The git must flow

a git survival kit - german version

kafawi

1. Mai 2017

Dieses Dokument ist speziell für das Praktikum des ESE Moduls 2017 des Studienganges Technische Informatik der HAW Hamburg erstellt worden und dient der Zusamenarbeit der Gruppe LANKE. Es werden hier die Grundideen mit dem Umgang mit git und dem ¨flow¨ erklärt. Die Commit Message Struktur sowie die Branchstruktur wird hier beschlossen.

Inhaltsverzeichnis

1 Einleitung **1**

2 Workflow **2**
2.1 Zusammenarbeit im Workflow . . . 2
2.1.1 Erstellung eines features und veröffentlichen . . . 3
2.1.2 Arbeiten an einem öffentlichen Feature . . . 3
2.1.3 Feature in den develop mergen . . . 4
2.1.4 Realese . . . 4

3 Konventionen **5**
3.1 Git Commit Message . . . 6
3.1.1 Struktur / template . . . 6

4 Cheat sheet **7**

5 Tipps und Tricks **8**
5.1 Autocompletion . . . 8
5.2 Alias . . . 8
5.3 ssh-key . . . 8
5.4 Tools . . . 8
5.5 Dose vs Unix . . . 9
5.6 persönliches .gitignore . . . 9

1 Einleitung

Dies soll ein Crashkurs in Sachen Git für dieses Projekt darstellen. Es wird sich hier auf das Nötigste beschränkt. Für tiefergreifende Informationen sollten externe Quellen verwendet werden, wie [Chacon and Straub, 2014], welches online frei zur verfügung steht.

Der Quellcode in den Listings zeigt nur die Gitbefehle, wie sie in einem Terminal verwendet werden. Die Ausgaben werden nicht aufgeführt. Als Prompt wird \$ verwendet. Davor steht der aktuelle Branch (checktout branch) in Klammern $(branchname)$\$. Kommentare werden mit einem Voranngeführten $\#$ eingeleitet.

```
1 (master)$ git checkout develop
2 # Update develop
3 (develop)$ git pull
4 (develop)$ git checkout -b feature/myfeature
5 (feature/myfeature)$ ...
```

Zunächst wird der Workflow, welcher in diesem Projekt praktiziert werden soll, erklärt. Darauf folgen die Konventionen im Umgag mit Git, wie die Struktur der Commit Messages. Darauf Folgt eine Auffrischung der Git Befehle. Im Letzten Kapitel werden nützliche Tricks und Kniffe vorgestellt.

2 Workflow

Wegen der sehr kleinen Anzahl von Entwicklern, wird ein zentralisierter Workflow angewendet, d.h. wir haben ein Referenzrepro auf einem Server, welcher uns den aktuellen Stand für unsere lokalen Repros anbietet. Dabei haben wir zwei Hauptbranches und mehrere dynamische Featurebranches, auf denen hauptsächlich Entwickelt wird. Dargestellt ist dies in der Abbildung 1.

