How to work with git

Contents

1 Preface **1**

2 Setup **1**

2.1 Git Extensions on Windows 2

3 Checkout aka clone **5**

4 Commit aka commit and push **6**

5 Remotes and branches **6**

6 Rebase **8**

7 Best practices **9**

7.1 Branches 9

7.2 Commits 9

7.3 Repository content 9

8 Conclusion **10**

1 Preface

This howto is intended to show the basic usage of git. This howto will not go into detail about advanced topics and the internal structure of git. If there is the need to understand it better `http://git-scm.com` has lots of information about this. Of course the bundled documentation can be used as well.

Apart from this there are some things that you need to know when working with branches and synchronising with remote repositories. I explain what happens with the standard git command line interface, but the same rules apply for tortoise git which takes away the need to remember all those commands and for sure simplifies things. Nevertheless I think, especially after working throug the git-svn howto, you need a basic understanding of those things.

2 Setup

The first step is to install git.

On Debian and most other distributions, there are packages like git or git-core. Don't confuse it with the GNU interactive tools which are sometimes also called git.

On windows I recommend using Git Extensions. See the next chapter for further information.

In order to work with a remote git repository, a ssh key is needed to authenticate. This key can be created with openSSH (`ssh-keygen -t rsa -b 2048`) or with putty. A bitsize of 2048 should be sufficient, but you are free to use a higher one. I personally don't use passwords to protect my key, because I consider the key safe on my computer. Also if you use a password, then you have to type it every time you upload your commits.

A little sidenote, the generated key can also be used to log in to the developer machines via ssh which is by the way the desired method. I also propose disabling password logins via ssh on the machines, but that needs to be discussed in a further meeting.

After the key has been generated, the public part has to be put on the machine hosting the git repository, so that it knows your host. You may notice, that the key gives you simple read-write access to the machine, so you could basically upload commits not made by you. This is the intended behaviour as git encourages workflows where one developer uploads reviewed code from other developers. If you need to prove, that a commit was made by you, than you can sign your commits with a pgp key beforehand.

The private key needs to be accessible by the programs using ssh. For openSSH that means `$HOME/.ssh/id_rsa`. Important however is that the putty and openSSH key formats are different, so convert the key with putty to have it in the correct format. The public part must be in openSSH format as the server runs it.

Besides the key a little configuration of your git installation is needed. Every commit is annotated with a name and email address, so these should be set before commiting things.

The git commands are

```
$ git config --global user.email 'my-name@domain'
$ git config --global user.name 'My Name'
```

Global settings are stored in `$HOME/.gitconfig` and can be overwritten on a per repository basis. You can for example assign different email addresses for different projects you work on.

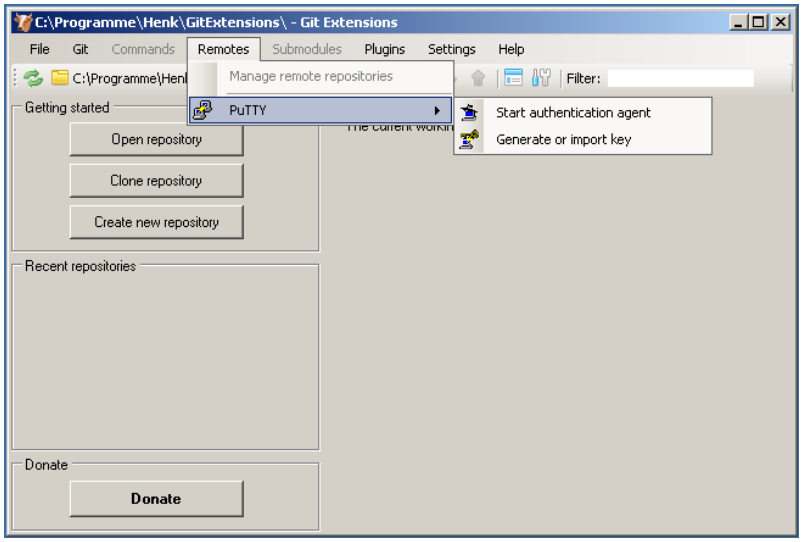
2.1 Git Extensions on Windows

In order to work with Git on Windows, you either have to use the command line tools provided by MSysGit or Cygwin, or use a tool like TortoiseGit or Git Extensions, which are more windows-like. Both of them are based on MSysGit so you can still use command line tools if like to. Since TortoiseGit does not work as supposed, I recommend the use of Git Extensions.

