ERHVERVSAKADEMI AARHUS

Git teori

Indholdsfortegnelse

1. Indledning

Git er et distribueret versionsstyringssystem. Det anvendes til at lave backup af programkode samt til at dele programkoden blandt udviklere, der arbejder på et fælles projekt. I denne note vises, hvordan man kan anvende de mest elementære funktioner i Git.

Git giver udvikleren (dig!) følgende fordele.

- Du har projektets historie. Du kan gå tilbage til et tidligere trin af dit projekt.
- Du og andre udviklere kan arbejde på det samme projekt samtidigt.
- Du kan eksperimentere med ny kode i dit projekt. På et senere tidspunkt kan du enten flette den nye kode med din eksisterende kode, eller du kan smide din nye kode væk.
- Der er en backup af projektet hos alle udviklere, der arbejder med projektet.

I det følgende afsnit vil de grundlæggende begreber, der anvendes i Git, blive forklaret.

2. Begreber i Git

1.2 Commit

Et commit er et øjebliksbillede af al din kode. Når der laves commit på forskellige tidspunkter i udviklingen, bliver udviklingshistorien for din kode gemt i form af en række commits. Et nyt commit peger tilbage på det foregående commit.

Det giver dig mulighed for at skifte mellem forskellige commits, dvs. at hoppe imellem forskellige tidspunkter i udviklingen af din kode. Når du skifter til et andet commit, bliver dit working directory (se nedenfor) opdateret, så det svarer til koden i det pågældende commit. Dette betyder, at du kan hoppe tilbage i tid og se, hvordan din kode så ud på et tidligere tidspunkt i udviklingsforløbet.

1.3 Repository, Staging area og Working directory

Begrebsmæssigt består Git af tre dele: *working directory*, *staging area* og *repository*.

Working directory indeholder den aktuelle tilstand af din kode, som du ser den i IntelliJ.

Staging area indeholder en liste med de filer, der skal inkluderes i den næste commit.

Repository indeholder alle de commits, der er lavet indtil nu.

Når du udvikler i IntelliJ, arbejder du kun med filer i working directory. Hver gang du gemmer din kode i IntelliJ, er det kun working directory, der opdateres. Når du vil have ændringerne i koden gemt i Git, altså når du vil lave et øjebliksbillede af din kode og gemme det i dit repository, skal der laves et commit. Et commit ændrer på dit repository og tømmer listen med filer i staging area, mens working directory er det samme før og efter et commit.

1.4 Working directory

Working directory er den mappe på din maskine, hvor din kode er placeret.

1.5 Staging area

Staging area (også kaldet for *index*) indeholder de filer, der skal med i det næste commit. Ved at vælge hvilke filer der er i staging area, kan du styre, om alle ændrede filer, eller kun nogle af de ændrede filer, skal med i dit næste commit.

Staging area kan vi (næsten) glemme, da vi (næsten) altid vælger at inkludere alle ændrede filer i næste commit.

Bemærk, at IntelliJ som standard ikke bruger staging area men en såkaldt changelist (kaldet Changes) til samme formål: At kunne forhindre nogle filer i at komme med i næste commit.

1.6 Git repository

Et repository er en (skjult) mappe på din maskine. Repositoriet indeholder alle de filer, der er nødvendige for at Git kan fungere (mappen hedder .git). Repositoriet en en undermappe i dit working directory.

Din kode er placeret to steder. Som læsbare tekstfiler i dit git working directory og som komprimerede filer i repositoriet (dvs. i .git-mappen).

1.7 Lokal og remote repository

Når flere udviklere arbejder på samme projekt, har de hver lokalt på deres egen maskine et git repository med alle commits, der er blevet tilføjet igennem projektets levetid. For at kunne dele kode med andre, laves ofte et fælles git repository på en server.

En udvikler kan lave *push* af commits fra sit lokale repository til det fælles repository på serveren, og andre udviklere kan herefter lave *pull* af commits fra serveren til deres lokale repositories. På den måde får alle repositories en kopi af alle commits udført af alle udviklere i projektet.

På din lokale maskine kan du arbejde med flere branches (branch: se næste afsnit), men *main branch* er typisk den branch du *pusher* til den fælles server, og den branch du *puller* tilbage til, når du vil have de andres ændringer fra serveren ned i dit eget repository (main branchen hed før i tiden main branchen). Andre branches på din lokale maskine vil du normalt ikke pushe til det fælles repository på serveren (du vil i stedet flette dem ind i *main* på din lokale maskine).

1.8 Branch

En branch er en navngiven række af commits. I praksis består en branch af en label, der peger på et bestemt commit. Alle de commits, der kommer *før* det commit, som labelen peger på, er med i den række af commits, som udgør branchen.

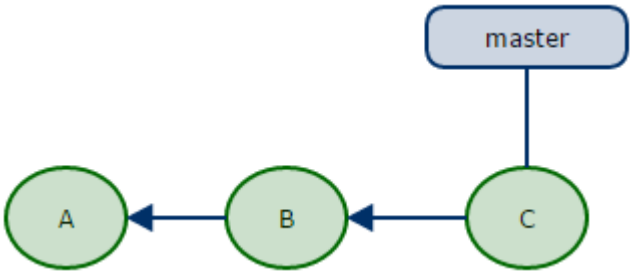
Du kan have mange branches. Branches gør det let at skifte mellem forskellige forløb. Fx kan man have en branch til test og en anden til udvikling. Man kan også have branches til forskellige version er af sit projekt. Det er muligt at oprette en ny branch ud fra en eksisterende branch og flette den nye branch tilbage i den eksisterende branch på et senere tidspunkt.

