

GITHUB

FRA FØRSTE COMMIT TIL PROFESJONELL FLYT

En praktisk terminalguide



Kenneth Bareksten

Innhold

[Del 1: Fundamentet — Hva, hvorfor og hvordan?](#) 1.1 Marerittet uten versjonskontroll 1.2 Git vs. GitHub 1.3 Hvordan Git fungerer: de tre områdene

[Del 2: Riggning av verktøykassa](#) 2.1 Treenigheten 2.2 SSH-nøkler: én gang og aldri igjen 2.3 Identitet og innlogging

[Del 3: Den daglige flyten](#) 3.1 De 5 viktige og noen til 3.2 .gitignore: Hva GitHub ikke trenger å vite 3.3 Commit-meldinger som faktisk hjelper

[Del 4: Parallelle universer](#) 4.1 Branches: Eksperimenter uten fare 4.2 Merge vs. Rebase: To veier til målet 4.3 Git Worktree: Proff-trikset

[Del 5: Samarbeid og etikette](#) 5.1 Forking: Bidra uten tilgang 5.2 Pull Requests: Mer enn bare kode 5.3 Code Review: Vær snill og vær nyttig 5.4 Issues og Milestones: Hold orden på oppgavene

[Del 6: Sikkerhet og automatisering](#) 6.1 GitHub Secrets: Passordene dine fortjener bedre 6.2 GitHub Actions: Roboten som gjør jobben for deg 6.3 Branch Protection: Litt byråkrati som redder livet ditt 6.4 Dependabot: Automatiske sikkerhetsoppdateringer

[Del 7: Vis det frem](#) 7.1 README: Førsteintrykket ditt på nettet 7.2 GitHub Pages: Fra repo til nettside på fem minutter 7.3 Tags og releases: Versjonering av prosjektet

[Del 8: Når alt brenner](#) 8.1 Merge Conflict: To endret samme linje 8.2 Detached HEAD: Du er ingen steder og alle steder 8.3 Angre på ulike måter 8.4 Du pushet noe du ikke skulle 8.5 git reflog: Tidsmaskinens tidslinje

[Oppslagsverk](#)

[Etterord](#)

INNLEDNING

Har du noen gang følt at Git og GitHub virker som et lukket brorskap? At terminalen ser ut som noe fra en Matrix-film, og at frykten for å slette alt hindrer deg i å trykke på knappene?

Denne boka er skrevet for deg som kan litt fra før. Boka er terminalbasert fra start til slutt. Du lærer Git slik profesjonelle utviklere bruker det: gjennom kommandolinjen. Det betyr ikke at du må være komfortabel med terminalen allerede.

Del 1 og 2 tar deg gjennom det du trenger. Men hvis du forventer en guide til knappene i GitHub sitt nettgrensesnitt, er ikke dette den boka. Her skriver du kommandoer og du forstår hva de gjør.

Målet er konkret: gi deg arbeidsvanene til en profesjonell utvikler. Fjerne mystikken rundt Git, temme terminalen og la deg samarbeide med hvem som helst uten å lage kaos.

Hvert kapittel avsluttes med en øvingsoppgave. Det er der læringen skjer.

Boka er din guide til installasjon, terminalbruk, avanserte konsepter, sikkerhet og profesjonell etikette. På nettsiden [Git for nybegynnere](#) kan du prøve det du leser interaktivt og få repetert de viktigste konseptene.

Lykke til og god læring!

Del 1: Fundamentet: Hva, hvorfor og hvordan?

1.1 Marerittet uten versjonskontroll

Du kjenner til det: mappen med fil_v2_FERDIG_LEVER.docx og fil_v2_FERDIG_LEVER_rettet_2.docx. Kanskje en mappe som heter Gammelt og en som heter GammelVersjon2. Ingen aner hva som er nyest, og ingen aner hva som ble endret fra den ene til den andre.

Multipliser dette med fem utviklere på samme kodebase, og du har oppskriften på et prosjekt som ligner mer på et slagsted enn et produkt.

Git er tidsmaskinen som løser dette. Hvert lagringspunkt heter en commit og er som et fotografi av hele prosjektet på et bestemt tidspunkt. Du kan hoppe tilbake til et hvilket som helst fotografi, se hva som ble endret og av hvem, og gjenopprette det du trodde var borte for alltid.

```
* commit 0181ee5ca59643cce090389252ad4688b3107b80 (HEAD -> main)
| Author: barx10 <152427691+barx10@users.noreply.github.com>
| Date: Wed Jan 14 22:34:54 2026 +0100
|
|     Fix: Render location markers (Pin, Target, etc.) on map export with correct positioning
|
* commit ace168226dd8e2b4c6ffc748006012342f7d447e
| Author: barx10 <152427691+barx10@users.noreply.github.com>
| Date: Wed Jan 14 21:38:32 2026 +0100
|
|     UI Cleanup & Enhancements:
|     - Removed Carto-Art API key integration (now fully client-side)
|     - Curated map styles to 10 unique themes with distinct colors
|     - Fixed High-Res Export: Correct zoom levels and exact viewport matching
|     - Added Theme Color Picker for gradients and boxes
|     - Restyled Scale Bar to classic ruler style
|     - Fixed side panel layout issues
```

Fig. 1. Med kommandoen **git log --graph** i terminalen, får du oversikt over hva du har jobbet med en aktuell dag.

January 18, 2026

Created 13 commits in 1 repository

Opened 9 pull requests in 1 repository

barx10/tidstreneren 9 merged

- [Redesign UI with fixed sidebar layout for better accessibility](#) Jan 18
- [Claude/add language toggle u9 yi a](#) Jan 18
- [Add Puter.js/ElevenLabs TTS for more natural speech](#) Jan 18
- [Add English Norwegian language toggle](#) Jan 18

Fig. 2. Den samme oversikten kan du finne på Github.

1.2 Git vs Github

Disse to navnene brukes om hverandre hele tiden, men de er ikke det samme.

Git er programvaren som kjører lokalt på din maskin. Den holder styr på alle endringer, trenger ikke internett og kan brukes helt offline. Tenk på Git som motoren i en bil.

GitHub er nettsiden i skyen der du lagrer og viser frem koden din. GitHub bruker Git under panseret, men legger til et grensesnitt, samarbeidsverktøy og automatisering. Det er garasjen der bilen står når du ikke kjører.

Du kan bruke Git uten GitHub. Men i praksis bruker nesten alle GitHub for å samarbeide og ta sikkerhetskopier.

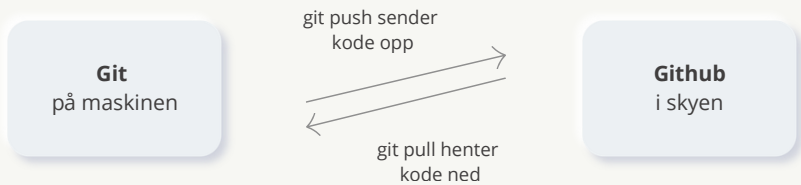


Fig. 3. Prinsippet til hvordan git push og git pull fungerer.

1.3 Hvordan Git fungerer: de tre områdene

Når du jobber med Git, befinner filene dine seg alltid i ett av tre områder. Å forstå disse er nøkkelen til å ikke bli forvirret når ting ikke oppfører seg som forventet.

- *Arbeidsmappen (Working Directory)*: Filene slik de ser ut på din maskin akkurat nå. Når du redigerer noe i en editor er det her endringen skjer.
- *Klargjøringsområdet (Staging Area)*: Tenk på dette som en koffert du pakker før en reise. Du legger i de tingene du vil ha med i neste commit og kan la andre ting ligge igjen.
- *Repoet (Repository)*: Når du committer, tas det et fotografi av alt i kofferten og lagres permanent i prosjektets historikk.



Fig. 4. Relasjonen mellom de tre områdene og kommandoene du bruker for å kommunisere mellom dem.

