Labo 1 GitHub

In dit eerste labo leer je Git gebruiken via de opdrachtlijn enerzijds en vanuit NetBeans IDE anderzijds. Aangezien NetBeans en andere omgevingen achter de schermen dezelfde opdrachten uitvoeren, kun je naar believen afwisselen.

1 Via de opdrachtlijn

We beginnen met het invoeren van Git-opdrachten via de opdrachtlijn. Of je nu Windows, OS X of Linux op je computer hebt staan, de opdrachten zijn identiek. In deze labo's gebruiken we Windows, al is dat strikt genomen geen vereiste.

1.1 Git Bash

Hoewel je onder Windows Git-opdrachten kunt intikken in Command Prompt of Opdrachtprompt (ook weleens foutief een 'DOS-venster' genoemd), is het aan te bevelen om in plaats daarvan gebruik te maken van **Git Bash**. Hierin is Bash de shell, de interactieve omgeving waarin je opdrachten uitvoert, zoals die onder andere wordt aangeboden op Linux-machines. Git Bash vult Bash aan met enkele handigheidjes.

Op de labo-pc's is Git Bash reeds geïnstalleerd en kun je een shellvenster openen door te dubbelklikken op de snelkoppeling. Wanneer je thuis via de officiële website Git installeert, zul je dezelfde snelkoppeling zien verschijnen.

Bij het opstarten van (Git) Bash zie je de zogenaamde prompt: een stukje tekst dat je informatie geeft over de toestand van de shell. Standaard vind je daarin je gebruikersnaam, gevolgd door de naam van de computer en de map waarin je werkt. Je persoonlijke directory wordt afgekort tot een tilde (~). De knipperende cursor na het dollarteken vertelt je dat Git Bash wacht op jouw eerste opdracht.

Een belangrijk gevolg van het gebruik van Git Bash, en meteen ook het enige échte verschil met Command Prompt, is dat padnamen worden vertaald naar een Linux-achtige gedaante. Backslashes worden hierbij vervangen door schuine strepen, en stationsaanduidingen zoals `C:` worden vervangen door `/c` en volgende. Om hier een voorbeeld van te zien, voer de opdracht `pwd` in en druk op Enter. Onder jouw invoer verschijnt dan als uitvoer het volledige pad van de directory waarin je je momenteel bevindt.

Met de opdracht `cd` gevolgd door een padnaam kun je deze werkdirectory wijzigen. Bevat de padnaam spaties, dan plaats je hem tussen aanhalingstekens. Als je USB-stick bijvoorbeeld de stationsaanduiding `E:` kreeg toegewezen en je naar de directory `E:\School\Labo Softwareontwikkeling` wilt navigeren, dan voer je deze opdracht in:

```
cd "/e/School/Labo Softwareontwikkeling"
```

Je hoeft lange padnamen niet helemaal zelf te typen. Wanneer je de eerste karakters ingevoerd hebt, kun je op Tab drukken om Bash deze te laten aanvullen. Zijn er meerdere mogelijkheden, druk dan nogmaals op Tab om alle mogelijkheden te zien, of voeg extra karakters toe.

Wil je een eerder ingevoerde opdracht herhalen, dan kun je met behulp van Pijl omhoog en Pijl omlaag de geschiedenis doorlopen. De horizontale pijltjestoetsen laten je dan weer toe om de cursor te verplaatsen doorheen de huidige opdracht. Met Home en End kun je snel naar het begin en einde van de opdracht springen.

Het is niet mogelijk om de muis te gebruiken. Wel kun je net als bij een klassiek Command Prompt-venster op het venstericoon klikken en de bewerkmodus activeren. Hiermee kun je onder andere tekst selecteren, die je dan kunt plakken door te rechtsklikken. Probeer dit echter te vermijden; indien je meerdere regels tekst plakt, kan Bash zich immers vreemd gaan gedragen.

Met deze korte inleiding weet je ruim voldoende af van Git Bash. Hoog tijd dus om aan de slag te gaan met Git zelf.

1.2 Git

In de inleidende presentatie werden de meest courante Git-commando's al even toegelicht. Zo'n commando begint steeds met het woordje `git`, gevolgd door een spatie en de eigenlijke opdracht. Laten we beginnen met Git te vertellen wie we zijn. Deze informatie zal dan worden overgenomen wanneer we een commit aanmaken. Voer de volgende opdracht in:

```
git config --global user.name "Jouw volledige naam"
```

Vul natuurlijk wel eerst je volledige naam in (tussen aanhalingstekens) en druk dan op Enter. Als de opdracht geslaagd is, verschijnt er geen uitvoer; dit is een conventie bij shells.

