# Nova's Git-Survival-Guide

## Das komplette Cheat-Sheet zur Git-Serie

*Von Nova Chen, Junior Entwicklerin bei Java Fleet Systems Consulting*
*Version 4.0 (Final) - Oktober 2024*

---

## 📋 Inhaltsverzeichnis



---

# 1. Git-Grundlagen

### Die drei Git-Bereiche

```
Working Directory → Staging Area → Repository
     (Dateien)    →   (git add)  →  (git commit)
```

**Working Directory:** Dein normaler Projektordner (sichtbar)
**Staging Area:** `.git/index` (unsichtbar, "Wartezone")
**Repository:** `.git/objects/` (unsichtbar, komplette Historie)

### Remote vs. Local

**Local Repository:**

- Auf deinem Computer
- Komplett offline nutzbar
- Nur du siehst deine Commits

**Remote Repository:**

- Auf Server (GitHub/GitLab/Bitbucket)
- Backup + Team-Kollaboration
- Optional! Git funktioniert ohne Remote

### SVN vs. Git

| SVN (Zentralisiert) | Git (Verteilt) |
|---|---|
| Zentraler Server nötig | Komplett lokal möglich |
| Keine lokale Historie | Vollständige lokale Historie |
| "Letzter gewinnt" | Merge oder Conflict |
| Online-Zwang | Offline-fähig |

---

# 2. Die wichtigsten Kommandos

## Repository erstellen/klonen

```
# Neues Repository erstellen
git init

# Existierendes Repository klonen
git clone <url>
```

## Grundlegende Workflow

```
# Status anzeigen (IMMER nutzen!)
git status

# Dateien zur Staging Area hinzufügen
git add <datei>
git add .                    # Alle Änderungen

# Commit erstellen
git commit -m "Deine Nachricht"

# Zum Remote pushen
git push

# Vom Remote pullen
git pull
```

## Änderungen anschauen

```
# Unterschiede anzeigen (Working vs Staging)
git diff

# Unterschiede anzeigen (Staging vs Repository)
git diff --cached

# Commit-Historie
git log
git log --oneline
git log --graph --oneline --all
```

## Undo-Kommandos

```
# Änderung rückgängig (Working Directory)
git checkout -- <datei>

# Aus Staging Area entfernen
git restore --staged <datei>
```

```
# Letzten Commit rückgängig (Änderungen behalten)
git reset --soft HEAD~1

# Letzten Commit komplett löschen (VORSICHT!)
git reset --hard HEAD~1

# Commit rückgängig (sicher für Teams)
git revert <commit-hash>

# Alles wiederherstellen
git reflog                    # Alle Aktionen anzeigen
git checkout <commit-hash>    # Zu bestimmtem Stand zurück
```

---

# 3. Branching & Merging

## Branch-Kommandos

```
# Branch erstellen
git branch <branch-name>

# Branch erstellen UND wechseln
git checkout -b <branch-name>

# Modern (Git 2.23+)
git switch -c <branch-name>

# Branches anzeigen
git branch                    # Lokale Branches
git branch -r                 # Remote Branches
git branch -a                 # Alle Branches

# Zu Branch wechseln
git checkout <branch-name>
git switch <branch-name>

# Branch löschen
git branch -d <branch-name>   # Sicher (nur wenn gemergt)
git branch -D <branch-name>   # Force (auch ungemergt)

# Remote Branch löschen
git push origin --delete <branch-name>
```

## Branch-Naming-Conventions

```
feature/user-authentication   # Neue Features
bugfix/login-timeout-error    # Bug-Fixes
hotfix/security-vulnerability # Dringende Production-Fixes
refactor/extract-user-service # Refactoring
docs/update-readme           # Dokumentation
test/add-integration-tests   # Tests
```

## Merging

```
# Branch mergen
git checkout main
git merge <branch-name>
```

```
# Merge abbrechen (bei Conflicts)
git merge --abort

# Fast-Forward erzwingen
git merge --no-ff <branch-name>
```

