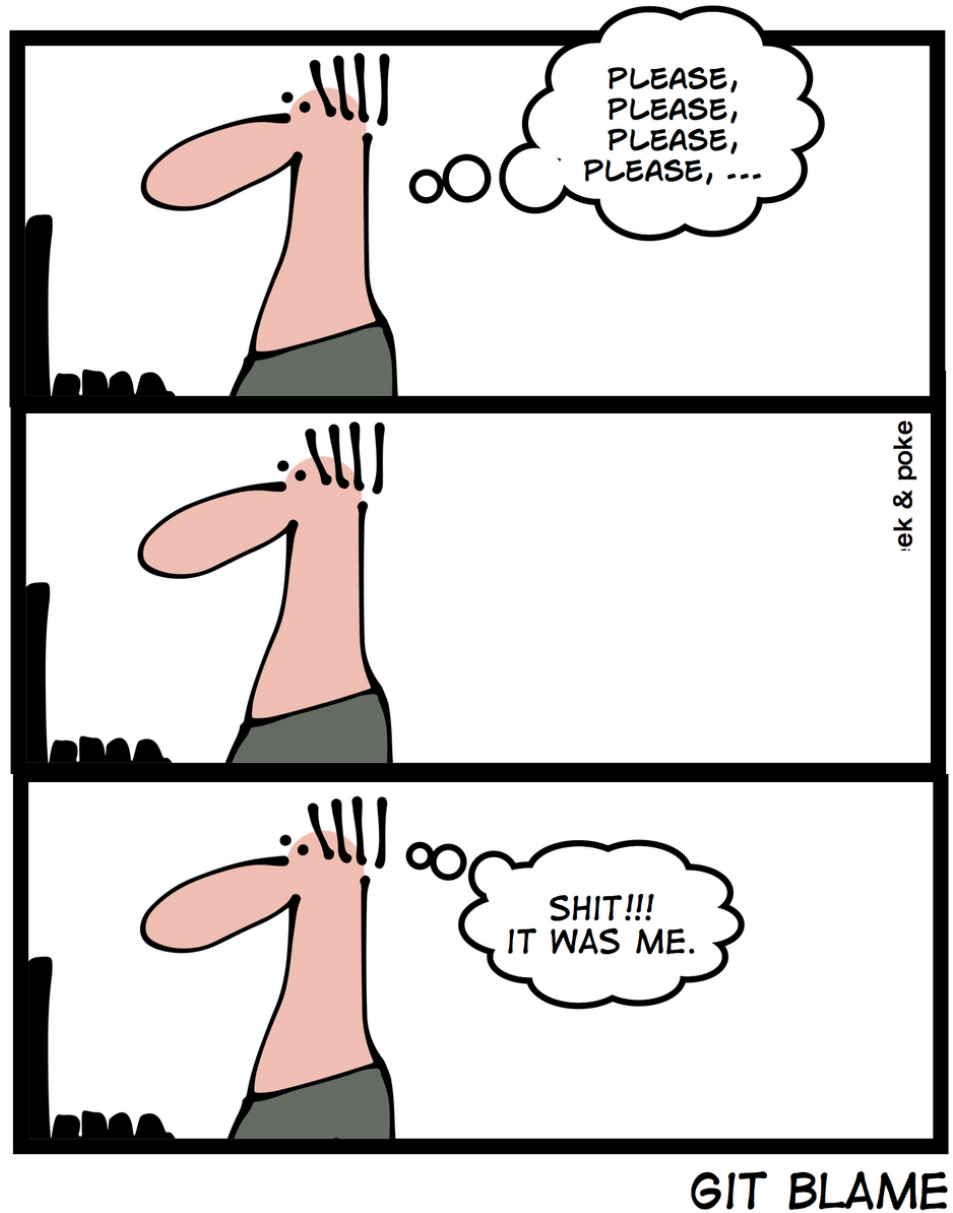
Versionsverwaltung

Vorlesungsinhalt

- Versionsverwaltung
- git
- GitLab
- Git in IntelliJ
- TortoiseGit
- Aufgaben



Versionsverwaltung

Wozu Versionsverwaltung?

1. Statt Dateinamen à la `xyz01` oder `xyzV1vom010416`: jede (aufhebenswerte) Version ist verfügbar und kann ggf. wiederhergestellt werden
2. Datensicherung - aktuellste Version liegt auf mindestens zwei Rechnern (dem eigenen und einem fremden (Server))
3. Abgleich des Codes im Team
   - Änderungen an denselben Dateien können / müssen synchronisiert werden
   - Löschen / Hinzufügen von Dateien muss synchronisiert werden
4. Nachvollziehbarkeit von Änderungen: wer hat *was*, *wann* und *warum* geändert?
5. Basis für Continuous Integration (CI)
6. → Konfigurationsmanagement

Arten der Versionsverwaltung

Zentrale Systeme

- Nur die aktuellen Dateiversionen liegen lokal, zentral alle Revisionen
- Für lange Zeit war CVS (Concurrent Version System) das Verwaltungssystem für viele Entwickler (es gab aber auch noch weitere Produkte)
- Als Nachfolger von CVS wurde Subversion eingeführt

Verteilte Systeme

- Das „zentrale“ Repository liegt als Kopie auf jedem Rechner und über diese Kopie läuft die Synchronisation
- Es gibt verschiedene Umsetzungen dieser Idee
- Die größte Verbreitung hat git erreicht

git

Version Control System

- Bei zunehmend verteilten Entwicklungsstandorten ist ein **verteiltes** Versionsmanagement ein natürliches Evolutionsziel.
- Git wurde von Linus Torvald als effizientes, verteilte Entwicklung unterstützendes System entwickelt.
- Git ist für die Ansprüche einer Hausarbeit viel zu mächtig. Da es aber inzwischen ein de facto Standard ist, wird es auch in der NORDAKADEMIE verwendet.

git's twelve

1. Repository
2. Working Directory
3. Staging Area
4. Commit
5. Log
6. Diff
7. Branch
8. Fast forward
9. Merge
10. Remote
11. Push
12. Pull, Fetch

1. Repository

- Git unterstützt die Versionskontrolle von Dateien innerhalb eines Verzeichnisses, dem sog. **Working Directory**.

