GIT LES

Version control met GIT

J.J. Strootman

INHOUDS-OPGAVE

GIT

WAT IS GIT?

HET WAT EN WAAROM VAN VERSIEBEHEER.

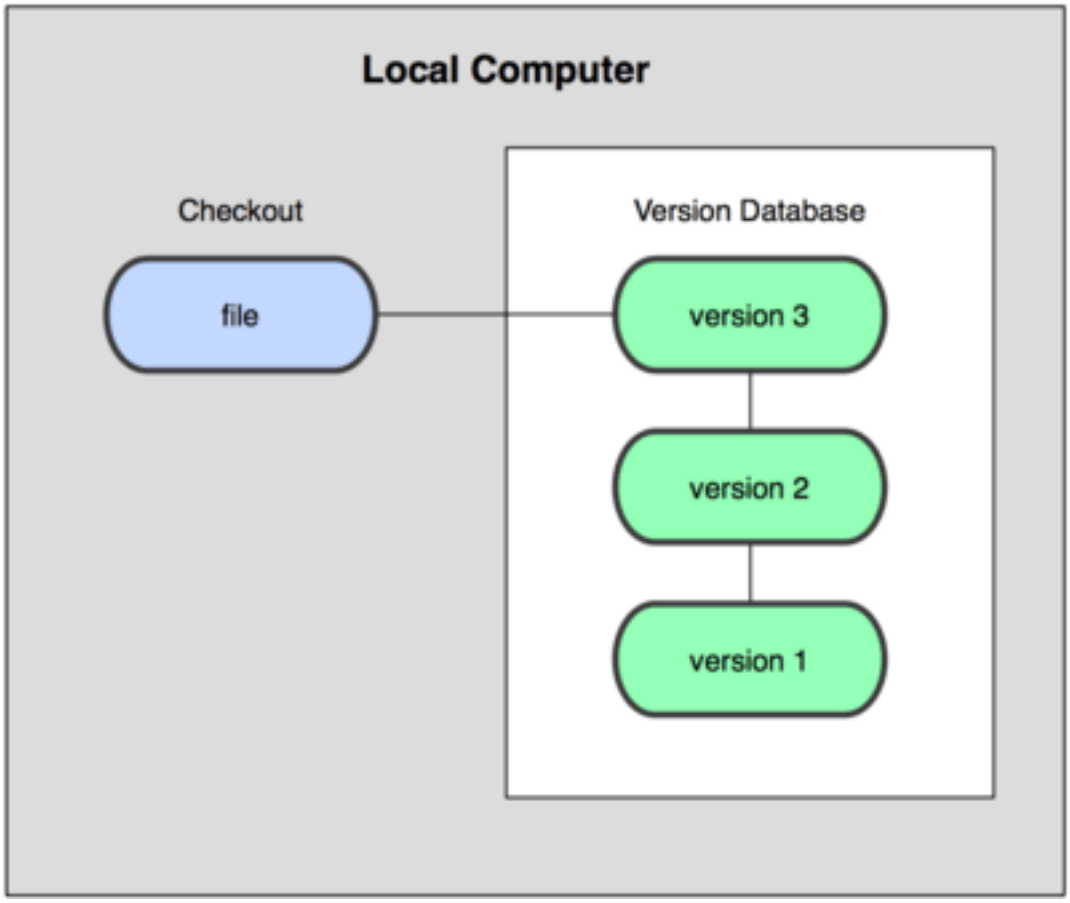
Wat is versiebeheer, en wat heeft dat met jou te maken? Versiebeheer is het systeem waarbij veranderingen in een bestand of groep van bestanden over de tijd wordt bijgehouden, zodat je later specifieke versies kan opvragen. In de voorbeelden in dit boek is het broncode van computersoftware waarvan de versies beheerd worden maar in principe kan elk soort bestand op een computer aan versiebeheer worden onderworpen.

Als je een grafisch ontwerper bent of websites ontwerpt en elke versie van een afbeelding of opmaak wilt bewaren (wat je vrijwel zeker zult willen), is het verstandig een versiebeheersysteem (Version Control System in het Engels, afgekort tot VCS) te gebruiken. Als je dat gebruikt kan je eerdere versies van bestanden of het hele project terughalen, wijzigingen tussen twee momenten in tijd bekijken, zien wie het laatst iets aangepast heeft wat een probleem zou kunnen veroorzaken, wie een probleem heeft veroorzaakt en wanneer en nog veel meer. Een VCS gebruiken betekent meestal ook dat je de situatie gemakkelijk terug kan draaien als je een fout maakt of bestanden kwijtraakt. Daarbij komt nog dat dit allemaal heel weinig extra belasting met zich mee brengt.

LOKALE VERSIEBEHEERSYSTEMEN

Veel mensen hebben hun eigen versiebeheer methode: ze kopiëren bestanden naar een andere map (en als ze slim zijn, geven ze die map ook een datum). Deze methode wordt veel gebruikt omdat het zo simpel is, maar het is ook ongelofelijk foutgevoelig. Het is makkelijk te vergeten in welke map je zit en naar het verkeerde bestand te schrijven, of onbedoeld over bestanden heen te kopiëren.

Om met dit probleem om te gaan hebben programmeurs lang geleden lokale VCS'en ontwikkeld die een simpele database gebruikten om alle veranderingen aan bestanden te beheren (zie Figuur 1-1).

