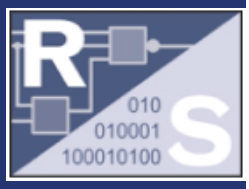
RS - git beginner guide

TECHNISCHE
UNIVERSITÄT
DARMSTADT



Inhaltsverzeichnis

1 Rechnersysteme git user guide

Git is a enormous powerful tool to manage the versions of a project. Whenever handling big projects the same problems are appearing over and over again. Common ones are:

- I messed up my program how to revert the changes to a state where it worked?
- My laptop broke and my latest copy on the USB stick is 2 weeks behind or I don't have a backup.
- Me and a friend are working on a program how can we synchronize our project without always sending emails back and forth?

Those problems vanish if one uses git correctly. Git lets you:

- manage different states of a project
- quickly rewind your project or single files to a prior state
- try new features on a separate branch
- backup the project on a server
- synchronize hundreds of people working on the same project
- abandon your USB-Sticks and relief your email for distributing files

The following guide should show you basic things that can be done with git.

1.1 generate rsa-key

For authentication against the git Server you need to create a RSA key. The key consists of a private and a public key file. The public key has the file ending .pub. Generate a key-pair with:

```
ssh-keygen
```

It is recommended to secure the private key with a password.
Send only the public key to your supervisor. He/She will grant you access to the RS-git and give you rights to create your own branch.

1.2 configure git before first use

For configuration, tell git and all other people working on the project who you are. This allows to track who changed what. For this purpose, open a terminal and execute the following name with your name and email:

```
git config --global user.name "Name Surname"
git config --global user.email me@email.com
```

1.3 clone a repository

The first thing when entering a project is to clone it from the server. Therefor we open a terminal and look which repositories we can download:

```
ssh git@git.rs.tu-darmstadt.de
```

This would return a list with the repos you can download and the level of access you have on it.

```
PTY allocation request failed on channel 0
hello , this is gitolite 2.2-1 (Debian) running on git 1.7.9.5
the gitolite config gives you the following access:
    @R_ @W_     playground
     R   W      test2
     R   W      testing
Connection to orion closed.
```

We now pick the repo playground and download it into the current directory with:

```
git clone git@git.rs.tu-darmstadt.de:playground
```

1.4 first sneak into the repo

Let's go into the directory:

```
cd playground
```

Since we are now in a git repository, we can make use of all fancy git commands. First let's have a look at what happened whith the repo in the past:

```
git log
or
git log --graph --all --pretty=format:"%Cblue%h%Creset [%Cgreen%ar%Creset]
% %s%C(yellow)%d%Creset"
```

This shows you the past commits as well as how the branches developed and the unique commit identifier.

1.5 creating a branch

We will now create our own branch and copy of the current workspace. The command

```
git branch -a
```

shows the different branches that we have on our repository. Branches with “remote/origin” upfront are branches on the git-server. All other branches are on the local filesystem. Entering

```
git branch mynewbranch
```

will create a new local branch as a copy of the current branch (check with “git branch -a” command). You can stitch to this branch with

```
git checkout mynewbranch
```

again checking with “git branch -a” you will see that the star moved from master to mynewbranch. Alternatively the command

```
git checkout -b mynewbranch
```

will create and switch to the new branch in one step instead if two.

1.6 Creating files and add them to version control

Now we will add a file to the new branch with

```
echo "Test">test.txt
```

we created a file called “test.txt” containing the text “Test”.
With

```
git status
```

we can check the status of the repo. Now git tells us that have untracked or unmanaged files. In order to make git aware of the new file we will have to add the file to the index with

```
git add test.txt
```

This will only tell git that we have a file where we want to do something with. The file is now under Version control yet. However if you accidentally added a wrong file you can undo this by

```
git reset HEAD test.txt
```

Now that the file is in the index “git status” tells you what to do next:

```
git commit -m "Added test file"
```

will put all added files under version control in your local repository. Always make sure to state a useful message that you and others are able to see what changed in this commit. So we have now added a new file to our branch. If we now execute

```
git checkout master
ls
git checkout mynewbranch
ls
```

you can see how the files are automatically changed in the filesystem, depending on the branch we're currently on.

