KENNISMAKING MET GIT

Versiebeheer (deel 2)

Versiebeheer geeft inzicht in de ontwikkeling van de inhoud in bestanden en maakt het mogelijk om naar een eerdere versie terug te gaan. Git is een populair derde generatie versiebeheersysteem. In dit artikel maken we kennis met enkele basiscommando's, die je helpen te starten met Git. **Matto Fransen**

Dit is het tweede artikel in deze serie over versiebeheersystemen. In het vorige artikel maakten we kennis met RCS en bespraken we de vier W's van versiebeheer: Wat, Waarom, Wanneer en Wie. Deze vier W's staan centraal in alle versiebeheersystemen. In dit tweede artikel kijken we naar de ontwikkeling van versiebeheersystemen en maken we kennis met Git.

RCS stamt uit 1982 en is een versiebeheersysteem van de eerste generatie. RCS werkt op bestandsniveau en gebruikt een lokale repository. De eerste generatie versiebeheersystemen werken op basis van locks op bestanden, zodat maar een persoon tegelijkertijd aan een bestand kan werken. Voor deze locks wordt gebruik gemaakt van de Unix-bestandpermissies.

TWEEDE GENERATIE

Dick Grune van de Universiteit van Amsterdam ontwikkelde in 1986 CVS (Concurrent Versions System), waarmee het tijdperk van de tweede generatie versiebeheersystemen begon. CVS gebruikt een centrale repository, bijvoorbeeld op een speciaal daarvoor ingerichte server. De gebruikers checken de code uit naar hun lokale systeem en de wijzigingen worden later weer naar de centrale repository ingecheckt. Hierbij kunnen meerdere gebruikers tegelijkertijd aan hetzelfde bestand werken, vandaar het gebruik van "concurrent" in de naam. Bij het inchecken zorgt de intelligentie van CVS dat de verschillende wijzigingen goed verwerkt worden ("ge-merged"), ook wanneer verschillende mensen aan hetzelfde bestand gewerkt hebben. Verder werken de commando's van CVS op de complete directory-boom, inclusief de subdirectories.

De invoering van een centrale repository, de mogelijkheid om gelijktijdig aan bestanden te werken en het werken op de complete directory-boom, heeft een enorme impact en leidt tot een andere manier van denken over versiebeheer. Daarom ziet men de komst van CVS als de start van een nieuwe generatie.

In de jaren '90 groeide CVS uit tot de standaard voor versiebeheer. De meeste open source software werd onder controle van CVS ontwikkeld. Hierbij werd CVS ook vaak gebruikt voor het verspreiden van de broncode, bijvoorbeeld van de bekende BSD-systemen en hun port-trees.

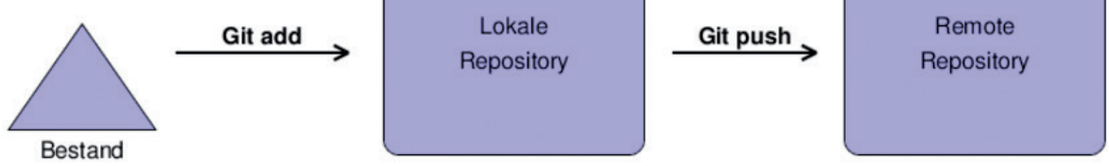
CVS heeft echter ook z'n beperkingen. Zo kan het niet goed omgaan met het hernoemen van bestanden en het verplaatsen van bestanden naar een andere subdirectory. In 2001 kwam Subversion (ook wel SVN), een versiebeheersysteem zonder deze beperkingen.

DERDE GENERATIE

Rond 2005 ontstonden de derde generatie versiebeheersystemen, de zogenaamde gedistribueerde versiebeheersystemen. Het grote verschil ten opzichte van de tweede generatie is het loslaten van een centrale repository en het gebruik maken van een gedistribueerde repository. Elke ontwikkelaar heeft een eigen kopie van de repository en checkt daar bestanden uit en in.

De ontwikkelaars wisselen hun wijzigingen met elkaar uit en houden hun repositories synchroon. De komst van Github versnelde de adaptatie van Git, waardoor dit nu het meest bekende versiebeheersysteem is.

Git add plaatst het bestand in de lokale repository en push stuurt het naar de remote repository



Git is ontwikkeld door Linus Torvalds als versiebeheersysteem voor het managen van de ontwikkeling van de Linux kernel. Git schaalt goed naar grote omgevingen en is erg snel. In RCS en in CVS heeft elk bestand een eigen versienummer. Git kent echter geen versienummers voor individuele bestanden, maar gebruikt een versienummer voor de gehele directory-boom. In feite vormt elke versie een snapshot van deze boom. Voor het nummeren van versies wordt gebruik gemaakt van hashes. Daardoor is een versienummer een lange hexadecimale string.