1.9 Skift til en anden branch

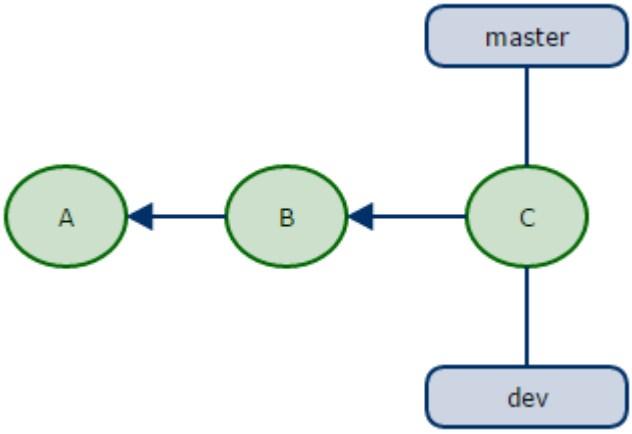
Når du skifter til et anden branch, vil working directory blive opdateret, så det indeholder koden hørende til den branch, der er skiftet til. Dette kan give problemer, hvis du i den aktuelle branch har nogle ændringer, der endnu ikke er committet, og som vil blive overskrevet, hvis du skifter til en anden branch. Her vil Git advare dig, så du har mulighed for at comitte dine ændringer i den aktuelle branch, inden du skifter branch.
(Der er en anden mulighed: Man kan gemme sine ændringer i den aktuelle branch midlertidigt i noget der hedder en *git stash*, så ændringerne ikke går tabt, når man skifter til en anden branch.)

2 Branches i Git

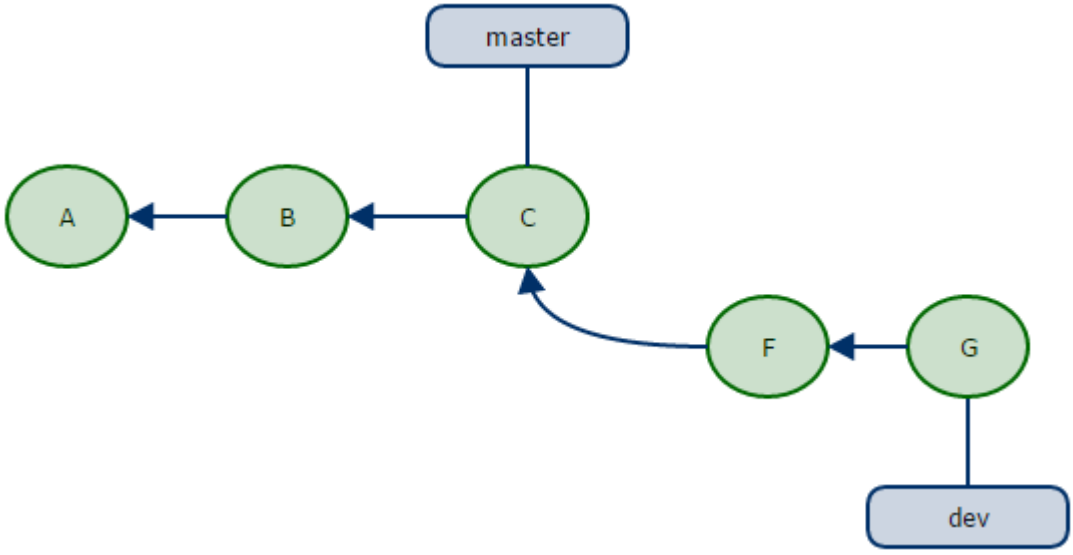
Vi starter i nedenstående situation, hvor vi har en *master branch*, på din lokale maskine, hvorpå der er udført 3 *commits* (A, B og C). Branchen har en label *master* der peger på den sidst skabte *commit*.



Derefter opretter vi en ny branch *dev*. Der oprettes blot en ny label der peger på samme commit, som master branchen peger på. De to branches, dev og master, indeholder de samme commits til at starte med.

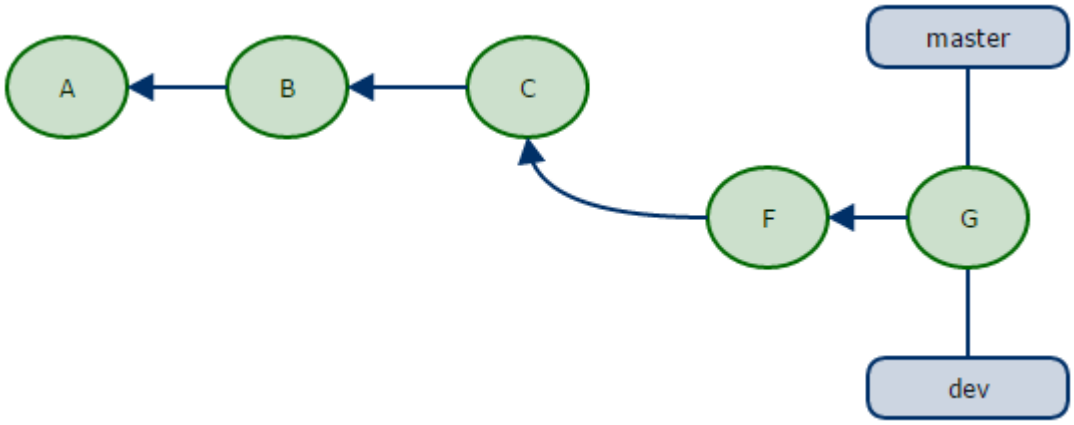


Der arbejdes videre på branchen *dev* og der laves to commits (F og G) på denne. Det betyder at de to branches nu ikke længere indeholder de samme commits.