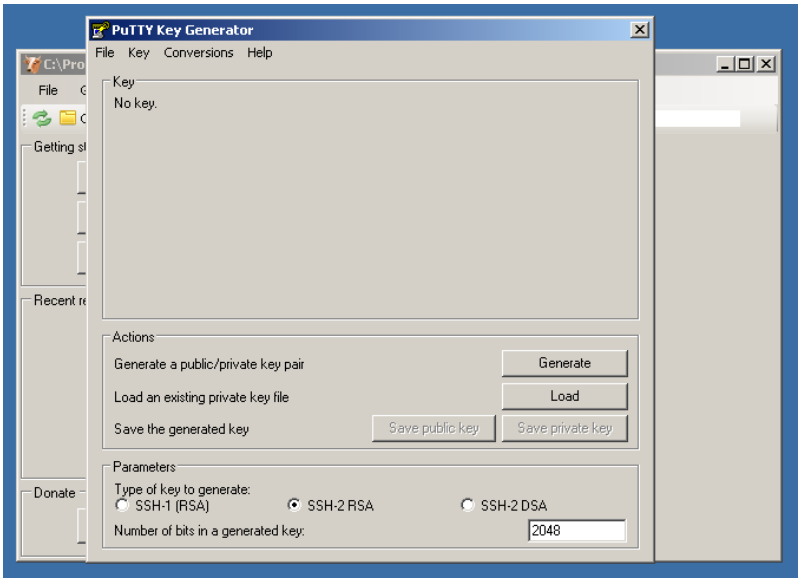
Before you begin, make sure that you have installed the Microsoft dotNet-Framework 2.0. If this is not the case, download it from here `http://www.microsoft.com/downloads/details.aspx?displaylang=de&FamilyID=0856eacb-4362-4b0d-8edd-aab15c5e04f5` and install it.

First download Git Extensions from `http://code.google.com/p/gitextensions/downloads/list`. I recommend to use the complete setup that includes MSysGit and the mergetool KDiff3. Run the installer with default options. You are done.

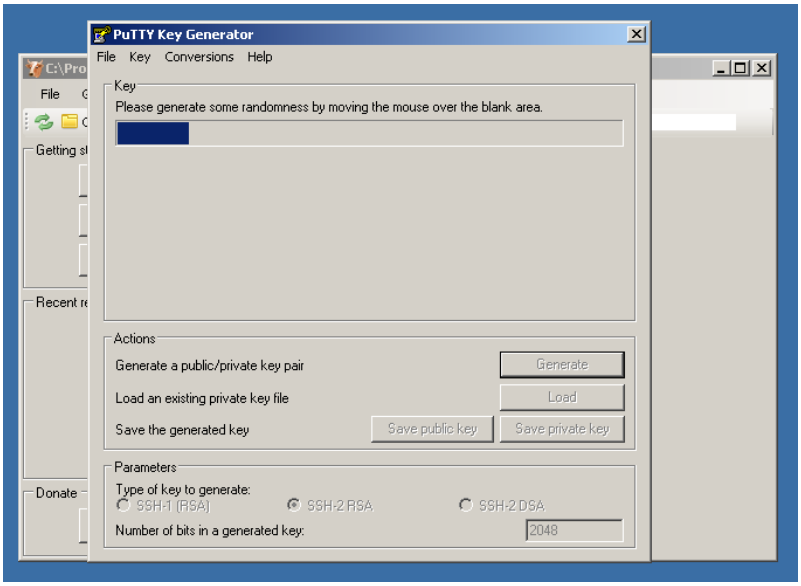
Next you have to generate a ssh key with PuttyGen, that is bundled with Git Extensions. Open Git Extensions and choose "'Remotes"' –> "Putty" –> "Generate or import key".