Git is gemaakt voor een gedistribueerde omgeving, waarin ontwikkelaars verspreid over de gehele wereld met elkaar kunnen samenwerken. Het is dus niet verwonderlijk dat Git over meerdere protocollen kan werken, zoals bijvoorbeeld SSH, HTTP en FTP.

GIT REPOSITORY AANMAKEN

We gaan nu zelf aan de slag met Git. Controleer met **git --version** dat Git op je systeem is geïnstalleerd. Zo niet, installeer het dan via de packagemanager.

Normaal gesproken staat de repository remote op een server: ergens op het internet of in je lokale netwerk. Dit gaan we nabootsen op onze eigen Linux machine. Hiervoor is het nodig dat je met SSH op je eigen systeem kunt inloggen, controleer dit met **ssh localhost**. Wanneer je geen verbindingmet je machine krijgt, moet je waarschijnlijk nog de OpenSSH server installeren, bijvoorbeeld met:

```
sudo apt-get install openssh-server
```

Maak op je systeem een directory voor de oefenrepository aan, bijvoorbeeld met:

```
mkdir ~/oefenrepo.git
```

Binnen deze directory (dus na **cd ~/oefenrepo.git**) doe je nu:

```
git init . –bare
```

Hiermee geven we aan dat we Git willen initialiseren en de punt is in Unix de verkorte notatie voor **deze directory**. Met **--bare** geven we aan, dat we een kale repository willen zonder werk-directory. Git geeft als antwoord dat een lege Git repository is aangemaakt. Met **ls -l** kun je zien dat de nodige bestanden en subdirectories zijn aangemaakt. Het is gebruikelijk dat de directorynaam van kale (bare) repositories de extensie **.git** hebben.

Wij doen alsof onze oefenrepository de remote repository is op een cloudserver en maken een lokale kloon. De extensie .git kunnen we weglaten.

```
git clone user@localhost:oefenrepo
```

In plaats van **user** zet je hier je gebruikersnaam. Door het format **user@hostname:repository** weet Git dat je via SSH wilt klonen. Git geeft je een waarschuwing dat je een lege repository gekloond hebt. Je hebt nu een nieuwe directory met de naam **oefenrepo**. Wanneer je daarin gaat met **cd oefenrepo** zie je met **ls -la** dat je een lege directory hebt, met daarin een verborgen directory genaamd **.git**. Bekijk het config-bestand in deze directory. Onder **remote origin** zie je de URL van de remote repository.

GIT COMMIT

Net als in het vorige artikel over RCS gaan we nu weer een bestand aanmaken, inchecken en vervolgens enkele wijzigingen in Git inchecken.

Maak in de directory **oefenrepo** een eenvoudig tekstbestand met zo'n vijf regels tekst en noem dit **tekstbestand**. We checken dit tekstbestand in (zie **Listing 5**).

Wanneer dit je eerste Git commit is op het huidige systeem is, krijg je de melding dat Git je gebruikersnaam en e-mail heeft gegokt. Via **git config --global** configureer je deze waarden, die daarna bij je toekomstige commits voortaan worden meegegeven.

Git slaat de configuratie op in het configuratiebestand .gitconfig in de root van je home directory. Hieronder volgen de commando's om je naam en e-mailadres in Git te configureren (zie **Listing 6**).

Open het tekstbestand in je editor, voeg een regel toe en sla deze weer op. Wanneer je nu het commando **git status** invoert, dan zie je dat je een nog niet verwerkte wijziging hebt. Geef weer het commando **git add tekstbestand** in. Doe nog geen commit en vraag weer de status op. Git geeft nu aan, dat je een nog te committen wijziging hebt. Met **git commit -m "Wijzig boodschap"** checken we de wijziging in Git in.

Bekijk met **git log** welke versie we hebben en welke wijzigingen zijn geregistreerd.

We committen nog een paar wijzigingen. Open het tekstbestand, wijzig de inhoud van een regel, sla het op en doe weer een **git add** en een **git commit**, bijvoorbeeld met de boodschap "**wijziging van een regel**". Open het tekstbestand weer. Verwijder een regel, sla het op en doe weer een **git add** en **git commit**, bijvoorbeeld met de boodschap "**regel verwijderd**".

GIT BLAME

Doe weer een **git log**. In de output hiervan staat de nieuwste versie bovenaan en de oudste versie onder. Elke versie heeft een versienummer in de vorm van een hash, bijvoorbeeld **1b36d5b91ac49157341707f6a593b91531cf8aae**. Je kunt de wijzigingen ten opzichte van een willekeurige vorige versie opvragen met **git diff <hash>** (zie **Listing 7**).

Wanneer je de **diff** opvraagt met de hash van de allereerste versie van het bestand, dan zie je alle wijzigingen. In dit geval dus van de toegevoegde, gewijzigde en verwijderde regel.

