technische universität
dortmund

Wirtschafts- und
Sozialwissenschaftliche Fakultät

Git Dokumentation

FiRRM
Finance, Risk & Resource Management

Inhaltsverzeichnis

1 Einführung in Git

1.1 Git und andere Versionsverwaltungssysteme

In dieser Dokumentation erfahrt ihr was Git ist, wozu es dient, wie ihr es einrichtet und wie es funktioniert. Außerdem wird euch gezeigt, wie ihr euch mit unserem Lehrstuhlserver verbindet, um an Projekten mitzuarbeiten.
Zunächst einmal ist Git eine Software, die zur Versionsverwaltung eingesetzt wird. Allgemein dient Versionsverwaltungssoftware dazu, eine Folge von Änderungen an einer Datei zu erfassen, sodass man im Nachhinein jede Version der Datei aufrufen kann. Gerade bei der Entwicklung von Software, bei der mehrere Parteien involviert sind, stellt sie ein wichtiges Werkzeug dar. Es erlaubt euch, ausgewählte Dateien in einen früheren Zustand zurückzusetzen, das gesamte Projekt in einen früheren Zustand zurückzusetzen, Änderungen im Laufe der Zeit zu vergleichen, zu sehen, wer zuletzt etwas geändert hat, das ein Problem verursachen könnte, wer ein Problem eingeführt hat und wann, usw. Die Verwendung einer Versionsverwaltungssoftware bedeutet demnach auch, dass beschädigte oder verlorene Dateien mit wenig Aufwand wieder hergestellt werden können. Bei der kollaborativen Entwicklung von Projekten ist es nötig, dass alle Teilnehmer auf die Daten zugreifen können. Eine Lösung für dieses Problem bieten zentrale Versionsverwaltungssysteme. Diese Systeme verfügen über einen einzigen Server, der alle versionierten Dateien enthält und eine Reihe von Clients, die Dateien von diesem zentralen Ort auschecken. Offensichtlich haben diese zentralisierten System mit dem Problem zu kämpfen, dass das ganze Projekt direkt abhängig ist von der Zuverlässigkeit eines einzigen Servers. Geht der Server für eine Stunde down, so wäre in diesem Zeitraum keine Kollaboration möglich. Auch könnten Fortschritte nicht abgespeichert werden. Fiele der Server gänzlich aus, so wäre dies sogar mit dem Verlust aller nicht lokal gespeicherten Daten verbunden. Dieses Problem wird durch den Einsatz eines verteilten Versionsverwaltungssystems gelöst. In einem verteilten Versionsverwaltungssystem (z.B. Git) beziehen die Clients nicht nur die neuesten Änderungen der Dateien, sondern kopieren das gesamte Repository einschließlich seiner vollständigen Historie. Wenn also ein Server ausfällt, über den die Kollaboration erfolgte, können alle Client-Repositories auf den Server kopiert werden, um die Daten wieder herzustellen. Darüber hinaus lassen sich verschiedene voneinander getrennte Repositories verwalten, sodass auch innerhalb des selben Projekts verschiedene Workflows eingerichtet werden können. Der Hauptunterschied zwischen Git und anderen Versionsverwaltungssystemen ist die Art und Weise, wie Git seine Daten interpretiert. Die meisten

anderen Systeme speichern Informationen konzeptionell als eine Liste von dateibasierten Änderungen. Sie (z.B. CVS, Subversion, Perforce, Perforce, Bazaar, usw.) interpretieren die gespeicherten Daten als eine Menge von Dateien und ihren vorgenommenen Änderungen über die Zeit. Git interpretiert und speichert seine Daten hingegen nicht auf diese Art und Weise. Stattdessen betrachtet Git seine Daten eher als eine Serie von Momentaufnahmen eines Miniatur-Dateisystems. Jedes Mal, wenn Daten geändert werden, speichert Git im Grunde eine Momentaufnahme davon, wie die Dateien in diesem Moment aussehen. Zusätzlich speichert Git einen Verweis auf diese Momentaufnahme. Aus Effizienzgründen speichert Git unveränderte Dateien nicht erneut ab, sondern speichert lediglich einen Link zu der vorherigen identischen Datei, die bereits zuvor gespeichert wurde. Git interpretiert seine Daten also viel mehr als einen Stream von Momentaufnahmen. Ein weiterer Vorteil von Git ist seine Integrität. Durch den Einsatz von Hashes, ist es nicht möglich Änderungen am Projekt vorzunehmen, ohne dass Git diese aufzeichnet.
Innerhalb von Git können Dateien einen von drei Zuständen annehmen: committed, modified und staged:

- Committed bedeutet, dass die Daten sicher in eurer lokalen Datenbank gespeichert wurden.
- Modified bedeutet, dass ihr die Datei geändert, aber noch nicht in eure Datenbank übertragen habt.
- Staged bedeutet, dass ihr eine modifizierte Datei markiert habt, sodass sie bei der nächsten Momentaufnahme enthalten sein soll.

