..."
        npx tsc --noEmit
        if [ $? -ne 0 ]; then
            echo "TypeScript type checking failed."
            exit 1
        fi
    fi
fi

# Run tests for changed files
```

```
if [ -n "$js_files" ]; then
    echo "Running tests for changed files..."
    npx jest --findRelatedTests $js_files --passWithNoTests
    if [ $? -ne 0 ]; then
        echo "Tests failed for changed files."
        exit 1
    fi
fi
```

Security-Focused Hooks

```
#!/bin/sh
# .git/hooks/pre-commit - Security focused

# Check for secrets
echo "Scanning for secrets..."
if git diff --cached | grep -E "(password|secret|key|token)" -i; then
    echo "Warning: Potential secrets detected in commit"
    echo "Please review the changes carefully"
fi

# Check for hardcoded IPs and URLs
if git diff --cached | grep -E "([0-9]{1,3}\.){3}[0-9]{1,3}|https?://[a-zA-Z0-9.-]+"; then
    echo "Warning: Hardcoded IPs or URLs detected"
    echo "Consider using configuration files instead"
fi

# Scan for common security issues
security_patterns=(
    "eval\("
    "exec\("
    "system\("
    "shell_exec\("
    "passthru\("
    "innerHTML\s*="
    "document\.write\("
)

for pattern in "${security_patterns[@]}"; do
    if git diff --cached | grep -E "$pattern"; then
        echo "Security warning: Potentially dangerous pattern detected: $pattern"
    fi
done

# Run security scanner if available
```

```
if command -v bandit >/dev/null 2>&1; then
    python_files=$(git diff --cached --name-only --diff-filter=ACM | grep '\.py$')
    if [ -n "$python_files" ]; then
        echo "Running Bandit security scanner..."
        bandit $python_files
    fi
fi
```

Hook Automation Tools

Husky (Node.js)

Husky simplifies Git hooks management for Node.js projects:

```
# Install Husky
npm install --save-dev husky

# Initialize Husky
npx husky install

# Add pre-commit hook
npx husky add .husky/pre-commit "npm test"

# Add commit-msg hook
npx husky add .husky/commit-msg 'npx commitlint --edit "$1"'
```

Package.json configuration:

```
{
  "scripts": {
    "prepare": "husky install"
  },
  "husky": {
    "hooks": {
      "pre-commit": "lint-staged",
      "commit-msg": "commitlint -E HUSKY_GIT_PARAMS"
    }
  }
}
```

Pre-commit Framework

The pre-commit framework provides a multi-language hook manager:

```yaml
# .pre-commit-config.yaml
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.4.0
    hooks:
      - id: trailing-whitespace
      - id: end-of-file-fixer
      - id: check-yaml
      - id: check-added-large-files
      - id: check-merge-conflict

  - repo: https://github.com/psf/black
    rev: 22.10.0
    hooks:
      - id: black
        language_version: python3

  - repo: https://github.com/pycqa/flake8
    rev: 5.0.4
    hooks:
      - id: flake8

  - repo: https://github.com/pre-commit/mirrors-eslint
    rev: v8.28.0
    hooks:
      - id: eslint
        files: \.(js|ts|jsx|tsx)$
        types: [file]
```

Installation and usage:

```bash
# Install pre-commit
pip install pre-commit

# Install hooks
pre-commit install

# Run on all files
pre-commit run --all-files

# Update hooks
pre-commit autoupdate
```

Continuous Integration Integration

GitHub Actions with Hooks

```yaml
# .github/workflows/hooks-validation.yml
name: Validate Hooks

on: [push, pull_request]

jobs:
  validate-hooks:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Setup Node.js
      uses: actions/setup-node@v3
      with:
        node-version: '18'

    - name: Install dependencies
      run: npm ci

    - name: Run pre-commit checks
      run: |
        # Simulate pre-commit hook
        .githooks/pre-commit

    - name: Validate commit messages
      run: |
        # Check commit message format
        git log --oneline -10 | while read line; do
          message=$(echo "$line" | cut -d' ' -f2-)
          if ! echo "$message" | grep -qE '^(feat|fix|docs|style|refactor|test|chore)(\(.+\)
            echo "Invalid commit message: $message"
            exit 1
          fi
        done
```

Hook Testing

```
#!/bin/bash
# test-hooks.sh

# Test pre-commit hook
echo "Testing pre-commit hook..."

# Create test repository
test_repo=$(mktemp -d)
cd "$test_repo"
git init

# Copy hook
cp "$OLDPWD/.git/hooks/pre-commit" .git/hooks/
chmod +x .git/hooks/pre-commit

# Test with good commit
echo "console.log('test');" > test.js
git add test.js

if .git/hooks/pre-commit; then
    echo "ERROR: Hook should have failed on console.log"
    exit 1
else
    echo "PASS: Hook correctly rejected console.log"
fi

# Test with clean commit
echo "function test() { return true; }" > test.js
git add test.js

if .git/hooks/pre-commit; then
    echo "PASS: Hook accepted clean code"
else
    echo "ERROR: Hook should have passed clean code"
    exit 1
fi

# Cleanup
cd "$OLDPWD"
rm -rf "$test_repo"

echo "All hook tests passed!"
```

Exercises

Exercise 1: Basic Hook Setup

1. Create pre-commit hook that checks for debugging statements
2. Create commit-msg hook that enforces message format
3. Test hooks with various scenarios
4. Set up hook sharing for team

Exercise 2: Advanced Hook Development

1. Create language-specific hooks for your project
2. Implement security scanning in hooks
3. Add notification system to post-commit hook
4. Create hook testing framework

Exercise 3: Hook Automation

1. Set up Husky or pre-commit framework
2. Configure multiple hooks with different tools
3. Integrate hooks with CI/CD pipeline
4. Create hook performance monitoring

Exercise 4: Server-Side Hooks

1. Set up server-side hooks for deployment
2. Implement branch protection with hooks
3. Create automated testing on push
4. Add monitoring and alerting to server hooks

Best Practices

Hook Development

1. **Keep hooks fast** - slow hooks frustrate developers
2. **Provide clear error messages** - help developers fix issues
3. **Make hooks configurable** - allow team customization
4. **Test hooks thoroughly** - prevent blocking legitimate commits
5. **Document hook behavior** - team should understand what hooks do

Hook Management

1. **Version control hook scripts** - track changes to hooks
2. **Consistent installation** - automate hook setup for team
3. **Regular updates** - keep hooks current with project needs
4. **Performance monitoring** - track hook execution time
5. **Graceful degradation** - handle missing dependencies

Security Considerations

1. **Validate hook inputs** - don't trust user data
2. **Limit hook permissions** - run with minimal privileges
3. **Secure hook distribution** - verify hook integrity
4. **Audit hook changes** - review modifications carefully
5. **Monitor hook execution** - detect malicious activity

Summary

Git hooks provide powerful automation capabilities:

- **Quality assurance**: Automated testing and linting
- **Policy enforcement**: Commit message formats, branch protection
- **Integration**: CI/CD, notifications, deployments
- **Security**: Vulnerability scanning, secret detection
- **Workflow optimization**: Automated tasks and validations

Key concepts mastered: - Client-side and server-side hook types - Hook development and testing - Team hook distribution strategies - Integration with automation tools - Security and performance considerations

Git hooks transform Git from a simple version control tool into a comprehensive development workflow platform. They enable teams to maintain code quality, enforce standards, and automate repetitive tasks, ultimately improving productivity and reducing errors.

The next chapter will explore troubleshooting and recovery techniques, building upon the automation foundation to handle problems when they arise.

Troubleshooting

Chapter 14: Troubleshooting and Recovery

Common Git Problems and Solutions

Repository Corruption

Symptoms

- Git commands fail with "object not found" errors
- Repository appears corrupted or inconsistent
- Strange behavior during operations

Diagnosis

```
# Check repository integrity
git fsck --full

# Check for dangling objects
git fsck --unreachable

# Verify pack files
git verify-pack -v .git/objects/pack/pack-*.idx

# Check object database
git count-objects -v
```

Recovery Steps

```
# 1. Backup current repository
cp -r .git .git.backup

# 2. Try garbage collection
git gc --aggressive --prune=now

# 3. Rebuild index if corrupted
rm .git/index
git reset
```

```
# 4. If severe corruption, clone fresh copy
git clone --mirror <remote-url> .git.new
mv .git .git.corrupted
mv .git.new .git
git reset --hard HEAD
```

Lost Commits

Using Reflog

```
# View reflog to find lost commits
git reflog

# Look for entries like:
# abc1234 HEAD@{5}: reset: moving to HEAD~3
# def5678 HEAD@{6}: commit: Important work

# Recover lost commit
git checkout def5678
git branch recovery-branch

# Or reset to lost commit
git reset --hard def5678
```

Finding Dangling Commits

```
# Find unreachable commits
git fsck --unreachable | grep commit

# Examine unreachable commits
git show <commit-hash>

# Create branch from recovered commit
git branch recovered-work <commit-hash>
```

Recovery from Hard Reset

```
# Find commit before reset
git reflog | grep "reset:"

# Example output:
# abc1234 HEAD@{1}: reset: moving to HEAD~5
```

```
# def5678 HEAD@{2}: commit: Work before reset

# Recover to state before reset
git reset --hard HEAD@{2}

# Or create branch to preserve current state
git branch before-recovery
git reset --hard def5678
```

Branch Recovery

Deleted Branch Recovery

```
# Find deleted branch in reflog
git reflog | grep "checkout: moving from deleted-branch"

# Example:
# abc1234 HEAD@{10}: checkout: moving from feature-branch to main

# Recreate branch
git branch feature-branch abc1234

# Verify recovery
git log feature-branch --oneline
```

Corrupted Branch Reference

```
# Check branch references
cat .git/refs/heads/branch-name

# If reference is corrupted, find correct commit
git log --all --grep="last known commit message"

# Update branch reference
git update-ref refs/heads/branch-name <correct-commit-hash>
```

Merge Conflicts Resolution

Complex Merge Conflicts

```
# Abort current merge
git merge --abort

# Use different merge strategy
git merge -X ours feature-branch    # Prefer our changes
git merge -X theirs feature-branch  # Prefer their changes

# Use specific merge strategy
git merge -s recursive -X patience feature-branch
```

Binary File Conflicts

```
# For binary files, choose one version
git checkout --ours binary-file.jpg
git checkout --theirs binary-file.jpg

# Add resolved file
git add binary-file.jpg
git commit
```

Resolving Rename Conflicts

```
# When files are renamed differently in branches
# Git shows both versions

# Choose the correct name and remove the other
git rm old-name.txt
git add new-name.txt
git commit
```

Rebase Problems

Rebase Conflicts

```
# During rebase, conflicts occur
git status  # Shows conflicted files

# Resolve conflicts in files
```

```
vim conflicted-file.txt

# Stage resolved files
git add conflicted-file.txt

# Continue rebase
git rebase --continue

# Or skip problematic commit
git rebase --skip

# Or abort rebase
git rebase --abort
```

Rebase Gone Wrong

```
# Find state before rebase
git reflog | grep "rebase"

# Example:
# abc1234 HEAD@{5}: rebase -i (start): checkout HEAD~3
# def5678 HEAD@{6}: commit: Work before rebase

# Reset to before rebase
git reset --hard HEAD@{6}

# Or create recovery branch
git branch rebase-recovery HEAD@{6}
```

Interactive Rebase Issues

```
# If rebase editor shows wrong commits
git rebase --edit-todo

# If you need to modify a commit during rebase
git commit --amend
git rebase --continue

# If rebase creates duplicate commits
git rebase --onto <new-base> <old-base> <branch>
```

Remote Repository Issues

Push Rejected

```
# Error: Updates were rejected because remote contains work

# Solution 1: Pull and merge
git pull origin main
git push origin main

# Solution 2: Pull with rebase
git pull --rebase origin main
git push origin main

# Solution 3: Force push (dangerous)
git push --force-with-lease origin main
```

Diverged Branches

```
# Local and remote have diverged
git status
# Your branch and 'origin/main' have diverged

# View divergence
git log --oneline --graph origin/main main

# Solution 1: Merge remote changes
git merge origin/main

# Solution 2: Rebase onto remote
git rebase origin/main

# Solution 3: Reset to remote (loses local changes)
git reset --hard origin/main
```

Authentication Issues

```
# SSH key problems
ssh -T git@github.com

# If key not found
ssh-add ~/.ssh/id_rsa

# Generate new key if needed
```

```
ssh-keygen -t ed25519 -C "your.email@example.com"

# HTTPS credential issues
git config --global credential.helper store
# Or use credential manager
git config --global credential.helper manager
```

Working Directory Issues

Untracked Files Blocking Operations

```
# Error: The following untracked files would be overwritten

# Solution 1: Stash including untracked files
git stash -u

# Solution 2: Remove untracked files
git clean -fd

# Solution 3: Add files to .gitignore
echo "problematic-file" >> .gitignore
git add .gitignore
git commit -m "Add gitignore entry"
```

File Permission Issues

```
# Git tracking file permission changes
git config core.filemode false

# Reset file permissions
git diff --summary | grep --color=never "mode change" | cut -d' ' -f7- | xargs chmod 644

# For executable files
find . -name "*.sh" -exec chmod +x {} \;
```

Line Ending Issues

```
# Configure line ending handling
git config --global core.autocrlf true   # Windows
git config --global core.autocrlf input  # macOS/Linux

# Fix existing line ending issues
```

```
git add --renormalize .
git commit -m "Normalize line endings"
```

Advanced Recovery Techniques

Repository Reconstruction

From Corrupted Repository

```
# Create new repository
mkdir recovered-repo
cd recovered-repo
git init

# Extract objects from corrupted repository
cp -r ../corrupted-repo/.git/objects .git/

# Find all commits
git fsck --unreachable | grep commit | cut -d' ' -f3 > commits.txt

# Examine commits to find branch heads
while read commit; do
    echo "=== $commit ==="
    git log --oneline -1 $commit
    git show --name-only $commit
done < commits.txt

# Recreate branches
git branch main <main-branch-commit>
git branch feature <feature-branch-commit>
```

From Backup Files

```
# If you have file backups but no Git history
git init
git add .
git commit -m "Restore from backup"

# If you have multiple backup versions
for backup in backup-*; do
    cp -r "$backup"/* .
    git add .
    git commit -m "Restore from $backup"
```

```
done
```

Data Recovery Tools

Git Filter-Repo for History Cleanup

```
# Install git-filter-repo
pip install git-filter-repo

# Remove sensitive file from entire history
git filter-repo --path sensitive-file.txt --invert-paths

# Remove large files
git filter-repo --strip-blobs-bigger-than 10M

# Change author information
echo "Old Name <old@email.com> = New Name <new@email.com>" > mailmap
git filter-repo --mailmap mailmap
```

BFG Repo-Cleaner

```
# Download BFG from https://rtyley.github.io/bfg-repo-cleaner/

# Remove passwords from history
java -jar bfg.jar --replace-text passwords.txt my-repo.git

# Remove large files
java -jar bfg.jar --strip-blobs-bigger-than 100M my-repo.git

# Clean up after BFG
cd my-repo.git
git reflog expire --expire=now --all
git gc --prune=now --aggressive
```

Emergency Procedures

Complete Repository Recovery

```
#!/bin/bash
# emergency-recovery.sh

REPO_PATH="$1"
```

```
BACKUP_PATH="${REPO_PATH}.emergency-backup"

echo "Starting emergency recovery for $REPO_PATH"

# 1. Create backup
echo "Creating backup..."
cp -r "$REPO_PATH" "$BACKUP_PATH"

# 2. Check repository integrity
echo "Checking repository integrity..."
cd "$REPO_PATH"
if ! git fsck --full; then
    echo "Repository corruption detected"
fi

# 3. Attempt automatic recovery
echo "Attempting automatic recovery..."
git gc --aggressive --prune=now

# 4. Check if recovery successful
if git fsck --full; then
    echo "Recovery successful"
else
    echo "Automatic recovery failed, manual intervention required"
    exit 1
fi

# 5. Verify branches
echo "Verifying branches..."
git branch -a

# 6. Check recent commits
echo "Recent commits:"
git log --oneline -10

echo "Emergency recovery completed"
```

Disaster Recovery Plan

```
#!/bin/bash
# disaster-recovery.sh

# Configuration
REMOTE_URL="https://github.com/user/repo.git"
LOCAL_PATH="./recovered-repo"
```

```
BACKUP_PATH="./repo-backup"

echo "=== DISASTER RECOVERY PROCEDURE ==="

# Step 1: Assess damage
echo "1. Assessing repository damage..."
if [ -d ".git" ]; then
    git fsck --full > fsck-report.txt 2>&1
    if [ $? -eq 0 ]; then
        echo "Repository appears intact"
        exit 0
    else
        echo "Repository corruption detected"
    fi
else
    echo "No Git repository found"
fi

# Step 2: Create backup of current state
echo "2. Creating backup of current state..."
if [ -d ".git" ]; then
    cp -r . "$BACKUP_PATH"
    echo "Backup created at $BACKUP_PATH"
fi

# Step 3: Clone fresh copy from remote
echo "3. Cloning fresh copy from remote..."
git clone "$REMOTE_URL" "$LOCAL_PATH"

# Step 4: Recover local changes
echo "4. Attempting to recover local changes..."
if [ -d "$BACKUP_PATH" ]; then
    cd "$LOCAL_PATH"

    # Copy working directory changes
    rsync -av --exclude='.git' "$BACKUP_PATH/" ./

    # Show what was recovered
    git status

    echo "Local changes recovered. Review and commit as needed."
fi

echo "=== RECOVERY COMPLETE ==="
```

Preventive Measures

Repository Health Monitoring

Health Check Script

```
#!/bin/bash
# repo-health-check.sh

echo "=== REPOSITORY HEALTH CHECK ==="

# Check repository integrity
echo "Checking repository integrity..."
if git fsck --full --strict; then
    echo "  Repository integrity: OK"
else
    echo "  Repository integrity: ISSUES FOUND"
fi

# Check repository size
echo "Checking repository size..."
repo_size=$(du -sh .git | cut -f1)
echo "Repository size: $repo_size"

# Check for large objects
echo "Checking for large objects..."
git rev-list --objects --all | \
git cat-file --batch-check='%(objecttype) %(objectname) %(objectsize) %(rest)' | \
grep '^blob' | sort -k3nr | head -5 | \
while read type hash size path; do
    echo "Large object: $path ($size bytes)"
done

# Check branch count
branch_count=$(git branch -a | wc -l)
echo "Branch count: $branch_count"

# Check recent activity
echo "Recent activity (last 7 days):"
git log --since="7 days ago" --oneline | wc -l | xargs echo "Commits:"

# Check for potential issues
echo "Checking for potential issues..."

# Uncommitted changes
if ! git diff-index --quiet HEAD --; then
```

```
    echo "  Uncommitted changes detected"
fi

# Untracked files
if [ -n "$(git ls-files --others --exclude-standard)" ]; then
    echo "  Untracked files present"
fi

# Stashes
stash_count=$(git stash list | wc -l)
if [ $stash_count -gt 0 ]; then
    echo "  $stash_count stashes present"
fi

echo "=== HEALTH CHECK COMPLETE ==="
```

Backup Strategies

Automated Backup Script

```
#!/bin/bash
# backup-repo.sh

REPO_PATH="$1"
BACKUP_DIR="/backups/git-repos"
DATE=$(date +%Y%m%d_%H%M%S)
REPO_NAME=$(basename "$REPO_PATH")

# Create backup directory
mkdir -p "$BACKUP_DIR"

# Create bare clone backup
echo "Creating backup of $REPO_NAME..."
git clone --bare "$REPO_PATH" "$BACKUP_DIR/${REPO_NAME}_${DATE}.git"

# Compress backup
cd "$BACKUP_DIR"
tar -czf "${REPO_NAME}_${DATE}.tar.gz" "${REPO_NAME}_${DATE}.git"
rm -rf "${REPO_NAME}_${DATE}.git"

# Keep only last 10 backups
ls -t "${REPO_NAME}"_*.tar.gz | tail -n +11 | xargs rm -f

echo "Backup completed: $BACKUP_DIR/${REPO_NAME}_${DATE}.tar.gz"
```

Remote Backup Strategy

```
#!/bin/bash
# remote-backup.sh

# Multiple remote backups
git remote add backup-github https://github.com/user/repo-backup.git
git remote add backup-gitlab https://gitlab.com/user/repo-backup.git

# Push to all remotes
for remote in origin backup-github backup-gitlab; do
    echo "Pushing to $remote..."
    git push "$remote" --all
    git push "$remote" --tags
done
```

Monitoring and Alerting

Repository Monitor

```
#!/bin/bash
# repo-monitor.sh

REPO_PATH="$1"
ALERT_EMAIL="admin@example.com"

cd "$REPO_PATH"

# Check for corruption
if ! git fsck --full --quiet; then
    echo "Repository corruption detected in $REPO_PATH" | \
    mail -s "Git Repository Alert" "$ALERT_EMAIL"
fi

# Check repository size
size=$(du -s .git | cut -f1)
if [ $size -gt 1000000 ]; then  # 1GB in KB
    echo "Repository size exceeded 1GB: $REPO_PATH" | \
    mail -s "Git Repository Size Alert" "$ALERT_EMAIL"
fi

# Check for old branches
git for-each-ref --format='%(refname:short) %(committerdate)' refs/heads | \
while read branch date; do
    if [ $(date -d "$date" +%s) -lt $(date -d "90 days ago" +%s) ]; then
```

```
        echo "Old branch detected: $branch (last commit: $date)"
    fi
done
```

Exercises

Exercise 1: Corruption Recovery

1. Create a repository and simulate corruption
2. Practice using fsck and recovery commands
3. Implement automated health checks
4. Test backup and restore procedures

Exercise 2: Conflict Resolution

1. Create complex merge conflicts
2. Practice different resolution strategies
3. Handle binary file conflicts
4. Resolve rebase conflicts

Exercise 3: History Recovery

1. Simulate lost commits and branches
2. Use reflog for recovery
3. Practice filter-repo for cleanup
4. Implement emergency recovery procedures

Exercise 4: Preventive Measures

1. Set up automated backups
2. Implement repository monitoring
3. Create disaster recovery plan
4. Test recovery procedures

Best Practices

Prevention

1. **Regular backups**: Automate repository backups
2. **Health monitoring**: Regular integrity checks
3. **Team training**: Educate team on Git best practices
4. **Branch protection**: Prevent direct pushes to important branches
5. **Code review**: Catch issues before they enter main branch

Recovery

1. **Stay calm**: Don't panic when issues occur
2. **Backup first**: Always backup before attempting recovery
3. **Understand the problem**: Diagnose before attempting fixes
4. **Document incidents**: Learn from recovery experiences
5. **Test procedures**: Regularly test recovery procedures

Communication

1. **Alert team**: Inform team of repository issues
2. **Document steps**: Record recovery procedures
3. **Share knowledge**: Train team members on recovery
4. **Post-mortem**: Analyze incidents to prevent recurrence
5. **Update procedures**: Improve based on experience

Summary

Effective Git troubleshooting and recovery requires:

- **Diagnostic skills**: Understanding Git internals and error messages
- **Recovery techniques**: Using reflog, fsck, and specialized tools
- **Preventive measures**: Monitoring, backups, and health checks
- **Emergency procedures**: Systematic approach to disaster recovery
- **Team coordination**: Communication and documentation

Key skills developed: - Repository corruption diagnosis and repair - Commit and branch recovery techniques - Conflict resolution strategies - Advanced recovery tools usage - Preventive monitoring implementation

Git's distributed nature provides inherent resilience, but understanding recovery techniques ensures you can handle any situation confidently. These skills are essential for maintaining repository health and team productivity.

The next chapter will explore GitHub's advanced features, building upon this solid foundation of Git troubleshooting and recovery skills.

GitHub Actions

Chapter 15: GitHub Actions and CI/CD

Introduction to GitHub Actions

GitHub Actions is a powerful automation platform that allows you to build, test, and deploy your code directly from your GitHub repository. It enables continuous integration and continuous deployment (CI/CD) workflows.

Key Concepts

- **Workflow**: Automated process defined in YAML files
- **Event**: Trigger that starts a workflow (push, pull request, schedule)
- **Job**: Set of steps that execute on the same runner
- **Step**: Individual task within a job
- **Action**: Reusable unit of code for a step
- **Runner**: Server that executes workflows

Benefits of GitHub Actions

- **Integrated**: Built into GitHub platform
- **Flexible**: Supports any language and framework
- **Scalable**: Runs on GitHub-hosted or self-hosted runners
- **Community**: Large marketplace of pre-built actions
- **Cost-effective**: Free tier for public repositories

Workflow Basics

Workflow File Structure

Workflows are defined in `.github/workflows/` directory as YAML files:

```yaml
name: CI/CD Pipeline

on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]
```

```yaml
jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3
    - name: Setup Node.js
      uses: actions/setup-node@v3
      with:
        node-version: '18'
    - name: Install dependencies
      run: npm install
    - name: Run tests
      run: npm test
```

Workflow Triggers (Events)

Push Events

```yaml
on:
  push:
    branches: [ main, develop ]
    paths: [ 'src/**', 'tests/**' ]
    tags: [ 'v*' ]
```

Pull Request Events

```yaml
on:
  pull_request:
    branches: [ main ]
    types: [ opened, synchronize, reopened ]
```

Scheduled Events

```yaml
on:
  schedule:
    - cron: '0 2 * * *'  # Daily at 2 AM UTC
```

Manual Triggers

```yaml
on:
  workflow_dispatch:
    inputs:
      environment:
        description: 'Environment to deploy to'
        required: true
        default: 'staging'
        type: choice
        options:
        - staging
        - production
```

Multiple Events

```yaml
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
  schedule:
    - cron: '0 2 * * *'
  workflow_dispatch:
```

Jobs and Steps

Job Configuration

```yaml
jobs:
  build:
    name: Build Application
    runs-on: ubuntu-latest
    timeout-minutes: 30

    strategy:
      matrix:
        node-version: [16, 18, 20]
        os: [ubuntu-latest, windows-latest, macos-latest]

    steps:
      # Job steps here
```

Step Types

Using Actions

```yaml
steps:
- name: Checkout code
  uses: actions/checkout@v3

- name: Setup Node.js
  uses: actions/setup-node@v3
  with:
    node-version: '18'
    cache: 'npm'
```

Running Commands

```yaml
steps:
- name: Install dependencies
  run: npm install

- name: Run multiple commands
  run: |
    echo "Building application..."
    npm run build
    echo "Build complete!"
```

Conditional Steps

```yaml
steps:
- name: Deploy to production
  if: github.ref == 'refs/heads/main'
  run: npm run deploy:prod

- name: Deploy to staging
  if: github.ref != 'refs/heads/main'
  run: npm run deploy:staging
```

Common CI/CD Patterns

Node.js Application Workflow

```yaml
name: Node.js CI/CD

on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest

    strategy:
      matrix:
        node-version: [16, 18, 20]

    steps:
    - uses: actions/checkout@v3

    - name: Use Node.js ${{ matrix.node-version }}
      uses: actions/setup-node@v3
      with:
        node-version: ${{ matrix.node-version }}
        cache: 'npm'

    - name: Install dependencies
      run: npm ci

    - name: Run linter
      run: npm run lint

    - name: Run tests
      run: npm test

    - name: Run build
      run: npm run build

    - name: Upload coverage reports
      uses: codecov/codecov-action@v3
      if: matrix.node-version == '18'

  deploy:
```

```
    needs: test
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'

    steps:
    - uses: actions/checkout@v3

    - name: Setup Node.js
      uses: actions/setup-node@v3
      with:
        node-version: '18'
        cache: 'npm'

    - name: Install dependencies
      run: npm ci

    - name: Build application
      run: npm run build

    - name: Deploy to production
      run: npm run deploy
      env:
        DEPLOY_TOKEN: ${{ secrets.DEPLOY_TOKEN }}
```

Python Application Workflow

```
name: Python CI/CD

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest

    strategy:
      matrix:
        python-version: [3.8, 3.9, '3.10', 3.11]

    steps:
    - uses: actions/checkout@v3
```

```
    - name: Set up Python ${{ matrix.python-version }}
      uses: actions/setup-python@v4
      with:
        python-version: ${{ matrix.python-version }}

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt
        pip install -r requirements-dev.txt

    - name: Lint with flake8
      run: |
        flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
        flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 --statistics

    - name: Test with pytest
      run: |
        pytest --cov=./ --cov-report=xml

    - name: Upload coverage to Codecov
      uses: codecov/codecov-action@v3
      if: matrix.python-version == '3.10'
```

Docker Build and Push

```
name: Docker Build and Push

on:
  push:
    branches: [ main ]
    tags: [ 'v*' ]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout
      uses: actions/checkout@v3

    - name: Set up Docker Buildx
      uses: docker/setup-buildx-action@v2

    - name: Login to Docker Hub
```

```
      uses: docker/login-action@v2
      with:
        username: ${{ secrets.DOCKERHUB_USERNAME }}
        password: ${{ secrets.DOCKERHUB_TOKEN }}

    - name: Extract metadata
      id: meta
      uses: docker/metadata-action@v4
      with:
        images: myusername/myapp
        tags: |
          type=ref,event=branch
          type=ref,event=pr
          type=semver,pattern={{version}}
          type=semver,pattern={{major}}.{{minor}}

    - name: Build and push
      uses: docker/build-push-action@v4
      with:
        context: .
        push: true
        tags: ${{ steps.meta.outputs.tags }}
        labels: ${{ steps.meta.outputs.labels }}
        cache-from: type=gha
        cache-to: type=gha,mode=max
```

Secrets and Environment Variables

Managing Secrets

```
# Repository Settings > Secrets and variables > Actions
# Add secrets like:
# - API_KEYS
# - DEPLOY_TOKENS
# - DATABASE_PASSWORDS

steps:
- name: Deploy application
  run: ./deploy.sh
  env:
    API_KEY: ${{ secrets.API_KEY }}
    DATABASE_URL: ${{ secrets.DATABASE_URL }}
```

Environment Variables

```yaml
env:
  NODE_ENV: production
  API_URL: https://api.example.com

jobs:
  build:
    runs-on: ubuntu-latest
    env:
      BUILD_ENV: staging

    steps:
    - name: Build with environment
      run: npm run build
      env:
        SPECIFIC_VAR: value
```

Environment Protection

```yaml
jobs:
  deploy:
    runs-on: ubuntu-latest
    environment: production  # Requires approval for production

    steps:
    - name: Deploy to production
      run: ./deploy.sh
      env:
        PROD_TOKEN: ${{ secrets.PROD_TOKEN }}
```

Matrix Builds

Basic Matrix

```yaml
jobs:
  test:
    runs-on: ubuntu-latest

    strategy:
      matrix:
        node-version: [16, 18, 20]
        os: [ubuntu-latest, windows-latest, macos-latest]
```

```
    steps:
    - uses: actions/checkout@v3
    - uses: actions/setup-node@v3
      with:
        node-version: ${{ matrix.node-version }}
```

Complex Matrix

```
jobs:
  test:
    runs-on: ${{ matrix.os }}

    strategy:
      fail-fast: false
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]
        node-version: [16, 18, 20]
        include:
          - os: ubuntu-latest
            node-version: 20
            experimental: true
        exclude:
          - os: windows-latest
            node-version: 16

    steps:
    - name: Test on ${{ matrix.os }} with Node ${{ matrix.node-version }}
      run: npm test
```

Artifacts and Caching

Uploading Artifacts

```
steps:
- name: Build application
  run: npm run build

- name: Upload build artifacts
  uses: actions/upload-artifact@v3
  with:
    name: build-files
    path: |
      dist/
```

```
      build/
    retention-days: 30
```

Downloading Artifacts

```yaml
steps:
- name: Download build artifacts
  uses: actions/download-artifact@v3
  with:
    name: build-files
    path: ./artifacts
```

Caching Dependencies

```yaml
steps:
- uses: actions/checkout@v3

- name: Cache Node modules
  uses: actions/cache@v3
  with:
    path: ~/.npm
    key: ${{ runner.os }}-node-${{ hashFiles('**/package-lock.json') }}
    restore-keys: |
      ${{ runner.os }}-node-

- name: Install dependencies
  run: npm ci
```

Custom Actions

JavaScript Action

```yaml
# .github/actions/hello-world/action.yml
name: 'Hello World'
description: 'Greet someone and record the time'
inputs:
  who-to-greet:
    description: 'Who to greet'
    required: true
    default: 'World'
outputs:
```

```
  time:
    description: 'The time we greeted you'
runs:
  using: 'node16'
  main: 'index.js'
```

```
// .github/actions/hello-world/index.js
const core = require('@actions/core');
const github = require('@actions/github');

try {
  const nameToGreet = core.getInput('who-to-greet');
  console.log(`Hello ${nameToGreet}!`);

  const time = (new Date()).toTimeString();
  core.setOutput("time", time);

  const payload = JSON.stringify(github.context.payload, undefined, 2);
  console.log(`The event payload: ${payload}`);
} catch (error) {
  core.setFailed(error.message);
}
```

Using Custom Action

```
steps:
- uses: actions/checkout@v3
- uses: ./.github/actions/hello-world
  with:
    who-to-greet: 'GitHub Actions'
```

Deployment Strategies

Blue-Green Deployment

```
name: Blue-Green Deployment

on:
  push:
    branches: [ main ]

jobs:
```

```yaml
  deploy:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Deploy to green environment
      run: ./deploy-green.sh
      env:
        DEPLOY_TOKEN: ${{ secrets.DEPLOY_TOKEN }}

    - name: Run health checks
      run: ./health-check.sh green

    - name: Switch traffic to green
      run: ./switch-traffic.sh green

    - name: Cleanup blue environment
      run: ./cleanup-blue.sh
```

Rolling Deployment

```yaml
name: Rolling Deployment

on:
  push:
    branches: [ main ]

jobs:
  deploy:
    runs-on: ubuntu-latest

    strategy:
      matrix:
        server: [server1, server2, server3]

    steps:
    - uses: actions/checkout@v3

    - name: Deploy to ${{ matrix.server }}
      run: ./deploy.sh ${{ matrix.server }}
      env:
        SERVER_TOKEN: ${{ secrets.SERVER_TOKEN }}

    - name: Health check ${{ matrix.server }}
```

```
      run: ./health-check.sh ${{ matrix.server }}
```

Canary Deployment

```yaml
name: Canary Deployment

on:
  push:
    branches: [ main ]

jobs:
  canary:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Deploy canary (10% traffic)
      run: ./deploy-canary.sh 10
      env:
        DEPLOY_TOKEN: ${{ secrets.DEPLOY_TOKEN }}

    - name: Monitor canary metrics
      run: ./monitor-canary.sh
      timeout-minutes: 10

    - name: Full deployment
      if: success()
      run: ./deploy-full.sh

    - name: Rollback canary
      if: failure()
      run: ./rollback-canary.sh
```

Monitoring and Notifications

Slack Notifications

```yaml
steps:
- name: Notify Slack on success
  if: success()
  uses: 8398a7/action-slack@v3
  with:
```

```
    status: success
    text: 'Deployment successful! :rocket:'
  env:
    SLACK_WEBHOOK_URL: ${{ secrets.SLACK_WEBHOOK_URL }}

- name: Notify Slack on failure
  if: failure()
  uses: 8398a7/action-slack@v3
  with:
    status: failure
    text: 'Deployment failed! :x:'
  env:
    SLACK_WEBHOOK_URL: ${{ secrets.SLACK_WEBHOOK_URL }}
```

Email Notifications

```
steps:
- name: Send email notification
  if: always()
  uses: dawidd6/action-send-mail@v3
  with:
    server_address: smtp.gmail.com
    server_port: 465
    username: ${{ secrets.EMAIL_USERNAME }}
    password: ${{ secrets.EMAIL_PASSWORD }}
    subject: 'Build ${{ job.status }}: ${{ github.repository }}'
    body: |
      Build ${{ job.status }} for commit ${{ github.sha }}

      Repository: ${{ github.repository }}
      Branch: ${{ github.ref }}
      Commit: ${{ github.sha }}
      Author: ${{ github.actor }}
    to: team@example.com
    from: ci@example.com
```

Security Best Practices

Secure Secrets Management

```yaml
# Use environment-specific secrets
jobs:
  deploy-staging:
    environment: staging
    steps:
    - name: Deploy
      run: ./deploy.sh
      env:
        API_KEY: ${{ secrets.STAGING_API_KEY }}

  deploy-production:
    environment: production
    steps:
    - name: Deploy
      run: ./deploy.sh
      env:
        API_KEY: ${{ secrets.PRODUCTION_API_KEY }}
```

Least Privilege Access

```yaml
permissions:
  contents: read
  packages: write
  security-events: write

jobs:
  security-scan:
    runs-on: ubuntu-latest
    permissions:
      security-events: write

    steps:
    - uses: actions/checkout@v3
    - name: Run security scan
      uses: github/codeql-action/analyze@v2
```

Input Validation

```yaml
on:
  workflow_dispatch:
    inputs:
      environment:
        type: choice
        options:
        - staging
        - production
        required: true

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
    - name: Validate input
      run: |
        if [[ "${{ github.event.inputs.environment }}" != "staging" && "${{ github.event.inp
          echo "Invalid environment specified"
          exit 1
        fi
```

Exercises

Exercise 1: Basic CI Pipeline

1. Create a simple Node.js or Python project
2. Set up a basic CI workflow that:
   - Runs on push and pull requests
   - Tests multiple versions
   - Runs linting and tests
3. Test the workflow with different scenarios

Exercise 2: Docker Build Pipeline

1. Create a Dockerfile for your application
2. Set up a workflow that:
   - Builds Docker image
   - Pushes to registry
   - Tags appropriately
3. Test with different triggers

Exercise 3: Deployment Pipeline

1. Set up a staging and production environment
2. Create a deployment workflow that:
    - Deploys to staging automatically
    - Requires approval for production
    - Includes rollback capability
3. Test the approval process

Exercise 4: Custom Action

1. Create a custom action for your specific needs
2. Use the action in a workflow
3. Publish the action to the marketplace (optional)

Best Practices Summary

Workflow Design

1. **Keep workflows focused**: One responsibility per workflow
2. **Use matrix builds**: Test across multiple environments
3. **Fail fast**: Stop on first failure when appropriate
4. **Cache dependencies**: Speed up builds with caching
5. **Use artifacts**: Share data between jobs

Security

1. **Protect secrets**: Never log or expose secrets
2. **Use least privilege**: Minimal permissions required
3. **Validate inputs**: Check user inputs thoroughly
4. **Environment protection**: Require approvals for sensitive deployments
5. **Regular updates**: Keep actions and dependencies current

Performance

1. **Parallel jobs**: Run independent jobs concurrently
2. **Conditional execution**: Skip unnecessary steps
3. **Efficient caching**: Cache at appropriate levels
4. **Resource optimization**: Choose appropriate runner sizes
5. **Artifact management**: Clean up old artifacts

Summary

GitHub Actions provides comprehensive CI/CD capabilities:

- **Automated workflows**: Trigger on various events
- **Flexible execution**: Support for any language or platform
- **Integrated platform**: Built into GitHub ecosystem
- **Scalable infrastructure**: GitHub-hosted and self-hosted runners
- **Rich marketplace**: Thousands of pre-built actions

Key concepts mastered: - Workflow syntax and structure - Job and step configuration - Secrets and environment management - Matrix builds and parallel execution - Custom actions development - Deployment strategies - Security best practices

GitHub Actions enables modern DevOps practices, automating the entire software development lifecycle from code commit to production deployment. The next chapter will explore additional GitHub features that complement these automation capabilities.

GitHub Advanced

Chapter 16: GitHub Advanced Features

GitHub Pages

GitHub Pages allows you to host static websites directly from your GitHub repositories.

Setting Up GitHub Pages

Basic Setup

```
# Method 1: Use main branch
# 1. Create repository with HTML files
# 2. Go to Settings > Pages
# 3. Select source: Deploy from branch
# 4. Choose branch: main
# 5. Site available at: https://username.github.io/repository

# Method 2: Use docs/ folder
mkdir docs
echo "<h1>My Project Documentation</h1>" > docs/index.html
git add docs/
git commit -m "Add documentation"
git push origin main
# Configure Pages to use docs/ folder
```

Custom Domain

```
# Add CNAME file to repository root
echo "www.example.com" > CNAME
git add CNAME
git commit -m "Add custom domain"
git push origin main

# Configure DNS:
# CNAME record: www -> username.github.io
# A records for apex domain:
# 185.199.108.153
# 185.199.109.153
```

```
# 185.199.110.153
# 185.199.111.153
```

Jekyll Integration

Basic Jekyll Site

```yaml
# _config.yml
title: My Project
description: A great project description
baseurl: "/repository-name"
url: "https://username.github.io"

markdown: kramdown
highlighter: rouge
theme: minima

plugins:
  - jekyll-feed
  - jekyll-sitemap
```

```
---
layout: default
title: Home
---

# Welcome to My Project

This is the homepage of my project.

## Features

- Feature 1
- Feature 2
- Feature 3
```

Custom Jekyll Theme

```
# Create Gemfile
cat > Gemfile << 'EOF'
source "https://rubygems.org"
gem "github-pages", group: :jekyll_plugins
gem "jekyll-theme-minimal"
EOF
```

```
# Install dependencies
bundle install

# Update _config.yml
echo "theme: jekyll-theme-minimal" >> _config.yml
```

Advanced Pages Features

GitHub Actions for Pages

```
# .github/workflows/pages.yml
name: Deploy to GitHub Pages

on:
  push:
    branches: [ main ]
  workflow_dispatch:

permissions:
  contents: read
  pages: write
  id-token: write

concurrency:
  group: "pages"
  cancel-in-progress: true

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v3

      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '18'
          cache: 'npm'

      - name: Install dependencies
        run: npm ci

      - name: Build site
```

```yaml
        run: npm run build

      - name: Upload artifact
        uses: actions/upload-pages-artifact@v1
        with:
          path: ./dist

  deploy:
    environment:
      name: github-pages
      url: ${{ steps.deployment.outputs.page_url }}
    runs-on: ubuntu-latest
    needs: build
    steps:
      - name: Deploy to GitHub Pages
        id: deployment
        uses: actions/deploy-pages@v1
```

GitHub Packages

GitHub Packages is a package hosting service integrated with GitHub.

Publishing Packages

npm Package

```json
{
  "name": "@username/package-name",
  "version": "1.0.0",
  "publishConfig": {
    "registry": "https://npm.pkg.github.com"
  },
  "repository": {
    "type": "git",
    "url": "https://github.com/username/package-name.git"
  }
}
```

```bash
# Configure npm registry
echo "@username:registry=https://npm.pkg.github.com" >> .npmrc

# Login to GitHub Packages
npm login --registry=https://npm.pkg.github.com
```

```
# Publish package
npm publish
```

Docker Package

```
# Dockerfile
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
COPY . .
EXPOSE 3000
CMD ["npm", "start"]
```

```
# Build image
docker build -t ghcr.io/username/app-name:latest .

# Login to GitHub Container Registry
echo $GITHUB_TOKEN | docker login ghcr.io -u username --password-stdin

# Push image
docker push ghcr.io/username/app-name:latest
```

GitHub Actions for Package Publishing

```
name: Publish Package

on:
  release:
    types: [published]

jobs:
  publish-npm:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '18'
          registry-url: 'https://npm.pkg.github.com'
```

```yaml
      - name: Install dependencies
        run: npm ci

      - name: Publish to GitHub Packages
        run: npm publish
        env:
          NODE_AUTH_TOKEN: ${{ secrets.GITHUB_TOKEN }}

  publish-docker:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Login to Container Registry
        uses: docker/login-action@v2
        with:
          registry: ghcr.io
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      - name: Build and push
        uses: docker/build-push-action@v4
        with:
          context: .
          push: true
          tags: ghcr.io/${{ github.repository }}:latest
```

Security Features

Dependabot

Dependabot Configuration

```yaml
# .github/dependabot.yml
version: 2
updates:
  - package-ecosystem: "npm"
    directory: "/"
    schedule:
      interval: "weekly"
    open-pull-requests-limit: 10
    reviewers:
      - "username"
    assignees:
```

```
      - "username"
    commit-message:
      prefix: "npm"
      include: "scope"

  - package-ecosystem: "docker"
    directory: "/"
    schedule:
      interval: "weekly"

  - package-ecosystem: "github-actions"
    directory: "/"
    schedule:
      interval: "weekly"
```

Custom Dependabot Configuration

```
# Advanced Dependabot settings
version: 2
updates:
  - package-ecosystem: "npm"
    directory: "/"
    schedule:
      interval: "daily"
      time: "04:00"
      timezone: "America/New_York"
    allow:
      - dependency-type: "direct"
      - dependency-type: "indirect"
        update-type: "security"
    ignore:
      - dependency-name: "express"
        versions: ["4.x", "5.x"]
    labels:
      - "dependencies"
      - "npm"
    milestone: 4
    target-branch: "develop"
```

Security Advisories

Creating Security Advisory

```
# Security Advisory Template

## Summary
Brief description of the vulnerability.

## Details
Detailed explanation of the security issue.

## Impact
What kind of impact this vulnerability has.

## Patches
Information about patches or fixes.

## Workarounds
Temporary workarounds if patches aren't available.

## References
Links to additional information.

## Credits
Credit to security researchers who found the issue.
```

Private Vulnerability Reporting

```
# Enable private vulnerability reporting
# 1. Go to repository Settings
# 2. Security & analysis section
# 3. Enable "Private vulnerability reporting"

# Researchers can then report vulnerabilities privately
# You can collaborate on fixes before public disclosure
```

Code Scanning

CodeQL Analysis

```yaml
# .github/workflows/codeql-analysis.yml
name: "CodeQL"

on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]
  schedule:
    - cron: '0 2 * * 1'

jobs:
  analyze:
    name: Analyze
    runs-on: ubuntu-latest
    permissions:
      actions: read
      contents: read
      security-events: write

    strategy:
      fail-fast: false
      matrix:
        language: [ 'javascript', 'python' ]

    steps:
    - name: Checkout repository
      uses: actions/checkout@v3

    - name: Initialize CodeQL
      uses: github/codeql-action/init@v2
      with:
        languages: ${{ matrix.language }}
        queries: security-extended,security-and-quality

    - name: Autobuild
      uses: github/codeql-action/autobuild@v2

    - name: Perform CodeQL Analysis
      uses: github/codeql-action/analyze@v2
```

Third-Party Security Tools

```yaml
# Snyk security scanning
name: Snyk Security

on: [push, pull_request]

jobs:
  security:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Run Snyk to check for vulnerabilities
        uses: snyk/actions/node@master
        env:
          SNYK_TOKEN: ${{ secrets.SNYK_TOKEN }}
        with:
          args: --severity-threshold=high
```

Secret Scanning

Secret Scanning Configuration

```yaml
# GitHub automatically scans for secrets
# Common patterns detected:
# - API keys
# - Tokens
# - Passwords
# - Private keys
# - Database connection strings

# Custom patterns can be defined for organizations
```

Preventing Secret Commits

```yaml
# Pre-commit hook to detect secrets
#!/bin/sh
# .git/hooks/pre-commit

# Check for common secret patterns
if git diff --cached | grep -E "(password|secret|key|token)" -i; then
    echo "Warning: Potential secrets detected"
```

```
    echo "Please review your changes"
fi

# Use tools like git-secrets
git secrets --scan
```

GitHub API

REST API Usage

Basic API Calls

```
# Get repository information
curl -H "Authorization: token $GITHUB_TOKEN" \
     https://api.github.com/repos/username/repository

# List issues
curl -H "Authorization: token $GITHUB_TOKEN" \
     https://api.github.com/repos/username/repository/issues

# Create issue
curl -X POST \
     -H "Authorization: token $GITHUB_TOKEN" \
     -H "Content-Type: application/json" \
     -d '{"title":"Bug report","body":"Description of bug"}' \
     https://api.github.com/repos/username/repository/issues
```

API with JavaScript

```
// Using Octokit
const { Octokit } = require("@octokit/rest");

const octokit = new Octokit({
  auth: process.env.GITHUB_TOKEN,
});

// Get repository
async function getRepository() {
  const { data } = await octokit.rest.repos.get({
    owner: "username",
    repo: "repository",
  });
  return data;
```

```
}

// Create issue
async function createIssue(title, body) {
  const { data } = await octokit.rest.issues.create({
    owner: "username",
    repo: "repository",
    title: title,
    body: body,
  });
  return data;
}

// List pull requests
async function listPullRequests() {
  const { data } = await octokit.rest.pulls.list({
    owner: "username",
    repo: "repository",
    state: "open",
  });
  return data;
}
```

GraphQL API

Basic GraphQL Queries

```
# Get repository information
query {
  repository(owner: "username", name: "repository") {
    name
    description
    stargazerCount
    forkCount
    issues(first: 10) {
      nodes {
        title
        state
        createdAt
      }
    }
  }
}
```

```
# Execute GraphQL query
curl -X POST \
  -H "Authorization: bearer $GITHUB_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"query":"query { viewer { login } }"}' \
  https://api.github.com/graphql
```

Advanced GraphQL Operations

```
// Using GraphQL with Octokit
const { graphql } = require("@octokit/graphql");

const graphqlWithAuth = graphql.defaults({
  headers: {
    authorization: `token ${process.env.GITHUB_TOKEN}`,
  },
});

