

GIT 101

Simon Alexander Alsing
201304202
aalsing@gmail.com

9. marts 2016

Resumé

Dette er en lille huskeliste der skal bruges i forbindelse med Projekt opgavens kode, når den skal versionsstyres over git. Ideen er at du kan finde de mest gængse kommandoer. Når jeg støder på nye som kan være relevante vil jeg opdatere guiden. Guiden vil udelukkende beskæftige sig med kommandoer gennem git BASH, men bruger du GIT GUI vil denne guide kunne give dig en ide om hvordan det hele fungerer.

Indhold

1	Terminal	3
2	Begynd et nyt git projekt	4
3	Everyday Commands	4
3.1	status	4
3.2	add	4
3.3	commit	5
3.4	push	5
3.5	diff	6
3.6	pull	6
3.7	log	7
3.8	mv	7
3.9	bisect	7
4	Branches	8
4.1	Skift branch	8
4.2	Opdater listen med branches	8
4.3	Ny branch	8
4.4	Merge	9
5	Git Ignore	9

1 Terminal

Som nævnt i abstract vil jeg kun bruge GIT BASH i dette dokument, og derfor er der brug for at introducere terminalen. Kender du allerede til brug af en terminal kan du snildt springe ned til afsnit 2. Kender du ikke til det, er her nogle hurtige tip.

- Når du bruger terminalen er det vigtigste at forstå, at du navigerer på samme måde som du ellers gør. Når du åbner et dokument du skriver noter i, vil du typisk have dette gemt i en mappe. Når du skal åbne filen navigerer du til denne mappe fra "Dokumenter" til "EnMappe" til "EnAndenMappe", til du sidst kommer ind til mappen som indeholder dine noter. På samme måde gør du i en terminal, men her er det mere explicit. Når du bruger git bash, bruger du "/" til at skelne mellem hver mappe i stedet for "\", så en sti ville være:

```
C:/Dokumenter/EnMappe/EnAndenMappe/
```

Stien er den samme, men du fortæller systemet præcis hvordan den finder frem til det.

- Du skifter sti med kommandoen **cd**, som står for "change directory". Vi kan altså skifte ind i mappen "EnAndenMappe" ved følgende kommando:

```
cd C:/Dokumenter/EnMappe/EnAndenMappe/
```

Men husk nu på at stierne er Case sensitive, så skriver du med småt, når mappen er med stort, så kommer du ikke det rigtige sted hen.

- Nu ved du hvordan man skifter sti, men når du er i gang med at skifte sti, som altså er case-sensitive, så er det dejligt hvis nu man kunne få nogle shortcuts til at skrive de lange stier. Her kan du bruge Tap, som altså hjælper med autocomplete når du har skrevet et par af bogstaverne i hvert mappenavn - meget brugbart hvis man skal skrive stien selv.
- Er du bange for at skrive stien, kan du åbne mappen du skal finde frem til som du normalt ville. Herfra højreklikker du og vælger "Git BASH Here" fra menuen.

Mangler der ting til listen så skriv til mig, den vil løbende blive opdateret.

2 Begynd et nyt git projekt

Når et projekt er oprettet på github, kommer der en URL specifikt for dette projekt. Denne URL skal du og alle andre i projektet bruge for at samarbejde omkring projektet. Når du har URL'en for projektet, åbnes git bash i en mappe du ønsker at gemme projektet i. Husk på at git laver en mappe med navnet for projektet, du behøver derfor ikke at oprette en ny mappe til projektet. Du skifter sti med kommandoen `cd`, se afsnit 1 for mere. Herefter skal du klonе projektet via kommandoen:

```
git clone URL
```

Hvor URL erstattes med url/adressen til det pågældende repository.

3 Everyday Commands

3.1 status

Du bruger `status` kommandoen til at se hvilke ting du har ændret siden sidste `commit`. Kommandoen skrives:

```
git status
```

Du vil opleve op mod fem forskellige beskeder fra `git status`, `added`, `removed`, `moved`, `tracked` og `untracked`. De tre første giver sig selv, mens `tracked` betyder at git følger ændringerne i filen og opfatter den som en del af projektet, mens `untracked` er filer som ikke er tilføjet til projektet, men ligger i projekt-mappen lokalt på din computer. `Tracked` og `untracked` beskeder kan man komme af med ved hjælp af kommandoen `add`, se sektion 3.2

Se <https://git-scm.com/docs/git-status> for mere information.

