GIT - Eine Einführung

Hauke Zühl <hzuehl@hauke-zuehl.de>

11. Mai 2018

Inhaltsverzeichnis

Kapitel 1

Was ist GIT?

GIT ist ein sogenanntes “Versionskontrollsystem” von Dateien in Projekten, das heisst, man kann die Veränderungen, die an Dateien im Laufe der Zeit durchgeführt werden, nachvollziehen. Dabei werden nicht nur Zeitstempel gespeichert, sondern auch welcher Benutzer die Änderungen durchgeführt hat. Dies ist dann ein Versionsstand eines Projektes.

GIT speichert also nur Veränderungen. Gut, bei sog. “Binärdateien”, also zum Beispiel LibreOffice-Dokumenten, geht das nicht, aber das ist ein anderes Thema. Immerhin kann man aber auch LO-Dokumente speichern, wie man an und für sich alles, was eine Datei ist, in GIT gespeichert werden kann.

Was GIT besonders macht, ist die Tatsache, dass es dezentral ist. Das bedeutet, dass jeder, der ein Projekt aus GIT herunterlädt (man sagt auch “auschecken” dazu) den kompletten Versionsverlauf auf seinem Rechner als lokale Kopie verliegen hat.

Im ersten Moment mag das ein Nachteil sein, denn man will ja nicht die Dateiversionen haben, die Max Mustermann vor 10 Jahren in GIT hochgeladen hat, aber keine Sorge, GIT arbeitet recht platzsparend.

Der Vorteil dieser Methode ist, dass man dadurch mindestens eine Sicherheitskopie auf einem Rechner hat. Arbeiten mehrere Leute an einem Projekt, kommen so diverse Sicherheitskopien zustande.

Diese Einführung soll mit einfachen Worten und Beispielen an GIT heranführen. Ich konzentriere mich dabei auf die Konsole, denn nur auf der Konsole kann man meiner Meinung nach die Arbeitsweise von GIT verinnerlichen. Grafische Clients kann man später immer noch einsetzen!

Kapitel 2

Fachbegriffe

Ohne Fachbegriffe, die überwiegend in Englisch sind, kommt man leider bei der Benutzung und der Erklärung von GIT nicht aus, deshalb hier die grundlegenden Fachbegriffe:

"Repository":

Man kann es als "Projekt" bezeichnen. Im Repository (verkürzt auch nur "Repo" genannt) findet man die Verzeichnnisse und Dateien des Projektes wieder.

"Einchecken":

Neue Dateien, die zum Repo hinzugefügt werden, werden "eingecheckt", aber wenn auch Änderungen an Dateien dem Repo hinzugefügt werden, spricht man von "einchecken". Auch "commit" genannt.

"Auschecken" / "Klonen":

Unter "Auschecken" versteht man umgangssprachlich(!) das Herunterladen eines GIT-Repositories. Korrekterweise bedeutet "auschecken", dass sog. "Zweige" auscheckt. Der Begriff des Zweiges wird später genauer erläutert. An dieser Stelle sei nur sovierl dazu gesagt, dass man mit Zweigen Projekte weiter unterteilen kann.

Wenn man ein Repository klont, dann bezieht sich das auf das GIT-Kommando, um ein Repository von einem GIT-Server herunterzuladen.

"Branch":

Ein "Branch" ist ein Zweig eines Projektes. Dazu später mehr.

"Merge":

Das Zusammenführen zweier Zweige wird als "merge" bezeichnet.

Kapitel 3

Erste Schritte mit GIT

Um mit GIT zu arbeiten, braucht man erst einmal das Programm.
Unter Linux kann man sich GIT mit Hilfe des distributionseigenen Installationsprogrammes installieren. Unter Debian oder Ubuntu zum Beispiel per *aptitude install git*, bzw. per *apt-get install git*. RedHat oder CentOS liefert GIT per *yum install git* auf die Platte. Und unter MacOS dürfte *brew install git* zum Erfolg führen. Mangels MAC kann ich das nicht prüfen.
Ein Tip: Vergiss erst einmal grafische Clients, das ist Blödsinn, denn damit lernt man nicht mit GIT umzugehen und zu verstehen! Denk auch nicht an irgendwelche Server oder gar an GitHub! Wir fangen einfach an und das ist lokal bei dir auf deinem Computer.

Los geht's!

Öffne eine Konsole deiner Wahl. Bash, zsh, ksh, egal. Irgendwas, wo du mit Hilfe deiner Tastatur deinem Rechner Kommandos senden kannst[1].
Es empfiehlt sich, für Git-Repos ein eigenes Verzeichnis anzulegen. Zum Beispiel *git* oder *projekte*, oder, oder, oder.
Ich selbst verwende *git*.

Also: *mkdir git*, dann *cd git*.

Jetzt legen wir ein Verzeichnis für unser erstes Projekt an. Lass uns das Projekt "Welt" nennen: *mkdir Welt*. Per *cd Welt* wechseln wir in das neue Verzeichnis. Jetzt wird es ernst, wir erzeugen unser erstes GIT-Repository: *git init*. Yes!
Wenn man nun per *ls -oha* nachguckt, sollte das Verzeichnis so aussehen:

```
insgesamt 12K
drwxrwxr-x  3 hauke 4,0K Mar 22 13:03 .
drwxrwxr-x 18 hauke 4,0K Mar 22 13:03 ..
drwxrwxr-x  7 hauke 4,0K Mar 22 13:03 .git
```

Das Verzeichnis .git ist ein verstecktes Verzeichnis, dort stehen Informationen für GIT in diversen Dateien.

[1] Manche Shells benötigen den Befehl *rehash*, wenn neue Software hinzugekommen ist

Wir können nun loslegen, Dateien anzulegen. Fangen wir erst einmal einfach an und legen eine Datei LIESMICH an. Ich verwende an dieser Stelle den Editor vi, wenn du aber einen anderen Editor verwenden magst, nutze diesen; es geht nur darum, Text in eine einfache Datei zu schreiben (Officeprogramme scheiden also aus!).
Der Inhalt der Datei soll ganz einfach sein:

```
Projekt Welt
```

Abspeichern nicht vergessen!
Jetzt sieht unser Verzeichnis (*ls -oha*) so aus:

```
insgesamt 16K
drwxrwxr-x  3 hauke 4,0K Mar 22 13:20 .
drwxrwxr-x 18 hauke 4,0K Mar 22 13:03 ..
drwxrwxr-x  7 hauke 4,0K Mar 22 13:06 .git
-rw-rw-r--  1 hauke   14 Mar 22 13:20 LIESMICH
```