TIPS

`git status` er din beste venn. Kjør den når som helst for å se nøyaktig hvilke filer som er endret, hvilke som er klargjort og hva som er klart for commit.

Øvingsoppgave del 1

Du har lest om tidsmaskinen. Nå skal du skru den på. Åpne terminalen og kjør dette steg for steg:

```
mkdir mitt-første-repo
cd mitt-første-repo
git init
```

`mkdir` oppretter en mappe, `cd` bytter til mappa og `git` følger nå med på alt som skjer i mappen. Lag en fil:

```
echo "Hei, Git" > notat.txt
```

`echo` skriver tekst til terminalen. `>` omdirigerer den til en fil i stedet for skjermen. Resultatet er en ny fil kalt `notat.txt` med innholdet "Hei, Git". Du kan også bare lage filen i hvilken som helst teksteditor, effekten er den samme.

Nå som filen finnes, se hva Git tenker om den:

```
git status
```

Du er i arbeidsmappen. Git ser filen, men den er verken klargjort eller lagret. Legg den i kofferten:

```
git add notat.txt
git status
```

Se hvordan meldingen endret seg. Filen er nå i staging area. Ta fotografiet:

```
git commit -m "feat: legg til notat"  
git log --oneline
```

-m lar deg skrive commit-meldingen direkte i kommandoen. Uten den åpner Git en teksteditor og venter på at du skal skrive den der. "feat: legg til notat" er selve meldingen — feat: er et prefiks som sier at dette er ny funksjonalitet, hentet fra Conventional Commits-systemet du leser mer om i Del 3.

git log --oneline viser historikken kompakt, én commit per linje. Du ser en kort hash (en unik ID for committen) og meldingen din. Der er tidsmaskinen din, synlig for første gang.



Husk dette!

- Git er en tidsmaskin som tar fotografier av prosjektet ditt. Hvert fotografi heter en commit.
- Git og GitHub er ikke det samme. Git er motoren på maskinen din. GitHub er garasjen på nettet.
- Filer befinner seg alltid i ett av tre områder: arbeidsmappen der du jobber, staging area der du pakker kofferten, og repositoret der fotografiene lagres permanent.
- git status forteller deg nøyaktig hvor du er. Kjør den ofte.

Del 2: Rigging av verktøykassa

2.1 Treenigheten du bør kunne noe om

Tre verktøy er alt du trenger for å komme i gang:

- Git: Last ned fra git-scm.com. På Mac kan du også bruke Homebrew: `brew install git`
- VS Code: Din editor, fra code.visualstudio.com. Har innebygget terminal og Git-integrasjon.
- GitHub CLI (gh): Kommandolinjeverktøyet som kobler terminalen rett til GitHub-kontoen din. Last ned fra cli.github.com.

Etter installasjon, verifiser at alt er på plass:

```
git --version
gh --version
```

2.2 SSH-nøkler: én gang og aldri igjen

Når du pusher til GitHub, må GitHub vite at det er deg. Brukernavn og passord fungerer, men med tostegsbekreftelse er det raskt tungvint. SSH-nøkler løser dette på et elegant vis.

Du lager et digitalt nøkkelpar. Den *private* nøkkelen ligger på din maskin og forlater aldri datamaskinen din. Den *offentlige* nøkkelen gir du til GitHub. Når du kobler til, matcher de og GitHub vet at det er deg uten at du trenger å taste passord.

Generer nøkkelparet:

```
ssh-keygen -t ed25519 -C "din@epost.no"
```

Trykk Enter på alle spørsmål for å bruke standardverdier. Kopier deretter innholdet av den offentlige nøkkelen:

```
# Mac/Linux:
cat ~/.ssh/id_ed25519.pub

# Windows (PowerShell):
Get-Content ~/.ssh/id_ed25519.pub
```

Gå til GitHub → Settings → SSH and GPG keys → New SSH key, og lim inn innholdet. Ferdig, dette gjør du bare én gang per maskin.

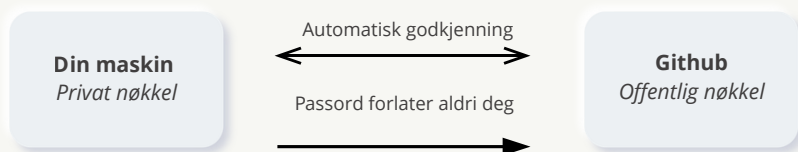


Fig. 5. Kun godkjenningen blir automatisk overført, ikke ømfintlig informasjon

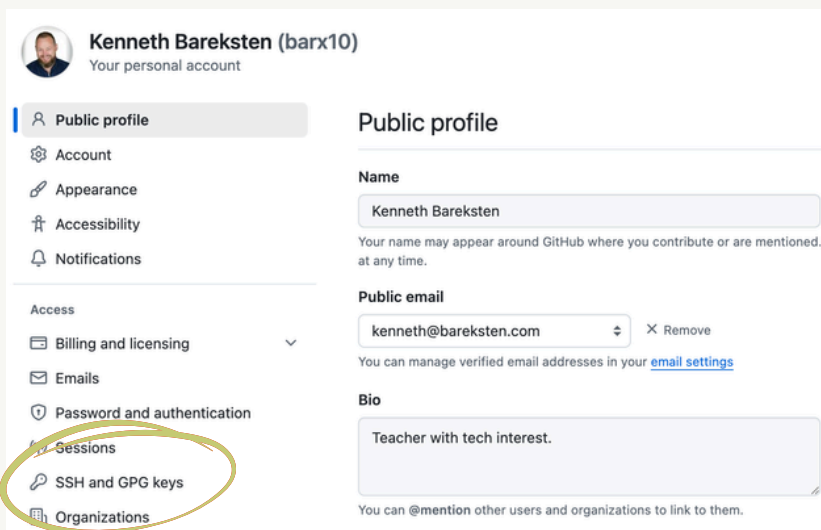


Fig. 6. Trykk på bildet av deg opp i høyre hjørnet, SSH- og GPG-nøkler finner du på venstre side.

2.3 Identitet og innlogging

Fortell Git hvem du er. Disse kommandoene setter navn og e-post på alle commits du lager:

```
git config --global user.name "Ditt Navn"
git config --global user.email "din@epost.no"
```

Koble deretter GitHub CLI til kontoen din:

```
gh auth login
```

Velg GitHub.com, SSH som protokoll og følg instruksene i nettleseren.

TIPS

Vil du sjekke hvilken konfigurasjon Git bruker akkurat nå? Kjør: `git config --list`

Øvingsoppgave del 2

Du skal nå koble maskinen din permanent til GitHub. Etter dette trenger du aldri taste passord igjen. Fortell Git hvem du er:

```
git config --global user.name "Ditt Navn"
git config --global user.email "din@epost.no"
```

Generer SSH-nøkkelparet:

```
ssh-keygen -t ed25519 -C "din@epost.no"
```

Trykk Enter på alle spørsmål. To filer blir opprettet: den private nøkkelen (aldri del denne) og den offentlige nøkkelen som ender på `.pub`. Kopier den offentlige nøkkelen:

```
# Mac:
cat ~/.ssh/id_ed25519.pub | pbcopy

# Linux:
cat ~/.ssh/id_ed25519.pub | xclip -sel clip

# Windows:
cat ~/.ssh/id_ed25519.pub | clip
```

Gå til github.com/settings/keys, klikk New SSH key, gi den et navn du kjenner igjen (f.eks. "MacBook hjemme"), og lim inn. Klikk Add SSH key, velg Authentication. Test at alt henger sammen i terminalen:

```
ssh -T git@github.com
```

Får du Hi [brukernavn]! er du ferdig. Maskinen og GitHub kjenner hverandre nå.

Vel du også verifisere at Git-konfigurasjonen ble riktig?

```
git config --list
```

Du skal se navn og e-post i listen. Hvis noe er feil, kjør `git config --global` på nytt med riktig verdi.



Husk dette!

