TTK4235: Versjonskontroll via git

Vår 2018

1 Intro

Versjonskontroll er en måte å ta "bilder" av en fil, slik at du kan se hvordan den har endret seg, og når endringene ble gjort. Programmet `git` er et veldig utbredt verktøy som lar deg gjøre nettopp dette. I tillegg er det også veldig gunstig om man jobber flere i lag på de samme filene.

Denne øvingen er ment som en introduksjon til praktisk bruk av `git`. For å illustrere de mest brukte delene av `git` skal vi skrive litt enkel C-kode, og versjonskontrollere denne.

1.1 Oppsett

Først og fremst må vi ha `git`. Datamaskinene på Sanntidssalen har allerede `git` installert, men siden `git` er et veldig nyttig verktøy, er det anbefalt å skaffe det selv også. Du kan få tak i `git` via git-scm.com, eller via en pakkebehandler om operativsystemet ditt har en[1].

Vi trenger også en C-kompilator for å demonstrere versjonskontroll av kompilerte filer. Denne oppgaveteksten kommer til å bruke `gcc`, men en hvilken som helst kompilator vil fungere. Datamaskinene på Sanntidssalen har allerede `gcc` installert.

2 Oppgave

Åpne en terminal og skriv inn `git --version`. Avhengig av hvilken versjon av `git` som er installert, vil det komme opp noe i duren av "git version 2.16.0". Dette er en enkel test for å se om `git` er riktig installert og ligger i `PATH`-variabelen til operativsystemet.

2.1 Oppsett

Før vi kan bruke `git`, må vi gjøre noen enkle konfigurasjoner. Først og fremst må `git` vite hvem du er. Dette må bare gjøres én gang, og består av

[1] apt, yum, pacman etc for Linux. Homebrew for mac.

to kommandoer:

```
git config --global user.name "Reodor Felgen"
git config --global user.email "felgen@klypa.ntnu.no"
```

Når dere legger ved flagget `--global`, så vil `git` lagre navn og epost i filen `~/.gitconfig`. Om dere jobber på en datamaskin andre bruker - slik som i heisprosjektet, kan det være lurt å konfigurere `git` lokalt. Det gjøres fra et allerede opprettet `git`-repository, ved å kalle de samme kommandoene uten `--global`-flagget.

For ekstra brukervennlighet, har `git` muligheten til å definere aliaser for lange kommandoer som ofte brukes. For eksempel brukes kommandoen `git checkout` ofte, så mange liker å aliase denne til `git co`. Dette er helt opp til dere. Måten man oppretter et alias på, er slik:

```
git config --global alias.lg "log --all --oneline --graph
↪  --decorate"
```

Dette vil lage aliaset `git lg`, som betyr det samme som `git log --all --oneline --graph --decorate`. Akkurat dette aliaset er veldig hjelpsomt, og vi vil bruke det i denne øvingen.

2.2 `git init`

For å bruke `git`, må vi først lage et såkalt repository - eller en *oppbevaringsmappe*. Sett nå at vi ønsker å skrive kode i en mappe kalt "demo", og at vi ønsker at `git` skal følge med koden vi legger der. For å gjøre dette, gjør vi følgende fra terminalen:

```
mkdir demo
cd demo
git init
```

`mkdir` (*make directory*) vil opprette mappen "demo" for oss, og `cd` (*change directory*) vil flytte oss inn i den. Kommandoen `git init` vil opprette en gjemt mappe med navn ".git" inne i "demo", som `git` vil bruke for å holde styr på de andre filene i mappen.

2.3 `git status`, `git add`, `git commit`

Dere har nå en tom `git`-mappe. Kall kommandoen `git status`. Hvis dere ikke har gjort noen endringer i mappen så langt, vil dere få tilbake en melding som:

```
On branch master
```

```
No commits yet

nothing to commit (create/copy files and use "git add" to
↪  track)
```

Stort sett er `git` veldig behjelpelig med å fortelle hva som foregår. Om dere lurer på hva som skjer, er utskriften fra `git` oftest et godt svar.