3.2 add

Når du har oprettet en ny fil eller ændret en eksisterende fil, skal denne tilføjes til næste push ved hjælp af `add` kommandoen. Lad os sige du har lagt et nyt billede ind i projektet og derved skal tilføje filen `"octocat.png"`, så er kommandoen:

```
git add octocat.png
```

Har du flere filer som skal addes, kan disse skrives efter hinanden, fx skulle vi også addes `hej.txt` er kommandoen:

```
git add octocat.png hej.txt
```

Herved har vi added begge filer til projektet. Husk på at når du i terminalen skal henvise til en fil, så er det den fulde sti fra hvor du er, du skal skrive. Har du fx en fil `"hej2.txt"` til at ligge i mappen `"test"` er kommandoen:

```
git add octocat.png hej.txt test/hej2.txt
```

Herved har du tilføjet tre filer, hvor `"octocat.png"` og `"hej.txt"` er i rod-mappen, mens `"hej2.txt"` er i mappen `"test"`. Har du mange filer som har samme filendelse, kan du bruge stjerne:

```
git add *.png
```

Hermed får du alle de elementer som har filendelsen .png - men det er altså ligegyldigt om de hedder octocat eller carsten.
Se <https://git-scm.com/docs/git-add> for mere information.

3.3 commit

`commit` laver et snapshot af hvilke ændringer du har tilføjet tidligere via `add` kommandoen (se sektion 3.2). `commit` samler dine ændringer og med parameteren `-m` gives en streng med som beskriver ændringerne. Det kan ikke understreges nok, hvor vigtigt det er at lave en god commit besked **HVER GANG**. Dette skyldes at skal du finde tilbage til en tidligere version hvor en given ting fungerede, kan en dårlig commit besked være til hinder for det. Samtidig giver `commit` et godt overblik over projektforsløbet og det ødelægger du ved dårlige beskeder. Man kan få en oversigt over alle commits i hele projektet via loggen, se sektion 3.7. En god commit besked vil være en som beskriver hvad der er ændret. Har vi eksempelvis tilføjet billeder af en katten Egon på stranden, vil beskeden for eksempel være:

"tilføjet billeder af katten Egon på stranden"

Beskeden kommer efter parameteren `-m` i gåseøjnene. En commit besked kan i vores billedeksempel være:

```
git commit -m "tilføjet billeder af katten Egon på stranden"
```

husk `-m` parameteren, som fortæller at nu kommer en streng som er `commit` beskeden. Glemmer du `-m` parameteren kommer du ind i en vim editor og dermed kan man kun bruge vim commands (For at komme ud herfra, skriv `:wq` plus enter. Det vil sige shift + punktum, w, q. Dette burde annullere dit commit, da du prøvede at lave et commit med en tom besked).

Se <https://git-scm.com/docs/git-commit> for mere information.

3.4 push

Et `push` er her hvor du gemmer alle dine ændringer oppe på GitHub eller hvor dit repository nu ligger. Det er her hvor de andre i projektet modtager dine ændringer. Kommandoen er:

```
git push origin [branch]
```

Læg mærke til buzzwordet `branch` - det er en placeholder for den `branch` som du ønsker at pushe dine ændringer til. Dette vil typisk være den `branch` du allerede er på. For mere (se sektion 4)

Du har mulighed for at smide parameteren `-u` med, som "husker" hvilken branch du pusher til. Derved skal du første gang skrive:

```
git push -u origin [branch]
```

For at pushe til den givne branch. Næste gang du skal lave et `push` skrives kun:

```
git push
```

Hvis du får `mergeconflict` ved et `push` kan du med fordel se sektion 3.5
Se <https://git-scm.com/docs/git-push> for mere information.

3.5 diff

Har du ændret i nogle linjer i en fil, som lige er blevet redigeret af en anden i projektet, vil dette give en mergeconflict. Fejlen vil være:

```
$ git pull
Auto-merging [FILE]
CONFLICT (content): Merge conflict in [FILE]
Automatic merge failed; fix conflicts and then commit the result.
```

Her vil det være dejligt at se hvilke ændringer der laver konflikt med den eksisterende fil. Git diff giver dig et overblik over hvilke linjer du har tilføjet til filen og hvilke der konflikter. Kommandoen er:

```
git diff [FILE]
```

