# Git

## OG GITHUB

…OG GITKRAKEN… OG GITLAB… OG BITBUCKET…

# Agenda

1. Hva er git (og github)?
2. Begreper og prinsipper – grunnleggende
3. Demo
4. Begreper og prinsipper – for samarbeid
5. Merge conflicts – Git-samarbeids største utfordring

# Hva er Git?

- Distributed version control system ← system/motor for versjonskontroll
- Primær funksjon: spore endringer i mapper og filer…
- …men brukes også for samarbeid, prosjektstyring, utviklingsplan og lansering

- Git er kun et system; det trenger et interface for å kunne brukes. Default interface er CLI (Command Line Interface / Terminal)

# Hva er GitHub?

- **GitHub** er en Git-server med skylagring, altså en server som hoster kode i mapper i skya
- **GitHub *Desktop*** er et GUI – altså en software med et grensesnitt som gjør bruken av Git mer visuell – og for mange enklere å bruke
- Det finnes mange ulike grensesnitt. Git brukes også som motor for versjonskontroll i store kjente utviklingsmiljøer (som for eksempel Microsoft DevOps og Azure, eller Jira og BitBucket)

# Hvorfor GitHub?

- Gode Git-rutiner og webgrensesnitt
- Tilgjengelig (gratis / lav kostnad)
- Stort community – mye og god hjelp
- GitHub Desktop tilbyr et enkelt grensesnitt inn i Git-verden

# Hvorfor ikke GitHub?

Andre grensesnitt tilbyr flere funksjoner som kan være nyttige, spesielt ved større samarbeidsprosjekter. Men flere funksjoner betyr også mindre oversikt og mer å sette seg inn i.

Se https://stackshare.io/stackups/github-vs-gitkraken og https://kinsta.com/blog/gitlab-vs-github/ for sammenligninger av GitHub mot GitKraken og GitLab

Hvis du er relativt ny med Git, anbefaler jeg å starte med GitHub, som gir god oversikt over de viktigste funksjonene!

# Git-funksjoner: Terminologi

En grunnleggende forståelse for Git-"språket", og hva det innebærer av funksjonalitet og rettigheter, gjør det mye enklere å utnytte. La oss se på **sammenhengen** mellom en del **ord** i Git-verden.

# Repository

- Prosjektmappe
- Public (open source) eller Private
- Kan gi samarbeidsrettigheter til andre GIT-brukere uavhengig av public/private
- En eier
- Opprettes på Git-server

# Clone

- Lager en lokal kopi av et **repository**
- Den lokale kopien synker mot original-**repositoriet**
- Både egne (eide) og andres **repositories** kan clones
- Du må være registrert som **collaborator** eller eie repositoriet for å kunne sende **commits** (endre) i **repositoriet**

# Fork

- Lager en kopi av et **repository**
- Kopien eies av deg, men synkes ikke (automatisk) mot original-repositoriet
- Du kan sende et **pull request** til original-repositoriet for å få koden vurdert for **merge** mot original-repositoriet
- Typisk flyt for open-source samarbeidsprosjekter

# .gitignore

- Fil som forteller Git hvilke mapper og filer som IKKE skal synkroniseres mellom lokalt og remote repository
- Spesielt viktig i node.js-baserte prosjekter

# Staging area

- Mellomsted mellom endringer du har gjort og en **commit**
- Tenk «flytteeske»; du legger ting i esken, tar kanskje noen ut, legger til flere før du teiper den sammen og skriver på en lapp om hva innholdet er

# Commit

- En (lokal) versjon
- Lagres på arbeidsstedet, må «sendes» (**push**) til Git-server for å sees i prosjektet

# Head

- Den «fremste» (nyeste) **commiten**, altså status quo for et **repository** eller en **branch**
- Inneholder en referanse til siste gjeldende **commit**.

# Commit message

- Melding som følger en **commit** (lappen som beskriver innholdet i flyttekassa)
- Kan skape forferdelige samarbeidsutfordringer hvis de ikke er gode!
  (Mer om det senere...)