Dette er vel og bra, men så langt har ikke `git` gjort stort for oss. Opprett filen "main.c" i mappen "demo", og skriv inn det følgende med en hvilken som helst tekstbehandler (sublime, gedit, atom, vim er noen alternativer).

```c
#include <stdio.h>

int main(){
        return 0;
}
```

Lagre denne filen, og kjør `git status` på nytt. Dere vil nå se noe i duren av:

```
On branch master

No commits yet

Untracked files: [...]

        main.c

[...] (use "git add" to track)
```

Denne utskriften forteller oss at vi nå har en ny fil i "demo"-mappen ved navn "main.c", som `git` foreløpig ikke bryr seg om. Kall nå `git add main.c`, etterfulgt av `git status`. Nå vil dere se

```
[...]
Changes to be commited:
        (use "git rm --cached <file>..." to unstage)

        new file: main.c
```

Dette betyr at `git` har lagt til "main.c" i sitt *staging area*, som er stedet før `git` "tar et bilde" av mappen.

Kjør nå `git commit -m "added main.c"`, etterfulgt av `git lg`[2]. Dere vil nå se

[2]Dette er aliaset vi opprettet i seksjon 2.1

```
* 1c9576f (HEAD -> master) added main.c
```

Det første vi ser på denne linjen er en stjerne. Denne stjernen representerer et "bilde" som `git` har tatt for oss. De sju heksadesimale tallene som følger etter er et utsnitt av en hash på 40 bokstaver, som `git` bruker for å identifisere "bilder", eller *commits*.

I parentesen står det `(HEAD -> master)`, som betyr at hodepekeren til `git` peker til akkurat dette "bildet", som befinner seg på grenen "master".

Til slutt står det "`added main.c`", som er *commit*-meldingen vi ga dette bildet, for å beskrive hva vi har gjort.

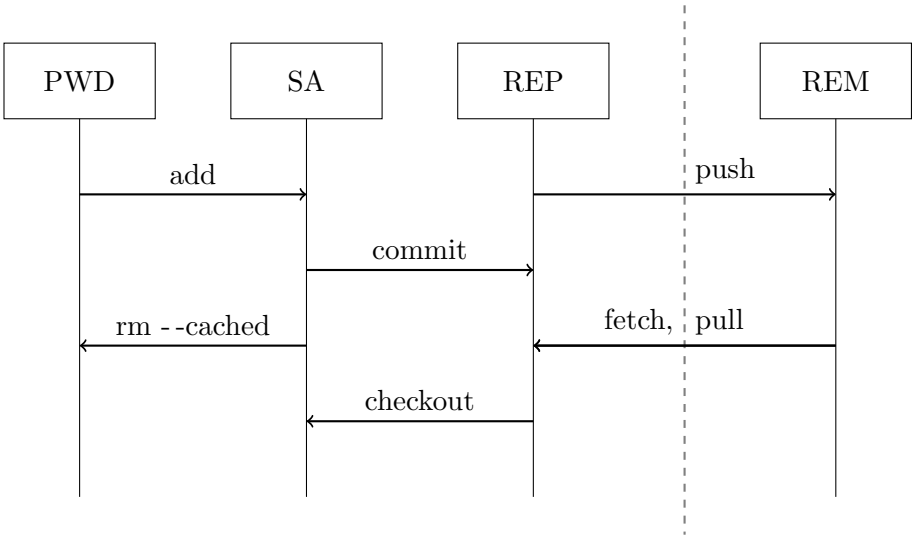

Figure 1: Arbeidsflyt i `git`.

2.4 Terminologi og `gits` indre

Arbeidsflyten i `git` er illustrert i figur 1. Terminologien er som følger:

PWD "Current working directory", dette er mappen som `git` holder styr på.

SA "Staging area", her legges filer `git` skal "ta bilde av", før de legges til i historikken.

REP "Repository", dette er all historien `git` kjenner til. Bilder som er tatt av tidligere versjoner av filer legges til her, som en ny stjerne i en graf som representerer alt som har skjedd hittil.