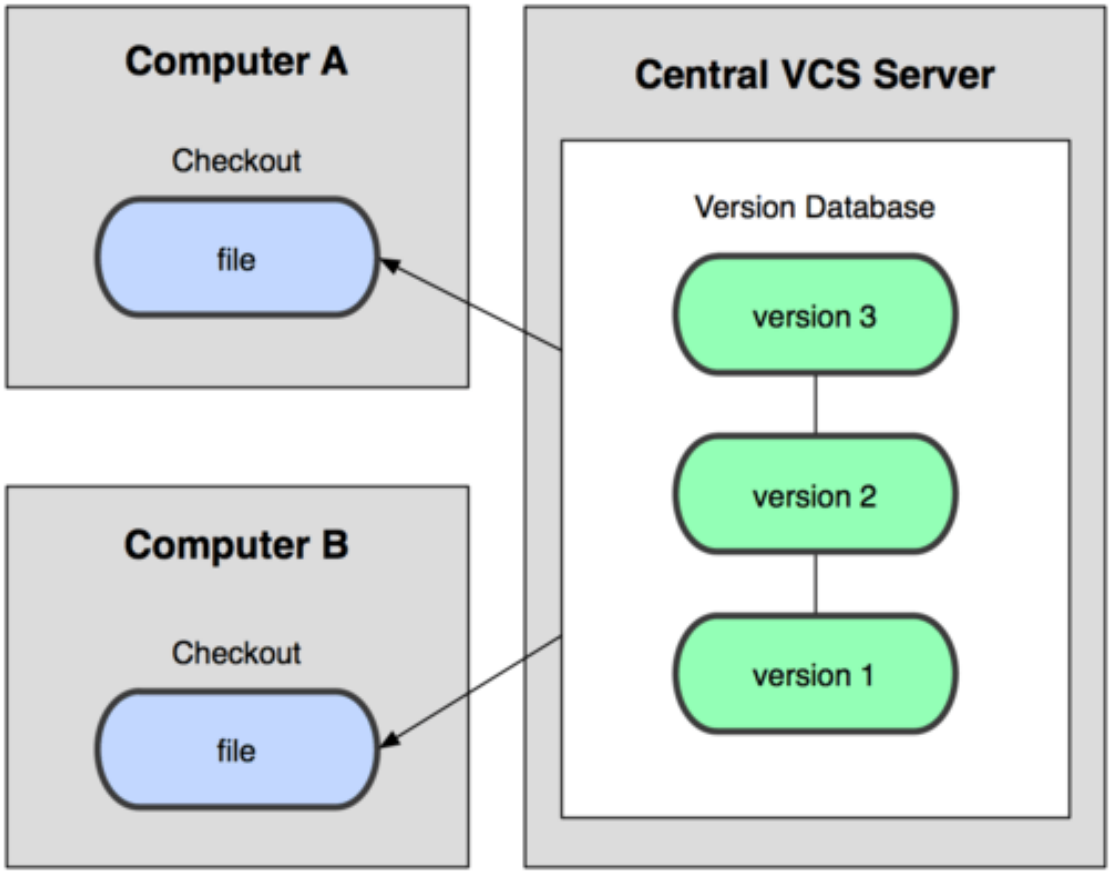


Figuur 1-1. Een diagram van een lokaal versiebeheersysteem.

Een populair gereedschap voor VCS was een systeem genaamd rcs, wat vandaag de dag nog steeds met veel computers wordt meegeleverd. Zelfs het populaire besturingssysteem Mac OS X heeft rcs als je de Developer Tools installeert. Dit gereedschap werkt in principe door verzamelingen van ‘patches’ (de verschillen tussen bestanden) van de opvolgende bestandsversies in een speciaal formaat op de harde schijf op te slaan. Zo kan je een bestand reproduceren zoals deze er uitzag op enig moment in tijd door alle patches bij elkaar op te tellen.

GECENTRALISEERDE VERSIEBEHEERSYSTEMEN

De volgende belangrijke uitdaging waar mensen tegenaan lopen is dat ze samen moeten werken met ontwikkelaars op andere computers. Om deze uitdaging aan te gaan ontwikkelde men Gecentraliseerde Versiebeheersystemen (Centralized Version Control Systems, afgekort CVCSs). Deze systemen, zoals CVS, Subversion en Perforce, hebben één centrale server waarop alle versies van de bestanden staan en een aantal clients die de bestanden daar van ophalen (‘check out’). Vele jaren was dit de standaard voor versiebeheer (zie Figuur 1-2).