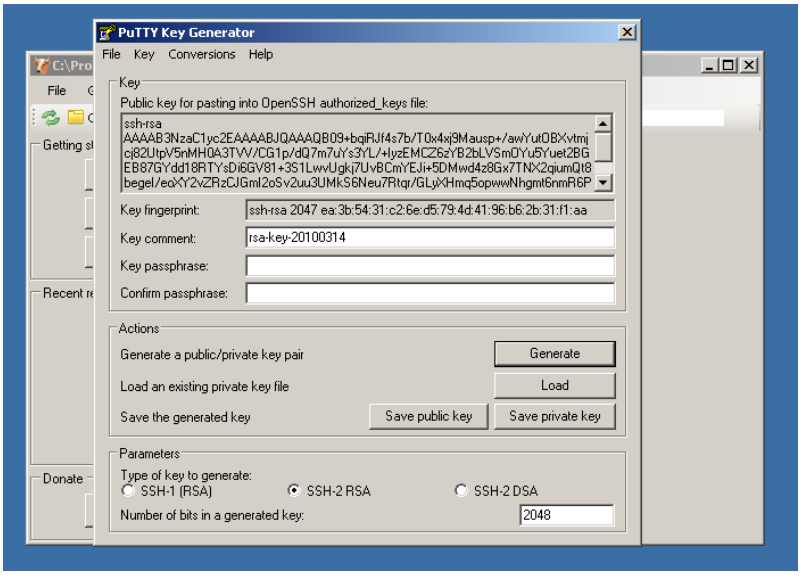
In order to generate a ssh key pair, select "Key" –> "Generate key pair" or click "Generate".



Move your mouse over the blank area to generate some randomness, that is needed for key generation.



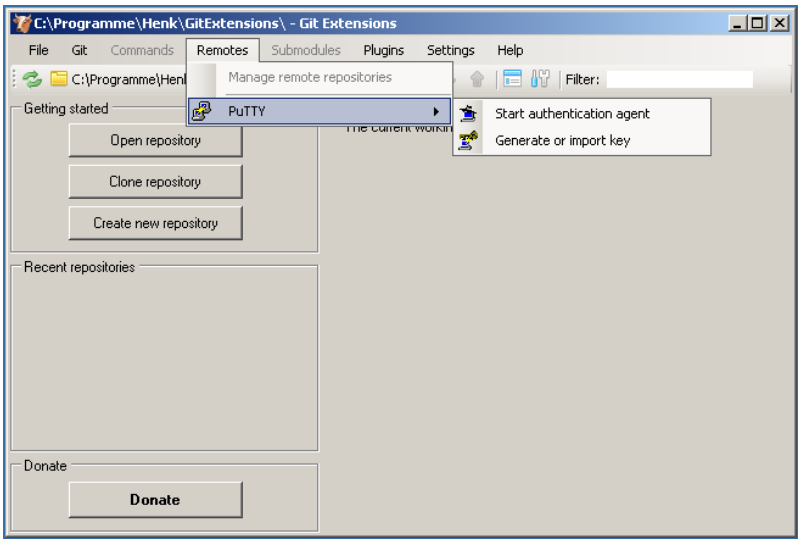
After the key generation process has finished, you will get the following picture. Your public key is shown in the text area on the upper part of PuttyGen. Copy and email it to Tobias, so he can add your public key to the list of trusted keys.



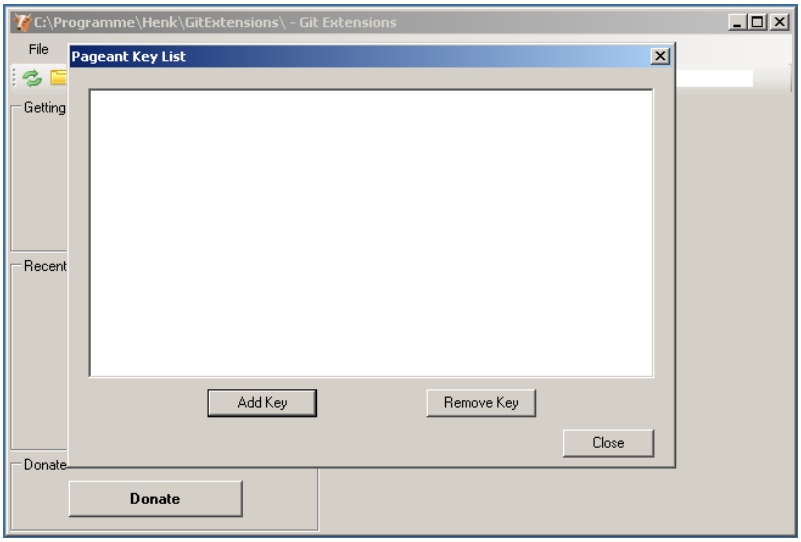
Now you can set a password for your private key. Additionally you can enter a comment. After that save your private key to your hard disk. Click “Save private key”, select a directory, choose a name and save it.