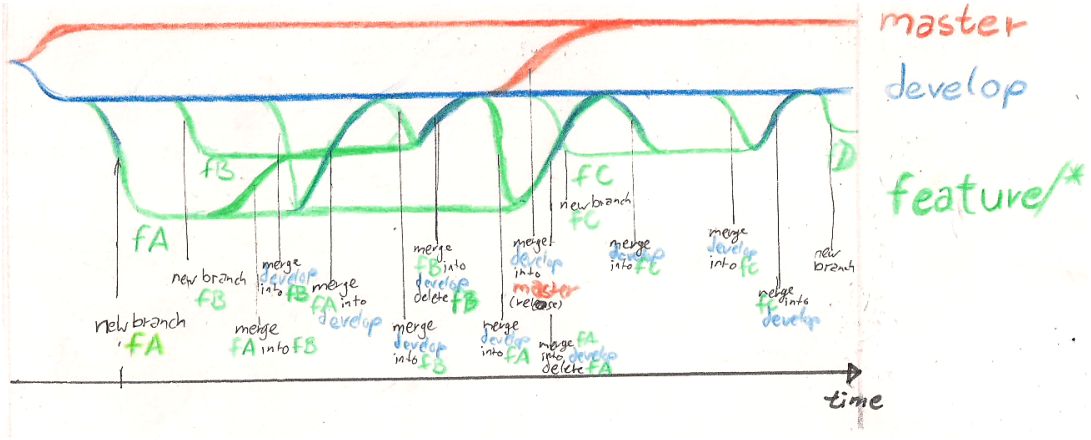

Abbildung 1: Der Workflow, wie er in diesem Projekt stattfinden soll, ist hier dargestellt. Der Masterbranch `master` ist für die Releases bestimmt und nur der `develop` wird kurz vor einem Release gemerged. Vom Entwicklungsbranch `develop` werden die jeweiligen Featurebanches `feature/*` abgeleitet und auf diesen wird in kleinen Teams gearbeitet. Wenn ein Feature fertig gestellt wird, wird dieses auf dem mit dem `remote` upgedateten `develop` eingepflegt. Wird der Featurebranch nicht mehr gebraucht, wird dieser lokal und im remote gelöscht (`git branch -dr origin/feature/*`).

master Dieser `branch` ist nur für Realeses bestimmt, und wird nur mit dem `develop` gemerged. (neuster Stand des Projektes für den nächsten Praktikumstermine)

develop Stellt die Basis für jedes `feature/*` da. Jedes einzelne `feature/*` kann in den `develop` gemerged werden, sobald dieser für annehmbar (qualitativer Inhalt) befunden wurde.

feature/* Hier werden die einzelnen Tasks in Code umgewandelt. Diese `branches` leiten sich von den dem `develop` ab und werde in diesen bei der Fertigstellung des `feature/*` zurückgeführt und ggf. gelöscht. Es kann zwischen den `feature/*` auch gemerged werden, falls diese ein bestimmtes `feature/fA` dringend braucht, welches sich noch nicht auf dem develop befindet, da es z.B. in der Review befindet. Vor jedem merge in den `develop` muss das `feature/myF` den aktuellen Stand des `origin/develop` in sich integrieren (mergen).

2.1 Zusammenarbeit im Workflow

Wenn mehrere Entwickler an einem `feature/*` arbeiten, dann müssen sie sich miteinander Absprechen, bzw. vor jedem Sprint, wird der aktuelle Stand dieses `origin/feature/*` auf den lokalen `branches` angeglichen.

2.1.1 Erstellung eines features und veröffentlichen

Im nebenstehen Listing wird ein Ablauf beschrieben, wie man ein neuen Featurebranch anlegt und diesen veröffentlicht. Dabei kann es auch vorkommen, dass ein anderer Entwickler einen Featurebranch mit gleicher Intention angelegt hat und diesen vorher schon veröffentlicht hat. Dann muss man seinen eigenen Branch mit dem schon vorhandenen zusammenführen. Wichtig ist, dass jeder `feature/*` von einem aktuellen `develop` abstammt.

```
(master)$ git checkout develop
# Update develop
(develop)$ git pull
# create and checkout new branch
(develop)$ git checkout -b feature/fA
# working some on some files and stage them
(feature/fA)$ ...
(feature/fA)$ git commit
# looking for new branches in remote
(feature/fA)$ git fetch
(feature/fA)$ git branch -a
# two possibilities
#     1. my feature is not a existing topic
#        -> publish
(feature/fA)$ git push -u origin feature/fA
#     2. my partner already has a fAA-branch
(feature/fA)$ git checkout --track origin/feature/fAA
(feature/fAA)$ git merge feature/fA
# delete my branch
(feature/fAA)$ git branch -d feature/fA
(feature/fAA)$ git push
```

2.1.2 Arbeiten an einem öffentlichen Feature

Es wird zunächst der featurebranch lokal auf den neusten Stand gebracht. Dann sollte man erstmal angucken, was sich in geändert hat. Nach getaner Arbeit wird `feature/*` aktualisiert und und entsprechend getestet. Erst dann wird `feature/*` auf das remote gepusht.

```
# update develop
(develop)$ git pull
# get featurebranch fA
# if not existing
    (develop)$ git checkout --track origin/feature/fA
# else
    (develop)$ git checkout feature/fA
    (feature/fA)$ git pull
# whats new?
(feature/fA)$ git log --since=1.weeks
# starting with Sprint and commit
(feature/fA)$ ...
(feature/fA)$ git commit
# update from repro
(feature/fA)$ git fetch
# while a new version of fA is online
    (feature/fA)$ git merge origin/feature/fA
    # testing after merge
    (feature/fA)$ ...
    (feature/fA)$ git commit
    (feature/fA)$ git fetch
# publish / push
(feature/fA)$ git push
```