Wat hebben we zonet uitgespookt? We analyseren de opdracht even:

- We weten intussen dat Git-opdrachten met het woordje `git` beginnen.
- Daarna volgt `config`, wat erop wijst dat we met de configuratie van Git willen werken.
- Misschien wel het vreemdste deel van de opdracht is het fragment `--global`. Het liggend streepje vooraan betekent dat dit een optie is. Bij vele shellopdrachten kunnen dergelijke opties worden toegevoegd om het gedrag ervan te beïnvloeden. Opties die met één streepje beginnen bestaan uit een enkele letter, zoals bijvoorbeeld `-x`, en zijn daardoor minder duidelijk dan hun tegenhangers met twee streepjes. In dit geval duidt de optie `--global` erop dat we willen werken met de Git-configuratie voor het hele systeem en niet enkel voor de huidige repository; we hebben immers nog geen repository aangemaakt.
- De naam van de configuratieparameter waarmee we willen werken is `user.name`.
- Omdat we ook de waarde van de parameter opgeven, in dit geval onze volledige naam, weet Git dat we deze wensen te wijzigen. Indien we de waarde weglaten, vertelt Git ons de reeds ingestelde waarde. Probeer dit gerust ook even uit.

Op analoge wijze vertellen we Git vervolgens nog ons e-mailadres, en dan kunnen we echt aan de slag:

```
git config --global user.email "voornaam.familienaam@ugent.be"
```

1.3 Je eerste repository

Nu Git helemaal klaar is voor gebruik, kunnen we een nieuwe repository aanmaken. Voorlopig werken we enkel lokaal en maken we dus nog geen gebruik van GitHub of andere Git-servers.

We zullen een eenvoudige applicatie ontwikkelen voor het beheer van een dierenwinkel. Onze repository krijgt daarom de naam `petstore`. In principe mag een repository eender welke naam hebben, maar gemakshalve gebruik je beter geen spaties of vreemde tekens. Het liggend streepje en het laag streepje komen wel courant voor.

Allereerst hebben we een directory nodig om onze repository in op te slaan. We navigeren met `cd` naar de gewenste directory en gebruiken vervolgens de Bash-opdracht `mkdir` om een nieuwe subdirectory aan te maken:

```
cd /c/temp
mkdir petstore
cd petstore
```

Zorg ervoor dat je je in de directory `petstore` bevindt alvorens verder te gaan! Twijfel je, voer dan de opdracht `pwd` nogmaals uit.

Eens in de (lege) directory `petstore` aangekomen, kunnen we deze transformeren in een Git-repository. We moeten hiervoor enkel deze opdracht uitvoeren:

```
git init
```

Wanneer we als uitvoer Initialized empty Git repository krijgen, is de opdracht geslaagd en is onze repository klaar voor gebruik. Achter de schermen heeft Git de verborgen directory `.git` aangemaakt, waar alle Git-gerelateerde informatie zal worden opgeslagen. Opdrachten die met `git` beginnen, lezen of manipuleren de inhoud van deze directory. Het is niet de bedoeling om de inhoud handmatig aan te passen.

1.4 Je eerste commit

We zouden onze Java-kennis nu kunnen botvieren op de dierenwinkel, maar omdat dit een Git-labo is, spelen we even vals. Pak dit archiefbestand uit in de map `petstore` om de Java-code klaar te zetten.

Hoewel we nu bestanden hebben toegevoegd aan de directory, zitten deze nog niet in de repository en dus niet in de geschiedenis van de code. Deze geschiedenis bestaat immers uit opeenvolgende commits, terwijl we nog geen enkele commit hebben aangemaakt.

Wel weet Git al dat er nieuwe bestanden zijn verschenen. De opdracht `git status` geeft onder Untracked files een overzicht van bestanden die kunnen worden toegevoegd aan de repository.

We wensen nu alle `.java`-bestanden in de huidige map te groeperen in een commit. Om bestanden toe te voegen aan een commit, is er de opdracht `git add`, gevolgd door de namen van bestanden. We zouden alle bestandsnamen manueel kunnen opgeven, maar we kunnen ook handig gebruikmaken van de ondersteuning van patterns in Bash:

```
git add *.java
```

Opnieuw krijgen we geen uitvoer te zien (of hooguit enkele waarschuwingen omtrent regeleindes), wat erop wijst dat het toevoegen is gelukt. Wanneer we opnieuw `git status` uitvoeren, merken we dat de bestanden in kwestie inderdaad zijn verhuisd naar de lijst Changes to be committed.