To get access to our remote repository, Git Extensions needs to know about your private key. If you have entered a password for your private key, you should use Pageant for key management, otherwise Git Extensions will handle the private key by itself. Pageant caches your password, so you do not need to enter it every time you access our remote repository.

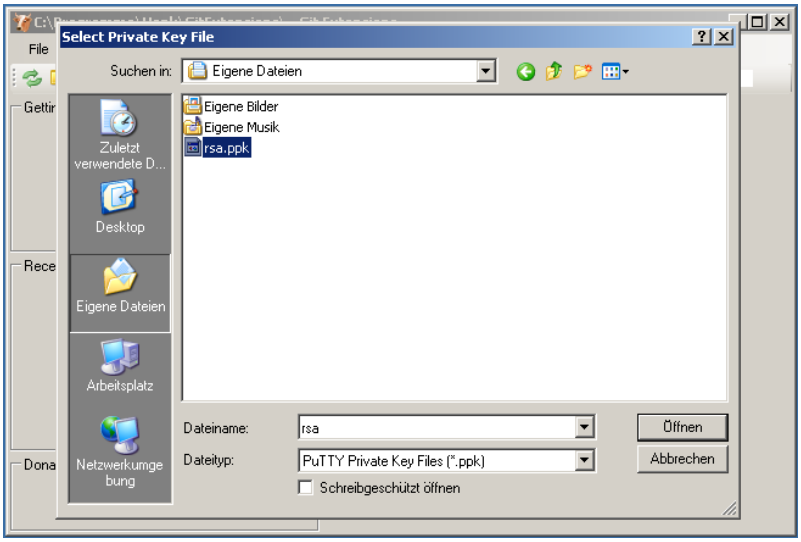
To configure Pageant, open Git Extensions and select “Remotes” –> “Putty” –> “Start authentication agent”.



Pageant will be placed in your tray. Open it and click “Add Key”.



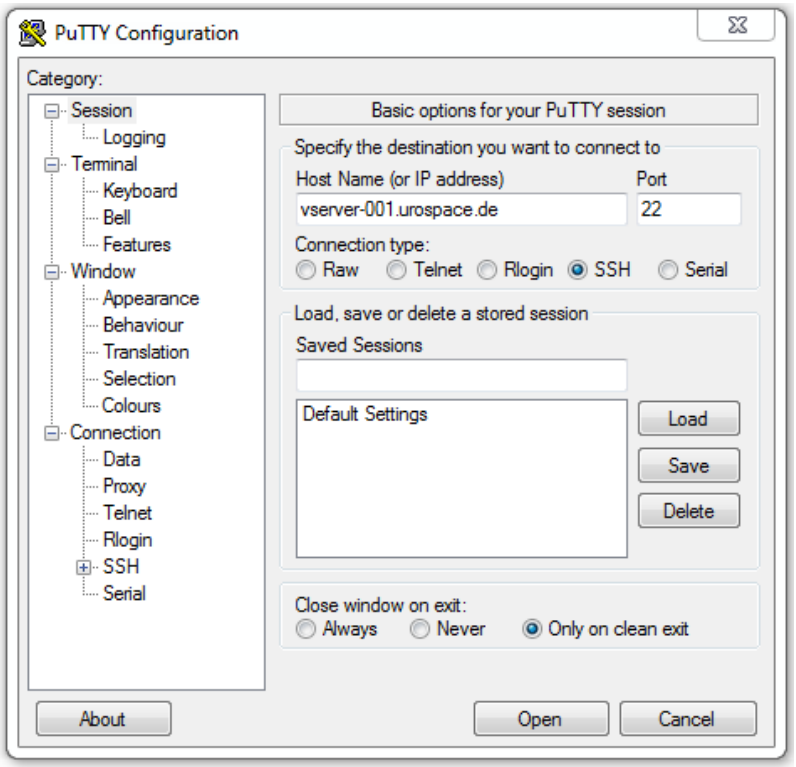
Navigate to your private key on your hard disk, select it, and click “Open”.