Anlegen eines Working Directory

```
frank:~$ mkdir rezepte
frank:~$ cd rezepte
frank:~/rezepte$
```

- Ein git **Repository** speichert die Versionsinformationen des Working Directory. Es liegt in dem Working Directory im versteckten Unterordner .git .

Anlegen eines Git Repository

```
frank:~/rezepte$git init
Initialisierte git-Repository /home/frank/rezepte/.git/
frank:~/rezepte$
```

2. Working Directory

- Dateien im Working Directory

Erstellen von Dateien

```
frank:~/rezepte$ echo Hefeteig, Tomaten, Käse >pizza.txt
frank:~/rezepte$ echo Nudeln, Tomatensauce >spaghetti.txt
frank:~/rezepte$ echo Ei, ??? >spiegelei.txt
frank:~/rezepte$ ls
pizza.txt     spaghetti.txt   spiegelei.txt
frank:~/rezepte$
```

- Dateien im Working Directory stellen einen Arbeitszustand dar. Fertige Arbeitsergebnisse werden in die Versionsverwaltung übergeben.

3. Staging Area

- Ein konsistenter Zustand abgestimmter Arbeitsergebnisse wird in das Repository übergeben. Welche Dateien zum abgestimmten Ergebnis gehören, wird durch Zugehörigkeit zur **Staging area** festgelegt.
- Datei in Staging area kopieren: git add <Datei>
- Datei aus Staging area entfernen: git rm --cached <Datei>

Zusammenstellen abgestimmter Arbeitsergebnisse

```
frank:~/rezepte$ git add pizza.txt spaghetti.txt
frank:~/rezepte$ git status
Auf Branch main
Initialer Commit
zum Commit vorgemerkte Änderungen:
  neue Datei:     pizza.txt
  neue Datei:     spaghetti.txt
Unbeobachtete Dateien:
  spiegelei.txt
```

4. Commit

- Ein **Commit** ist definierter Zustand der Dateien des Working Directory, der jederzeit wieder hergestellt werden kann.
- git commit -m “Kommentar“

Es gibt unbeobachtete Dateien

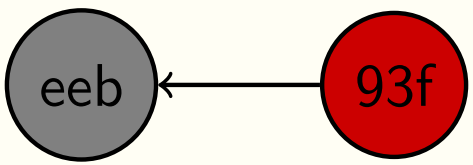
```
frank:~/rezepte$ git commit -m "Added␣pizza␣and␣spaghetti"
[main (Basis-Commit) eebcf9f] Added pizza and spaghetti
 2 files changed, 2 insertions(+)
 create mode 100644 pizza.txt
 create mode 100644 spaghetti.txt
frank:~/rezepte$ git status
Auf Branch main
Unbeobachtete Dateien:
     spiegelei.txt
```

5. Log

- Ein **Log** beschreibt die Commit Historie des aktuellen Zustands:

Log nach zwei Commits

```
frank:~/rezepte$ echo Ei, Margarine >spiegelei.txt
frank:~/rezepte$ git add spiegelei.txt
frank:~/rezepte$ git commit -m "Added␣spiegelei.txt"
[main 93fcdbb] added spiegelei
 1 file changed, 1 insertion(+)
 create mode 100644 spiegelei.txt
frank:~/rezepte$ git log --oneline
93fcdbb Added spiegelei
eebcf9f Added pizza and spaghetti
```



Praxistipp

- Gute Richtlinien für Commit Meldungen zu haben und sich danach zu richten, macht die Zusammenarbeit mit anderen und die Arbeit mit Git selbst sehr viel einfacher.
- Hochwertige Commit Meldungen zu schreiben, macht anderen und sich selbst das Leben erheblich einfacher. Im Allgemeinen sollte eine Commit Meldung mit einer einzelnen Zeile anfangen, die nicht länger als 50 Zeichen sein sollte. Dann sollte eine leere Zeile folgen und schließlich eine ausführlichere Beschreibung der Änderungen.
- Wenn möglich, Commits möglichst leichtverständlich zu gestalten:
  - Commits sollten thematisch zusammengehören.
  - Nicht ein ganzes Wochenende lang an fünf verschiedenen Problemen arbeiten und am Montag einen einzigen, riesigen Commit vornehmen.
  - Selbst wenn am Wochenende keine Commits angelegt wurden, verwendet man die Staging Area, um Änderungen auf mehrere Commits aufzuteilen und jeweils mit einer verständlichen Meldung zu versehen.

6. Diff

- Ein **Diff** beschreibt den Unterschied zwischen zwei Commits (resp. Working Directory):

```
frank:~/rezepte$ echo Hefeteig, Tomatensauce, Käse >pizza.txt
frank:~/rezepte$ git add pizza.txt
frank:~/rezepte$ git commit -m "Modified␣pizza,␣uses␣Tomatensauce␣now"
[main 92e3aa6] Modified pizza, uses Tomatensauce now
 1 file changed, 1 insertion(+), 1 deletion(-)
frank:~/rezepte$ git diff 93fcdbb
diff --git a/pizza.txt b/pizza.txt
index d7766d4..81a23b2 100644
--- a/pizza.txt
+++ b/pizza.txt
 -1 +1
-Hefeteig, Tomaten, Käse
+Hefeteig, Tomatensauce, Käse
```

Was wird womit verglichen

- git diff <commit1> <commit2> vergleicht zwei commits
- git diff <commit> vergleicht <commit> mit Working Directory
- git diff --cached <commit> vergleicht <commit> mit Staging Area
- git diff --cached vergleicht Staging Area mit Working Directory

7. Branches

- Ein **Branch** ist eine lineare Entwicklungshistorie innerhalb des Repositories.
- Branches ermöglichen die parallele ungestörte Weiterentwicklung eines Projekts. Z.B. jeder Entwickler entwickelt an seinem Feature-Branch.
- Bisher wurde der main Branch bearbeitet.