REM "Remote", dette er stort sett en ekstern server (men kan være en annen lokal mappe), dit `git` vil dytte lokale endringer, og ta endringer gjort av andre fra.

Om dere lurer på hvorfor "current working directory" forkortes som "PWD", kommer dette av kommandoen `pwd` - som vil skrive ut nåværende arbeidsmappe.

Til nå har vi ikke satt opp noen ekstern server, så vi har ikke vært borte i "REM". Vi kommer dit senere, men først skal vi bygge videre på den lokale `git`grafen vår.

2.5 `git diff`, `git checkout`

Endre mainfilen til å inneholde dette:

```c
#include <stdio.h>

int main(){
        printf("Hello world\n");
        return 0;
}
```

Kjør deretter `git status`. Dere vil nå se

```
Changes not staged for commit: [...]

        modified: main.c

no changes added to commit [...]
```

Dette forteller oss at `git` vet at mainfilen har endret seg, men fordi vi ikke har lagt den til i *staging area*, vil ikke `git` ta et bilde av den.

Nå vet vi jo egentlig hva som har endret seg, men kall allikevel `git diff main.c`. `git` vil da skrive ut

```
@@ -1,5 + 1,6 @@
 #include <stdio.h>

 int main(){
+        printf("Hello world\n");
         return 0;
 }
```

Som dere ser, forteller `git` oss hva som har skjedd; et plusstegn representerer en linje som har blitt lagt til, mens et minustegn representerer en linje som har blitt tatt bort. Dette er en grei kommando å ha om dere endrer mange filer på en gang, og glemmer litt hva dere har gjort før dere "tar et nytt bilde".

Kjør nå `git add main.c`, etterfulgt av `git commit -m "classic example code"`, etterfulgt av `git lg`. Nå vil dere se

```
* f330827 (HEAD -> master) classic example code
* 1c9576f added main.c
```

Dette betyr at vi nettopp tok et nytt bilde av mainfilen, og at vi la denne til øverst i "historikktreet". Den "gamle" koden som ikke gjorde noe ligger fortsatt i `git`s minne, men grenen kalt "master" (og også vår hodepeker) peker til den nye `printf`-koden vi akkurat skrev.

Med `git` er det mulig å ta en titt på eldre kode, ved å hoppe tilbake i

historikkgrafen. Kjør nå `git checkout 1c9576f`[3]. Dere vil nå få en melding som sier at dere er i "detached HEAD state". Dette høres skummelt ut, men betyr ikke annet enn at hodepekeren ikke er på en gren akkurat nå. Dette illustreres lett ved å kalle `git lg`:

```
* f330827 (master) classic example code
* 1c9576f (HEAD) added main.c
```

Dette forteller oss at hodepekeren peker til den "gamle" koden, mens "master"-grenen er på den nyeste koden. Om dere nå åpner mainfilen på nytt vil dere se at linjen med `printf` er borte. Slik kan man inspisere tidligere kode, uten å måtte ha "utkommentert" kode innblandet i en ny fil.

Kall `git checkout master` for å komme tilbake til den nye koden.

2.6 `git help`

Ingen går rundt og husker alle mulige flagg hver enkelt `git`kommando tar. Derfor har `git` en innebygd hjelpefunksjon som er grei å bruke om man er i tvil. Kall `git help commit`, for å ta opp hjelpesiden til `git commit`. Finn ut hva flagget `-s` gjør.

2.7 `git branch`, `git merge`

Når dere jobber flere på samme kodebase, kommer versjonskonflikter til å oppstå. Slik er det bare. Heldigvis er dette et problem skaperene[4] av `git` var vel kjent med. Derfor er `git` godt utrustet til å hjelpe dere når en konflikt oppstår.

Kall først `git branch other`, etterfulgt av `git checkout other`. Dette vil lage en ny gren, kalt "other", og hoppe til den. Denne grenen skal simulere at dere er to som jobber på samme kode.
Det er forresten så vanlig å opprette en gren og skifte til den med en gang at `git` har en snarvei for dette, nemlig `git checkout -b <grennavn>`.