// Complex query
const query = `
  query($owner: String!, $repo: String!, $first: Int!) {
    repository(owner: $owner, name: $repo) {
      pullRequests(first: $first, states: OPEN) {
        nodes {
          title
          author {
            login
          }
          reviews(first: 5) {
            nodes {
              state
              author {
                login
              }
            }
          }
        }
      }
    }
  }
`;

async function getPullRequestsWithReviews() {
  const result = await graphqlWithAuth(query, {
    owner: "username",
```

```
    repo: "repository",
    first: 10,
  });
  return result.repository.pullRequests.nodes;
}
```

GitHub CLI

Installation and Setup

```
# Install GitHub CLI
# macOS
brew install gh

# Ubuntu
curl -fsSL https://cli.github.com/packages/githubcli-archive-keyring.gpg | sudo dd of=/usr/s
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/githubcli-archive
sudo apt update
sudo apt install gh

# Authenticate
gh auth login
```

Common CLI Operations

Repository Management

```
# Clone repository
gh repo clone username/repository

# Create repository
gh repo create my-new-repo --public --description "My new repository"

# Fork repository
gh repo fork username/repository

# View repository
gh repo view username/repository

# List repositories
gh repo list username
```

Issue Management

```
# List issues
gh issue list

# Create issue
gh issue create --title "Bug report" --body "Description of bug"

# View issue
gh issue view 123

# Close issue
gh issue close 123

# Assign issue
gh issue edit 123 --add-assignee username
```

Pull Request Management

```
# Create pull request
gh pr create --title "Add new feature" --body "Description of changes"

# List pull requests
gh pr list

# View pull request
gh pr view 456

# Checkout pull request
gh pr checkout 456

# Merge pull request
gh pr merge 456 --squash

# Review pull request
gh pr review 456 --approve --body "Looks good!"
```

CLI Automation

Scripting with GitHub CLI

```
#!/bin/bash
# automated-pr-workflow.sh

# Create feature branch
git checkout -b feature/new-functionality

# Make changes
echo "New functionality" > new-feature.txt
git add new-feature.txt
git commit -m "Add new functionality"

# Push branch
git push -u origin feature/new-functionality

# Create pull request
gh pr create \
  --title "Add new functionality" \
  --body "This PR adds new functionality to the application" \
  --reviewer teammate1,teammate2 \
  --assignee myself

# Wait for approval and merge
echo "Pull request created. Waiting for approval..."
```

GitHub CLI Extensions

```
# Install extensions
gh extension install github/gh-copilot

# List installed extensions
gh extension list

# Use extension
gh copilot suggest "create a function to sort an array"

# Create custom extension
gh extension create my-extension
```

Advanced Automation

GitHub Apps

Creating GitHub App

```
// app.js - Simple GitHub App
const { App } = require("@octokit/app");
const { Octokit } = require("@octokit/rest");

const app = new App({
  appId: process.env.GITHUB_APP_ID,
  privateKey: process.env.GITHUB_PRIVATE_KEY,
});

// Handle webhook events
app.webhooks.on("issues.opened", async ({ octokit, payload }) => {
  const issue = payload.issue;

  // Auto-label new issues
  await octokit.rest.issues.addLabels({
    owner: payload.repository.owner.login,
    repo: payload.repository.name,
    issue_number: issue.number,
    labels: ["triage"],
  });

  // Add welcome comment
  await octokit.rest.issues.createComment({
    owner: payload.repository.owner.login,
    repo: payload.repository.name,
    issue_number: issue.number,
    body: "Thank you for opening this issue! We'll review it soon.",
  });
});

// Start server
const port = process.env.PORT || 3000;
app.webhooks.listen(port);
```

Webhooks

Webhook Configuration

```
// webhook-handler.js
const crypto = require("crypto");
const express = require("express");

const app = express();
app.use(express.json());

// Verify webhook signature
function verifySignature(payload, signature) {
  const hmac = crypto.createHmac("sha256", process.env.WEBHOOK_SECRET);
  const digest = "sha256=" + hmac.update(payload).digest("hex");
  return crypto.timingSafeEqual(Buffer.from(signature), Buffer.from(digest));
}

// Handle webhook events
app.post("/webhook", (req, res) => {
  const signature = req.headers["x-hub-signature-256"];
  const payload = JSON.stringify(req.body);

  if (!verifySignature(payload, signature)) {
    return res.status(401).send("Unauthorized");
  }

  const event = req.headers["x-github-event"];

  switch (event) {
    case "push":
      handlePushEvent(req.body);
      break;
    case "pull_request":
      handlePullRequestEvent(req.body);
      break;
    case "issues":
      handleIssueEvent(req.body);
      break;
  }

  res.status(200).send("OK");
});

function handlePushEvent(payload) {
  console.log(`Push to ${payload.repository.full_name}`);
```

```
  // Trigger deployment, run tests, etc.
}

function handlePullRequestEvent(payload) {
  console.log(`PR ${payload.action} in ${payload.repository.full_name}`);
  // Run checks, assign reviewers, etc.
}

app.listen(3000, () => {
  console.log("Webhook server listening on port 3000");
});
```

Exercises

Exercise 1: GitHub Pages Setup

1. Create a project documentation site with GitHub Pages
2. Set up custom domain and SSL
3. Implement automated deployment with GitHub Actions
4. Add Jekyll theme and customization

Exercise 2: Package Publishing

1. Create and publish an npm package to GitHub Packages
2. Set up Docker image publishing
3. Implement automated publishing on releases
4. Configure package access and permissions

Exercise 3: Security Implementation

1. Set up Dependabot for dependency updates
2. Configure CodeQL security scanning
3. Create security advisory for test vulnerability
4. Implement secret scanning prevention

Exercise 4: API Integration

1. Build application using GitHub REST API
2. Create GraphQL queries for complex data
3. Implement GitHub CLI automation scripts
4. Create simple GitHub App with webhooks

Best Practices

GitHub Pages

1. **Use custom domains** for professional appearance
2. **Implement HTTPS** for security
3. **Optimize for performance** with static site generators
4. **Automate deployments** with GitHub Actions
5. **Monitor site analytics** and performance

Security

1. **Enable all security features** available
2. **Regular security audits** of dependencies
3. **Implement secret scanning** prevention
4. **Monitor security advisories** for dependencies
5. **Train team** on security best practices

API Usage

1. **Use authentication** for all API calls
2. **Implement rate limiting** handling
3. **Cache responses** when appropriate
4. **Handle errors gracefully** in applications
5. **Use GraphQL** for complex queries

Automation

1. **Start simple** and build complexity gradually
2. **Test webhooks** thoroughly before production
3. **Monitor automation** performance and errors
4. **Document automation** workflows for team
5. **Implement proper error handling** and logging

Summary

GitHub's advanced features provide comprehensive platform capabilities:

- **GitHub Pages**: Static site hosting with custom domains and automation
- **GitHub Packages**: Integrated package registry for multiple ecosystems
- **Security Features**: Comprehensive security scanning and vulnerability management
- **APIs**: Powerful REST and GraphQL APIs for integration
- **GitHub CLI**: Command-line interface for automation and productivity
- **Apps and Webhooks**: Custom integrations and automation

Key skills developed: - Static site deployment and management - Package publishing and distribution - Security implementation and monitoring - API integration and automation - CLI scripting and workflow optimization - Custom app development and webhook handling

These advanced features transform GitHub from a simple code hosting platform into a comprehensive development ecosystem, enabling teams to build, secure, and deploy applications efficiently.

The next chapter will explore open source contribution, building upon these advanced GitHub features to participate effectively in the global open source community.

Open Source

Chapter 17: Open Source Contribution

Understanding Open Source

Open source software is software with source code that anyone can inspect, modify, and enhance. Contributing to open source projects is a valuable way to improve skills, build reputation, and give back to the community.

Benefits of Open Source Contribution

For Contributors

- **Skill Development**: Learn from experienced developers
- **Portfolio Building**: Showcase your work publicly
- **Networking**: Connect with developers worldwide
- **Career Advancement**: Demonstrate expertise to employers
- **Learning Opportunities**: Exposure to different technologies and practices

For Projects

- **Community Growth**: Expand user and contributor base
- **Quality Improvement**: More eyes on code means fewer bugs
- **Feature Development**: Contributors add desired functionality
- **Documentation**: Community helps improve documentation
- **Sustainability**: Shared maintenance burden

Types of Open Source Licenses

Permissive Licenses

```
MIT License
- Very permissive
- Allows commercial use
- Minimal restrictions

Apache License 2.0
- Patent protection included
- Requires attribution
- Popular for enterprise projects
```

```
BSD License
- Similar to MIT
- Multiple variants (2-clause, 3-clause)
- Used by many system projects
```

Copyleft Licenses

```
GNU GPL v3
- Requires derivative works to be open source
- Strong copyleft protection
- Used by Linux kernel and many GNU projects

GNU LGPL
- Lesser GPL for libraries
- Allows linking with proprietary software
- Good for library projects

Mozilla Public License 2.0
- File-level copyleft
- Allows mixing with proprietary code
- Used by Firefox and other Mozilla projects
```

Finding Projects to Contribute To

Discovering Projects

GitHub Exploration

```
# Search for projects by language
# Go to GitHub and search: language:python

# Look for beginner-friendly labels
# Search: label:"good first issue" language:javascript

# Find projects needing help
# Search: label:"help wanted" stars:>100

# Explore trending repositories
# Visit: https://github.com/trending
```

Specialized Platforms

```
# Platforms for finding open source projects:
- GitHub Explore: https://github.com/explore
- GitLab: https://gitlab.com/explore
- First Timers Only: https://www.firsttimersonly.com/
- Up For Grabs: https://up-for-grabs.net/
- Good First Issues: https://goodfirstissues.com/
- CodeTriage: https://www.codetriage.com/
```

Evaluating Projects

Project Health Indicators

```
# Check project activity
git log --oneline --since="3 months ago" | wc -l

# Look for recent commits
git log --oneline -10

# Check contributor activity
git shortlog -sn | head -10

# Examine issue response time
# Look at recent issues and their response times
```

Assessment Criteria

```
# Project evaluation checklist:
- [ ] Active development (recent commits)
- [ ] Responsive maintainers (issue responses)
- [ ] Clear documentation (README, CONTRIBUTING)
- [ ] Welcoming community (code of conduct)
- [ ] Good test coverage
- [ ] Continuous integration setup
- [ ] Clear issue labeling system
- [ ] Beginner-friendly issues available
```

Making Your First Contribution

Preparation Steps

Setting Up Development Environment

```
# 1. Fork the repository on GitHub
# 2. Clone your fork
git clone https://github.com/yourusername/project-name.git
cd project-name

# 3. Add upstream remote
git remote add upstream https://github.com/original-owner/project-name.git

# 4. Install dependencies (varies by project)
npm install          # Node.js projects
pip install -r requirements.txt  # Python projects
bundle install       # Ruby projects

# 5. Run tests to ensure everything works
npm test            # Node.js
python -m pytest   # Python
bundle exec rspec   # Ruby
```

Understanding the Codebase

```
# Explore project structure
tree -L 2

# Read important files
cat README.md
cat CONTRIBUTING.md
cat CODE_OF_CONDUCT.md

# Understand build process
cat package.json     # Node.js
cat Makefile        # C/C++
cat setup.py        # Python

# Run the project locally
npm start           # Web applications
python main.py      # Python scripts
```

Types of Contributions

Documentation Improvements

```
# Common documentation contributions:
- Fix typos and grammar errors
- Improve code examples
- Add missing documentation
- Translate documentation
- Update outdated information

# Example documentation fix:
## Before:
```javascript
// This function calcuates the sum
function add(a, b) {
    return a + b;
}
```

After:

```javascript
/**
 * Calculates the sum of two numbers
 * @param {number} a - First number
 * @param {number} b - Second number
 * @returns {number} The sum of a and b
 */
function add(a, b) {
    return a + b;
}
```

Bug Fixes
```bash
# 1. Find a bug report issue
# 2. Reproduce the bug locally
# 3. Create a branch for the fix
git checkout -b fix/issue-123-login-error

# 4. Write a test that demonstrates the bug
# 5. Fix the bug
# 6. Ensure all tests pass
npm test

# 7. Commit with descriptive message
git commit -m "Fix login error when username contains spaces
```
```

```
- Add input sanitization for usernames
- Update validation regex to allow spaces
- Add test cases for edge cases

Fixes #123"
```

Feature Implementation

```
# 1. Discuss feature in issue first
# 2. Create feature branch
git checkout -b feature/add-dark-mode

# 3. Implement feature incrementally
# 4. Add tests for new functionality
# 5. Update documentation
# 6. Ensure backward compatibility

# 7. Commit changes
git add .
git commit -m "Add dark mode support

- Add theme toggle component
- Implement CSS variables for theming
- Add user preference persistence
- Update documentation with theme usage

Implements #456"
```

Contribution Workflow

Standard Workflow

```
# 1. Sync with upstream
git checkout main
git pull upstream main
git push origin main

# 2. Create feature branch
git checkout -b contribution/improve-error-handling

# 3. Make changes and commit
git add .
git commit -m "Improve error handling in API client"
```

```
# 4. Push to your fork
git push -u origin contribution/improve-error-handling

# 5. Create pull request on GitHub
# 6. Address review feedback
# 7. Merge after approval
```

Handling Review Feedback

```
# Make requested changes
vim src/api-client.js

# Commit changes
git add src/api-client.js
git commit -m "Address review feedback: improve error messages"

# Push updates (automatically updates PR)
git push origin contribution/improve-error-handling

# If major changes needed, consider squashing commits
git rebase -i HEAD~3
# Squash related commits together
git push --force-with-lease origin contribution/improve-error-handling
```

Advanced Contribution Strategies

Becoming a Regular Contributor

Building Relationships

```
# Strategies for building relationships:
1. **Be consistent**: Regular, small contributions
2. **Be helpful**: Answer questions in issues
3. **Be respectful**: Follow code of conduct
4. **Be patient**: Understand maintainer constraints
5. **Be proactive**: Identify and solve problems

# Communication best practices:
- Use clear, concise language
- Provide context for changes
- Ask questions when unsure
- Thank maintainers for their time
- Help other contributors
```

Taking on Larger Tasks

```
# Progression path:
# 1. Documentation fixes
# 2. Small bug fixes
# 3. Feature implementations
# 4. Architecture improvements
# 5. Maintainer responsibilities

# Example: Refactoring project structure
git checkout -b refactor/improve-module-organization

# Plan the refactoring
# 1. Discuss with maintainers first
# 2. Create detailed plan
# 3. Implement in small, reviewable chunks
# 4. Maintain backward compatibility
# 5. Update documentation and examples
```

Contributing to Different Types of Projects

Library/Framework Contributions

```
// Example: Contributing to a JavaScript library
// 1. Understand the API design principles
// 2. Maintain consistency with existing code
// 3. Consider backward compatibility
// 4. Write comprehensive tests

// Before contribution:
function processData(data) {
    return data.map(item => item.value);
}

// After contribution (adding error handling):
function processData(data) {
    if (!Array.isArray(data)) {
        throw new TypeError('Expected data to be an array');
    }

    return data.map(item => {
        if (!item || typeof item.value === 'undefined') {
            throw new Error('Invalid data item: missing value property');
        }
        return item.value;
```

```
    });
}
```

Application Contributions

```
# Contributing to applications (web apps, desktop apps, etc.)
# 1. Understand user workflows
# 2. Consider UX/UI implications
# 3. Test across different environments
# 4. Consider accessibility requirements

# Example: Adding accessibility features
git checkout -b feature/improve-keyboard-navigation

# Add keyboard navigation support
# Update ARIA labels
# Test with screen readers
# Update documentation
```

Documentation Projects

```
# Contributing to documentation projects:
1. **Accuracy**: Ensure technical accuracy
2. **Clarity**: Write for the target audience
3. **Completeness**: Cover all necessary topics
4. **Examples**: Provide practical examples
5. **Structure**: Organize information logically

# Example documentation contribution:
## API Reference

### `authenticate(credentials)`

Authenticates a user with the provided credentials.

#### Parameters
- `credentials` (Object): User credentials
  - `username` (string): User's username
  - `password` (string): User's password

#### Returns
- `Promise<User>`: Resolves to authenticated user object

#### Example
```

```javascript
const user = await authenticate({
    username: 'john_doe',
    password: 'secure_password'
});
console.log(`Welcome, ${user.name}!`);
```

Throws

- `AuthenticationError`: When credentials are invalid
- `NetworkError`: When unable to connect to authentication service

```
## Maintaining Open Source Projects

### Project Setup

#### Repository Structure
```

project-name/ .github/ ISSUE_TEMPLATE/ bug_report.md feature_request.md question.md PULL_REQUEST_TEMPLATE.md workflows/ ci.yml release.yml docs/ api.md contributing.md installation.md src/ tests/ .gitignore CODE_OF_CONDUCT.md CONTRIBUTING.md LICENSE README.md package.json

```
#### Essential Files

##### README.md Template
```markdown
# Project Name

Brief description of what the project does.

## Features

- Feature 1
- Feature 2
- Feature 3

## Installation

```bash
npm install project-name
```

Quick Start

```
const project = require('project-name');

// Basic usage example
const result = project.doSomething();
console.log(result);
```

Documentation

- API Reference
- Contributing Guide
- Installation Guide

Contributing

We welcome contributions! Please see our Contributing Guide for details.

License

This project is licensed under the MIT License - see the LICENSE file for details.

Support

- Issue Tracker
- Discussions
- Documentation

```
##### CONTRIBUTING.md Template
```markdown
# Contributing to Project Name

Thank you for your interest in contributing! This guide will help you get started.

## Code of Conduct

This project adheres to our [Code of Conduct](CODE_OF_CONDUCT.md). By participating, you are ex

## Getting Started

1. Fork the repository
```
```

```
2. Clone your fork: `git clone https://github.com/yourusername/project-name.git`
3. Install dependencies: `npm install`
4. Run tests: `npm test`

## Development Workflow

1. Create a branch: `git checkout -b feature/your-feature-name`
2. Make your changes
3. Add tests for your changes
4. Run tests: `npm test`
5. Commit your changes: `git commit -m "Add your feature"`
6. Push to your fork: `git push origin feature/your-feature-name`
7. Create a pull request

## Coding Standards

- Use ESLint configuration provided
- Write tests for new features
- Update documentation as needed
- Follow existing code style

## Commit Message Format
```

type(scope): description

body (optional)

footer (optional)

```
Types: feat, fix, docs, style, refactor, test, chore

## Pull Request Process

1. Ensure all tests pass
2. Update documentation if needed
3. Add yourself to CONTRIBUTORS.md
4. Request review from maintainers

## Questions?

Feel free to open an issue or start a discussion!
```

Community Management

Issue Management

```
# Label system for issues
bug          # Something isn't working
enhancement  # New feature or request
documentation # Improvements or additions to documentation
good first issue # Good for newcomers
help wanted  # Extra attention is needed
question     # Further information is requested
wontfix      # This will not be worked on
duplicate    # This issue or pull request already exists
```

Automated Issue Management

```
# .github/workflows/issue-management.yml
name: Issue Management

on:
  issues:
    types: [opened]

jobs:
  label-issues:
    runs-on: ubuntu-latest
    steps:
      - name: Label new issues
        uses: actions/github-script@v6
        with:
          script: |
            github.rest.issues.addLabels({
              issue_number: context.issue.number,
              owner: context.repo.owner,
              repo: context.repo.repo,
              labels: ['triage']
            });

            github.rest.issues.createComment({
              issue_number: context.issue.number,
              owner: context.repo.owner,
              repo: context.repo.repo,
              body: 'Thank you for opening this issue! A maintainer will review it soon.'
            });
```

Release Management

```
# Semantic versioning
# MAJOR.MINOR.PATCH
# 1.0.0 -> 1.0.1 (patch: bug fixes)
# 1.0.1 -> 1.1.0 (minor: new features)
# 1.1.0 -> 2.0.0 (major: breaking changes)

# Create release
git tag -a v1.2.0 -m "Release version 1.2.0"
git push origin v1.2.0

# Automated releases with GitHub Actions
# .github/workflows/release.yml
name: Release

on:
  push:
    tags:
      - 'v*'

jobs:
  release:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Create Release
        uses: actions/create-release@v1
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        with:
          tag_name: ${{ github.ref }}
          release_name: Release ${{ github.ref }}
          draft: false
          prerelease: false
```

Open Source Best Practices

For Contributors

Communication

```
# Effective communication guidelines:
1. **Be clear and specific** in issue descriptions
2. **Provide context** for your contributions
3. **Ask questions** when requirements are unclear
4. **Be patient** with review processes
5. **Thank maintainers** for their time and effort

# Example good issue report:
## Bug Report

**Description**: Login fails when username contains special characters

**Steps to Reproduce**:
1. Navigate to login page
2. Enter username: "user@domain.com"
3. Enter valid password
4. Click login button

**Expected Behavior**: User should be logged in successfully

**Actual Behavior**: Error message "Invalid username format"

**Environment**:
- Browser: Chrome 91.0.4472.124
- OS: macOS 11.4
- App Version: 2.1.0

**Additional Context**: This worked in version 2.0.0
```

Code Quality

```
// Example of good contribution practices

// Before: Poor code quality
function calc(x, y) {
    return x + y;
}

// After: Good code quality
```

```
/**
 * Calculates the sum of two numbers
 * @param {number} firstNumber - The first number to add
 * @param {number} secondNumber - The second number to add
 * @returns {number} The sum of the two numbers
 * @throws {TypeError} When either parameter is not a number
 */
function calculateSum(firstNumber, secondNumber) {
    if (typeof firstNumber !== 'number' || typeof secondNumber !== 'number') {
        throw new TypeError('Both parameters must be numbers');
    }

    return firstNumber + secondNumber;
}

// Include comprehensive tests
describe('calculateSum', () => {
    test('should add two positive numbers', () => {
        expect(calculateSum(2, 3)).toBe(5);
    });

    test('should handle negative numbers', () => {
        expect(calculateSum(-2, 3)).toBe(1);
    });

    test('should throw error for non-number inputs', () => {
        expect(() => calculateSum('2', 3)).toThrow(TypeError);
    });
});
```

For Maintainers

Welcoming New Contributors

```
# Strategies for welcoming new contributors:
1. **Create beginner-friendly issues** with "good first issue" label
2. **Provide detailed contribution guidelines**
3. **Respond promptly** to questions and pull requests
4. **Give constructive feedback** during code reviews
5. **Recognize contributions** publicly

# Example welcoming response:
"Thank you for your first contribution to our project! This is a great start.
I've left some feedback on your pull request to help improve the code.
Don't worry about getting everything perfect on the first try - we're here to help!"
```

Sustainable Maintenance

```
# Strategies for sustainable maintenance:
# 1. Automate repetitive tasks
# 2. Delegate responsibilities to trusted contributors
# 3. Set clear boundaries and expectations
# 4. Take breaks to prevent burnout
# 5. Build a community of maintainers

# Example automation for maintenance tasks
# .github/workflows/stale.yml
name: Mark stale issues and pull requests

on:
  schedule:
  - cron: "0 0 * * *"

jobs:
  stale:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/stale@v5
      with:
        repo-token: ${{ secrets.GITHUB_TOKEN }}
        stale-issue-message: 'This issue has been automatically marked as stale because it h
        stale-pr-message: 'This pull request has been automatically marked as stale because
        days-before-stale: 60
        days-before-close: 7
```

Exercises

Exercise 1: First Contribution

1. Find an open source project with "good first issue" labels
2. Set up the development environment
3. Make a small contribution (documentation fix or simple bug fix)
4. Submit a pull request and engage with feedback

Exercise 2: Regular Contribution

1. Choose a project to contribute to regularly
2. Make multiple contributions over time
3. Build relationships with maintainers
4. Take on progressively larger tasks

Exercise 3: Project Maintenance

1. Create your own open source project
2. Set up proper documentation and contribution guidelines
3. Implement automation for issue management
4. Practice managing contributions from others

Exercise 4: Community Building

1. Help other contributors in project discussions
2. Write blog posts about your open source experience
3. Speak at meetups or conferences about open source
4. Mentor new contributors

Summary

Open source contribution is a rewarding way to:

- **Develop skills** through real-world projects
- **Build reputation** in the developer community
- **Network** with developers worldwide
- **Give back** to the software community
- **Learn** from experienced developers

Key skills developed: - Project evaluation and selection - Contribution workflow mastery - Code review and feedback handling - Community communication - Project maintenance and leadership

Whether contributing to existing projects or maintaining your own, open source participation is essential for professional growth and community building in software development.

The next chapter will explore team collaboration best practices, building upon open source principles to create effective development teams.

Team Collaboration

Chapter 18: Team Collaboration Best Practices

Establishing Team Workflows

Choosing the Right Branching Strategy

Team Size Considerations

Small Teams (2-5 developers)

```
# GitHub Flow - Simple and effective
# 1. Create feature branch from main
git checkout main
git pull origin main
git checkout -b feature/user-authentication