We zouden nu nog meer wijzigingen kunnen aanbrengen, maar in dit geval is onze commit helemaal klaar. We kunnen Git dus vertellen dat we deze wensen toe te voegen aan de geschiedenis. Het enige wat we nog nodig hebben is een commit message die de inhoud beschrijft. Omdat het onze eerste commit betreft, kunnen we kort zijn. De optie `-m` van de opdracht `git commit` laat ons toe om de commit message mee te geven. Omdat deze spaties bevat, plaatsen we hem tussen aanhalingstekens:

```
git commit -m "Initial commit"
```

We krijgen dan wat uitleg te zien over de wijzigingen die de commit met zich meebrengt, wat erop wijst dat Git de geschiedenis heeft bijgewerkt. We kunnen dit uiteraard ook nagaan: de opdracht `git log` geeft als uitvoer één enkele commit, namelijk onze allereerste.

1.5 Intermezzo: pager en editor

Bij een 'echte' repository zal de uitvoer van `git log` natuurlijk behoorlijk wat langer uitvallen. Git geeft deze in dit geval door aan een zogenaamde pager, die toelaat om te scrollen met behulp van de pijltjestoetsen. Om de pager te verlaten, druk je op `q`.

Wegens een bug kan het voorkomen dat de waarschuwing Terminal is not fully functional verschijnt en de pager niet naar behoren werkt. Je kunt deze (permanent) oplossen door volgende twee opdrachten uit te voeren in Git Bash:

```
echo export TERM=msys >>~/.bashrc
export TERM=msys
```

Zoals je intussen weet, is `-m` bij `git commit` een optie en mag ze dus worden weggelaten. Voeren we `git commit` uit zonder `-m`, dan belanden we in een teksteditor. Meestal is dit Vim, een zeer krachtige, maar niet altijd even gebruiksvriendelijke editor. Om in Vim tekst in te voeren, moet je eerst insert mode activeren door op `i` te drukken. Ben je klaar met typen, dan druk je op Esc om insert mode weer te verlaten. Voer vervolgens het commando `:wq` in (dit staat voor write en quit) en druk op Enter om Vim te verlaten.

1.6 Foutje?

Terug naar onze Java-applicatie. Waarschijnlijk heb je blindelings het archiefbestand uitgepakt en meteen de code toegevoegd aan je kersverse repository. Zoals we je echter al op het hart drukten, is het aan te raden om ervoor te zorgen dat élke commit uitsluitend compileerbare, uitvoerbare en wérkende code bevat. Is dit echter wel het geval? Laten we de `.java`-bestanden even compileren:

```
javac *.java
```

Niet wat je verwachtte? Tijd om dit euvel recht te zetten. Open het bestand dat fouten bevat met een editor naar keuze en pas het aan. Tracht het opnieuw te compileren. Wanneer de fouten zijn verholpen, kun je het programma uitvoeren met de opdracht:

```
java PetStore
```

Eens je het programma aan de praat hebt gekregen, is het tijd geworden om de rechtzettingen op te slaan in een nieuwe commit. Je kunt de `add`-opdracht van hogerop, inclusief het sterretje, gewoon herhalen. Git zal detecteren welke bestanden daadwerkelijk zijn gewijzigd en enkel wijzigingen opnemen in de nieuwe commit.

Voer vervolgens `git commit` uit met een gepaste commit message. Gebruik de opdracht `git log` om na te gaan of de commit wel degelijk werd toegevoegd aan de geschiedenis.

1.7 Iets te ijverig?

Nu werkt de applicatie al, maar als je de code even verder uitspit, zul je merken dat een van de `.java`-bestanden geen enkele bijdrage levert. We kunnen dit dan ook beter verwijderen uit de repository.

Merk op dat we het bestand zullen verwijderen uit toekomstige commits, maar dat het aanwezig zal blijven in de twee commits die zich reeds in de geschiedenis bevinden. Dit is immers de essentie van versiebeheer: de commits zijn snapshots van de code op dat moment en worden niet beïnvloed door toekomstige wijzigingen.

