

# Van start met GitHub & git

Continuous Integration Basics

## INHOUD

<b>1</b>	<b>INLEIDING</b>	<b>4</b>
<b>2</b>	<b>WAT IS CONTINUOUS INTEGRATION?</b>	<b>5</b>
2.1	Definitie	5
2.2	Basis stappen softwareontwikkeling	6
2.3	Basis Continuous Integration stappen	7
2.3.1	Schematische voorstelling Continuous Integration	8
2.4	Waarom is er iets als Continuous Integration nodig?	8
<b>3</b>	<b>CONTINUOUS INTEGRATION - DELIVERY - DEPLOYMENT</b>	<b>9</b>
3.1	Continuous	9
3.2	Continuous Integration	10
3.3	Continuous Delivery	10
3.4	Continuous Deployment	10
3.5	Schematische voorstelling van de verschillen	11
<b>4</b>	<b>GITHUB SETUP</b>	<b>12</b>
4.1	Account aanmaken	12
4.2	Account configuratie (verplicht)	14
4.3	GitHub Student Pack (vrijblijvend)	15
<b>5</b>	<b>WAT IS GIT?</b>	<b>16</b>
<b>6</b>	<b>GIT INSTALLATIE</b>	<b>17</b>
6.1	Downloaden van git	17
6.2	Aanbevolen installatie instellingen	17
6.2.1	Windows explorer integration	17
6.2.2	Default editor	17
6.2.3	Initial branch name	17
6.2.4	Path environment	18
6.2.5	Git Credential Manager	18
6.2.6	Overige installatievensters	18
6.3	Verdere configuratie	18
<b>7</b>	<b>VAN START MET GIT</b>	<b>20</b>
7.1	Een eerste kennismaking	20

7.1.1	Een git repository aanmaken	20
7.1.2	Bestanden toevoegen aan je repository (stagen)	21
7.1.3	Wijzigingen committen	22
7.1.4	Een bestand gestaged dat je niet wou stagen?	23
<b>7.2</b>	<b>Hoe werkt GIT nu?</b>	<b>24</b>
<b>7.3</b>	<b>De drie toestanden in git</b>	<b>25</b>
7.3.1	Working directory	25
7.3.2	Staging Area	25
7.3.3	Git directory (repository)	25
<b>7.4</b>	<b>Algemene workflow met git</b>	<b>26</b>
<b>8</b>	<b>EEN AANTAL COMMANDO'S</b>	<b>27</b>

---

## 1 INLEIDING

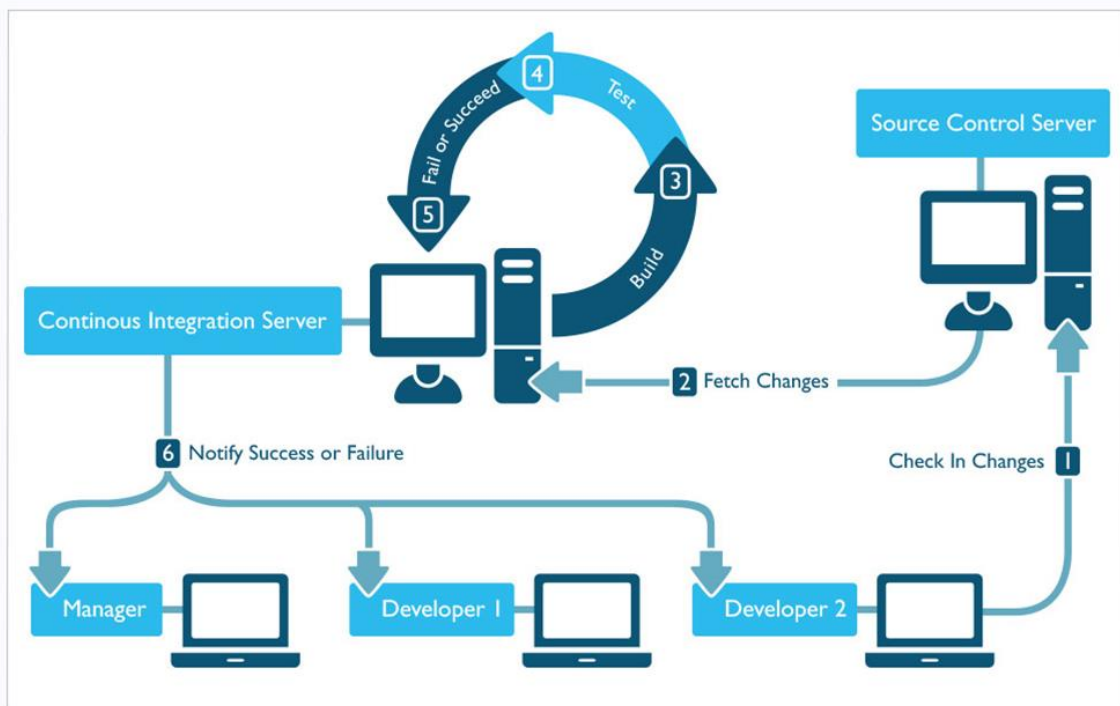
Als developer zijn we steeds gedreven om voor onze klant(en) de best mogelijke applicaties op te leveren. Dit alles uiteraard met een minimum aan extra werk.

We kunnen er echter niet om heen dat applicaties, overheen de tijd, ook heel vaak evolueren in complexiteit. Desondanks de evoluties in de editors die we ter beschikking hebben (*Visual Studio Community, Visual Studio Code, ...*) en de evoluties in het **.NET framework** is en blijft het maken van *grotere (soms ook complexere)* applicaties geen sinecure.

Een van de zaken die ons met bovenstaand probleem helpen kan en tevens ook de sleutel vormt tot het verbeteren van applicaties en productiviteit is het automatiseren van een deel van het werk. **Continuous Integration (CI)** is een van de beste manieren om dit te doen.

In deze module maak je kennis met een aantal bouwstenen in **Continuous Integration**. Deze bouwstenen vormen de fundering waarop je in alle andere modules in deze opleiding verder zal bouwen. Uitermate belangrijke materie dus!

In dit hoofdstuk kijken we in de eerste plaats even waar **Continuous Integration** over gaat en waarom je dit zou moeten gebruiken. We verduidelijken ook even de verschillen tussen **Continuous Integration**, **Continuous Deployment** en **Continuous Delivery**. Op deze manier kan je alle begrippen wat beter in de juiste context gaan plaatsen voor we wat dieper ingaan op de materie zelf.



1-1 Continuous Integration process overzicht © pepgotesting.com

## 2 WAT IS CONTINUOUS INTEGRATION?

### 2.1 DEFINITIE

Er zijn vele definities terug te vinden als we even op zoek gaan online naar **Continuous Integration**. Een van de betere definities is onderstaande:

*Continuous Integration is a **software development** practice where members of a team **integrate their work frequently**, usually each person integrates at least daily leading to multiple integrations per day. Each integration is **verified** by an **automated build (including test)** to **detect integration errors as quickly as possible**. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.*

- Martin Fowler <https://www.martinfowler.com/>

Als we bovenstaande definitie vrij vertalen krijgen we het volgende:

**Continuous Integration** is een softwareontwikkeling praktijk waarbij leden van een ontwikkelteam geregeld hun werk integreren, vaak zal elke persoon dit minstens dagelijks doen. Dit leidt tot meerdere integraties per dag. Elke integratie wordt gecontroleerd door een automatische build van de code (inclusief testen) om integratie fouten zo snel mogelijk te detecteren.

Zoals vaak het geval met definities, schetsen deze uiteindelijk de situatie waar je als team naartoe dient te streven als je **Continuous Integration** omarmt binnen als onderdeel van je softwareontwikkeling. Aangezien we op dit ogenblik echter nog aan het prille begin van de opleiding staan zullen we hier stap voor stap naartoe werken in deze module.

## 2.2 BASIS STAPPEN SOFTWAREONTWIKKELING

Er zijn een heel reeks softwareontwikkeling processen/methodologieën beschikbaar, los van wat uiteindelijk als proces/methodologie gekozen wordt zal je uiteindelijk als ontwikkelaar zo goed als zeker volgende stappen gaan uitvoeren tijdens het schrijven van je code.