Dies führt uns zu den drei Hauptbereichen eines Git-Projekts: dem Git-Directory, dem Working-Tree und der Staging-Area.
Die Git-Directory ist der Ort, an dem Git die Metadaten und Objektdatenbank für euer Projekt speichert. Dies ist der wichtigste Teil von Git. Die Git-Directory ist der Teil, der kopiert wird, wenn ihr ein Repository von einem anderen Computer klont. Der Working-Tree ist ein einzelnes Auschecken einer Version des Projekts. Diese Dateien werden aus der komprimierten Datenbank in der Git-Directory herausgezogen und auf der Festplatte abgelegt. So könnt ihr diese nun verwenden oder verändern. Die Staging-Area ist eine Datei, die normalerweise in eurem Git-Verzeichnis enthalten ist und Informationen darüber speichert, was in euren nächsten Commit einfließt.
Der grundlegende Git-Workflow sieht ungefähr wie folgt aus:

1. Ihr modifiziert Dateien in eurem Working-Tree.
2. Ihr staged selektiv nur die Änderungen, die Teil des nächsten Committs sein sollen. Somit werden nur die selektierten Änderungen der Staging-Area hinzugefügt.
3. Ihr führt einen Commit durch, der die Dateien aus der Staging-Area nimmt und eine Momentaufnahme dieser in eurer Git-Directory speichert.

Wenn sich eine bestimmte Version einer Datei in der Git-Directory befindet, ist sie 'commited'. Wenn sie modifiziert wurde und der Staging-Area hinzugefügt wurde, ist sie 'staged'. Wurde sie verändert, jedoch nicht gestaged, so heißt sie 'modified'.

Es gibt mehrere Möglichkeiten Git auf eurem Rechner zu verwenden. Zwar gibt es auch grafische Benutzeroberflächen, jedoch werden wir Git über die Kommandozeile bedienen. Zum einen ist die Bedienung einer entsprechenden GUI intuitiv zu erlernen, wenn man die Verwendung mit Hilfe der Kommandozeile beherrscht (andersherum eher nicht). Zum anderen ist es nur möglich alle Git-Befehle zu verwenden, wenn man die Kommandozeile nutzt.

1.2 Installation

1.2.1 Linux

Wenn ihr die grundlegenden Git-Tools unter Linux via eines binary Installers installieren möchtet, könnt ihr dies in der Regel über das grundlegende Paketverwaltungs-Tool tun, das eurer Distribution beiliegt. Wenn ihr z.B. Fedora verwendet (oder eine eng verwandte RPM-basierte Distribution wie RHEL oder CentOS), könnt ihr dnf verwenden:

```
$ sudo dnf install git-all
```

Wenn ihr auf einer Debian-basierten Distribution wie Ubuntu arbeitet, versucht es mit apt-get:

```
$ sudo apt-get install git-all
```

Für weitere Informationen, könnt ihr die Seite http://git-scm.com/download/mac besuchen.

1.2.2 Mac

Es gibt mehrere Möglichkeiten, Git auf einem Mac zu installieren. Am einfachsten ist es wahrscheinlich, die Xcode Command Line Tools zu installieren. Auf Mavericks (10.9) oder höher könnt ihr dies tun, indem ihr einfach versucht, Git vom Terminal aus zu starten.

```
$ git --version
```

Wenn ihr es noch nicht installiert habt, werdet ihr aufgefordert, es zu installieren. Wenn ihr eine aktuellere Version wünscht, könnt ihr diese auch über einen binary Installer installieren. Ein macOS Git-Installer steht auf der Git-Website (http://git-scm.com/download/mac) zum Download zur Verfügung.

1.2.3 Windows

Es gibt auch einige Möglichkeiten, Git unter Windows zu installieren. Der offiziellste Build steht auf der Git-Website zum Download bereit. Geht dazu einfach auf http://git-scm.com/download/win und der Download startet automatisch. Beachtet dabei, dass es sich hierbei um ein Projekt namens 'Git for Windows' handelt, das von Git selbst getrennt ist; weitere Informationen dazu findet ihr unter https://git-for-windows.github.io/.