- SSH-nøkler er engangsarbeid. Gjør det én gang per maskin, og du slipper passord for alltid.
- Git vet ikke hvem du er uten git config. Navn og e-post på commits er permanent, sett det riktig fra starten.
- GitHub CLI (gh) er ikke påkrevd, men det gjør hverdagen merkbart enklere.

Del 3: Den daglige flyten (Terminalen)

3.1 De fem viktige og noen til

De fem viktige

```
git status      # Hva er endret? Kjør denne først, alltid
git add .      # Klargjør ALLE endringer
git commit -m "beskriv hva og hvorfor"
git push       # Send til GitHub
git pull       # Hent andres endringer
```

Når du vil se hva som har skjedd

```
git log         # Hele historikken (trykk q for å avslutte)
git log --oneline # Kompakt: én commit per linje
git log --oneline --graph # Med visuell branching-struktur
git diff        # Hva er endret men ikke klargjort?
git diff --staged # Hva er klargjort for neste commit?
```

Når du vil sette ting på pause

git stash er som å legge halvferdig arbeid i en skuff. Nyttig når du raskt må bytte branch uten å committe noe du ikke er ferdig med.

```
git stash      # Legg endringene i skuffen
git stash pop  # Ta dem ut igjen
```

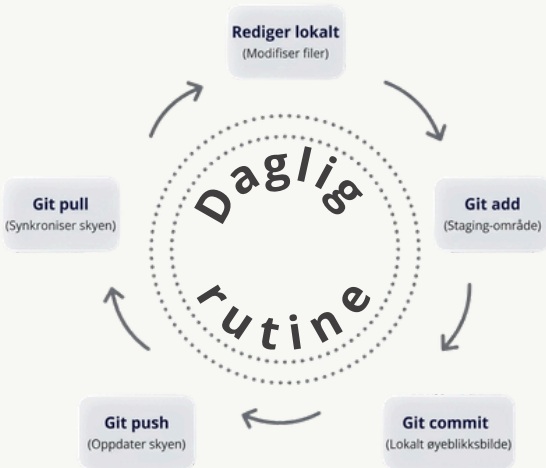


Fig. 7. Daglig arbeidsflyt som bør bli en del av rutinen din.

3.2 .gitignore: Hva GitHub ikke trenger å vite

Noen filer skal aldri havne på GitHub. Passord og API-nøkler er åpenbart, men også tunge avhengighetsmapper og maskinspesifikke innstillinger.

Lag en fil som heter `.gitignore` i rotmappen til prosjektet:

```
# Hemmeligheter
.env
.env.local

# Node.js
node_modules/

# Python
__pycache__/*
*.pyc
venv/

# Operativsystem
.DS_Store
Thumbs.db

# Editor (legg til din egen editor her, eller bruk
gitignore.io)
.vscode/
.idea/
```

Du trenger ikke huske disse. På gitignore.io kan du generere en ferdig `.gitignore` for akkurat ditt programmeringsspråk og operativsystem.

TIPS

Sett opp `.gitignore` før første commit. Har du allerede pushet en fil ved uhell, må du først fjerne den fra Git med `git rm --cached <filnavn>`, legge den til `.gitignore`, og committe på nytt.





3.3 Commit-meldinger som faktisk hjelper

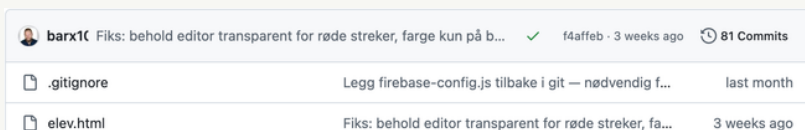
"Fikset greier" og "oppdatert" er nytteløse commit-meldinger. Om to måneder husker du ikke hva du mente. Om to år er det en annen person som skal lese det.

En god commit-melding svarer på to spørsmål: hva ble endret, og hvorfor? Mange prosjekter bruker *Conventional Commits*, et lite system med faste prefikser:

```
feat: legg til påloggingsside med Google OAuth
fix: fiks krasj når brukernavn er tomt
docs: oppdater README med installasjonsinstrukser
refactor: flytt database-logikk til egen modul
chore: oppdater avhengigheter til nyeste versjon
```

Prefikset gjør det enkelt å skanne historikken. En liten investering som betaler seg mange ganger.

Dårlige commit-meldinger	Gode commit-meldinger
 Fikset greier med siden	 feat: legg til påloggingsside
 Oppdatering	 fix: krasj ved tomt brukernavn



Commit Message	Author	Date	Commits
Legg firebase-config.js tilbake i git — nødvendig f...	f4affeb	3 weeks ago	81 Commits
Fiks: behold editor transparent for røde streker, fa...	elev.html	3 weeks ago	

Fig. 8. Skjermbilde fra et GitHub-repo som viser beskrivende commit-meldinger knyttet til hver fil.

Øvingsoppgave del 3

Du har nå verktøyene. Nå skal du bruke dem slik en utvikler faktisk jobber. Opprett et nytt repo direkte fra terminalen:

```
gh repo create daglig-flyt --public --clone
cd daglig-flyt
```

Lag en .gitignore før du skriver en eneste linje kode:

```
echo ".DS_Store" > .gitignore
echo "node_modules/" >> .gitignore
git add .gitignore
git commit -m "chore: legg til gitignore"
```

Lag en fil og skriv noe i den:

```
echo "# Daglig flyt" > README.md
echo "Dette repoet er en øvelse i den daglige Git-flyten." >>
README.md
```

Sjekk status, klargjør og commit:

```
git status
git add README.md
git status
```

Legg merke til at meldingen endret seg mellom de to git status-kallene. Nå tar du fotografiet:

```
git commit -m "docs: legg til README"
git log --oneline
```

Push til GitHub og verifiser at det dukker opp:

```
git push
gh repo view --web
```

Nettleseren åpner repoet ditt. Du skal se begge committene i historikken og README-en gjengitt på forsiden. Det er slik alle prosjekter ser ut når de er gjort riktig fra starten.



Husk dette!

- Den daglige flyten er alltid den samme: git add, git commit, git push. Alt annet er varianter av dette.
- Commit-meldinger er kommunikasjon, ikke kvitteringer. Fremtidsdeg leser dem.
- .gitignore settes opp før første commit. Etterpå er det ekstraarbeid.
- git stash er skuffen for halvferdig arbeid. Bruk den i stedet for å committe noe du ikke er ferdig med.

Del 4: Parallele universer (Branches & Worktrees)

4.1 Branches: Eksperimenter uten fare

En branch er en kopi av prosjektet der du kan eksperimentere fritt uten å røre det som faktisk fungerer. Når du er fornøyd, slår du den sammen med main, som hovedfil i repoet. Den skal alltid ha siste oppdaterte kode.

```
git checkout -b feature/pålogging # Lag og bytt til ny branch
git branch                        # Se hvilke branches som
finnes
git checkout main                 # Bytt tilbake til main
git merge feature/pålogging      # Slå sammen med main
git branch -d feature/pålogging  # Slett branchen når du er
ferdig
```

Navngiving av branches er ikke tilfeldig. Et system mange bruker er:

- feature/navn – ny funksjonalitet
- fix/navn – fikser en feil
- hotfix/navn – kritisk feil som må fikses umiddelbart
- docs/navn – dokumentasjon
- refactor/navn – rydding i koden uten ny funksjonalitet

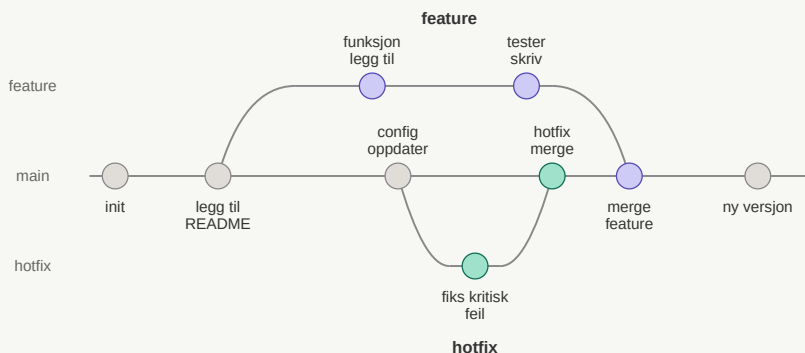


Fig. 9. Git-graf som viser tre parallele historikker: main-grenen løper horisontalt, feature-grenen spretter av og inneholder to commits før den merges tilbake, og hotfix-grenen løser en kritisk feil direkte på main. Sirkler er commits, linjer er historikk.