Hvor [FILE] erstattes med filens navn. Herefter vil du i terminalen se forskellene fra din ændring og den eksisterende fil i projektets REPO. Har vi eksempelvis filen test.txt som oprindeligt har indeholdt Goddag2 på første linje og får merge konflikt med linjen Goddag3, som vi laver en diff på, vil output være:

```
diff --cc test.txt
index fb3dfd9,9590b33..0000000
--- a/test.txt
+++ b/test.txt
@@@ -1,1 -1,1 +1,5 @@@
-Goddag2
++<<<<<<< HEAD
+Goddag3
+=====
++Goddag2
++>>>>>>> d7c0c25d6375a74def1f1bb906a0e5ef888103e6
```

Herfra kan du se at Goddag2 er den gamle (se efter minustegn), mens Goddag3 er den nye. Du kan nu gå ind i test.txt og vælge hvilken ændring der skal gemmes. Her skal du fjerne

```
<<<<<<< HEAD
=====
>>>>>>> d7c0c25d6375a74def1f1bb906a0e5ef888103e6
```

Som er hvad git har indsat for at vise dig hvor konflikten er. Dette kan forekomme flere steder i filen. Du skal også forholde dig til hvilken ændring der skal blive, evt beholde dem begge!

3.6 pull

pull henter ændringer ned fra en branch. pull sørger for at din lokale kopi er up to date med serveren. Kommandoen er:

```
git pull origin [branch]
```

hvor branch er den branch du vil pull ændringer til - det er altså ikke alle branches som bliver opdateret.

Se <https://git-scm.com/docs/git-pull> for mere information.

3.7 log

Git loggen er et godt værktøj til at give overblik over hvilke ændringer der er sket i projektets levetid. Hver gang et medlem af projektet har lavet et commit eller et merge af to branches, vil dette kunne ses på loggen med navn og tidspunkt. Den giver dig en log for den branch du er på. Kommandoen er:

```
git log
```

Hvis der er flere beskeder i loggen end der kan være i terminalvinduet, kan du scrolle op og ned med piletasterne samt page up og page down. For at komme ud af loggen kan du taste q.

Se <https://git-scm.com/docs/git-log> for mere information.

3.8 mv

For at flytte en fil bruges kommandoen **mv**. Hvis man flytter en fil i filsystemet uden at fortælle det til git, vil git tro at filen er slettet og at der er oprettet en ny fil, der hvor den er flyttet hen. Ved at bruge **mv** bevarer man således historik for filen, så man kan se hvilke ændringer der er sket hvornår. **mv** bruges også når man vil omdøbe en fil (dvs. filen bliver flyttet fra det gamle navn til det nye navn). Kommandoen for at flytte "hej.txt" ind i en undermappe der hedder "ny-mappe":

```
git mv hej.txt ny-mappe/hej.txt
```

Kommandoen for at omdøbe "hej.txt" til "hejsa.txt" er:

```
git mv hej.txt hejsa.txt
```

Se <https://git-scm.com/docs/git-mv> for mere information.

3.9 bisect

bisect er et godt værktøj til at finde ud af hvornår en fejl er introduceret, og hvad fejlen er. Hvis du står med et projekt der tidligere har virket, men ikke virker i det seneste commit og ikke umiddelbart kan finde fejlen, er **bisect** nyttig. **bisect** laver en binær søgning af alle commits, som hjælper til med hurtigt at finde hvor fejlen er introduceret. Det er vigtigt at forstå, at **bisect** er en serie af kommandoer, som giver dig svar på de spørgsmål du måtte have. Til at starte **bisect** bruges:

```
git bisect start
```

Herefter fortæller man git hvilket commit der ikke virker. Som oftest vil det være det seneste commit, dvs. det commit man er på nu. Kommandoen for dette er:

```
git bisect bad
```

Til sidst skal vi fortælle git hvad den seneste gode version var. Dette gøres med

```
git bisect good [good-commit]
```

Her er **good-commit** hashen eller tagget for det seneste gode commit (ses i loggen, se sektion 3.7). **bisect** checker nu et commit ud for dig. Git vælger

nu et commit mellem det som virker og det som ikke virker. Hvis fejlen du troubleshooter ikke er i den aktuelle version(en version mellem good og bad), taster du

```
git bisect good
```

ellers, hvis fejlen stadig eksisterer i den aktuelle version taster du

```
git bisect bad
```

Dette bliver ved indtil der ikke er flere muligheder, så punktet hvor fejlen blev introduceret er fundet. For at komme tilbage til det nyeste commit og ud af **bisect** tastes

```
git bisect reset
```

Se <https://git-scm.com/docs/git-bisect> for mere information.