# 2. Develop and push regularly
git add .
git commit -m "Add login form"
git push -u origin feature/user-authentication

# 3. Create pull request when ready
# 4. Review, approve, and merge
# 5. Delete feature branch
```

Medium Teams (5-15 developers)

```
# Git Flow - More structured approach
# Main branches: main (production), develop (integration)

# Feature development
git checkout develop
git pull origin develop
git checkout -b feature/payment-integration

# After feature completion
git checkout develop
git merge --no-ff feature/payment-integration
git push origin develop
git branch -d feature/payment-integration
```

```
# Release process
git checkout -b release/v1.2.0 develop
# Prepare release, fix bugs
git checkout main
git merge --no-ff release/v1.2.0
git tag -a v1.2.0 -m "Release version 1.2.0"
git checkout develop
git merge --no-ff release/v1.2.0
```

Large Teams (15+ developers)

```
# GitLab Flow with environment branches
# Branches: main (development), staging, production

# Feature development
git checkout main
git pull origin main
git checkout -b feature/new-dashboard

# After merge to main, promote through environments
git checkout staging
git merge main
git push origin staging

# After testing, promote to production
git checkout production
git merge staging
git push origin production
```

Code Review Processes

Review Assignment Strategies

Round-Robin Assignment

```
# .github/CODEOWNERS
# Global owners
* @team-lead @senior-dev

# Frontend code
/src/frontend/ @frontend-team

# Backend code
/src/backend/ @backend-team
```

```
# Database migrations
/migrations/ @database-team @team-lead

# Documentation
/docs/ @tech-writer @team-lead

# CI/CD configuration
/.github/ @devops-team @team-lead
```

Expertise-Based Assignment

```
// Example: Automated reviewer assignment based on file changes
// .github/workflows/assign-reviewers.yml
name: Assign Reviewers

on:
  pull_request:
    types: [opened]

jobs:
  assign-reviewers:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Assign reviewers based on changed files
        uses: actions/github-script@v6
        with:
          script: |
            const { data: files } = await github.rest.pulls.listFiles({
              owner: context.repo.owner,
              repo: context.repo.repo,
              pull_number: context.issue.number,
            });

            let reviewers = [];

            // Check for frontend changes
            if (files.some(file => file.filename.includes('frontend/'))) {
              reviewers.push('frontend-expert');
            }

            // Check for backend changes
            if (files.some(file => file.filename.includes('backend/'))) {
              reviewers.push('backend-expert');
```

```
            }

            // Check for database changes
            if (files.some(file => file.filename.includes('migrations/'))) {
              reviewers.push('database-expert');
            }

            if (reviewers.length > 0) {
              await github.rest.pulls.requestReviewers({
                owner: context.repo.owner,
                repo: context.repo.repo,
                pull_number: context.issue.number,
                reviewers: reviewers,
              });
            }
```

Review Quality Standards

Comprehensive Review Checklist

```
# Code Review Checklist

## Functionality
- [ ] Code solves the stated problem
- [ ] Edge cases are handled appropriately
- [ ] Error handling is comprehensive
- [ ] Performance implications are considered

## Code Quality
- [ ] Code is readable and well-structured
- [ ] Functions and variables have descriptive names
- [ ] Code follows team style guidelines
- [ ] No code duplication without justification
- [ ] Comments explain "why" not "what"

## Testing
- [ ] New functionality has appropriate tests
- [ ] Tests cover edge cases and error conditions
- [ ] Existing tests still pass
- [ ] Test names are descriptive

## Security
- [ ] No sensitive information in code
- [ ] Input validation is present
- [ ] Authentication/authorization is correct
- [ ] No obvious security vulnerabilities
```

```
## Documentation
- [ ] Public APIs are documented
- [ ] README updated if needed
- [ ] Breaking changes are documented
- [ ] Migration guides provided if necessary
```

Review Response Templates

```
# Positive Feedback Templates

## Approval with praise
"Great work on this feature! The code is clean and well-tested. I particularly like how you

## Constructive suggestions
"This is a solid implementation overall. I have a few suggestions that might improve maintai
1. Consider extracting the validation logic into a separate function
2. The error messages could be more user-friendly
3. Adding a few more test cases for edge conditions would be great

These are minor improvements - the core functionality looks good!"

## Learning opportunity
"Thanks for this contribution! I learned something new from your approach to handling async
```

Branch Protection Rules

Essential Protection Settings

```
# Branch protection configuration (via GitHub API or web interface)
protection_rules:
  main:
    required_status_checks:
      strict: true
      contexts:
        - "ci/tests"
        - "ci/lint"
        - "ci/security-scan"

    required_pull_request_reviews:
      required_approving_review_count: 2
      dismiss_stale_reviews: true
      require_code_owner_reviews: true
      restrict_pushes: true
```

```
    restrictions:
      users: []
      teams: ["maintainers"]

    enforce_admins: false
    allow_force_pushes: false
    allow_deletions: false
```

Advanced Protection Strategies

```
# Pre-receive hook for additional protection
#!/bin/bash
# hooks/pre-receive

while read oldrev newrev refname; do
    branch=$(echo $refname | cut -d/ -f3)

    # Protect main branch
    if [ "$branch" = "main" ]; then
        # Ensure all commits are signed
        for commit in $(git rev-list $oldrev..$newrev); do
            if ! git verify-commit $commit 2>/dev/null; then
                echo "Error: Unsigned commit $commit"
                echo "All commits to main must be signed"
                exit 1
            fi
        done

        # Ensure commits follow conventional format
        for commit in $(git rev-list $oldrev..$newrev); do
            message=$(git log -1 --pretty=%s $commit)
            if ! echo "$message" | grep -qE '^(feat|fix|docs|style|refactor|test|chore)(\(.+
                echo "Error: Invalid commit message format in $commit"
                echo "Use: type(scope): description"
                exit 1
            fi
        done
    fi
done
```

Communication Strategies

Documentation Standards

Team Documentation Structure

```
docs/
  team/
      onboarding.md
      coding-standards.md
      review-process.md
      deployment-guide.md
  architecture/
      overview.md
      database-schema.md
      api-design.md
  processes/
      git-workflow.md
      release-process.md
      incident-response.md
  templates/
       pull-request.md
       issue-report.md
       architecture-decision.md
```

Living Documentation

```
# Architecture Decision Record (ADR) Template
# docs/architecture/adr-001-database-choice.md

# ADR-001: Database Technology Choice

## Status
Accepted

## Context
We need to choose a database technology for our new microservice that will handle user data

## Decision
We will use PostgreSQL as our primary database.

## Consequences

### Positive
- ACID compliance ensures data consistency
```

```
- Rich ecosystem and tooling
- Team has existing expertise
- Good performance for our use case

### Negative
- Additional operational complexity compared to managed solutions
- Need to handle backups and maintenance

## Alternatives Considered
- MongoDB: Rejected due to consistency requirements
- DynamoDB: Rejected due to cost and vendor lock-in concerns

## Implementation Notes
- Use connection pooling with pgbouncer
- Implement read replicas for scaling
- Set up automated backups

## Review Date
2024-06-01
```

Meeting Practices

Effective Stand-ups

```
# Daily Stand-up Format

## Each team member shares:
1. **Yesterday**: What did you accomplish?
2. **Today**: What will you work on?
3. **Blockers**: What's preventing progress?

## Guidelines:
- Keep updates under 2 minutes per person
- Focus on work, not detailed technical discussions
- Identify blockers for follow-up after meeting
- Use visual aids (board, screen share) when helpful

## Example Update:
"Yesterday I completed the user authentication API and started on the password reset feature
```

Code Review Meetings

```
# Weekly Code Review Session

## Purpose:
- Review complex or controversial changes
- Share knowledge across team
- Discuss architectural decisions
- Identify patterns and improvements

## Format:
1. **Preparation** (5 min): Review agenda and PRs
2. **PR Reviews** (30 min): Walk through selected PRs
3. **Discussion** (15 min): Patterns, improvements, questions
4. **Action Items** (5 min): Document follow-ups

## Selection Criteria for PRs:
- Large or complex changes
- New patterns or approaches
- Security-sensitive code
- Performance-critical changes
- Educational value for team
```

Conflict Resolution

Technical Disagreements

```
# Technical Conflict Resolution Process

## Step 1: Direct Discussion
- Schedule focused discussion between disagreeing parties
- Present technical arguments with evidence
- Listen to understand, not to win
- Document different approaches and trade-offs

## Step 2: Team Input
- Present options to broader team
- Gather input from relevant experts
- Consider long-term implications
- Evaluate against project goals

## Step 3: Decision Making
- Technical lead or architect makes final decision
- Document decision and reasoning
- Commit to chosen approach as team
```

```
- Plan review/revision if needed

## Step 4: Follow-up
- Monitor implementation of decision
- Gather feedback on outcomes
- Learn from results for future decisions
```

Process Conflicts

```
# Example: Resolving Git workflow conflicts

# Situation: Team split between Git Flow and GitHub Flow

# Resolution process:
# 1. Document current pain points
echo "Current issues with workflow:" > workflow-analysis.md
echo "- Long-lived feature branches causing conflicts" >> workflow-analysis.md
echo "- Complex release process slowing deployments" >> workflow-analysis.md
echo "- New team members struggling with Git Flow complexity" >> workflow-analysis.md

# 2. Trial period with alternative approach
git checkout -b trial/github-flow-experiment
# Document trial results after 2 weeks

# 3. Team retrospective and decision
# 4. Update team documentation
# 5. Provide training on chosen workflow
```

Quality Assurance

Automated Testing Integration

CI/CD Pipeline for Quality

```
# .github/workflows/quality-assurance.yml
name: Quality Assurance

on:
  pull_request:
    branches: [ main, develop ]
  push:
    branches: [ main, develop ]
```

```yaml
jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node-version: [16, 18, 20]

    steps:
    - uses: actions/checkout@v3

    - name: Setup Node.js ${{ matrix.node-version }}
      uses: actions/setup-node@v3
      with:
        node-version: ${{ matrix.node-version }}
        cache: 'npm'

    - name: Install dependencies
      run: npm ci

    - name: Run linting
      run: npm run lint

    - name: Run type checking
      run: npm run type-check

    - name: Run unit tests
      run: npm run test:unit

    - name: Run integration tests
      run: npm run test:integration

    - name: Generate coverage report
      run: npm run test:coverage

    - name: Upload coverage to Codecov
      uses: codecov/codecov-action@v3

  security:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3

    - name: Run security audit
      run: npm audit --audit-level high

    - name: Run Snyk security scan
      uses: snyk/actions/node@master
```

```yaml
      env:
        SNYK_TOKEN: ${{ secrets.SNYK_TOKEN }}

  performance:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3

    - name: Setup Node.js
      uses: actions/setup-node@v3
      with:
        node-version: '18'
        cache: 'npm'

    - name: Install dependencies
      run: npm ci

    - name: Build application
      run: npm run build

    - name: Run performance tests
      run: npm run test:performance

    - name: Bundle size analysis
      run: npm run analyze:bundle
```

Quality Gates

```javascript
// quality-gates.js - Custom quality gate checks
const fs = require('fs');
const path = require('path');

class QualityGates {
  static async checkCodeCoverage() {
    const coverageFile = path.join(__dirname, 'coverage/coverage-summary.json');

    if (!fs.existsSync(coverageFile)) {
      throw new Error('Coverage report not found');
    }

    const coverage = JSON.parse(fs.readFileSync(coverageFile, 'utf8'));
    const threshold = 80;

    const metrics = ['lines', 'functions', 'branches', 'statements'];
```

```
    for (const metric of metrics) {
      const percentage = coverage.total[metric].pct;
      if (percentage < threshold) {
        throw new Error(`${metric} coverage ${percentage}% below threshold ${threshold}%`);
      }
    }

    console.log(' Code coverage meets requirements');
  }

  static async checkBundleSize() {
    const bundleStatsFile = path.join(__dirname, 'dist/bundle-stats.json');

    if (!fs.existsSync(bundleStatsFile)) {
      throw new Error('Bundle stats not found');
    }

    const stats = JSON.parse(fs.readFileSync(bundleStatsFile, 'utf8'));
    const maxSize = 500 * 1024; // 500KB

    if (stats.size > maxSize) {
      throw new Error(`Bundle size ${stats.size} exceeds maximum ${maxSize}`);
    }

    console.log(' Bundle size within limits');
  }

  static async checkDependencyVulnerabilities() {
    const { execSync } = require('child_process');

    try {
      execSync('npm audit --audit-level high', { stdio: 'pipe' });
      console.log(' No high-severity vulnerabilities found');
    } catch (error) {
      throw new Error('High-severity vulnerabilities detected');
    }
  }
}

// Run quality gates
async function runQualityGates() {
  try {
    await QualityGates.checkCodeCoverage();
    await QualityGates.checkBundleSize();
    await QualityGates.checkDependencyVulnerabilities();
```

```
    console.log('  All quality gates passed!');
    process.exit(0);
  } catch (error) {
    console.error('  Quality gate failed:', error.message);
    process.exit(1);
  }
}

if (require.main === module) {
  runQualityGates();
}

module.exports = QualityGates;
```

Code Style and Standards

Automated Code Formatting

```
// .eslintrc.json
{
  "extends": [
    "eslint:recommended",
    "@typescript-eslint/recommended",
    "prettier"
  ],
  "plugins": ["@typescript-eslint", "import", "jest"],
  "rules": {
    "no-console": "warn",
    "no-debugger": "error",
    "prefer-const": "error",
    "no-var": "error",
    "import/order": ["error", {
      "groups": ["builtin", "external", "internal", "parent", "sibling", "index"],
      "newlines-between": "always"
    }],
    "@typescript-eslint/no-unused-vars": "error",
    "@typescript-eslint/explicit-function-return-type": "warn"
  },
  "env": {
    "node": true,
    "jest": true,
    "es2021": true
  }
}
```

```javascript
// prettier.config.js
module.exports = {
  semi: true,
  trailingComma: 'es5',
  singleQuote: true,
  printWidth: 80,
  tabWidth: 2,
  useTabs: false,
  bracketSpacing: true,
  arrowParens: 'avoid',
  endOfLine: 'lf'
};
```

Pre-commit Quality Checks

```yaml
# .pre-commit-config.yaml
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.4.0
    hooks:
      - id: trailing-whitespace
      - id: end-of-file-fixer
      - id: check-yaml
      - id: check-json
      - id: check-merge-conflict
      - id: check-added-large-files
        args: ['--maxkb=500']

  - repo: https://github.com/pre-commit/mirrors-eslint
    rev: v8.28.0
    hooks:
      - id: eslint
        files: \.(js|ts|jsx|tsx)$
        types: [file]
        additional_dependencies:
          - eslint@8.28.0
          - '@typescript-eslint/eslint-plugin@5.44.0'
          - '@typescript-eslint/parser@5.44.0'

  - repo: https://github.com/pre-commit/mirrors-prettier
    rev: v3.0.0-alpha.4
    hooks:
      - id: prettier
        files: \.(js|ts|jsx|tsx|json|css|md)$
```

```yaml
  - repo: local
    hooks:
      - id: jest-tests
        name: jest-tests
        entry: npm run test:unit
        language: system
        pass_filenames: false
        always_run: true
```

Performance and Scalability

Repository Management at Scale

Large Repository Strategies

```bash
# Git LFS for large files
git lfs install
git lfs track "*.psd"
git lfs track "*.zip"
git lfs track "assets/videos/*"

# Sparse checkout for large repositories
git config core.sparseCheckout true
echo "src/frontend/" > .git/info/sparse-checkout
echo "docs/" >> .git/info/sparse-checkout
git read-tree -m -u HEAD

# Shallow clones for CI/CD
git clone --depth 1 --single-branch --branch main https://github.com/user/repo.git

# Partial clone (Git 2.19+)
git clone --filter=blob:none https://github.com/user/repo.git
```

Monorepo Management

```javascript
// lerna.json - Managing multiple packages
{
  "version": "independent",
  "npmClient": "npm",
  "command": {
    "publish": {
      "conventionalCommits": true,
      "message": "chore(release): publish",
```

```
      "registry": "https://registry.npmjs.org/"
    },
    "bootstrap": {
      "ignore": "component-*",
      "npmClientArgs": ["--no-package-lock"]
    }
  },
  "packages": [
    "packages/*"
  ]
}
```

```
# Monorepo workflow with Lerna
# Bootstrap dependencies
lerna bootstrap

# Run tests across all packages
lerna run test

# Publish changed packages
lerna publish

# Run command in specific package
lerna run build --scope=@myorg/package-name

# Add dependency to specific package
lerna add lodash --scope=@myorg/utils
```

Team Scaling Strategies

Microteam Organization

```
# Team Structure for Large Projects

## Core Teams (2-3 people each)
- **Frontend Team**: UI components, user experience
- **Backend Team**: APIs, business logic, data processing
- **DevOps Team**: Infrastructure, deployment, monitoring
- **QA Team**: Testing, quality assurance, automation

## Cross-functional Responsibilities
- **Tech Leads**: Architecture decisions, code review oversight
- **Product Owner**: Requirements, prioritization, stakeholder communication
- **Scrum Master**: Process facilitation, impediment removal
```

```
## Communication Patterns
- **Daily standups**: Within each team
- **Weekly sync**: Cross-team coordination
- **Monthly architecture review**: Technical alignment
- **Quarterly planning**: Strategic alignment
```

Code Ownership Models

```
# .github/CODEOWNERS - Distributed ownership
# Global fallback
* @tech-leads

# Frontend ownership
/src/frontend/ @frontend-team
/src/components/ @frontend-team
*.css @frontend-team
*.scss @frontend-team

# Backend ownership
/src/backend/ @backend-team
/src/api/ @backend-team
/src/services/ @backend-team

# Infrastructure ownership
/infrastructure/ @devops-team
/docker/ @devops-team
/.github/workflows/ @devops-team
/k8s/ @devops-team

# Database ownership
/migrations/ @backend-team @database-admin
/seeds/ @backend-team @database-admin

# Documentation ownership
/docs/ @tech-writer @tech-leads
README.md @tech-writer @tech-leads

# Security-sensitive files
/src/auth/ @security-team @tech-leads
/src/crypto/ @security-team @tech-leads
```

Exercises

Exercise 1: Workflow Implementation

1. Choose appropriate branching strategy for your team size
2. Set up branch protection rules
3. Create pull request templates
4. Implement automated quality checks

Exercise 2: Code Review Process

1. Establish code review guidelines
2. Set up CODEOWNERS file
3. Create review checklists
4. Practice giving constructive feedback

Exercise 3: Quality Automation

1. Set up comprehensive CI/CD pipeline
2. Implement quality gates
3. Configure automated code formatting
4. Add security scanning

Exercise 4: Team Communication

1. Create team documentation structure
2. Establish meeting cadences
3. Implement conflict resolution process
4. Set up monitoring and alerting

Best Practices Summary

Workflow Management

1. **Choose appropriate complexity** for team size and experience
2. **Automate repetitive tasks** to reduce human error
3. **Protect important branches** with comprehensive rules
4. **Document processes clearly** for team consistency
5. **Regular process retrospectives** for continuous improvement

Code Quality

1. **Consistent code style** through automation
2. **Comprehensive testing** at multiple levels
3. **Security scanning** integrated into workflow
4. **Performance monitoring** for critical paths
5. **Regular dependency updates** for security and features

Communication

1. **Clear documentation** that stays current
2. **Regular synchronization** across team members
3. **Constructive feedback** in code reviews
4. **Conflict resolution** processes for technical disagreements
5. **Knowledge sharing** through reviews and discussions

Scalability

1. **Repository organization** for large codebases
2. **Team structure** that supports growth
3. **Ownership models** that distribute responsibility
4. **Automation** that scales with team size
5. **Monitoring** for process effectiveness

Summary

Effective team collaboration requires:

- **Structured workflows** appropriate for team size and complexity
- **Quality processes** that maintain code standards
- **Clear communication** channels and practices
- **Scalable systems** that grow with the team
- **Continuous improvement** through retrospectives and adaptation

Key skills developed: - Workflow design and implementation - Code review leadership - Quality assurance automation - Team communication facilitation - Conflict resolution and decision making

These practices form the foundation for high-performing development teams that can deliver quality software efficiently while maintaining team satisfaction and growth.

The next chapter will explore Git in enterprise environments, building upon these team collaboration principles for larger organizational contexts.

Enterprise

Chapter 19: Git in Enterprise Environments

Enterprise Git Challenges

Scale and Complexity

Large Repository Management

```
# Enterprise repositories often face:
# - Hundreds of thousands of files
# - Gigabytes of history
# - Thousands of contributors
# - Complex branching strategies
# - Regulatory compliance requirements

# Example: Large enterprise repository statistics
git count-objects -vH
# count 0
# size 0
# in-pack 2847291
# packs 1
# size-pack 2.43 GiB
# prune-packable 0
# garbage 0
# size-garbage 0

# Performance optimization for large repos
git config core.preloadindex true
git config core.fscache true
git config gc.auto 256
```

Multi-Team Coordination

```
# Enterprise team structure example:
## Global Teams (100+ developers)
- **Platform Team**: Core infrastructure and shared libraries
- **Product Teams**: Feature development (5-8 teams)
- **Security Team**: Security reviews and compliance
- **DevOps Team**: CI/CD and infrastructure
```

```
- **QA Team**: Testing and quality assurance

## Coordination Challenges:
- Dependency management across teams
- Release coordination
- Code ownership and reviews
- Consistent standards and practices
- Knowledge sharing and documentation
```

Security and Compliance

Enterprise Security Requirements

```
# Signed commits requirement
git config --global commit.gpgsign true
git config --global user.signingkey YOUR_GPG_KEY_ID

# Verify all commits in a range
git log --show-signature v1.0..v2.0

# Enterprise GPG key management
# 1. Centralized key server
# 2. Key rotation policies
# 3. Revocation procedures
# 4. Backup and recovery
```

Compliance Frameworks

```
# SOX compliance example
compliance_requirements:
  change_management:
    - All changes must be tracked in Git
    - Commits must be linked to approved change requests
    - No direct pushes to production branches
    - All changes must be reviewed and approved

  audit_trail:
    - Complete history of all changes
    - Author identification for all commits
    - Timestamp accuracy and integrity
    - Immutable history (no force pushes)

  access_control:
    - Role-based repository access
```

```
    - Regular access reviews
    - Principle of least privilege
    - Segregation of duties
```

Enterprise Git Hosting Solutions

GitHub Enterprise

GitHub Enterprise Server (GHES)

```
# On-premises GitHub installation
# Features:
# - Complete GitHub functionality on-premises
# - LDAP/SAML integration
# - Advanced security features
# - Compliance reporting
# - High availability configuration

# Example GHES configuration
# /data/user/common/enterprise.ghl
{
  "github_hostname": "github.company.com",
  "auth": {
    "type": "ldap",
    "ldap": {
      "host": "ldap.company.com",
      "port": 636,
      "encryption": "ssl",
      "search_domain": "company.com"
    }
  },
  "security": {
    "two_factor_authentication": "required",
    "signed_commits": "required"
  }
}
```

GitHub Enterprise Cloud

```
# Enterprise cloud configuration
enterprise_settings:
  saml_sso:
    enabled: true
```

```
    identity_provider: "Okta"
    required_for_all_members: true

  ip_allow_list:
    enabled: true
    allowed_ranges:
      - "192.168.1.0/24"
      - "10.0.0.0/8"

  advanced_security:
    secret_scanning: true
    dependency_review: true
    code_scanning: true
    push_protection: true
```

GitLab Enterprise

GitLab Self-Managed

```
# /etc/gitlab/gitlab.rb configuration
external_url 'https://gitlab.company.com'

# LDAP configuration
gitlab_rails['ldap_enabled'] = true
gitlab_rails['ldap_servers'] = {
  'main' => {
    'label' => 'LDAP',
    'host' => 'ldap.company.com',
    'port' => 636,
    'uid' => 'sAMAccountName',
    'encryption' => 'simple_tls',
    'base' => 'dc=company,dc=com'
  }
}

# Security settings
gitlab_rails['omniauth_block_auto_created_users'] = false
gitlab_rails['omniauth_allow_single_sign_on'] = ['saml']
gitlab_rails['omniauth_auto_link_ldap_user'] = true

# Backup configuration
gitlab_rails['backup_keep_time'] = 604800
gitlab_rails['backup_path'] = "/var/opt/gitlab/backups"
```

Azure DevOps

Azure DevOps Server

```
{
  "version": "2019.1",
  "features": {
    "active_directory_integration": true,
    "branch_policies": true,
    "pull_request_workflows": true,
    "build_automation": true,
    "release_management": true
  },
  "security": {
    "authentication": "Windows Authentication",
    "authorization": "TFS Groups",
    "ssl_required": true
  }
}
```

Large Repository Strategies

Monorepo vs Multi-repo

Monorepo Advantages

