VSP AG
BORSIGSTRASSE 18
65 205 WIESBADEN

GIT

Richtlinien und Arbeitsprozess

Datum: September 5, 2011

Erstellt von: Patrick-Emil Zörner

Contents

List of Figures

1 Basics

1.1 Getting started

The programme git itself and all the commands have help pages e.g.: `git --help` will show you the general and most commonly used commands. The first one listed is `git add`. This command like all other commands has its own help called by: `git add --help`. Alternatively and maybe semantically more meaningful and faster `git help add`. Choose whatever suites you best.

Literature

A formatted and hyperlinked version of the latest git documentation can be viewed at kernel.org. There is also a »gittutorial« at kernel.org and for more in-depth introduction known as the »Git User's Manual« also at kernel.org. For offline as well as online read there is a book called »Pro Git« at progit.org. Another good online read is gitready.com with many examples for beginner, intermediate and advanced git users.

Most basic workflow before writing code

First of all ensure the only *"dirty"* files in your typo3 instance whether local or remote are: *.htaccess, db_script, robots.txt* and *typo3conf/localconf.php*. They are required to be in git because they also might be »point of failure« on the live platform. This is done as shown on line two in listing 1 (section 1.3). The output should look like listing 2.

Listing 1: Most basic example for getting started

```
1 # Show the branch (assume current branch is master)
2 # If not see branching how to checkout a branch
3 > git branch
```

```
4  # Show the status
5  > git status
6  # Hide dirty files
7  > git stash
8  # Get the current master from upstream repository origin
9  > git pull --rebase origin master
10 # Restore the locale configuration
11 > git stash pop
```

Now in order to be able to `pull` you need a *"clean"* working directory. A pretty neat feature in git is the `stash` (see `git stash --help`) which stashes the changes in a *"dirty"* working directory away. See the command on line four of listing 1.

In this example the assumption is made that the current local branch is the master branch and it should be updated from a remote repository (e.g.: github.com).

Listing 2: Output for git status

```
1  # On branch master
2  # Changed but not updated:
3  #   (use "git add <file>..." to update what will be committed)
4  #   (use "git checkout -- <file>..." to discard changes in working directory)
5  #
6  #       modified:   .htaccess
7  #       modified:   db_script
8  #       modified:   robots.txt
9  #       modified:   typo3conf/localconf.php
10 #
11 no changes added to commit (use "git add" and/or "git commit -a")
```

Understanding pull A simple `pull` is a `fetch` followed by a `merge`. In most cases that is not wanted and the `rebase` is the preferred method that is why that option is used on line eight of listing 1. It takes the other branch and attempts to put the current branch on top of it (see `git rebase --help` for a more

detailed explanation). On a conflict during `pull -rebase` it can happen that the rebase fails. In that case you „fall out“ of the current branch and find yourself in `no branch`. This means you have a detached head. The `origin master` tells git to pull from master in the remote repository. Specifying so ensures that git does not attempt to `pull` from a local branch.
On line ten of listing 1 the local configuration of the server is played back using `pop`. The command removes the last stashed state from the stash applying it to the current working directory. Basically it is an undo of line four. This can fail with conflicts.

1.2 Adding, moving (renaming) and removing

To add a file to the git use `git add FILENAME`. The Syntax to recursively add a directory is `git add PATH/DIRNAME`. When files are in the git repository it is more efficient to also tell git that these files moved (have been renamed) or removed. Use the cached option to remove files from repository but not locally.

Listing 3: Removing

```
1 > git mv SOURCE DESTINATION
2 > git rm
3 > git rm --cached
```

1.3 Branching and committing

It is useful to create a new branch if you work on a new project e.g.: a bugtracker ticket number and/or note identifier within this ticket and you are encouraged to do so. The concept and principle is to »divide and conquer«. It must always be possible to quickly find out which purpose a branch serves and who the owner is.

Therefore the branch name should look like this:
`nickname_000_[_bugtrackerTicketNumber][_noteNumber][_description].`
Creating a branch without a second argument will lead to HEAD being the base of the new branch, that is the currently checked out branch.