1. Haal de nodige bestanden op vanuit de source code repository
2. Breng wijzigingen aan in de code
3. Klik op **Build** in het Visual Studio-menu *(en hoop dat alles wordt gecompileerd)*
4. Ga terug naar **stap 2** *(je had waarschijnlijk een of meer compilatiefouten)*
5. Voer **Unit Tests** uit en hoop dat alle vinkjes groen kleuren *(we gaan ervan uit dat er Unit Tests zijn)*
6. Ga terug naar **stap 2** *(een of meerdere Unit Tests falen, er dient nog iets aangepast te worden)*
7. Wijzig je code, zodat deze begrijpelijker/leesbaarder wordt, ga vervolgens terug naar **stap 5**
8. Maak een update van je code naar de source code repository

Als bovenstaande stappen zo goed als zeker altijd voorkomen tijdens het schrijven van code, waar zal **Continuous Integration** dan een meerwaarde gaan vormen voor jou als ontwikkelaar?

Wel een aantal van deze stappen worden soms bewust of onbewust over het hoofd gezien door een developer. Werk je alleen aan een stuk software dan loop je verder in het ontwikkelproces tegen problemen aan die je zelf in een eerdere stap gemaakt hebt. Wat niet alleen voor de nodige frustratie maar ook voor heel wat tijdverlies kan zorgen. Werk je in team, dan kan het gerust zijn dat jouw wijzigingen in een stuk code *(welke ook gebruikt wordt door andere developers)* iets op een andere plaats stuk maakt. Of dat jouw code ervoor zorgt dat de uitkomst van een bepaald stukje logica er ineens heel anders uitziet!



## 2.3 BASIS CONTINUOUS INTEGRATION STAPPEN

**Continuous Integration** zal ervoor zorgen dat je voor een paar van de stappen in je softwareontwikkeling de nodige ondersteuning hebt. Ondersteuning die ervoor zorgt dat je kwalitatieve, werkende en correcte code gaat schrijven, voor jezelf, maar ook voor alle teamleden die meewerken aan het softwareproject.

1. Haal de nodige bestanden op vanuit de source code repository
2. Breng wijzigingen aan in de code
3. Klik op **Build** in het Visual Studio-menu *(en hoop dat alles wordt gecompileerd)*
4. Ga terug naar **stap 2** *(je had waarschijnlijk een of meer compilatiefouten)*
5. Voer **Unit Tests** uit en hoop dat alle vinkjes groen kleuren *(we gaan ervan uit dat er Unit Tests zijn)*
6. Ga terug naar **stap 2** *(een of meerdere Unit Tests falen, er dient nog iets aangepast te worden)*
7. Wijzig je code, zodat deze begrijpelijker/leesbaarder wordt, ga vervolgens terug naar **stap 5**
8. Maak een update van je code naar de source code repository

Is dit nu niet exact hetzelfde dan wat we hierboven overlopen hadden? Klopt! Echter zal **Continuous Integration** dit process gaan aanvullen met de nodige ondersteuning waar we het zopas over hadden.

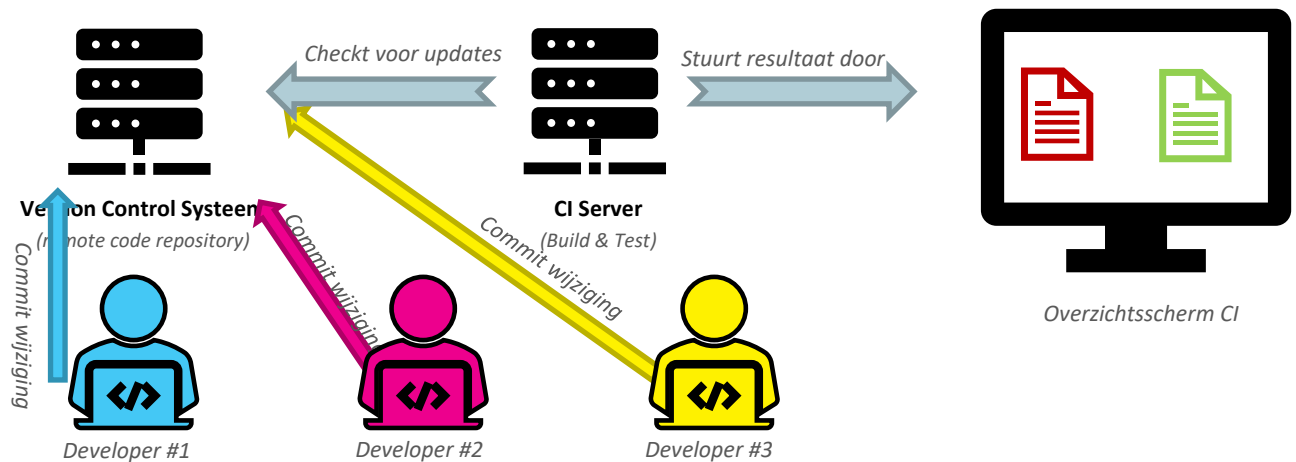
9. Een geautomatiseerd systeem houdt je code repository in de gaten en als er updates gemaakt worden haalt deze de laatste versie van de code op
10. Het geautomatiseerd systeem build vervolgens de code
11. Na een succesvolle build zal dit systeem ook alle Unit Tests gaan uitvoeren
12. Na deze stappen stuurt dit systeem de nodige feedback naar de developer/het development team. Op deze manier is iedereen op de hoogte van de huidige status van het softwareproject.

Maar als je zelf nu stap 1 t.e.m. 8 telkens correct en aandachtig gaat uitvoeren heeft dit dan nog nut om alles nogmaals te doorlopen? Zeker! Het automatiseren van het bouwen, testen en uitvoeren van andere processen via **CI** zorgt ervoor dat de code van meerdere personen correct integreert, compileert en functioneert. Deze code kan dan ook zonder probleem op een andere computer dan de jouwe worden gereproduceerd. Zelfs al werk je alleen, het consequent gebruik van **CI** zal je helpen om de kwaliteit van je code te verbeteren.

Er zijn tal van opties en een evenredig aantal tools die je kan gebruiken om met **Continuous Integration** aan de slag te gaan. In deze module zal je kennis maken met **git** *(als distributed version control system)*, **GitHub** *(cloud-based service voor het Git version control system)* en **xUnit** *(Unit Test framework)*.



### 2.3.1 SCHEMATISCHE VOORSTELLING CONTINUOUS INTEGRATION



Als een build mislukt, blokkeert het CI-systeem de voortgang naar verdere fases/stages. Het team ontvangt een rapport en herstelt de build zo snel mogelijk, meestal wordt deze actie als prioritair door het team behandeld.

### 2.4 WAAROM IS ER IETS ALS CONTINUOUS INTEGRATION NODIG?

De meeste technologiebedrijven maken op vandaag gebruik van **CI**. Door deze manier van werken wordt het softwareontwikkelingsproces voorspelbaar en betrouwbaar. Dit levert dan ook een aantal voordelen op:

- Kleinere/beheersbare wijzigingen in de code
- Snelle isolatie van fouten in de code
- Sneller in staat een fout op te lossen
- Betrouwbaarheid van de testen verhoogt *(door de gelimiteerde grootte van de wijzigingen)*
- Kleinere backlog *(kleine defecten komen naar boven d.m.v. testen)*
- Verhoging van transparantie en verantwoordelijkheid in het team *(rapportage)*
- Verlaagde kost *(automatisering verlaagt de kans op fouten)*

Kleine, gecontroleerde, code veranderingen kunnen vaak en snel uitgevoerd worden. Door alle integratiestappen te automatiseren vermijden ontwikkelaars repetitief werk en menselijke fouten. De beslissing over wanneer, en hoe, tests moeten uitgevoerd worden ligt niet langer bij de developer maar bij de **CI**-setup. Deze zal alle tests, na elke commit, uitvoeren en het testresultaat weergeven.