Om dere nå kaller `git lg`, vil dere se følgende:

```
* f330827 (HEAD -> other, master) classic example code
* 1c9576f added main.c
```

Nå har vi to grener som begge peker til den nyeste koden, men vi er på grenen "other", og ikke "master". Endre deretter mainfilen slik:

[3] Hashsummen deres kan være annerledes enn oppgavetekstens. Kjør `git lg` for å se.
[4] Hovedsakelig Linus Torvalds; også skaperen av Linux.

```c
#include <stdio.h>

int main(){
        printf("Hello world\n");
        printf("...and Mars\n");
        return 0;
}
```

Kjør så `git add main.c`, og `git commit -m "greet mars as well"`. Kjør `git lg`, og dere får:

```
* d6d26ad (HEAD -> other) greet mars as well
* f330827 (master) classic example code
* 1c9576f added main.c
```

Her ser vi at grenen "master" fortsatt ligger på koden med "Hello world", mens grenen "other" ligger på koden med "...and Mars".

Sett nå at dere er to som jobber i par, og at du har skrevet "world"-versjonen, mens partneren din har skrevet "mars"-versjonen. I mellomtiden har du skrevet videre på din kode, som vi nå skal simulere. Kall `git checkout master`, slik at du er på "din" gren. Endre deretter mainfilen slik:

```c
#include <stdio.h>

int main(){
        printf("Hello world\n");

        if(1 > 0){
                return 1;
        }

        return 0;
}
```

Kjør så `git add main.c`, og `git commit -m "assert truth"`. Kall til slutt `git lg`. Nå skal historikkgrafen se slik ut:

```
* af4c17f (HEAD -> master) assert truth
| * 57b9e8c (other) greet mars as well
|/
* f330827 classic example code
* 1c9576f added main.c
```

Dette representerer at du og partneren var enige på *commiten* med hash "f330827", men at dere deretter har divergert til hver deres gren. Nå ønsker dere enighet i hva som skal være i kodebasen, så du kaller `git merge other`,

for å flette din "master"-gren inn i "other"-grenen. Øyeblikkelig vil `git` klage med en slik feilmelding:

```
Auto-merging main.
CONFLICT (content): Merge conflict in main.c
Automatic merge failed; fix conflicts and then commit the
↪  result
```

Kall så `git status`, som vil fortelle følgende:

```
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   main.c

no changes added to commit [...]
```

Her forteller `git status` oss nøyaktig hva som foregår:

1. Vi holder på med en sammenslåing, men `git` kunne ikke fullføre, fordi både "master" og "other" har endret på mainfilen.
2. Vi kan fikse konflikten og kjøre `git commit` for å manuelt fullføre sammenslåingen.
3. Vi kan kjøre `git merge --abort`, om vi ikke lenger vil slå sammen grenene.

Vi skal rydde opp i koden, og fullføre sammenslåingen. Åpne mainfilen på nytt, og dere vil se dette:

```
#include <stdio.h>

int main(){
        printf("Hello world\n");
<<<<<<< HEAD

        if(1 > 0){
                return 1;
        }

=======
```

```
        printf("...and Mars\n");
>>>>>>> other
        return 0;
}
```

Her vil dere se at `git` automatisk har satt inn konfliktmarkører der koden var forskjellig. Alt mellom `<<<<<<< HEAD` og `=======` var på "master"-grenen, mens alt som ligger mellom `=======` og `>>>>>>> other` lå på "other"-grenen.