Jetzt fragen wir mal GIT, was es davon hält, dass in dem Repository eine Datei neu angelegt wurde. Dazu geben wir *git status* ein. Das Ergebnis sollte in etwa so aussehen:

```
Auf Branch master

Initialer Commit

Unversionierte Dateien:
  (benutzen Sie "git add <Datei>...", um die Aenderungen
   zum Commit vorzumerken)

        LIESMICH

nichts zum Commit vorgemerkt, aber es gibt unversionierte
Dateien
(benutzen Sie "git add" zum Versionieren)
```

GIT gibt uns einige Informationen:
Wir sind auf dem Branch (Zweig) “master”. Dazu - wie gesagt - später mehr.
Unser erstes Einchecken (commit) steht bevor; es ist der initiale Commit.
Git hat festgestellt, dass es Dateien gibt, über die es nichts weiss. Unversionierte Dateien also. Die Datei(en) wird/werden aufgezählt und es gibt - zumindest bei mir - einen hilfreichen Tip, was man machen sollte (*git add ...*).
Also, ran an den Speck und die Datei per *git add LIESMICH* zum Einchecken vormerken. Ein *git status* sollte nun folgendes ergeben:

```
Auf Branch master

Initialer Commit

zum Commit vorgemerkte Aenderungen:
  (benutzen Sie "git rm --cached <Datei>..." zum
   Entfernen aus der Staging-Area)

        new file:   LIESMICH
```

Prima, das sieht schon anders, besser aus!
Jetzt kann ich natürlich weitere Dateien anlegen und per *git add ...* vormerken, oder ich kann diese eine Datei in das Repository übernehmen, den "commit" also durchführen. *git commit LIESMICH* führt den Commit auf die Datei LIESMICH aus. Es sollte sich ein Editorfenster öffnen, denn GIT bietet dir jetzt an, in einem Kommentar kurz zu erläutern, was du gemacht hast. Dieser Text taucht dann in der Historie des Projektes später auf. Beispiel: "LIESMICH angelegt". Kurz und knackig sollte der Kommentar sein, aber auch so geschrieben, dass du später noch weisst, was du seinerzeit gemacht hast.
GIT gibt dir auch hier wieder eine Rückmeldung:

```
[master (Basis-Commit) 19a30b3] LIESMICH angelegt
 1 file changed, 2 insertions(+)
 create mode 100644 LIESMICH
```

In der ersten Zeile wird der Branch angezigt, der Commit-Hash (ist auch an dieser Stelle nicht weiter wichtig, ich erwähne es aber dennoch, der Übersicht halber) und dein Kommentar. In der zweite Zeile wird die Anzahl der veränderten Dateien, sowie hinzugekommene Zeilen (insertions) und (hier noch nicht sichtbar) die Anzahl der entfernten Zeilen. In der dritten Zeile stehen dann noch Information bzgl. des Modus der Datei (create), die Dateirechte und der Dateiname.
Super, damit ist die erste Version des Projektes "Welt" im GIT-Repository angekommen.

Am Beispiel einer weiteren Datei zeige ich dir nun auch erste "Vereinfachungen", denn mal angenommen, du hast auf einem Satz mehrere Dutzend Dateien neu in deinem Projekt, dann wirst du nicht per *git add datei1*, *git add datei2*, usw. jede einzelne Datei vormerken. Die Sache ginge ja dann so später beim commit weiter. Also, eher wird der BER fertig, als dass du mit deinem Projekt weiterkommst!
Als Beispiel gibt es jetzt das Shellskript "welt.sh":

```
#!/bin/sh

echo "Hallo Welt"
```

Jetzt kommt das "add" mal etwas anders: *git add .* Du hast den Punkt hinter "add" bemerkt? Gut! Damit hast du jetzt alle neuen Dateien in diesem Verzeichnis (Unterverzeichnisse, sofern vorhanden mit eingeschlossen) zum einchecken vorgemerkt. *git status* wird dir das anzeigen.
Jetzt das tatsächliche Einchecken in das Repository und da verrate ich dir den nächsten Trick: Den Kommentar direkt angeben, ohne dass ein Editor erst öffnet: *git commit -a -m 'Projekt erweitert, neue Dateien hinzugefuegt'.*
Das erkläre ich mal genauer:
-a bedeutet, dass alle (-a = all) Dateien, die vorher per *git add .*, bzw. per *git add dateiname* vorgemerkt wurden nun in das Repo übernommen werden sollen. Du siehst: Man kann auch erst einmal "sammeln".
-m bedeutet, dass nun ein Kommentar kommt (-m = message). Der Text sollte in Anführungszeichen stehen, sonst macht die Shell Ärger.
Und damit hast du bereits die ersten Schritte gemacht.
Hier noch mal zusammengefasst die Schritte, die im Wesentlichen immer wie-

der wiederholt werden, nachdem man das Projektverzeichnis angelegt und GIT initialisiert hat:

- Datei anlegen
- git add
- git commit
- Datei anlegen
- git add
- git commit
- ...

Oder aber man macht:

- Datei anlegen
- Datei anlegen
- Datei anlegen
- Datei anlegen
- Datei anlegen
- Datei anlegen
- Datei anlegen
- Datei anlegen
- Datei anlegen
- git add .
- git commit -a
- Datei anlegen
- ...

Kapitel 4

GIT konfigurieren

Oftmals sehen die Logs von GIT ziemlich dämllich aus, was meistens daran liegt, dass die Benutzer ihr GIT nicht ordentlich konfiguriert haben. Für eine vernünftige Konfiguration bedarf es des korrekten Namens und der Mailadresse. Dies geschieht per *git config ....* Beispiel: *git config user.name 'Max Mustermann'*, um den Namen festzulegen und *git config user.email 'mmuster@exampel.kom* für die Mailadresse.
Damit sind Name und Mailadresse für dieses eine Projekt festgelegt. Damit das für alle GIT-Repositories gilt, fügt man die Option *–global* hinzu, also *git config –global user.name 'Max Mustermann'* und *git config –global user.email 'mmuster@exampel.kom.*
Man kann natürlich noch eine Menge mehr konfigurieren, aber hier geht es um die Grundlagen.

Kapitel 5

Zweige

Bisher haben wir uns geradlinig auf dem Zeitstrahl bewegt, aber wenn man im Team an einem großen Projekt arbeitet, dann erledigt jeder, bzw. jede entsprechende Unteraufgaben. Um solche Unteraufgaben zu organisieren, nutzt man in Versionskontrollsystemen sog. "Zweige". Leider kann man das nicht direkt mit einem Baum vergleichen, denn zwar gibt es sowohl beim Baum als auch in GIT einen Hauptzweig/Stamm und (mehr oder weniger) viele Zweige, bei einem Baum werden die Zweige aber nicht wieder zum Stamm zurückgeführt. Bei Versionskontrollsystemen jedoch werden Zweige früher oder später wieder in den Hauptzweig überführt.