# Push

Handling for å sende alle lokalt lagrede **commits** til remote **repository** på Git-server

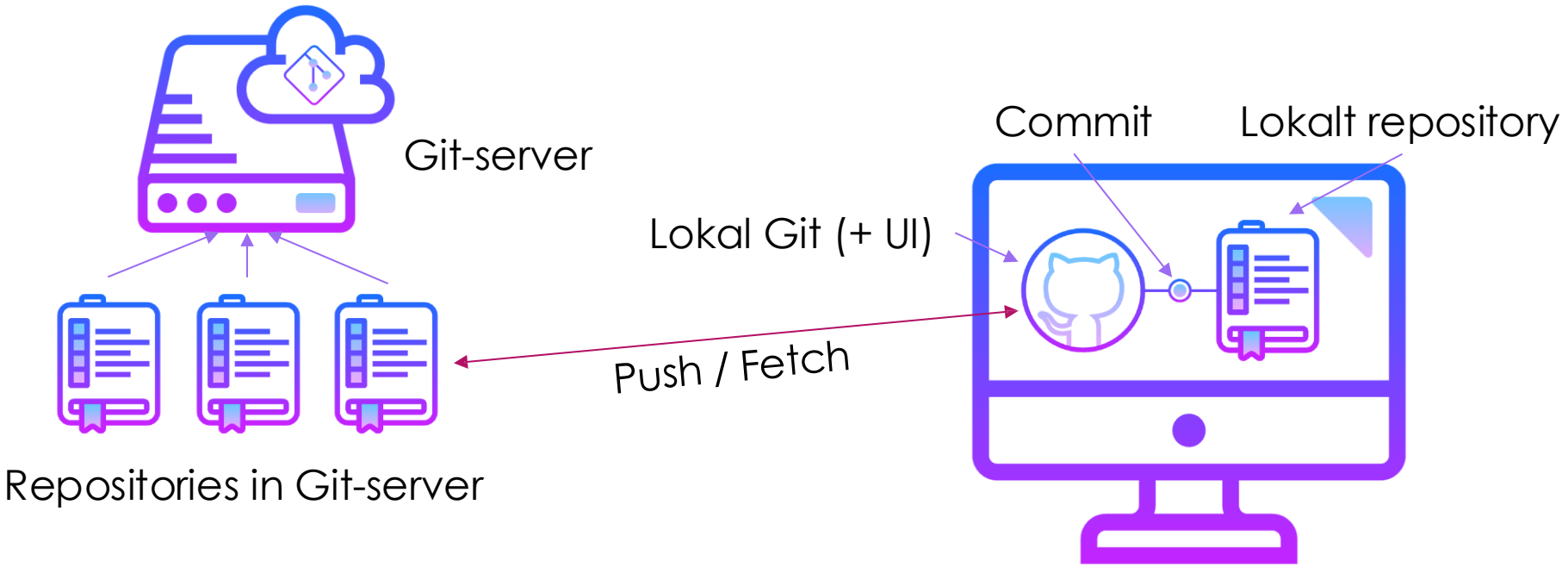
# Branch

- En arbeidsgren i et Git-prosjekt
- Et prosjekt kan bestå av mange branches
- Brukes typisk for å skille features/funksjonalitet og ulike programmere i felles prosjekter

# Publish

- Første gang man lager en **branch** (inkludert branchen main – hovedbranchen (eller rota) som lages ved oppretting av nytt prosjekt) må den publiseres
- Dette oppretter typisk kobling mellom lokal og remote **repository**

# Flyt


Git-server
Commit
Lokalt repository
Lokal Git (+ UI)
Push / Fetch
Repositories in Git-server

# Demo

# Collaboration

# Litt mer terminologi…

…for oversikt over samarbeidsbegreper!

# Collaborator

En felles utvikler/tester som er invitert som samarbeidende i et **repository** på en Git-server.