Weiterentwicklung an einem Branch

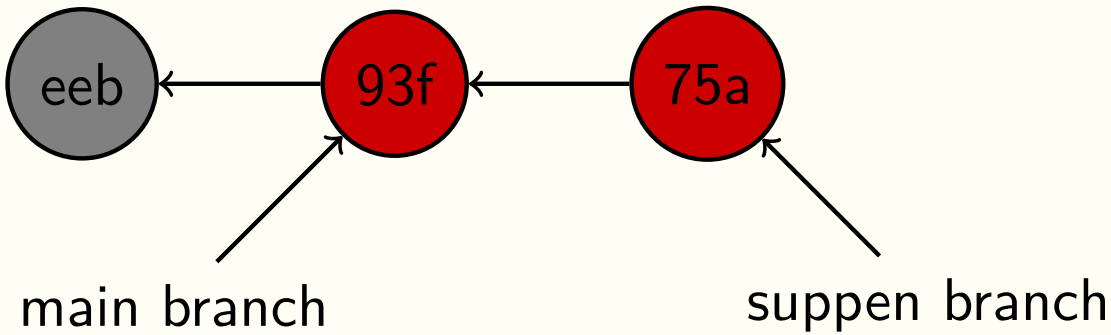
```
frank:~/rezepte$ git branch suppen
frank:~/rezepte$ git branch
* main
  suppen
frank:~/rezepte$ git checkout suppen
frank:~/rezepte$ git branch
  main
* suppen
```

Branch Development

- Branches können unabhängig voneinander entwickelt werden.
- Aufgaben, die innerhalb eines Branches entwickelt werden, stören in einem anderen Branch nicht, da sie nicht sichtbar sind.

Weiterentwicklung an einem Branch

```
frank:~/rezepte$ echo Kartoffeln, Wasser >kartoffelsuppe.txt
frank:~/rezepte$ git add .
frank:~/rezepte$ git commit -m "Added kartoffelsuppe"
```



Praxistipp

Git ist ein äußerst flexibles Versionsverwaltungssystem. Insofern sind die denkbaren Einsatzszenarien sehr vielfältig. Dennoch möchte ich eine Variante empfehlen:

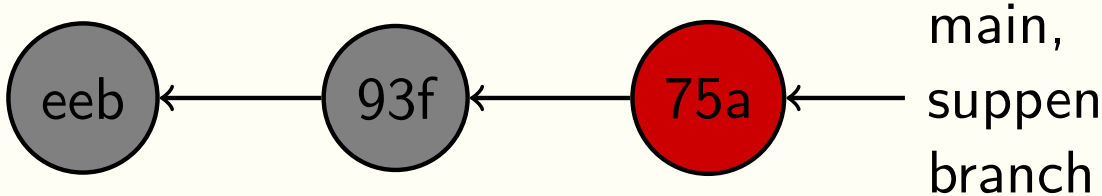
- Der main Branch enthält immer einen aktuellen lauffähigen Zustand, der getestete Funktionalitäten zur Verfügung stellt.
- Jeder Entwickler arbeitet an einem Feature-Branch. Die Feature-Branches werden regelmäßig lokal committed und ggf. in Gitlab übertragen.
- Ist das Feature implementiert, erfolgt die Konsolidierung zunächst mit dem lokalen main und dann mit dem remote main, dazu später.

8. Fast Forward: Branch Konsolidierung

- Gibt es keine konkurrierende Entwicklung in zwei unterschiedlichen Branches, so ist es leicht möglich die Branches zu synchronisieren.
- Fast Forward merges erzeugen dabei keinen neuen Commit.

Weiterentwicklung an einem Branch

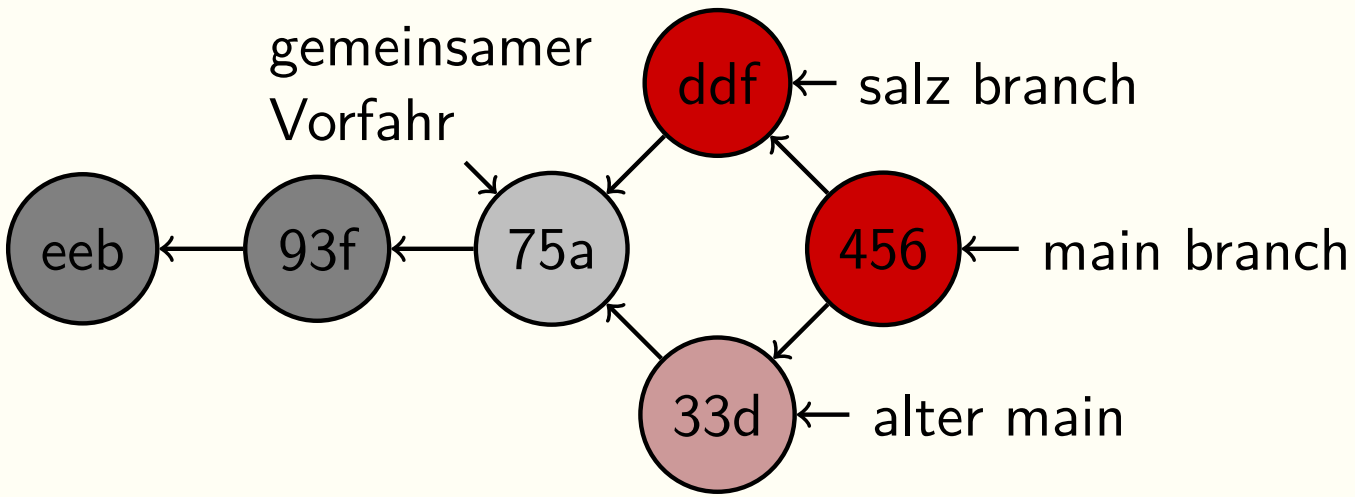
```
frank:~/rezepte$ git checkout main
frank:~/rezepte$ git merge suppen
```