1.7 Pushing changes to the server

Whenever you like to share the current state of the project with others or just backing up your project on the server, a push is the right thing to do. When we have just created a new branch we'll have to upload this branch to the server with:

```
git push —u origin mynewbranch
```

If you now execute "git branch -a" you'll see a remote branch on the server called mynewbranch and one local. For all other following files on that branch it is sufficient to execute: (after add/commit)

```
git push origin
```

1.8 Getting updates from the server

When you developed on a branch where also other people are working on, you should update to the newest version from time to time and especially before you push to the server. This doesn't happen automatically. Others might have pushed files to the server repo copy and you have still an old version of the files locally. You can now do a

```
git fetch origin
```

that checks the changed files at the server since your last checkout. Afterwards you will execute

```
git merge /remotes/origin/your current branch
```

The fetch and merge command is actually only on the branch you're currently working on. Each branch needs to be updated manually but this can be done in one step instead of two with:

```
git pull origin
```

1.9 recovering a previous state

There are many different ways to go back to a previous state for either one file or the whole project. Some of the most common ones are described here.

1.9.1 revert complete repo to a previous state

One possible scenario might be that you have changed files and committed a few times and now you see that the thing you're trying to do could have been done much simpler. So you would like to go back to a certain state and start over again but not deleting the current progress.

So we go back to our example and create some new files on "mynewbranch":

```
echo "Another test">newfile.txt
echo "a">>test.txt
echo "b">>test.txt
echo "c">>test.txt
echo "d">>test.txt
```

So we created newfile.txt containing “Another test” and we appended a,b,c,d in test.txt. Finally add the files to the index and commit them. “git log” tells us our current commit history.

```
commit 13a4a0b3b2d61cdec480c896cc7bda99dd67579a
Author: me
Date:   Tue Jan 28 08:47:46 2014 +0100

    another test file

commit d5543ac5b3d3a85cad63c983cdb4abc14e0a06a3
Author: me
Date:   Mon Jan 27 18:22:56 2014 +0100

    Added test file

commit 0c63c6e504b2b7dd4e6c2486a7200a308a7d38ef
Author: me
Date:   Mon Jan 27 14:24:40 2014 +0100

    initial commit
```

Now we would like revert our changes back to a previous state. The state can be identified by the unique commit ID (hash/random number). Thus we type

```
git checkout d5543
```

Git will now warn that we are in detached head mode. This is fine if you would just like to look at the previous state and then go back to the current version. The “git branch -a” shows you that you are on no branch. Take a look at the files that are in the current directory with “ls” and you will see that the folder looks like at the previous commit. You could now go back to the latest version of the project with “git checkout mynewbranch” or further develop from this previous version. For further development you should create a branch based on the current version with:

```
git checkout —b tryout
```

A “git branch -a” tells you that you are on tryout branch.

1.9.2 revert changes on a single file

For showing how to revert a single file we modify our test file and create a new one as well:

```
echo "xx">anotherfile.txt
echo "ff">>test.txt
```

```
cat test.txt
Test
ff
```

We again add the files to the index and commit them. To now revert test.txt to it's previous state we look at "git log"

```
commit 9b0c97d6e37b6d6ad91ecec408b7102b8ec7fae2
Author: me
Date:   Tue Jan 28 10:15:04 2014 +0100

    mod2

commit d5543ac5b3d3a85cad63c983cdb4abc14e0a06a3
Author: me
Date:   Mon Jan 27 18:22:56 2014 +0100

    Added test file

commit 0c63c6e504b2b7dd4e6c2486a7200a308a7d38ef
Author: me
Date:   Mon Jan 27 14:24:40 2014 +0100

    initial commit
```

and revert it one commit before with:

```
git checkout d5543 test.txt
cat test.txt
Test
```

Instead of the ID you could also use a branch name... execute:

```
git checkout mynewbranch test.txt
cat test.txt
Test
a
b
c
d
```