Pageant will prompt you for your password. Enter it and click “OK”. Now you can access our remote repository.

In some cases Git Extensions refuses to accept the server’s ssh key. If this happens, you need to connect to the server via Putty once, in order to accept the fingerprint of the server’s ssh key. Putty can be obtained from `http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html`.

Run Putty, enter `vserver-001.urospace.de` as host name and click "‘Open"’. After that you will be prompted to accept the server’s fingerprint. Do so and close Putty. You are done!



3 Checkout aka clone

A checkout in git is called clone, because unlike a checkout in subversion a git clone represents a full repository not just a checkout of the latest revision.

The clone is done with the following command

```
$ git clone git@hostname-or-ip:repository.git
```

This creates a directory repository on the local machine and downloads the remote repository to it. If you like another directory name for your local repository append it to the command. The path specifier above is a shortcut for a ssh url. The first element `git@hostname` says that the repository lies on hostname and the login is done with

the user git. The user depends on the access rights system on the remote machine and is in our case always git. For the curious, scripts on the server side give your ssh key access to login via the user git and after login test if you have read and/or write access to the repository.

The second part of the url, `:repository.git`, says i want the directory repository.git which is located relative (the colon) to the users home directory. Again for the curious, this url schema also works for ssh logins and scp operations. In the case of git clone, the colon resolves to `$HOME/repositories` instead of `$HOME`.

4 Commit aka commit and push

A commit in git is a subset of a commit in subversion. If you commit data you basically save the differences to the parent commits (in git there can be more than one parent) and annotate it with a commit message. At this stage, the commit is only present locally in your copy of the repository, whereas in subversion the commit is also uploaded to the remote repository. You upload your new commits via git push, which gets more attention in a later section.

If you just want to make a change and publish it directly afterwards you can use the following workflow.

```
$ # make changes in the repository
$ echo "Hello World" > README
$ git status # show changes in the repository
# On branch master
# Untracked files:
# (use "git add <file>..." to include in what will be committed)
#
# README
nothing added to commit but untracked files present (use "git add" to track)
$ # add change to repository. important: not file is added, but the change in the file
$ # also use add for files already tracked by git
$ git add README
$ # create commit with message. if no message is given, a editor is launched to write it
$ git commit -m "created README file"
$ # upload commit to our remote repository. its always a good style to specify the remote
$ # and branch you want to push to. if you know what you are doing, you can omit this
$ # information if your local branch tracks a remote branch.
$ git push origin master
```

5 Remotes and branches

Remotes are propably the most import difference between a centralized vcs like subversionand git. In git you have a complete repository on your local machine. This is basically all you need to work in the repository. Subversion on the contrary consists of a local checkout which is connected to a server which hosts the repository. Keeping everything on your local machine is however not satisfactory if you develop in a team, so a central server, like in subversion, is needed to exchange your commits. Given, that such a repository server exists, you can add its location to your repository with

```
$ git remote add origin git@hostname-or-ip:repository.git
```

This command is implicitly executed when you clone a repository. The origin parameter is an alias for the remote host. You can choose whatever name you want here, origin is just the git default when cloning. It is now somewhat obvious, that a local repository can be connected to several remote repositories. When you go through the git-svn howto you can see that even other repository types can be connected to your local copy.

Now that the remote is connected you can ask it for changes. There are several ways to do this in git, because the high level commands are based low level commands. I will present two high level ways of doing it here. The first method uses the git remote command. This command has the advantage, that it doesn't modify local branches. It can therefore be safely executed. The command is

```
$ git remote update
```

If you want the new commits in your local branch, you must merge or rebase it with the remote branch. These topics are discussed in a later section.