9. Three Way Merge

- Entsteht, wenn an zwei Branches unabhängig weiterentwickelt wird.

```
frank:~/rezepte$ git checkout -b salz
frank:~/rezepte$ echo "Salz\nHefeteig, Tomatensauce, Käse" >pizza.txt
frank:~/rezepte$ git commit -a -m "Modified pizza, uses Salz"
frank:~/rezepte$ git checkout main
frank:~/rezepte$ echo Oregano >>pizza.txt
frank:~/rezepte$ git commit -a -m "Modified pizza, uses Oregano"
frank:~/rezepte$ git merge -m "mergeBranch Salz" salz
```



Merge Konflikte

- Sind die unabhängigen Entwicklungen an verschiedenen Stellen innerhalb einer Datei, kann das merge Programm diese automatisch konsolidieren.
- Eine Zeile wird nur in einem Branch geändert.
- Hinzufügen von Zeilen an unterschiedlichen Stellen.
- Häufig entstehen jedoch **Merge-Konflikte**, die automatisiert nicht gelöst werden können.
  - Änderungen an derselben Zeile. Hier ist nicht erkennbar, welche Änderung übernommen werden soll.
  - Einfügungen an derselben Position. Hier kann die Reihenfolge nicht festgelegt werden.
- Im Fall von Merge Konflikten muss der mergende Entwickler vorgeben, wie der Konflikt aufgelöst werden soll.
- git stellt verschiedene Merge Strategien zur Verfügung und ermöglicht die Einbindung von eigenen Merge Programmen.

10. Remote (nicht unbedingt zentrale) Repositories

- Bisher wurde immer mit einem lokalen Repository gearbeitet. Dies ermöglicht zwar die versionsgesteuerte Entwicklung lokal ohne Internetverbindung, aber noch nicht, die Entwicklung kooperativ durchzuführen.
- git verwendet **remote** Repositories, um mehrere Entwicklerstandorte miteinander zu koordinieren. Dabei ist keine zentrale Struktur verpflichtend, sondern es können auch komplexere netzartige Strukturen miteinander koordiniert werden.
- Zugriff auf remote Repositories erfolgt über
  - https: Eingabe eines Passworts bei jedem Austausch erforderlich.
  - ssh: Verwendung einer PKI möglich. Der Nutzer identifiziert sich durch einen zum Remote Repository hochgeladenen public key.

git Kommandos die remote Repos betreffen

- **git remote** ... definiert eine Verbindung zu einem remote Repository.
- **git push** ... überträgt einen lokalen Branch in ein remote Repository.
- **git fetch** ... überträgt einen Branch aus einem remote Repository ins lokale.
- **git pull** ... überträgt einen Branch aus einem remote Repository ins lokale. Merge integriert.
- **git clone** ... erstellt Kopie eines remote Repository.

git Setup

Für Git können einige identifizierende Werte hinterlegt werden. Dies in der Git config möglich.

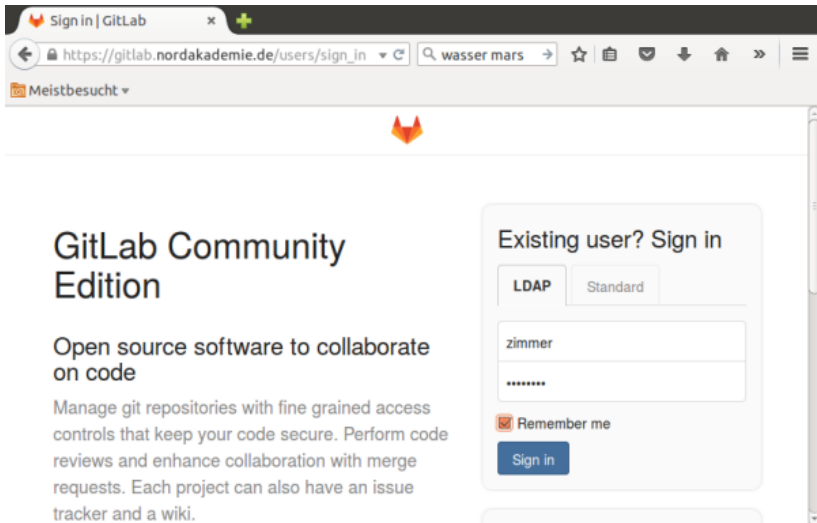
Git config ändern

```
git config --global user.name "FIRST_NAME␣LAST_NAME"
git config --global user.email "MY_NAME@example.com"
```

GitLab

GitLab

Die Nordakademie verwendet ein remote ansteuerbares Repository auf der Basis der Open Source Software **GitLab**.