4.2 Merge vs. Rebase: To veier til målet

Når du skal få endringene fra en branch inn i main, har du to verktøy: merge og rebase. Begge gjør jobben, men historikken ser forskjellig ut etterpå

git merge: Bevar historikken som den er

Merge slår to branches sammen og lager en ny merge-commit. Du kan tydelig se når to arbeidstråder ble flettet. Det er trygt, det er ærlig og det er hva de fleste bruker.

```
git checkout main
git merge feature/pålogging
```

git rebase: Ryddig historikk, høyere risiko

Rebase "spiller av" commitene dine på toppen av en annen branch. Resultatet er en pen, lineær historikk uten flettecommitt. Baksiden er at det skriver om historikken, noe som kan skape problemer hvis andre jobber på samme branch.

```
git checkout feature/pålogging
git rebase main
```

⚠️ ADVARSEL

Bruk aldri rebase på commits som allerede er pushet og delt med andre. Det skriver om historikken og får andres kopier til å krasje.

Tommelfingerregel: bruk merge i team og når du er usikker. Bruk rebase når du jobber alene på en lokal branch og vil ha pen historikk før en Pull Request.

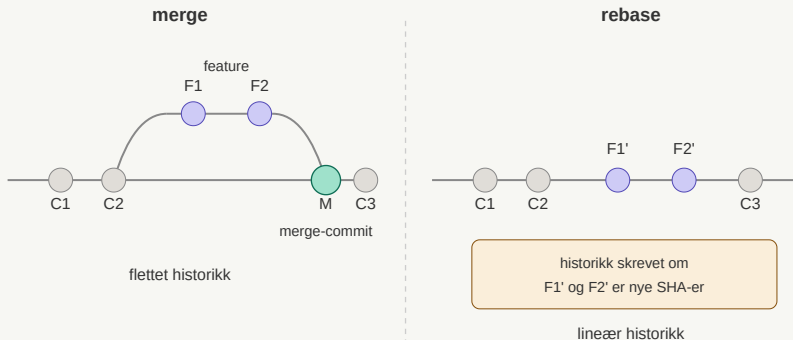


Fig. 10. Merge bevarer fullstendig historikk med en egen merge-commit der grenene møtes. Rebase skriver om historikken slik at feature-commits legges lineært etter siste commit på main. F1' og F2' er teknisk nye commits med nye SHA-er, selv om innholdet er det samme.

SHA, som vist i fig. 10, er en unik fingeravtrykk-kode som Git genererer for hver commit. Den ser slik ut: a3f8c21. Når du gjør rebase, lager Git nye commits med nytt innhold (ny plassering i historikken), og da endres SHA-en selv om kodeendringen er identisk. Det er derfor F1' og F2' er "nye" commits i teknisk forstand.

4.3 Git Worktree: Proff-trikset

Må du fikse en hastefeil i main akkurat når du er midt i å bygge en ny funksjon? Normalt måtte du stashe eller committe halvferdig arbeid. Med worktrees slipper du det.

Et worktree lar deg ha to versjoner av prosjektet åpent i separate mapper samtidig, begge koblet til det samme repositoriet.

```
git worktree add ../hotfix main # Lag ny mappe med main-branchen # ...gjør fiksen i ../hotfix...
git worktree remove ../hotfix # Rydd opp når du er ferdig
```

Åpne ../hotfix i din foretrukne editor, fiks feilen, commit og push. Så går du tilbake til der du var uten at noe er forstyrret.

Øvingsoppgave — Del 4

Du skal nå oppleve hvordan parallelt arbeid faktisk fungerer i Git. Lag en ny branch og bytt til den:

```
git checkout -b feature/om-meg
```

Lag en fil og skriv noe i den:

```
echo "# Om meg" > om-meg.md
echo "Jeg lærer Git." >> om-meg.md
```

Klargjør og commit:

```
git add om-meg.md
git commit -m "feat: legg til om-meg-side"
```

Bytt tilbake til main og legg merke til at om-meg.md forsvinner fra mappen:

```
git checkout main ls
```

Filen eksisterer fortsatt, men bare på branchen. Slå sammen:

```
git merge feature/om-meg ls
```

Nå er filen tilbake. Rydd opp branchen siden jobben er gjort:

```
git branch -d feature/om-meg
```

Se hva som skjedde i historikken:

```
git log --oneline --graph
```

Du ser en linje som greiner ut og fletter seg tilbake. Det er den visuelle bekreftelsen på at branching og merging fungerte som det skal. I et team ville dette vært arbeidet til to forskjellige utviklere som aldri forstyrret hverandre



Husk dette!

- En branch koster ingenting å lage. Eksperimenter fritt og main forblir urørt.
- Merge bevarer historikken som den er. Rebase rydder den opp, men skriver den om. Når du er usikker: bruk merge.
- Aldri rebase på commits som allerede er delt med andre.
- Worktrees lar deg jobbe på to branches samtidig uten å forstyrre noen av dem.

Del 5: Samarbeid og etikette

5.1 Forking: Bidra uten tilgang

Når du vil bidra til et prosjekt du ikke eier, kan du ikke pushe direkte. Da forker du repoet. En fork er din personlige kopi av et andres prosjekt på GitHub. Du gjør endringer i din fork, og sender en Pull Request til det originale repoet.

1. Klikk Fork på GitHub (øvre høyre på reposiden)
2. Klon din fork: `git clone git@github.com:dittbrukernavn/repo.git`
3. Lag en branch, gjør endringer, commit og push til din fork
4. Gå til GitHub og klikk "Compare & pull request"
5. Beskriv hva du har gjort og hvorfor, og send inn

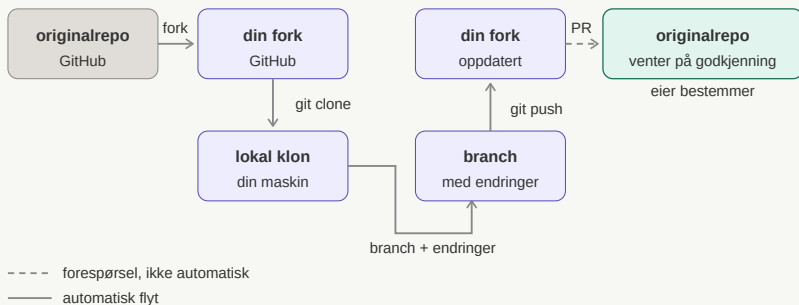


Fig. 11. Fork-workflow i seks steg: originalrepoet forkes til din egen kopi på GitHub, klones ned til maskinen, endringer gjøres i en egen branch, pushes tilbake til forken, og sendes som en pull request til originalrepoet. Den stiplede pilen markerer at en PR er en forespørsel om å flette inn kode, ikke en automatisk handling. Eieren av originalrepoet bestemmer om endringene godtas.

For å holde din fork oppdatert med originalrepoet:

```
git remote add upstream git@github.com:originalbruker/repo.git
git pull upstream main
git push origin main
```

5.2 Pull Requests: Mer enn bare kode

En Pull Request (PR) er en forespørsel om å slå sammen kode. Men like mye er det en samtale. En god PR gir kontekst: hva løser dette, og er det noe man bør teste spesielt?