1.3 Erste Schritte

Git bietet ein Tool namens 'git config', mit dem sich die Konfigurationsvariablen einsehen und verändern lassen. So lassen sich sowohl die Darstellung als auch die Funktionsweise von Git anpassen. Beispielsweise solltet ihr zunächst Git eure Identität angeben. Dazu tragt ihr eure E-Mail-Adresse und euren Benutzernamen ein. Dies ist wichtig, da Git diese Informationen für jeden eurer Commits verwendet.

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Um eure aktuelle Konfiguration einzusehen, gebt ihr folgenden Befehl ein:

```
$ git config --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

Möchtet ihr hingegen einzelne Variablen auslesen, so erreicht ihr dies mit

```
$ git config user.name
John Doe
```

Zu Beginn ist der Standardeditor eures Rechners als Editor ausgewählt. Dieser Editor wird verwendet, wenn Git Nachrichten von euch benötigt. Zu empfehlen wäre euch allerdings, nicht den Standardeditor zu verwenden, sondern 'notepad++' (https://notepad-plus-plus.org/download/v7.5.6.html). Diesen ladet ihr unter angegebener Adresse herunter. Anschließend richtet ihr ihn unter Windows mittels

```
$ git config --global core.editor "'C:\Program Files (x86)\Notepad++\
    notepad++.exe'-multiInst -nosession"
```

unter Angabe des Pfades der notepad++.exe-Datei als Standardeditor ein. Für andere Betriebssystem verweise ich an dieser Stelle wieder auf https://git-scm.com/book/en/v2.
Git bietet darüber hinaus eine nützliche Hilfefunktion, solltet ihr Informationen über bestimmte Git-Funktionen benötigen. Mit

```
$ git help
```

erhaltet ihr eine Übersicht über geläufige Git-Funktionen. Wünscht ihr hingegen Informationen zu spezifischen Befehlen zu erhalten, so erreicht ihr dies durch Anfügen des jeweiligen Befehls. Eine Hilfsseite für den branch-Befehl öffnet sich beispielsweise nach folgender Eingabe

```
$ git help branch
```

2 Grundlagen

2.1 Repository einrichten

Um ein Git-Repository zu erhalten gibt es im Allgemeinen zwei Wege.

1. Ihr könnt ein bestehendes lokales Verzeichnis, auf das die Versionsverwaltung aktuell nicht angewendet wird, auswählen und zu einem Versionsverwaltungsverzeichnis umfunktionieren

2. Ihr könnt ein bereits bestehendes Git-Repository klonen

Wir betrachten zunächst den ersteren der beiden Fälle. Zunächst müsst ihr dafür das entsprechende Verzeichnis auswählen. In diesem Fall habe ich dafür zuvor einen Ordner 'my_project' angelegt. Für Windows erreicht ihr dies beispielsweise folgendermaßen (Für andere Betriebssysteme: Schlagt in https://git-scm.com/book/en/v2 nach):

```
$ cd C:\Benutzer\my_project
```

Anschließend gebt ihr folgenden Befehl ein:

```
$ git init
```

Dies erzeugt ein neues Unterverzeichnis namens .git, das alle notwendigen Repository-Dateien enthält. Wollt ihr statt eines neuen (leeren) Ordners, bereits bestehende Dateien versionsverwalten, so müsst ihr diese Dateien auswählen und den Befehl 'add' durchführen. Anschließend führt ihr einen 'commit'-Befehl durch. Z.B.

```
$ git add testdatei.xlsx
$ git commit -m 'hier commit-Kommentar einfuegen'
```

Nun kommen wir zur zweiten (und für uns interessanteren) Möglichkeit, ein Git-Repository zu erhalten. Wenn ihr eine Kopie eines bestehenden Git-Repositorys beziehen möchtet - zum Beispiel ein Projekt, an dem ihr mitarbeiten möchtet - benötigt ihr den Befehl 'git clone'. Wollt ihr z.B. die frei verfügbare Bibliothek 'libgit2' klonen und in einem Ordner 'mylibgit' unterbringen, so erreicht ihr dies mit