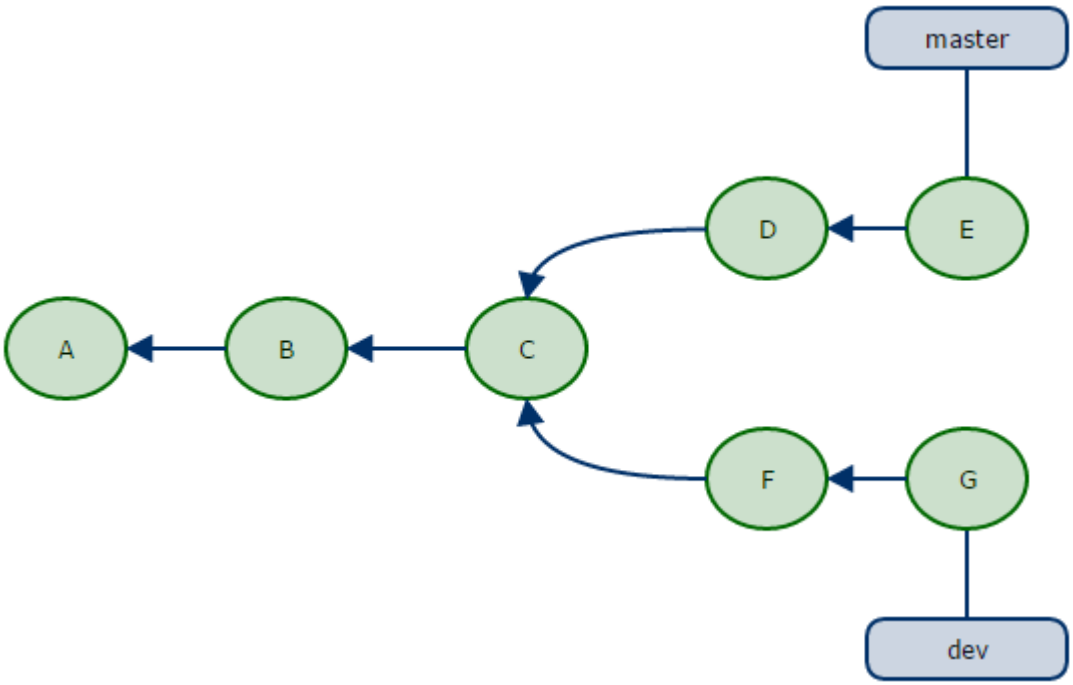


Hvis vi ikke har brug for at anvende den kode vi har fået lavet, kan vi skifte tilbage til *master* branch og glemme alt om *dev*. Vi kan endda slette *dev*, hvis vi ønsker det (det sletter kun lablen *dev*, men vi kan ikke senere (let) få fat i F og G).

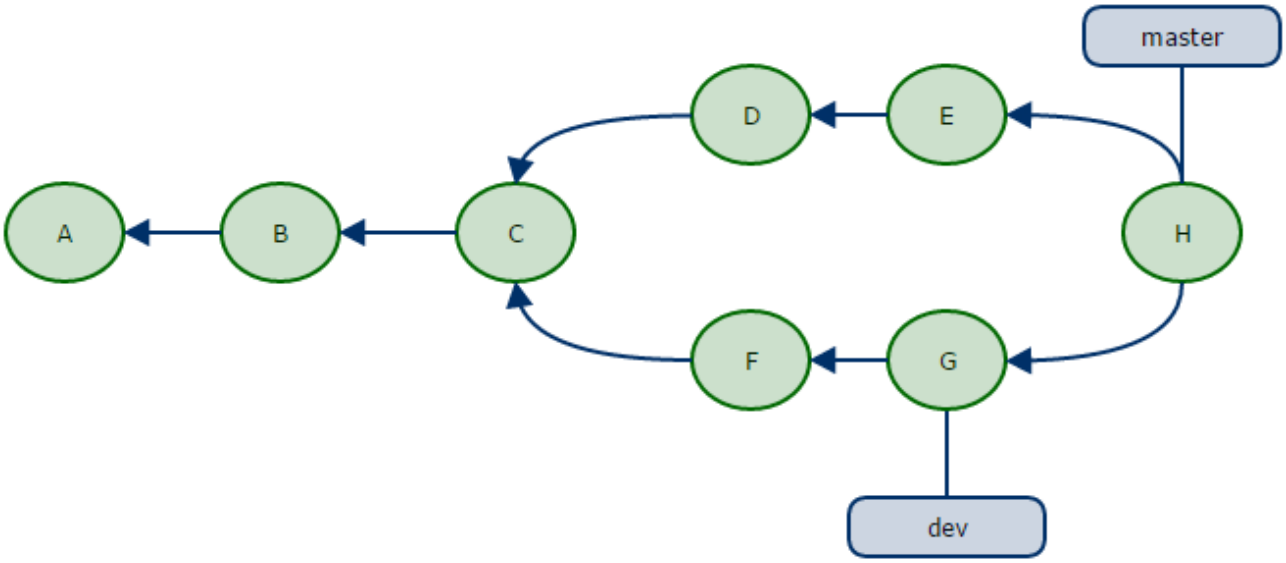
Hvis vi er tilfredse med koden i branchen *dev*, kan vi flette koden ind i *master* branch. Git vil da blot flytte master branch label til at være det samme som *dev* branch. Det kaldes at lave en *fast-forward merge* – se illustrationen herunder.