- Skriv en tydelig tittel – gjerne med Conventional Commits-stil: feat: legg til ...
- Beskriv hva endringen gjør og hvorfor
- Legg ved skjermbilder hvis det er noe visuelt
- Hold PR-en liten. En liten PR får raskere tilbakemelding
- Koble til en issue med "Closes #42" i beskrivelsen, det lukker issuen automatisk når PR-en merges

TIPS

Lag en PR-mal slik at du eller teamet bruker samme format. Lag filen `.github/pull_request_template.md` og GitHub fyller den inn automatisk i alle nye PR-er.

5.3 Code Review: Vær snill og vær nyttig

Når du reviewer kode er du ikke dommer, du er medforfatter. Målet er et bedre produkt, ikke å vinne en debatt.

- Spør heller enn å påstå: "Hva om vi prøver X?" i stedet for "Dette er feil"
- Skill mellom blokkere (må fikses) og forslag (hyggelig om du fikser det)
- Ros det som er bra. Code review er ikke bare en feilliste
- Hold tonen profesjonell. Kommentarene er permanente.

5.4 Issues og Milestones: Hold orden på oppgavene

GitHub Issues er ikke bare for feilmeldinger. Det er et fullverdig oppgavehåndteringssystem. Bruk issues til å tracke bugs, planlegge funksjoner og diskutere idéer.

```
gh issue list                # Se alle åpne issues
gh issue create              # Lag en ny issue
interaktivt
gh issue create --title "Fikse login" --body "Beskrivelse..."
```

Milestones samler relaterte issues under et felles mål, for eksempel "Versjon 1.0". Det gir en rask oversikt over fremdriften. Issue-maler gjør det enkelt for folk å rapportere feil på en strukturert måte, lag dem i `.github/ISSUE_TEMPLATE/`.

Øvingsoppgave — Del 5

Du skal nå bidra til et prosjekt du ikke eier. Slik fungerer nesten all åpen kildekode-utvikling.

Gå til github.com/barx10/Git_for_nybegynnere og klikk Fork øverst til høyre. GitHub lager nå din egen kopi av repoet.

For å klonе forken din lokalt:

```
gh repo clone Git_for_nybegynnere
cd Git_for_nybegynnere
```

Lag en branch for endringen:

```
git checkout -b fix/skrivefeil
```

Åpne README-filen i din foretrukne editor, finn en skrivefeil eller et komma som mangler, og lagre. Klargjør og commit

```
git add README.md
git commit -m "fix: rett skrivefeil i README"
```

Push branchen til din fork:

```
git push -u origin fix/skrivefeil
```

Gå til GitHub. Du vil se en gul boks øverst som tilbyr å lage en Pull Request. Klikk Compare & pull request, skriv en kort beskrivelse av hva du endret og hvorfor, og send inn.

Du har nå gjort nøyaktig det millioner av utviklere gjør hver dag når de bidrar til prosjekter de ikke eier. Svaret du får tilbake, enten det er godkjenning, spørsmål eller avslag, er en normal del av prosessen. Det er sånn kode blir bedre.



Husk dette!

- En fork er din personlige kopi av et prosjekt du ikke eier. En Pull Request er forespørselen om å få endringene dine inn igjen.
- En god PR er liten, tydelig og gir kontekst. Store PR-er tar lengre tid å reviewe og skaper mer konflikt.
- Code review handler om produktet, ikke personen. Spør heller enn å påstå.

Del 6: Sikkerhet og automatisering

6.1 GitHub Secrets: Passordene dine fortjener bedre

API-nøkler og passord i koden er en klassisk feil. Pushet en gang til GitHub, og det kan være kompromittert før du rekker å slette det, bots scanner GitHub kontinuerlig.

Regelen er enkel: hemmeligheter skal aldri være i koden. Bruk i stedet:

- env-filer lokalt, som er lagt til i `.gitignore`
- GitHub Secrets for hemmeligheter som brukes i automatiserte prosesser

Legg til en secret: Settings → Secrets and variables → Actions → New repository secret. I en workflow bruker du den slik:

```
env:  
  API_KEY: ${{ secrets.MIN_API_NOKKEL }}
```

ADVARSEL

Hvis du ved et uhell pusher en hemmelighet, er det ikke nok å slette filen og pushe igjen. Hemmeligheten er i historikken. Se Del 8.4 for hva du gjør.

6.2 GitHub Actions: Roboten som gjør jobben for deg

GitHub Actions lar deg kjøre skript automatisk når noe skjer i repoet. Vanligste bruk er å kjøre tester automatisk når noen pusher kode.

En workflow er en YAML-fil i `.github/workflows/`. På neste side er et eksempel som kjører tester på hvert push:

```
# .github/workflows/test.yml

name: Kjør tester

on:
  push:
    branches: [main, "feature/**"]
  pull_request:
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: 20
      - run: npm install
      - run: npm test
```

Dette er bare begynnelsen. Actions kan brukes til å deploye nettsider, sende varsler, lage releases automatisk og mye mer.

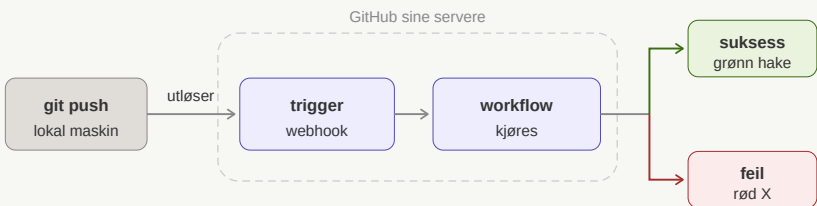


Fig. 12. GitHub Actions-flyt: et push utløser automatisk en webhook som starter en forhåndsdefinert workflow på GitHubs egne servere. Resultatet er enten en grønn hake (alle steg bestått) eller en rød X (noe feilet). Alt skjer uten at du gjør noe manuelt etter pushen

6.3 Branch Protection: Litt byråkrati som redder livet ditt

I et team vil du aldri at noen skal pushe direkte til main uten at koden er reviewet. Branch protection rules hindrer det.

Aktiver dem under Settings → Branches → Add rule:

- Require a pull request before merging. ingen direkte push til main
- Require status checks to pass – testene må være grønne før man kan merge

- Require approvals, minst én annen må ha godkjent PR-en. Selv om du jobber alene kan dette være nyttig for å tvinge deg selv til å gjøre ting riktig.

6.4 Dependabot: Automatiske sikkerhetsoppdateringer

Bibliotekene du bruker får regelmessige sikkerhetsoppdateringer. Dependabot er en robot som automatisk oppretter PR-er for slike oppdateringer, så du slipper å holde styr på det manuelt.

Aktiver Dependabot under Security → Dependabot i GitHub eller legg til en konfigurasjonsfil:

```
# .github/dependabot.yml

version: 2
updates:
  - package-ecosystem: npm
    directory: /
    schedule:
      interval: weekly
```

Øvingsoppgave — Del 6

Du skal nå sette opp din første automatiserte prosess. Fra nå av kan du la maskiner gjøre jobben. Opprett mappestrukturen GitHub Actions forventer:

```
mkdir -p .github/workflows
```

Lag workflow-filen:

```
cat > .github/workflows/hello.yml << 'EOF'
name: Hei fra Actions

on:
  push:
    branches: [main]

jobs:
  hei:
    runs-on: ubuntu-latest
    steps:
      - name: Si hei
        run: echo "Hei fra Actions!"
      - name: Vis dato og tid
        run: date
      - name: Vis hvem som trigget workflow-en
        run: echo "Utløst av ${{ github.actor }}"

EOF
```

Commit og push:

```
git add .github/workflows/hello.yml
git commit -m "chore: legg til første GitHub Actions workflow"
git push
```

Gå til Actions-fanen på GitHub. Du skal se workflow-en kjøre, først en oransje sirkel mens den venter, så grønn hake når den er ferdig. Klikk deg inn på kjøringen og åpne hvert steg. Du ser nøyaktig hva som ble kjørt og hva maskinen svarte.

```
gh run list
gh run view --log
```

Det du nettopp satte opp er prinsippet bak all moderne programvareutvikling. Tester, bygg og deploy kjører ikke manuelt, de kjører selv hver gang noen pusher kode.