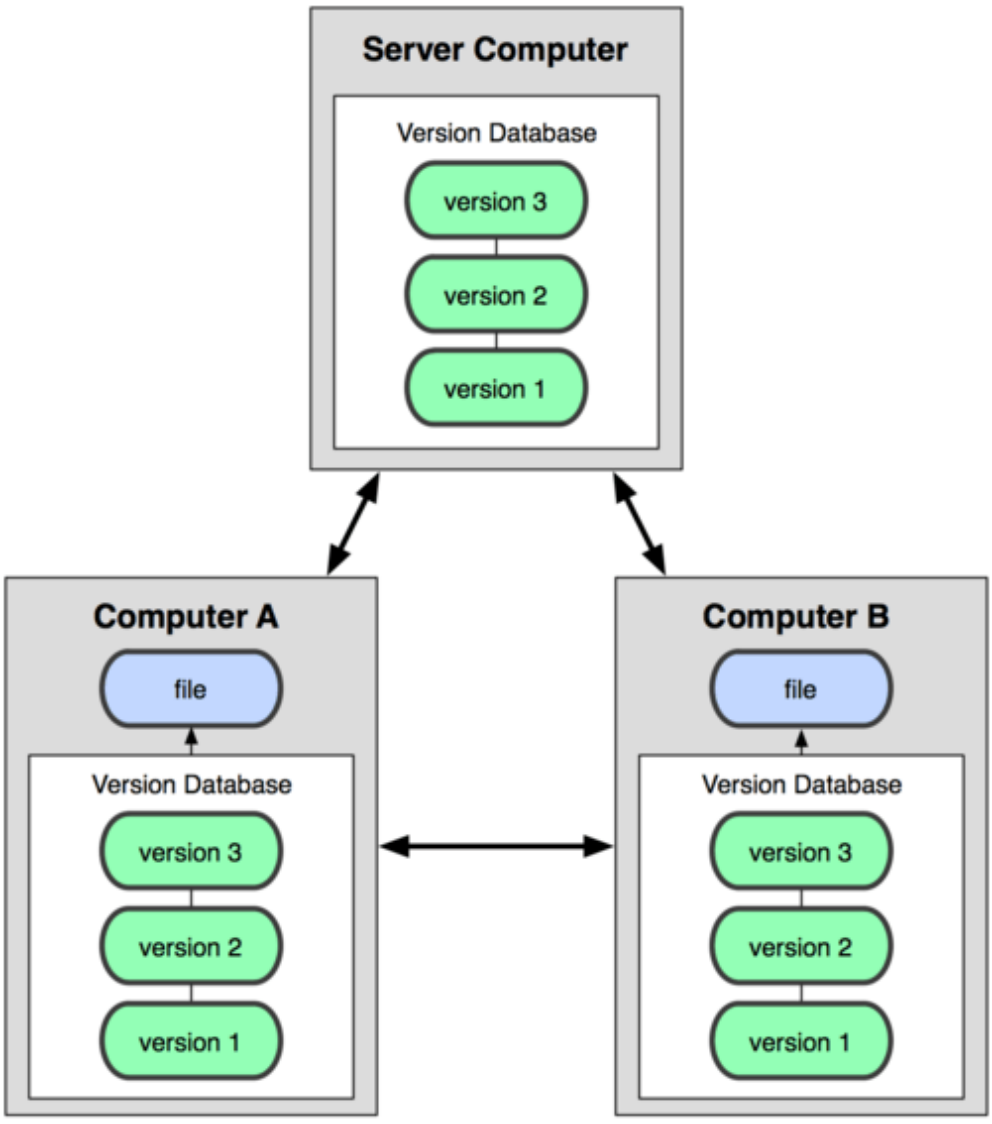
Figuur 1-2. Een diagram van een gecentraliseerd versiebeheersysteem.

Deze manier van versiebeheer biedt veel voordelen, zeker ten opzichte van lokale VCS'en. Bijvoorbeeld weet iedereen, tot op zekere hoogte, wat de overige project-medewerkers aan het doen zijn. Beheerders hebben een hoge mate van controle over wie wat kan doen, en het is veel eenvoudiger om een CVCS te beheren dan te moeten werken met lokale databases op elke cliënt.

Maar helaas, deze methode heeft ook behoorlijke nadelen. De duidelijkste is de 'single point of failure': als de centrale server plat gaat en een uur later weer terug online komt kan niemand in dat uur samenwerken of versies bewaren van de dingen waar ze aan werken. Als de harde schrijf waar de centrale database op staat corrupt raakt en er geen backups van zijn verlies je echt alles; de hele geschiedenis van het project, behalve de momentopnames die mensen op hun eigen computers hebben staan. Lokale VCS'en hebben hetzelfde probleem: als je de hele geschiedenis van het project op één enkele plaats bewaart, loop je ook kans alles te verliezen.

GEDISTRIBUEERDE VERSIEBEHEERSYSTEMEN

En hier verschijnen Gedistribueerde versiebeheersystemen (Distributed Version Control Systems, DVCS's) ten tonele. In een DVCS (zoals Git, Mercurial, Bazaar en Darcs) downloaden cliënten niet simpelweg de laatste momentopnames van de bestanden: de hele opslagplaats (de 'repository') wordt gekopiëerd. Dus als een server neergaat en deze systemen werkten via die server samen dan kan de repository van elke willekeurige cliënt terug worden gekopiëerd naar de server om deze te herstellen. Elke checkout is dus in feite een complete backup van alle data (zie Figuur 1-3).



Figuur 1-3. Diagram van een gedistribueerd versiebeheersysteem

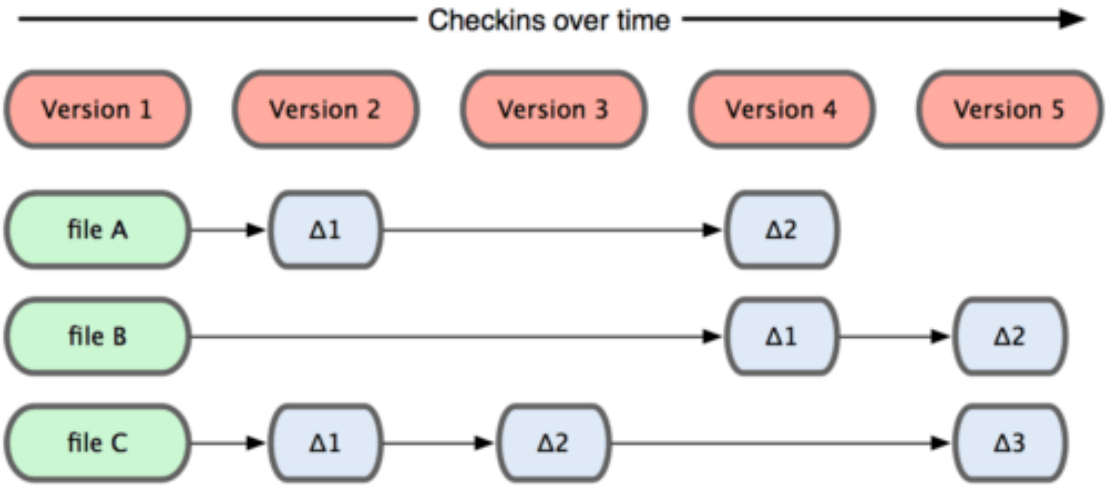
Bovendien kunnen veel van deze systemen behoorlijk goed omgaan met meerdere (niet-lokale) repositories tegelijk, zodat je met verschillende groepen mensen op verschillende manieren tegelijk aan hetzelfde project kan werken. Hierdoor kan je verschillende werkprocessen (‘workflows’) opzetten die niet mogelijk waren geweest met gecentraliseerde systemen zoals hiërarchische modellen.