Am Anfang befindet man sich immer auf dem Hauptzweig, dem sog. "Masterbranch". Du hast das bereits in diversen Meldungen von GIT gesehen.
Das sieht dann so aus:

```
master A--B--C--D--E--F
```

Um nun einen neuen Zweig zu erzeugen, gibt man *git checkout -b <Zweigname>*[1] ein. Angenommen, ich will als Unteraufgabe zum Projekt "Welt" ein "Hallo" programmieren, dann erzeuge ich den Zweig "Hallo" per *git checkout -b hallo*. Wir sind nun im Zweig "hallo". Um das zu überprüfen gibt es *git branch*:

```
* hallo
  master
```

Der kleine Stern in der ersten Spalte zeigt den Zweig an, in dem wir uns aktuell befinden.
Will ich auf den Zwei "master" zurückwechslen, gebe ich *git checkout master* ein. Wichtig: Hier wird kein "-b" verwendet, da es den Zweig "master" bereits gibt!

Bist du also im Zweig "hallo", gehst du wieder wie immer vor: Datei(en) anlegen, vormerken, einchecken.
Das sieht dann so aus:

[1] -b bedeutet, dass der Zweig erzeugt und sofort ausgecheckt wird, d.h. man befindet sich sofort auf diesem Zweig

```
master A--B--C--D--E--F
           \
hallo       G--H--I--J--K
```

Das sollte soweit klar sein, aber irgendwann kommt der Punkt, an dem du den Zweig “hallo” wieder in den Zweig “master” zurückführen willst/musst.
Dazu wechselst du erst einmal in den Zweig “master” zurück. Nun sagst du GIT, dass du den Zweig “hallo” in den Masterbranch “mergen” willst: *git merge hallo*. Es öffnet sich wieder ein Editor, indem (meistens) bereits eine Information vorgegeben ist:

```
Merge branch 'hallo'
```

Hinweis: Es muss(!) ein Kommentar angegeben werden, und sei es nur der vorgegebene Standardkommentar!
Wenn alles gutgegangen ist, sind nun alle Dateien des “hallo”-Zweiges im Masterbranch.
Das sieht dann so aus:

```
master A--B--C--D--E--F---L
           \             /
hallo       G--H--I--J--K
```

In der Softwareentwicklung wird generell mit mehreren Zweigen gearbeitet, um zu verhindern, dass ungetestete und in der Entwicklung befindliche Dateien Einzug in das fertige Projekt nehmen. Es gibt daher einmal natürlich den Zweig “master”, in dem das fertige Produkt liegt, dann den Zweig “develop”, in dem entwickelt wird und von dem weitere Zweige mit weiteren Unteraufgaben abgeleitet werden. Wenn in “develop” alles ordentllich läuft, dann wird dieser in den Masterbranch gemergt und das Produkt wird veröffentlicht. Dadurch ergibt sich mit der Zeit eine durchaus recht breite Struktur, die aber dem Team hilft, dass man sich nicht in die Quere kommt und letztendlich auch den Überblick behält.

Kapitel 6

Ein erstes Fazit

Du solltest nun in der Lage sein, lokal auf deinem Rechner GIT-Repositories anzulegen, Dateien hinzuzufügen und diese auch in das Repo zu laden. Ausserdem solltest du dir auch über Zweige klar sein und diese verwenden können, denn mit Hilfe von Zweigen wird eine koordinierte Teamarbeit in einem Projekt erst möglich, aber auch wenn du alleine an einem Projekt arbeitest, solltest du dir angewöhnen, für jede Teilaufgabe einen eigenen Zweig anzulegen.

Kapitel 7

Dateien löschen und umbenennen

Dateien anlegen kannst du jetzt schon flüssig, was aber ist, wenn du eine versionierte Datei nicht mehr brauchst, oder wenn du dich beim Namen vertippt hast?
Keine Panik, auch das geht mit GIT.
Um eine Datei zu löschen, die im GIT-Repository ist, verwendest du den Removebefehl: *git rm <dateiname>*. Danach führst du ein *git commit -m 'Kommentar'* aus und die Datei ist von jetzt an nicht mehr vorhanden.
Umbenennen geht ebenfalls sehr einfach: *git mv <alter name> <neuer name>*. Auch danach wieder ein *git commit -m 'Kommentar'* und die Datei hat von nun an einen neuen Namen.
Moment mal, was soll das heissen "von nun an"?
Ganz einfach: Da man mit GIT auch ältere Versionen des Repositories auschecken kann, hat in früheren Versionen die Datei natürlich noch ihren alten Namen, bzw. ist im Repo noch vorhanden! Wie man frühere Versionen auscheckt, erkläre ich aber sehr viel später.

Kapitel 8

GIT-Server

8.1 Einstieg

So, jetzt kommt der sog. “heisse Scheiss”! GIT in Verwendung mit einem GIT-Server!
Bisher hast du deine Repositories lokal auf deinem Rechner gehabt und das ist am Anfang auch gut so, aber stelle dir vor, deine Festplatte zerlegt sich und dein echt tolles Projekt wird damit geschreddert. Dann nützt dir auch GIT nichts mehr. Gut, du hättest regelmäßig eine Sicherung deines Repos auf DVD, CD, USB-Stick oder wo auch immer anlegen könnenen, aber mal ehrlich, das ist nicht Sinn und Zweck von GIT. Ausserdem funktioniert so die Teamarbeit an einem Projekt nicht. Also benötigt man externen Speicherplatz. Auf einem GIT-Server.
Als Plattform kannst du öffentliche Server nehmen. GitHub sei hier erwähnt.
Du kannst dir einen GIT-Server auch auf einem kleinen Raspberry Pi in deinem Heimnetzwerk einrichten. Oder du mietest dir einen eigenen Rootserver und packst dort deinen GIT-Server drauf. Alles kein Problem!

8.2 GitHub

GitHub ist eine - für nichtkommerzielle Projekte - kostenlose Plattform, die GIT anbietet. Man kann natürlich Geld zahlen, dann bekommt man etwas mehr an “Features” auf der Plattform, aber für den Hausgebrauch reicht die kostenlose Variante.
Du meldest dich bei GitHub an und hinterlegst dort deinen öffentlichen SSH-Schlüssel[1].
Du kannst dort deine Projekte komplett verwalten, die Seite ist sehr übersichtlich aufgebaut und es gibt auch eine Hilfefunktion.