### 3 CONTINUOUS INTEGRATION - DELIVERY - DEPLOYMENT

Als je even gaat rondneuzen op het internet kom je al snel een aantal andere termen in het **Continuous** verhaal tegen. Naast **Continuous Integration** zal je ook **Continuous Delivery** en **Continuous Deployment** terugvinden. Maar waar zitten nu de verschillen? Laat ons eerst met het gemeenschappelijke beginnen, **Continuous**.

#### 3.1 CONTINUOUS

**Continuous** betekent in de context van alle drie automatisering. Automatisering bij het omzetten van broncode in te leveren producten, bij testen en valideren, en zelfs bij het installeren en configureren van software.

Het betekent echter niet dat deze processen continu worden uitgevoerd. Maar wel dat wanneer een nieuwe verandering in het systeem wordt geïntroduceerd, deze automatisch snel kan worden verwerkt. Op deze manier krijg je dus frequenter builds en snellere tests, packages en releases ten opzichte van andere manieren van werken. Verder zorgt dit uiteraard voor snelle identificatie en melding van storingen, zodat deze snel kunnen worden opgelost. Een term die hiervoor gebruikt wordt is *fail-fast*.

**Continuous** impliceert hier ook dat een verandering kan plaatsvinden, zonder menselijke tussenkomst, door de fasen van builds, testen, verpakking, etc. Gecombineerd vormen deze fasen een '*pipeline*' van processen en applicaties om wijzigingen in de broncode op te nemen en om te zetten in een product, klaar voor release. Deze *pipeline* kan verschillende namen hebben, waaronder *Continuous Delivery Pipeline*, *Deployment Pipeline*, of *Release Pipeline*.



### 3.2 CONTINUOUS INTEGRATION

---

We hebben in de inleiding de inhoud van **CI** al gedetailleerd de revue laten passeren, dus houden we het hier bij een samenvatting.

In **Continuous Integration** worden afzonderlijke wijzigingen van een developer samengevoegd en getest. Het doel van **CI** is om snel individuele codewijzigingen te valideren vooraleer deze deel uitmaken/opgenomen worden in de basis code. Dit gebeurt door middel van een **build** en het uitvoeren van aanwezige **Unit Tests**.

### 3.3 CONTINUOUS DELIVERY

---

In **Continuous Delivery** kunnen de geïsoleerde wijzigingen die zijn samengevoegd en gevalideerd tijdens Continuous Integration worden gecombineerd met de resterende productcode. Vervolgens wordt dat resultaat door een volgende reeks van testen gehaald. Het doel van **Continuous Delivery** is niet noodzakelijk om het eindresultaat te implementeren, maar wel om te bewijzen dat het eindresultaat inzetbaar is. Het proces van **Continuous Delivery** is de *Continuous Delivery Pipeline (CDP)*.

In deze bijkomende stap komen ondermeer volgende termen naar boven;

- Artifacts
- Versioning
- Coding Metrics & Analysis
- Testing (*dit maal, Integration, Functional en Acceptance tests*)

### 3.4 CONTINUOUS DEPLOYMENT

---

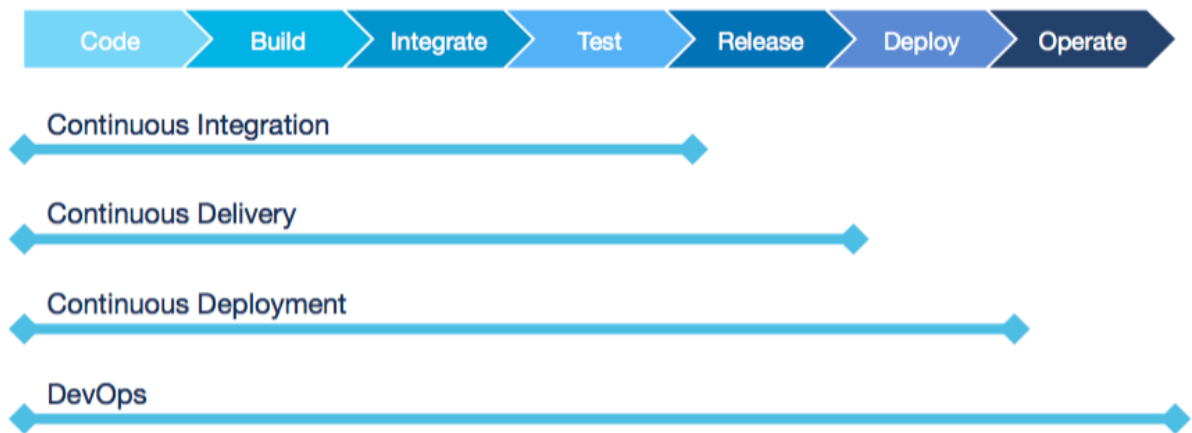
**Continuous Deployment** is een proces dat *kan* worden toegevoegd aan het einde van de **Continuous Delivery** pipeline. Het idee is dat, aangezien de **Continuous Delivery** pipeline heeft bewezen dat de laatste wijzigingen inzetbaar zijn, we het implementatieproces ook moeten automatiseren. Op deze manier zouden we met het resultaat van de **Continuous Delivery** pipeline door gaan en het resultaat ook meteen gaan implementeren.

Implementatie kan veel dingen betekenen, als we dit simpel houden zouden we gerust kunnen stellen dat dit eenvoudig weg het beschikbaar stellen van de release voor de klant is. Dat kan betekenen dat de nieuwe versie vervolgens in de cloud wordt uitgevoerd, downloadbaar wordt gemaakt of op een andere gekende locatie wordt bijgewerkt.

Hier kan dan verder nog een onderscheid gemaakt worden naar **Staged** Deployment of **Blue/Green** Deployment. We laten deze termen hier al eens vallen, maar deze komen verder niet aan bod in deze cursus

### 3.5 SCHEMATISCHE VOORSTELLING VAN DE VERSCHILLEN

---



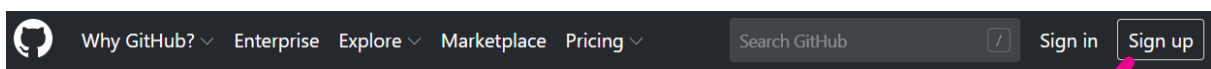
## 4 GITHUB SETUP

Vrijwel elke module in de opleiding zal op een gegeven moment gebruik maken van Github om oefeningen, opdrachten en of examens te faciliteren. Het is daarom dan ook zeer belangrijk dat je een werkende Github account hebt die bovendien ook goed geconfigureerd is volgens de vereisten van de opleiding.

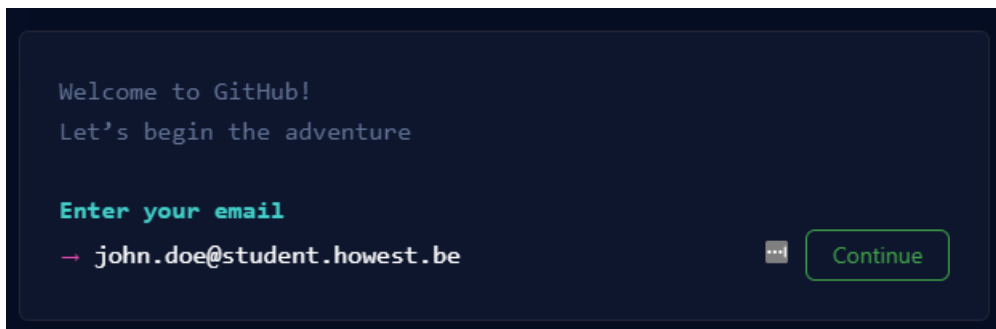
Heb je nog geen Github account? Volg dan even de stappen in **Account aanmaken**. Indien je er wel al eentje hebt, controleer dan met behulp van **Account configuratie** of je account correct geconfigureerd is volgens de vereisten van de opleiding.

### 4.1 ACCOUNT AANMAKEN

- Surf naar <https://github.com/> en klik daar vervolgens de optie **Sign up** aan.



- Gebruik je [voornaam.naam@student.howest.be](mailto:voornaam.naam@student.howest.be) emailadres om je te registreren.