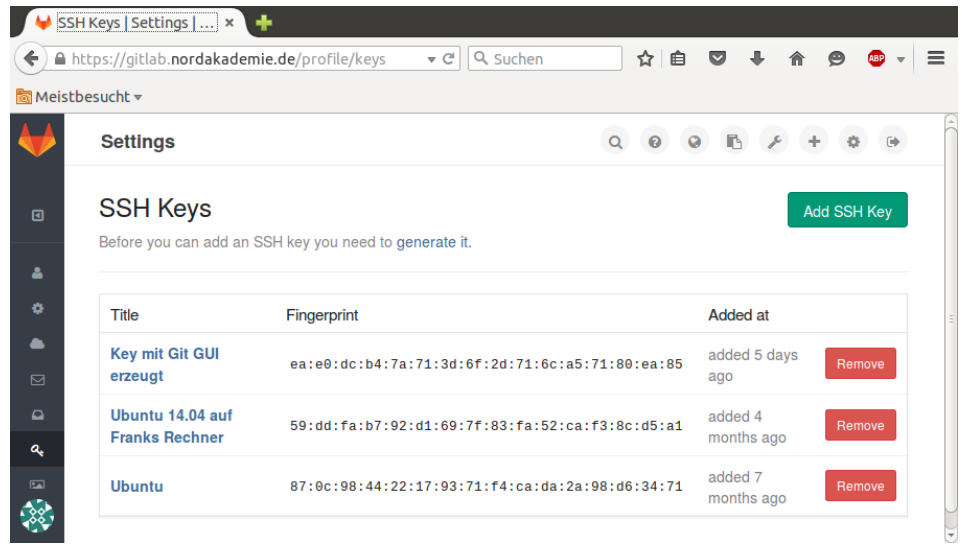


- Gitlab ist ein Web-Frontend für git.
- Hat LDAP-Verbindung zum AD der NORDAKADEMIE: Anmeldung mit gewohnter Nutzerkennung und dem gewohnten Passwort.
- Zugriff auf gitlab mit:
  - https: Nutzer und Kennworteingabe beim push/pull erforderlich.
  - ssh: private und public key-pair erforderlich.
- Web-Oberfläche ist erreichbar unter: https://gitlab2.nordakademie.de

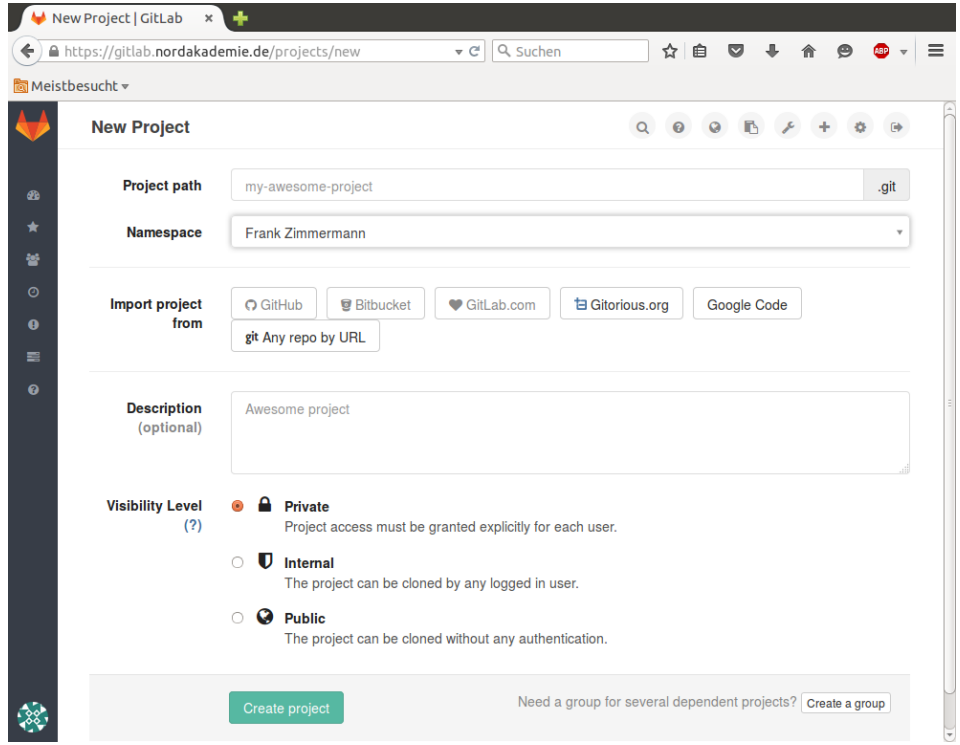
Erzeugen eines pp Keypair mit git-gui

Der Zugriff mit ssh setzt einige Vorarbeiten voraus:

- Erzeugung eines private public key-pairs in den Dateien id_rsa.pub und id_ rsa mit
  1. Eclipse (Win, Mac OS, Linux)
  2. git-gui (Win)
  3. ssh-keygen (Mac OS, Linux)
- Dateien werden im Verzeichnis .ssh im user home Directory abgelegt.
- Der public key muss in gitlab im Nutzerprofil hinterlegt werden.



Erzeugen eines Projekt Repository mit gitlab



- Nachdem lokal ein key-pair erzeugt wurde und der public key in gitlab bekannt gemacht wurde, kann man auf git ein neues Projekt anlegen.
- Der Zugriff auf das Projekt kann über den **Visibility Level** gesteuert werden.

Lokale Verknüpfung zum Remote Repository

```
git config --global user.name "Vorname Nachname"
git config --global user.email "vorname.nachname@nordakademie.de"
git remote add origin https://gitlab2.nordakademie.de/LDAPUSR/Rezepte.git
```

Push main Branch

Durch das Push Kommando kann ein Branch (im Beispiel der main Branch) in das remote Repository übertragen werden. Das remote Repository wird über den im git remote add vergebenen namen **origin** identifiziert.