### Merge-Conflicts lösen

```
# 1. Status prüfen
git status

# 2. Dateien öffnen und Conflicts lösen
#    Marker entfernen: <<<<<<< ======= >>>>>>>

# 3. Als gelöst markieren
git add <datei>

# 4. Merge abschließen
git commit
```

### Merge vs. Rebase

#### Merge (Sicher):

```
git merge <branch>
# → Erzeugt Merge-Commit
# → Historie bleibt erhalten
# → Team-freundlich
```

#### Rebase (Für lokale Branches):

```
git rebase <branch>
# → Lineare Historie
# → Commits werden umgeschrieben
# → NUR für lokale/eigene Branches!
```

⚠️ **GOLDENE REGEL:** Never rebase public branches!

---

# 4. Pull Requests & Kollaboration

### Pull Request Workflow

```
# 1. Feature-Branch erstellen
git checkout -b feature/awesome-thing

# 2. Entwickeln und committen
git commit -am "feat: Add awesome thing"

# 3. Pushen
git push origin feature/awesome-thing

# 4. Pull Request auf GitLab/GitHub erstellen

# 5. Nach Merge: Aufräumen
git checkout main
git pull
git branch -d feature/awesome-thing
```

### Conventional Commits Format

```
<type>(<scope>): <subject>

<body>

<footer>
```

**Types:**

- `feat:` - Neues Feature
- `fix:` - Bug-Fix
- `docs:` - Dokumentation
- `style:` - Formatting
- `refactor:` - Code-Refactoring
- `test:` - Tests
- `chore:` - Build/Dependencies

**Beispiele:**

```
feat(auth): Add JWT authentication
fix(login): Resolve null pointer exception
docs(readme): Update installation instructions
test(auth): Add unit tests for UserService
```

### Interactive Rebase

```
# Letzten 5 Commits bearbeiten
git rebase -i HEAD~5

# Im Editor:
# pick   = Commit behalten
# reword = Commit-Message ändern
# squash = Mit vorherigem Commit kombinieren (Message behalten)
# fixup  = Mit vorherigem Commit kombinieren (Message wegwerfen)
# drop   = Commit löschen

# Nach Rebase pushen
git push --force-with-lease
```

### Force Push (Sicher!)

```
# ❌ GEFÄHRLICH - überschreibt alles
git push --force

# ✅ SICHER - prüft ob andere gepusht haben
git push --force-with-lease
```

---

# 5. Git-Aliases (Top 20)

### In .gitconfig einfügen:

```
[alias]
    # 1. Status & Basics
    st = status
```

```
    co = checkout
    br = branch
    ci = commit

    # 2. Der legendäre "oops"
    oops = reset --soft HEAD~1

    # 3. Schöne Log-Ansicht
    lg = log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s
%Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit

    # 4. One-line Log
    l = log --oneline

    # 5. Letzten Commit ändern
    amend = commit --amend --no-edit

    # 6. Force push (sicher)
    fpush = push --force-with-lease

    # 7. Branches aufräumen
    clean-branches = "!git branch --merged | grep -v '\\*\\|main\\|develop' |
xargs -n 1 git branch -d"

    # 8. Alle Änderungen sehen
    changes = diff --name-status

    # 9. Uncommit (aber Änderungen behalten)
    uncommit = reset --mixed HEAD~1

    # 10. Aktuellen Branch anzeigen
    current = rev-parse --abbrev-ref HEAD

    # 11. Contributors anzeigen
    contributors = shortlog --summary --numbered

    # 12. Aliases anzeigen
    aliases = config --get-regexp alias

    # 13. Last commit
    last = log -1 HEAD --stat

    # 14. Alle Remotes anzeigen
    remotes = remote -v

    # 15. Unstage all
    unstage = reset HEAD

    # 16. Discard changes
    discard = checkout --

    # 17. Stash mit Message
    stash-save = stash save -u

    # 18. Show stash
    stash-show = stash show -p

    # 19. Branch umbenennen
    rename = branch -m

    # 20. Wer hat diese Zeile geschrieben?
    blame-line = blame -L
```