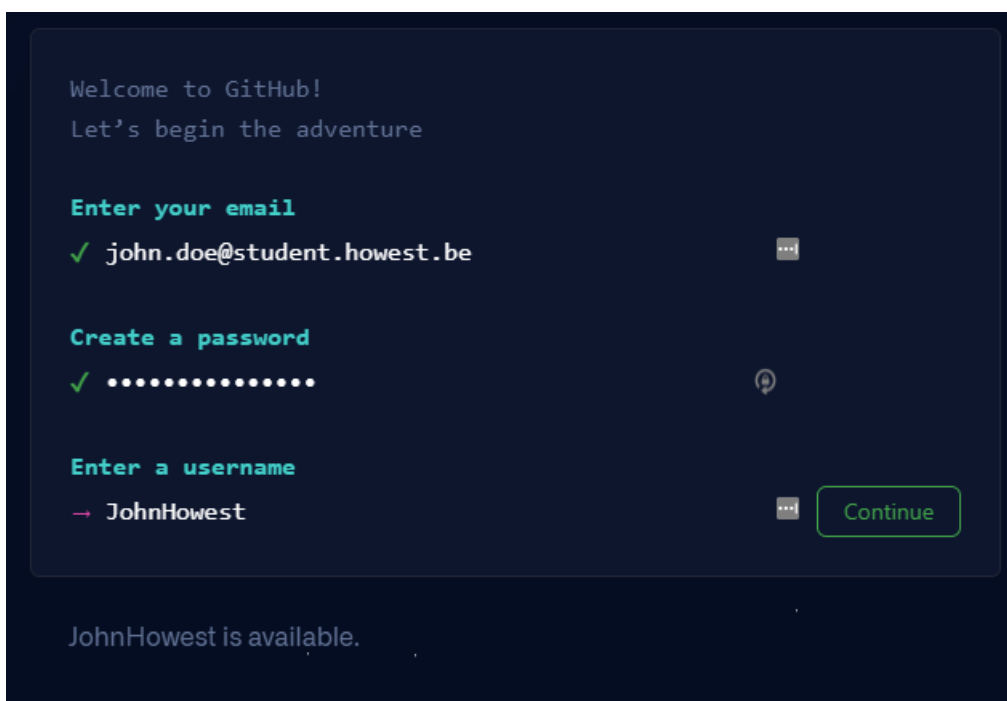
Welcome to GitHub!  
Let's begin the adventure

Enter your email

→ john.doe@student.howest.be

Continue

- Kies een (sterk) wachtwoord en een gebruikersnaam (*vrije keuze*).



Welcome to GitHub!  
Let's begin the adventure

Enter your email

✓ john.doe@student.howest.be

Create a password

✓ .....

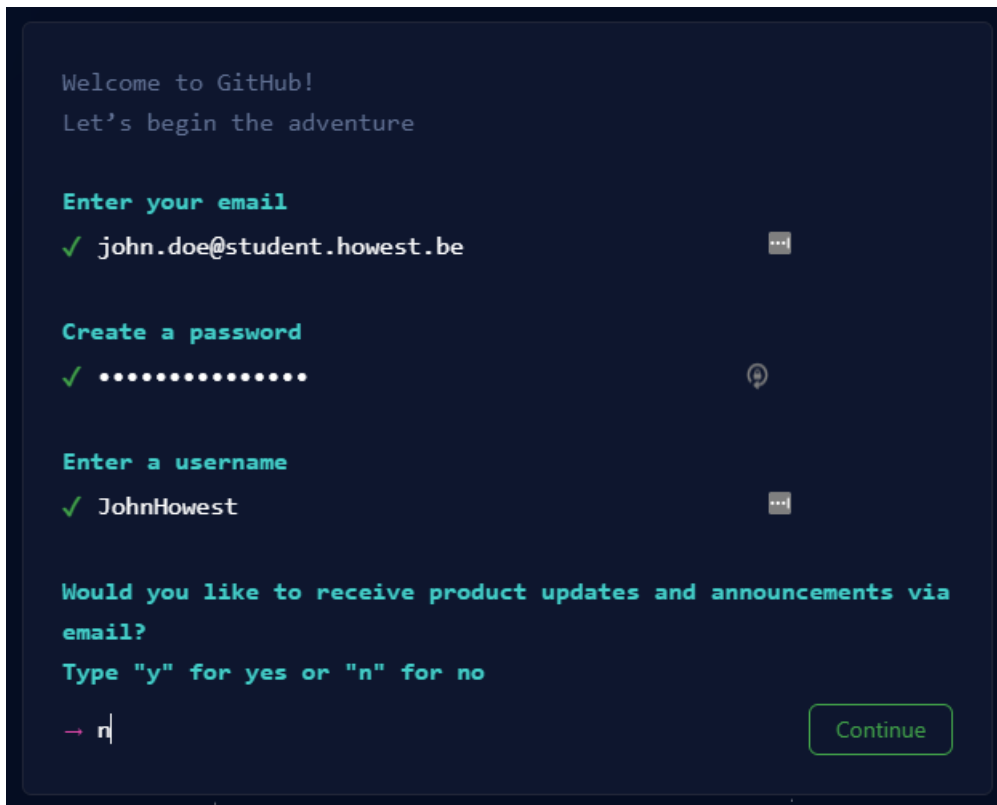
Enter a username

→ JohnHowest

Continue

JohnHowest is available.

- Je kiest zelf of je al dan niet e-mails van GitHub wil ontvangen met algemene updates en aankondigingen.



Welcome to GitHub!  
Let's begin the adventure

Enter your email  
✓ john.doe@student.howest.be

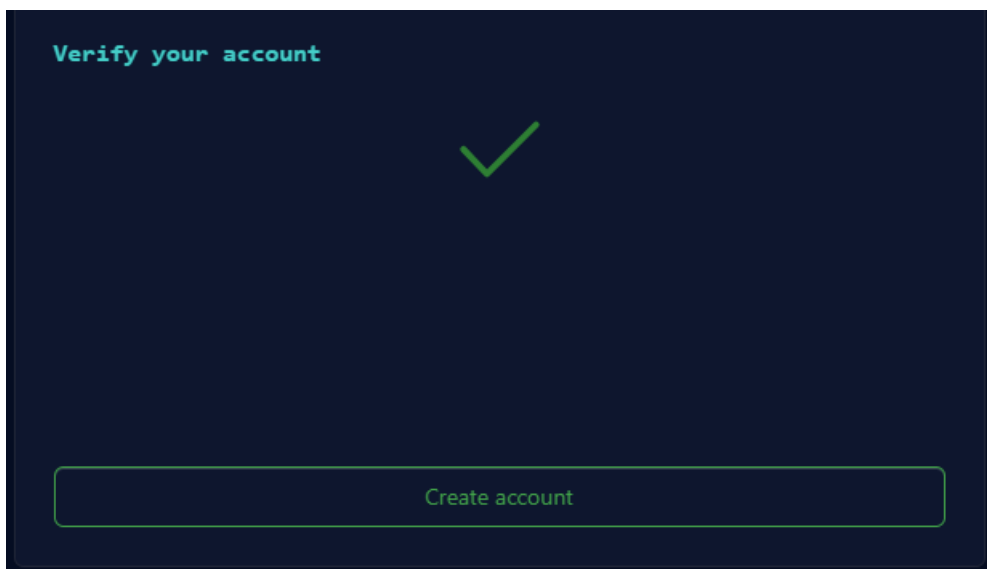
Create a password  
✓ .....

Enter a username  
✓ JohnHowest

Would you like to receive product updates and announcements via email?  
Type "y" for yes or "n" for no  
→ n

Continue

- Je krijgt nu eventueel nog een *CAPTCHA*-vraag te zien om te bewijzen dat je geen computer bent. Na deze verificatie, druk je op "Create account".