DE BASIS VAN GIT

Dus, wat is Git in een notendop? Dit is een belangrijke paragraaf om in je op te nemen, omdat, als je goed begrijpt wat Git is en de fundamenten van de interne werking begrijpt, het waarschijnlijk een stuk makkelijker wordt om Git effectief te gebruiken. Probeer, als je Git aan het leren bent, te vergeten wat je al weet over andere VCS'en zoals Subversion en Perforce; zo kan je verwarring bij gebruik door de subtiele verschillen voorkomen. Git gaat op een hele andere manier met informatie om dan die andere systemen, ook al lijken de verschillende commando's behoorlijk op elkaar. Als je die verschillen begrijpt, kan je voorkomen dat je verward raakt als je Git gebruikt.

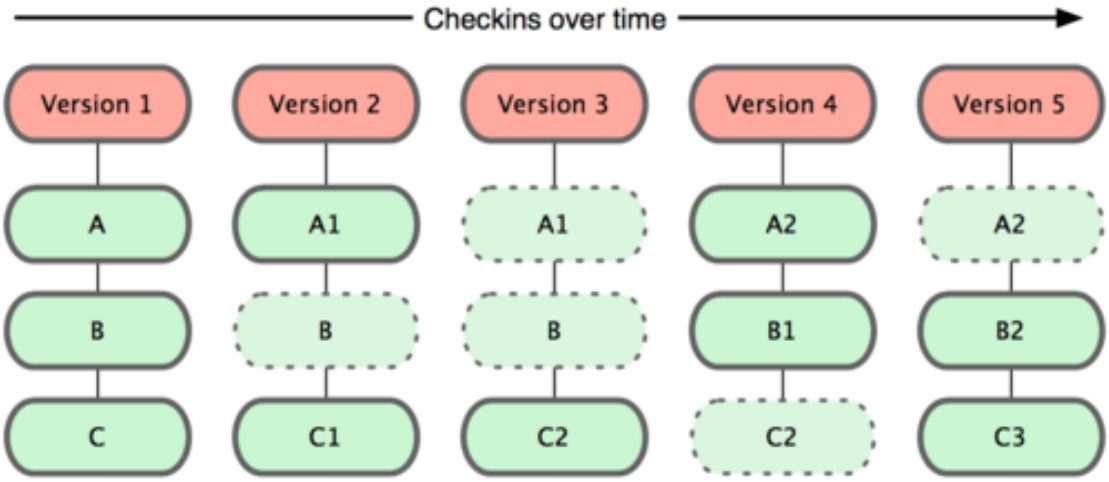
MOMENTOPNAMES IN PLAATS VAN VERSCHILLEN

Een groot verschil tussen Git en elke andere VCS (inclusief Subversion en consorten) is hoe Git denkt over zijn data. Conceptueel bewaren de meeste andere systemen informatie als een lijst van veranderingen per bestand. Deze systemen (CVS, Subversion, Perforce, Bazaar, enzovoort) zien de informatie die ze bewaren als een aantal bestanden en de veranderingen die aan die bestanden zijn aangebracht over de tijd, zoals geïllustreerd in Figuur 1-4.



Figuur 1-4. Andere systemen bewaren data meestal als veranderingen aan een basisversie van elk bestand.

Git ziet en bewaart zijn data heel anders. De kijk van Git op zijn data kan worden uitgelegd als een reeks momentopnames (snapshots) van een miniatuurbestandssysteem. Elke keer dat je 'commit' (de status van je project in Git opslaat) neemt het een soort van foto van hoe al je bestanden er op dat moment uitzien en slaat een verwijzing naar die foto op. Voor efficiëntie slaat Git ongewijzigde bestanden niet elke keer opnieuw op, alleen een verwijzing naar het eerdere identieke bestand dat het eerder al opgeslagen had. In Figuur 1-5 kan je zie hoe Git ongeveer over zijn data denkt.



Figuur 1-5. Git bewaart data als momentopnames van het project.

Dat is een belangrijk onderscheid tussen Git en bijna alle overige VCS'en. Hierdoor moet Git bijna elk aspect van versiebeheer heroverwegen, terwijl de meeste andere systemen het hebben overgenomen van voorgaande generaties. Dit maakt Git meer een soort mini-bestandssysteem met een paar ongelooflijk krachtige gereedschappen, in plaats van niets meer of minder dan een VCS. We zullen een paar van de voordelen die je krijgt als je op die manier over data denkt gaan onderzoeken, als we 'branching' (gesplitste ontwikkeling) toelichten in Hoofdstuk 3.

BIJNA ALLES IS LOKAAL

De meeste handelingen in Git hebben alleen lokale bestanden en hulpmiddelen nodig. Normaalgesproken is geen informatie nodig van een andere computer in je netwerk. Als je gewend bent aan een CVCS, waar de meeste handelingen vertraagd worden door het netwerk, lijkt Git door de goden van snelheid begenadigd met onwereldse krachten. Omdat je de hele geschiedenis van het project op je lokale harde schijf hebt staan, lijken de meeste acties geen tijd in beslag te nemen.