Listing 4: Branching

```
# Lists all local branches; marks the current working branch with a star
> git branch
# Create a new branch called NEWBRANCH
> git branch NEWBRANCH
# Make NEWBRANCH the current working branch
> git checkout NEWBRANCH
# Switch back to master branch
> git checkout master
# Create a new branch called NEWBRANCH and make it the current working /
    branch
> git checkout -b NEWBRANCH
# Practical examples
> git branch werksfarbe_6790_htmlAndStylesheetClosedFunds
> git checkout werksfarbe_6790_htmlAndStylesheetClosedFunds
> git co -b paddy_6789_123456_extension_vsp_ophirum
```

Branch ex post

Work on a different branch than the current but move the commits to the new working branch. For example: Move entire master branch to `NEWBRANCH` and rebuild the master from origin/master afterwards.

Listing 5: Branch ex post

```
# The flag -m means move branch master to NEWBRANCH
# and do the corresponding reflog (git reflog --help)
> git branch -m master NEWBRANCH
# Get master from remote repository reset the commits
> git checkout -b master origin/master
```

A branch should consist of at least a single commit. The rule is: **commit as often as possible!** Each commit must have a specific description of what the changes are which you can not leave empty. Specific means: „Do not point out the obvious!“ e.g: »made changes«, »this and that« or »...« are counterproductive because you could write that in all you commit messages regardless of what the commit was. Mentioning that a commit has modifications is pointless. That is what a commit is about namely „Making changes in files“. Therefore specify what the changes were about and/or why they were necessary. The more information a commit message has the better.

Listing 6: Commit

```
1 # Commit a single file called FILE
2 > git commit -m'A really good summary of what the commit is about. Feel free/
      to explain what changes are involved and why they where necessary. Name/
      people that suggested the changes.' FILE
3 # Commit all files. Use with caution!
4 # (NB.: omitting the commit message leads git to open an editor)
5 > git commit -a
```

To make the review acceptable commit after each step of your work. Having worked on subject A commit before you move on to subject B. Commits must be made at latest after one hour of work on source files. There should not be more than ten modified files in a commit. An unlimited number of new files may be committed though. Avoid forcing the reviewer to scroll unnecessarily when inspecting a commit. Do not mix new files and modified files in one commit if it can be avoided. It is better to make a commit for all new files and explicitly mention the fact that you added files or an extension in the commit message.

Correct a commit message **NB.:** Do not use the amend option after a push. The option is like doing a `reset HEAD 1` and git commit. The commit will

not be edited but made new. But after a push there is still the old edit. Without the force option there will be no chance of continuing. This especially bad if somebody pulled in the meantime.

Listing 7: Correct a commit message

```
1 > git commit --amend -m'your new message'
```

2 Advanced

Listing 8: Get an overview of remote branches

```
# Get the remote changes
> git fetch
# Show all branches
> git remote show origin
# Show all known branches
> git branch -a
```

2.1 Merge conflicts

On merge conflicts listing 9 there are mostly three stages of the files involved: "base" 1, "ours" 2 and "theirs" 3 see listing 10. Files without merge conflict have only one stage stage namely 0. Sometimes there is no "ours" or no "theirs". That happens when the file is not existent in one repository. Base is the common source of ours and theirs. Ours is the version in the remote repository (github). Theirs is the local version we are working on (the developing server).

Listing 9: Merge conflict after pull rebase

```
# Not currently on any branch.
# Unmerged paths:
#   (use "git reset HEAD <file>..." to unstage)
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       both modified:    FILE

# NB: during rebase the HEAD got "detached" (see above output)
> git branch
* (no branch)
  master
...
```

The stages of a file and their content may be displayed as shown in listing 10 from left to right there are: file permissions, the SHA1, the stage and the filename of the file.

Listing 10: List file status

```
> git ls-files -s FILE
100755 b42be60e564a0f9a948d08b37fec6ec603793d7e 1       FILE
100755 2c3b3cac7dc0275c073edaae7cee5dd90c90f210 2       FILE
100755 8bf1e221687ce7ec48b180a2f0880239ce34455b 3       FILE
# View the FILE in the different versions
> git show 2c3b3cac7dc0275c073edaae7cee5dd90c90f210 #"ours"
> git show 8bf1e221687ce7ec48b180a2f0880239ce34455b #"theirs"
```

In listing 11/listing 12 we see an example how you can hand merge a file that is stale locally (Home is still present in "their" file) but already has changed in the remote origin (Home has been removed in "our" file).