Lad os se på den situation, hvor der både er lavet commits på *master* og *dev* branchen. Det kunne fx være, at de nye commits D og E i master-branchen kommer fra en *pull* på et remote repository, og F og G er lavet lokalt af udvikleren selv.



I dette tilfælde kan Git ikke lave *fast-forward merge*, når vi fletter *dev* branchen ind i *master*. Git laver i stedet en *merge commit* (kaldet H i figuren nedenfor). Efter denne merge indeholder *master branch* alle commits, da master peger på H, der igen peger på alle tidligere commits.



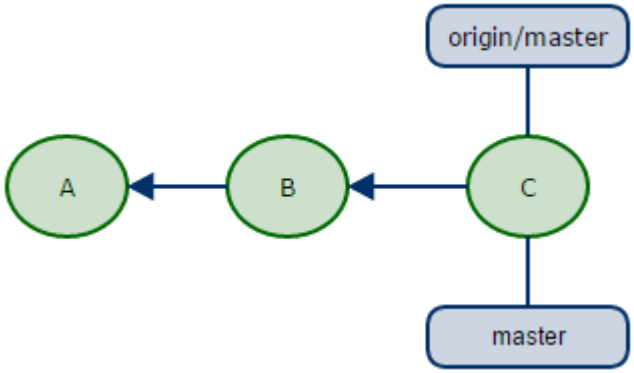
Hvis der ikke skal arbejdes videre på branchen *dev*, kan denne slettes. Det er blot lablen *dev*, der slettes, da de to commits F og G stadig eksisterer i master-branchen i rækken af commits op til commit H.

I de fleste tilfælde kan Git automatisk generere ovenstående *merge commit*. Hvis commit E og commit G indeholder ændringer til de *samme* linjer i den *samme* fil, kan Git ikke afgøre, hvordan de ændrede linjer skal se ud, og der opstår en *merge conflict*. Herefter er det op til udvikleren at gå ind i IntelliJ og vælge hvordan filerne skal se ud, før merge gøres færdigt.

3 Lokalt repository klonet fra remote repository

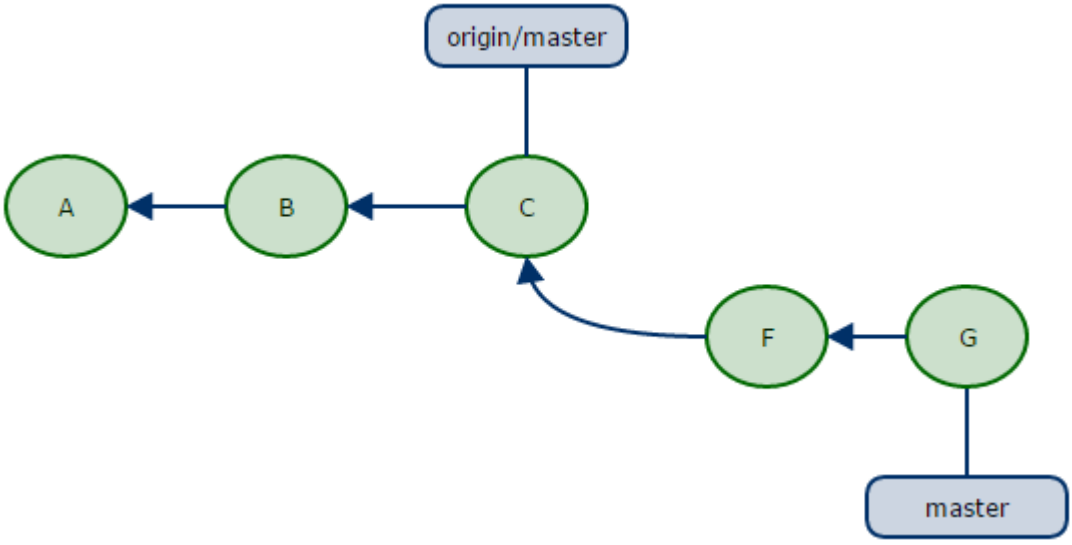
Når du *cloner* et *remote repository* fra den fælles server til din egen maskine, får du et lokalt repository, der indeholder en præcis kopi af indholdet fra remote repository. Dit lokale repository indeholder som minimum to branches: *master* branch og en branch kaldet *origin/master*.

Origin/master branchen vil følge (eng: tracke) *master* branchen på remote repository. Dit lokale repository vil derfor se således ud efter en cloning af et remote repository med 3 commits:

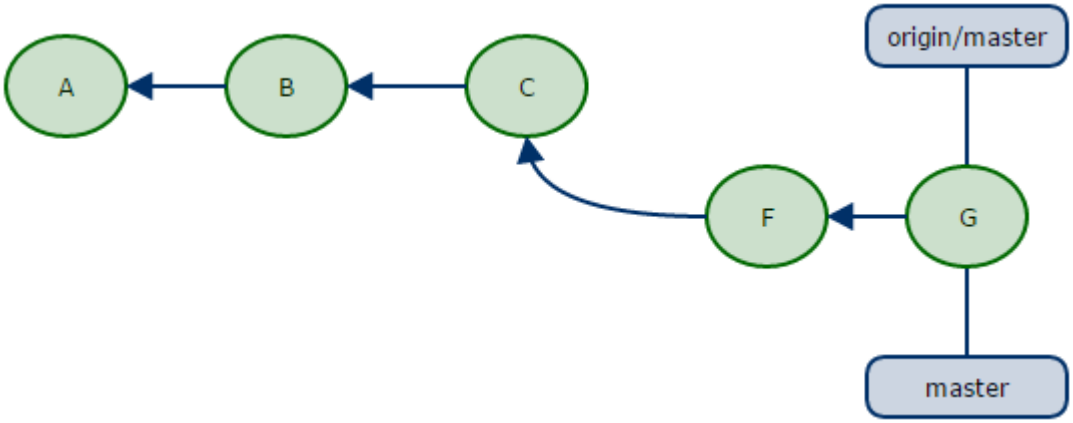


Det er ikke tilladt at lave commits til *origin/master* branch, da denne branch skal svare til commits på det *remote repository* du har klonet fra.

Lad os antage, at du har lavet to nye commits på *master* branch i dit lokale repository:



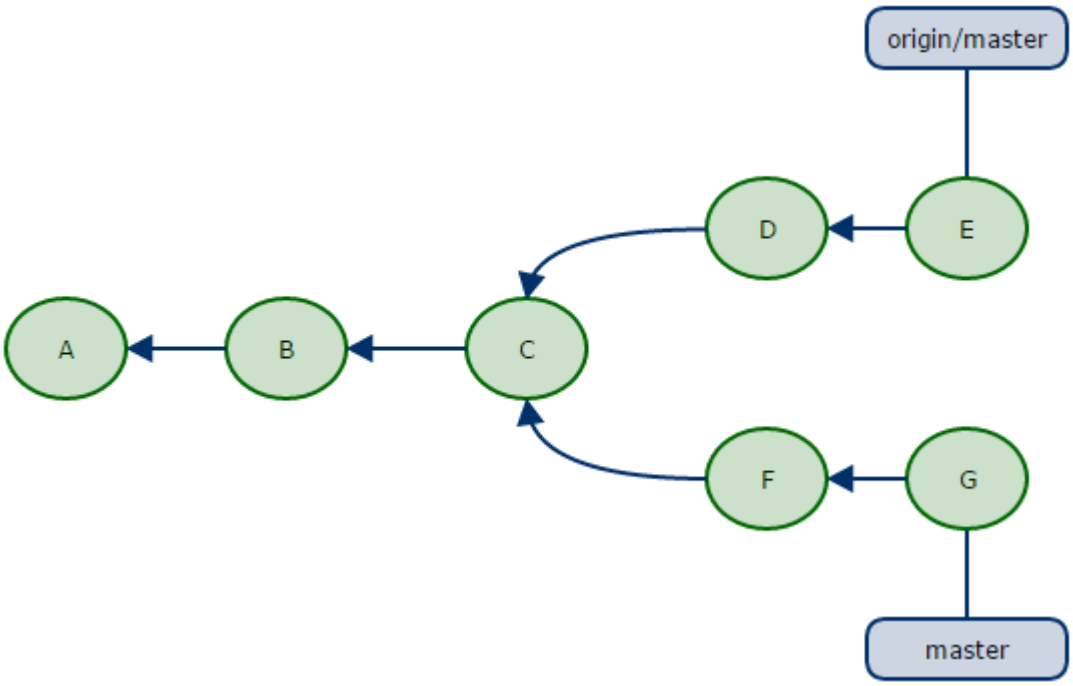
Hvis du laver et *push* af din *master* branch til remote repository, og remote repository ikke har fået nye commits siden commit C, vil Git opdatere remote repository med dine nye commits og opdatere dit lokale repository til følgende:



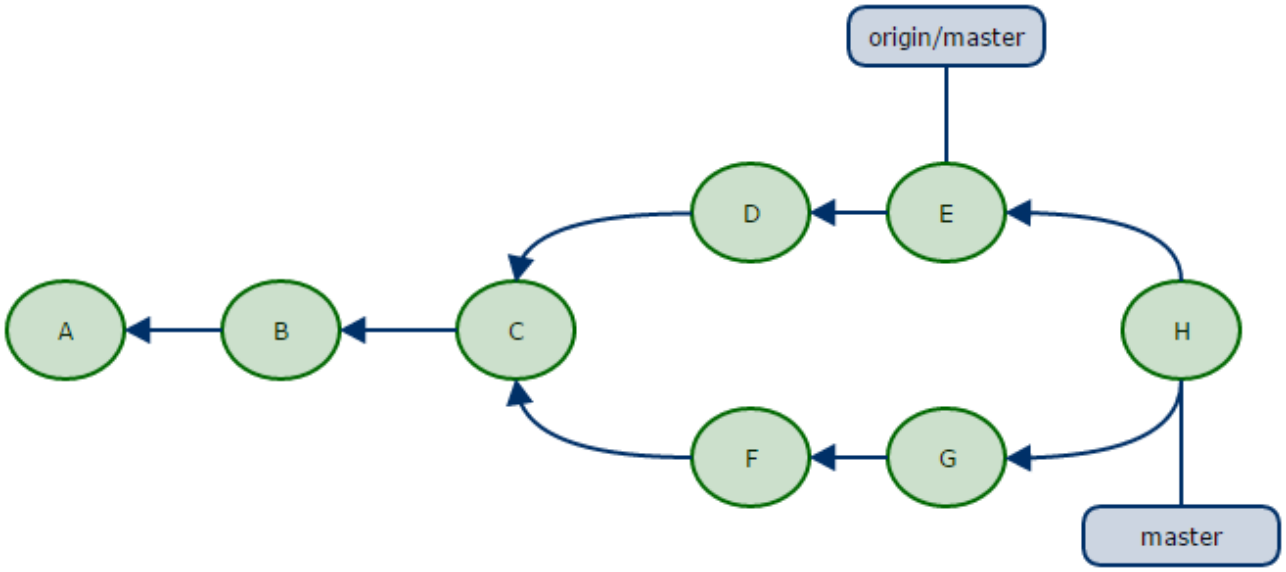
Dit lokale repository og remote repository er nu synkroniserede.