Een voorbeeld: om de geschiedenis van je project te doorlopen hoeft Git niet bij een of andere server de geschiedenis van je project op te vragen; het leest simpelweg jouw lokale database. Dat betekent dat je de geschiedenis van het project bijna direct te zien krijgt. Als je de veranderingen wilt zien tussen de huidige versie van een bestand en de versie van een maand geleden kan Git het bestand van een maand geleden opzoeken en lokaal de verschillen berekenen, in plaats van aan een niet-lokale server te moeten vragen om het te doen, of de oudere versie van het bestand ophalen van een server om het vervolgens lokaal te doen.

Dit betekent dat er maar heel weinig is dat je niet kunt doen als je offline bent of zonder VPN zit. Als je in een vliegtuig of trein zit en je wilt nog even wat werken, kan je vrolijk doorgaan met commits maken tot je een netwerkverbinding hebt zodat je dat werk kunt uploaden. Als je naar huis gaat en je VPN cliënt niet aan de praat krijgt kan je nog steeds doorwerken. Bij veel andere systemen is dat onmogelijk of zeer onaangenaam. Als je bijvoorbeeld Perforce gebruikt kun je niet zo veel doen als je niet verbonden bent met de server. Met Subversion en CVS kun je bestanden bewerken maar je kunt geen commits maken naar je database (omdat die offline is). Dat lijkt misschien niet zo belangrijk maar je zult nog versteld staan wat een verschil het kan maken.

GIT HEEFT INTEGRITEIT

Alles in Git krijgt een controlegetal (‘checksum’) voordat het wordt opgeslagen en er wordt later met dat controlegetal ernaar gerefereerd. Dat betekent dat het onmogelijk is om de inhoud van een bestand of map te veranderen zonder dat Git er weet van heeft. Deze functionaliteit is in het diepste deel van Git ingebouwd en staat centraal in zijn filosofie. Je kunt geen informatie kwijtraken als het wordt verstuurd en bestanden kunnen niet corrupt raken zonder dat Git het kan opmerken.

Het mechanisme dat Git gebruikt voor deze controlegetallen heet een SHA-1-hash. Dat is een tekenreeks van 40 karakters lang, bestaande uit hexadecimale tekens (0–9 en a–f) en wordt berekend uit de inhoud van een bestand of directory-structuur in Git. Een SHA-1-hash ziet er ongeveer zo uit:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Je zult deze hashwaarden overal tegenkomen omdat Git er zoveel gebruik van maakt. Sterker nog, Git bewaart alles niet onder een bestandsnaam maar in de database van Git met de hash van de inhoud als sleutel.

GIT VOEGT NORMAAL GESPROKEN ALLEEN DATA TOE

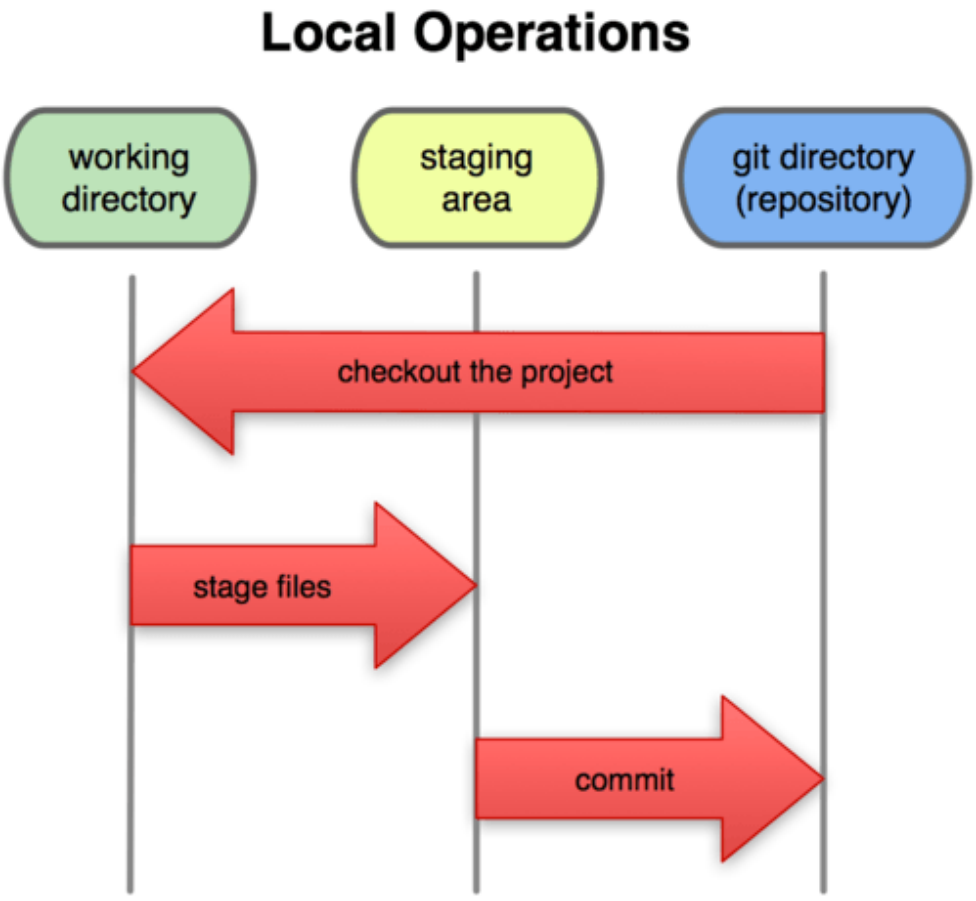
Bijna alles wat je in Git doet, leidt tot toevoeging van data in de Git database. Het is erg moeilijk om het systeem iets te laten doen wat je niet ongedaan kan maken of het de gegevens te laten wissen op wat voor manier dan ook. Zoals met elke VCS kun je veranderingen verliezen of verhaspelen als je deze nog niet hebt gecommit; maar als je dat eenmaal hebt gedaan, is het erg moeilijk om die data te verliezen, zeker als je de lokale database regelmatig uploadt (‘push’) naar een andere repository.