For å fortelle `git` at konfliktene er tatt hånd om, må du redigere filen slik den skal være, og så legge den til i `git` på vanlig måte. Endre mainfilen slik:

```c
#include <stdio.h>

int main(){
        printf("Hello world\n");
        printf("...and Mars\n");

        if(1 > 0){
                return 1;
        }

        return 0;
}
```

For å legge den til i `git`, kjører vi så `git add main.c`, etterfulgt av `git commit -m "merged work"`. Kall `git lg`, og se resultatet:

```
*   e3192ad (HEAD -> master) merged work
|\
| * 57b9e8c (other) greet mars as well
* | af4c17f assert truth
|/
* f330827 classic example code
* 1c9576f added main.c
```

Altså er kodebasene nå slått sammen, men som dere ser, vil ikke "other"-grenen automatisk trekkes etter. Om dere ikke planlegger å gjøre den neste frivillige oppgaven, kan dere fjerne "other"-grenen ved å kalle `git branch -d other`.

2.8 Frivillig: `git rebase`

Som dere så, gjorde `git merge` akkurat det vi ville - vi fikk slått sammen kodebasene, og samlet den oppdaterte koden inn i en ny *commit*. Allikevel vil `git` huske på splitten vi hadde. Altså vil `git` skrive dette i historieboka:

1. Var enige ved "classic code example".
2. Divergerte i to grener en stund.
3. Slo sammen grenene.
4. Lagde ny *commit* "merged work".

Dette er helt greit, men om dere er mange som jobber på samme kode, kan det fort bli veldig mange slike avstikkere som kommer inn igjen. Om dere da må bla tilbake i gammel kode, kan det ta lang tid å finne tilbake til riktig gren. Derfor har `git` en kommando kalt `rebase`, som lar oss trikse litt med historikken, slik at det *ser* ut som om endringene kom i sekvens.

Om vi bruker denne teknikken, vil vi miste litt informasjon, fordi `git` ikke vil huske at de to grenene divergerte, men til gjengjeld kan det bli veldig mye enklere å finne frem senere. Vi skal demonstrere teknikken her.

Kall først `git reset --hard af4c17f`. Da vil vi gå tilbake i tid til før vi kjørte sammenslåingen. Om dere kaller `git lg`, vil dere nå se:

```
* af4c17f (HEAD -> master) assert truth
| * 57b9e8c (other) greet mars as well
|/
* f330827 classic example code
* 1c9576f added main.c
```

Kjør deretter `git rebase other`. Dere vil få en tilsvarende feilmelding som i sted, men denne gangen vil den være litt mer utdypende. Åpne mainfilen på nytt, og endre den slik:

```
#include <stdio.h>

int main(){
        printf("Hello world\n");
        printf("...and Mars\n");

        if(1 > 0){
                return 1;
        }

        return 0;
}
```

Kall nå `git add main.c`, og så `git rebase --continue`. Dere vil nå få meldingen "`Applying: assert truth`". Hvis dere nå kjører `git lg`, vil dere se en mye enklere historikk enn tidligere:

```
* f8cd2e1 (HEAD -> master) assert truth
* 57b9e8c (other) greet mars as well
* f330827 classic example code
* 1c9576f added main.c
```

Som dere ser, er historikken enklere, men ikke helt nøyaktig lenger. Vi vet jo at vi hadde to grener som divergerte, men vi har nå fortalt `git` at dette ikke skal skrives ned i historien.

Det er ingenting magisk ved å kjøre en *rebase* fremfor en *merge*, men denne taktikken kan være litt skumlere: For det første går informasjon tapt. Vær sikker på at informasjonen som kastes bort ikke er til nytte uansett.
For det andre: Om du har dyttet arbeidet ditt til en server som andre har tilgang på, bør du virkelig *ikke* kjøre en *rebase*. Om du gjør det, vil alle de du samarbeider med måtte kopiere *din* kode, fordi du ødelegger deres "grafer" i prosessen. Derfor anbefales det ikke at dere bruker `git rebase` under heisprosjektet, med mindre dere har lest mer om taktikken (`git help rebase` eller git-scm.com).