Hvis du forsøger et *push* til remote repository, når remote repository indeholder nye commits, der ikke er på din lokale *origin/master* branch, vil Git ikke gennemføre *push*, men i stedet informerer dig om, at din lokale *origin/master* er ude af sync.

Du bliver derfor nødt til at lave et *fetch* fra remote repository til din lokale maskine for at opdatere dit lokale repository. Efter et sådan *fetch* vil dit lokale repository se ud som følger (D og E kommer fra remote repository):

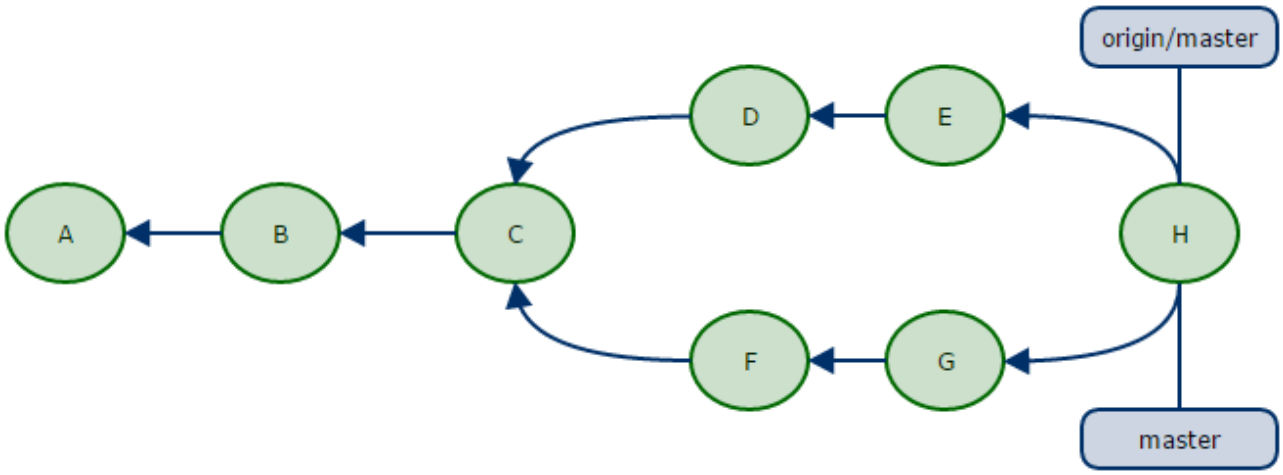


Nu skal du lave en merge af *origin/master* til din lokale *master* branch (og løse eventuelle merge konflikter der måtte opstå).



Fetch efterfulgt af merge kan udføres af én kommando, som hedder *pull*.

Herefter kan du lave et *push* til remote repository igen. Git vil opdatere remote repository og opdatere origin/master branchen i dit lokale repository som følger:



Det lokale repository og remote repository er igen synkroniserede.

Såfremt du er så uheldig at remote repository har fået et nyt commit, imens du var i gang med at merge, da må du lave yderligere et *pull*, inden du får lov til at afslutte med et push.

4 Git Workflow: Hvordan man arbejder med git i et team

Der er mange bud, på hvordan man bedst benytter git på et udviklingsprojekt. I det følgende præsenteres to måder man kan gøre tingene på. For udviklere, der ikke tidligere har arbejdet med versionsstyringsværktøjer, anbefales i førstes omgang, at man følger det simple workflow.

4.1 Det simple workflow

I det simple workflow benyttes der kun én branch, og det er main-branchen. Der comittes til main-branchen hver gang man har lavet ændringer, man vil gemme. Når man har lavet noget færdigt, man gerne vil dele med sit team, pusher man til remote repository, og beder de andre medlemmer om at lave et pull for at få hentet de nye ændringer. Det kan være en fordel at bede de andre medlemmer om at holde nallerne væk fra remote repository, imens man pusher, så der ikke opstår problemer med push undervejs.

I det simple workflow er det også nemt at få ændringer ind fra de andre medlemmer. Man laver simpelthen en pull og ser, om man får noget nyt kode hentet ned. Der kan opstå konflikter, men ofte kan git selv finde ud af at merge nye ændringer ind i ens lokale master-branch uden problemer.

Fordelen ved det simple workflow er, at det er simpelt – ingen branching!

Ulempen er, at al udvikling sker på den samme branch, nemlig main branch. Samtidig er det den branch, der indeholder den "officielle" kode, der er delt med de andre udviklere. Det betyder, at hvis man har flere medlemmer, der udvikler på forskellige features, kan det være svært at styre. Derfor anbefales et smartere workflow til længerevarende og større projekter.