```
$ git clone https://github.com/libgit2/libgit2  mylibgit
```

Obiger Befehl erzeugt ein Verzeichnis namens 'mylibgit', initialisiert ein .git-Verzeichnis darin, lädt alle Daten für dieses Repository herunter und checkt eine Arbeitskopie der neuesten Version aus. Wenn ihr in das neue mylibgit-Verzeichnis geht, das gerade erstellt wurde, seht ihr dort die Projektdateien, die ihr bearbeiten oder verwenden könnt.
Für unsere Zwecke wollen wir nun beleuchten, wie ihr ein bestehendes Git-Repository von unserem Lehrstuhlserver klonen könnt. Ein solches Git-Repository wird auch als Remote-Repository bezeichnet, da es ein Repository darstellt, das im Internet oder in einem Netzwerk gehostet ist. Bei kollaborativen Projekten dieser Art ist es daher mit dem Klonen nicht getan. Vielmehr wollt ihr eure Arbeitsergebnisse später auch dem gehosteten Projekt beisteuern. Dazu erfahrt ihr in einem späteren Kapitel mehr. Zunächst ist es nötig, euch mit unserem Server zu verbinden. Zu Beginn müsst ihr euch daher einen sogenannten 'SSH public key' generieren. Glücklicherweise liegt das dazu benötigte Tool 'ssh-keygen' bereits vor, sofern ihr Git für Windows installiert habt. Für Linux und Mac wird das Tool durch das SSH-Paket bereitgestellt. Durch den Befehl

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/Max.Mustermann/.ssh/
    id_rsa):
Created directory '/c/Users/Max.Mustermann/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /c/Users/Max.Mustermann/.ssh/
    id_rsa.
Your public key has been saved in /c/Users/Max.Mustermann/.ssh/id_rsa.
    pub.
The key fingerprint is:
SHA256:X/ghTn0oyeUBaVuV20n4XXg0H74iV8yrcaxxDvtss7Y Max.Mustermann@KD-
    HKFDHF-001
```

erreicht ihr das gewünschte Ergebnis. Zuerst soll angegeben/bestätigt werden, wo der Schlüssel gespeichert werden soll (.ssh/id_rsa). Anschließend wird zweimal nach einer Passphrase gefragt, die ihr leer lassen könnt, wenn ihr kein Passwort eingeben möchtet, um den Key zu verwenden. Den Key findet ihr nun in angegebenem Ordner. Er besteht aus zwei Dateien, dem 'private key' und dem 'public key'. An dieser Stelle müsst ihr uns unter der Adresse Timotheos.Paraskevopoulos@tu-dortmund.de kontaktieren, sodass wir euch mit Hilfe eurer Uni-E-mail-Adresse einen Gitlab-Account einrichten können. Ihr erhaltet eine E-mail, in der ihr aufgefordert werdet, ein Passwort zu erstellen und euch bei Gitlab zu registrieren. Dazu folgt ihr den Links aus der E-Mail. Anschließend meldet ihr euch mit eurem Account bei Gitlab an. Im nächsten Schritt tragt ihr euren erstellten

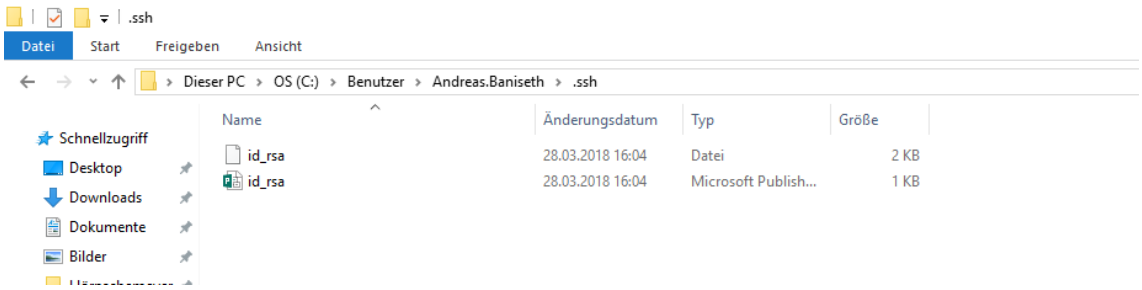

Abbildung 2.1: Hier seht ihr die beiden soeben erstellten private key und public key

public key bei Gitlab ein. Dazu geht ihr auf das Symbol am oberen rechten Rand und wählt 'Settings' aus. Als nächstes klickt ihr auf 'SSH-keys' am linken Bildschirmrand.