Husk dette!

- Hemmeligheter hører ikke hjemme i koden. Én gang pushet til GitHub er de kompromittert — uavhengig av om du sletter dem etterpå.
- GitHub Actions kjører automatisk når noe skjer i repoet. Tester som kjører selv er tester som faktisk kjøres.
- Branch protection hindrer at noen pusher direkte til main. Sett det opp selv om du jobber alene.

Del 7: Vis det frem

7.1 README: Førsteintrykket ditt på nettet

README.md er det første folk ser når de besøker repoet ditt. En god README er forskjellen mellom et prosjekt som virker seriøst og et som ser ut som en hasteimport.

En god README inneholder som regel:

- Hva er dette prosjektet? (én setning, rett på sak)
- Skjerm bilde eller GIF av produktet i aksjon (vis, ikke bare fortell)
- Installasjonsinstruksjoner (steg for steg, ingen forkunnskaper antatt)
- Brukseksempel med kode
- Lenke til demo eller live-versjon

README-en skrives i Markdown. Det er et enkelt språk der # gir overskrift, ** gir fet tekst, og ``` gir kodeblokker.

```
# Prosjektnavn

Kort beskrivelse av hva prosjektet gjør.

## Installasjon

```bash
npm install
npm start
```

## Bruk

```javascript
const app = require("./app");
app.run();
```
```

Badges er små ikoner som viser build-status, versjon, lisens og lignende. Nettsiden shields.io genererer dem:

```
![Build]
(https://img.shields.io/github/actions/workflow/status/bruker/repo/test.yml)
![Version]
(https://img.shields.io/github/v/release/bruker/repo)
```

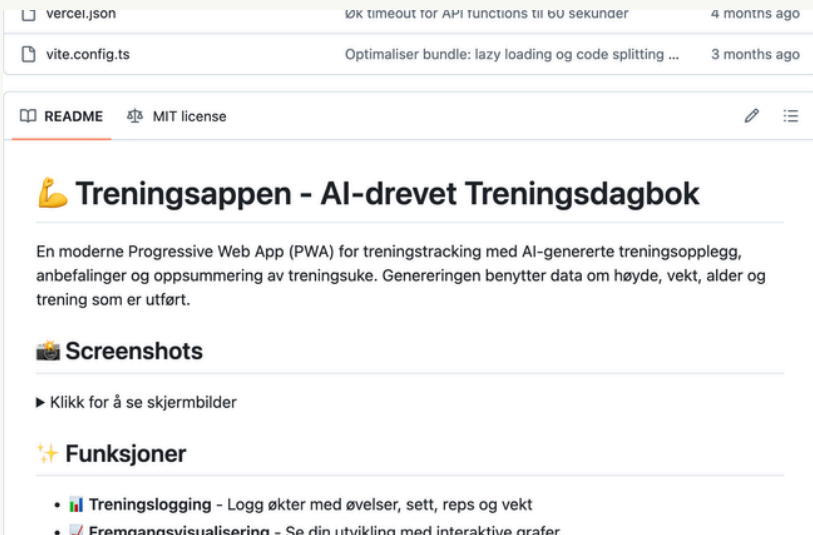


Fig. 13. Skjerm bilde fra repoet av Treningsappen. Oversiktlig beskrivelse av app, funksjoner og screenshots. [Se Repoet her](#) og andre apper [fra Lærerviv](#).

7.2 GitHub Pages: Fra repo til nettside på fem minutter

GitHub Pages lar deg hoste en statisk nettside rett fra et repo, gratis. Perfekt for porteføljesider, prosjektdokumentasjon eller en landing page.

Enkleste måte: lag en `index.html` i repoet, gå til Settings → Pages, og velg main-branchen som kilde. GitHub gir deg en URL som ser slik ut: brukernavn.github.io/reponavn.

Vil du ha din personlige side? Lag et repo som heter nøyaktig brukernavn.github.io. Alt du legger i dette repoet blir tilgjengelig på <https://brukernavn.github.io>.

Se skjermbildet på neste side for en visuelt kart.

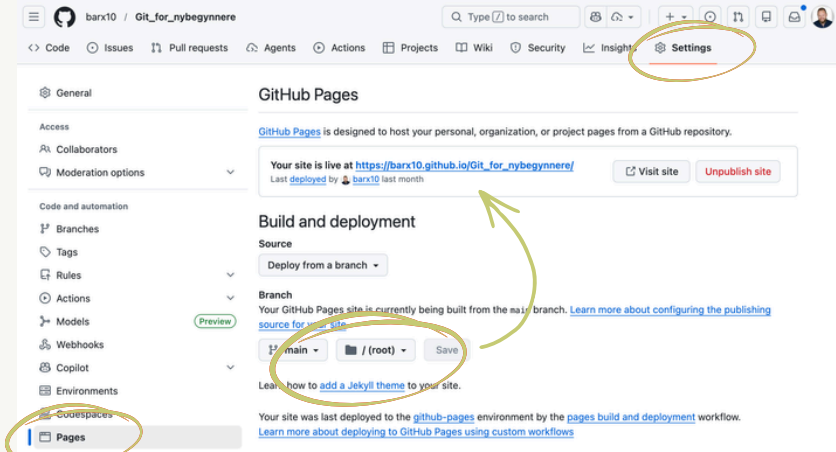


Fig. 14. I repoet: trykk Settings, Pages på venstre side og deretter velg main og root og trykk save. Da får du etter noen sekunder et nettsted med dedikert adresse.

TIPS

Vurder om du trenger å betale for en hosting-tjeneste. Hvis du kun har en enkel nettside i HTML med relativt enkle funksjoner, så kan du bruke Githubs Pages. Det er gratis og holdes oppdatert kontinuerlig rundt eventuelle endringer du foretar i kodebasen.

7.3 Tags og releases: Versjonering av prosjektet

Når prosjektet når et viktig punkt, for eksempel en stabil versjon eller en ny funksjon, kan du tagge det. En tag er som et permanent bokmerke i historikken.

Standard versjonering følger MAJOR.MINOR.PATCH (semantisk versjonering):

- MAJOR: Bakoverkompatibiliteten brytes (1.0.0 → 2.0.0)
- MINOR: Ny funksjonalitet, bakoverkompatibel (1.0.0 → 1.1.0)
- PATCH: Feilretting, bakoverkompatibel (1.0.0 → 1.0.1)

```
git tag v1.0.0          # Lag en tag på nåværende commit
git push origin v1.0.0 # Push taggen til GitHub
```

På GitHub kan du lage en release fra en tag: Releases → Draft a new release. Du kan legge ved filer, skrive en changelog (endringslogg) og merke den som "pre-release" eller "latest".



Fig. 15. Semantisk versjonering: tre tall med tre ulike betydninger. MAJOR bryter bakoverkompatibilitet, MINOR legger til noe nytt uten å ødelegge det gamle, PATCH retter feil.

Bakoverkompatibilitet

Bakoverkompatibilitet betyr at kode som fungerte mot versjon 1.x fortsatt fungerer mot 1.y uten endringer. Når MAJOR økes til 2.0.0 er det et signal om at noe er fjernet eller endret på en måte som kan knekke eksisterende bruk. MINOR og PATCH er trygge oppdateringer: du kan oppgradere uten å frykte at noe slutter å virke.

Øvingsoppgave — Del 7

Du skal nå publisere noe på internett og sette et offisielt versjonsstempel på det. Lag en enkel nettside:

```
cat > index.html << 'EOF'  
<!DOCTYPE html>  
<html lang="no">  
<head>  
  <meta charset="UTF-8">  
  <title>Min første GitHub Pages-side</title>  
</head>  
<body>  
  <h1>Hei fra GitHub Pages</h1>  
  <p>Denne siden hostes direkte fra et Git-repositorium.</p>  
</body>  
</html>  
EOF
```

Commit og push:

```
git add index.html
git commit -m "feat: legg til index.html"
git push
```

Gå til Settings → Pages i repoet ditt. Under Source velger du Deploy from a branch, velg main og klikk Save. GitHub bygger siden — det tar gjerne ett til to minutter. URLen du får er brukernavn.github.io/reponavn.