---

# 6. Git-Hooks Templates

## Pre-Commit Hook

### `.githooks/pre-commit`:

```
#!/bin/sh
echo "🔍 Running pre-commit checks..."

# 1. Tests
echo "→ Running tests..."
mvn test
if [ $? -ne 0 ]; then
    echo "❌ Tests failed!"
    exit 1
fi

# 2. Code Formatting
echo "→ Checking code format..."
mvn spotless:check
if [ $? -ne 0 ]; then
    echo "❌ Code not formatted! Run: mvn spotless:apply"
    exit 1
fi

# 3. No debug statements
echo "→ Checking for debug statements..."
if git diff --cached | grep -E "console\\.log|System\\.out\\.print"; then
    echo "❌ Found debug statements!"
    exit 1
fi

echo "✅ All checks passed!"
```

## Commit-Msg Hook

### `.githooks/commit-msg`:

```
#!/bin/sh
commit_msg_file=$1
commit_msg=$(cat "$commit_msg_file")

# Check Conventional Commits format
if ! echo "$commit_msg" | grep -qE "^(feat|fix|docs|style|refactor|test|chore)(\
(.+\))?: .+"; then
    echo "❌ Invalid commit message format!"
    echo ""
    echo "Format: <type>(<scope>): <subject>"
    echo "Example: feat(auth): Add JWT authentication"
    exit 1
fi

# Check subject length
subject_length=$(echo "$commit_msg" | head -1 | wc -c)
if [ $subject_length -gt 72 ]; then
    echo "❌ Commit subject too long! Max 72 characters."
    exit 1
fi

echo "✅ Commit message OK!"
```

### Pre-Push Hook

#### `.githooks/pre-push`:

```
#!/bin/sh
echo "🚀 Running pre-push checks..."

# Full test suite
echo "→ Running full test suite..."
mvn clean test
if [ $? -ne 0 ]; then
    echo "❌ Tests failed!"
    exit 1
fi

# Security check
echo "→ Running security scan..."
mvn dependency:check
if [ $? -ne 0 ]; then
    echo "⚠️  Security vulnerabilities found!"
fi

echo "✅ All checks passed!"
```

### Hooks aktivieren:

```
# 1. Executable machen
chmod +x .githooks/pre-commit
chmod +x .githooks/commit-msg
chmod +x .githooks/pre-push

# 2. Git konfigurieren
git config core.hooksPath .githooks
```

---

# 7. Troubleshooting

### "Gestern lief es noch!"

```
# Alle Commits anschauen
git log --oneline

# Zu "gestern" zurück
git checkout <commit-hash>

# Oder: Letzten Commit rückgängig
git reset --hard HEAD~1
```

### Merge-Conflict-Panik

```
# 1. Ruhig bleiben!
# 2. Status prüfen
git status

# 3. Conflict-Dateien öffnen
# 4. Marker finden und entfernen: <<<<<<< ======= >>>>>>>
# 5. Code kombinieren/auswählen

# 6. Als gelöst markieren
```

```
git add <datei>

# 7. Merge abschließen
git commit
```

## Branch versehentlich gelöscht

```
# Alle Aktionen anzeigen
git reflog

# Branch wiederherstellen
git checkout -b <branch-name> <commit-hash>
```

## Falscher Branch committed

```
# Commit rückgängig (Änderungen behalten)
git reset --soft HEAD~1

# Zu richtigem Branch wechseln
git checkout <richtiger-branch>

# Erneut committen
git commit -m "Richtige Message"
```

## Sensitive Daten committed

```
# Letzten Commit entfernen (LOKAL!)
git reset --hard HEAD~1

# Falls schon gepusht:
# 1. Datei zu .gitignore hinzufügen
# 2. Aus Git entfernen (bleibt lokal)
git rm --cached <datei>
git commit -m "Remove sensitive file"
git push

# 3. Credentials ändern (wichtig!)
```