8.3 GIT-Server im Heimnetzwerk

Für einen GIT-Server im eigenen Heimnetzwerk eignet sich ein Raspberry Pi wunderbar. Der sollte natürlich über LAN (und nicht via WLAN) mit dem Netz

[1] Wie man SSH-Schlüssel erstellt, mögest du bitte im WWW selbständig suchen

verbunden sein, damit die Datenübertragung möglichst hastig erfolgen kann. Eine FritzBox zum Beispiel hat Gigabitinterfaces, da kann man schon mal gut was über die Leitung jagen (wobei des Raspberries Interface langsamer ist).
Als GIT-Serversoftware empfehle ich an dieser Stelle gogs[2].
Wenn du dich damit beschäftigen willst, solltest du aber über gute Linuxkenntnisse verfügen.
Mit gogs hast du eine Weboberfläche, ähnlich wie bei GitHub, unter der du deine Projekte recht einfach verwalten kannst. Du hast auch die Möglichkeit, private Projekte dort anzulegen und mit Hilfe der Benutzerverwaltung kannst du auch Einfluss darauf nehmen, was deineTeammitgleider alles so dürfen.

8.3.1 gogs

Gogs ist ein kleines, aber feines Projekt, das ein Webfrontend für einen git-Server bereitstellt. Es ist in der Programmiersprache go geschrieben und benötigt nur minimale Resourcen.

8.3.1.1 Vorbereiten der Datenbank

Bevor du dir gogs herunterlädst und installierst, solltest du das Datenbanksystem auswählen, das gogs später benutzen soll. Du kannst zwischen MariaDB/MySQL und PostgreSQL wählen. Gut, es giobt noch SQLite, MSSQL oder TiDB nutzen, aber die ersten beiden will man nicht wirklich und TiDB ist wohl noch zu speziell. Achte darauf, dass MySQL/MariaDB auch tatsächlich auf dem Netzwerkinterface horcht, das du bei der Installation angibst!

8.3.1.1.1 PostgreSQL Am einfachsten loggst du dich per “sudo su - postgres” als Benutzer postgres in deiner Shell auf deinem GIT-Server ein, dann erzeugst du per “createuser -P -d gogs” einen Benutzer für die Datenbank “gogs”. Im nächsten Schritt loggst du dich auf der Shell per “psql -h localhost template1 gogs” in das Datenbanksystem ein. Hier reicht nun ein “CREATE DATABASE gogs;”, um die Datenbank für gogs anzulegen.

8.3.1.1.2 MariaDB/MySQL Du loggst dich mit dem administrativen Benutzer in MariaDB/MySQL, den du bei der Installation eingerichtet hast, in das Datenbanksystem ein. Das ist oft der Datenbankbenutzer “root” (nicht zu verwechseln mit dem Systembenutzer “root”). Also “mysql -p -u root”. Nun erfolgt ein “GRANT ALL PRIVILEGES ON gogs.* to ’gogs’@’localhost’ IDENTIFIED BY ’geheimes passwort’;”. Dann loggst du dich aus, loggst dich per “mysql -p -u gogs” wieder ein und führst ein “CREATE DATABASE gogs CHARACTER SET utf8mb4;” aus, um die Datenbank für gogs anzulegen.
Hinweis: Sollte bei Benutzung von MariaDB während der Installation im Webbrowser die Fehlermeldung “Datenbankeinstellungen sind nicht korrekt: Error 1071: Specified key was too long; max key length is 767 bytes”, dann folgende Befehle in der MariaDB-Kommandozeile ausführen:

```
set global innodb_large_prefix=on;
set global innodb_file_format=Barracuda;
```

[2] https://gogs.io/

8.3.1.2 Installation

Bevor du dir gleich die Software runterlädst, lege bitte einen Benutzer "git" auf deinem Server an, sofern der Benutzer noch nicht existiert.
Von https://gogs.io/docs/installation/install_from_binary lädt man sich die passende Version für seinen Server runter und entpackt diese. Das entstandene Verzeichnis wird dann in das Benutzerverzeichnis des Benutzers "git" verschoben und diesem zu eigen gemacht (chown).
Im Verzeichnis gogs/scripts findest du Beispiele, um gogs auf deinem System bei Systemstart automatisch zu starten. Das geht zum Beispiel via init oder systemd.
Natürlich solltest du die Dateien entsprechend für dein System anpassen, sonst stimmen die Pfade unter Umständen nicht.

8.3.1.3 Konfigurationsdatei

Um später noch ein wenig Feintuning vorzunehmen, kann man im Verzeichnis "custom/conf" die Datei "app.ini" ändern, die bei der Installation automatisch angelegt wird.
Ein Beispiel für eine app.ini:

```
APP_NAME = Gogs
RUN_USER = git
RUN_MODE = prod

[repository]
ROOT = /home/git/repositories

[database]
DB_TYPE  = postgres
HOST     = 127.0.0.1:5432
NAME     = gogs
USER     = gogs
PASSWD   = GEHEIM
SSL_MODE = disable
PATH     = data/gogs.db

[server]
DOMAIN           = git.hauke-zuehl.de
HTTP_PORT        = 3000
ROOT_URL         = https://git.hauke-zuehl.de/
DISABLE_SSH      = false
SSH_PORT         = 22
START_SSH_SERVER = false
OFFLINE_MODE     = false

[mailer]
ENABLED = false

[service]
REGISTER_EMAIL_CONFIRM = true
```

```
ENABLE_NOTIFY_MAIL     = true
DISABLE_REGISTRATION   = true
ENABLE_CAPTCHA         = true
REQUIRE_SIGNIN_VIEW    = true

[picture]
DISABLE_GRAVATAR        = false
ENABLE_FEDERATED_AVATAR = true

[session]
PROVIDER = file

[log]
MODE      = file
LEVEL     = Info
ROOT_PATH = /home/git/gogs/log

[security]
INSTALL_LOCK = true
```

Bitte daran denken, dass diese Datei als Eigentümer den Benutzer “git” haben muss! Wenn du aber bei der Installation oben alles korrekt eingetragen hast, sollte es keine Probleme geben.
In diesem Beispiel habe ich übrigens PostgreSQL als Datenbanksystem verwendet, du kannst aber gerne MariaDB oder MySQL verwenden.