1.9.3 revert a complete branch

To show how to reset the entire branch, we commit the modified test.txt. With "git log" we again look at the past few commits

```
commit 8322ce7df0494d2c511abff4c0ce6ebeb6c16283
Author: me
Date:   Tue Jan 28 10:30:20 2014 +0100
```

```
    next commit

commit 9b0c97d6e37b6d6ad91ecec408b7102b8ec7fae2
Author: me
Date:   Tue Jan 28 10:15:04 2014 +0100

    mod2

commit d5543ac5b3d3a85cad63c983cdb4abc14e0a06a3
Author: me
Date:   Mon Jan 27 18:22:56 2014 +0100

    Added test file

commit 0c63c6e504b2b7dd4e6c2486a7200a308a7d38ef
Author: me
Date:   Mon Jan 27 14:24:40 2014 +0100

    initial commit
```

We would now like to go back to commit named “mod2” and we don’t care what happens to the commit “next commit” since it was crap and we don’t want to keep it anymore. So we execute:

```
git reset ––hard d5543
```

NOTE: The “reset –hard” command should be treated with care since it can not be undone!
The command

```
ls
README test.txt
cat test.txt
Test
```

shows that we have reset the branch “tryout” to the state where we initially started from.

1.10 merge

When it goes towards the end of a project merging becomes more and more important. Merging will integrate your changes into the main branch which is in most cases called “master”.

The merge policy run by RS is:

1. merge changes from the target-branch into the source-branch
2. resolve merge conflicts if any
3. again test the design with the new changes if any

4. repeat step 1-3 if the target-branch has new commits since the last merge

5. merge changes from source-branch to target-branch

NOTE: Usually only your supervisor has write permission to the master branch! With this policy we'd like to ensure that the target-branch always contains a stable running copy.

To demonstrate a merge (with conflicts) we merge the "tryout" branch with "mynewbranch". We now modify the "test.txt" file so that it will produce a merge conflict and the README such that git can automatically merge.

```
echo "a">>test.txt
echo "conflicting line">>test.txt
echo "b">>test.txt
echo "c">>test.txt
echo "silent merge">>README
```

The changes must be added and committed. We're now merging "mynewbranch" into "tryout".
Executing:

```
git merge mynewbranch
Auto-merging test.txt
CONFLICT (content): Merge conflict in test.txt
Automatic merge failed; fix conflicts and then commit the result.
```

results in a conflict like predicted. "git status" gives us a hint which files are conflicting. You can have a look at the conflicting file:

```
cat test.txt
Test
a
<<<<<<< HEAD
conflicting line
=======
b
>>>>>>> mynewbranch
c
d
```

Git already marked where a conflict appeared and which is from which branch. you may now start "nano test.txt" to modify the file how you think it should be or even execute "git mergetool" which makes it easier to resolve conflicts for larger files. We now modify "test.txt" and neither chose "b" nor "conflicting line" for this file rather than "bb" since we think it fits best.
"test.txt" now looks like:

```
cat test.txt
Test
a
bb
c
```

```
d
```

If you inspect the “tryout” branch you will see that it also received “newfile.txt” automatically and “README” changes were kept since “mynewbranch” doesn’t have any updates here.

```
git commit -m "After commit"
```

adds the merge to the Version control.

You should now test if the branch “tryout” is a releasable version (e.g. runs without errors if it’s a program) and then finally merge it into the target-branch “mynewbranch”:

```
git checkout mynewbranch
git merge tryout
```

We now see that the merge succeeded without any conflicts. Usually the “test.txt” line “b” and “bb” would conflict. This is not the case, due to the fact that we first merged “mynewbranch” into “tryout” and then “tryout” into “mynewbranch”.
Since we merged our work from “tryout into “mynewbranch” we don’t need “tryout” anymore and we can remove it with:

```
git branch -d tryout
```

And since the tutorial is over please be so kind and delete everything you have changed in the playground on the server with:

```
git checkout master
git branch -d mynewbranch
git push origin :mynewbranch
```

You can always use the playground repository to tryout things that you are unsure about in your productive repo.