4.2 Workflow med flere branches

I workflow med flere branches udvikler man i en anden branch end master-branchen. Den anden branch vil typisk hedde *develop* (eller lignende). Når man har udviklet et eller andet på *develop*, man gerne vil publicere, sker det som følger:

1. Man skifter først fra *develop* branchen til *main* branchen. Husk at comitte på develop, *inden* du skifter til main, så dine ændringer på *develop* er gemt og klar til at blive merget.
   Du skifter branch ved: Git -> Branches -> main -> CheckOut.
2. Herefter laver man et pull på *main* for at sikre sig, at man har de nyeste commits fra origin/main (fra remote repository).
   Du puller med: Git -> Pull
3. Hvis man sidder og arbejder med sit team, kan man her passende gøre opmærksom på, at man lige om lidt laver et push til *main* branchen på remote, og at de andre helst ikke skal røre ved remote, før man er færdig.
4. Når man har sikret sig, at ens lokale *main* branch er opdateret med det nyeste (det er den, hvis pull går godt), laver man en merge af den lokale *develop* branch ind i den lokale *main* branch.
   Du merger med: Git → Merge, og i vinduet "Merge into main" vælge *develop*.
5. Herefter laver man en push af den lokale *main* branch til remote repository. Går det godt, har man nu fået publiceret de nyeste ændringer til remote main.
6. Gør herefter de andre i teamet opmærksom på, at man er færdig med at pushe, så de kan få de nyeste ændringer hentet ind i deres lokale *main* branch med pull.
7. For at udvikle ny kode, skiftes til *develop* branchen, så man kan udvikle videre på den. Start med at lave en merge af *main* ind i develop, så main og develop er på samme commit (nødvendigt, hvis nye commits er skabt under punkt 2).

At pushe fra en bestemt branch gøres ved at skifte til den pågældende branch, og herefter udføre en push. Det er normalt kun den branch, man står på, der bliver pushet. Der er en **god idé kun at pushe main til remote**.

Fordelen ved dette workflow med flere branches er, at man holder sin lokale udvikling på *develop* adskilt fra den officielle kode på *main*, der er delt med de andre udviklere via remote repository. Det betyder, at man altid kan hente og se de andres commits til den fælles kode ved at pulle til sin egen main branch fra remote main, uden at det forstyrrer ens egen udvikling på develop.

4.3 Merge konflikter og hvordan de løses

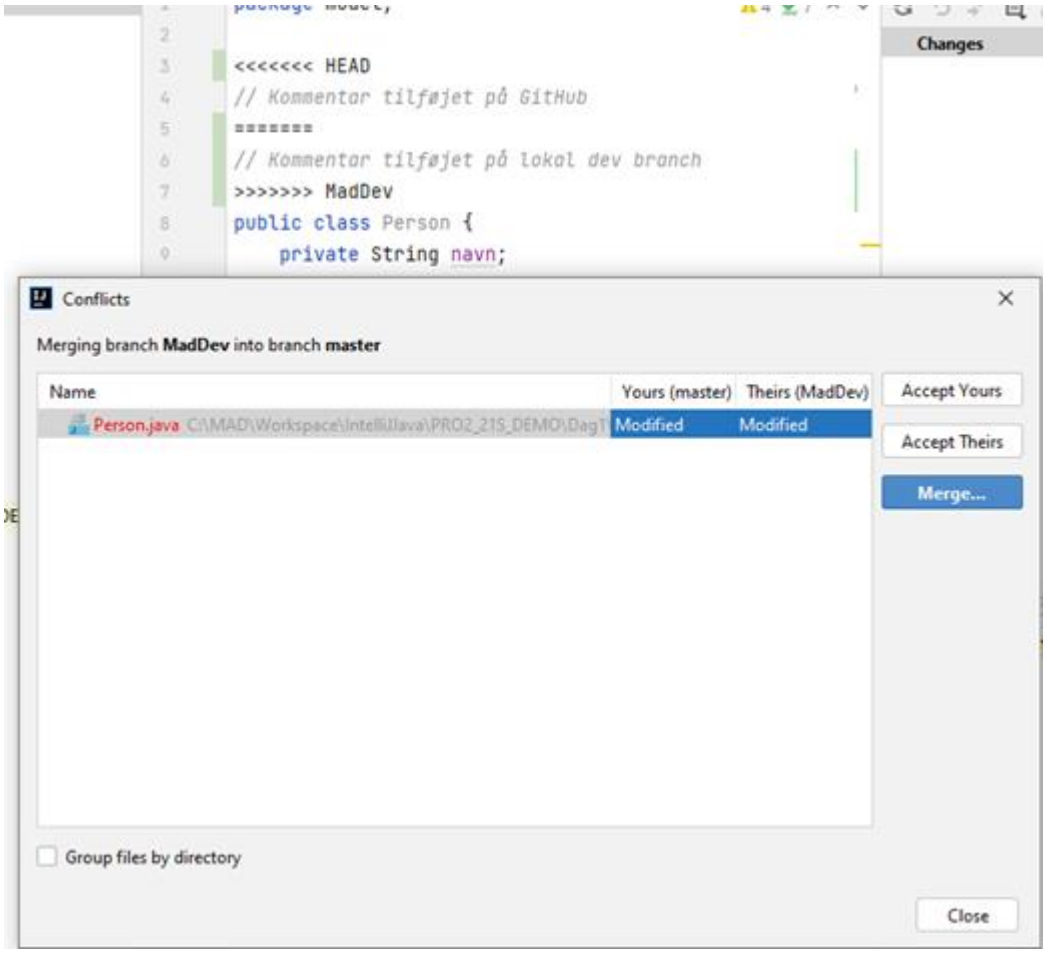
I praksis kan man (nemt) komme ud for, at man har ændret noget på en lokal branch, og der i mellemtiden er andre der har pushet ændringer op på den samme branch på GitHub.