Aktualisieren des Remote Repository

```
frank:~/rezepte$ git push origin main
Counting objects: 25, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (25/25), 1.94 KiB | 0 bytes/s, done.
Total 25 (delta 7), reused 0 (delta 0)
To LDAPUSR@gitlab2.nordakademie.de:LDAPUSR/Rezepte.git
```

Kopien eines Branches

Es ist vielleicht noch einmal an der Zeit, daran zu erinnern, wie viele Kopien einer Datei jetzt gehalten werden:

1. Der lokale Stand im Working Directory.
2. Der Stand in der Staging Area.
3. Der aktuelle Stand des Branches im lokalen Repository.
4. Der aktuelle Stand des Branches im remote Repository.

Pull main Branch

Ist der Branch anderweitig geändert worden, muss der lokale Stand nachgezogen werden. Dies ist mit einem Fast Forward oder mit einem Merge verbunden. Im Falle von Merge Konflikten muss man die bereinigten Kopien konsolidieren.

Aktualisieren des remote Repositories

```
frank:~/rezepte$ git fetch origin main

...

frank:~/rezepte$ git checkout main
frank:~/rezepte$ git merge origin/main

... bereinigen der Merge Konflikte

frank:~/rezepte$ git commit -a -m "Merged"
frank:~/rezepte$ git push origin main
```

Workflow Konsolidierung lokaler main Branch 1/2

- Ist die lokale Integration eines Feature-Branches abgeschlossen, muss die Entwicklung mit dem remote Repo erfolgen.
- Meist ist kein Fast-Forward (**git push origin main** schlägt fehl!) des remote main Branches möglich, da inzwischen andere Entwickler ihre Entwicklungen integriert haben und der main Branch des remote Repositories weiter ist, als der main Branch des lokalen Repositories. Ein merge ist unvermeidlich.
- Der merge muss auf einem lokalen Repository durchgeführt werden.
- Deshalb muss mittels eines **git fetch origin main** eine Kopie des aktuellen Entwicklungsstand gehoben werden. Dieser Entwicklungsstand entspricht dem lokalen Branch origin/main.

Workflow Konsolidierung lokaler main Branch 2/2

- Im lokalen main branch (**git checkout main**) muss nun der origin/main Stand hineingemergt werden (**git merge origin/main**).
- Mergekonflikte werden im Working Directory manuell bereinigt.
- Durch Übertragung in die Staging Area (**git add <datei>**) und einen Commit (**git commit -m "merged" main**), wird der lokale main branch eine Weiterentwicklung des origin/main.
- Danach kann der lokale main branch in das remote Repository gepusht (**git push origin main**) werden, der Push ist ein Fast Forward.

Git in IntelliJ

Git in IntelliJ 1/2

- IntelliJ beinhaltet Git Unterstützung.
- Diese ermöglicht den Zugriff auf die Git Funktionalitäten mit einer graphischen Oberfläche.
- Nicht immer ist klar, welche Aufrufe von Git Befehlen sich hinter den Menüpunkten von IntelliJ verbergen.
- Ein gutes Verständnis der Konzepte von git macht den Umgang mit der Oberfläche einfach.
- Verständnis von Git ist vorausgesetzt, um die grafische Git Oberfläche sinnvoll zu verwenden.

Git in IntelliJ 2/2

Es ist sinnvoll solche Funktionalität in der IDE zu haben - selbst wenn man sie niemals direkt verwendet!

- Refactorings können Auswirkungen auf die im Repository liegenden Dateien haben (bspw. Namensänderungen).
- Die Plugins sind überlicherweise so integriert, dass diese Operationen für das Versionsmanagement nachvollziehbar sind

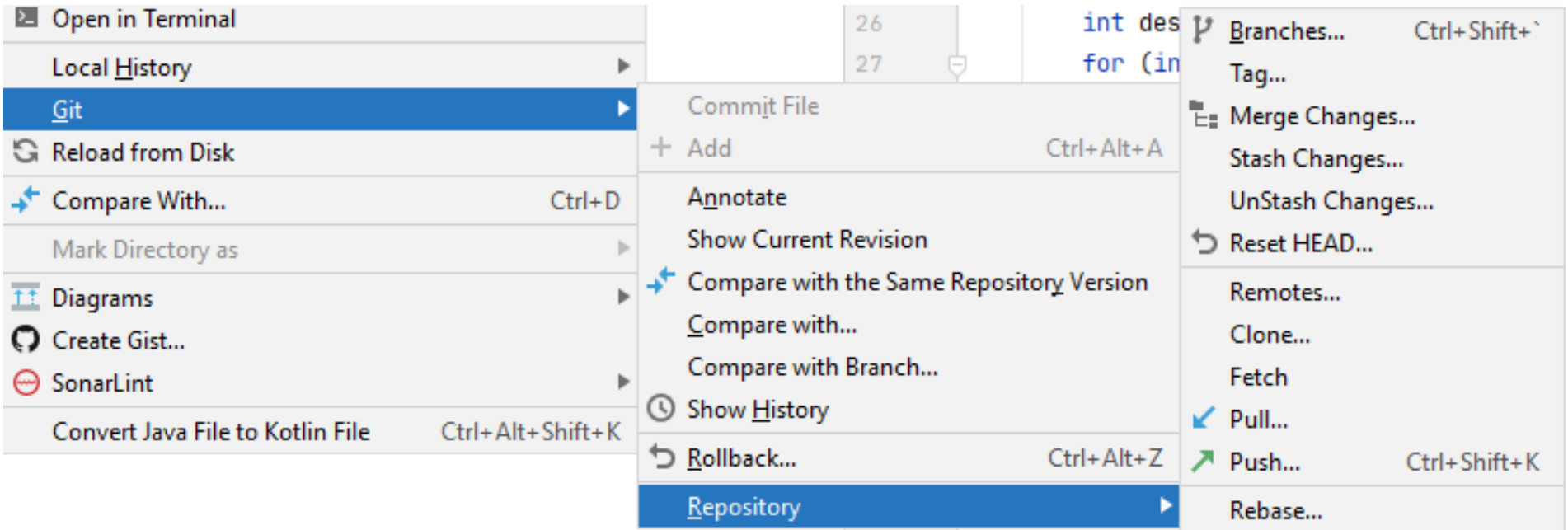
Konfiguration

- Git kann über die Einstellungen konfiguriert werden. Für gewöhnlich sind die Standardeinstellungen sinnvoll.
- Die IDE kann aber auch an spezifische Git Workflows angepasst werden.