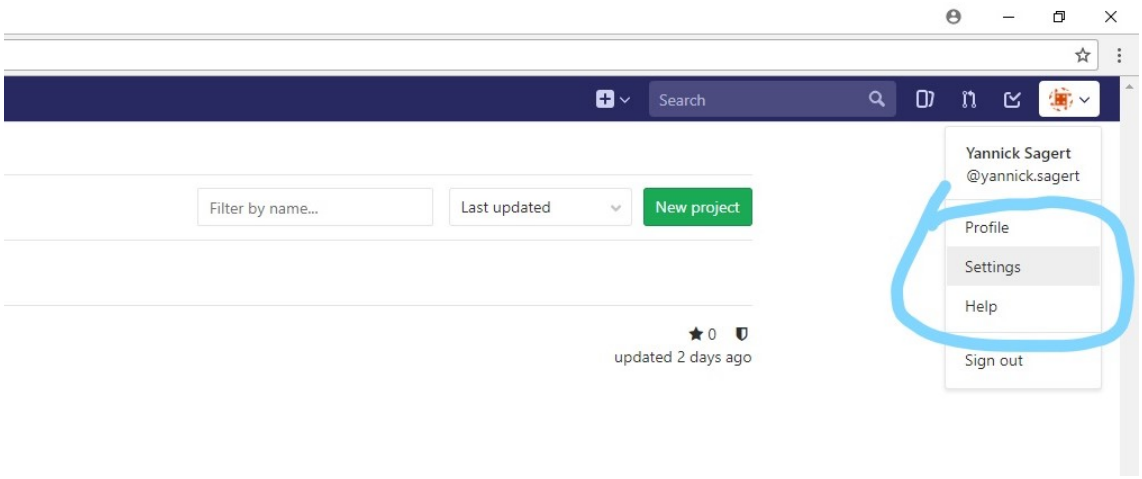

Abbildung 2.2: 'settings' findet ihr am oberen rechten Bildschirmrand

Anschließend kopiert ihr den Inhalt der Datei, die den public key enthält und fügt ihn im Textfeld 'Key' ein. Danach klickt ihr auf 'Add key'. Weitere Informationen hierzu findet ihr unter https://help.github.com/articles/connecting-to-github-with-ssh/.
Anschließend klont ihr das Projekt, an dem ihr mitwirken wollt. Zur Veranschaulichung habe ich ein Beispielprojekt namens 'testprojekt' erstellt. Um ein neues Repository zu erstellen, würdet ihr nun

```
$git clone git@git.firrm.de:yannick.sagert/testprojekt.git
$cd testprojekt
```

durchführen. Durch 'cd testprojekt' gelangt ihr direkt in den entsprechenden Ordner.

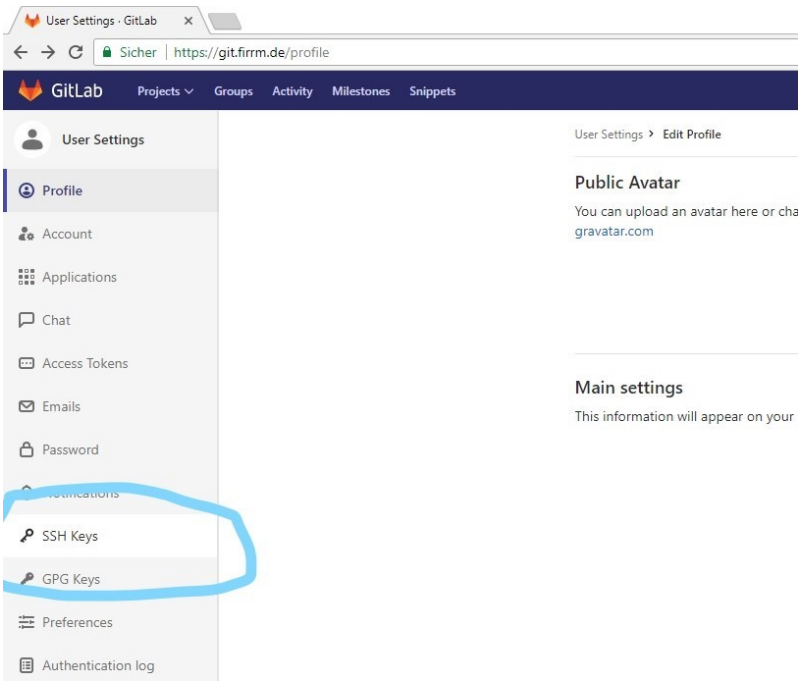

Abbildung 2.3: 'SSH-keys' findet ihr am linken Bildschirmrand

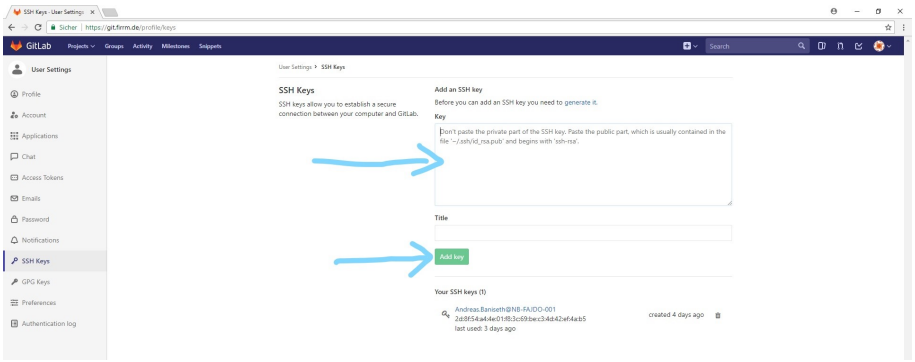

Abbildung 2.4: In diesem Fenster tragt ihr euren public key ein und fügt diesen anschließend hinzu.