Nå setter du et versjonsstempel:

```
git tag v0.1.0
git push origin v0.1.0
```

Opprett en release fra taggen:

```
gh release create v0.1.0 --title "Versjon 0.1.0" --notes
"Første versjon. Enkel landingsside publisert via GitHub
Pages."
```

Verifiser at alt er på plass:

```
gh release list
```

Du har nå en offentlig nettside med en offisiell versjon i historikken. Alle som besøker repoet ditt kan se hva versjon 0.1.0 inneholdt, når den ble sluppet og hvem som slapp den. Det er slik profesjonelle prosjekter dokumenterer fremgang.



Husk dette!

- README-en er førsteinntrykket. En dårlig README gjør at folk ikke gidder å se nærmere på prosjektet ditt.
- GitHub Pages er gratis hosting for statiske sider. Det tar fem minutter å sette opp.
- Semantisk versjonering er en kontrakt med brukerne dine. MAJOR betyr at noe brakk, MINOR betyr noe nytt, PATCH betyr en fiks.

Del 8: Når alt brenner (Krisesenteret)

8.1 Merge Conflict: To endret samme linje

En merge-konflikt oppstår når to branches har endret nøyaktig samme linje på ulike måter, og Git ikke vet hvilken versjon som skal vinne.

De fleste editorer med Git-støtte (som VS Code, Sublime Text osv.) markerer konflikten tydelig med to alternativer: dine endringer (HEAD) og innkommende endringer. Velg den du vil beholde, eller kombiner begge manuelt. Lagre filen, så:

```
git add filnavn          # Marker konflikten som løst
git commit               # Fullfør mergen
```

```
<<<<<< HEAD (din versjon)
return "Hei, " + name + "!";
                                     din versjon
===== KONFLIKT =====
return `Hallo, ${name}!`;           innkommende
>>>>>> feature/ny-hilsen (innkommende)
```

din versjon (HEAD) innkommende endring konfliktlinje

Fig. 16. En merge-konflikt slik den ser ut i editoren: grønt viser din versjon (HEAD), blått viser innkommende endring fra den andre branchen. Den røde streken markerer selve konfliktpunktet der Git har gitt opp å velge.

Konfliktmarkører

Git setter inn konfliktmarkørene automatisk i filen. Alt mellom <<<<<<< og ===== er din versjon, alt mellom ===== og >>>>>>> er den innkommende.

Du løser konflikten ved å slette markørene og beholde det som skal stå, enten den ene versjonen, den andre, eller en kombinasjon. Deretter lagrer du, kjører git add på filen og committer.

8.2 Detached HEAD: Du er ingen steder og alle steder

"You are in detached HEAD state" er en av de mer forvirrende meldingene Git kan gi deg. Det betyr at du ikke er på en branch, du har sjekket ut en spesifikk commit direkte. Endringer du gjør her kan gå tapt.

Løsningen er enkel:

```
# Ta vare på arbeidet ved å lage en branch der du er:  
git checkout -b reddet-arbeid
```

```
# Eller bare gå tilbake til main:  
git checkout main
```

8.3 Angre på ulike måter

Det finnes mange måter å angre på, og de har ulike konsekvenser. Her er de vanligste:

```
# Angre staging (før commit):  
git restore --staged filnavn  
  
# Angre endringer i en fil (før staging):  
git restore filnavn  
  
# Angre siste commit, behold endringene i arbeidsmappen:  
git reset HEAD~1  
  
# Angre siste commit på en trygg måte (lager en ny  
"omgjørings-commit"):  
git revert HEAD
```



ADVARSEL

git reset --hard fjerner commits OG endringer permanent. Vær veldig forsiktig. Bruk git revert når du er usikker, det er tryggere fordi det ikke sletter historikk.

8.4 Du pushet noe du ikke skulle

API-nøkkelen er på GitHub. Hjertet banker. Her er hva du gjør i riktig rekkefølge:

1. Invalider nøkkelen umiddelbart i tjenestens kontrollpanel (AWS, Stripe, osv.). Koden er allerede kompromittert uavhengig av hva du gjør videre.
2. Fjern filen fra Git-historikken (installer git-filter-repo med pip install git-filter-repo):

```
git filter-repo --path .env --invert-paths
```

3. Force push den rensede historikken:

```
git push --force
```

4. Roter alle hemmeligheter som kan ha vært eksponert. Husk: å slette commiten er ikke nok alene. GitHub cacher innhold og andre kan ha klonet repoet. Invalider alltid først.

8.5 git reflog: Tidsmaskinens tidslinje

git reflog er det ultimate sikkerhetsnettet. Det viser alt du har gjort i det lokale repositoret, også commits du "slettet". Så lenge du ikke har ryddet opp manuelt, kan du finne tilbake til nærmest hva som helst.

```
git reflog                # Se alle operasjoner
git checkout abc1234      # Sjekk ut en gammel commit
git checkout -b reddet    # Lag en branch av den og ta vare på
arbeidet
```

TIPS

git reflog redder deg når ingenting annet fungerer. Det er godt å vite at dette sikkerhetsnettet finnes, du kan eksperimentere med tryggere samvittighet.

Øvingsoppgave — Del 8

Du skal nå gjøre en feil med vilje og rydde opp på den trygge måten. Lag en fil med en bevisst feil:

```
echo "Git er et sentralisert versjonskontrollsystem." >
feil.md
git add feil.md
git commit -m "docs: legg til forklaring av Git"
```

Det stemmer ikke. Git er distribuert, ikke sentralisert. Se hva historikken sier:

```
git log --oneline
```

Angre committen på den trygge måten:

```
git revert HEAD
```

Git åpner en editor med en ferdig commit-melding. Lagre og lukk den. Se nå på historikken igjen:

```
git log --oneline
```

Begge commits er der. Feilen og omgjøringen. Ingenting er slettet, Git la til en ny commit som reverserte endringen. Det er den viktige forskjellen fra git reset, som skriver om historikken. Sjekk at filen faktisk er borte:

```
cat feil.md
```

Til slutt, se på hva revert-committen faktisk inneholder:

```
git show HEAD
```

Du ser nøyaktig hva som ble reversert og hvorfor. Det er slik et profesjonelt prosjekt håndterer feil, ikke ved å late som de aldri skjedde, men ved å dokumentere at de ble rettet.



Husk dette!

- En merge conflict er ikke en katastrofe. Det er Git som ber deg ta en beslutning.
- git revert er den trygge måten å angre på. Den sletter ikke historikk, den legger til en ny commit som reverserer.
- git reflog er det ultimate sikkerhetsnettet. Nesten ingenting er permanent tapt så lenge du ikke har ryddet opp manuelt.
- Invalider alltid API-nøkler og passord umiddelbart hvis de havner på GitHub. Å slette committen er ikke nok alene

Oppslagverk

Oppsett

Kommando	Beskrivelse
<code>git config --global user.name "Navn"</code>	Sett navn på commits
<code>git config --global user.email "epost"</code>	Sett e-post på commits
<code>git config --list</code>	Vis all konfigurasjon
<code>git config --list grep user</code>	Vis bare navn og e-post
<code>ssh-keygen -t ed25519 -C "epost"</code>	Generer SSH-nøkkelpar
<code>ssh -T git@github.com</code>	Test SSH-tilkobling til GitHub
<code>gh auth login</code>	Koble GitHub CLI til kontoen din

Starte et prosjekt

Kommando	Beskrivelse
<code>git init</code>	Nytt lokalt repo i gjeldende mappe
<code>gh repo create navn --public</code>	Opprett nytt offentlig repo på GitHub
<code>gh repo create navn --private</code>	Opprett privat repo på GitHub
<code>git clone git@github.com:bruker/repo.git</code>	Klon eksisterende repo via SSH
<code>gh repo clone bruker/repo</code>	Klon via GitHub CLI
<code>gh repo view --web</code>	Åpne repoet i nettleseren