Wanneer meerdere mensen hetzelfde bestand bewerken, is het soms handig om achteraf te kunnen zien, wie wat gedaan heeft. Via Git kunnen we dit opvragen met behulp van het commando **git blame**, gevolgd door de naam van het betreffende bestand. In ons geval is dat dus **git blame tekstbestand**.

Je krijgt daarbij per regel te zien, wie wanneer de laatste wijziging in die regel gemaakt heeft.

LISTING 5

```
git add tekstbestand git commit -m "Bestand aangemaakt"
```

LISTING 6

```
git config --global user.name "Pietje Puk"
git config --global user.email pietjepuk@onsdorp.nl
```

LISTING 7

```
git diff 1b36d5b91ac49157341707f6a593b91531cf8aae
```

LISTING 8

```
mkdir subdir
echo "fietspomp" > subdir/tweedebestand
git status
```

We gaan onze repository uitbreiden met een subdirectory en maken daar een bestand in aan (zie **Listing 8**).

Git meldt dat er bestanden zijn die nog niet getracked worden. We nemen het nieuwe bestand in de repository op met **git add subdir/tweedebestand**, gevolgd door een **git commit -m** met een commit-boodschap. Bij het opvragen laat **git diff** nu de wijzigingen in beide bestanden zien.

Wij hebben nu onze vier W's in werking gezien. "Wat" is gewijzigd, zien we met **git diff**, "Waarom" het is gewijzigd zijn de commit-boodschappen die we met **git log** zien, en "Wanneer" en "Wie" zien we met **git blame**.

PUSH EN PULL

Tot nu toe hebben we in onze lokale kloon gewerkt. We kunnen de wijzigingen naar de remote repository sturen door middel van **git push**. Hiermee worden onze wijzigingen in de remote repository gemerged. Dat wil zeggen dat Git zijn best gaat doen om onze wijzigingen in de remote repository te verwerken.

In ons geval is dat niets bijzonders, maar wanneer meerdere mensen een kloon gemaakt hebben en hun wijzigingen hebben gepushed, dan moet Git meer uitzoekwerk doen om alles op zijn goede plek te krijgen.

Je moet zelf zorgen dat je lokale kloon synchroon blijft met de remote repository. Voordat je in je gekloonde directory aan de gang gaat, doe je daarom meestal eerst in deze directory een **git pull**. Git zoekt dan in het config bestand op wat de remote origin is en update de kloon.

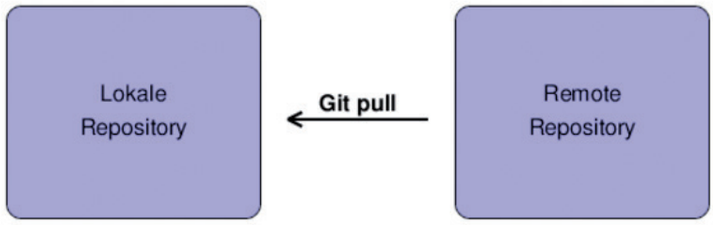
We proberen dit uit door een tweede kloon te maken. Maak in je home directory een nieuwe directory, bijvoorbeeld **tweede_kloon**. Doe daarin weer een **git clone** op dezelfde manier als hierboven. De nieuwe kloon bevat uiteraard de laatste wijzigingen, die wij vanaf de eerste kloon gepushed hebben.

Nu brengen we een wijziging aan, bijvoorbeeld met: **echo "laptop" > derdebestand**. Met **git add** en **git commit -m "derde bestand toegevoegd"** nemen we deze op in de repository. Vervolgens doen we weer een **git push**.

Ga terug naar de eerste kloon met **cd ~/oefenrepo**. Met **ls** zie je dat dit derde bestand er niet is en **git status** zegt ook dat we iets missen. Nu doen we **git pull**. Git start SSH op en maakt de gekloonde repository up to date.

LISTING 9

```
echo "computer" >> derdebestand
git add derdebestand
git commit -m 'regel toegevoegd'.
```



Git pull haalt de laatste wijzigingen van de remoterepository op en update de lokale repository

Nu zien we ons derde bestand wel. We voegen een regel toe aan dit bestand en committen de wijziging (zie **Listing 9**).

Wanneer je nu **git status** doet, dan zegt Git dat we één commit voorlopen op de remote repository. Via **git push** kunnen we nu de wijziging naar de repository sturen.

TOT SLOT

Git is een erg uitgebreid versiebeheersysteem en we hebben hier maar een heel klein tipje van de sluier opgelicht. We hopen dat het genoeg is om je op weg te helpen. Wanneer je besluit om in je eigen netwerk repositories op een server zetten, kijk dan ook eens naar Gitweb. Hiermee krijg je in je browser inzage in al je repositories.