Ofte sker der ikke noget ved det, da git er smart nok til at kunne flette ændringer sammen, når man laver et pull. Men hvis f.eks. to personer har ændret i den samme linje i den samme fil, skaber det en konflikt ved pull.

I det følgende vil vi fremprovokere sådan en konflikt for at illustrerer, hvordan det kan ske, og for at vise, hvordan det kan løses. Bemærk, at der arbejdes med en branch kaldet MadDev.

Lad os lave et lokalt commit. Find et passende IntelliJ projekt som du har liggende i et git repository.

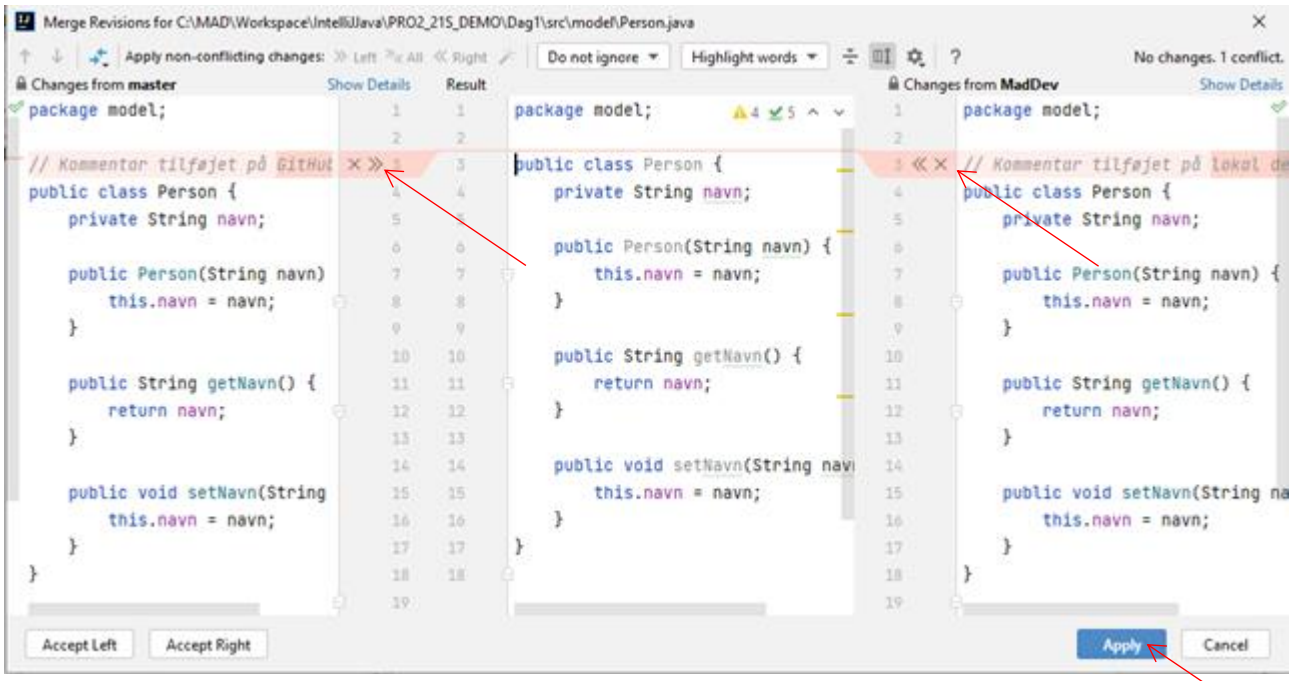
1. I en lokal udviklingsbranch (figuren herunder kalder den MadDev) laves en lille triviel ændring i en fil (kunne f.eks. være at ændre i en kode-kommentar i en Java-klasse).
2. Commit ændringen – du skal *ikke* pushe den til GitHub endnu.
3. Hop ind på GitHub og lav en ændring i *den samme linje i den samme fil* på main branchen. Du kan redigere direkte i filer på GitHub ved at åbne dem fra Source-tabben på dit repository.
4. Skift til main branchen lokalt, og pull ændringen fra main på GitHub ned i din lokale main. Det vil gå godt.
5. Lav nu en merge af MadDev ind i main. Dette vil resultere i en konflikt.



Filen med konflikt har nu fået følgende indhold hvor de to modstridende ændringer forekommer:

```
<<<<<<< HEAD
    // Kommentar tilføjet på GitHub
=======
    // Kommentar tilføjet på lokal dev branch
>>>>>>> MadDev
```

Filen i eksemplet her hedder Person.java, og der er tale om en Java kodekommentar, som både er ændret på GitHub og lokalt. De to commits består begge af en ændring til den samme linje i filen. Det er nu op til udvikleren at vælge, hvordan det endelige resultat skal se ud. Klik på knappen Merge… og vælg/fravælg hvilke ændringer du ønsker:



Lad os beholde den linje hvor der står at filen er ændret på GitHub. Vi klikker derfor på >> til venstre og på X til højre hvorefter der klikkes på knappen Apply.

Herefter er der lavet et merge commit, og ændringerne kan pushes til GitHub. Ofte vil du have din dev branch til at følge med main branchen, og der laves derfor et skift til dev branch efterfulgt af en merge fra main, og alt er opdateret.

Kigger man på listen over over commits på GitHub, vil man efter push kunne se, at der er blevet foretaget et merge-commit: Merge branch 'MadDev'