## Push rejected

```
# Andere haben gepusht - erst pullen
git pull

# Bei Conflicts: lösen
# Dann pushen
git push
```

## Vim-Hölle

```
# Schnell raus:
# ESC drücken, dann:
:q!

# Dauerhaft VS Code nutzen:
git config --global core.editor "code --wait"
```

---

# 8. .gitconfig Best Practices

## Komplette .gitconfig

```
[user]
    name = Nova Chen
    email = nova.chen@java-developer.online

[core]
    editor = code --wait
    autocrlf = input
    excludesfile = ~/.gitignore_global

[init]
    defaultBranch = main

[push]
    default = current
    autoSetupRemote = true

[pull]
    rebase = true

[fetch]
    prune = true

[merge]
    conflictstyle = diff3

[diff]
    tool = vscode

[difftool "vscode"]
    cmd = code --wait --diff $LOCAL $REMOTE

[merge]
    tool = vscode

[mergetool "vscode"]
    cmd = code --wait $MERGED

[rerere]
    enabled = true

[alias]
    # Siehe Sektion 5 für alle Aliases
    st = status
    co = checkout
    br = branch
    ci = commit
    oops = reset --soft HEAD~1
    # ... (alle anderen)
```

## Global .gitignore

**`~/.gitignore_global`:**

```
# OS
.DS_Store
Thumbs.db

# IDEs
```

```
.idea/
.vscode/
*.swp
*.swo
*~

# Build
target/
build/
dist/
*.class
*.jar
*.war

# Logs
*.log

# Environment
.env
.env.local
```

**Aktivieren:**

```
git config --global core.excludesfile ~/.gitignore_global
```

---

# 🎯 Quick Reference

## Häufigste Kommandos

```
git status              # Status prüfen
git add .               # Alle Änderungen stagen
git commit -m "msg"     # Commit erstellen
git push                # Pushen
git pull                # Pullen
git log --oneline       # Historie anzeigen
git checkout -b name    # Branch erstellen + wechseln
git merge branch        # Branch mergen
```

## Notfall-Kommandos

```
git reflog              # Alle Aktionen anzeigen
git reset --soft HEAD~1 # Letzten Commit rückgängig
git merge --abort       # Merge abbrechen
git clean -fd           # Untracked files löschen
```

## Wichtige Konzepte

- **"Gestern lief es noch"** → `git checkout <hash>`
- **main ist heilig** → Niemals direkt auf main entwickeln!
- **Branches sind Schutz** → Danke Linus!
- **Historie ist editierbar** → Aber nur vor dem Push!
- **git status ist dein Freund** → Nutze es ständig!

---

## 📚 Weitere Resourcen

**Offizielle Dokumentation:**

- https://git-scm.com/doc

**Interaktive Tutorials:**

- https://learngitbranching.js.org/

**Bücher:**

- "Pro Git" (kostenlos online)

**Nova's Blog-Serie:**

- Teil 1: Git-Grundlagen
- Teil 2: Branching & Merging
- Teil 3: Pull Requests & Kollaboration
- Teil 4: Automation & Workflows

---

## 💡 Nova's Wichtigste Lektionen

1. **Git ist Logik, keine Magie**
2. **"Gestern lief es noch" ist keine Ausrede - es ist ein Git-Kommando**
3. **Branches schützen vor "Blödmännern"** (Linus' Original-Motivation!)
4. **Historie ist editierbar bis zum Push**
5. **Automation spart Nerven** (Hooks!)
6. **CLI + IDE = Perfect Team**
7. **git status ist dein bester Freund**
8. **Never rebase public branches**

---

## 🎓 Von Git-Noob zu Git-Confident

**Woche 1:** "Was ist ein Repository?" → Git-Grundlagen verstanden
**Woche 2:** Branch-Chaos → Merge-Confidence
**Woche 3:** 47 Commits → 3 saubere Commits
**Woche 4:** Git-Panik → Git-Produktivität

**Transformation komplett!** 🚀

---

*Nova's Git-Survival-Guide - Version 4.0 (Final)*

*https://java-developer.online*

**Happy Coding & Clean Commits!** 🎉