Listing 11: Example of merge conflict

```
...
+<<<<<<< Updated upstream
    <li class="first-item"><a href=""></a></li>
+=======
+   <li class="first-item"><a href="">Home</a></li>
+>>>>>>> Stashed changes
...
```

By editing the file by hand as shown in listing 12 the preferred version is restored.

Listing 12: Example of merge conflict resolved

```
...
    <li class="first-item"><a href=""></a></li>
...
```

Alternatively, because obviously "our" file is needed, that version may be fetched as shown in listing 13.

Listing 13: Example of merge conflict resolved alternative

```
> git checkout --ours FILE
```

In another example a merge conflict has happened in on of the following files: .htaccess, db_script, robots.txt or typo3conf/localconf.php listing 2. In order to keep the version that is running on the developing server and discard remote changes `git checkout -theirs FILE` is used.

Mergetool

After solving the conflicts or as a conflict resolution mergetool is run. Basically when started it needs some resolution tool. The default can be set in the .gitconfig file in the users home directory. In fig. 2.1 a typical resolution Situation is shown.

The fig. 2.1 shows the situation after the local file has been edited. Saving the file and quitting all buffers will tell git the merge was successful.
If ours or theirs was checked out beforehand and the file therefore is not saved in editor vimdiff git asks you to confirm the merge success as shown in listing 14.

Listing 14: Closed the editor without saving

```
>git mergetool
Merging:
css.css

Normal merge conflict for 'css.css':
  {local}: modified
  {remote}: modified
Hit return to start merge resolution tool (vimdiff):
4 Dateien zum Editieren
```

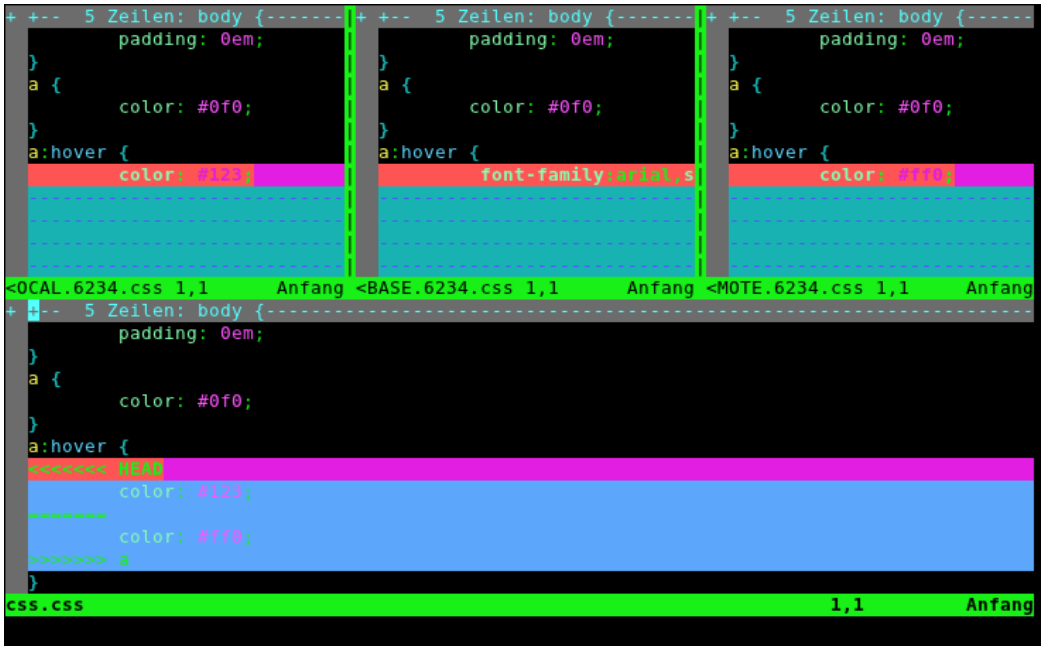

Figure 1: Diff view with base, ours, theirs and current file

Figure 2: Edited the file

```
10 css.css seems unchanged.
11 Was the merge successful? [y/n]
```

2.2 Finding branches knowing the commit SHA1

Listing 15: Finding branches knowing the commit SHA1

```
1 > git branch --contains 864afa58a82c58e4367ac4e2b70b57d667494a8c
2 * master
3 > git branch -r --contains 0fc352ba9abea4d9ec8e
4   origin/werksfarbe_0008632_umbau_der_tabs_auf_first_child
```

2.3 Config file

Use the `help config` command to find out about all the settings.