2.2 Änderungen im Repository aufzeichnen

An dieser Stelle solltet ihr bereits über ein intaktes Git-Repository auf eurem lokalen Rechner verfügen und eine Kopie all seiner Dateien vor euch haben. Im weiteren Verlauf wollt ihr nun Änderungen an diesen Dateien vornehmen und Momentaufnahmen dieser Änderungen in euer Repository übertragen, sobald euer Projekt einen Zustand erreicht, den ihr aufzeichnen wollt. Jede eurer Dateien nimmt einen von zwei Zuständen an. Sogenannte getrackte Dateien sind Dateien, die in eurer letzten Momentaufnahme bereits enthalten waren. Diese Dateien können unmodified, modified oder staged sein. Es handelt sich also um alle Dateien, die Git bereits kennt. Zu den ungetrackten Dateien zählt man alle Dateien, die weder in der letzten Momentaufnahme enthalten, noch in der staging area enthalten sind. Zur Veranschaulichung: Habt ihr soeben ein Repository geklont, so sind alle eure Dateien getrackt und unmodified. Wenn ihr Dateien bearbeitet, werden

sie von Git als modifiziert angesehen, da sie seit eurem letzten commit verändert wurden. Anschließend stellt ihr diese Dateien selektiv in eurer nächsten Momentaufnahme bereit und der Zyklus wiederholt sich.

2.2.1 Status der Dateien einsehen

Das Hauptwerkzeug, mit dem ihr feststellen könnt, welche Dateien sich in welchem Zustand befinden, ist der Befehl 'git status'. Wenn ihr diesen Befehl direkt nach einem Klon ausführt, solltet ihr etwas sehen, das dem Folgenden sehr nahe kommt:

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

Der Befehl gibt Auskunft darüber, dass keine eurer getrackten Dateien verändert wurden. Darüber hinaus liegen auch keine ungetrackten Dateien vor, andernfalls wären sie hier aufgelistet. Schließlich gibt der Befehl noch Auskunft darüber aus, in welchem 'branch' (deutsch: Zweig) ihr euch befindet. Auf dieses Thema wollen wir später noch im Detail zurückkommen.

2.2.2 Neue Dateien tracken

Fügt ihr eurem Projekt nun eine neue Datei hinzu, so würde diese Datei als ungetrackte Datei gelistet werden. Dies lässt sich mit dem Befehl 'git status' leicht überprüfen. Um diese Datei nun zu tracken und eurem Projekt mit der nächsten Momentaufnahme hinzuzufügen, führt ihr zunächst einen add-Befehl aus. In diesem Beispiel seht ihr, wie ein Word-Dokument hinzugefügt werden soll.

```
$ git add beispiel.docx
```

Nun ist die Datei staged, d.h. sie wartet nun darauf, mit der nächsten Momentaufnahme in euer Projekt aufgenommen zu werden. Dies erreicht ihr mit einem commit-Befehl, wie wir ihn weiter oben schon verwendet haben.

```
$ git commit -m 'Euer commit-Kommentar'
```

In dieser Variante (man beachte den Zusatz '-m') könnt ihr euren commit-Kommentar direkt über die Konsole eingeben. Führt ihr hingegen folgenden Befehl aus,

```
$ git commit
```

so öffnet sich euer eingangs eingerichteter Editor und ihr könnt euren commit-Kommentar über diesen eingeben.
Das ganze Prinzip ändert sich übrigens nicht, wenn ihr eine bereits getrackte Datei modifiziert und anschließend erneut trackt.

2.2.3 Commit-History einsehen

Nachdem ihr mehrere commits erstellt oder ein Repository mit einer bestehenden commit-History geklont habt, werdet ihr hin und wieder zurückblicken wollen, was bisher in eurem Projekt geschehen ist. Das einfachste und leistungsfähigste Werkzeug dafür ist der Befehl 'git log'. Nach Eingabe dieses Befehls werden euch alle bisherigen commits inklusive jeweiliger commit-Kommentare in umgekehrter chronologischer Reihenfolge angezeigt. Natürlich gibt es eine Anzahl von auswählbaren Optionen, um weitere Informationen zu erhalten. Dazu verweise ich an dieser Stelle gerne wieder auf https://git-scm.com/book/en/v2.