Menüpunkte

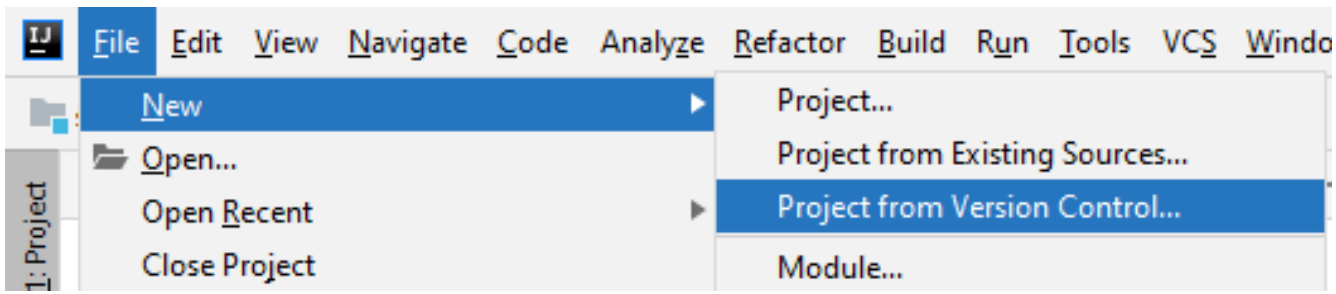
Die üblichen git Kommandos kann man über das Kontextmenü benutzen.

- Add entspricht git add
- Commit entspricht git commit
- Show History zeigt die Bearbeitungsschritte der Datei.
- Unter dem Punkt Repository können neben mehreren weiteren Optionen Branches gewechselt und Interaktionen mit einem Remote Repository durchgeführt werden.



Clone Remote

- Remote Repositories können mit dem Erstellen eines neuen Projekts geklont werden.
- Wählen Sie File → New → Project from Version Control und geben Sie die entsprechende URL an.
- Nutzen Sie die https Variante des Repositorypfads, sodass Sie sich mit Nutzername und Passwort authentifizieren können.



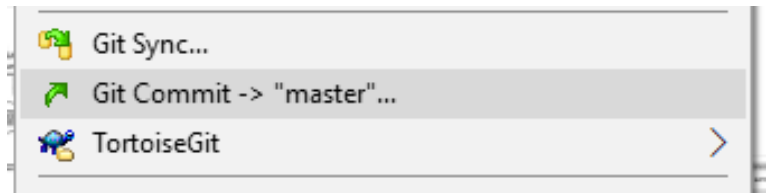
TortoiseGit

TortoiseGit

- https://tortoisegit.org/
- TortoiseGit ist eine Shell-Erweiterung für Windows (basiert auf TortoiseSVN)
- Die git Befehle stehen dadurch per Kontextmenü in Ordnern/auf Dateien zur Verfügung
- Ein-/Ausgaben und diverse Befehle werden durch Dialoge unterstützt