Listing 16: Configuring git

```
1  > git config --global user.name 'Patrick-Emil Zoerner'
2  > git config --global user.email 'zoerner@vsp.ag'
3  > git config --global github.user paddyez
4  > git config --global github.token 0123456789yourf0123456789token
5  # Colors for black background and green as font color
6  > git config --global color.branch.current 'yellow bold'
7  > git config --global color.branch.remote 'cyan bold'
8  > git config --global color.diff.new 'yellow bold'
9  > git config --global color.diff.old 'red bold'
10 > git config --global color.diff.meta 'cyan bold'
11 > git config --global color.diff.frag 'white bold'
12 > git config --global color.diff.commit 'white bold'
13 > git config --global color.status.added 'yellow bold'
14 > git config --global color.status.changed 'cyan bold'
15 > git config --global color.status.untracked 'red bold'
```

Alternatively and if you know what you are doing you can edit the file .gitconfig.

Listing 17: .gitconfig

```
[user]
        name = Patrick-Emil Zoerner
        email = zoerner@vsp.ag
[github]
        user = paddyez
        token = 0123456789yourf0123456789token
[merge]
        tool = vimdiff
[diff]
        color = auto
[alias]
        st = status
        ci = commit
        co = checkout
        br = branch
        di = diff --ignore-all-space
[color "branch"]
        current = yellow bold
        remote = cyan bold
[color "diff"]
        new = yellow bold
        old = red bold
        meta = cyan bold
        frag = white bold
        commit = white bold
[color "status"]
        added = yellow bold
        changed = cyan bold
        untracked = red bold
```

2.4 Make non git repository a git repository

```
git init
git remote add origin git://git/vsp.ag
git pull --rebase origin master
```

2.5 Install a git public repository on debian

Listing 18: Installing

```
> sudo apt-get install git-core gitweb
> sudo mkdir /var/www/git
> [ -d "/var/cache/git" ] || sudo mkdir /var/cache/git
> sudo vim /etc/apache2/sites-available/git
--8<--
<VirtualHost *:80>
        ServerName      git
        DocumentRoot    /var/www/git/
        <Directory /var/www/git>
                Allow from all
                AllowOverride all
                Order allow,deny
                Options ExecCGI
                <Files gitweb.cgi>
                        SetHandler cgi-script
                </Files>
        </Directory>
        DirectoryIndex gitweb.cgi
        SetEnv  GITWEB_CONFIG  /etc/gitweb.conf

        CustomLog       /var/log/apache2/git_access.log combined
        ErrorLog        /var/log/apache2/git_error.log
        RewriteLog      /var/log/apache2/git_rewrite.log
        RewriteLogLevel 3
</VirtualHost>
-->8--
> sudo a2ensite git
> sudo cp /usr/share/gitweb/* /var/www/git
```

```
29 > sudo cp /usr/lib/cgi-bin/gitweb.cgi /var/www/git
30 > cd /var/www/git
31 > ln -s /usr/lib/cgi-bin/gitweb.cgi index.cgi
32 > sudo vim /etc/gitweb.conf
33 --8<--
34 # path to git projects (<project>.git)
35 $projectroot = "/var/cache/git";
36 # directory to use for temp files
37 $git_temp = "/tmp";
38 # target of the home link on top of all pages
39 #$home_link = $my_uri || "/";
40 # html text to include at home page
41 $home_text = "indextext.html";
42 # file with project list; by default, simply scan the projectroot dir.
43 $projects_list = $projectroot;
44 # stylesheet to use
45 $stylesheet = "gitweb.css";
46 # logo to use
47 $logo = "git-logo.png";
48 # the 'favicon'
49 $favicon = "git-favicon.png";
50 -->8--
51 > sudo etc/init.d/apache2 reload
52 > cd /var/cache/git/
53 > mkdir vsp.ag
54 > cd vsp.ag
55 > git init
56 > git config --bool core.bare true
57 > echo "VSP AG & Fondsvermittlung24.de" > .git/description
58 > git commit -a
59 > touch .git/git-daemon-export-ok
60 > git daemon --base-path=/var/cache/git --detach --syslog --export-all --/
      enable=receive-pack
61 > git clone git://git/vsp.ag vsp.ag
```