Settings > Collaborators

# Merge

- Å slå sammen to kodebaser ved å ta en **branch** inn i en annen
- Viktig del av samarbeid i kodeprosjekter

# Review

- Å se over og godkjenne kode ved en **merge** av en **branch** før den merges
- Viktig del av samarbeid i kodeprosjekter

# Pull request

- Be om at dine endringer i en **branch** blir **reviewet** og merget inn i en annen branch
- Viktig del av samarbeid i kodeprosjekter

# Merge conflict

- Gitprosjekters største mareritt...
- Oppstår når koden i to **brancher** er endret i begge branchene
- Må løses før man får foreta en **commit** av de mergede endringene
  (Mer om dette kommer)

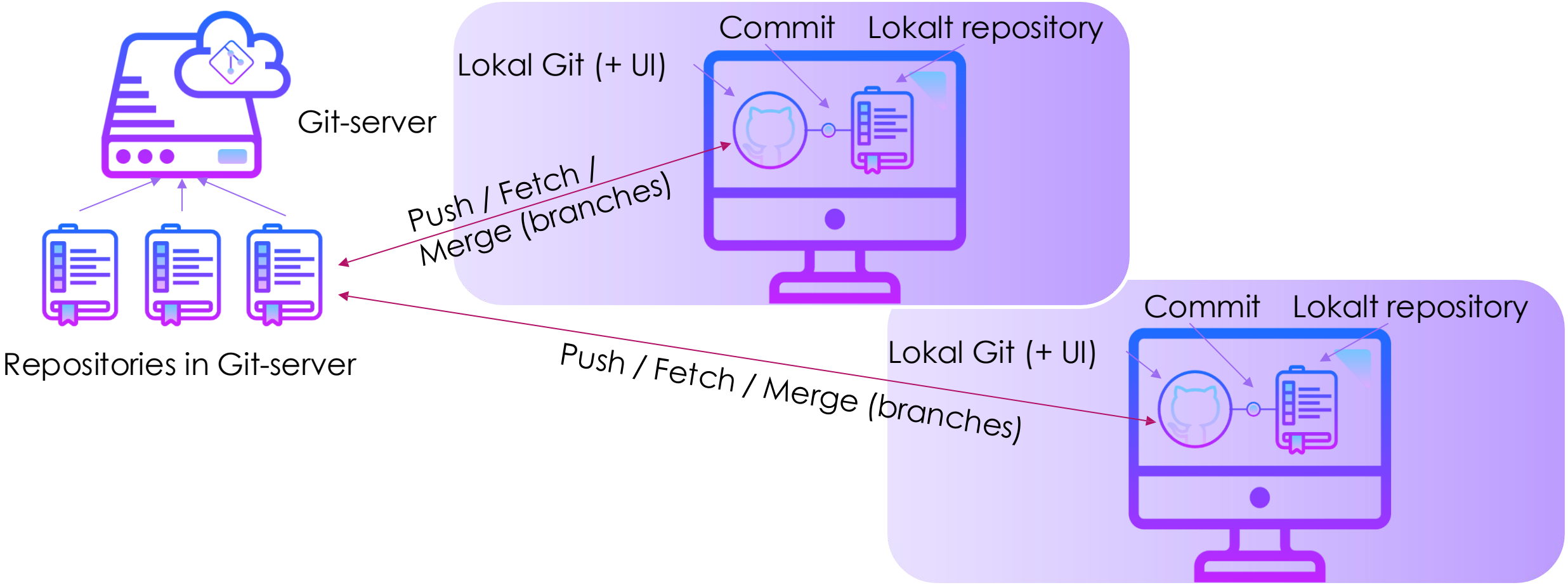
# Fetch

- Hent endringer som er sendt til et **repository**
- Typisk brukt hvis du har **clonet** eller **forket** et repository
- Gjøres automatisk før en **merge** for å sjekke eventuelle endringer og konflikter i original-repositoriet

# Issues

- Meldinger om feil/bugs i kode i et **repository**
- Issues brukes ofte av både kodere og testere for å påpeke feil/mangler/bugs som må løses.
- Issues kan refereres til i **commit messages** med hashtag+referansenummer (#123)