8.3.1.4 Der erste Start

Du startest gogs zunächst über die Kommandozeil per “./gogs web” und gibst dann in deinem Webbrowser die URL “http://localhost:3000” ein.
Du wirst nun durch die Konfiguration der Datenbank und von gogs geführt. Bei der Domain solltest du den Server- oder einen ordentlichen Domainnamen eingeben. Hast du also zum Beispiel einen kleinen RaspberryPi zu Hause, auf dem gogs laufen soll, und dieser heisst “pi.deinzuhause.net” dann gibst du eben diesen Namen bei der Domain und der Application URL ein.
Den Datenbankbenutzer für gogs und die entsprechende Datenbank solltest du nun auch anlegen, sonst kracht es bei der Einrichtung der Tabellen.
Hast du soweit alles eingetragen, kannst du dich anmelden und deine Projekte mit gogs verwalten.
Der erste Benutzer, der sich registriert bei gogs, wird übrigens als Administrator angelegt, also Obacht! Von nun kannst du gogs in aller Ruhe erforschen und eigene Repositores anlegen, Organiosationen und Teams erzeugen, usw.!

8.4 GIT from scratch

Ich habe keine bessere Überschrift gefunden, aber wenn du keine Lust auf GitHub oder gogs oder wie auch immer das coole (Web-)Frontend heisst, dann reicht auch...git.
Dann legst du als Benutzer “root” einen Benutzer “git” an. Hast du das erledigt,

wirst du zum Benutzer “git” und legst ein Verzeichnis “.ssh” im Benutzerverzeichnis von “git” an. In das Verzeichnis “.ssh” kommen in die Datei “authorized_keys” die öffentlichen SSH-Schlüssel der Benutzer, die git benutzen dürfen. Warum wird das mit den Schlüsseln gemacht?
Ganz einfach: Ansonsten müsstest du jedem Teammitglied das Passwort des Benutzers “git” verraten und das willst du wirklich nicht!
Also:

```
sudo adduser git
su git
mkdir .ssh
chmod 700 .ssh
```

Jetzt kannst du - als Benutzer “git” anfangen, Repositories anzulegen:

```
mkdir Welt.git
cd Welt.git
git --bare init
```

Du siehst hier wieder den “init”-Befehl, diesmal jedoch mit der Option “–bare”. Diese Option sorgt dafür, dass kein sog. “Arbeitsverzeichnis” angelegt wird, d.h. es fehlt das .git-Verzeichnis, dafür sind dort andere Dateien und Verzeichnisse erzeugt werden, die zur Verwaltung eines GIT-Repositories nötig sind (guck einfach mal per *ls -oha* nach). Okay, du hast keine Lust? So sieht das Verzeichnis dann aus:

```
drwxrwxr-x  7 git 4,0K Mar 22 18:11 .
drwxrwxr-x 19 git 4,0K Mar 22 18:11 ..
drwxrwxr-x  2 git 4,0K Mar 22 18:11 branches
-rw-rw-r--  1 git   66 Mar 22 18:11 config
-rw-rw-r--  1 git   73 Mar 22 18:11 description
-rw-rw-r--  1 git   23 Mar 22 18:11 HEAD
drwxrwxr-x  2 git 4,0K Mar 22 18:11 hooks
drwxrwxr-x  2 git 4,0K Mar 22 18:11 info
drwxrwxr-x  4 git 4,0K Mar 22 18:11 objects
drwxrwxr-x  4 git 4,0K Mar 22 18:11 refs
```

Falls du es dir noch nicht gedacht hast: Du kannst das lokal auf deinem Rechner machen! Damit wird dein Rechner zum GIT-Server! Und was auf deinem Rechner funktioniert, klappt auch auf einem Raspberry Pi, der in deinem Keller oder auf dem Dachboden rumliegt oder auf dem dicken Server, der irgendwo in einem Rechenzentrum in Deutschland steht.

Kapitel 9

Projekt “Welt” auf den Server bringen

Nachdem du also einen GIT-Server irgendwie ans Laufen gebracht hast, oder du dich bei GitHub angemeldet hast, deinen öffentlichen SSH-Schlüssel auf den Server geladen hast, soll das “Welt”-Projekt nun auf den entfernten Server gebracht werden.

Ob dein GIT-Server nun auf deinem Raspberry oder auf deinem lokalen Server läuft, der Unterschied liegt in der Benennung des Rechners. Ich erkläre dir die folgenden Schritte unter der Annahme, dass du den GIT-Server auf deinem Rechner installiert hast. Die Transferleistung hin zum Raspberry Pi oder einem Rootserver überlasse ich dir.
Um dein Projekt auf den Server hochzuladen, musst du dich im Repoverzeichnis befinden, wo wir uns am Anfang dieses Papiers aufgehalten haben. Also *cd /git/Welt*. Zu Beginn wollen wir den Masterbranch hochladen, also zur Sicherheit ein *git checkout master* machen.
Achtung, jetzt geht es los:

```
git remote add origin git@localhost:Welt.git
git push origin master
```

Damit hast du deinen Masterbranch auf den Masterbranch des GIT-Servers geladen, wobei der Server dafür sorgt, dass deine Änderungen mit dem bereits bestehenden Branch “gemergt” werden. Bei einem krachneuen Branch ist das natürlich nicht nötig, aber später einmal, wenn du ein ganzes Team an deinem Projekt mitwerkeln lässt, dann ist das wichtig.

Also, mal die Erklärung:
Mit der ersten Zeile hast du deinem Repository gesagt, dass unter dem Namen “origin” ein entferntes (remote) Repo existiert. Und zwar beim Benutzer “git” auf dem Rechner “localhost” im Repository “Welt.git”.
Mit der zweite Zeile hast du dann den aktuellen Branch hochgeladen (push) und zwar zu dem entfernten Repository “origin” in dessen Branch “master”.
Die Bezeichnung “origin” für ein entferntes Repository ist Standard. Du hättest es auch “Pizza” oder “Koeln” nennen können, Namen sind Schall und Rauch.

Dein “push” wäre dann *git push Pizza master*, bzw. *git push Koeln master*. Es ist nur ein Name für die Verbindungsdaten zum Server (git@localhost:Welt.git).

Jetzt hat sich deine Arbeitsweise etwas erweitert:

- Datei anlegen
- Datei anlegen
- Datei anlegen
- Datei anlegen
- Datei anlegen
- Datei anlegen
- git add .
- git commit -a
- Datei anlegen
- Datei anlegen
- Datei anlegen
- Datei anlegen
- git add .
- git commit -a
- git push origin master
- ...