```
# Single repository for entire organization
company-monorepo/
   services/
       user-service/
       payment-service/
       notification-service/
   libraries/
       shared-utils/
       ui-components/
       api-client/
   tools/
       build-scripts/
       deployment/
   docs/
        architecture/
        processes/

# Benefits:
```

```
# - Atomic changes across services
# - Simplified dependency management
# - Consistent tooling and standards
# - Easy code sharing and refactoring
```

Monorepo Challenges and Solutions

```
# Challenge: Large repository performance
# Solution: Sparse checkout and partial clone
git config core.sparseCheckout true
echo "services/user-service/" > .git/info/sparse-checkout
echo "libraries/shared-utils/" >> .git/info/sparse-checkout
git read-tree -m -u HEAD

# Challenge: Build performance
# Solution: Incremental builds with tools like Bazel
# BUILD file example
load("@rules_nodejs//nodejs:defs.bzl", "nodejs_binary")

nodejs_binary(
    name = "user-service",
    data = [
        "//libraries/shared-utils",
        "@npm//express",
    ],
    entry_point = "src/main.js",
)

# Challenge: CI/CD complexity
# Solution: Affected project detection
#!/bin/bash
# detect-changes.sh
CHANGED_FILES=$(git diff --name-only HEAD~1 HEAD)
AFFECTED_SERVICES=""

if echo "$CHANGED_FILES" | grep -q "services/user-service/"; then
    AFFECTED_SERVICES="$AFFECTED_SERVICES user-service"
fi

if echo "$CHANGED_FILES" | grep -q "libraries/shared-utils/"; then
    # Shared library changed, rebuild all services
    AFFECTED_SERVICES="user-service payment-service notification-service"
fi

echo "Affected services: $AFFECTED_SERVICES"
```

Git LFS for Large Files

Enterprise LFS Configuration

```
# Install Git LFS
git lfs install

# Configure LFS server (enterprise)
git config lfs.url https://lfs.company.com/repo.git/info/lfs

# Track large file types
git lfs track "*.psd"
git lfs track "*.zip"
git lfs track "*.dmg"
git lfs track "design-assets/**"

# LFS storage quotas and policies
git lfs env
# Endpoint=https://lfs.company.com/repo.git/info/lfs (auth=basic)
# LocalWorkingDir=/path/to/repo
# LocalGitDir=/path/to/repo/.git
# LocalGitStorageDir=/path/to/repo/.git/lfs
# LocalMediaDir=/path/to/repo/.git/lfs/objects
# TempDir=/path/to/repo/.git/lfs/tmp
# ConcurrentTransfers=3
# TusTransfers=false
# BasicTransfersOnly=false
```

LFS Migration Strategy

```
# Migrate existing large files to LFS
git lfs migrate import --include="*.zip,*.dmg,*.psd" --everything

# Verify migration
git lfs ls-files

# Clean up old objects
git reflog expire --expire-unreachable=now --all
git gc --prune=now --aggressive

# Force push migrated history (coordinate with team)
git push --force-with-lease --all origin
git push --force-with-lease --tags origin
```

Enterprise Workflows

Release Management

Enterprise Release Process

```yaml
# .github/workflows/enterprise-release.yml
name: Enterprise Release Process

on:
  push:
    tags:
      - 'v*'

jobs:
  security-scan:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Security vulnerability scan
        uses: securecodewarrior/github-action-add-sarif@v1
        with:
          sarif-file: security-scan-results.sarif

      - name: License compliance check
        run: |
          npm install -g license-checker
          license-checker --onlyAllow 'MIT;Apache-2.0;BSD-3-Clause'

  compliance-check:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: SOX compliance verification
        run: |
          # Verify all commits are signed
          git log --pretty="format:%H %G?" v1.0..HEAD | grep -v " G" && exit 1 || true

          # Verify change request links
          git log --pretty="format:%s" v1.0..HEAD | grep -E "CR-[0-9]+" || exit 1

  build-and-test:
    runs-on: ubuntu-latest
    needs: [security-scan, compliance-check]
```

```
    steps:
      - uses: actions/checkout@v3

      - name: Build application
        run: npm run build:production

      - name: Run comprehensive tests
        run: |
          npm run test:unit
          npm run test:integration
          npm run test:e2e
          npm run test:performance

  deploy-staging:
    runs-on: ubuntu-latest
    needs: build-and-test
    environment: staging
    steps:
      - name: Deploy to staging
        run: ./deploy.sh staging

      - name: Run smoke tests
        run: npm run test:smoke staging

  approval-gate:
    runs-on: ubuntu-latest
    needs: deploy-staging
    environment: production-approval
    steps:
      - name: Wait for approval
        run: echo "Waiting for production deployment approval"

  deploy-production:
    runs-on: ubuntu-latest
    needs: approval-gate
    environment: production
    steps:
      - name: Deploy to production
        run: ./deploy.sh production

      - name: Verify deployment
        run: npm run test:smoke production

      - name: Update change management system
        run: |
          curl -X POST "$CHANGE_MGMT_API/deployments" \
            -H "Authorization: Bearer $API_TOKEN" \
```

```
            -d "{\"version\": \"$GITHUB_REF\", \"status\": \"deployed\"}"
```

Multi-Environment Management

Environment-Specific Branches

```
# Environment branch strategy
# main -> develop -> staging -> production

# Feature development
git checkout develop
git checkout -b feature/new-payment-method

# After feature completion
git checkout develop
git merge --no-ff feature/new-payment-method

# Promote to staging
git checkout staging
git merge develop
git push origin staging

# After staging validation
git checkout production
git merge staging
git tag -a v1.2.0 -m "Release 1.2.0"
git push origin production --tags
```

Configuration Management

```
// config/environments.js
const environments = {
  development: {
    database: {
      host: 'localhost',
      port: 5432,
      name: 'app_dev'
    },
    api: {
      baseUrl: 'http://localhost:3000'
    },
    features: {
      debugMode: true,
```

```
      experimentalFeatures: true
    }
  },

  staging: {
    database: {
      host: 'staging-db.company.com',
      port: 5432,
      name: 'app_staging'
    },
    api: {
      baseUrl: 'https://api-staging.company.com'
    },
    features: {
      debugMode: false,
      experimentalFeatures: true
    }
  },

  production: {
    database: {
      host: 'prod-db.company.com',
      port: 5432,
      name: 'app_production'
    },
    api: {
      baseUrl: 'https://api.company.com'
    },
    features: {
      debugMode: false,
      experimentalFeatures: false
    }
  }
};

module.exports = environments[process.env.NODE_ENV || 'development'];
```

Integration with Enterprise Tools

Identity and Access Management

LDAP Integration

```
# Git credential helper for LDAP
git config --global credential.helper manager

# Configure LDAP authentication for Git operations
git config --global http.sslverify true
git config --global http.sslcert /path/to/company-cert.pem

# Example LDAP authentication script
#!/bin/bash
# git-credential-ldap
case "$1" in
get)
    echo "protocol=https"
    echo "host=git.company.com"
    echo "username=$LDAP_USERNAME"
    echo "password=$LDAP_PASSWORD"
    ;;
store)
    # Store credentials securely
    ;;
erase)
    # Clear stored credentials
    ;;
esac
```

SAML/SSO Integration

```
# GitHub Enterprise SAML configuration
saml:
  sso_url: "https://sso.company.com/saml/login"
  certificate: |
    -----BEGIN CERTIFICATE-----
    MIICXjCCAcegAwIBAgIJAKS...
    -----END CERTIFICATE-----

  attribute_mapping:
    username: "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name"
    email: "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress"
    full_name: "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname"
```

Change Management Integration

ServiceNow Integration

```
// servicenow-integration.js
const ServiceNow = require('servicenow-rest-api');

class ChangeManagementIntegration {
  constructor() {
    this.sn = new ServiceNow({
      instance: 'company.service-now.com',
      username: process.env.SN_USERNAME,
      password: process.env.SN_PASSWORD
    });
  }

  async createChangeRequest(pullRequest) {
    const changeRequest = {
      short_description: `Deploy: ${pullRequest.title}`,
      description: pullRequest.body,
      category: 'Software',
      priority: '3',
      risk: '3',
      impact: '3',
      requested_by: pullRequest.author,
      implementation_plan: this.generateImplementationPlan(pullRequest)
    };

    const response = await this.sn.create('change_request', changeRequest);
    return response.number;
  }

  async linkCommitToChange(commitHash, changeNumber) {
    // Link Git commit to ServiceNow change request
    const linkData = {
      change_request: changeNumber,
      git_commit: commitHash,
      repository: process.env.GITHUB_REPOSITORY
    };

    await this.sn.create('u_git_commits', linkData);
  }

  generateImplementationPlan(pullRequest) {
    return `
1. Code Review Completed: ${pullRequest.reviews.length} reviews
```

```
2. Automated Tests Passed: All CI checks green
3. Security Scan: No high-severity issues
4. Deployment Plan: Blue-green deployment to production
5. Rollback Plan: Automated rollback available
6. Monitoring: Application metrics and logs monitored
    `;
  }
}

module.exports = ChangeManagementIntegration;
```

Monitoring and Alerting

Git Operations Monitoring

```
// git-monitoring.js
const prometheus = require('prom-client');

// Metrics collection
const gitOperationsCounter = new prometheus.Counter({
  name: 'git_operations_total',
  help: 'Total number of Git operations',
  labelNames: ['operation', 'repository', 'user', 'status']
});

const gitOperationDuration = new prometheus.Histogram({
  name: 'git_operation_duration_seconds',
  help: 'Duration of Git operations',
  labelNames: ['operation', 'repository'],
  buckets: [0.1, 0.5, 1, 2, 5, 10, 30]
});

class GitMonitoring {
  static recordOperation(operation, repository, user, status, duration) {
    gitOperationsCounter
      .labels(operation, repository, user, status)
      .inc();

    gitOperationDuration
      .labels(operation, repository)
      .observe(duration);
  }

  static async getRepositoryHealth(repository) {
    const metrics = {
```

```
      commitFrequency: await this.getCommitFrequency(repository),
      branchCount: await this.getBranchCount(repository),
      contributorCount: await this.getContributorCount(repository),
      codeChurn: await this.getCodeChurn(repository),
      testCoverage: await this.getTestCoverage(repository)
    };

    return metrics;
  }

  static async alertOnAnomalies(repository, metrics) {
    // Alert on unusual patterns
    if (metrics.commitFrequency < 0.1) { // Less than 1 commit per 10 days
      await this.sendAlert('Low commit frequency', repository, metrics);
    }

    if (metrics.branchCount > 100) {
      await this.sendAlert('Too many branches', repository, metrics);
    }

    if (metrics.testCoverage < 0.8) {
      await this.sendAlert('Low test coverage', repository, metrics);
    }
  }
}

module.exports = GitMonitoring;
```

Performance Optimization

Server-Side Optimization

Git Server Configuration

```
# /etc/gitconfig - Server-wide Git configuration
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
    precomposeunicode = true

[pack]
    threads = 0
```

```
    deltaCacheSize = 2g
    packSizeLimit = 2g
    windowMemory = 1g

[gc]
    auto = 6700
    autopacklimit = 50
    pruneexpire = "2 weeks ago"

[receive]
    fsckObjects = true
    denyNonFastForwards = true
    denyDeletes = true

[transfer]
    fsckObjects = true
```

Repository Maintenance Automation

```
#!/bin/bash
# git-maintenance.sh - Automated repository maintenance

REPO_PATH="$1"
LOG_FILE="/var/log/git-maintenance.log"

log() {
    echo "$(date): $1" >> "$LOG_FILE"
}

cd "$REPO_PATH" || exit 1

log "Starting maintenance for $REPO_PATH"

# Garbage collection
log "Running garbage collection"
git gc --aggressive --prune=now

# Repack repository
log "Repacking repository"
git repack -ad

# Verify repository integrity
log "Verifying repository integrity"
if git fsck --full --strict; then
    log "Repository integrity check passed"
```

```
else
    log "ERROR: Repository integrity check failed"
    # Send alert to operations team
    curl -X POST "$ALERT_WEBHOOK" \
        -d "{\"text\":\"Repository integrity check failed: $REPO_PATH\"}"
fi

# Clean up old reflog entries
log "Cleaning reflog"
git reflog expire --expire=30.days --all

# Update server info for dumb HTTP transport
git update-server-info

log "Maintenance completed for $REPO_PATH"
```

Client-Side Optimization

Developer Workstation Setup

```
# .gitconfig optimizations for enterprise environments
[core]
    preloadindex = true
    fscache = true
    longpaths = true

[status]
    showUntrackedFiles = normal

[diff]
    algorithm = histogram
    compactionHeuristic = true

[merge]
    tool = vscode

[pull]
    rebase = true

[push]
    default = simple
    followTags = true

[credential]
    helper = manager
```

```
[http]
    postBuffer = 524288000
    sslVerify = true

[alias]
    # Performance-optimized aliases
    st = status -sb
    co = checkout
    br = branch
    ci = commit
    unstage = reset HEAD --
    last = log -1 HEAD
    visual = !gitk

    # Enterprise-specific aliases
    sync = !git fetch upstream && git rebase upstream/main
    review = !git push origin HEAD && gh pr create
    deploy = !git tag -a $(date +v%Y%m%d-%H%M%S) -m "Deploy $(date)" && git push --tags
```

Disaster Recovery

Backup Strategies

Comprehensive Backup Solution

```
#!/bin/bash
# enterprise-git-backup.sh

BACKUP_ROOT="/backups/git"
DATE=$(date +%Y%m%d_%H%M%S)
RETENTION_DAYS=90

# Repository list
REPOSITORIES=(
    "https://git.company.com/platform/core.git"
    "https://git.company.com/platform/api.git"
    "https://git.company.com/platform/frontend.git"
)

backup_repository() {
    local repo_url="$1"
    local repo_name=$(basename "$repo_url" .git)
    local backup_path="$BACKUP_ROOT/$repo_name"
```

```
    echo "Backing up $repo_name..."

    # Create backup directory
    mkdir -p "$backup_path"

    # Clone or update mirror
    if [ -d "$backup_path/mirror.git" ]; then
        cd "$backup_path/mirror.git"
        git remote update
    else
        git clone --mirror "$repo_url" "$backup_path/mirror.git"
    fi

    # Create compressed backup
    cd "$backup_path"
    tar -czf "${repo_name}_${DATE}.tar.gz" mirror.git/

    # Verify backup integrity
    if tar -tzf "${repo_name}_${DATE}.tar.gz" > /dev/null; then
        echo "Backup verified: ${repo_name}_${DATE}.tar.gz"
    else
        echo "ERROR: Backup verification failed for $repo_name"
        exit 1
    fi

    # Clean old backups
    find "$backup_path" -name "${repo_name}_*.tar.gz" -mtime +$RETENTION_DAYS -delete
}

# Backup all repositories
for repo in "${REPOSITORIES[@]}"; do
    backup_repository "$repo"
done

# Upload to offsite storage
aws s3 sync "$BACKUP_ROOT" s3://company-git-backups/ --delete

echo "Backup completed successfully"
```

Recovery Procedures

Repository Recovery Process

```
#!/bin/bash
# git-recovery.sh

REPO_NAME="$1"
BACKUP_DATE="$2"
RECOVERY_PATH="/recovery/$REPO_NAME"

if [ -z "$REPO_NAME" ] || [ -z "$BACKUP_DATE" ]; then
    echo "Usage: $0 <repository-name> <backup-date>"
    echo "Example: $0 core-api 20231201_143000"
    exit 1
fi

echo "Starting recovery for $REPO_NAME from backup $BACKUP_DATE"

# Download backup from offsite storage
aws s3 cp "s3://company-git-backups/$REPO_NAME/${REPO_NAME}_${BACKUP_DATE}.tar.gz" /tmp/

# Extract backup
mkdir -p "$RECOVERY_PATH"
cd "$RECOVERY_PATH"
tar -xzf "/tmp/${REPO_NAME}_${BACKUP_DATE}.tar.gz"

# Verify repository integrity
cd mirror.git
if git fsck --full; then
    echo "Repository integrity verified"
else
    echo "ERROR: Repository integrity check failed"
    exit 1
fi

# Convert mirror to normal repository
cd ..
git clone mirror.git/ recovered-repo/
cd recovered-repo

# Verify all branches and tags
git branch -a
git tag -l

echo "Recovery completed. Repository available at: $RECOVERY_PATH/recovered-repo"
```

```
echo "Next steps:"
echo "1. Review recovered repository"
echo "2. Push to new remote if needed"
echo "3. Update team with new repository location"
```

Exercises

Exercise 1: Enterprise Setup

1. Set up enterprise Git hosting solution
2. Configure LDAP/SAML authentication
3. Implement branch protection policies
4. Set up compliance monitoring

Exercise 2: Large Repository Management

1. Implement Git LFS for large files
2. Set up monorepo with build optimization
3. Configure sparse checkout for teams
4. Implement repository maintenance automation

Exercise 3: Integration Development

1. Integrate with change management system
2. Set up monitoring and alerting
3. Implement automated compliance checks
4. Create deployment automation

Exercise 4: Disaster Recovery

1. Implement comprehensive backup strategy
2. Test repository recovery procedures
3. Create disaster recovery documentation
4. Train team on recovery processes

Best Practices Summary

Security and Compliance

1. **Implement signed commits** for audit trails
2. **Regular security scanning** of repositories and dependencies
3. **Access control** with principle of least privilege

4. **Compliance automation** for regulatory requirements
5. **Regular security training** for development teams

Performance and Scale

1. **Repository optimization** for large codebases
2. **Efficient branching strategies** for team coordination
3. **Automated maintenance** to prevent performance degradation
4. **Monitoring and alerting** for proactive issue resolution
5. **Client optimization** for developer productivity

Integration and Automation

1. **Seamless tool integration** with enterprise systems
2. **Automated workflows** for consistency and efficiency
3. **Comprehensive monitoring** of Git operations
4. **Change management integration** for compliance
5. **Disaster recovery planning** for business continuity

Summary

Enterprise Git environments require:

- **Scalable infrastructure** to handle large teams and repositories
- **Security and compliance** measures for regulatory requirements
- **Integration capabilities** with enterprise tools and systems
- **Performance optimization** for developer productivity
- **Disaster recovery** planning for business continuity

Key skills developed: - Enterprise Git hosting configuration - Large-scale repository management - Security and compliance implementation - Enterprise tool integration - Performance optimization and monitoring - Disaster recovery planning and execution

These enterprise practices ensure Git can scale to support large organizations while maintaining security, compliance, and performance standards required in professional environments.

This comprehensive coverage of Git and GitHub, from fundamentals to enterprise implementation, provides the knowledge needed to effectively use version control in any development context.

Advanced Topics

Chapter 20: Advanced Topics and Future

Git Performance Optimization

Repository Size Management

Large repositories can impact performance. Here are strategies to optimize:

Shallow Clones

```
# Clone with limited history
git clone --depth 1 https://github.com/user/repo.git

# Deepen shallow clone
git fetch --unshallow

# Shallow clone specific branch
git clone --depth 1 --branch main https://github.com/user/repo.git
```

Sparse Checkout

```
# Enable sparse checkout
git config core.sparseCheckout true

# Define which directories to include
echo "src/" > .git/info/sparse-checkout
echo "docs/" >> .git/info/sparse-checkout

# Apply sparse checkout
git read-tree -m -u HEAD
```

Git LFS (Large File Storage)

```
# Install Git LFS
git lfs install

# Track large files
```

```
git lfs track "*.psd"
git lfs track "*.zip"
git lfs track "videos/*"

# Add .gitattributes
git add .gitattributes

# Large files are now stored in LFS
git add large-file.zip
git commit -m "Add large file via LFS"
```

Repository Maintenance

Garbage Collection

```
# Manual garbage collection
git gc

# Aggressive garbage collection
git gc --aggressive

# Prune unreachable objects
git prune

# Check repository size
git count-objects -vH
```

Pack File Optimization

```
# Repack repository
git repack -ad

# Repack with delta compression
git repack -a -d --depth=50 --window=50

# Verify pack integrity
git verify-pack -v .git/objects/pack/pack-*.idx
```

Custom Git Commands

Creating Git Aliases

Simple Aliases

```
# Shorthand commands
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.ci commit
git config --global alias.st status

# Complex aliases
git config --global alias.unstage 'reset HEAD --'
git config --global alias.last 'log -1 HEAD'
git config --global alias.visual '!gitk'
```

Advanced Aliases

```
# Pretty log format
git config --global alias.lg "log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow

# Show branches with last commit
git config --global alias.br-last "for-each-ref --sort=-committerdate refs/heads/ --format='

# Find commits by message
git config --global alias.find "log --all --full-history -- "
```

Custom Git Scripts

Git Cleanup Script

```
#!/bin/bash
# ~/.local/bin/git-cleanup

# Delete merged branches
git branch --merged | grep -v "\*\|main\|develop" | xargs -n 1 git branch -d

# Prune remote tracking branches
git remote prune origin

# Garbage collect
git gc --prune=now
```

```
echo "Repository cleanup complete!"
```

Git Release Script

```
#!/bin/bash
# ~/.local/bin/git-release

VERSION=$1
if [ -z "$VERSION" ]; then
    echo "Usage: git release <version>"
    exit 1
fi

# Create release branch
git checkout -b release/$VERSION

# Update version file
echo $VERSION > VERSION
git add VERSION
git commit -m "Bump version to $VERSION"

# Merge to main
git checkout main
git merge --no-ff release/$VERSION

# Create tag
git tag -a v$VERSION -m "Release version $VERSION"

# Merge back to develop
git checkout develop
git merge --no-ff release/$VERSION

# Clean up
git branch -d release/$VERSION

echo "Release $VERSION created successfully!"
```

Git Internals Deep Dive

Object Database Exploration

Understanding Object Types

```
# Find all objects
find .git/objects -type f | head -10

# Examine object types
for obj in $(git rev-list --objects --all | cut -d' ' -f1); do
    echo "$obj: $(git cat-file -t $obj)"
done | head -20

# Find largest objects
git rev-list --objects --all | \
git cat-file --batch-check='%(objecttype) %(objectname) %(objectsize) %(rest)' | \
grep '^blob' | sort -k3nr | head -10
```

Pack File Analysis

```
# Analyze pack files
git verify-pack -v .git/objects/pack/pack-*.idx | \
sort -k3nr | head -20

# Find what's taking up space
git rev-list --objects --all | \
git cat-file --batch-check='%(objecttype) %(objectname) %(objectsize) %(rest)' | \
awk '/^blob/ {print substr($0,6)}' | sort -k2nr | head -20
```

Custom Merge Drivers

Creating Custom Merge Driver

```
# Configure custom merge driver
git config merge.ours.driver true
git config merge.ours.name "always use our version"

# In .gitattributes
echo "*.generated merge=ours" >> .gitattributes
```

Database Schema Merge Driver

```
#!/bin/bash
# Custom merge driver for database schemas

BASE=$1
LOCAL=$2
REMOTE=$3

# Custom logic to merge database schemas
# This is a simplified example
if [ -f "$LOCAL" ] && [ -f "$REMOTE" ]; then
    # Merge schemas intelligently
    python merge_schemas.py "$BASE" "$LOCAL" "$REMOTE" > "$LOCAL"
fi

exit 0
```

Alternative Git Interfaces

GUI Applications

GitKraken

- Professional Git GUI
- Visual commit history
- Merge conflict resolution
- Integration with GitHub/GitLab

Sourcetree

- Free Git GUI by Atlassian
- Visual branching and merging
- Built-in Git Flow support
- Cross-platform

GitHub Desktop

- Simple, user-friendly interface
- Seamless GitHub integration
- Visual diff and merge tools
- Beginner-friendly

IDE Integration

VS Code Git Integration

```
// settings.json
{
    "git.enableSmartCommit": true,
    "git.confirmSync": false,
    "git.autofetch": true,
    "git.showPushSuccessNotification": true,
    "gitlens.hovers.currentLine.over": "line",
    "gitlens.currentLine.enabled": true
}
```

JetBrains IDEs

- Built-in Git support
- Visual merge tools
- Branch management
- Commit history visualization

Web-based Git

GitPod

```
# .gitpod.yml
tasks:
  - init: npm install
    command: npm start

ports:
  - port: 3000
    onOpen: open-preview

vscode:
  extensions:
    - ms-vscode.vscode-typescript-next
```

GitHub Codespaces

```
// .devcontainer/devcontainer.json
{
    "name": "Node.js",
    "image": "mcr.microsoft.com/vscode/devcontainers/javascript-node:16",
    "features": {
        "ghcr.io/devcontainers/features/git:1": {}
    },
    "customizations": {
        "vscode": {
            "extensions": [
                "ms-vscode.vscode-typescript-next"
            ]
        }
    },
    "postCreateCommand": "npm install"
}
```

Future of Version Control

Git Evolution

Performance Improvements

- Partial clone improvements
- Better handling of large repositories
- Faster operations on Windows
- Improved network protocols

New Features

- Better merge algorithms
- Enhanced security features
- Improved user experience
- Better integration with cloud services

Alternative Version Control Systems

Mercurial