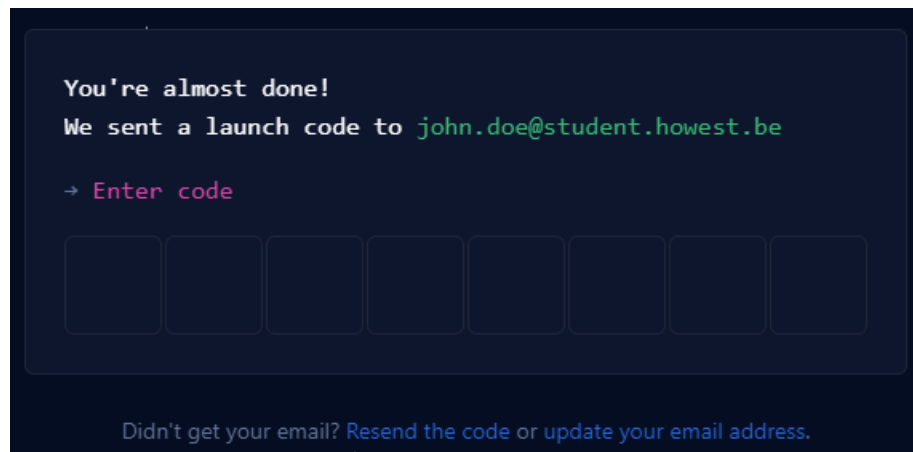


Verify your account

✓

Create account

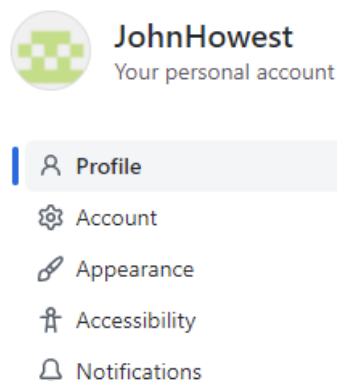
- Je ontvangt nu een email op je ingevulde Howest emailadres, met een verificatiecode. Vul deze code in om je emailadres te verifiëren. Vervolgens wordt je ingelogd op GitHub met je zonet aangemaakte account.



## 4.2 ACCOUNT CONFIGURATIE (VERPLICHT)

Eenmaal je ingelogd bent, klik je bovenaan rechts op het icoontje dat je avatar voorstelt. Maak vervolgens de keuze **Settings** in het menu dat tevoorschijn komt.

Je komt normaalgezien nu automatisch op **Profile** in je **Settings** menu terecht. Zo niet, klik dan zelf op **Profile**.

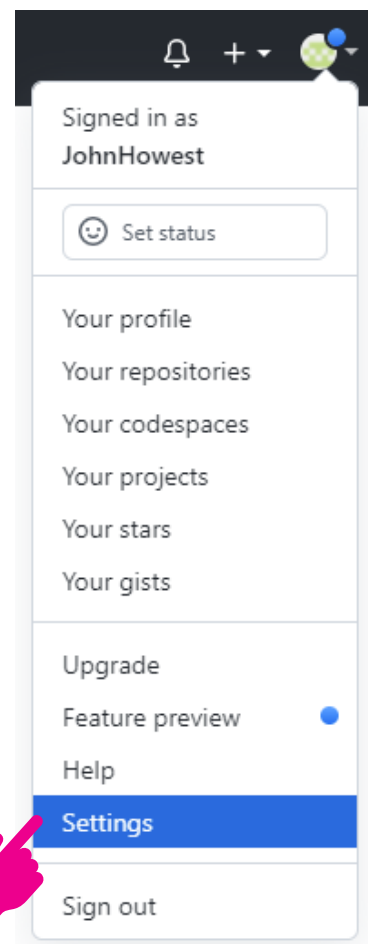


**BELANGRIJK!!** Zorg ervoor dat in je Public Profile je **Name** aangevuld wordt met je **Naam** en **Voornaam** zoals deze voorkomt in je inschrijving bij Howest.

### Public profile

Name

Your name may appear around GitHub where you contribute or are mentioned. You can remove it at any time.



Controleer vervolgens nog even je **emailadres** (op dezelfde Profile settings pagina). Je public email dient je **@student.howest.be** emailadres te zijn. Is er niks ingevuld en kan je ook niks selecteren in de combobox? Dan is de verificatie van je emailadres nog niet voltooid! Dit is het geval op onderstaande screenshot. Ga in

dat geval nog eens op zoek naar de verificatiemail die je van GitHub ontvangen zou moeten hebben, en werk de verificatiestap eerst af.

#### Public email

Please add a verified email address ▾



You have set your email address to private. To toggle email privacy, go to [email settings](#) and uncheck "Keep my email address private."

Andere velden zijn niet verplicht om in te vullen, maar vul gerust je profiel verder aan waar je dit zelf wenst te doen. Je kan bv. ook een profielfoto (avatar) instellen.

### 4.3 GITHUB STUDENT PACK (VRIJBLIJVEND)

---

Github ondersteunt al jaren educatieve projecten en het gebruik van Github in het onderwijs. Hierbovenop voorzien ze heel wat **gratis** voordelen en/of accounts via verschillende partners.

Check zeker en vast even de voordelen via volgende link <https://education.github.com/pack#offers>  
En haal je **Student Developer Pack** via de knop **Get the Pack** binnen!



## 5 WAT IS GIT?

Git is een **gedistribueerd** versiebeheerssysteem. Git is oorspronkelijk gemaakt door Linus Torvalds voor de ontwikkeling van de Linux kernel.

Git verschilt enigszins van centrale versiebeheerssystemen zoals Subversion (SVN) en Team Foundation Service (TFS).

Bij **centrale** versiebeheerssystemen worden revisies standaard geüpload naar een server wanneer een nieuwe versie ingediend wordt door een deelnemer, waar de oorspronkelijke versie wordt overschreven.

Bij Git daarentegen worden nieuwe versie ingediend in een lokale kopie van de opslagplaats wanneer een deelnemer aanpassingen heeft gemaakt. Op **elk gewenst moment** kan deze lokale opslagplaats **gesynchroniseerd** worden met de server. Voordeel hiervan is dat nieuwe revisies kunnen ingediend worden, zelfs bij ontbreken van internetverbinding, terwijl voor centrale versiebeheerssystemen altijd een verbinding nodig is met de server.

### Principes versiebeheerssysteem

- Gedistribueerde versie controle
- Coördineert werk tussen verschillende ontwikkelaars
- Volgt veranderingen op en wie ze heeft uitgevoerd
- Terug schakeling naar eerdere versies mogelijk
- Zowel lokale als globale repositories mogelijk

### Concepten

- Opvolging van geschiedenis code
- Neemt 'snapshots' van de code
- Je beslist zelf wanneer je een 'snapshot' maakt door een commit
- Files moeten gestaged worden voor een commit.





## 6 GIT INSTALLATIE

### 6.1 DOWNLOADEN VAN GIT

Git installeer je op een computer en veelal wordt gebruik gemaakt van een promptvenster (Terminal) om de nodige commando's in te geven, waardoor je aan een map git-functionaliteiten toekent. Hiernaast werden ook een aantal grafische interfaces ontwikkeld of Git geïntegreerd in een toepassing (Visual Studio, Visual Studio Code, ...).

Om van Git gebruik te kunnen maken, download je eerst en vooral de versie op <https://git-scm.com/> voor jouw besturingssysteem.

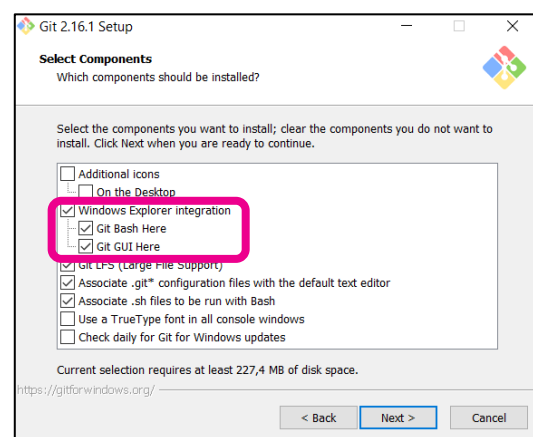


### 6.2 AANBEVOLEN INSTALLATIE INSTELLINGEN

Bij het installeren van Git doorloop je enkele instellingen. Hier kiezen we standaard de default waarden, in onderstaande stappen staan we even stil bij enkele vensters en hun aanbevolen instellingen.