# Flyt i samarbeid



# Samarbeid I Git krever god kommunikasjon

Commit-meldinger og issues er primærmåtene å kommunisere på i Git-prosjekter

# Hvordan skrive gode commit messages?

- Forklar hvorfor endringer gjøres
- Hvilke effekter har endringen (påvirker det bevisst annen kode)?
- Ikke anta at leseren forstår det originale problemet
- Ikke anta at koden er selvforklarende
- Les commit-meldingen for å se om den hinter til forbedret kode/-struktur
- Den første setningen I en commit-melding er den viktigste!
- Beskriv alle begrensninger ved den innsendte koden
- Referer til GitHub issues ved bruk av #XXX

# Hvordan skrive gode commit messages?

**Tommelfingerregel**

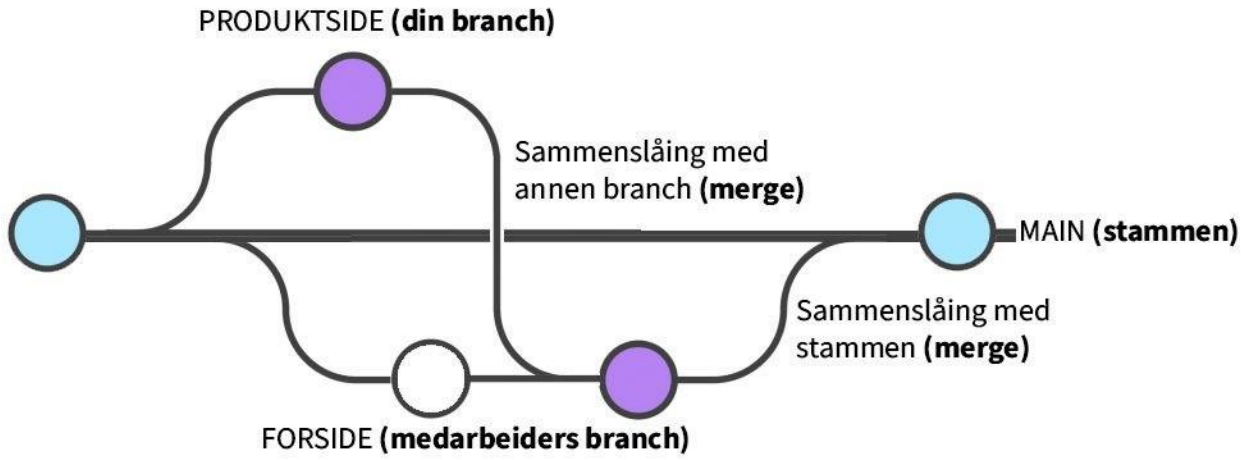
En god commitmelding kan alltid avslutte denne setningen:

**Hvis anvendt, vil denne commiten <*din commit-overskrift her*>**

*Eng: if applies, this commit will <your subject here>*

# Branching

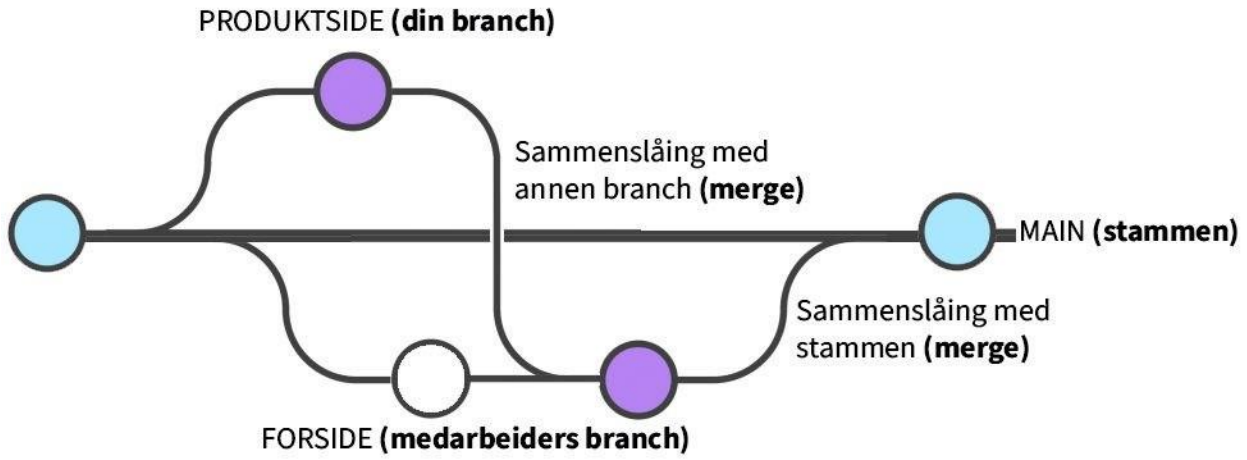
- Alle repositories har minst en branch (Git default: Main)
- Alle kodere i et Git-prosjekt kan opprette og merge branches.
- Branches fungerer som greiner i prosjekttreet som vokser ut av roten Main, og vokser tilbake inn i Main når de er ferdige




Make your own Git Branch diagrams: https://www.gliffy.com/blog/gitflow-diagrams

# Branching: Keep Main clean

- Beste praksis er å aldri kode direkte mot Main.
- La Main være alltid nyeste versjon av den fungerende kodebasen, altså vil Main alltid være siste oppdaterte, fungerende kode
- Dette gjør det mulig å la Main være kilden for alle nye brancher.




Make your own Git Branch diagrams: https://www.gliffy.com/blog/gitflow-diagrams

# Branching: Git Flow

En gammel, men fortsatt mye brukt modell/strategi for branching

Typisk fem typer brancher i et prosjekt. To hovedtyper:

- **Main**: hovedbranchen. Jobbes ikke *på*, kun *mot*.
- **Develop**: utviklingsmiljø. Hovedbranchen for koding.

Og tre støttetyper:

- **Feature**: utvikling av en bestemt funksjon
- **Release**: en branch for publisering/lansering av en ferdig versjon av prosjektet
- **Hotfix**: raske kodeendringer som fikser viktige bugs/feil

Mer om Git Flow: https://www.gitkraken.com/learn/git/git-flow

# Branching: GitHub Flow

En moderne, enklere modell/strategi for branching:

- **Opprett** branch med forklarende navn, eks: *add-code-of-conduct* eller *setup-product-page-template*
- **Utfør** koding/utvikling/endringer I henhold til branchens formål. Sørg for å gjøre kode ferdig, slik at branchen ikke inneholder uløste utfordringer.
- Lag en **Pull Request** så collaborators kan gå gjennom og eventuelt foreslå endringer i koden. Løs eventuelle forslag og bugs.
- Når Pull Requesten er godkjent av reviewerne, **merge** branchen mot Main
- **Slett** branchen

Dokumentasjon for alle stegene: https://docs.github.com/en/get-started/quickstart/github-flow

# Branching: Best practices

- Bli enige om en modell/strategi alle prosjektdeltagere skal følge.
- I samarbeidsprosjekter skal Main aldri jobbes på. Main er samlestedet for ferdig kode.
- I samarbeidsprosjekter bør færrest mulig kodere jobbe på samme branch.
- Bryt ned features i små, håndterbare deler. Lag branches med gode, beskrivende navn. Merge ofte.

# Litt mer terminologi…

…for fordypning av merge conflicts!

# Stash

- Lagre endringer trygt på et skjult sted (the stash stack)
- Å stashe working directory betyr å tilbakestille det til siste commit (versjon)

# Rebase

- Endre basen til en bransje fra en commit til en annen - så det ser ut som branchen er laget fra en annen commit
- Internt lager Git en ny commit og legger den på branchens base.