Dit maakt het gebruik van Git zo plezierig omdat je weet dat je kunt experimenteren zonder het gevaar te lopen jezelf behoorlijk in de nesten te werken. Zie Hoofdstuk 9 voor een iets diepgaandere uitleg over hoe Git zijn data bewaart en hoe je de data die verloren lijkt kunt terughalen.

DE DRIE TOESTANDEN

Let nu goed op. Dit is het belangrijkste dat je over Git moet weten als je wilt dat de rest van het leerproces gladjes verloopt. Git heeft drie hoofdtoestanden waarin bestanden zich kunnen bevinden: gecommit ('commited'), aangepast ('modified') en voorbereid voor een commit ('staged'). Committed houdt in dat alle data veilig opgeslagen is in je lokale database. Modified betekent dat je het bestand hebt gewijzigd maar dat je nog niet naar je database gecommit hebt. Staged betekent dat je al hebt aangegeven dat de huidige versie van het aangepaste bestand in je volgende commit meegenomen moet worden.

Dit brengt ons tot de drie hoofdonderdelen van een Gitproject: de Git directory, de werk-directory, en de wachtrij voor een commit ('staging area')



Figuur 1-6. Werk-directory, wachtrij en Git directory

De Git directory is waar Git de metadata en objectdatabase van je project opslaat. Dit is het belangrijkste deel van Git. Deze directory wordt gekopieerd wanneer je een repository kloont vanaf een andere computer.

De werk directory is een checkout van een bepaalde versie van het project. Deze bestanden worden uit de gecomprimeerde database in de Git directory gehaald en op de harde schijf geplaatst waar jij ze kunt gebruiken of bewerken.

De wachtrij is een simpel bestand, dat zich normaalgesproken in je Git directory bevindt, waar informatie opgeslagen wordt over wat in de volgende commit meegaat. Het wordt soms de index genoemd, maar tegenwoordig wordt het de staging area genoemd.

De algemene workflow met Git gaat ongeveer zo:

1. *Je bewerkt bestanden in je werk directory.*

2. *Je bereidt de bestanden voor (staged), waardoor momentopnames (snapshots) worden toegevoegd aan de staging area.*

3. *Je maakt een commit. Een commit neemt alle snapshots van de staging area en bewaart die voorgoed in je Git directory.*

Als een bepaalde versie van een bestand in de Git directory staat, wordt het beschouwd als gecommit. Als het is aangepast, maar wel aan de staging area is toegevoegd, is het staged. En als het veranderd is sinds het was uitgechecked maar niet staged is, is het aangepast. In Hoofdstuk 2 leer je meer over deze toestanden en hoe je er je voordeel mee kunt doen, maar ook hoe je de staging area compleet over kunt slaan.

EEN PRAKTISCHE CASE

We gaan aan de hand van een praktische case laten zien hoe we met GIT moeten werken. Op Github.com wordt een git repository beschikbaar gesteld welke tijdens de lessen zal worden gebruikt.

GIT INSTALLEREN

We gaan GIT gebruiken door middel van de commandline tool. Onder het besturingssysteem OS X en Linux hoeven we alleen de tool zelf te installeren. Dit kan op verschillende manieren. Op de website van GIT kun je terugvinden hoe je dit moet doen.

Onder Windows gaan we met Git Bash een Linux terminal omgeving installeren, zodat we op dezelfde manier als onder OS X en Linux van een commandline omgeving gebruik kunnen maken en daarbij dezelfde commandline opdrachten kunnen gebruiken. Zoek in de Google zoekmachine maar op de term Git Bash en je vindt een duidelijke link naar de download. Installeer deze en zet een snelkoppeling op het bureaublad.

GIT KLAARMAKEN VOOR EERSTE GEBRUIK

Nu je Git op je computer hebt staan, is het handig dat je een paar dingen doet om je Git omgeving aan je voorkeuren aan te passen. Je hoeft deze instellingen normaliter maar één keer te doen, ze blijven hetzelfde als je een nieuwe versie van Git installeert. Je kunt ze ook op elk moment weer veranderen door de commando's opnieuw uit te voeren.

Git bevat standaard een stuk gereedschap genaamd `git config`, waarmee je de configuratie-eigenschappen kunt bekijken en veranderen, die alle aspecten van het uiterlijk en gedrag van Git regelen. Deze eigenschappen kunnen op drie verschillende plaatsen worden bewaard:

Het bestand `/etc/gitconfig`: Bevat eigenschappen voor elk account op de computer en al hun repositories.

Als je de optie `--system` meegeeft aan `git config`, zal het de configuratiegegevens in dit bestand lezen en schrijven.

Het bestand `~/.gitconfig`: Eigenschappen voor jouw account. Je kunt Git dit bestand laten lezen en schrijven door de optie `--global` mee te geven.

Het configuratiebestand in de Git directory (dus `.git/config`) van de repository die je op het moment gebruikt: Specifiek voor die ene repository. Elk niveau neemt voorrang boven het voorgaande, dus waarden die in `.git/config` zijn gebruikt zullen worden gebruikt in plaats van die in `/etc/gitconfig`.