Den daglige flyten

Kommando	Beskrivelse
git status	Hva er endret? Kjør denne først, alltid
git add filnavn	Klargjør én spesifikk fil
git add .	Klargjør alle endringer
git commit -m "melding"	Ta et fotografi med beskrivelse
git push	Send commits til GitHub
git pull	Hent andres commits fra GitHub
git stash	Legg halvferdig arbeid i skuffen
git stash pop	Hent arbeidet ut av skuffen
git stash list	Se hva som ligger i skuffen

Starte et prosjekt

Kommando	Beskrivelse
git log	Hele historikken
git log --oneline	Kompakt, én commit per linje
git log --oneline --graph	Med visuell branch-struktur
git log --author="Navn"	Bare commits fra én person
git diff	Hva er endret men ikke klargjort?
git diff --staged	Hva er klargjort for neste commit?
git show abc1234	Vis innholdet i én spesifikk commit
git reflog	Alt lokalt, også "slettede" commits

Branches

Kommando	Beskrivelse
git branch	List alle branches
git checkout -b navn	Lag ny branch og bytt til den
git checkout navn	Bytt til eksisterende branch
git merge navn	Slå sammen branch
git branch -d navn	Slett branch som er merget
git branch -D navn	Slett branch uansett (tvang)
git push -u origin navn	Push ny branch til GitHub

Navnekonvensjoner

- feature/Ny funksjonalitet
- fix/Feilretting
- hotfix/Kritisk feil som må fikses nå
- docs/Dokumentasjon
- refactor/Rydding uten ny funksjonalitet
- chore/ Vedlikehold og avhengigheter

Merge og Rebase

Kommando	Beskrivelse
git merge feature/navn	Flett branch, bevar historikk
git rebase main	Spill av commits på toppen av main
git merge --abort	Avbryt en merge som gikk galt
git rebase --abort	Avbryt en rebase som gikk galt

Angre

Kommando	Beskrivelse
<code>git restore filnavn</code>	Angre endringer i fil (før staging)
<code>git restore --staged filnavn</code>	Fjern fil fra staging, behold endringer
<code>git revert HEAD</code>	Angre siste commit — lager ny omgjøringscommit
<code>git revert abc1234</code>	Angre en spesifikk commit
<code>git reset HEAD~1</code>	Angre siste commit, behold endringer i arbeidsmapp
<code>git reset --hard HEAD~1</code>	⚠ Angre siste commit, slett endringer permanent

Angre

Kommando	Beskrivelse
<code>git worktree add ../mappe branch</code>	Åpne branch i ny mappe
<code>git worktree list</code>	Se alle aktive worktrees
<code>git worktree remove ../mappe</code>	Fjern worktree når du er ferdig

Hotfix mens du jobber på feature

`git worktree add ../hotfix main`

åpne ../hotfix i editor, fiks feilen, commit og push

`git worktree remove ../hotfix`

- # tilbake til feature, ingenting er forstyrret

Fork og Pull Request

Kommando	Beskrivelse
<code>gh repo fork bruker/repo --clone</code>	Fork og klon i én kommando
<code>git remote add upstream git@...</code>	Koble til originalrepoet
<code>git pull upstream main</code>	Hent oppdateringer fra originalrepoet
<code>gh pr create</code>	Lag Pull Request interaktivt
<code>gh pr list</code>	Se åpne Pull Requests
<code>gh pr checkout 42</code>	Sjekk ut PR nummer 42 lokalt
<code>gh pr merge 42</code>	Merge PR nummer 42

Bidra til andres prosjekt:

```
gh repo fork bruker/repo --clone
cd repo
git checkout -b fix/beskrivelse
git add .
git commit -m "fix: beskriv hva du fikset"
git push -u origin fix/beskrivelse
gh pr create
```

Issues

Kommando	Beskrivelse
<code>gh issue list</code>	Se alle åpne issues
<code>gh issue create</code>	Lag ny issue interaktivt
<code>gh issue view 42</code>	Vis issue nummer 42
<code>gh issue close 42</code>	Lukk issue nummer 42

Tags og Releases

Kommando	Beskrivelse
<code>git tag v1.0.0</code>	Lag tag på gjeldende commit
<code>git tag -a v1.0.0 -m "beskrivelse"</code>	Lag annotert tag med melding
<code>git push origin v1.0.0</code>	Push én spesifikk tag
<code>git push --tags</code>	Push alle tags
<code>git tag</code>	List alle tags
<code>gh release create v1.0.0</code>	Lag release fra tag
<code>gh release list</code>	Se alle releases

Type	Eksempel	Når
MAJOR	1.0.0 → 2.0.0	Bakoverkompatibilitet brytes
MINOR	1.0.0 → 1.1.0	Ny funksjonalitet, kompatibel
PATCH	1.0.0 → 1.0.1	Feilretting, kompatibel

GitHub Actions

Kommando	Beskrivelse
<code>gh run list</code>	Se siste workflow-kjøringer
<code>gh run view</code>	Vis detaljer om siste kjøring
<code>gh run view --log</code>	Vis full logg
<code>gh workflow list</code>	Se alle workflows i repoet

Commit-meldinger (Conventional Commits)

Prefiks	Bruk
feat:	Ny funksjonalitet
fix:	Feilretting
docs:	Dokumentasjon
refactor:	Rydding uten ny funksjonalitet
chore:	Vedlikehold og avhengigheter
test:	Tester
style:	Formatering, ingen logikkendring

Krisesenteret

Pushet en hemmelighet:

1. Invalider nøkkelen/passordet umiddelbart i tjenestens kontrollpanel
2. `git filter-repo --path .env --invert-paths`
3. `git push --force`
4. Roter alle eksponerte hemmeligheter

Merge conflict:

1. Åpne filen, finn konfliktmarkørene
2. Velg hvilken versjon som skal beholdes
3. `git add filnavn`
4. `git commit`

Mistet en commit:

```
git reflog  
git checkout abc1234  
git checkout -b reddet
```

Feil branch:

```
git stash  
git checkout riktig-branch  
git stash pop
```

Detached HEAD:

```
git checkout -b reddet-arbeid
```

Etterord

Du begynte med en mappe full av filer ingen visste var nyest. Du ender med verktøyene en profesjonell utvikler bruker hver dag. Det er ikke lite.

Git er ikke vanskelig når du forstår hva det faktisk gjør. Det tar fotografier. Det lar deg eksperimentere uten å ødelegge noe. Det gir deg muligheten til å jobbe med andre uten at alt blir kaos. Det meste som virker mystisk i begynnelsen — branches, merging, HEAD, rebase — er egentlig bare varianter av det samme grunnprinsippet: historikken er din, og du bestemmer hva som skjer med den.

Det som skiller en nybegynner fra en erfaren utvikler er sjelden kunnskap om obskure kommandoer. Det er vanen. git status før alt annet. Commit-meldinger som faktisk beskriver hva og hvorfor. Branches for alt som kan gå galt. Det er disse vanene du nå har forutsetningene for å bygge.

Neste steg er ikke å lese mer. Det er å bruke det. Ta et prosjekt du jobber med, gjør det til et Git-repo, og begynn å committe. Du vil gjøre feil. Du vil havne i detached HEAD state og lure på hva som skjedde. Du vil en dag se på git reflog og finne noe du trodde var borte for alltid. Det er da du skjønner at du har lært noe ordentlig.

Lykke til, og husk at git revert eksisterer.

© 2026 KENNETH BAREKSTEN

Denne boka er lisensiert under Creative Commons (CC BY 4.0). Det betyr at du kan dele, bruke og tilpasse innholdet, også kommersielt, så lenge du krediterer forfatteren.

Kontakt: kenneth@laererliv.no
Nettside: laererliv.no