4 Branches

Med branching skal du tænke på et træ, stammen i et gitprojekt er Master branchen - alt brancher ud fra denne(i hvert fald i vores projekt). Når du er på master branchen kan du kun se hvad den indeholder - og altså ikke noget fra andre branches. Når du laver en ny branch, laves en kopi fra den branch du er på. Du får altså indholdet fra den branch du var på, men herefter får du ikke de ændringer der sker på den originale branch og den får ikke de ændringer du laver på den nye. Disse ændringer kan man selvfølgelig overføre fra en branch til en anden (se sektion 4.4).

4.1 Skift branch

Du skifter branch ved kommandoen:

```
git checkout [branch]
```

hvor **branch** erstattes med hvad branchen hedder, hvis du ikke er sikker på hvilke du har adgang til se ??

4.2 Opdater listen med branches

Hvis ikke alle branches kommer frem er det fordi du ikke har opdateret din oversigt over branches, dette gøres ved:

```
git remote update
```

Herefter burde alle virke.

4.3 Ny branch

Du laver en ny branch ved kommandoen:

```
git checkout -b [branch]
```

så ligesom at skifte branch, bare med parametren **-b** som indikerer at du laver en ny. **branch** argumentet skiftes selvfølgelig til det som man vil kalde branchen.

4.4 Merge

Merge er en funktion som kan bruge på flere måder, jeg vil kun gennemgå at kopiere en branch over i en anden. Den fungerer ved at du skal være på den branch som du ønsker at der skal merges til. Herefter skrives kommandoen:

```
git merge [branch]
```

hvor **branch** erstattes med den branch som der skal merges fra.

5 Git Ignore

En gitignore fil kan være smart hvis man arbejder i projekter som består af mapper som ikke skal medtages. Dette vil typisk være debug mapper. Der findes mange online gitignore filer som man kan hente, der er tilpasset til forskellige projekter. Selvom de ofte vil få jobbet gjort kan det være nødvendigt at skrive dele selv.



Når der laves en gitignore ville denne typisk laves i starten af projektet. Laves projektet på GitHub har du muligheden for at vælge en tilpasset gitignore fil, som passer til sproget du skriver projektet i - for os typisk C++. På figur 1 ser du vinduet i GitHub som laver et nyt projekt eller repository som det kaldes. Læg mærke til den nederste del, der er valgt en gitignore til C++ og der laves en readme. Readme er en beskrivelse af projektet - bruges typisk i opensource projekter til at give en hurtig beskrivelse af hvad projektet er og hvordan man evt. bruger det / installerer det. .gitignore og README er noget man typisk vil inkludere i alle projekter.

Helt generelt er en git ignore fil, en fil uden navn med filendelsen '.gitignore'. Filen laves ved at åbne git bash og navigere til mappen med projektet i (se afsnit 1). Herefter bruger vi kommandoen touch til at lave filen:

```
touch .gitignore
```

Create a new repository


A repository contains all the files for your project, including the revision history.


Owner	Repository name
 Quanalogy ▾	/ Semesterprojekt2 

Great repository names are short and memorable. Need inspiration? How about **supreme-broccoli**.

Description (optional)


Andet semesterprojekt

☒  **Public**
Anyone can see this repository. You choose who can commit.

☐  **Private**
You choose who can see and commit to this repository.

☒ **Initialize this repository with a README**
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **C++** ▾

Add a license: **None** ▾ 

Create repository

Figur 1: Et nyt repository på GitHub.

Filen åbnes en en text editor som **IKKE** bruger encoding (dvs. ikke Word), det åbnes derfor i **notepad**, **notepad++** eller lignende. I filen skrives stien til en fil, mappe, eller hele filendelser/navne. Her er et eksempel fra den fil som GitHub laver til dig for et C++ projekt:

```
\verb!###! Compiled Object files\\
*.slo\\
*.lo\\
*.o\\
*.obj\\

\verb!#! Precompiled Headers\\
*.gch\\
*.pch\\

\verb!#! Compiled Dynamic libraries\\
*.so\\
*.dylib\\
*.dll\\

\verb!#! Fortran module files\\
*.mod\\

\verb!#! Compiled Static libraries\\
*.lai\\
*.la\\
*.a\\
*.lib\\

\verb!#! Executables\\
*.exe\\
*.out\\
*.app!\\
```