Het bestand van de schijf verwijderen kunnen we doen met behulp van Windows Explorer, of nog met de Bash-opdracht `rm` (remove). Er stelt zich dan echter een probleem: het bestand is wel verdwenen van de schijf, maar Git is hier niet

van op de hoogte. In plaats daarvan moeten we dan ook de `rm`-opdracht van Git gebruiken, die zich verder identiek gedraagt aan die van Bash. Indien we bijvoorbeeld `PetStore.java` willen verwijderen, gebruiken we de opdracht:

```
git rm PetStore.java
```

Op analoge wijze kunnen we bestandsnamen wijzigen met behulp van `git mv` (move), dat zich gedraagt zoals `mv`. Om bijvoorbeeld `A.java` om te dopen tot `B.java`, zouden we de opdracht `git mv A.java B.java` gebruiken.

Per ongeluk `PetStore.java` of een ander bestand verwijderd? Maak dan gebruik van de kracht van Git en haal de laatst gekende bestandsinhoud op uit de geschiedenis:

```
git checkout HEAD PetStore.java
```

Hierin verwijst `HEAD` naar de 'leeskop' die bijhoudt waar in de geschiedenis van de repository we het laatst waren aanbeland, en dus op welke commit de huidige wijzigingen zullen gebaseerd zijn. Om meer info over die commit te verkrijgen, kunnen we de opdracht `show` gebruiken:

```
git show HEAD
```

Zoals je intussen al hebt gemerkt, beschikt Git over talloze ingebouwde opdrachten met eindeloze mogelijkheden. Je hoeft deze uiteraard niet allemaal uit het hoofd te kennen. Gaandeweg zul je de belangrijkste onder de knie krijgen.

1.8 Op naar GitHub!

Alvorens we enkele mogelijkheden van Git in NetBeans bekijken, zullen we onze `petstore`-repository nog publiceren op de GitHub-installatie van de universiteit. Hierdoor kun je de volledige geschiedenis van je code afhalen en wijzigen vanop eender welke computer. Eventueel kun je deze ook delen met iedereen die toegang heeft tot GitHub.

Eerst moeten we wel eenmalig een zogenaamde publieke sleutel aanmaken. Die sleutel dient als identiteitsbewijs en vervangt dus je wachtwoord bij het verbinden met GitHub. Je maakt zo'n sleutel aan door de volgende opdracht uit te voeren in Git Bash:

```
ssh-keygen -t rsa -C "voornaam.familienaam@ugent.be"
```

Wanneer `ssh-keygen` vraagt waar je de sleutel wilt opslaan, mag je de standaardwaarde aanvaarden door op Enter te drukken. Je kunt de sleutel desgewenst ook beveiligen met een wachtwoord, dat je zult moeten opgeven telkens je verbinding maakt met GitHub.

Vervolgens moet je de gegenereerde sleutel nog registreren bij GitHub. Log in op GitHub met je UGent-account als je dit nog niet deed. De settings van GitHub vind je door het menu open te klappen dat bij je 'profile picture' hoort (rechtsboven). Daar navigeer je verder naar de pagina `SSH and GPG keys`. Voeg een nieuwe SSH-key toe (knop '`New SSH Key`' rechtsboven). De titel kies je zelf; in het tweede tekstveld komt de key. Die bevindt zich in het tekstbestand `id_rsa.pub` in de subdirectory `.ssh` van jouw gebruikersdirectory. Vind je deze niet terug, open dan het adres `%USERPROFILE%\.ssh` in Windows Explorer en open vervolgens `id_rsa.pub` met bijvoorbeeld Notepad.

Eens je de sleutel hebt toegevoegd, hoef je enkel nog een repository aan te maken op GitHub en je lokale repository ermee te synchroniseren. Surf naar de pagina New Repository. Daar kom je, vanop de startpagina, door de groene knop rechts '`New repository`' aan te klikken. Vul bij Repository name de naam `petstore` in en maak de repository aan. Je krijgt dan een bevestigingspagina te zien.

Onder Quick setup vind je op de bevestigingspagina de URL van je repository. Misschien is standaard de HTTP-URL geselecteerd, terwijl wij enkel geïnteresseerd zijn in de SSH-URL. Klik deze laatste dus aan alvorens verder te gaan.

Onderaan de bevestigingspagina vind je dan de twee opdrachten die je in Git Bash moet uitvoeren om je lokale repository eraan te koppelen:

```
git remote add origin git@github.ugent.be:gebruikersnaam/petstore.git
git push -u origin master
```

Na het uitvoeren van de `push`-opdracht, bevindt zich in de repository op GitHub exact dezelfde inhoud als in deze op de pc. Bijgevolg kun je op GitHub rondklikken om de bestandsinhouden te bekijken, maar ook bijvoorbeeld de geschiedenis van elk bestand doorlopen.