2.1.3 Feature in den develop mergen

Das Feature ist aktuell und fertig geprüft, sodass es in den `develop` überführt werden kann. So wird zunächst der lokale `develop` geupdatet und anschließend in den `feature/*` eingefügt. Dann werden Tests durchgeführt, ggf. Konflikte gelöst und anschließend wird der `feature/*` in den `develop` einpflegt. Wenn `feature/*` nicht mehr gebraucht wird, so wird dieser vom lokalen und remote gelöscht.

```
(feature/fA)$ git checkout develop
# Update develop
(develop)$ git pull
# do :
    #switch to feature
    (develop)$ git checkout feature/fA
    # merge devolop into feature
    (feature/fA)$ git merge develop
    # testing
    (feature/fA)$ ...
    (feature/fA)$ git commit
    (feature/fA)$ git checkout develop
    # update develop again
    (develop)$ pull
# while ( develop changes) -> do
(develop)$ git merge feature/fA
# if feature is no longer needed
    # delete lokal
    (develop)$ git branch -d feature/fA
    # delete remote
    (develop)$ git push origin :feature/fA
    (develop)$ git branch -dr origin/feature/fA
```

2.1.4 Realese

Zu einer gewissen Zeit wird dem `master` der `develop` einverleibt. Dabei sollte sich auf `develop` ein vorzeigbarer Snapshot des Projektes befinden. Gegebenen falls kann man hier dem Commit einen Tag geben. Es wird aber ein kommentierter Tag bevorzugt.

```
# update develop
(develop)$ git pull
# update master
(develop)$ git checkout master
(master)$ git pull
# merge develop into master
(master)$ git merge develop
# publish
(master)$ git push
# tagging
(master)$ git tag -a v0.1
(master)$ git push origin v0.1
```

3 Konventionen

An diese Regeln sollte sich jeder Entwickler halten. Diese Regeln sorgen dafür, dass sich der Workflow einstellen kann und es zu keinen komischen Ausfällen kommt.

- update before you work (update = pull = fetch + merge)
- update before you merge and push
- never rebase in an public branch
- never use git commit –amend to a published commit
- write commit messages in an editor with a template
- all feature branches start with `feature/`

3.1 Git Commit Message

Es wird die Grundstrukur der AngularJS Communitiy [AngularJS, 2017] verwendet und auf unsere Bedürfnisse angepasst. Die Commit messages werden in Englisch verfasst.

3.1.1 Struktur / template

Ein Template ist in der Datei `.gitmessage` im Wurzelverzeichnis unseres Projektes und wird mit `git config commit.template ".gitmessage"` eingepflegt. Diese Vorlage wird in deinen Editor geladen, wenn ein Commit ohne "-m" Flag aufgerufen wird.

```
<type>(<scope>): <subject>
BLANKLINE
<body>
BLANKLINE
<footer>
--- EXAMPLE ---
docs(RDD): Add stakeholder list

Stakeholder are a hint, how we have to
designe our Project and how we have to
behave.

Close stakeholder card
```

head: Hier werden die wichtigesten Informationen zusammengefasst
- Er darf maximal 50 Zeichen lang sein

<type>: Typ des Commits, welche da sind:
- docs: Änderungen in der Documentation
- feat: generell Änderung am Produktcode
- test: Tests: keine Änderung am Produktcode
- refac: Refactoring am Produktcode
- fix: gefixte Bugs
- style: änderung am Style (Einrückung etc.)

<scope>: optional
- Datei, Module, Layer auf die sich die Änderung bezeiht
- Wenn mehrere betroffen sind: (*) und Liste im <body>

<subject>: Was bewirkt die Änderung im Kern?
- " If applied, this commit will <subject> "
- Imparativ und Präsens
- Beginnend mit einem Großbuchstaben
- Endet ohne Punkt

<body>: Warum wurde die Änderung gemacht?
- Imparativ und Präsens.
- Listen werden mit [-] unterteilt (z.B. betroffene Datein).
- Maximal 72 Zeichen pro Zeile.

<footer>: optional
- Tickets oder Referenzen zu Artikeln.
- BREAKING CHANGES: kurze Beschreibung.
- Maximal 72 Zeichen pro Zeile.