```
# Similar distributed model
hg clone https://example.com/repo
hg commit -m "Commit message"
hg push
```

Fossil

```
# Integrated bug tracking and wiki
fossil clone https://example.com/repo.fossil repo
fossil open repo.fossil
fossil commit -m "Commit message"
```

Pijul

```
# Patch-based version control
pijul clone https://example.com/repo
pijul record -m "Commit message"
pijul push
```

Emerging Trends

AI-Assisted Development

- Automated code review
- Intelligent merge conflict resolution
- Predictive branching strategies
- Smart commit message generation

Cloud-Native Git

- Serverless Git operations
- Distributed build systems
- Container-based development
- Microservice repository patterns

Advanced Collaboration Patterns

Monorepo Management

Tools and Strategies

```
# Lerna for JavaScript monorepos
npx lerna init
lerna bootstrap
lerna run test
lerna publish

# Bazel for large-scale builds
bazel build //...
bazel test //...

# Git subtree for monorepo management
git subtree add --prefix=libs/shared https://github.com/user/shared.git main
git subtree pull --prefix=libs/shared https://github.com/user/shared.git main
```

Sparse Checkout for Monorepos

```
# Enable sparse checkout
git config core.sparseCheckout true

# Define team-specific directories
echo "frontend/" > .git/info/sparse-checkout
echo "shared/" >> .git/info/sparse-checkout
echo "docs/" >> .git/info/sparse-checkout

# Apply sparse checkout
git read-tree -m -u HEAD
```

Distributed Development

Multi-Remote Workflows

```
# Multiple upstream repositories
git remote add upstream-a https://github.com/org-a/repo.git
git remote add upstream-b https://github.com/org-b/repo.git

# Sync with multiple upstreams
git fetch upstream-a
```

```
git fetch upstream-b

# Merge changes from different upstreams
git merge upstream-a/main
git merge upstream-b/feature-x
```

Cross-Repository Dependencies

```
# Git submodules for dependencies
git submodule add https://github.com/user/lib.git libs/external
git submodule update --init --recursive

# Git subtree for embedded dependencies
git subtree add --prefix=vendor/lib https://github.com/user/lib.git main --squash
```

Security and Compliance

Advanced Security Features

Signed Commits

```
# Generate GPG key
gpg --gen-key

# Configure Git to use GPG key
git config --global user.signingkey YOUR_KEY_ID
git config --global commit.gpgsign true

# Sign commits
git commit -S -m "Signed commit"

# Verify signatures
git log --show-signature
```

Commit Verification

```
# Verify commit signatures
git verify-commit HEAD

# Show signature information
git log --pretty="format:%h %G? %aN  %s"
```

```
# G = good signature
# B = bad signature
# U = good signature with unknown validity
# X = good signature that has expired
# Y = good signature made by expired key
# R = good signature made by revoked key
# E = signature can't be checked
```

Compliance and Auditing

Audit Trail

```
# Complete repository history
git log --all --full-history --date=iso --pretty=fuller

# File-specific audit trail
git log --follow --patch -- sensitive-file.txt

# Author and committer information
git log --pretty=format:"%h %an %ae %cn %ce %ad %cd %s" --date=iso
```

Compliance Reporting

```
#!/bin/bash
# Generate compliance report

echo "Repository Compliance Report"
echo "Generated: $(date)"
echo "Repository: $(git remote get-url origin)"
echo

echo "Recent Activity:"
git log --since="30 days ago" --pretty=format:"%ad %an: %s" --date=short

echo -e "\n\nBranch Protection Status:"
# This would typically query your Git hosting platform's API

echo -e "\n\nSigned Commits:"
git log --show-signature --since="30 days ago" | grep -c "Good signature"
```

Performance Monitoring

Repository Health Metrics

Size and Performance Monitoring

```
#!/bin/bash
# Repository health check

echo "Repository Health Report"
echo "========================"

echo "Repository size:"
du -sh .git

echo -e "\nObject count:"
git count-objects -v

echo -e "\nLargest files:"
git rev-list --objects --all | \
git cat-file --batch-check='%(objecttype) %(objectname) %(objectsize) %(rest)' | \
grep '^blob' | sort -k3nr | head -10

echo -e "\nBranch count:"
git branch -a | wc -l

echo -e "\nRecent activity:"
git log --oneline --since="7 days ago" | wc -l
```

Performance Benchmarking

```
#!/bin/bash
# Git operation benchmarks

echo "Git Performance Benchmarks"
echo "==========================="

echo "Clone time:"
time git clone --quiet https://github.com/user/repo.git temp-repo
rm -rf temp-repo

echo -e "\nStatus time:"
time git status > /dev/null
```

```
echo -e "\nLog time:"
time git log --oneline -100 > /dev/null

echo -e "\nDiff time:"
time git diff HEAD~10 HEAD > /dev/null
```

Exercises

Exercise 1: Performance Optimization

1. Create a repository with large files and many commits
2. Implement Git LFS for large files
3. Use sparse checkout to optimize working directory
4. Measure performance improvements

Exercise 2: Custom Git Commands

1. Create custom Git aliases for your common workflows
2. Write a Git script for automated releases
3. Implement a custom merge driver
4. Test your custom commands in different scenarios

Exercise 3: Advanced Collaboration

1. Set up a monorepo with multiple projects
2. Implement cross-repository dependencies
3. Create automated compliance reporting
4. Test distributed development workflows

Exercise 4: Security Implementation

1. Set up GPG signing for commits
2. Implement branch protection rules
3. Create audit trails for sensitive files
4. Test security verification processes

Best Practices Summary

Performance

1. **Monitor repository size** regularly
2. **Use Git LFS** for large files

3. **Implement sparse checkout** for large repositories
4. **Regular maintenance** with garbage collection
5. **Optimize pack files** for better performance

Security

1. **Sign commits** for authenticity
2. **Protect sensitive branches** with rules
3. **Regular security audits** of repository access
4. **Implement compliance** reporting
5. **Use secure authentication** methods

Collaboration

1. **Choose appropriate** repository structure
2. **Implement consistent** workflows
3. **Automate repetitive** tasks
4. **Monitor team** productivity
5. **Continuous improvement** of processes

Summary

This comprehensive guide has covered Git and GitHub from fundamentals to advanced topics:

Core Concepts Mastered

- **Version control principles** and Git's distributed model
- **Repository management** and file tracking
- **Branching and merging** strategies
- **Remote collaboration** workflows
- **GitHub platform** features and tools

Advanced Skills Developed

- **History rewriting** and repository maintenance
- **Custom workflows** and automation
- **Performance optimization** techniques
- **Security implementation** and compliance
- **Enterprise-scale** Git management

Professional Workflows

- **Team collaboration** best practices
- **CI/CD implementation** with GitHub Actions
- **Open source contribution** processes
- **Code review** and quality assurance
- **Project management** integration

Future Readiness

- **Emerging technologies** and trends
- **Alternative tools** and interfaces
- **Scalability considerations** for growing teams
- **Security and compliance** requirements
- **Performance optimization** strategies

Continuing Your Git Journey

Next Steps

1. **Practice regularly**: Use Git daily to build muscle memory
2. **Contribute to open source**: Apply skills in real projects
3. **Stay updated**: Follow Git and GitHub developments
4. **Share knowledge**: Teach others and learn from community
5. **Specialize**: Focus on areas relevant to your work

Resources for Continued Learning

- **Official Git documentation**: git-scm.com
- **GitHub documentation**: docs.github.com
- **Pro Git book**: Available free online
- **Git community**: Forums, Stack Overflow, Reddit
- **Conferences and meetups**: Local and virtual events

Building Expertise

- **Experiment safely**: Use test repositories for learning
- **Read source code**: Study how others use Git
- **Automate workflows**: Create tools for your team
- **Mentor others**: Teaching reinforces learning
- **Stay curious**: Always ask “why” and “how”

Git and GitHub are foundational tools in modern software development. Mastering them opens doors to effective collaboration, professional development practices, and successful project management. The journey from beginner to expert is ongoing, with new features, best practices, and use cases constantly evolving.

Remember: Git is a tool to serve your development process, not the other way around. Use these skills to build better software, collaborate more effectively, and contribute to the global development community.

Happy coding, and may your commits always be meaningful!

Misc

Chapter 21: Miscellaneous Git Tools and Alternatives

This chapter covers additional Git-related tools and modern alternatives that can enhance your version control workflow:

What You'll Learn

- **Gitea**: Self-hosted Git service with web interface
- **GitLab**: Comprehensive DevOps platform with Git hosting
- **Jujutsu (jj)**: Modern version control system built on Git

Chapter Contents

Why These Tools Matter

While Git is the de facto standard for version control, the ecosystem around it continues to evolve. Self-hosted solutions like Gitea and GitLab provide alternatives to GitHub for organizations wanting more control, while tools like Jujutsu represent the next evolution in version control user experience.

Each tool serves different needs: - **Gitea**: Lightweight, fast, and easy to deploy - **GitLab**: Full-featured DevOps platform with CI/CD - **Jujutsu**: Modern UX with powerful conflict resolution

Gitea: Self-Hosted Git Service

Gitea is a lightweight, self-hosted Git service written in Go. It provides a GitHub-like web interface for managing Git repositories, issues, pull requests, and more.

What is Gitea?

Gitea is a community-managed fork of Gogs, designed to be: - **Lightweight**: Minimal resource requirements - **Fast**: Written in Go for performance - **Easy to deploy**: Single binary with minimal dependencies - **Cross-platform**: Runs on Linux, macOS, Windows, and ARM

Installation Methods

Method 1: Binary Installation (Recommended)

Linux/macOS Installation

```
# Download the latest binary
wget -O gitea https://dl.gitea.io/gitea/1.21.3/gitea-1.21.3-linux-amd64
chmod +x gitea

# Create gitea user
sudo adduser --system --shell /bin/bash --gecos 'Git Version Control' --group --disabled-pas

# Create directory structure
sudo mkdir -p /var/lib/gitea/{custom,data,log}
sudo chown -R git:git /var/lib/gitea/
sudo chmod -R 750 /var/lib/gitea/

# Move binary to system location
sudo cp gitea /usr/local/bin/gitea
sudo chown root:root /usr/local/bin/gitea
sudo chmod 755 /usr/local/bin/gitea
```

Create Systemd Service

```
# Create service file
sudo tee /etc/systemd/system/gitea.service > /dev/null <<EOF
[Unit]
Description=Gitea (Git with a cup of tea)
After=syslog.target
After=network.target

[Service]
Type=simple
User=git
Group=git
WorkingDirectory=/var/lib/gitea/
ExecStart=/usr/local/bin/gitea web --config /etc/gitea/app.ini
Restart=always
Environment=USER=git HOME=/home/git GITEA_WORK_DIR=/var/lib/gitea

[Install]
WantedBy=multi-user.target
EOF

# Enable and start service
sudo systemctl daemon-reload
sudo systemctl enable gitea
sudo systemctl start gitea
```

Method 2: Docker Installation

Basic Docker Setup

```
# Create docker-compose.yml
cat > docker-compose.yml <<EOF
version: "3"

networks:
  gitea:
    external: false

services:
  server:
    image: gitea/gitea:1.21.3
    container_name: gitea
    environment:
      - USER_UID=1000
```

```
      - USER_GID=1000
      - GITEA__database__DB_TYPE=mysql
      - GITEA__database__HOST=db:3306
      - GITEA__database__NAME=gitea
      - GITEA__database__USER=gitea
      - GITEA__database__PASSWD=gitea
    restart: always
    networks:
      - gitea
    volumes:
      - ./gitea:/data
      - /etc/timezone:/etc/timezone:ro
      - /etc/localtime:/etc/localtime:ro
    ports:
      - "3000:3000"
      - "222:22"
    depends_on:
      - db

  db:
    image: mysql:8
    restart: always
    environment:
      - MYSQL_ROOT_PASSWORD=gitea
      - MYSQL_USER=gitea
      - MYSQL_PASSWORD=gitea
      - MYSQL_DATABASE=gitea
    networks:
      - gitea
    volumes:
      - ./mysql:/var/lib/mysql
EOF

# Start services
docker-compose up -d
```

Production Docker Setup with Nginx

```
# Extended docker-compose.yml with reverse proxy
cat > docker-compose.yml <<EOF
version: "3"

networks:
  gitea:
    external: false
```

```yaml
services:
  server:
    image: gitea/gitea:1.21.3
    container_name: gitea
    environment:
      - USER_UID=1000
      - USER_GID=1000
      - GITEA__database__DB_TYPE=postgres
      - GITEA__database__HOST=db:5432
      - GITEA__database__NAME=gitea
      - GITEA__database__USER=gitea
      - GITEA__database__PASSWD=gitea
      - GITEA__server__DOMAIN=git.yourdomain.com
      - GITEA__server__SSH_DOMAIN=git.yourdomain.com
      - GITEA__server__ROOT_URL=https://git.yourdomain.com/
    restart: always
    networks:
      - gitea
    volumes:
      - ./gitea:/data
      - /etc/timezone:/etc/timezone:ro
      - /etc/localtime:/etc/localtime:ro
    expose:
      - "3000"
    ports:
      - "222:22"
    depends_on:
      - db

  db:
    image: postgres:14
    restart: always
    environment:
      - POSTGRES_USER=gitea
      - POSTGRES_PASSWORD=gitea
      - POSTGRES_DB=gitea
    networks:
      - gitea
    volumes:
      - ./postgres:/var/lib/postgresql/data

  nginx:
    image: nginx:alpine
    container_name: gitea-nginx
    restart: always
    ports:
```

```
      - "80:80"
      - "443:443"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
      - ./ssl:/etc/nginx/ssl:ro
    networks:
      - gitea
    depends_on:
      - server
EOF
```

Initial Configuration

Web-based Setup

1. **Access Gitea**: Navigate to `http://localhost:3000`
2. **Database Configuration**:
   - Database Type: SQLite3 (for simple setups) or PostgreSQL/MySQL
   - Host: localhost:5432 (for PostgreSQL)
   - Username/Password: As configured
   - Database Name: gitea
3. **General Settings**:
   - Site Title: Your Organization Name
   - Repository Root Path: `/var/lib/gitea/gitea-repositories`
   - Git LFS Root Path: `/var/lib/gitea/data/lfs`
   - Run As Username: `git`
4. **Server and Third-Party Service Settings**:
   - SSH Server Domain: your-domain.com
   - SSH Port: 22 (or 222 if using Docker)
   - HTTP Port: 3000
   - Application URL: `https://git.yourdomain.com/`

Configuration File (app.ini)

```
# /etc/gitea/app.ini
APP_NAME = Your Company Git Service
RUN_MODE = prod

[repository]
ROOT = /var/lib/gitea/gitea-repositories
SCRIPT_TYPE = bash
DETECTED_CHARSETS_ORDER = UTF-8, UTF-16BE, UTF-16LE, UTF-32BE, UTF-32LE, ISO-8859, windows-1
```

```
ANSI_CHARSET =
FORCE_PRIVATE = false
DEFAULT_PRIVATE = last
DEFAULT_PUSH_CREATE_PRIVATE = true
MAX_CREATION_LIMIT = -1
MIRROR_QUEUE_LENGTH = 1000
PULL_REQUEST_QUEUE_LENGTH = 1000
PREFERRED_LICENSES = Apache License 2.0,MIT License
DISABLE_HTTP_GIT = false
ACCESS_CONTROL_ALLOW_ORIGIN =
USE_COMPAT_SSH_URI = false

[server]
APP_DATA_PATH = /var/lib/gitea/data
DOMAIN = git.yourdomain.com
HTTP_PORT = 3000
ROOT_URL = https://git.yourdomain.com/
DISABLE_SSH = false
SSH_PORT = 22
SSH_LISTEN_PORT = 22
SSH_DOMAIN = git.yourdomain.com
OFFLINE_MODE = false
DISABLE_ROUTER_LOG = false
CERT_FILE =
KEY_FILE =
STATIC_ROOT_PATH =
ENABLE_GZIP = false
LANDING_PAGE = home
LFS_START_SERVER = true
LFS_CONTENT_PATH = /var/lib/gitea/data/lfs
LFS_JWT_SECRET =

[database]
PATH = /var/lib/gitea/data/gitea.db
DB_TYPE = sqlite3
HOST =
NAME =
USER =
PASSWD =
LOG_SQL = false
SQLITE_TIMEOUT = 500
ITERATE_BUFFER_SIZE = 50

[security]
INSTALL_LOCK = true
SECRET_KEY =
LOGIN_REMEMBER_DAYS = 7
```

```
COOKIE_USERNAME = gitea_awesome
COOKIE_REMEMBER_NAME = gitea_incredible
REVERSE_PROXY_AUTHENTICATION = false
REVERSE_PROXY_AUTO_REGISTRATION = false
DISABLE_GIT_HOOKS = false
ONLY_ALLOW_PUSH_IF_GITEA_ENVIRONMENT_SET = true
IMPORT_LOCAL_PATHS = false
INTERNAL_TOKEN =
PASSWORD_HASH_ALGO = pbkdf2

[service]
ACTIVE_CODE_LIVE_MINUTES = 180
RESET_PASSWD_CODE_LIVE_MINUTES = 180
REGISTER_EMAIL_CONFIRM = false
DISABLE_REGISTRATION = false
ALLOW_ONLY_EXTERNAL_REGISTRATION = false
REQUIRE_SIGNIN_VIEW = false
ENABLE_NOTIFY_MAIL = false
ENABLE_REVERSE_PROXY_AUTO_REGISTRATION = false
ENABLE_REVERSE_PROXY_EMAIL = false
ENABLE_CAPTCHA = false
DEFAULT_KEEP_EMAIL_PRIVATE = false
DEFAULT_ALLOW_CREATE_ORGANIZATION = true
DEFAULT_ENABLE_TIMETRACKING = true
NO_REPLY_ADDRESS = noreply.git.yourdomain.com

[mailer]
ENABLED = true
HOST = smtp.gmail.com:587
FROM = git@yourdomain.com
USER = git@yourdomain.com
PASSWD = your-app-password
SKIP_VERIFY = false
USE_SENDMAIL = false
SENDMAIL_PATH = sendmail
SENDMAIL_ARGS =

[log]
MODE = file
LEVEL = info
ROOT_PATH = /var/lib/gitea/log
```

Real-World Setup Scenarios

Scenario 1: Small Team Development Server

Requirements: 5-person development team, internal network only

```
#!/bin/bash
# setup-team-gitea.sh

# Install dependencies
sudo apt update
sudo apt install -y git curl wget

# Create gitea user and directories
sudo adduser --system --shell /bin/bash --gecos 'Git Version Control' --group --disabled-pas
sudo mkdir -p /var/lib/gitea/{custom,data,log}
sudo chown -R git:git /var/lib/gitea/
sudo chmod -R 750 /var/lib/gitea/

# Download and install Gitea
GITEA_VERSION="1.21.3"
wget -O gitea https://dl.gitea.io/gitea/${GITEA_VERSION}/gitea-${GITEA_VERSION}-linux-amd64
chmod +x gitea
sudo mv gitea /usr/local/bin/

# Create configuration directory
sudo mkdir -p /etc/gitea
sudo chown root:git /etc/gitea
sudo chmod 770 /etc/gitea

# Create systemd service
sudo tee /etc/systemd/system/gitea.service > /dev/null <<EOF
[Unit]
Description=Gitea (Git with a cup of tea)
After=syslog.target
After=network.target

[Service]
Type=simple
User=git
Group=git
WorkingDirectory=/var/lib/gitea/
ExecStart=/usr/local/bin/gitea web --config /etc/gitea/app.ini
Restart=always
Environment=USER=git HOME=/home/git GITEA_WORK_DIR=/var/lib/gitea

[Install]
```

```
WantedBy=multi-user.target
EOF

# Enable and start
sudo systemctl daemon-reload
sudo systemctl enable gitea
sudo systemctl start gitea

echo "Gitea is now running on http://localhost:3000"
echo "Complete setup through the web interface"
```

Scenario 2: Production Environment with SSL

Requirements: Public-facing, SSL-enabled, with backup strategy

```
#!/bin/bash
# production-gitea-setup.sh

# Install Nginx and Certbot
sudo apt update
sudo apt install -y nginx certbot python3-certbot-nginx

# Configure Nginx
sudo tee /etc/nginx/sites-available/gitea > /dev/null <<EOF
server {
    listen 80;
    server_name git.yourdomain.com;
    return 301 https://\$server_name\$request_uri;
}

server {
    listen 443 ssl http2;
    server_name git.yourdomain.com;

    # SSL configuration will be added by certbot

    client_max_body_size 512M;

    location / {
        proxy_pass http://localhost:3000;
        proxy_set_header Host \$host;
        proxy_set_header X-Real-IP \$remote_addr;
        proxy_set_header X-Forwarded-For \$proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto \$scheme;
    }
}
```

```
EOF

# Enable site
sudo ln -s /etc/nginx/sites-available/gitea /etc/nginx/sites-enabled/
sudo nginx -t
sudo systemctl reload nginx

# Get SSL certificate
sudo certbot --nginx -d git.yourdomain.com

# Setup backup script
sudo tee /usr/local/bin/backup-gitea.sh > /dev/null <<EOF
#!/bin/bash
BACKUP_DIR="/backup/gitea"
DATE=\$(date +%Y%m%d_%H%M%S)

# Create backup directory
mkdir -p \$BACKUP_DIR

# Stop Gitea
systemctl stop gitea

# Backup data
tar -czf \$BACKUP_DIR/gitea-data-\$DATE.tar.gz -C /var/lib/gitea .
cp /etc/gitea/app.ini \$BACKUP_DIR/app.ini-\$DATE

# Backup database (if using external DB)
# mysqldump -u gitea -p gitea > \$BACKUP_DIR/gitea-db-\$DATE.sql

# Start Gitea
systemctl start gitea

# Clean old backups (keep 7 days)
find \$BACKUP_DIR -name "*.tar.gz" -mtime +7 -delete
find \$BACKUP_DIR -name "app.ini-*" -mtime +7 -delete

echo "Backup completed: \$DATE"
EOF

chmod +x /usr/local/bin/backup-gitea.sh

# Add to crontab for daily backups
echo "0 2 * * * /usr/local/bin/backup-gitea.sh" | sudo crontab -
```

Repository Management

Creating Your First Repository

1. **Through Web Interface**:
   - Click "+" → "New Repository"
   - Fill in repository details
   - Choose visibility (public/private)
   - Initialize with README, .gitignore, license
2. **Through Git CLI**:

```
# Create local repository
mkdir my-project
cd my-project
git init
echo "# My Project" > README.md
git add README.md
git commit -m "Initial commit"

# Add Gitea remote
git remote add origin https://git.yourdomain.com/username/my-project.git
git push -u origin main
```

Repository Settings and Features

Branch Protection

```
# Access via: Repository → Settings → Branches
# Configure:
# - Protect main branch
# - Require pull request reviews
# - Dismiss stale reviews
# - Require status checks
# - Restrict pushes
```

Webhooks Configuration

```
# Repository → Settings → Webhooks
# Add webhook for CI/CD integration:
# URL: https://ci.yourdomain.com/gitea-webhook
# Content Type: application/json
# Events: Push, Pull Request, Issues
```

User and Organization Management

Creating Organizations

```
# Through web interface:
# 1. Click "+" → "New Organization"
# 2. Fill organization details
# 3. Set visibility and permissions
# 4. Add members and teams
```

Team Management Example

```
# Create teams for different access levels:
# - Developers: Read/Write access to repositories
# - Maintainers: Admin access to repositories
# - Viewers: Read-only access

# Team permissions:
# - Read: Clone, pull
# - Write: Clone, pull, push
# - Admin: All permissions + settings
```

Integration Examples

CI/CD Integration with Drone

```
# .drone.yml
kind: pipeline
type: docker
name: default

steps:
- name: test
  image: node:16
  commands:
  - npm install
  - npm test

- name: build
  image: node:16
  commands:
  - npm run build
```

```yaml
- name: deploy
  image: plugins/ssh
  settings:
    host: production-server.com
    username: deploy
    key:
      from_secret: ssh_key
    script:
      - cd /var/www/app
      - git pull origin main
      - npm install --production
      - pm2 restart app

trigger:
  branch:
  - main
  event:
  - push
```

Integration with External Authentication

LDAP Configuration

```
[auth.ldap]
ENABLED = true
HOST = ldap.company.com
PORT = 389
SECURITY_PROTOCOL = unencrypted
SKIP_TLS_VERIFY = false
BIND_DN = cn=gitea,ou=service,dc=company,dc=com
BIND_PASSWORD = password
USER_BASE = ou=users,dc=company,dc=com
USER_FILTER = (&(objectClass=person)(uid=%s))
ADMIN_FILTER = (memberOf=cn=gitea-admins,ou=groups,dc=company,dc=com)
USERNAME_ATTRIBUTE = uid
FIRSTNAME_ATTRIBUTE = givenName
SURNAME_ATTRIBUTE = sn
EMAIL_ATTRIBUTE = mail
```

Maintenance and Monitoring

Health Checks