Indien je de applicatie verder wenst te wijzigen, en dus de repository-inhoud wilt bijwerken, werk je gewoon verder met de lokale repository. Na elke `commit`-opdracht kun je meteen een `push`-opdracht uitvoeren om ook de GitHub-repository up-to-date te houden. Hoe vaak je dit daadwerkelijk doet, is echter een kwestie van persoonlijke voorkeur.

2 Via NetBeans

Je hebt nu al heel wat functionaliteit van Git onder de knie. Aangezien we in het opleidingsonderdeel Softwareontwikkeling 1 intensief gebruikmaken van NetBeans IDE, loont het de moeite om de handmatig ingevoerde opdrachten te vergelijken met hun NetBeans-equivalent. Je mag echter vrij kiezen hoe je te werk gaat.

Om de mogelijkheden van Git in NetBeans uit te proberen, krijg je opnieuw afgewerkte broncode ter beschikking. In dit archiefbestand bevindt zich het NetBeans-project `petstore-nb`, dat nog geen Git-functionaliteit bevat. We zullen nu NetBeans gebruiken om een Git-repository aan te maken, enkele commits uit te voeren, en de inhoud te pushen naar GitHub.

2.1 Git toevoegen

Importeer het project `petstore-nb` in NetBeans door onder File te kiezen voor Import Project en vervolgens From ZIP… Het project verschijnt dan onder Projects.

Rechtsklik op de naam van het project en selecteer onder Versioning de opdracht Initialize Git Repository. In het dialoogvenster dat verschijnt kun je opgeven in welke directory je de repository wilt aanmaken. De standaardwaarde is de projectdirectory, dus je kunt deze aanvaarden.

Daarmee is de Git-repository aangemaakt. Je kunt de uitvoer van de opdracht bekijken onder het tabblad Output. Bemerk dat de namen van alle bestanden in het project een groene kleur hebben gekregen.

2.2 Een eerste commit

Tijd om onze bestanden toe te voegen aan de repository. Je kunt in NetBeans op twee manieren interageren met Git. Allereerst kun je rechtsklikken op een bestand of directory, of meerdere tegelijk, en naar het submenu Git navigeren. Als alternatief kun je in de menubalk voor Team kiezen. Merk echter op dat dit laatste inwerkt op de bestanden of directory's die momenteel actief zijn. Dit kan een selectie uit de `.java`-bestanden op het tabblad Projects zijn, maar evengoed het actieve tabblad, wat tot verwarring kan leiden. Rechtsklikken geniet daarom onze voorkeur.

In tegenstelling tot `git commit` vereist de NetBeans-opdracht Commit… niet dat je vooraf bestanden toevoegt. Je zult de opdracht Add van NetBeans dan ook veel minder vaak gebruiken dan de opdracht `git add`. Rechtsklik dus op het project en kies onder Git meteen voor Commit… Er verschijnt een dialoogvenster met informatie over de nieuwe commit. Voer bovenaan alvast de beschrijving `Initial commit` in.

Onderaan het dialoogvenster kun je de bestanden selecteren die deel uitmaken van de commit. Je vindt hier niet enkel de `.java`-bronbestanden, maar ook onder andere de inhoud van de directory `nbproject` en het bestand `build.xml`.

Gemakshalve kiezen we ervoor om de door NetBeans voorgestelde inhoud te aanvaarden. Directory's met binaire inhoud, zoals `build`, werden immers automatisch weggelaten, zoals het hoort. Klik op OK om de commit aan te maken.

Merk op dat het tabblad Projects niet alle bestanden weergeeft. Wens je de volledige inhoud van je projectdirectory te bekijken, schakel dan over naar het tabblad Files. Op die manier kun je onder andere de inhoud van de directory `nbproject` bekijken.

2.3 De geschiedenis bekijken

De Git-geschiedenis is hiermee geïnitialiseerd. Door in het rechtsklikmenu onder Git te kiezen voor Show History, kunnen we de geschiedenis doorzoeken. Er verschijnt een nieuw tabblad, dat initieel leeg is. Klik onderaan op Search om de 10 nieuwste commits weer te geven. Uiteraard wordt er nog maar eentje gevonden.

Door deze enige commit open te klappen met behulp van het plusteken, verkrijg je de details ervan. Je kunt ook rechtsklikken op elk gewijzigd bestand om de opgeslagen inhoud te bekijken. Aangezien we nog maar één commit uitvoerden, stemt deze inhoud perfect overeen met de code, maar naarmate de geschiedenis van de repository aangroeit, zal dit natuurlijk veranderen.