Op systemen met Windows zoekt Git naar het `.gitconfig`-bestand in de `$HOME` directory (`%USERPROFILE%` in een Windows omgeving) wat zich vertaalt in `C:\Documents and Settings\$USER` of `C:\Users\$USER` voor de meesten, afhankelijk van de versie (`$USER` is `%USERNAME%` in Windows omgevingen). Het zoekt ook nog steeds naar `/etc/gitconfig`, maar dan gerelateerd aan de plek waar je MSys hebt staan, en dat is de plek is waar je Git op je Windowscomputer geïnstalleerd hebt.

JOUW IDENTITEIT

Het eerste wat je zou moeten doen nadat je Git geïnstalleerd hebt, is je gebruikersnaam en e-mail adres opgeven. Dat is belangrijk omdat elke commit in Git deze informatie gebruikt, en het onveranderlijk ingebed zit in de commits die je ronddeelt:

```
$ git config --global user.name "John Doe"

$ git config --global user.email johndoe@example.com
```

Nogmaals, dit hoef je maar één keer te doen als je de `--global` optie erbij opgeeft, omdat Git die informatie zal gebruiken voor alles wat je doet op dat systeem. Als je een andere naam of e-mail wilt gebruiken voor specifieke projecten, kun je het commando uitvoeren zonder de `--global` optie als je in de directory van dat project zit.

JE TEKSTVERWERKER

Nu Git weet wie je bent, kun je de standaard tekstverwerker instellen die gebruikt zal worden als Git je een bericht in wil laten typen. Normaliter gebruikt Git de standaardtekstverwerker van je systeem, wat meestal Vi of Vim is. Als je een andere tekstverwerker wilt gebruiken, zoals Emacs, kun je het volgende doen:

```
$ git config --global core.editor emacs
```

JE DIFFPROGRAMMA

Een andere nuttige optie die je misschien wel wilt instellen is het standaard diffprogramma om mergeconflicten op te lossen. Stel dat je vimdiff wilt gebruiken:

```
$ git config --global merge.tool vimdiff
```

Git accepteert kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge en opendiff als geldige merge tools. Je kunt ook een ander programma gebruiken, zie Hoofdstuk 7 voor meer informatie daarover.

JE INSTELLINGEN CONTROLEREN

Als je je instellingen wilt controleren, kan je het `git config --list` commando gebruiken voor een lijst met alle instellingen die Git vanaf die locatie kan vinden:

```
$ git config --list
user.name=Scott Chacon
user.email=schacon@gmail.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

Je zult sommige sleutels misschien meerdere keren langs zien komen, omdat Git dezelfde sleutel uit verschillende bestanden heeft gelezen (bijvoorbeeld `/etc/gitconfig` en `~/.gitconfig`). In dit geval gebruikt Git de laatste waarde van elke unieke sleutel die het tegenkomt.

Je kan ook bekijken wat Git als instelling heeft bij een specifieke sleutel door git config {sleutel} in te voeren:

```
$ git config user.name
Scott Chacon
```

HULP KRIJGEN

Als je ooit hulp nodig hebt terwijl je Git gebruikt, zijn er drie manieren om de gebruiksaanwijzing (manpage) voor elk Git commando te krijgen:

```
$ git help <actie>

$ git <actie> --help

$ man git-<actie>
```

Bijvoorbeeld, je kunt de gebruikershandleiding voor het config commando krijgen door het volgende te typen:

```
$ git help config
```

Deze commando's zijn prettig omdat je ze overal kunt opvragen, zelfs als je offline bent.

COMMANDLINE OPDRACHTEN

Toon de actieve repo

Dit is dus de repo waarin je feitelijk je code maakt.

Opdracht:

git status

```
$ git status
On branch les05-php-41a
Your branch is up-to-date with 'origin/les05-php-41a'.
nothing to commit, working directory clean
```

Voorbeeld 1

In bovenstaande screenshot, Voorbeeld 1, zie je dat hier gewerkt wordt aan de branch les05-php-41a.

Nieuwe branch maken

Op basis van de branch die nu actief is een nieuwe branch maken.

Opdracht:

git checkout –b <nieuwe naam van de branch>

```
$ git checkout -b les05-php-voorbeeld
Switched to a new branch 'les05-php-voorbeeld'
```

Voorbeeld 2

Hierboven, Voorbeeld 2, zie je hoe een nieuwe branch met de naam les05-php-voorbeeld is aangemaakt. Deze is dan ook gelijk actief.

Nieuwe branch pushen naar Github

Als we een nieuwe branch aanmaken is deze niet automatisch ook aangemaakt op Github. Dit moet je dan wel handmatig doen.

Opdracht:

git push –set-upstream origin <nieuwe naam van de branch>

```
$ git push --set-upstream origin les05-php-voorbeeld
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/johanstr/les05-php.git
 * [new branch]      les05-php-voorbeeld -> les05-php-voorbeeld
Branch les05-php-voorbeeld set up to track remote branch les05-php-voorbeeld fro
m origin.
```

Voorbeeld 3

We hebben nu de nieuwe branch ook op Github gezet.

Inloggegevens lokaal opslaan

Iedere keer als je gaat pushen naar Github zal normaal gesproken om je gebruikersnaam en wachtwoord worden gevraagd. Als je dit niet steeds weer opnieuw wil invoeren dan kun je de volgende opdracht gebruiken om deze gegevens lokaal te registreren. Dan hoef je 'm slechts nog 1 keer in te tikken en daarna niet meer.

Opdracht:

git config –global credential.helper wincred

Nieuwe repo aanmaken

In dit voorbeeld is een nieuwe lege repo op Github al aangemaakt. En heb je op je eigen schijf in de rootmap van je project al bestanden.

Ga via de terminal eerst naar de rootmap van je project.

Opdracht:

git init

git add .

git commit –m "je opmerkingen bij deze commit"

git remote add origin <url van de repo op github>.git

git push –u origin master

```
$ git init
Initialized empty Git repository in /Users/docent/Documents/htdocs/school/1516/klas1/progra
mmeren/41a-php/git-workshop/.git/
```

```
$ git add .
```

```
$ git commit -m "Eerste commit"
[master (root-commit) 6c431ce] Eerste commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 index.php
```

```
$ git remote add origin https://github.com/johanstr/git-workshop.git
```

```
$ git push -u origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 216 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/johanstr/git-workshop.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

Voorbeeld 4

Status van je werk

Soms wil je weten wat voor wijzigingen er allemaal geregistreerd zijn in je lokale repo en of er iets is om te pushen naar Github.