2.9 Gjemt historie

Om dere gjorde den frivillige oppgaven, kall først `git checkout master`, og så `git reset --hard e3192ad`, slik at historikken ser slik ut igjen:

```
*   e3192ad (HEAD -> master) merged work
|\
| * 57b9e8c (other) greet mars as well
* | af4c17f assert truth
|/
* f330827 classic example code
* 1c9576f added main.c
```

Dette illustrerer forresten en annen kjekk greie ved `git`; ingen ferdige *commits* slettes helt før de samles inn av `git` sin søppelhåndterer - som tar 90 dager, så lenge dere ikke endrer standardverdien. En *commit* vil slettes når denne perioden er utløpt hvis det ikke er noen grener som inkluderer denne *commiten*.
Forresten: Selv om `git` ikke har slettet en *commit*, vil den ikke vises via `git lg`. Dette er fordi *commits* uten en gren regnes som "*dangling*", og `git` vil i utgangspunktet ikke vise disse. Kall `git fsck --lost-found` for å se alle *dangling commits* - eller `git reflog` for å se alt du har gjort i det siste.

2.10 `git remote`, `git push`, `git fetch`

Dette er det siste dere trenger for å være greit gode med `git`. Når dere er ferdige med denne deloppgaven, burde dere stort sett kunne bruke `git`

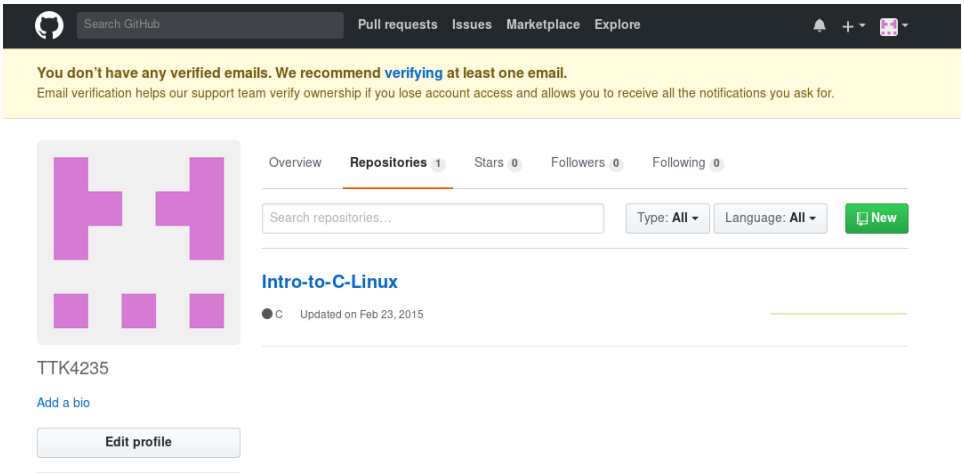

Figure 2: GitHub hjemmeskjerm.

gjennom heisprosjektet og senere uten for mye problemer.

Å ha `git` fungerende lokalt på én datamaskin er kjekt det, men om man skal samarbeide med flere, er det greit å kunne dytte ting til en felles server.

NTNU har sin egen `git`server (git.it.ntnu.no), men av erfaring velger de fleste å bruke GitHub. GitHub er den mest kjente `git`serveren, og har en rekke fordeler om man lager en bruker med en epost tilknyttet et universitet - slik som studadressene deres. GitHub er fortsatt gratis uten en universitet-sepostadresse, men man kan blant annet ikke opprette private *repositories*. Dere står fritt til å velge hvilken server dere bruker (eller til og med en lokal mappe), men denne øvingen vil demonstrere GitHub.

Når dere logger på GitHub vil dere se et bilde som i figur 2. Trykk "New" oppe i høyre hjørne. Dette vil ta dere til en skjerm som i figure 3. Når dere har skrevet inn et navn, og trykket på "Create repository", vil dere kunne starte å bruke GitHub. Det første dere vil se er en velkomstskjerm, som i figur 4. Ignorér denne skjermen enn så lenge.

Fra terminalen kaller dere nå `git remote add origin https://github.com/TTK4235/demo`[5]. Dette vil fortelle `git` at "github.com/TTK4235/demo" skal legges til som et eksternt mål, under navnet "origin". Det er ikke noe spesielt ved navnet "origin", men det er det navnet `git` bruker som standard.