(Gesetzt den Fall, du bist im Masterbranch)

Kapitel 10

Arbeiten im Team

Du hast bisher alleine mit GIT gearbeitet, hast aber das Projekt, bzw. die Projekte mit einem GIT-Server verwaltet.
Dann ist der Schritt hin zur Teamarbeit für dich recht leicht, denn du weisst ja, dass du mit *git push ...* Veränderungen hochlädst.
Da aber deine Kolleginnen und Kollegen auch fleissig waren, solltest du deren Änderungen mindestestens einmal am Tag herunterladen und deine Arbeitskopie damit aktuell halten.
Das gilt besonders für die Zweige, die regelmäßigen Änderungen unterworfen sind. Dies sind meistens

- master
- develop
- testing

Um deine Arbeitskopie zu aktualisieren, wechselst du in den entsprechenden Zweig (zum Beispiel "develop") und führst dort *git pull* aus.
Damit werden die Änderungen vom Server heruntergeladen und in deinen Zweig eingepflegt.

Ein Beispiel aus dem realen Leben:
Wenn man in einer Software einen Fehler findet, wird erst einmal ein Bericht geschrieben und dieser Bericht bekommt eine sog. "Ticketnummer". Zum Beispiel 4711.
Du sollst nun Ticket 4711 bearbeiten. Es wird immer (meistens) aus dem Zweig "develop" ein neuer Zweig erstellt. Das heisst, du wechselst per *git checkout develop* in den Entwicklungszweig und führst hier *git pull* aus. Damit ist dein Zweig aktuell und du kannst mit *git checkout -b bug-4711* den Zweig erzeugen, um den Fehler zu beheben.
Bist du irgendwann fertig damit (und hast immer wieder *git pull origin bug-4711* gemacht, kannst du deinen Zweig wieder in den Developbranch "zurückmergen", bzw "zurückmergen" lassen. "Lassen" deshalb, weil in der Regel ein anderer Programmierer erst deinen Code testet und für den Merge freigibt, aber das ist abhängig von den Regeln der jeweiligen IT-Abteilung.

Kapitel 11

Ältere Version auschecken

Es wird selten gebraucht, aber manchmal denkt man: “Verdammt, an dem Punkt da vor ein paar Tagen/Wochen/Monaten, da lief das System besser!”
Besonders, wenn man mal “was ausprobieren” will, ist es ganz nett, wenn man eine “uralte” Version des Projektes auschecken kann.
Geht recht einfach!
Na gut, man muss ein wenig arbeiten, denn zuerst muss man den richtigen Commit wiederfinden. Um das Log eines GIT-Projektes anzuzeigen, verwendet man *git log*.
Das sieht zum Beispiel so aus:

```
commit 43632ef9d9ed259a33d030d2e71549bba752e97b
Author: Max Mustermann <hzuehl@phone-talk.net>
Date:   Tue Mar 27 18:10:22 2018 +0200

    blah => blubb

commit 90845d50545e2bb7069622dbe0c645241f25e9d2
Merge: 19a30b3 201d71f
Author: Max Mustermann <hzuehl@phone-talk.net>
Date:   Thu Mar 22 15:08:02 2018 +0100

    Merge branch 'hallo'

commit 201d71f352307d88b98aa4d1c5d5892b468948e7
Author: Max Mustermann <hzuehl@phone-talk.net>
Date:   Thu Mar 22 15:07:54 2018 +0100

    Blah

commit 19a30b330ab250a6d3ab3f0a9ecf1c6d9b2d9fd5
Author: Hauke Zühl <hzuehl@phone-talk.net>
Date:   Thu Mar 22 13:40:59 2018 +0100

    LIESMICH angelegt
```

Zur Info: Ich bin zur Zeit im Masterbranch.

```
-rw-rw-r-- 1 hauke  0 Mar 22 15:08 blubb
-rw-rw-r-- 1 hauke 14 Mar 22 13:20 LIESMICH
```

Jetzt will ich die Version vom 22.3, 13:40:59 auschecken. Das ist der Commit 19a30b330ab250a6d3ab3f0a9ecf1c6d9b2d9fd5. Dann mal los: *git checkout 19a30b330ab250a6d3ab3f0a9ecf1c6d9b2d9fd5*
Das ist verdammt lang, oder?
An dieser Stelle ein Hinweis: Es reichen die eersten 7 Zeichen des Hashes des Commits, also ein *git checkout 19a30b3* hätte es auch getan!
Wie dem auch sein, das Verzeichnis sieht jetzt so aus:

```
-rw-rw-r-- 1 hauke 14 Mar 22 13:20 LIESMICH
```

Es fehlt also eine Datei ("blubb")! Logisch, die gab es zu diesem Zeitpunkt noch nicht.
Aber Vorsicht: Es muss nicht sein, dass du dich dann noch in dem gewünschten Zweig befindest; du kannst an jeder Stelle jede Änderung auschecken und von da an weitermachen!
Tip: Das solltest du ein wenig üben und noch ein Tip: An dieser Stelle sind grafische GIT-Clients sehr nützlich, um dir eine gute Übersicht über dein Projekt zu geben.
Wenn du aber tapfer bist und auf der Konsole bleiben willst, bitte schön: *git log –graph* zaubert ein schönes Log auf den Bildschirm. Und wenn du die Kurzform der Commits haben willst, dann bringt dich *git log –abbrev-commit –graph* ans Ziel.

Kapitel 12

Ignorieren von Dateien

Ich starte mal mit einem Beispiel, um zu zeigen, was das Problem ist:
Wir gehen jetzt davon aus, dass wir ein C++-Projekt compilieren, d.h. aus dem Quellcode ein Programm "basteln" wollen. Wenn du dich damit nicht auskennst, ist das nicht schlimm, es geht um Dateien und nicht um irgendwelche freakigen Sachen.
Zuerst der Verzeichnisbaum eines "frischen" Repos:

```
hauke@apollo:~/git/Lara$ tree .
.
|-- CMakeLists.txt
|-- Doxyfile
|-- README.md
|-- sql
|   |-- address.sql
|   └-- customer.sql
└-- src
    |-- addons
    |   |-- Address
    |   |   |-- AddressAddon.cc
    |   |   |-- AddressAddon.h
    |   |   |-- AddressMainWindow.cc
    |   |   |-- AddressMainWindow.h
    |   |   └-- CMakeLists.txt
    |   └-- CMakeLists.txt
    |-- CMakeLists.txt
    |-- core
    |   |-- Addon.h
    |   |-- Base.cc
    |   |-- Base.h
    |   |-- CMakeLists.txt
    |   |-- Config.cc
    |   |-- Config.h
    |   |-- Convert.cc
    |   |-- Convert.h
    |   |-- Database.cc
```

```
|   |-- Database.h
|   |-- Files.cc
|   |-- Files.h
|   |-- IDatabase.cc
|   |-- IDatabase.h
|   |-- Lara.cc
|   |-- Lara.h
|   |-- Loader.cc
|   |-- Loader.h
|   |-- Map.cc
|   |-- Map.h
|   |-- models
|   |   |-- Address.h
|   |   `-- Customer.h
|   `-- UserData.h
|-- GUI
|   |-- CMakeLists.txt
|   |-- MainWindow.cc
|   `-- MainWindow.h
`-- main.cc

7 directories, 39 files
```