# Current changes

- Endringer i branchen du jobber på

# Incoming changes

- Endringer i branchen du forsøker å merge inn i din branch, eller
- Branchen du forsøker å merge oppå en annen branch (under en rebase)

# Merge conflicts

Typisk to tilfeller som skaper merge conflicts:

- **I starten av en merge**: skjer når det finnes endringer lokalt (I working directory eller staging area). Mergen stoppes for at disse endringene ikke skal bli overskrevet av innkommende (fra andre utviklere/brances) merge commits.
- **Under en merge**: konflikter mellom kilden (branchen) det merges fra og målet (branchen) det merges mot. Typisk har to utviklere gjort endringer I den samme filen, og Git vet da ikke hvilken av de nye kodeversjonene som skal brukes. Kanskje må deler av begge kodene brukes.

# Merge conflicts

Typen merge er relevant for merge conflicts:

- **Merge into current branch**: Hente endringer fra en annen branch inn i branchen du jobber på.
  Endringer fra den andre branchen blir **incoming changes**, endringer i branchen du jobber på er **current changes**.
- **Pull request (viktig når du reviewer)**: Send endringer i din branch inn i en annen branch via en (reviewed) merge. Endringer i din branch blir **incoming changes**, endringer i branchen du merger mot blir **current changes**.

# Merge conflicts

Git annoterer ulikheter i kode slik:

```
<<<<<<< HEAD (current change)
Kode fra din lokale branch
=========
Kode fra ekstern branch
>>>>>>>
```

Du som utvikler/reviewer må fjerne og rydde opp i disse annoteringene for å løse merge conflicten. En enkel tutorial kan sees på FreeCodeCamp.

# Merge conflicts

Hvis du bruker et IDE med Git-integrasjon (som VSCode), tilbyr de fleste av disse en snarvei til opprydning av annotering og kodeendringer når det oppstår en merge conflict.

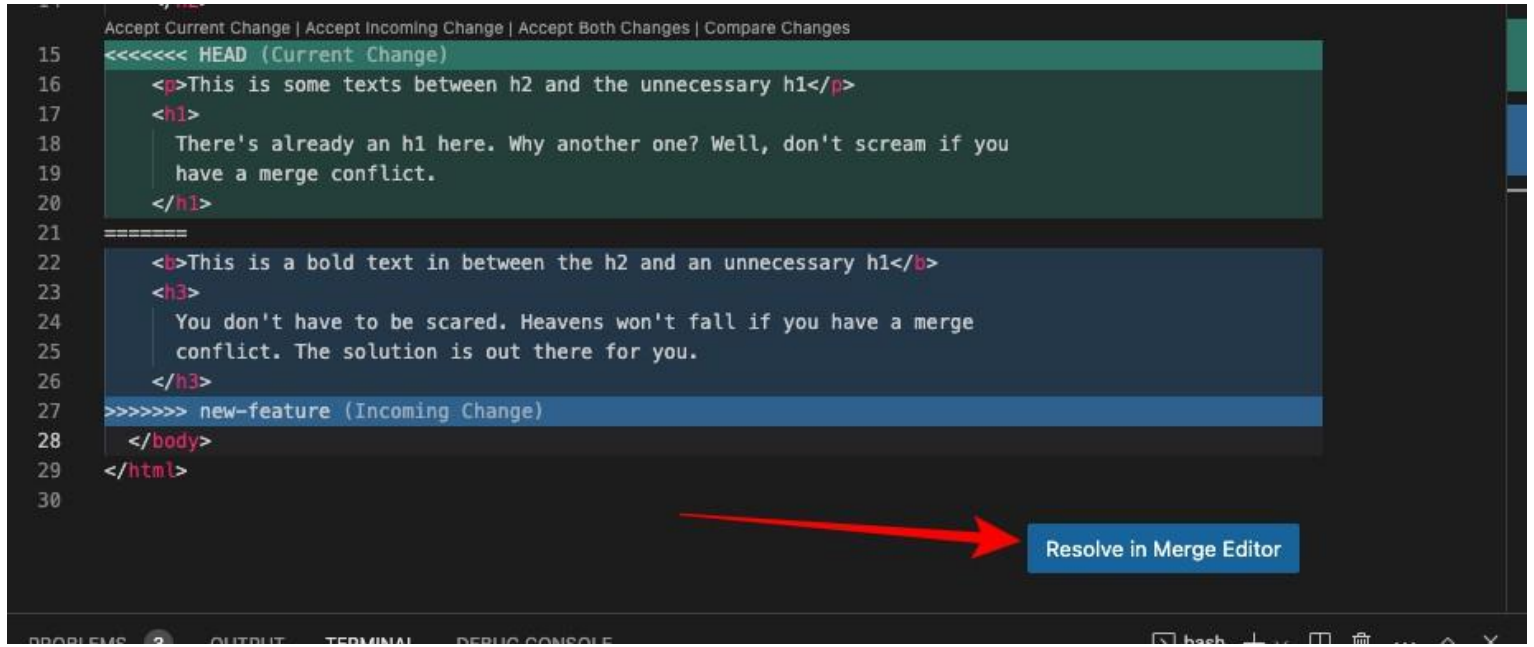
Hvis det er åpenbart at en av kodene skal beholdes og den andre kastes, får man gjerne valgene «**Keep current changes**» og «**Keep incoming changes**».