2.3 Arbeiten mit Remotes

Unter Remote-Repositories versteht man Versionen eures Projekts, die irgendwo im Internet oder im Netzwerk gehostet sind. Arbeitet ihr mit anderen Personen zusammen, müsst ihr diese Remote-Repositories managen und sowohl Daten ziehen als auch Daten hochladen. In diesem Abschnitt erfahrt ihr wie das funktioniert.
In Kapitel 2.1 habt ihr bereits erfahren, wie ihr ein bestehendes Repository klont. Dieses Repository wird anschließend von Git als eines eurer Remote-Repositories geführt. Eine Übersicht aller Remote-Server erhaltet ihr mit

```
1 $ git remote
  origin
```

Sofern ihr, wie in Kapitel 2.1 beschrieben, ein Repository von unserem Lehrstuhlserver geklont habt, wird dieser unter dem Namen 'origin' gelistet. Standardmäßig ist dies der Name, den Git einem Server zuordnet, von dem ihr geklont habt.
Abgesehen vom clone-Befehl, ist es ebenso möglich ein Remote explizit hinzuzufügen. Dabei habt ihr die Möglichkeit, einen von 'origin' verschiedenen Namen zu wählen. So richtet ihr mit

```
$ git remote add lehrstuhlserver git@git.firrm.de:yannick.sagert/
    testprojekt.git
2 $ git remote
  origin
```

```
4 lehrstuhlserver
```

einen weiteren Remote unter dem Namen 'lehrstuhlserver' ein. In diesem Beispiel handelt es sich bei 'origin' und 'lehrstuhlserver' um den gleichen Server. Angenommen ihr wollt nun die Daten von 'lehrstuhlserver', die ihr noch nicht besitzt herunterladen. Dann erreicht ihr dies mit

```
$ git fetch lehrstuhlserver
2 remote: Counting objects: 6, done.
remote: Compressing objects: 100% (5/5), done.
4 remote: Total 6 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
6 From git.firrm.de:yannick.sagert/testprojekt
 * [new branch]      master     -> lehrstuhlserver/master
```

Der master-Branch ist nun lokal verfügbar als lehrstuhlserver/master. Wären noch weitere Branches in dem Remote-Projekt vorhanden, so würden diese ebenfalls heruntergeladen werden. Die soeben erhaltenen Branches könnt ihr nun beispielsweise in einen eurer Branches mergen. Was man unter Branches und Merging versteht, erfahrt ihr in Kapitel 2.4.
Habt ihr nun lokal einige Änderungen an euren Dateien vorgenommen, so seid ihr sicher daran interessiert, diese dem Remote-Projekt hinzuzufügen. Dafür müsst ihr die Daten hochladen. Wenn ihr beispielsweise euren Master-Branch zum 'origin'-Server hinzufügen wollt, so erreicht ihr dies durch

```
1 $ git push origin master
```

Es gilt dabei zu beachten, dass ihr nur Daten pushen könnt, sofern sich euer lokales Repository auf dem aktuellen Stand befindet. D.h. es müssen zuvor alle Daten aus dem Remote-Repository gefetcht worden sein. Darüber hinaus gibt es natürlich noch viele weitere Befehle, die euch den Umgang mit Remotes erleichtern. So ist es zum Beispiel auch möglich Remotes nachträglich umzubenennen oder sie wieder zu löschen.

2.4 Branching

Branching beschreibt den Prozess, sich vom Hauptzweig der Entwicklung abzugabeln und parallel zum Hauptzweig weiterzuarbeiten, ohne diesem in die Quere zu kommen. Branching gilt als das Hauptfeature von Git. Um die Funktionsweise des Branchings zu verstehen, müsst ihr euch daran erinnern, in welcher Art und Weise Git Daten speichert. Wie eingangs beschrieben, speichert Git Daten nicht als eine Liste von dateibasierten Änderungen, sondern als Folge von Momentaufnahmen. Beim Durchführen eines Commits, speichert Git ein Commit-Objekt, das einen Zeiger auf die Momentaufnahme des