Opdracht:

git status

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   index.php

no changes added to commit (use "git add" and/or "git commit -a")
```

Voorbeeld 5

In bovenstaande screenshot, Voorbeeld 5, zie je dat een bestand is gewijzigd sinds de laatste commit.

Pushen van je werk

Hiermee stel je je werk veilig op Github en in je lokalen repo.

Opdracht:

git add .

git commit –m "je opmerkingen bij deze commit"

git push

```
$ git add .
```

```
$ git commit -m "Tweede commit"
[master 5f3faf4] Tweede commit
 1 file changed, 4 insertions(+)
```

```
$ git push
Counting objects: 3, done.
Writing objects: 100% (3/3), 268 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/johanstr/git-workshop.git
   6c431ce..5f3faf4  master -> master
```

Voorbeeld 6

HET PROJECT

Als voorbeeldproject nemen we een applicatie die met het PHP framework Laravel gemaakt wordt.

Beginsituatie

Git Master

We hebben het project al klaar staan op onze lokale harde schijf. We hebben nog niks aan de code gedaan. En dat gaan we ook zeker niet doen. Want dit wordt de beginsituatie voor alle developers in ons team. Wat we eventueel wel doen is alle (php-)bestanden toevoegen die voor het gehele project nog van belang zullen zijn.

Stap 1

Maak op Github een lege repository aan met een naam die duidelijk verwijst naar het project en het product wat zal worden gemaakt. Als je dat goed gedaan hebt krijg je een overzicht te zien op Github die je verteld wat je moet doen om lokaal het project te koppelen aan deze online repo.

Dit wordt in ons geval:

Voor klas B-ITB4-1a

https://github.com/johanstr/forum-41a

En voor de clustergroep AO

https://github.com/johanstr/forum-41bcd

Stap 2

Maak in de map van je lokale project een git repo aan met de commando:

```
$ git init
```

Hiermee initialiseren we de map als een lokale git repository.

Stap 3

Nu gaan we de lokale git repo koppelen aan de online repo die we in stap 1 hebben aangemaakt.

```
$ git remote add origin https://github.com/johanstr/forum-41a.git
```

Wat we hier hebben gedaan is eigenlijk niets anders dan het koppelen van de lokale git repo aan de online git repo, waarbij we de master branch de naam origin hebben gegeven.

Stap 4

We gaan nu alle bestanden, die lokaal in onze projectmap staan, toevoegen aan de lokale git repo en daarna pushen we de gehele lokale git repo naar de master branch in de online repo.

```
$ git add .

$ git commit –m “Project Forum opstart”

$ git push –u origin master
```

De opdracht git push –u ….. is eenmalig en dus alleen hier nodig.

Teamleden/Developers

Teamleden van het project halen vervolgens het project binnen op hun eigen harde schijf met de volgende opdracht:

```
$ git clone https://github.com/johanstr/forum-41a.git forum-41a
```

Met het laatste deel van de opdracht (forum-41a) geven we aan dat we de clone repo in de map forum-41a willen hebben. Als we dit niet doen dan wordt het standaard in een map geplaatst met exact dezelfde naam als de naam van de repo.

LET OP!!!!

Je programmeert nooit in de master branch. Deze branch is uitsluitend bedoeld om de versies bij te houden en al het werk van de verschillende developers in samen te voegen. Alleen de Git Master werkt met de master branch. Hij/zij is degene die alle andere branches, zoals van de developers, hier weer samenvoegt tot één geheel. Vanuit de master branch is de Git Master ook degene die uiteindelijke de branch voor release opbouwt.

Developers

Alle developers weten aan welk deel van de applicatie zij zullen gaan werken. Stel we hebben twee developers. Zij hebben de volgende verdeling afgesproken: Developer 1 ontwikkelt de login, Developer 2 ontwikkelt het registreren van nieuwe gebruikers.

Developer 1

Developer 1 maakt aan het begin van zijn werk een nieuwe branch aan met een naam die voldoet aan de afspraken die binnen het team en met de Git Master zijn gemaakt.

```
$ git checkout –b feature/login
```

In dit geval is er gekozen voor de naam feature/login.

Hiermee vertellen we dat het om een feature in de applicatie gaat en in dit specifieke geval om de login feature.

De branch is dan ook gelijk actief en er kan worden begonnen met coderen.

Developer 2

Developer 2 maakt aan het begin van zijn werk een nieuwe branch aan.

```
$ git checkout –b feature/registration
```

De branch is gelijk actief en er kan worden begonnen met coderen.

Je werk opslaan in de lokale repo

Beide developers doen het volgende

```
$ git dd .
$ git commit –m “Een duidelijk bericht die verteld wat er gedaan is”
$ git push
```

Samenvoegen van het werk van de developers met de master branch

Wanneer de developers hun werk hebben opgeslagen in de lokale en online repo (zie hierboven) kan de Git Master het werk samenvoegen in de master branch met de volgende opdrachten:

```
$ git merge feature/login
$ git merge feature/registration
$ git push
```

INDEX