[5]Bytt ut URLen med deres URL.

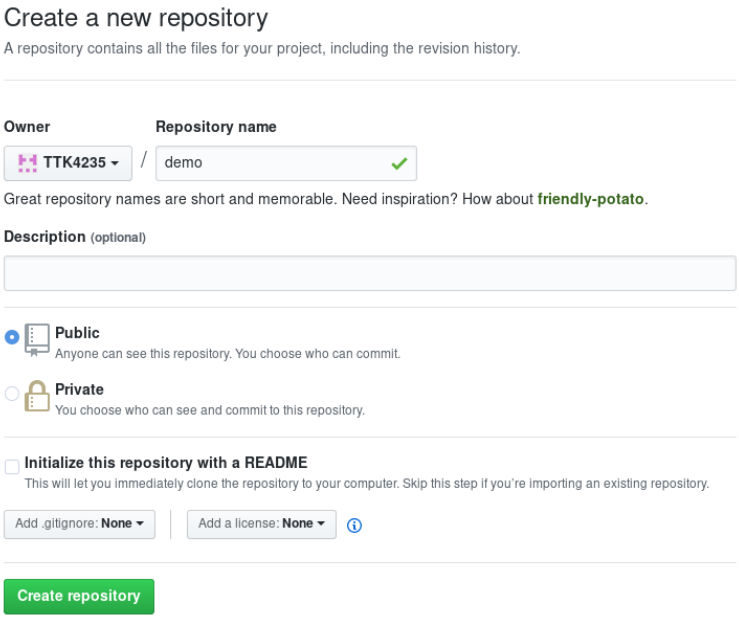

Figure 3: Opprett nytt *repository*.

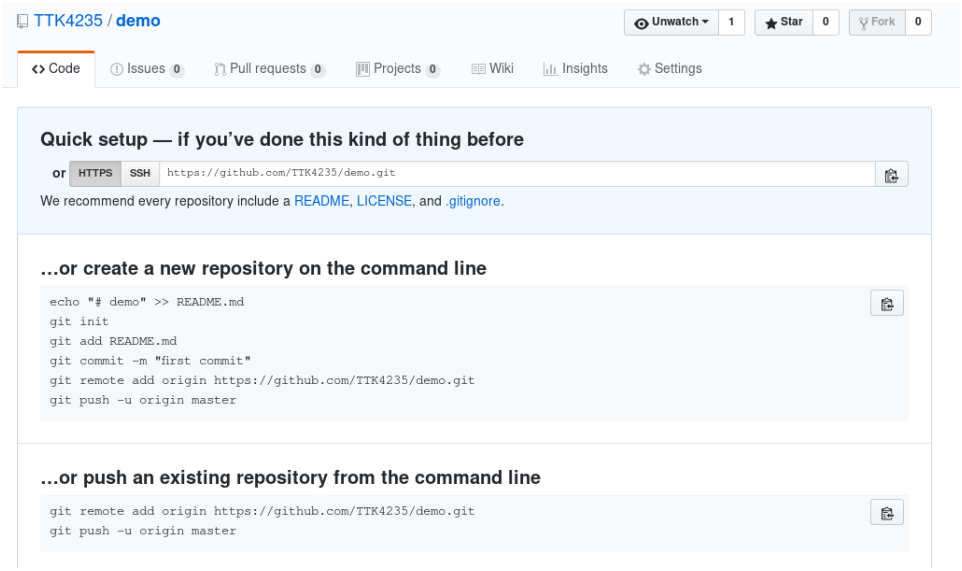

Figure 4: Nyopprettet *repository*.