Ein “git status” sähe so aus:

```
hauke@apollo:~/git/Lara$ git status
Auf Branch develop
Ihr Branch ist auf dem selben Stand wie 'origin/develop'.
nichts zu committen, Arbeitsverzeichnis unverändert
hauke@apollo:~/git/Lara$
```

Um dieses Projekt zu compilieren, muss ich folgende Schritte durchführen:

- mkdir build
- cd build
- cmake ../
- make

Das heisst, ich erzeuge ein neues Verzeichnis namens “build”, wechsele in das dortige Verzeichnis, führe ein wenig Magie aus und am Ende fällt das fertige Programm im Unterverzeichnis “src” raus.
Wenn ich das alles gemacht habe, sieht der Verzeichnisbaum so aus:

```
.
|-- build
|   |-- CMakeCache.txt
|   |-- CMakeFiles
|   |   |-- 2.8.12.2
|   |   |   |-- CMakeCCompiler.cmake
|   |   |   |-- CMakeCXXCompiler.cmake
```

```
|   |   |   |-- CMakeDetermineCompilerABI_C.bin
|   |   |   |-- CMakeDetermineCompilerABI_CXX.bin
|   |   |   |-- CMakeSystem.cmake
|   |   |   |-- CompilerIdC
|   |   |   |   |-- a.out
|   |   |   |   `-- CMakeCCompilerId.c
|   |   |   `-- CompilerIdCXX
|   |   |       |-- a.out
|   |   |       `-- CMakeCXXCompilerId.cpp
|   |   |-- cmake.check_cache
|   |   |-- CMakeDirectoryInformation.cmake
|   |   |-- CMakeOutput.log
|   |   |-- CMakeTmp
|   |   |-- Makefile2
|   |   |-- Makefile.cmake
|   |   |-- progress.marks
|   |   `-- TargetDirectories.txt
|   |-- cmake_install.cmake
|   |-- Makefile
|   `-- src
|       |-- addons
|       |   |-- Address
|       |   |   |-- address.so
|       |   |   |-- CMakeFiles
|       |   |   |   |-- address.dir
|       |   |   |   |   |-- AddressAddon.cc.o
|       |   |   |   |   |-- AddressMainWindow.cc.o
|       |   |   |   |   |-- build.make
|       |   |   |   |   |-- cmake_clean.cmake
|       |   |   |   |   |-- CXX.includecache
|       |   |   |   |   |-- DependInfo.cmake
|       |   |   |   |   |-- depend.internal
|       |   |   |   |   |-- depend.make
|       |   |   |   |   |-- flags.make
|       |   |   |   |   |-- link.txt
|       |   |   |   |   `-- progress.make
|       |   |   |   |-- CMakeDirectoryInformation.cmake
|       |   |   |   `-- progress.marks
|       |   |   |-- cmake_install.cmake
|       |   |   `-- Makefile
|       |   |-- CMakeFiles
|       |   |   |-- CMakeDirectoryInformation.cmake
|       |   |   `-- progress.marks
|       |   |-- cmake_install.cmake
|       |   `-- Makefile
|       |-- CMakeFiles
|       |   |-- CMakeDirectoryInformation.cmake
|       |   |-- lara.dir
|       |   |   |-- build.make
|       |   |   |-- cmake_clean.cmake
```

```
|       |   |   |-- core
|       |   |   |   |-- Base.cc.o
|       |   |   |   |-- Config.cc.o
|       |   |   |   |-- Convert.cc.o
|       |   |   |   |-- Database.cc.o
|       |   |   |   |-- Files.cc.o
|       |   |   |   |-- IDatabase.cc.o
|       |   |   |   |-- Lara.cc.o
|       |   |   |   |-- Loader.cc.o
|       |   |   |   `-- Map.cc.o
|       |   |   |-- CXX.includecache
|       |   |   |-- DependInfo.cmake
|       |   |   |-- depend.internal
|       |   |   |-- depend.make
|       |   |   |-- flags.make
|       |   |   |-- GUI
|       |   |   |   `-- MainWindow.cc.o
|       |   |   |-- link.txt
|       |   |   |-- main.cc.o
|       |   |   `-- progress.make
|       |   `-- progress.marks
|       |-- cmake_install.cmake
|       |-- lara
|       `-- Makefile
|-- CMakeLists.txt
|-- Doxyfile
|-- README.md
|-- sql
|   |-- address.sql
|   `-- customer.sql
`-- src
    |-- addons
    |   |-- Address
    |   |   |-- AddressAddon.cc
    |   |   |-- AddressAddon.h
    |   |   |-- AddressMainWindow.cc
    |   |   |-- AddressMainWindow.h
    |   |   `-- CMakeLists.txt
    |   `-- CMakeLists.txt
    |-- CMakeLists.txt
    |-- core
    |   |-- Addon.h
    |   |-- Base.cc
    |   |-- Base.h
    |   |-- CMakeLists.txt
    |   |-- Config.cc
    |   |-- Config.h
    |   |-- Convert.cc
    |   |-- Convert.h
    |   |-- Database.cc
```

```
|   |-- Database.h
|   |-- Files.cc
|   |-- Files.h
|   |-- IDatabase.cc
|   |-- IDatabase.h
|   |-- Lara.cc
|   |-- Lara.h
|   |-- Loader.cc
|   |-- Loader.h
|   |-- Map.cc
|   |-- Map.h
|   |-- models
|   |   |-- Address.h
|   |   `-- Customer.h
|   `-- UserData.h
|-- GUI
|   |-- CMakeLists.txt
|   |-- MainWindow.cc
|   `-- MainWindow.h
`-- main.cc

23 directories, 103 files
```

Du siehst den Unterschied!