```
#!/bin/bash
# gitea-health-check.sh

# Check if Gitea is running
if ! systemctl is-active --quiet gitea; then
    echo "ERROR: Gitea service is not running"
    systemctl start gitea
fi

# Check disk space
DISK_USAGE=$(df /var/lib/gitea | awk 'NR==2 {print $5}' | sed 's/%//')
if [ $DISK_USAGE -gt 80 ]; then
    echo "WARNING: Disk usage is ${DISK_USAGE}%"
fi

# Check database connectivity
if ! sudo -u git /usr/local/bin/gitea admin auth list > /dev/null 2>&1; then
    echo "ERROR: Database connection failed"
fi

# Check repository integrity
sudo -u git /usr/local/bin/gitea admin regenerate hooks
sudo -u git /usr/local/bin/gitea admin regenerate keys
```

Log Management

```
# Configure log rotation
sudo tee /etc/logrotate.d/gitea > /dev/null <<EOF
/var/lib/gitea/log/gitea.log {
    daily
    missingok
    rotate 52
    compress
    delaycompress
    notifempty
    create 640 git git
    postrotate
        systemctl reload gitea
    endscript
}
EOF
```

Migration Scenarios

Migrating from GitHub

```
# Using Gitea's built-in migration tool
# 1. Go to "+" → "New Migration"
# 2. Select "GitHub"
# 3. Enter GitHub repository URL
# 4. Provide GitHub token for private repos
# 5. Configure migration options:
#    - Migrate issues
#    - Migrate pull requests
#    - Migrate releases
#    - Migrate wiki
```

Bulk Repository Migration Script

```
#!/bin/bash
# bulk-migrate.sh

GITHUB_TOKEN="your-github-token"
GITEA_TOKEN="your-gitea-token"
GITEA_URL="https://git.yourdomain.com"
GITHUB_USER="source-username"

# Get list of repositories
curl -H "Authorization: token $GITHUB_TOKEN" \
     "https://api.github.com/users/$GITHUB_USER/repos?per_page=100" | \
     jq -r '.[].clone_url' > repos.txt

# Migrate each repository
while read -r repo_url; do
    repo_name=$(basename "$repo_url" .git)
    echo "Migrating $repo_name..."

    curl -X POST \
         -H "Authorization: token $GITEA_TOKEN" \
         -H "Content-Type: application/json" \
         -d "{
             \"clone_addr\": \"$repo_url\",
             \"repo_name\": \"$repo_name\",
             \"service\": \"github\",
             \"auth_token\": \"$GITHUB_TOKEN\",
             \"mirror\": false,
             \"private\": false,
```

```
            \"description\": \"Migrated from GitHub\"
        }" \
        "$GITEA_URL/api/v1/repos/migrate"
done < repos.txt
```

Troubleshooting Common Issues

Issue 1: SSH Key Authentication Problems

```
# Check SSH configuration
sudo -u git ssh -T git@localhost -p 22

# Verify SSH key format
ssh-keygen -l -f ~/.ssh/id_rsa.pub

# Check Gitea SSH configuration
sudo -u git /usr/local/bin/gitea admin auth list

# Regenerate SSH keys if needed
sudo -u git /usr/local/bin/gitea admin regenerate keys
```

Issue 2: Database Connection Problems

```
# Check database status
systemctl status postgresql  # or mysql

# Test database connection
sudo -u git /usr/local/bin/gitea admin auth list

# Check database logs
sudo journalctl -u postgresql -f

# Repair database if needed
sudo -u git /usr/local/bin/gitea doctor check --all
```

Issue 3: Performance Issues

```
# Check system resources
htop
df -h
free -h
```

```
# Optimize Gitea configuration
# In app.ini:
[server]
LFS_START_SERVER = false  # if not using LFS
ENABLE_GZIP = true

[database]
MAX_IDLE_CONNS = 30
MAX_OPEN_CONNS = 300
CONN_MAX_LIFETIME = 3s

[indexer]
ISSUE_INDEXER_TYPE = bleve
REPO_INDEXER_ENABLED = true
```

This comprehensive guide covers Gitea installation, configuration, and real-world usage scenarios. The examples provide practical setups for different environments, from small teams to production deployments.

GitLab: Complete DevOps Platform

GitLab is a comprehensive DevOps platform that provides Git repository management, CI/CD pipelines, issue tracking, and much more in a single application.

What is GitLab?

GitLab offers a complete DevOps lifecycle in one platform: - **Source Code Management**: Git repositories with advanced features - **CI/CD**: Built-in continuous integration and deployment - **Issue Tracking**: Project management and bug tracking - **Security**: Built-in security scanning and compliance - **Monitoring**: Application performance monitoring - **Package Registry**: Container and package management

GitLab Editions

- **GitLab CE (Community Edition)**: Free, open-source version
- **GitLab EE (Enterprise Edition)**: Paid version with advanced features
- **GitLab.com**: SaaS offering hosted by GitLab

Installation Methods

Method 1: Package Installation (Recommended)

Ubuntu/Debian Installation

```
# Install dependencies
sudo apt-get update
sudo apt-get install -y curl openssh-server ca-certificates tzdata perl

# Add GitLab package repository
curl https://packages.gitlab.com/install/repositories/gitlab/gitlab-ee/script.deb.sh | sudo

# Install GitLab
sudo EXTERNAL_URL="https://gitlab.yourdomain.com" apt-get install gitlab-ee

# For Community Edition, use:
# sudo EXTERNAL_URL="https://gitlab.yourdomain.com" apt-get install gitlab-ce
```

CentOS/RHEL Installation

```
# Install dependencies
sudo yum install -y curl policycoreutils-python openssh-server perl

# Enable SSH daemon
sudo systemctl enable sshd
sudo systemctl start sshd

# Configure firewall
sudo firewall-cmd --permanent --add-service=http
sudo firewall-cmd --permanent --add-service=https
sudo systemctl reload firewalld

# Add GitLab repository
curl https://packages.gitlab.com/install/repositories/gitlab/gitlab-ee/script.rpm.sh | sudo

# Install GitLab
sudo EXTERNAL_URL="https://gitlab.yourdomain.com" yum install -y gitlab-ee
```

Method 2: Docker Installation

Basic Docker Setup

```
# Create docker-compose.yml
cat > docker-compose.yml <<EOF
version: '3.6'
services:
  gitlab:
    image: gitlab/gitlab-ee:latest
    container_name: gitlab
    restart: always
    hostname: 'gitlab.yourdomain.com'
    environment:
      GITLAB_OMNIBUS_CONFIG: |
        external_url 'https://gitlab.yourdomain.com'
        gitlab_rails['gitlab_shell_ssh_port'] = 2224
        # Email configuration
        gitlab_rails['smtp_enable'] = true
        gitlab_rails['smtp_address'] = "smtp.gmail.com"
        gitlab_rails['smtp_port'] = 587
        gitlab_rails['smtp_user_name'] = "gitlab@yourdomain.com"
        gitlab_rails['smtp_password'] = "your-app-password"
        gitlab_rails['smtp_domain'] = "yourdomain.com"
        gitlab_rails['smtp_authentication'] = "login"
```

```
        gitlab_rails['smtp_enable_starttls_auto'] = true
        gitlab_rails['smtp_tls'] = false
        gitlab_rails['gitlab_email_from'] = 'gitlab@yourdomain.com'
    ports:
      - '80:80'
      - '443:443'
      - '2224:22'
    volumes:
      - '$GITLAB_HOME/config:/etc/gitlab'
      - '$GITLAB_HOME/logs:/var/log/gitlab'
      - '$GITLAB_HOME/data:/var/opt/gitlab'
    shm_size: '256m'
EOF

# Set GitLab home directory
export GITLAB_HOME=/srv/gitlab

# Create directories
sudo mkdir -p $GITLAB_HOME/{config,logs,data}

# Start GitLab
docker-compose up -d
```

Production Docker Setup with External Database

```
# docker-compose.yml for production
cat > docker-compose.yml <<EOF
version: '3.6'
services:
  redis:
    restart: always
    image: redis:6.2-alpine
    command:
    - --loglevel warning
    volumes:
    - redis-data:/var/lib/redis:Z

  postgresql:
    restart: always
    image: postgres:13.6-alpine
    environment:
      - POSTGRES_USER=gitlab
      - POSTGRES_PASSWORD=gitlab
      - POSTGRES_DB=gitlabhq_production
      - POSTGRES_EXTENSION=pg_trgm,btree_gist
```

```
    volumes:
    - postgresql-data:/var/lib/postgresql/data:Z

  gitlab:
    image: gitlab/gitlab-ee:latest
    restart: always
    hostname: 'gitlab.yourdomain.com'
    environment:
      GITLAB_OMNIBUS_CONFIG: |
        external_url 'https://gitlab.yourdomain.com'
        gitlab_rails['db_adapter'] = 'postgresql'
        gitlab_rails['db_encoding'] = 'unicode'
        gitlab_rails['db_host'] = 'postgresql'
        gitlab_rails['db_port'] = '5432'
        gitlab_rails['db_database'] = 'gitlabhq_production'
        gitlab_rails['db_username'] = 'gitlab'
        gitlab_rails['db_password'] = 'gitlab'

        redis['enable'] = false
        gitlab_rails['redis_host'] = 'redis'
        gitlab_rails['redis_port'] = '6379'

        # Disable built-in services
        postgresql['enable'] = false
        redis['enable'] = false

        # Performance tuning
        unicorn['worker_processes'] = 3
        sidekiq['max_concurrency'] = 25

        # Backup configuration
        gitlab_rails['backup_keep_time'] = 604800
        gitlab_rails['backup_path'] = "/var/opt/gitlab/backups"
    ports:
      - '80:80'
      - '443:443'
      - '2224:22'
    volumes:
      - gitlab-config:/etc/gitlab:Z
      - gitlab-logs:/var/log/gitlab:Z
      - gitlab-data:/var/opt/gitlab:Z
    depends_on:
      - redis
      - postgresql

volumes:
  gitlab-config:
```

```
  gitlab-logs:
  gitlab-data:
  postgresql-data:
  redis-data:
EOF
```

Method 3: Kubernetes Installation

```
# gitlab-values.yaml for Helm chart
global:
  hosts:
    domain: yourdomain.com
    https: true
  ingress:
    configureCertmanager: true
    class: nginx

certmanager:
  install: true

nginx-ingress:
  enabled: true

prometheus:
  install: true

gitlab-runner:
  install: true
  runners:
    privileged: true

postgresql:
  install: true
  postgresqlPassword: secure-password

redis:
  install: true

registry:
  enabled: true

minio:
  enabled: true
```

```
# Install using Helm
helm repo add gitlab https://charts.gitlab.io/
helm repo update
helm upgrade --install gitlab gitlab/gitlab \
  --timeout 600s \
  --set global.hosts.domain=yourdomain.com \
  --set global.hosts.externalIP=YOUR_EXTERNAL_IP \
  --set certmanager-issuer.email=admin@yourdomain.com \
  -f gitlab-values.yaml
```

Initial Configuration

First-Time Setup

1. **Access GitLab**: Navigate to your GitLab URL
2. **Set Root Password**: Create password for root user
3. **Sign In**: Use username `root` and your password

Configuration File (gitlab.rb)

```
# /etc/gitlab/gitlab.rb

# External URL
external_url 'https://gitlab.yourdomain.com'

# SSH settings
gitlab_rails['gitlab_shell_ssh_port'] = 22
gitlab_sshd['enable'] = true
gitlab_sshd['port'] = 22

# Email configuration
gitlab_rails['smtp_enable'] = true
gitlab_rails['smtp_address'] = "smtp.gmail.com"
gitlab_rails['smtp_port'] = 587
gitlab_rails['smtp_user_name'] = "gitlab@yourdomain.com"
gitlab_rails['smtp_password'] = "your-app-password"
gitlab_rails['smtp_domain'] = "yourdomain.com"
gitlab_rails['smtp_authentication'] = "login"
gitlab_rails['smtp_enable_starttls_auto'] = true
gitlab_rails['smtp_tls'] = false
gitlab_rails['gitlab_email_from'] = 'gitlab@yourdomain.com'
gitlab_rails['gitlab_email_reply_to'] = 'noreply@yourdomain.com'

# LDAP configuration
```

```
gitlab_rails['ldap_enabled'] = true
gitlab_rails['prevent_ldap_sign_in'] = false
gitlab_rails['ldap_servers'] = YAML.load <<-'EOS'
  main:
    label: 'LDAP'
    host: 'ldap.yourdomain.com'
    port: 389
    uid: 'sAMAccountName'
    bind_dn: 'CN=gitlab,OU=Service Accounts,DC=yourdomain,DC=com'
    password: 'ldap-password'
    encryption: 'plain'
    verify_certificates: true
    smartcard_auth: false
    active_directory: true
    allow_username_or_email_login: false
    lowercase_usernames: false
    block_auto_created_users: false
    base: 'DC=yourdomain,DC=com'
    user_filter: ''
    attributes:
      username: ['uid', 'userid', 'sAMAccountName']
      email: ['mail', 'email', 'userPrincipalName']
      name: 'cn'
      first_name: 'givenName'
      last_name: 'sn'
    group_base: 'OU=Groups,DC=yourdomain,DC=com'
    admin_group: 'GitLab Administrators'
EOS

# Database configuration (external PostgreSQL)
gitlab_rails['db_adapter'] = 'postgresql'
gitlab_rails['db_encoding'] = 'unicode'
gitlab_rails['db_host'] = 'postgres.yourdomain.com'
gitlab_rails['db_port'] = '5432'
gitlab_rails['db_database'] = 'gitlabhq_production'
gitlab_rails['db_username'] = 'gitlab'
gitlab_rails['db_password'] = 'secure-password'

# Redis configuration (external)
gitlab_rails['redis_host'] = 'redis.yourdomain.com'
gitlab_rails['redis_port'] = 6379
gitlab_rails['redis_password'] = 'redis-password'

# Disable built-in services when using external
postgresql['enable'] = false
redis['enable'] = false
```

```
# Performance tuning
unicorn['worker_processes'] = 4
unicorn['worker_timeout'] = 60
sidekiq['max_concurrency'] = 25

# Backup settings
gitlab_rails['backup_keep_time'] = 604800
gitlab_rails['backup_path'] = "/var/opt/gitlab/backups"

# Container Registry
registry_external_url 'https://registry.yourdomain.com'
gitlab_rails['registry_enabled'] = true

# Pages configuration
pages_external_url 'https://pages.yourdomain.com'
gitlab_pages['enable'] = true

# Monitoring
prometheus['enable'] = true
grafana['enable'] = true
alertmanager['enable'] = true

# Security
gitlab_rails['webhook_timeout'] = 10
gitlab_rails['gitlab_default_can_create_group'] = false
gitlab_rails['gitlab_username_changing_enabled'] = false
```

After editing the configuration:

```
sudo gitlab-ctl reconfigure
sudo gitlab-ctl restart
```

Real-World Setup Scenarios

Scenario 1: Small Development Team

Requirements: 10-person team, basic CI/CD, issue tracking

```
#!/bin/bash
# small-team-gitlab-setup.sh

# Install GitLab CE
curl https://packages.gitlab.com/install/repositories/gitlab/gitlab-ce/script.deb.sh | sudo
sudo EXTERNAL_URL="http://gitlab.local" apt-get install gitlab-ce
```

```
# Basic configuration
sudo tee -a /etc/gitlab/gitlab.rb <<EOF
# Basic settings for small team
external_url 'http://gitlab.local'

# Reduce resource usage
unicorn['worker_processes'] = 2
sidekiq['max_concurrency'] = 10

# Enable container registry
registry_external_url 'http://gitlab.local:5050'
gitlab_rails['registry_enabled'] = true

# Basic backup
gitlab_rails['backup_keep_time'] = 259200  # 3 days
EOF

sudo gitlab-ctl reconfigure

# Create initial groups and projects
gitlab-rails console -e production <<EOF
# Create development group
group = Group.create!(
  name: 'Development Team',
  path: 'dev-team',
  visibility_level: Gitlab::VisibilityLevel::PRIVATE
)

# Create sample project
project = Projects::CreateService.new(
  User.first,
  name: 'Sample Project',
  namespace: group,
  visibility_level: Gitlab::VisibilityLevel::PRIVATE
).execute

puts "Setup completed!"
EOF
```

Scenario 2: Enterprise Production Environment

Requirements: High availability, external database, LDAP integration, advanced security

```
#!/bin/bash
# enterprise-gitlab-setup.sh
```

```
# Install GitLab EE
curl https://packages.gitlab.com/install/repositories/gitlab/gitlab-ee/script.deb.sh | sudo
sudo EXTERNAL_URL="https://gitlab.company.com" apt-get install gitlab-ee

# Configure for enterprise use
sudo tee /etc/gitlab/gitlab.rb <<EOF
external_url 'https://gitlab.company.com'

# SSL configuration
nginx['redirect_http_to_https'] = true
nginx['ssl_certificate'] = "/etc/gitlab/ssl/gitlab.company.com.crt"
nginx['ssl_certificate_key'] = "/etc/gitlab/ssl/gitlab.company.com.key"

# External PostgreSQL
gitlab_rails['db_adapter'] = 'postgresql'
gitlab_rails['db_encoding'] = 'unicode'
gitlab_rails['db_host'] = 'postgres-primary.company.com'
gitlab_rails['db_port'] = '5432'
gitlab_rails['db_database'] = 'gitlabhq_production'
gitlab_rails['db_username'] = 'gitlab'
gitlab_rails['db_password'] = '$(cat /etc/gitlab/db_password)'
postgresql['enable'] = false

# External Redis
gitlab_rails['redis_host'] = 'redis-primary.company.com'
gitlab_rails['redis_port'] = 6379
gitlab_rails['redis_password'] = '$(cat /etc/gitlab/redis_password)'
redis['enable'] = false

# LDAP integration
gitlab_rails['ldap_enabled'] = true
gitlab_rails['ldap_servers'] = YAML.load <<-'EOS'
  main:
    label: 'Company LDAP'
    host: 'ldap.company.com'
    port: 636
    uid: 'sAMAccountName'
    bind_dn: 'CN=gitlab-service,OU=Service Accounts,DC=company,DC=com'
    password: '$(cat /etc/gitlab/ldap_password)'
    encryption: 'simple_tls'
    verify_certificates: true
    active_directory: true
    base: 'DC=company,DC=com'
    user_filter: '(memberOf=CN=GitLab Users,OU=Groups,DC=company,DC=com)'
    admin_group: 'GitLab Administrators'
EOS
```

```
# Performance optimization
unicorn['worker_processes'] = 8
unicorn['worker_timeout'] = 60
sidekiq['max_concurrency'] = 50

# Security settings
gitlab_rails['webhook_timeout'] = 10
gitlab_rails['gitlab_default_can_create_group'] = false
gitlab_rails['gitlab_username_changing_enabled'] = false
gitlab_rails['password_authentication_enabled_for_web'] = false
gitlab_rails['password_authentication_enabled_for_git'] = false

# Backup configuration
gitlab_rails['backup_keep_time'] = 2592000  # 30 days
gitlab_rails['backup_upload_connection'] = {
  'provider' => 'AWS',
  'region' => 'us-east-1',
  'aws_access_key_id' => '$(cat /etc/gitlab/aws_access_key)',
  'aws_secret_access_key' => '$(cat /etc/gitlab/aws_secret_key)'
}
gitlab_rails['backup_upload_remote_directory'] = 'gitlab-backups'

# Container Registry
registry_external_url 'https://registry.company.com'
gitlab_rails['registry_enabled'] = true

# Pages
pages_external_url 'https://pages.company.com'
gitlab_pages['enable'] = true

# Monitoring
prometheus['enable'] = true
grafana['enable'] = true
EOF

sudo gitlab-ctl reconfigure
```

Scenario 3: Multi-Node High Availability Setup

```
#!/bin/bash
# ha-gitlab-setup.sh

# Node 1: Application Server
cat > /etc/gitlab/gitlab.rb <<EOF
external_url 'https://gitlab.company.com'
```

```
# Disable services that will run elsewhere
postgresql['enable'] = false
redis['enable'] = false
nginx['enable'] = false
prometheus['enable'] = false
grafana['enable'] = false
alertmanager['enable'] = false
gitlab_exporter['enable'] = false

# External services
gitlab_rails['db_host'] = 'postgres-cluster.company.com'
gitlab_rails['redis_host'] = 'redis-cluster.company.com'

# Shared storage (NFS)
gitlab_rails['shared_path'] = '/mnt/gitlab/shared'
git_data_dirs({
  "default" => {
    "path" => "/mnt/gitlab/git-data"
  }
})

# Load balancer
gitlab_rails['trusted_proxies'] = ['10.0.0.0/8', '172.16.0.0/12', '192.168.0.0/16']
EOF

# Node 2: GitLab Pages
cat > /etc/gitlab/gitlab.rb <<EOF
external_url 'https://gitlab.company.com'
pages_external_url 'https://pages.company.com'

# Disable all services except Pages
postgresql['enable'] = false
redis['enable'] = false
nginx['enable'] = false
unicorn['enable'] = false
sidekiq['enable'] = false
gitlab_workhorse['enable'] = false
gitaly['enable'] = false
gitlab_pages['enable'] = true

# Pages configuration
gitlab_pages['external_http'] = ['0.0.0.0:80']
gitlab_pages['external_https'] = ['0.0.0.0:443']
EOF

# Node 3: Container Registry
```

```
cat > /etc/gitlab/gitlab.rb <<EOF
external_url 'https://gitlab.company.com'
registry_external_url 'https://registry.company.com'

# Disable all services except Registry
postgresql['enable'] = false
redis['enable'] = false
nginx['enable'] = false
unicorn['enable'] = false
sidekiq['enable'] = false
gitlab_workhorse['enable'] = false
gitaly['enable'] = false
registry['enable'] = true

# Registry configuration
registry['storage'] = {
  's3' => {
    'accesskey' => 'ACCESS_KEY',
    'secretkey' => 'SECRET_KEY',
    'bucket' => 'registry-bucket',
    'region' => 'us-east-1'
  }
}
EOF
```

CI/CD Pipeline Examples

Basic Pipeline Configuration

```
# .gitlab-ci.yml
stages:
  - test
  - build
  - deploy

variables:
  DOCKER_DRIVER: overlay2
  DOCKER_TLS_CERTDIR: "/certs"

before_script:
  - echo "Starting pipeline for $CI_COMMIT_REF_NAME"

# Test stage
test:unit:
```

```
  stage: test
  image: node:16
  script:
    - npm install
    - npm run test:unit
  coverage: '/Lines\s*:\s*(\d+\.\d+)%/'
  artifacts:
    reports:
      coverage_report:
        coverage_format: cobertura
        path: coverage/cobertura-coverage.xml
    paths:
      - coverage/
    expire_in: 1 week

test:lint:
  stage: test
  image: node:16
  script:
    - npm install
    - npm run lint
  allow_failure: true

# Build stage
build:docker:
  stage: build
  image: docker:20.10.16
  services:
    - docker:20.10.16-dind
  before_script:
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
  script:
    - docker build -t $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA .
    - docker push $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
    - docker tag $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA $CI_REGISTRY_IMAGE:latest
    - docker push $CI_REGISTRY_IMAGE:latest
  only:
    - main
    - develop

# Deploy stages
deploy:staging:
  stage: deploy
  image: alpine:latest
  before_script:
    - apk add --no-cache curl
  script:
```

```
    - curl -X POST "$STAGING_WEBHOOK_URL"
      -H "Content-Type: application/json"
      -d '{"image":"'$CI_REGISTRY_IMAGE:$CI_COMMIT_SHA'"}'
  environment:
    name: staging
    url: https://staging.yourdomain.com
  only:
    - develop

deploy:production:
  stage: deploy
  image: alpine:latest
  before_script:
    - apk add --no-cache curl
  script:
    - curl -X POST "$PRODUCTION_WEBHOOK_URL"
      -H "Content-Type: application/json"
      -d '{"image":"'$CI_REGISTRY_IMAGE:$CI_COMMIT_SHA'"}'
  environment:
    name: production
    url: https://yourdomain.com
  when: manual
  only:
    - main
```

Advanced Pipeline with Security Scanning

```
# .gitlab-ci.yml with security scanning
include:
  - template: Security/SAST.gitlab-ci.yml
  - template: Security/Dependency-Scanning.gitlab-ci.yml
  - template: Security/Container-Scanning.gitlab-ci.yml
  - template: Security/DAST.gitlab-ci.yml

stages:
  - test
  - security
  - build
  - deploy
  - review

variables:
  DOCKER_DRIVER: overlay2
  SAST_EXCLUDED_PATHS: "spec, test, tests, tmp"
  DS_EXCLUDED_PATHS: "spec, test, tests, tmp"
```