TortoiseGit

Shell Kontextmenü

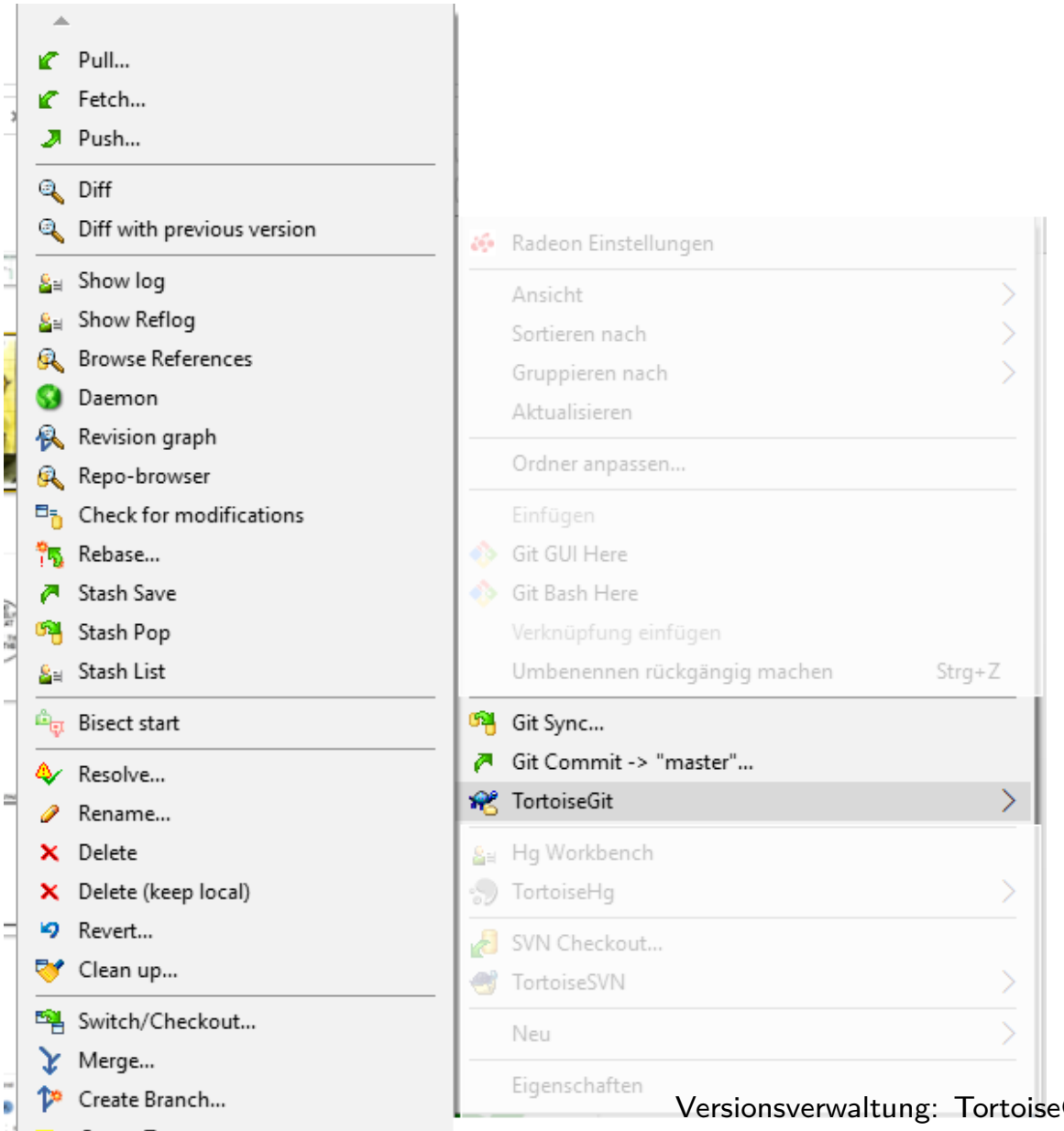


Die am häufigsten verwendeten Befehle stehen direkt zur Verfügung:

- Sync für push/pull Operationen
- Commit der aktuellen Änderungen
- Viele weitere Befehle können über das erweiterte Kontextmenü aufgerufen werden

TortoiseGit

Befehle im erweiterten Kontextmenü



- Pull, Fetch, Push
- Diff
- Logging, Stashing, ...
- Konfliktbehebung, Dateioperationen, Rückgängig machen, Aufräumen
- Branches wechseln, zusammenführen

Aufgaben

Aufgaben 1/5

- Beispiele Versionsverwaltung
  - Google Statemachines `https://github.com/oxo42/stateless4j`
  - Eclipse `https://git.eclipse.org/c/`
  - Linux `https://github.com/torvalds/linux`
- Finden Sie mindestens drei weitere Beispiele von öffentlichen Git Repositories, die bekannte oder von Ihnen benutzte Software verwaltet.

Aufgaben 2/5

- Öffnen Sie die GitBash
- Prüfen Sie mittels des Befehls `git --version`, ob Git im Pfad vorhanden ist und welche Version installiert ist
- Schauen Sie, welche Optionen Ihnen mit dem Ausführen von `git config` angeboten werden
- Konfigurieren Sie Ihr Git, indem Sie Ihren Namen und die E-Mail Adresse global in der Konfiguration setzen
- Schauen Sie sich Ihre Git Konfiguration mit `git config --list` an und verifizieren Sie, dass Ihre soeben getätigten Konfigurationen vorliegen
- Für Git Konfigurationen gibt es unterschiedliche File Locations. Stellen Sie mit `git config --global --list` sicher, dass die Konfiguration global vorliegt

Aufgaben 3/5

- Erstellen Sie ein neues Projekt in GitLab in Ihrem Namespace
- Importieren Sie dieses Projekt in IntelliJ mittels File → New → Project from Version Control
- Laden Sie Ihre Gruppenmitglieder zu dem Projekt ein und lassen Sie diese Ihr Projekt in ihrer IDE importieren
- Erstellen Sie eine Java Klasse A und committen + pushen Sie diese
- Ihr Gruppenmitglied soll diese nun bei sich „pullen“
- Sie fügen nun etwas am Anfang dieser Datei ein, Ihr Gruppenmitglied am Ende, beide sollen nun committen und pushen.
  - Ggf. muss einer von Ihnen vor dem „pushen“ nochmal „pullen“ - Warum?
- Sie ändern nun beide die erste Zeile der Datei (in dem Sie einen Kommentar mit ihrem eigenen Namen einfügen) und versuchen wieder beide diese ins Repository zu bringen
  - Ggf. muss einer von Ihnen vor dem „pushen“ nochmal „pullen“ und „mergen“ - Warum?

Aufgaben 4/5

- Versuchen Sie etwas vergleichbares, aber verwenden Sie ein anderes Werkzeug als eben (GitBash, EGit, Git in IntelliJ, TortoiseGit)
- Betrachten Sie Ihr Projekt in GitLab - schauen Sie sich die Statistiken an und was Sie über Ihr Projekt herausfinden können
- Finden Sie heraus, wer welche Zeile in Ihrer Klasse A zuletzt bearbeitet hat
- Schauen Sie sich die Unterschiede in der Klasse A zwischen 2 Revisionen an

Aufgaben 5/5

- Reflektieren Sie das bis hierhin gewonnene Wissen
- Gehen Sie anschließend in den moodle-Kurs dieser Vorlesung und führen Sie eigenständig das Git Quiz durch
- Nach Absolvieren vom Quiz lesen Sie sich vertiefend in die Dokumentation von Git ein
- Lesen Sie sich insbesondere bei den Befehlen ein, die Sie im Quiz falsch beantwortet haben oder bei denen Unklarheiten bestehen
- Danach machen Sie sich mit dem Befehl „git stash“ vertraut und probieren diesen aus
- Finden Sie heraus, was `git rebase` tut und wofür Sie es einsetzen können. Wann ist ein Rebase dem Merge gegenüber zu präferieren?
- Probieren Sie den Befehl `git rebase` aus.