Git würde jetzt also alle “neuen” Datein unterhalb von build finden und natürlich daraus schliessen, dass man diese auch ins Repo aufnehmen will:

```
Auf Branch develop
Ihr Branch ist auf dem selben Stand wie 'origin/develop'.
Unversionierte Dateien:
  (benutzen Sie "git add <Datei>...", um die Änderungen zum Commit vorzumerken)

build/

keine Änderungen zum Commit vorgemerkt (benutzen Sie "git add" und/oder "git commit -a")
```

Nein! Will man nicht!
Man hätte jetzt den ganzen unnützen “Müll” für eine bestimmte Plattform und das ist nicht Sinn eines Quellcode Repositories.
Ergo müssen wir dafür sorgen, dass git nicht nervt, wenn wir auf unserem Rechner das Programm bauen wollen. Dazu soll git also das gesamte Verzeichnis “build”, mit allen Dateien und Unterverzeichnissen ignorieren.
Dazu erstellen wir im Hauptverzeichnis unseres Repos die Datei “.gitignore” (man beachte den Punkt vor dem Dateinamen). Die sieht dann so aus:

```
build/
```

Wenn wir jetzt “git status” machen, sieht das so aus:

```
Auf Branch develop
Ihr Branch ist auf dem selben Stand wie 'origin/develop'.
nichts zu committen, Arbeitsverzeichnis unverändert
```

Cool! Genau das, was wir haben wollen! Das Verzeichnis “build” wird von git ignoriert.

Ein weiteres Beispiel:
Unter MacOS wird gerne die Datei “.DS_Store” angelegt. Da diese Datei für Nutzer anderer Systeme uninteressant, ja sogar nervig ist, sollte man also, wenn Maccies mit im Team sind, in die .gitignore die Datei aufnehmen. Dann sieht also die .gitignore für unser C++-Projekt so aus:

```
build/
.DS_Store
```

Je nach Projekt, Programmiersprache, verwendetem Editor, oder verwendeter IDE gibt es noch weitere Dateien, die für andere uninteressant oder unwichtig sind. Diese können dann nach und nach in die .gitignore aufgenommen werden, wobei natürlich auch Wildcards verwendet werden können.

Kapitel 13

GIT in IDEs

Die meisten IDEs bringen inzwischen Unterstützung für GIT mit.

13.1 Geany

Geany ist eine kleine, erweiterbare IDE, die kostenlos für diverse Plattformen angeboten wird.
Mit Hilfe eines Plugins kann die Verbindung zu GIT installiert werden. Nach der Aktivierung des Plugins steht dann unter dem Menü "Werkzeuge" der Eintrag "Versionskontrolle" zur Verfügung. Hier kann dann GIT für die zur Zeit bearbeitete Datei oder für das gesamte Projektverzeichnis verwendet werden.

Tip:
Geany legt individuelle Projektdateien in einem eigenen Verzeichnis an. Dies sollte, wenn möglich, nicht im Verzeichnis des GIT-Repos liegen, da es sonst Probleme mit anderen Teammitgliedern geben kann!

13.2 NetBeans

NetBeans ist eine recht verbreitete IDE, die kostenlos für diverse Plattformen angeboten wird.
Die Verbindung von NetBeans zu GIT kann über ein Plugin ggf. nachinstalliert werden.
Startet man NetBeans, hat man unter Team->Git die Möglichkeit, GIT-Repos zu verwenden.
Als Beispiel verwende ich nun das Repository unter https://github.com/hauke68/LibTgBotPP, das jedoch das C-/C++-Plugin voraussetzt.
Um dies anonym auszuchecken und in ein NetzBeans-Projekt zu packen, geht man wie folgt vor:
Unter Team->Git->Clone wird die obige URL eingetragen. Benutzer und Passwort bleiben leer. Im nächsten Schritt kann man die Zweige auswählen, die ausgecheckt werden sollen. Ich empfehle, erst einmal alle auszuchecken. Im dritten Schritt lassen wir alles so, wie es ist.

Ist alles ordnugsgemäß ausgecheckt, sollten wir im Git Repository Browser bereits das GIT-Repo sehen. Nun fehlt noch das NetBeans-Projekt.
Dazu auf File->New Project klicken und ein neues C-/C++-Projekt anlegen. Dabei darauf achten, dass im Fenster "Projects" "C/C++ Project with existing sources" ausgewählt ist. Im zweiten Schritt wählen wir das Verzeichnis aus, in dem die Quellen des Repos sind. Das ist natürlich das vorhin erzeugte Verzeichnis vom GIT-Repo.
Da das hier nur ein Beispiel ist, wählen wir als Configuration Mode "custom" aus. Jetzt nur noch auf "Next" klicken, bis nur noch "Finish" möglich ist. Voila, das NetBeans-Projekt existiert und man kann auch GIT als Versionskontrollsystem verwenden.
Wenn du nun eine Datei lädst, änderst und speicherst, kannst du unter "Team" sehen, dass es mehr Auswahlmöglichkeiten in Bezug auf GIT gibt.
Spiele hier einfach mal ein wenig rum. Da du das Repo anonym ausgecheckt hast, kannst du nichts kaputt machen. Etwas anderes wäre es natürlich, wenn du unser Einstiegsprojekt "Welt" mit Hilfe von NetBeans bearbeiten willst. Ich weiß aber nicht, in welche Kategorie dieses "Projekt" fällt.

Tip:
Da NetBeans im Verzeichnis des Quellcodes gerne das Verzeichnis "nbproject" anlegt, sollte dies in die .gitignore aufgenommen werden, das es sonst zu Konflikten mit dem nbproject anderer Teammitglieder kommen kann. Noch besser wäre es, nbproject ganz woanders hin auszulagern (Zum Beispiel in das eigene Benutzerverzeichnis).

Kapitel 14

Zum Ende noch ein paar Ideen und Worte

Zum Ende hin noch ein paar Anregungen bzgl. der Einsatzzwecke von GIT.
Man kann GIT alleine nutzen, kein Problem. Das fängt mit den eigenen Skripten und Programmen an, geht über Korrespondenz bis hin zu Konfigurationsdateien des Rechners (bzw. der Rechner). Nimm dir zum Beispiel einen Raspberry Pi als GIT- und Puppetserver, dann kannst du deine Linuxbüchsen ganz einfach per Puppet konfigurieren.
Oder du willst dir nicht gleich einen Cloudserver einrichten, dann kannst du deine persönlichen Dateien mit Hilfe von GIT verwalten.

Ich hoffe, ich konnte dir einen kleinen Einblick in die Arbeitsweise von GIT vermitteln. Es ist ein wenig Übung nötig, aber mit der Zeit hast du die Grundkommandos verinnerlicht und solltest keine Probleme mehr haben.
Tip: Committe oft! Damit ist nicht nur das “commit” gemeint, sondern durchaus auch das “push”. “push” sollte aber definitiv immer kurz vor Feierabend gemacht werden.
Mit dieser Einstellung sorgst du dafür, dass man deine Änderungen bessser nachvollziehen kann; es ist einfacher, zwei oder drei Zeilen später nachzuvollziehen, als wenn du 234 Zeilen in 20 Dateien änderst und man diese Änderungen später nachvollziehen will.
Viel Spaß mit GIT.