Basert på merge-typen, velg hvilken kodebase som skal beholdes, og IDE-et gjør resten for deg.

**OBS: Dette fungerer kun hvis en av kodebasene skal beholdes til fordel for den andre.**

# Merge conflicts

I tilfeller hvor kode må bearbeides før man kan løse merge conflicten, tilbyr de fleste IDE-er en merge editor.




Bilde fra https://www.freecodecamp.org/news/how-to-fix-merge-conflicts-in-git/

# Merge conflicts

Noen gode ressurser for å forberede seg på merge conflicts.

- https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/addressing-merge-conflicts/resolving-a-merge-conflict-on-github
- https://www.freecodecamp.org/news/how-to-fix-merge-conflicts-in-git/
- https://www.developernation.net/blog/git-internals-part-3-understanding-the-staging-area-in-git
- https://www.atlassian.com/git/tutorials/merging-vs-rebasing
- https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase

Bilde fra https://www.freecodecamp.org/news/how-to-fix-merge-conflicts-in-git/

# Merge conflicts

Git har to kommandoer som nesten kan tenkes på som lokal Control + Z («undo»). Disse kan være nyttige både før man skal merge større endringer dersom noe av kodebasen krever mer arbeid, eller hvis en merge conflict blir tilnærmet umulig å løse og man må jobbe fra en tidligere fungerende versjon.

- **Git Checkout**: endrer HEAD-pekeren til en tidligere commit
- **Git Reset**: matcher repositoriet i sin helhet til en spesifisert commit
- **Git Revert**: angrer en hel commit

**Les mer: https://www.atlassian.com/git/tutorials/resetting-checking-out-and-reverting**

Bilde fra https://www.freecodecamp.org/news/how-to-fix-merge-conflicts-in-git/

# Noen verktøy

- Le Git Graph – Commit Graph for GitHub, Chrome extension
- GitHub Projects – Prosjektstyring med synk mot GitHubkontoen og/eller – repositoriet ditt/deres
- Ulike grensesnitt for Git:
  - GitHub Desktop (anbefalt for relativt nye Git-ere)
  - GitKraken (best for store prosjekter med mange personer)
  - GitLab (no free option, best for professional teams with large environments)

# Git command cheat sheet

GitHub har en PDF med fin oversikt over Git-kommandoer. Den kan lastes ned gratis på https://education.github.com/git-cheat-sheet-education.pdf

GitHub
GIT CHEAT SHEET

Git is the free and open source distributed version control system that's responsible for everything GitHub related that happens locally on your computer. This cheat sheet features the most important and commonly used Git commands for easy reference.

**INSTALLATION & GUIS**
With platform specific installers for Git, GitHub also provides the ease of staying up-to-date with the latest releases of the command line tool while providing a graphical user interface for day-to-day interaction, review, and repository synchronization.