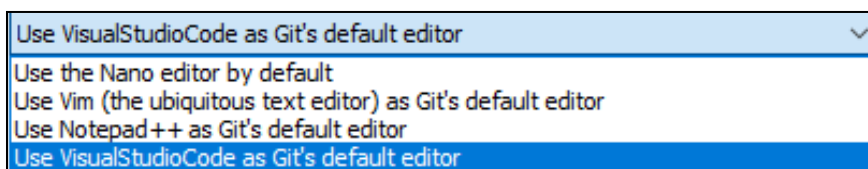
#### 6.2.1 WINDOWS EXPLORER INTEGRATION

- Zorgt dat git vanuit de Windows Verkenner kan opgestart worden.
- Zorgt dat ook de grafische tool Git GUI kan gestart worden vanuit Windows verkenner



#### 6.2.2 DEFAULT EDITOR

Standaard wordt VIM geïnstalleerd als basis editor, maar er wordt door de git installer reeds aangeraden deze niet te gebruiken. Wij kiezen **Visual Studio Code** als standaard editor.



Heb je Visual Studio Code niet geïnstalleerd staan, annuleer dan even de installatie om eerst Visual Studio Code te installeren. Deze kan je terugvinden via volgende link -> <https://code.visualstudio.com/>

#### 6.2.3 INITIAL BRANCH NAME

Git gebruikt voor de initiële branch standaard de naam "master". Recent is GitHub echter overgestapt op de naam "main" voor deze default branch. Ook git zal in de nabije toekomst de overstap maken, zoals te lezen staat in de installer. Eventueel kan je er nu al voor kiezen om als default branch name "main" te

gebruiken voor nieuwe lokaal aangemaakte repo's, door hier de optie **Override the default branch name for new repositories** aan te vinken en "main" in te vullen. Dit zorgt ervoor dat je zowel lokaal met Git en op GitHub de naam "main" zal gebruiken als default branch in een nieuwe repository.

## 6.2.4 PATH ENVIRONMENT

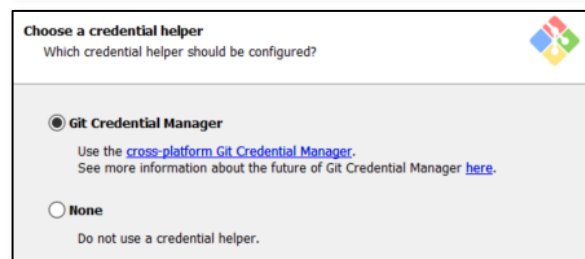
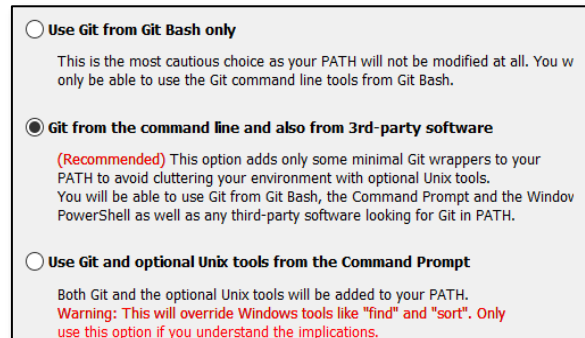
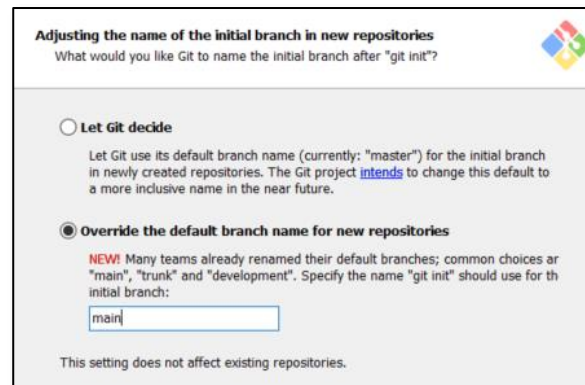
Hier is het aangeraden om voor de optie **Git from the command line and also from 3rd-party software** te kiezen. Dit zal je in staat stellen om Git en Git Bash commando's vanuit de Windows Command Prompt te gebruiken.

## 6.2.5 GIT CREDENTIAL MANAGER

Om te voorkomen dat je telkens je logingegevens moet herhalen als je wil synchroniseren met GitHub, is het sterk aangeraden om de Git Credential Manager te gebruiken. Deze zal ervoor zorgen dat je logingegevens onthouden worden op je computer. Laat hiervoor de standaard reeds aangevinkte optie **Git Credential Manager** aan staan in het venster "Choose a credential helper".

## 6.2.6 OVERIGE INSTALLATIEVENSTERS

Voor de overige installatie vensters mag je de **default waarden** die de installer voorstelt volgen. Je bent vrij om andere keuzes te maken tijdens de installatie.

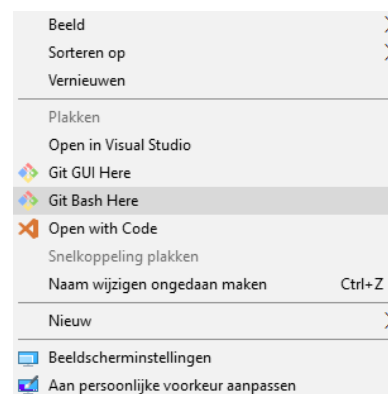
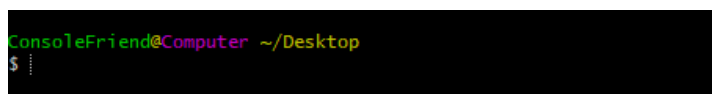


## 6.3 VERDERE CONFIGURATIE

Eenmaal de installatie voltooid is kan je even aan de slag en controleren of de basis configuratie van git reeds naar je wensen is. Om te controleren welke gebruikersnaam en e-mail gelinkt zijn naar je git installatie kan je onderstaande stappen volgen.

Via een rechtermuisklik op je bureaublad kan je de git console tevoorschijn laten komen door de keuze te maken voor **Git Bash Here**.

Dit geeft je een Console venster waar je mee aan de slag kan. Het spreekt voor zich dat je uiteraard ook via de Start functie naar **Git Bash** kan gaan. Dit is niet beperkt tot de rechtermuisklik.



Hier kunnen we vervolgens onze huidige instellingen gaan bekijken.

- Geef het commando **git config --global user.name** in om de huidig ingesteld gebruikersnaam voor het toestel waarop je werkt te controleren.

```
ConsoleFriend@Desktop ~/Desktop
$ git config --global user.name
ConsoleFriend
```

- Indien je nog geen, of geen correcte gebruikersnaam ziet staan kan je met hetzelfde commando je eigen naam gaan toevoegen/ updaten. Vergeet de **--global** parameter niet, zodat dit meteen voor alle toekomstige repositories in te stellen.

```
ConsoleFriend@Desktop ~/Desktop
$ git config --global user.name "ConsoleFriend"
```

- Uiteraard kan je dit door het voorgaande commando nog even controleren als je dit wenst.
- Via het commando **git config --global user.email** kan je op identieke manier de huidig ingestelde e-mail account gaan controleren.

```
ConsoleFriend@Desktop ~/Desktop
$ git config --global user.email
voornaam.naam@student.howest.be
```

- Identiek aan voorgaande stap kan je ook via het toevoegen van een string parameter de nodige instellingen gaan toevoegen/updaten indien nodig. Vergeet ook hier niet om gebruik te maken van de **--global** parameter om dit als standaard globale instellingen te maken.

```
ConsoleFriend@Desktop ~/Desktop
$ git config --global user.email "voornaam.naam@student.howest.be"
```

- Indien gewenst kan je door het initiële commando nog even controleren indien je gegevens correct gesaved werden.
- Vervolgens willen we ook nog aangeven dat git hoofdletter**ongevoelig** moet werken, omdat dit ook het geval is voor het standaard bestandssysteem van Windows. Voor Windows is het bestand test.txt bv. hetzelfde bestand als Test.txt, terwijl dat standaard voor git twee aparte bestanden zijn (ter info: git volgt hierin de Unix-aanpak die je ook zal terugvinden op Linux of Mac computers). Op Windows kan deze inconsistentie voor problemen zorgen, dus we stellen git als volgt in:

```
$ git config --global core.ignoreCase true
```

- Je basisinstelling voor git zijn bij deze gemaakt en je bent klaar om van start te gaan met git!