von euch bereitgestellten Inhalts enthält. Dieses Objekt enthält darüber hinaus auch euren Namen, eure E-Mail-Adresse, den Commit-Kommentar und Zeiger auf den Commit oder die Commits, die direkt vor diesem Commit stattfanden (also seine Eltern [1]). Der erste Commit enthält demnach keine Eltern, während ein herkömmlicher Commit über ein Elternteil verfügt. Ein Commit, der aus einem Merge (to merge = verschmelzen) von zwei oder mehreren Branches resultiert, besitzt mehrere Elternteile.
Um dies zu veranschaulichen, betrachten wir ein Directory mit drei Dateien, die ihr allesamt staged und anschließend commitet. Die Dateien werden im Git-Repository als BLOBs[2] gespeichert. Darüber hinaus erstellt Git ein Commit-Objekt, das die Metadaten und einen Zeiger auf den Projektbaum enthält. Euer Git Repository enthält nun fünf Objekte: Jeweils einen Blob für die Inhalte der drei Dateien, einen Baum, der den Inhalt des Directorys beschreibt und einen Commit mit Zeiger auf eben jenen Baum und auf alle Metadaten der Commits.
Würde man nun ein paar Änderungen an den Dateien vornehmen und erneut einen Commit-Befehl durchführen, so würde der neue Commit einen Zeiger auf den vorherigen Commit enthalten.
Ein Branch ist nun nichts anderes als ein beweglicher Zeiger, der auf einen dieser Commits verweist. Der Standardbranch in Git heißt 'master'. Wenn ihr anfangt Commit-Befehle durchzuführen, zeigt der master-Branch immer zu eurem letzten Commit. Bei jedem neuen Commit bewegt er sich automatisch weiter. Für nähere Erläuterungen und Veranschaulichungen dieses Prinzips könnt ihr in https://git-scm.com/book/en/v2 nachschlagen.

2.4.1 Erstellen eines neuen Branches

Beim Erstellen eines neuen Branches, erschafft man, wie im letzten Abschnitt beschrieben, also einen neuen Zeiger. Dieser Zeiger kann dann auf einen Commit verweisen. Führt ihr den Befehl

```
$ git branch testing
```

aus, so wird ein neuer Zeiger erstellt, der auf denjenigen Commit verweist, auf dem ihr euch gerade befindet. Zwangsläufig stellt sich nun die Frage, woher Git weiß, auf welchem Commit ihr euch aktuell befindet. Git verfügt über einen speziellen Zeiger namens 'HEAD'. Dieser Zeiger verweist immer auf den aktuell aktiven Branch. In diesem Fall zeigt er also auf den Branch 'master'. Nun haben wir einen neuen Branch erstellt. Jedoch hat der branch-Befehl noch nicht dazu geführt, dass wir auch zum Branch 'testing' wechseln. Um nun zu dem bereits existierenden Branch 'testing' zu wechseln, führt ihr

```
$ git checkout testing
```

[1] Begriff aus der Graphentheorie, Informatik
[2] Datenobjekt aus der Informatik

aus. Dies veranlasst 'Head' zum 'testing'-Branch zu wechseln. Führt ihr nun weitere Commit-Befehle aus, haben diese keine Auswirkungen auf den 'master'-Branch. Eine Übersicht aller vorhandenen Branches erhaltet ihr mit

```
$ git branch
```

Eine wichtige Bemerkung ist, dass sich eure Dateien natürlich verändern, je nach dem zu welchem Branch ihr wechselt. Außerdem ist es empfehlenswert Branches nur zu wechseln, wenn sich im aktuellen Branch keine Dateien in der Staging-Area befinden, die auf einen Commit warten. Dies könnte zu Konflikten mit dem Branch führen, zu dem ihr wechseln wollt. Für nähere Beschreibungen, könnt ihr in https://git-scm.com/book/en/v2 nachschlagen.

2.4.2 Merging

Wie im letzten Abschnitt beschrieben, stellen wir uns vor, dass ihr von eurem 'master'-Branch auf euren 'testing'-Branch gewechselt seid. Nun nehmt ihr hier einige Änderungen vor und führt anschließend einen commit-Befehl durch. Angenommen eure Änderungen erwiesen sich als zufriedenstellend, so seid ihr nun daran interessiert, diese in euren 'master'-Branch zu übernehmen. Dazu wechselt ihr mit dem checkout-Befehl in den Zielbranch und führt anschließend einen merge-Befehl aus. Dies könnte wie folgt aussehen:

```
$ git checkout master
$ git merge testing
Updating f42c576..3a0874c
Fast-forward
index.html | 2 ++
1 file changed, 2 insertions(+)
```

Da ihr den 'testing'-Branch anschließend nicht weiter benötigt, ist es ratsam diesen zu löschen. Dazu führt ihr

```
$ git branch -d testing
Deleted branch testing (3a0874c).
```

aus. Auch beim Merging können Konflikte auftreten. Wenn ihr beispielsweise den gleichen Teil einer Datei in zwei Branches, die ihr miteinander mergen wollt, unterschiedlich geändert habt, führt dies zu Problemen. Genauere Informationen findet ihr in https://git-scm.com/book/en/v2.