```yaml
# Custom security job
security:secrets:
  stage: security
  image: trufflesecurity/trufflehog:latest
  script:
    - trufflehog filesystem . --json > secrets-report.json
  artifacts:
    reports:
      secret_detection: secrets-report.json
    expire_in: 1 week
  allow_failure: true

# Infrastructure as Code scanning
security:terraform:
  stage: security
  image: bridgecrew/checkov:latest
  script:
    - checkov -d . --framework terraform --output json > terraform-security.json
  artifacts:
    reports:
      sast: terraform-security.json
    expire_in: 1 week
  only:
    changes:
      - "**/*.tf"
  allow_failure: true

# Performance testing
performance:
  stage: test
  image: sitespeedio/sitespeed.io:latest
  script:
    - sitespeed.io --budget budget.json https://staging.yourdomain.com
  artifacts:
    paths:
      - sitespeed-result/
    expire_in: 1 week
  only:
    - schedules

# Review app deployment
review:
  stage: review
  script:
    - helm upgrade --install review-$CI_COMMIT_REF_SLUG ./helm-chart
      --set image.tag=$CI_COMMIT_SHA
```

```
      --set ingress.host=review-$CI_COMMIT_REF_SLUG.yourdomain.com
  environment:
    name: review/$CI_COMMIT_REF_NAME
    url: https://review-$CI_COMMIT_REF_SLUG.yourdomain.com
    on_stop: stop_review
  only:
    - merge_requests

stop_review:
  stage: review
  script:
    - helm uninstall review-$CI_COMMIT_REF_SLUG
  environment:
    name: review/$CI_COMMIT_REF_NAME
    action: stop
  when: manual
  only:
    - merge_requests
```

GitLab Runner Configuration

Installing GitLab Runner

```
# Install GitLab Runner
curl -L "https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.deb.sh
sudo apt-get install gitlab-runner

# Register runner
sudo gitlab-runner register \
  --url "https://gitlab.yourdomain.com/" \
  --registration-token "YOUR_REGISTRATION_TOKEN" \
  --description "docker-runner" \
  --tag-list "docker,linux" \
  --executor "docker" \
  --docker-image alpine:latest \
  --docker-privileged=true \
  --docker-volumes="/certs/client"
```

Advanced Runner Configuration

```
# /etc/gitlab-runner/config.toml
concurrent = 4
check_interval = 0

[session_server]
  session_timeout = 1800

[[runners]]
  name = "docker-runner"
  url = "https://gitlab.yourdomain.com/"
  token = "RUNNER_TOKEN"
  executor = "docker"
  [runners.custom_build_dir]
  [runners.cache]
    [runners.cache.s3]
      ServerAddress = "s3.amazonaws.com"
      AccessKey = "ACCESS_KEY"
      SecretKey = "SECRET_KEY"
      BucketName = "gitlab-runner-cache"
      BucketLocation = "us-east-1"
  [runners.docker]
    tls_verify = false
    image = "alpine:latest"
    privileged = true
    disable_entrypoint_overwrite = false
    oom_kill_disable = false
    disable_cache = false
    volumes = ["/certs/client", "/cache"]
    shm_size = 0

[[runners]]
  name = "kubernetes-runner"
  url = "https://gitlab.yourdomain.com/"
  token = "RUNNER_TOKEN"
  executor = "kubernetes"
  [runners.kubernetes]
    host = ""
    namespace = "gitlab-runner"
    privileged = true
    image = "alpine:latest"
    [[runners.kubernetes.volumes.host_path]]
      name = "docker-sock"
      mount_path = "/var/run/docker.sock"
      host_path = "/var/run/docker.sock"
```

Project Management Features

Issue Templates

```
<!-- .gitlab/issue_templates/Bug.md -->
## Bug Report

### Description
A clear and concise description of what the bug is.

### Steps to Reproduce
1. Go to '...'
2. Click on '....'
3. Scroll down to '....'
4. See error

### Expected Behavior
A clear and concise description of what you expected to happen.

### Actual Behavior
A clear and concise description of what actually happened.

### Environment
- OS: [e.g. Ubuntu 20.04]
- Browser: [e.g. Chrome 91]
- Version: [e.g. 1.2.3]

### Additional Context
Add any other context about the problem here.

/label ~bug ~needs-investigation
/assign @maintainer
```

Merge Request Templates

```
<!-- .gitlab/merge_request_templates/Default.md -->
## Description
Brief description of changes

## Changes Made
- [ ] Feature A implemented
- [ ] Bug B fixed
- [ ] Documentation updated
```

```
## Testing
- [ ] Unit tests pass
- [ ] Integration tests pass
- [ ] Manual testing completed

## Checklist
- [ ] Code follows style guidelines
- [ ] Self-review completed
- [ ] Documentation updated
- [ ] No breaking changes (or documented)

## Related Issues
Closes #123
Related to #456

/assign @reviewer
/label ~feature
```

Backup and Maintenance

Automated Backup Script

```
#!/bin/bash
# gitlab-backup.sh

BACKUP_DIR="/backup/gitlab"
DATE=$(date +%Y%m%d_%H%M%S)
RETENTION_DAYS=30

# Create backup directory
mkdir -p $BACKUP_DIR

# Create GitLab backup
gitlab-backup create BACKUP=gitlab_backup_$DATE

# Copy configuration files
cp /etc/gitlab/gitlab.rb $BACKUP_DIR/gitlab.rb.$DATE
cp /etc/gitlab/gitlab-secrets.json $BACKUP_DIR/gitlab-secrets.json.$DATE

# Upload to S3 (optional)
if [ "$UPLOAD_TO_S3" = "true" ]; then
    aws s3 cp /var/opt/gitlab/backups/gitlab_backup_${DATE}_gitlab_backup.tar \
        s3://gitlab-backups/gitlab_backup_${DATE}_gitlab_backup.tar
    aws s3 cp $BACKUP_DIR/gitlab.rb.$DATE s3://gitlab-backups/
```

```
    aws s3 cp $BACKUP_DIR/gitlab-secrets.json.$DATE s3://gitlab-backups/
fi

# Clean old backups
find /var/opt/gitlab/backups -name "*.tar" -mtime +$RETENTION_DAYS -delete
find $BACKUP_DIR -name "gitlab.rb.*" -mtime +$RETENTION_DAYS -delete
find $BACKUP_DIR -name "gitlab-secrets.json.*" -mtime +$RETENTION_DAYS -delete

echo "Backup completed: $DATE"
```

Health Check Script

```
#!/bin/bash
# gitlab-health-check.sh

# Check GitLab status
if ! gitlab-ctl status | grep -q "run:"; then
    echo "ERROR: Some GitLab services are not running"
    gitlab-ctl status
    exit 1
fi

# Check disk space
DISK_USAGE=$(df /var/opt/gitlab | awk 'NR==2 {print $5}' | sed 's/%//')
if [ $DISK_USAGE -gt 80 ]; then
    echo "WARNING: Disk usage is ${DISK_USAGE}%"
fi

# Check database connectivity
if ! gitlab-rails runner "ActiveRecord::Base.connection.execute('SELECT 1')" > /dev/null 2>&
    echo "ERROR: Database connection failed"
    exit 1
fi

# Check Redis connectivity
if ! gitlab-rails runner "Gitlab::Redis::Cache.with { |redis| redis.ping }" > /dev/null 2>&1
    echo "ERROR: Redis connection failed"
    exit 1
fi

# Check repository storage
gitlab-rake gitlab:storage:check_repository_storages

echo "GitLab health check passed"
```

Migration and Integration

Migrating from GitHub

```
#!/bin/bash
# github-to-gitlab-migration.sh

GITHUB_TOKEN="your-github-token"
GITLAB_TOKEN="your-gitlab-token"
GITLAB_URL="https://gitlab.yourdomain.com"
GITHUB_ORG="source-org"
GITLAB_GROUP="target-group"

# Get GitHub repositories
curl -H "Authorization: token $GITHUB_TOKEN" \
     "https://api.github.com/orgs/$GITHUB_ORG/repos?per_page=100" | \
     jq -r '.[].name' > repos.txt

# Migrate each repository
while read -r repo_name; do
    echo "Migrating $repo_name..."

    # Create project in GitLab
    curl -X POST \
         -H "Authorization: Bearer $GITLAB_TOKEN" \
         -H "Content-Type: application/json" \
         -d "{
             \"name\": \"$repo_name\",
             \"namespace_id\": \"$GITLAB_GROUP_ID\",
             \"import_url\": \"https://$GITHUB_TOKEN@github.com/$GITHUB_ORG/$repo_name.git\"
             \"visibility\": \"private\"
         }" \
         "$GITLAB_URL/api/v4/projects"

    # Wait for import to complete
    sleep 10
done < repos.txt

echo "Migration completed"
```

This comprehensive GitLab guide covers installation, configuration, CI/CD setup, and real-world scenarios for different organizational needs.

Jujutsu (jj): Next-Generation Version Control

Jujutsu (jj) is a modern version control system that aims to provide a better user experience than Git while maintaining compatibility with Git repositories.

What is Jujutsu?

Jujutsu is designed to address many of Git's usability issues: - **No staging area**: Direct commit workflow - **Automatic conflict resolution**: Better merge handling - **Immutable history**: Operations create new commits instead of modifying existing ones - **Powerful revsets**: Advanced commit selection syntax - **Git compatibility**: Works with existing Git repositories

Key Concepts

Fundamental Differences from Git

1. **No Index/Staging Area**: Changes are committed directly
2. **Working Copy Commits**: Your working directory is always a commit
3. **Change IDs**: Commits have stable identifiers that survive rebasing
4. **Automatic Rebasing**: Operations automatically maintain clean history

Jujutsu vs Git Terminology

Git	Jujutsu	Description
HEAD	@	Current working copy
Branch	Branch	Named pointer to commit
Commit	Change	A set of changes with stable ID
Rebase	Rebase	Automatic history rewriting
Merge	Merge	Combining multiple parents

Installation

Method 1: Package Managers

macOS (Homebrew)

```
brew install jj
```

Linux (Cargo)

```
# Install Rust if not already installed
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source ~/.cargo/env

# Install jj
cargo install --git https://github.com/martinvonz/jj.git jj-cli
```

Ubuntu/Debian (from source)

```
# Install dependencies
sudo apt update
sudo apt install -y build-essential pkg-config libssl-dev

# Install Rust
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source ~/.cargo/env

# Clone and build jj
git clone https://github.com/martinvonz/jj.git
cd jj
cargo build --release
sudo cp target/release/jj /usr/local/bin/
```

Method 2: Pre-built Binaries

```
# Download latest release
curl -L https://github.com/martinvonz/jj/releases/latest/download/jj-linux-x86_64.tar.gz | t
sudo mv jj /usr/local/bin/
chmod +x /usr/local/bin/jj
```

Initial Setup and Configuration

First-Time Configuration

```
# Set up user information
jj config set --user user.name "Your Name"
jj config set --user user.email "your.email@example.com"

# Configure editor
jj config set --user ui.editor "code --wait"  # VS Code
# or
jj config set --user ui.editor "vim"          # Vim

# Configure diff tool
jj config set --user ui.diff.tool "code --wait --diff"
# or
jj config set --user ui.diff.tool "vimdiff"

# Configure merge tool
jj config set --user ui.merge-tool "code --wait"
```

Configuration File

```
# ~/.jjconfig.toml
[user]
name = "Your Name"
email = "your.email@example.com"

[ui]
editor = "code --wait"
diff-tool = "code --wait --diff"
merge-tool = "code --wait"
pager = "less -FRX"

[colors]
"commit_id prefix" = "blue"
"change_id prefix" = "magenta"
"branch" = "green"
"tag" = "yellow"
"working_copy" = "green bold"

[revsets]
log = "@ | ancestors(remote_branches()..@, 2) | trunk()"

[aliases]
```

```
l = ["log", "-r", "(@ | ancestors(remote_branches()..@, 2) | trunk())"]
ll = ["log"]
st = ["status"]
d = ["diff"]
s = ["show"]
```

Basic Operations

Creating and Initializing Repositories

Initialize New Repository

```
# Create new jj repository
mkdir my-project
cd my-project
jj init

# Initialize with Git backend (recommended)
jj init --git
```

Clone Existing Git Repository

```
# Clone from Git repository
jj git clone https://github.com/user/repo.git
cd repo

# Or clone and convert existing Git repo
git clone https://github.com/user/repo.git
cd repo
jj init --git-repo .
```

Working with Changes

Making Changes

```
# Check status
jj status

# View current changes
jj diff
```

```
# Create a new change (commit)
jj commit -m "Add new feature"

# Amend current change
echo "more content" >> file.txt
jj commit --amend -m "Add new feature with more content"

# Create new change and continue working
jj new
echo "another change" >> file2.txt
jj commit -m "Another feature"
```

Viewing History

```
# Show log (default view)
jj log

# Show detailed log
jj log --verbose

# Show specific range
jj log -r "main..@"

# Show graph
jj log --graph

# Custom log format
jj log --template 'commit_id.short() ++ " " ++ description.first_line() ++ "\n"'
```

Real-World Workflow Examples

Scenario 1: Feature Development

```
# Start new feature
cd my-project
jj new main -m "Start user authentication feature"

# Make changes
echo "class UserAuth:" > auth.py
echo "    def login(self, username, password):" >> auth.py
echo "        pass" >> auth.py

# Commit current state
```

```
jj commit -m "Add UserAuth class skeleton"

# Continue development
echo "    def logout(self):" >> auth.py
echo "        pass" >> auth.py
echo "    def validate_user(self, username):" >> auth.py
echo "        return True" >> auth.py

# Commit additional changes
jj commit -m "Add logout and validation methods"

# Add tests
echo "import unittest" > test_auth.py
echo "from auth import UserAuth" >> test_auth.py
echo "" >> test_auth.py
echo "class TestUserAuth(unittest.TestCase):" >> test_auth.py
echo "    def test_login(self):" >> test_auth.py
echo "        auth = UserAuth()" >> test_auth.py
echo "        # Test implementation" >> test_auth.py

jj commit -m "Add unit tests for UserAuth"

# View the feature branch
jj log -r "main..@"
```

Scenario 2: Bug Fix with History Rewriting

```
# Discover bug in earlier commit
jj log --oneline
# * 3a2b1c9 Add unit tests for UserAuth
# * 2b1c9d8 Add logout and validation methods
# * 1c9d8e7 Add UserAuth class skeleton
# * 9d8e7f6 (main) Initial commit

# Fix bug in the validation method
jj edit 2b1c9d8  # Edit the commit with the bug
echo "    def validate_user(self, username):" >> auth.py
echo "        return username is not None and len(username) > 0" >> auth.py

# Commit the fix
jj commit --amend -m "Add logout and validation methods (fix validation logic)"

# Return to latest change
jj edit @
```

```
# View updated history (automatic rebasing)
jj log -r "main..@"
```

Scenario 3: Collaborative Development

```
# Fetch latest changes
jj git fetch

# Rebase current work on latest main
jj rebase -d main

# Push changes
jj git push --branch feature-auth

# Create pull request (using GitHub CLI)
gh pr create --title "Add user authentication" --body "Implements login, logout, and user va

# After review, squash commits before merge
jj squash -r "main..@" -m "Add complete user authentication system"
```

Advanced Features

Revsets (Revision Selection)

Jujutsu uses a powerful query language for selecting commits:

```
# Current working copy
jj log -r "@"

# All ancestors of current commit
jj log -r "ancestors(@)"

# All descendants of main
jj log -r "descendants(main)"

# Commits between main and current
jj log -r "main..@"

# All branches
jj log -r "branches()"

# Remote branches
jj log -r "remote_branches()"
```

```
# Commits by author
jj log -r 'author("john@example.com")'

# Commits with specific message
jj log -r 'description(regex:"fix|bug")'

# Complex queries
jj log -r "(main | @) & ancestors(remote_branches())"
```

Conflict Resolution

Jujutsu has superior conflict resolution compared to Git:

```
# Create conflicting changes
jj new main -m "Feature A"
echo "feature A" > feature.txt
jj commit

jj new main -m "Feature B"
echo "feature B" > feature.txt
jj commit

# Merge (creates conflict)
jj merge

# View conflict
jj status
jj diff

# Resolve conflict
echo "feature A and B combined" > feature.txt
jj commit -m "Resolve conflict between features"

# Alternative: Use merge tool
jj resolve --tool
```

Working with Multiple Changes

```
# Create multiple parallel changes
jj new main -m "Feature 1"
echo "feature 1" > f1.txt
jj commit

jj new main -m "Feature 2"
```

```
echo "feature 2" > f2.txt
jj commit

jj new main -m "Feature 3"
echo "feature 3" > f3.txt
jj commit

# View all changes
jj log --graph

# Combine specific changes
jj merge f1-commit-id f2-commit-id -m "Combine features 1 and 2"

# Reorder changes
jj rebase -d main -s f3-commit-id
```

Integration with Git Workflows

Git Interoperability

```
# Work with Git remotes
jj git remote add origin https://github.com/user/repo.git
jj git fetch
jj git push

# Import Git branches
jj git import

# Export to Git
jj git export

# Work with Git hooks
# Jj respects Git hooks in .git/hooks/
```

Converting Git Repository

```
# Convert existing Git repo
cd existing-git-repo
jj init --git-repo .

# Import all Git history
jj git import
```

```
# Continue working with jj
jj status
jj log
```

Hybrid Workflow (Git + Jujutsu)

```
# Use jj for local development
jj new main -m "Local feature development"
# ... make changes ...
jj commit -m "Implement feature"

# Export to Git for pushing
jj git export
git push origin feature-branch

# Or push directly with jj
jj git push --branch feature-branch
```

Team Collaboration Scenarios

Scenario 1: Code Review Workflow

```
# Developer A: Create feature
jj new main -m "Add payment processing"
# ... implement feature ...
jj commit -m "Implement payment gateway integration"
jj commit -m "Add payment validation"
jj commit -m "Add payment tests"

# Push for review
jj git push --branch payment-feature

# Developer B: Review and suggest changes
jj git fetch
jj new payment-feature -m "Address review comments"
# ... make changes ...
jj commit -m "Fix validation edge cases"
jj commit -m "Improve error handling"

# Squash before merge
jj squash -r "payment-feature..@" -m "Add payment processing with review fixes"
jj git push --branch payment-feature --force
```

Scenario 2: Hotfix Workflow

```
# Critical bug discovered in production
jj new production -m "Hotfix: Fix critical security vulnerability"

# Quick fix
sed -i 's/vulnerable_function/secure_function/g' security.py
jj commit -m "Replace vulnerable function with secure implementation"

# Test the fix
python -m pytest tests/test_security.py
jj commit --amend -m "Fix critical security vulnerability (tested)"

# Deploy hotfix
jj git push --branch hotfix-security

# Merge to main after deployment
jj rebase -d main
jj git push --branch main
```

Scenario 3: Large Feature with Multiple Developers

```
# Lead developer: Create feature branch
jj new main -m "Start user management system"
jj commit -m "Add user model and basic structure"
jj git push --branch user-management

# Developer 1: Work on authentication
jj git fetch
jj new user-management -m "Implement user authentication"
# ... implement auth ...
jj commit -m "Add login/logout functionality"
jj git push --branch user-auth

# Developer 2: Work on user profiles
jj new user-management -m "Implement user profiles"
# ... implement profiles ...
jj commit -m "Add user profile management"
jj git push --branch user-profiles

# Lead developer: Integrate features
jj git fetch
jj merge user-auth user-profiles -m "Integrate authentication and profiles"

# Resolve any conflicts
```

```
jj status
# ... resolve conflicts if any ...
jj commit -m "Resolve integration conflicts"

# Final integration
jj git push --branch user-management
```

Advanced Configuration and Customization

Custom Templates

```
# ~/.jjconfig.toml
[templates]
log_oneline = '''
change_id.short() ++ " " ++
if(description, description.first_line(), "(no description)") ++
if(branches, " (" ++ branches.join(", ") ++ ")")
'''

log_detailed = '''
"Commit: " ++ commit_id.hex() ++ "\n" ++
"Change: " ++ change_id.hex() ++ "\n" ++
"Author: " ++ author.name() ++ " <" ++ author.email() ++ ">\n" ++
"Date: " ++ author.timestamp() ++ "\n" ++
if(branches, "Branches: " ++ branches.join(", ") ++ "\n") ++
"\n" ++ indent("    ", description) ++ "\n"
'''
```

Custom Commands (Aliases)

```
# ~/.jjconfig.toml
[aliases]
# Short status
s = ["status"]

# Detailed log with graph
lg = ["log", "--graph", "--template", "log_detailed"]

# Show changes in current branch
current = ["log", "-r", "main..@"]

# Quick commit with message
qc = ["commit", "-m"]
```

```
# Amend last commit
amend = ["commit", "--amend"]

# Create new change and commit
nc = ["new", "-m"]

# Show diff for specific change
show = ["diff", "-r"]

# Rebase current branch on main
sync = ["rebase", "-d", "main"]
```

Integration with Development Tools

VS Code Integration

```
// .vscode/settings.json
{
    "jujutsu.enable": true,
    "jujutsu.path": "/usr/local/bin/jj",
    "git.enabled": false,
    "scm.defaultViewMode": "tree"
}
```

Shell Integration

```
# ~/.bashrc or ~/.zshrc

# Jujutsu prompt integration
function jj_prompt() {
    if jj root >/dev/null 2>&1; then
        local branch=$(jj log -r @ --no-graph --template 'branches.join(" ")')
        local change_id=$(jj log -r @ --no-graph --template 'change_id.short()')
        if [ -n "$branch" ]; then
            echo " (jj:$branch)"
        else
            echo " (jj:$change_id)"
        fi
    fi
}

# Add to PS1
PS1='${PS1}$(jj_prompt)'
```

```
# Useful aliases
alias jst='jj status'
alias jl='jj log'
alias jd='jj diff'
alias jc='jj commit'
alias jn='jj new'
```

Performance and Optimization

Repository Maintenance

```
# Check repository health
jj debug check

# Optimize repository
jj debug reindex

# Clean up unreferenced changes
jj debug gc

# Repository statistics
jj debug stats
```

Large Repository Handling

```
# Configure for large repositories
jj config set --repo core.preload-index false
jj config set --repo core.fsmonitor true

# Shallow clone for large repositories
jj git clone --depth 1 https://github.com/large/repo.git

# Sparse checkout
jj sparse set path/to/needed/files
```

Migration Strategies

Gradual Migration from Git

```
# Phase 1: Parallel usage
# Keep using Git for team collaboration
# Use jj for local development

# Phase 2: Team adoption
# Train team on jj basics
# Use jj for feature branches
# Continue using Git for main branch

# Phase 3: Full migration
# Move all development to jj
# Use jj git commands for remote operations
# Eventually move to native jj hosting (when available)
```

Migration Script

```
#!/bin/bash
# migrate-to-jj.sh

REPO_URL=$1
REPO_NAME=$(basename "$REPO_URL" .git)

if [ -z "$REPO_URL" ]; then
    echo "Usage: $0 <git-repo-url>"
    exit 1
fi

echo "Migrating $REPO_URL to Jujutsu..."

# Clone with Git
git clone "$REPO_URL" "${REPO_NAME}-git"
cd "${REPO_NAME}-git"

# Initialize jj in the same directory
jj init --git-repo .

# Import all Git history
jj git import

# Set up remote
jj git remote add origin "$REPO_URL"
```

```
# Create jj-specific configuration
cat > .jjconfig.toml <<EOF
[ui]
default-command = "log"

[revsets]
log = "@ | ancestors(remote_branches()..@, 10) | heads(remote_branches())"

[aliases]
sync = ["git", "fetch", "--all-remotes"]
push = ["git", "push"]
EOF

echo "Migration completed!"
echo "Repository is now ready for jj usage"
echo "Run 'jj status' to get started"
```

Troubleshooting Common Issues

Issue 1: Conflict Resolution

```
# When conflicts occur
jj status  # Shows conflicted files

# View conflict markers
jj diff

# Resolve manually or with tool
jj resolve --tool

# Or edit files manually and commit
vim conflicted_file.txt
jj commit -m "Resolve conflict"
```

Issue 2: Lost Changes

```
# View all changes (including abandoned)
jj log --revisions 'all()'

# Restore abandoned change
jj new <abandoned-change-id>
```

```
# Or use operation log
jj op log
jj op restore <operation-id>
```

Issue 3: Git Synchronization Issues

```
# Force import from Git
jj git import --force

# Reset to match Git state
jj git export
git reset --hard origin/main
jj git import

# Check Git/jj consistency
jj debug check-git
```

This comprehensive guide covers Jujutsu from basic concepts to advanced usage, providing practical examples for teams transitioning from Git to this modern version control system.