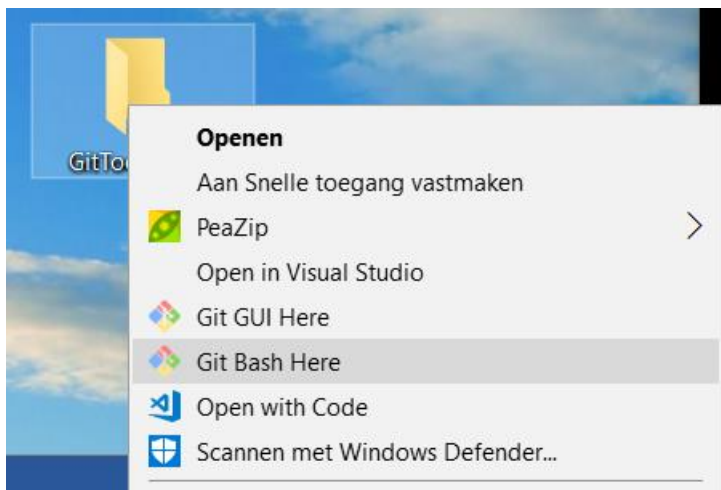
## 7 VAN START MET GIT

### 7.1 EEN EERSTE KENNISMAKING

Eenmaal Git geïnstalleerd is, kan je hiervan gebruik maken vanuit de mappenstructuur binnen Windows. Je maakt een map aan en kunt hier dan de git functionaliteiten op toepassen, gaande van initialiseren tot bepaling welke files je wenst op te volgen tot het maken van een online repository, die als back-up kan dienen of de startfolder waarvan je dan later kunt gaan samenwerken voor eenzelfde project. Dit deel leert je de basis voor het aanmaken van dergelijke file. Programma's zoals Visual Studio en Visual Studio Code hebben dergelijke functionaliteit standaard ingebakken, waardoor ze achter de schermen eigenlijk dezelfde code gaan gebruiken.

#### 7.1.1 EEN GIT REPOSITORY AANMAKEN

- Maak een Mapje met de naam **GitToepassing** aan (dit mag gerust op je bureaublad)
- Rechtsklik op dit mapje en kies vervolgens **Git Bash Here**



- Je opent hierdoor een **Git Bash console** en bevind je meteen in de werkmap **GitToepassing** op de Desktop.

```
ConsoleFriend@Computer ~/Desktop/GitToepassing
$ |
```

- Wanneer we even het commando **git status** ingeven kunnen we meteen zien dat we in deze directory nog **niet met git** aan de slag kunnen.

```
ConsoleFriend@Computer ~/Desktop/GitToepassing
$ git status
fatal: Not a git repository (or any of the parent directories): .git

ConsoleFriend@Computer ~/Desktop/GitToepassing
$ |
```

- We moeten eerst een **git repository** aanmaken. Dit doen we door het commando **git init** te gebruiken.

```
ConsoleFriend@Computer ~/Desktop/GitToepassing
$ git init
Initialized empty Git repository in C:/~/Desktop/GitToepassing/.git/

ConsoleFriend@Computer ~/Desktop/GitToepassing (master)
$ |
```

- Laten we even de inhoud bekijken van onze folder.  
We doen dit door middel van het commando **ls -a**  
We zien dat we een nieuwe (verborgen) folder erbij gekregen hebben genaamd **.git**

```
ConsoleFriend@Computer ~/Desktop/GitToepassing (master)
$ ls -a
./ ../ .git/

ConsoleFriend@Computer ~/Desktop/GitToepassing (master)
$ |
```

**! Let op** het commando wat we net gebruikten is **geen git commando** maar een **bash** commando!  
Dit kan je zelf al afleiden omdat dit commando **niet met het woordje git** begint!

## 7.1.2 BESTANDEN TOEVOEGEN AAN JE REPOSITORY (STAGEN)

- We voegen een aantal zaken toe aan onze **GitToepassing** folder. We voorzien een tekstbestand genaamd **test01.txt** en een folder **testFolder**. In de folder **testFolder** maken we een tweede tekstbestand aan genaamd **test02.txt**.  
Het aanmaken van deze bestanden kan je doen door middel van onderstaand commando **touch test01.txt**  
Het is uiteraard ook prima als je deze bestanden aanmaakt door middel van de rechtermuisknop en de keuze te maken voor **Nieuw -> tekstbestand**
- We controleren even of we nu al iets in onze git zitten hebben door het commando **git status** in te geven.

```
ConsoleFriend@Computer ~/Desktop/GitToepassing (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    test01.txt
    testFolder/

nothing added to commit but untracked files present (use "git add" to track)

ConsoleFriend@Computer ~/Desktop/GitToepassing (master)
$ |
```

- We zien onze bestanden staan, echter wordt er op dit ogenblik nog niks ‘bijgehouden’ door git. Hiervoor dienen we eerst onze bestanden te gaan ‘stagen’. Dit doen we door volgende commando’s in te geven:

```
git add test01.txt
git add testFolder/
```

```
ConsoleFriend@Computer ~/Desktop/GitToepassing (master)
$ git add test01.txt

ConsoleFriend@Computer ~/Desktop/GitToepassing (master)
$ git add testFolder/

ConsoleFriend@Computer ~/Desktop/GitToepassing (master)
$ |
```

- We controleren even met het commando **git status** hoe het er nu uit ziet. We zien onze bestanden nu staan onder ‘Changes to be committed’ en dat is prima.

```
ConsoleFriend@Computer ~/Desktop/GitToepassing (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   test01.txt
        new file:   testFolder/test02.txt

ConsoleFriend@Computer ~/Desktop/GitToepassing (master)
$ |
```

### 7.1.3 WIJZIGINGEN COMMITTEN

- Nu onze bestanden gestaged zijn kunnen we ze gaan committen. Hiervoor maken we gebruik van het commando **git commit -m "Mijn eerste commit"**

```
ConsoleFriend@Computer ~/Desktop/GitToepassing (master)
$ git commit -m "Mijn eerste commit"
[master (root-commit) 18b00aa] Mijn eerste commit
2 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 test01.txt
create mode 100644 testFolder/test02.txt

ConsoleFriend@Computer ~/Desktop/GitToepassing (master)
$ |
```

- Als we nu terug gaan kijken naar de huidige status via **git status**, zien we dat alles gecommited is en er geen openstaande wijzigingen meer zijn.

```
ConsoleFriend@Computer ~/Desktop/GitToepassing (master)
$ git status
On branch master
nothing to commit, working tree clean

ConsoleFriend@Computer ~/Desktop/GitToepassing (master)
$ |
```

- Willen we even gaan kijken naar wat er nu precies gecommit werd dan kunnen we dit uiteraard ook via het command `git log`.

```
ConsoleFriend@Computer ~/Desktop/GitToepassing (master)
$ git log
commit 18b00aa33ca920fd1fbbf40145e886e7ce45ca8e (HEAD -> master)
Author: ConsoleFriend <Joachim.Francois@Howest.be>
Date: Tue May 8 21:25:35 2018 +0200

    Mijn eerste commit

ConsoleFriend@Computer ~/Desktop/GitToepassing (master)
$ |
```

**Tip!** Eenmaal je meerdere commits hebt kan je deze in verkort formaat weergeven doormiddel van de parameter `--oneline`. Het volledige commando wordt vervolgens `git log --oneline`.