**GitHub for Windows**
https://windows.github.com

**GitHub for Mac**
https://mac.github.com

For Linux and Solaris platforms, the latest release is available on the official Git web site.

**Git for All Platforms**
http://git-scm.com

**SETUP**
Configuring user information used across all local repositories

`git config --global user.name "[firstname lastname]"`
set a name that is identifiable for credit when review version history

`git config --global user.email "[valid-email]"`
set an email address that will be associated with each history marker

`git config --global color.ui auto`
set automatic command line coloring for Git for easy reviewing

**SETUP & INIT**
Configuring user information, initializing and cloning repositories

`git init`
initialize an existing directory as a Git repository

`git clone [url]`
retrieve an entire repository from a hosted location via URL

**STAGE & SNAPSHOT**
Working with snapshots and the Git staging area

`git status`
show modified files in working directory, staged for your next commit

`git add [file]`
add a file as it looks now to your next commit (stage)

`git reset [file]`
unstage a file while retaining the changes in working directory

`git diff`
diff of what is changed but not staged

`git diff --staged`
diff of what is staged but not yet committed

`git commit -m "[descriptive message]"`
commit your staged content as a new commit snapshot

**BRANCH & MERGE**
Isolating work in branches, changing context, and integrating changes

`git branch`
list your branches. a * will appear next to the currently active branch

`git branch [branch-name]`
create a new branch at the current commit

`git checkout`
switch to another branch and check it out into your working directory

`git merge [branch]`
merge the specified branch's history into the current one

`git log`
show all commits in the current branch's history

**INSPECT & COMPARE**
Examining logs, diffs and object information

`git log`
show the commit history for the currently active branch

`git log branchB..branchA`
show the commits on branchA that are not on branchB

`git log --follow [file]`
show the commits that changed file, even across renames

`git diff branchB...branchA`
show the diff of what is in branchA that is not in branchB

`git show [SHA]`
show any object in Git in human-readable format

**SHARE & UPDATE**
Retrieving updates from another repository and updating local repos

`git remote add [alias] [url]`
add a git URL as an alias

`git fetch [alias]`
fetch down all the branches from that Git remote

`git merge [alias]/[branch]`
merge a remote branch into your current branch to bring it up to date

`git push [alias] [branch]`
Transmit local branch commits to the remote repository branch

`git pull`
fetch and merge any commits from the tracking remote branch

**TRACKING PATH CHANGES**
Versioning file removes and path changes

`git rm [file]`
delete the file from project and stage the removal for commit

`git mv [existing-path] [new-path]`
change an existing file path and stage the move

`git log --stat -M`
show all commit logs with indication of any paths that moved

**REWRITE HISTORY**
Rewriting branches, updating commits and clearing history

`git rebase [branch]`
apply any commits of current branch ahead of specified one

`git reset --hard [commit]`
clear staging area, rewrite working tree from specified commit

**IGNORING PATTERNS**
Preventing unintentional staging or commiting of files

```
logs/
*.notes
pattern*/
```

Save a file with desired patterns as .gitignore with either direct string matches or wildcard globs.

`git config --global core.excludesfile [file]`
system wide ignore pattern for all local repositories

**TEMPORARY COMMITS**
Temporarily store modified, tracked files in order to change branches

`git stash`
Save modified and staged changes

`git stash list`
list stack-order of stashed file changes

`git stash pop`
write working from top of stash stack

`git stash drop`
discard the changes from top of stash stack

**GitHub Education**

Teach and learn better, together. GitHub is free for students and teachers. Discounts available for other educational uses.

education@github.com
education.github.com

# Oppsummering

- Git fungerer perfekt når man er en…
- Commit kun ferdige løsninger.
- Commit messages er viktig for oversikt i prosjektet.
- Branches viktige for å dele opp prosjekt i features / collaborators.
- Merge conflicts oppstår alltid i samarbeidsprosjekter. Kontroll over kodebase, gode backup-rutiner og kommunikasjon rundt konflikter er viktige elementer i utviklingsflyten.