4 Cheat sheet

Local Changes

```
# List changed files in workingdir
git status

# Show changes to tracked files
# repro vs. working dir
git diff

# Stage current changes for commit
git add *

# Add just some changes
git add -p <file>

# Commit all staged files
git commit

# Commit alle changed tracked files
git commit -a

# Commit with detail info (diff)
git commit -v

# Change last commit
# to rewrite the commit msg
# (forbitten for  pubished commits)
git commit --amend

# Remove files (modifiations)
git rm <file>

# Rename files
git mv <form> <to>

# Remove files from stage
git rest HEAD <file>

# Undo all modifications in working dir
git checkout -- <file>

# Revert a Commit (makes a new commit)
git revert <commithash>
```

History and info

```
# Show all commits
git log

# Show changes for one file
git log -p <file>

# Show changers of file
git blame <file>
```

Config and Init

```
# set Username and adress
git config --global user.name <name>
git config --global user.email <email>

# Clone an existing repo
git clone <adr to repro>

# Create a new local repo
git init
```

Branches and Merge

```
# List all existing branches
git branch -av

# Switch to branch (set HEAD pointer)
git checkout <branch>

# Create a new branch
(base)$ git branch <new>

# Create a new trackingbranch from remote
git checkout --track <remote/branch>

# delete local branch
git branch -d <branch>

# Merge branch into base
(base)$ git merge <branch>

# use mergetool to resolve conflicts
git mergetool
```

Remote

```
# Show info of remote
git remote show <remote>

# Add new remote
git remote add <alias> <url>

# Download all changes from remote
# without merge them into HEAD
git fetch <remote>

# Download all changes from remote
# and merge them directly
git pull <remote>

# Publish local changes to remote
# Create a new branch
git push <remote> <branch>

# Delete branch on remote
git branch -dr <remote/branch>
```

5 Tipps und Tricks

5.1 Autocompletion

Mit einem kleinen Tool, kann man mit `TAB` Gitbefehle vervollständigen lassen. Unter Windows ist dies meist schon im Git-Packet enthalten. Linuxer mit bash können das mit folgendem Vorgehen einrichten.

```
# download in to $HOME
$ cd ~
$ curl -OL https://raw.githubusercontent.com/git/git/master/contrib/completion/git-completion.
    bash
$ mv ~/git-completion.bash ~/.git-completion.bash
```

Ändere deine `./bashrc`, indem du Folgendes einfügst:

```
if [-f ~/.git-completion.bash];then
  source ~/.git-completion.bash
fi
```

5.2 Alias

Setze aliase wo immer du kannst.

Beispiele:

```
# nice logg
git config --global alias.logg "
log␣--graph␣--decorate␣--oneline␣--no-merges␣--all"

# diff Stage vs HEAD
git config --global alias.difs "diff␣--staged"

# update develop
git config --global alias.updev "
!git␣checkout␣develop␣&&␣git␣pull␣&&␣git␣checkout␣-
&&␣:"
```

Einfacher ist es aber direkt in die Datei `/.gitconfig` reinzuschreiben.

5.3 ssh-key

Lege dir einen SSH-key an, um nicht immer wieder bei jedem `push` deinen Namen und Passwort einzugeben.

Die Anleitung findet man bei den jeweiligen Anbietern der Remoteserver: bitbucket

5.4 Tools

Setze Tools ein und lege sie fest:

```
git config --global core.editor vim
git config --global merge.tool vimdiff
```

Und rufe ggf. die Hilfe von git auf:

```
git help <cmd>
git <cmd> --help
man git -<cmd>
```

5.5 Dose vs Unix

Das alte Lied von den Zeilenenden. Um dies zu vermeiden, setze folgendes um:

```
# UNIX
git config --global core.autocrlf input
# WINDOS
git config --global core.autocrlf true
```

5.6 persönliches .gitignore

Da fast jeder Entwickler seine eigenen Werkzeuge verwendet, sollten die Nebenprodukte dieser nicht von Git gesehen werden.

Um dies zu verhindern, sollte eine Datei `/.global-gitignore` erstellt und git mit
`git config -global core.excludesfile ' /.global-gitignore'`
eingepflegt werden.

Gute Templates findet man unter github.

Literatur

AngularJS. Contributing to AngularJS. https://github.com/angular/angular.js/blob/master/CONTRIBUTING.md#commit (2017-04-28), 2017.

Scott Chacon and Ben Straub. *Pro Git*. Apress, Berkely, CA, USA, 2nd edition, 2014. ISBN 1484200772, 9781484200773. https://git-scm.com/book/en/v2 (2017-04-28).