#### 7.1.4 EEN BESTAND GESTAGED DAT JE NIET WOU STAGEN?

Het kan al eens gebeuren dat je een bestand gestaged hebt via het `git add` command maar dat je dit niet meteen mee wou nemen naar de staging. Hoe krijg je deze er dan terug uit?

Het commando `git reset` kan je hierbij helpen. Let echter wel op: het gebruik van git reset zorgt ervoor dat alle gestagede bestanden opnieuw 'unstaged' zullen worden.

Heb je nood om enkel 1 van meerdere bestanden te unstagen?

Dan kan je aan de slag met `git reset -- naamVanBestand`

Bijvoorbeeld: `git reset -- verkeerd.txt`

Wees wel steeds aandachtig als je met het git reset commando aan de slag gaat. Dit commando wordt namelijk ook gebruikt (*in andere vorm*) om commits ongedaan te maken. Hier komen we later nog op terug.

## 7.2 HOE WERKT GIT NU?

---

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. De 40 gebruikte karakters zijn (veel!) meer dan voldoende om ervoor te zorgen dat elk onderdeel van je repository een unieke hashcode krijgt. Sterker nog: doorgaans volstaan hoogstens 8 tot 10 karakters. Git laat dan ook toe om slechts een prefix te gebruiken van eender welke hashcode, zolang die nog uniek is binnen de repository. Je zal dus vaak kortere codes tegenkomen zoals bv.

**24b9da65**

Zelfs in gigantische repositories met honderdduizenden lijnen code volstaan hooguit de 12-tal eerste karakters om elke hashcode uniek te beschrijven. We kunnen er dus wel veilig van uitgaan dat we nooit in de problemen zullen komen met de totale 40 karakters, die een astronomisch groot aantal mogelijkheden bieden. In de praktijk gebruik je slechts zoveel karakters als nodig, dit kan mee groeien naarmate de repository uibreidt. Git zal je vriendelijk laten weten wanneer je onvoldoende karakters gebruikt, d.w.z. wanneer er meerdere hashcodes in de repository voorkomen met dezelfde opgegeven prefix.

Bijna alles wat je in Git doet, leidt tot toevoeging van data in de Git database. Deze database bevindt zich in de folder `.git`. 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 repository regelmatig uploadt ('push') naar een online repository. Dit maakt het gebruik van Git net zo tof, omdat je weet dat je kunt experimenteren zonder het gevaar te lopen jezelf behoorlijk in de nesten te werken.



## 7.3 DE DRIE TOESTANDEN IN GIT

Dit is het belangrijkste dat je over Git moet weten als je wilt dat de rest van het leerproces van Git vlotjes verloopt. Git heeft drie hoofdtoestanden waarin bestanden zich kunnen bevinden: gecommit ('committed'), 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 ('**working directory**'), en de wachtrij voor een commit ('**staging area**').

### 7.3.1 WORKING DIRECTORY

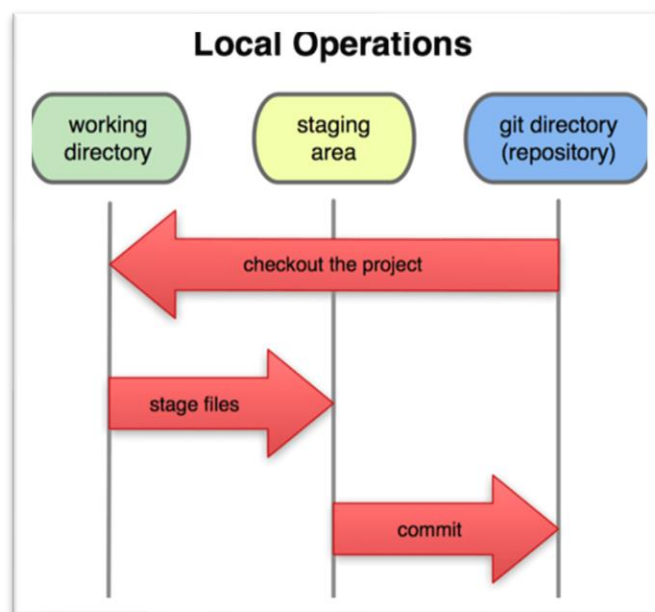
De werk directory is een checkout van een bepaalde versie van het project. Deze bestanden worden uit de gecomprimeerde database in de Git directory gehaald en op de harde schijf geplaatst waar jij ze kunt gebruiken of bewerken.

### 7.3.2 STAGING AREA

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.

### 7.3.3 GIT DIRECTORY (REPOSITORY)

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 kloonst vanaf een andere computer of online repository.



## 7.4 ALGEMENE WORKFLOW MET GIT

---

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 gecommited. 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.

## 8 EEN AANTAL COMMANDO'S

Hieronder kan je een kleine greep uit het arsenaal van git commando's terugvinden. Dit zijn hoofdzakelijk de commando's die in dit eerste deel aan bod kwamen.

Commando	Beschrijving
<code>git --help</code>	Geeft een overzicht van alle git-commando's
<code>git config --global user.name</code>	Ophalen van de gebruikersnaam gelinkt aan de git waar je in aan het werk bent. De <b>global</b> parameter haalt de gebruikersnaam op gelinkt aan de algemene configuratie van git op het toestel waarop je aan de slag bent.
<code>git config --global user.name "Naam"</code>	Het zetten van een gebruikersnaam aan de git waar je in aan het werk bent. De waarde die je opgeeft tussen de quotes is de nieuwe naam waar je wil naar updaten. De <b>global</b> parameter zal deze instelling maken voor algemene configuratie van git op het toestel waarop je aan de slag bent.
<code>git config --global user.email</code>	Ophalen van het e-mailadres gelinkt aan de git waar je in aan het werk bent. De <b>global</b> parameter haalt de gebruikersnaam op gelinkt aan de algemene configuratie van git op het toestel waarop je aan de slag bent.
<code>git config --global user.email "Mail@vb.com"</code>	Setten van het e-mailadres gelinkt aan de git waar je in aan het werk bent. De waarde die je opgeeft tussen de quotes is het nieuwe e-mailadres waar je wil naar updaten. De <b>global</b> parameter zal deze instelling maken voor algemene configuratie van git op het toestel waarop je aan de slag bent.
<code>git init</code>	Maakt van de map een lege git repository
<code>git init NaamVanFolder</code>	Maakt een nieuwe map aan <b>met</b> een lege git repository (vergeet nadien niet om zelf nog naar de juiste map te gaan!)
<code>git add &lt;file&gt;</code>	Het opgegeven bestand wordt met zijn huidige status 'gestaged'. Het bestand bevindt zich nu op de Staging Area
<code>git add .</code>	Idem als het voorgaande commando, echter zal dit alle nog niet gestagede bestanden gaan stagen in de folder waar je je bevindt! (alsook onderliggende folders)
<code>git add *</code>  vb. <code>git add *.txt</code> (enkel bestanden stagen met extensie txt)	De asterisk laat je toe om gebruik te maken van een 'wildcard'. Hierdoor kan je bestanden stagen die overeenkomen met een bepaald patroon.
<code>git status</code>	Controle van de filestatus (inclusief untracked files)
<code>git commit -m "naam van de commit"</code>	Het commiten (registreren) van je gestagede files naar je local repository. De parameter -m laat je toe om tussen quotes een referentie in te vullen voor deze commit.

<b>git log</b>	Geeft je een overzicht van alle gemaakte commits
<b>git log --oneline</b>	Geeft je een overzicht van alle gemaakte commits in beknopt formaat.

We geven hier alvast ook nog een aantal bash commando's mee die al dan niet aan bod kwamen tijdens dit hoofdstuk. Het spreekt voor zich dat wat je met deze commando's doet, je gaandeweg in de UI of rechtstreeks in je folder zal gaan doen. Opteer steeds voor wat het makkelijkste werkt.

Commando	Beschrijving
<b>ls</b>	Toon bestanden in de huidige directory
<b>mkdir NaamVanFolder</b>	Maakt een nieuwe folder aan op de locatie waar je het commando uitvoert
<b>touch NaamBestand.extensie</b>	Maakt een nieuw bestand aan met extensie op de locatie waar je het commando uitvoert