2.4 Nog een commit

Dubbelklik op de klasse `PetStore`. Helemaal onderaan, in de `main`-methode, worden de vier diersoorten ingesteld die onze dierenwinkel verkoopt. Echter, welke dierenwinkel verkoopt nu eenhoorns? Verwijder daarom de regel met `Unicorn` en sla het bestand op.

Het bestand is nu al gewijzigd, maar Git is hier nog niet van op de hoogte. Wel heeft de bestandsnaam een blauwe kleur gekregen. Dit wijst erop dat het bestand wijzigingen bevat die nog niet werden opgeslagen in een commit.

Voor we de commit doorvoeren, kunnen we echter nog een tweede wijziging aanbrengen. De klasse `Unicorn` is immers overbodig geworden. Verwijder daarom het bestand. Je hoeft geen bijkomende actie te ondernemen om Git op de hoogte te brengen van het verwijderen; NetBeans brengt dit in orde.

Onze commit is klaar, dus laten we hem toevoegen aan de geschiedenis. Rechtsklik nogmaals op het project en kies onder Git voor Commit… Onderaan het dialoogvenster zie je dit keer twee wijzigingen: `PetStore.java` werd aangepast en `Unicorn.java` verwijderd.

Vergeet niet om de commit message te wijzigen. Voer bijvoorbeeld de beschrijving `No longer selling unicorns :-(` in en klik vervolgens op OK om de commit op te slaan.

In de lijst met bestanden wordt `PetStore.java` opnieuw zwart; er zijn dus geen openstaande wijzigingen meer. Als je nogmaals kiest voor Show History, zul je merken dat er nu twee commits in de geschiedenis staan.

2.5 Over naar GitHub

Laten we nu ook dit project op GitHub publiceren. Surf opnieuw (via de startpagina of de bijgevoegde link) naar de pagina New Repository en maak de repository `petstore-nb` aan.

Om NetBeans duidelijk te maken waarheen de repository-inhoud moet worden gepusht, rechtsklik je wederom op het project en kies je onder Git voor het submenu Remote, waarin je de opdracht Push… terugvindt. Deze opent een dialoogvenster, waarin je de Repository URL kunt invoeren.

Hier heb je twee mogelijkheden; de tweede is het eenvoudigst.

1. Je gebruikt **SSH**. In GitHub, bij de knop '`Clone or download`' die bij je repository hoort, kies je voor SSH. Copieer de gegeven git@github.ugent.be:...-URL (gebruik het icoontje met clipboard). In NetBeans kies je in het contextmenu van het project voor Git - Remote - Push. Daar plak je de URL. Het veld Username mag je blanco laten; dit is `git`, maar staat al vooraan in de URL. Wel moet je nog je publieke sleutel meegeven. Activeer dus de optie Private/Public Key File en navigeer naar het bestand `id_rsa` (zonder de extensie `.pub`) in de directory `C:\Users\Gebruiker\.ssh`. Klik op Next om verbinding te maken.
2. Je werkt via **HTTPS**. In GitHub, bij de knop '`Clone or download`' die bij je repository hoort, kies je voor HTTPS. Copieer het gegeven adres; dit is de Repository URL. In NetBeans kies je in het contextmenu van het project voor Git - Remote - Push. Daar plak je de URL. User en password zijn die van je UGent-account.

Na de vorige stap (waarbij je dus de keuze had tussen SSH en HTTPS) krijg je een lijst met zogenaamde branches te zien. In het volgende labo gaan we daar in detail op in. Voorlopig volstaat het om een vinkje te plaatsen bij de enige branch en op Next te klikken. Tot slot klik je op Finish om een push uit te voeren.

De volgende keer dat je wenst te pushen naar GitHub, hoef je uiteraard deze configuratiestappen niet meer uit te voeren. Het volstaat dan om in het submenu Remote te kiezen voor Push to Upstream i.p.v. voor Push. We hebben dus in wezen de opdracht `git push -u origin master` van weleer uitgevoerd aan de hand van enkele dialoogvensters.

3 Vooruitblik

Hoewel we al aardig wat stof hebben behandeld, is het zeker niet de bedoeling om in deze labo's alle functionaliteit van Git te bespreken. In het tweede en laatste labo gaan we in op iets geavanceerdere functionaliteit van Git en bekijken we de mogelijkheden op het vlak van samenwerking.