Deretter kjører dere `git push --set-upstream origin master`. Dere vil bli spurt om brukernavn og passord i terminalen, og når dette er skrevet inn, blir de lokale filene dyttet til serveren. Når dere nå kaller `git lg` vil dere se følgende:

```
*   e3192ad (HEAD -> master, origin/master) merged work
|\
| * 57b9e8c (other) greet mars as well
* | af4c17f assert truth
|/
* f330827 classic example code
* 1c9576f added main.c
```

Den ekstra "`origin/master`" i utskriften forteller oss at GitHub er oppdatert med samme kode som "master"-grenen. Om dere oppdaterer nettleseren, vil dere se det samme der. Det er kun nødvendig å kjøre `git push --set-upstream origin master` første gang GitHub informeres om "master"-grenen. Etter dette er det nok å skrive simpelthen `git push`; eller `git push origin master` om man ønsker å være spesifikk.

Når dere jobber flere på samme kodebase, er det enklest å legge de andre på prosjektet inn som *collaborators*. Dette gjøres fra "Settings", og så "Collaborators".

Når dere så har jobbet hver for dere en stund, blir det etterhvert nødvendig å spørre GitHub om endringer fra andre prosjektdeltakere. Bruk `git fetch` for dette.

Det finnes også en annen kommando, `git pull`, som ofte brukes. Denne kommandoen er simpelthen en snarvei for `git fetch`, etterfulgt av `git merge FETCH_HEAD`.

2.11 .gitignore

Som ekstra "pynt på verket", skal vi demonstrere hvordan `git` kan fortelles at visse filer skal ignoreres. Kompilér først mainfilen, eksempelvis ved å kalle `gcc main.c -o a.out`. Dette vil produsere en kjørbar fil, og hvis dere nå kaller `git status` vil dere se at `git` har lagt merke til den nye filen; a.out.

Prøv å legge denne til ved å kalle `git add a.out`, etterfulgt av `git commit -m "traking elf file"`. Endre deretter mainfilen slik:

```c
#include <stdio.h>

int main(){
```

```
        printf("Hello world\n");
        printf("...and Jupiter\n");

        if(1 > 0){
                return 1;
        }

        return 0;
}
```

Det eneste vi har gjort her, er å bytte ut "Mars" med "Jupiter". Kall `git diff main.c`, og dere vil se følgende:

```
diff --git a/main.c b/main.c
index 5c2c469..e087c31 100644
--- a/main.c
+++ b/main.c
@@ -2,7 +2,7 @@

 int main(){
        printf("Hello world\n");
-       printf("...and Mars\n");
+       printf("...and Jupiter\n");

        if(1 > 0){
                return 1;
```

Det er altså klart at vi kun har endret en enkelt linje fra utskriften. Kall nå `gcc main.c -o a.out` igjen, etterfulgt av `git diff a.out`. Det eneste dere vil få er dette:

```
diff --git a/a.out b/a.out
index 8b5c00f..a2313a8 100755
Binary files a/a.out and b/a.out differ
```

Dette er fordi den kompilerte filen kan endre seg ganske mye, selv om en enkelt linje forandres. Prøv forresten å åpne "a.out" i en tekstbehandler. Det vil være noen lesbare snutter inne i filen, men stort sett vil den bare være tøv.

I de fleste tilfeller gir det ikke mening å *tracke* en kompilert fil via `git`, nettopp fordi filen ikke er lesbar for folk. Derfor har `git` en mekanisme for å ignorere enkelte filer.

Dette gjøres ved å opprette filen ".gitignore" i `git`mappen, og skrive inn navn på filer som `git` skal ignorere i denne. Det er mulig å bruke *wildcards* i denne filen, så om man skriver for eksempel

```
*.out
*.o
*.hex
```

i .gitignore, vil `git` ikke bry seg om filer som slutter på ".out", ".o", eller ".hex". Om det blir nødvendig å gjøre et unntak til denne regelen, er det også mulig. Om det foreksempel blir nødvendig å *tracke* filen "precompiled.hex", kan man gjøre det ved å kalle `git add -f precompiled.hex`, selv om .gitignore-filen ignorerer .hex-filer.

Om dere oppretter denne .gitignore-filen nå, vil `git` allikevel følge "a.out", fordi den ble lagt til før `git` ble fortalt å ignorere kompilerte filer. Dette kan ordnes ved å kjøre `git rm a.out` før neste *commit*.