Branches have already been mentioned. Now some details on what they are and what you can do with them. At first a short explanation what a branch is in subversion and what is meant to be done with it. Subversion usually has a three directory structure with the directories trunk, tags and branches. If you come to the point where you want to trie out things, for example replace the gui of your program with a different library, you will want to do it somewhere different than in the mainline development for obvious reasons. If you make this change alone you can copy your working directory and test your changes outside of the repository. Now let's say the gui experiment should replace the old later you are confronted with the tedious task of backporting your changes by hand. Even worse if you develop the new gui in a team. You have to stay in the repository or you are in big trouble distributing your changes. This is where the branches concept comes in. Subversion has the branches directory where you can create a copy of a revision and work on it with all the advantages of subversion. But still this is much better, some problems remain. You are working on a copy with no previous history, so backporting is still a mess. You are duplicating the working tree and thereby all of the files in it. In the case of binary libraries, this can be quite a lot of space not to mention the slowdown of the subversion repository.

When examining git branches the picture looks a bit different, thus the purpose remains the same. Branches in git are just pointers to a specific commit. So when a branch is created only a pointer to a commit object is created and no content is duplicated in the repository. Git has various commands to deal with branches as this one of its biggest advantages.

Branches can be created with the following command.

```
$ git branch new-branch-name
```

After the branch has been created it needs to be checked out, so that the working directory reflects the selected branch. For a branch change to complete, the working directory must be clean, meaning no tracked files must be modified. You can check this mit git status and reset the working directory with `git reset --hard` (see manpage for details).

```
$ git checkout new-branch-name # checkout existing branch
$ git checkout -b newer-branch # create and checkout branch
```

You can inspect your commit history and branches with `gitk --all` or the tortoise git builtin history viewer. When you do this you will notice that there are branches named like origin/master. These are remote branches. Remote branches mark the point in the commit history where a branch is pointing to on a remote repository. The can't be checked out like local branches, because changes should be made to local branches and then synchronised with the remotes. If you want to checkout a remote branch where no local equvivalent branch exists, you can create one with

```
$ git checkout -b local-branch origin/remote-branch
```

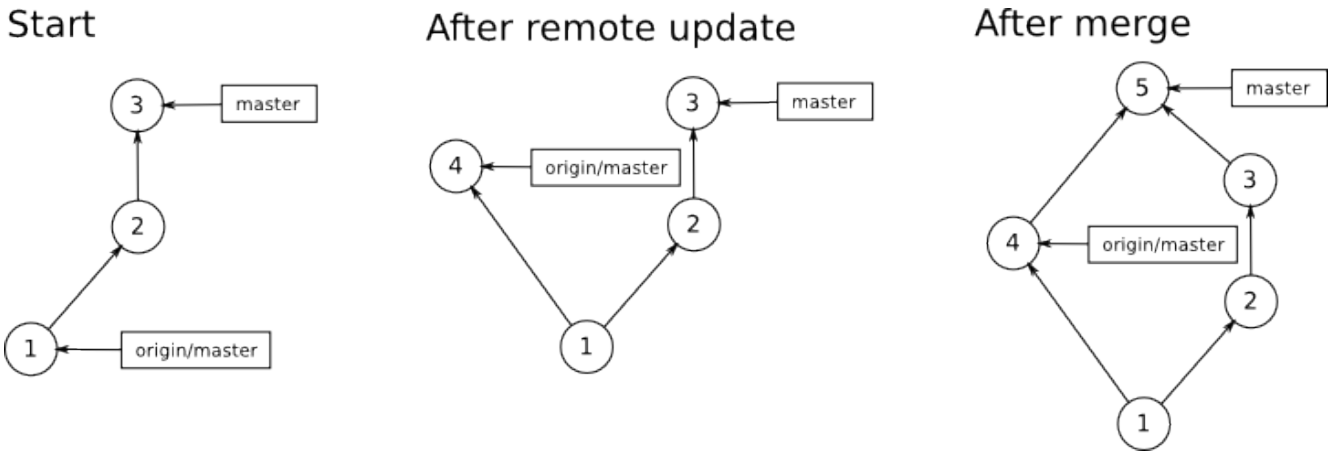
So now that branches are explained we can move on to synchronise our branches with the server. As mentioned before, there are several ways for doing this and one command that gets changes from the server has already been presented. Now suppose we worked on a local branch and some developer notifies us, that he wrote a patch and we should get it. Now consider this all happens in the master branch. We then see the remote master and a couple of commits we made recently on top of it. At the tip of the commit path we see our local branch pointer. We know, that the patch of the other developer also consists of at least one commit. That makes the commit the remote branch points to a fork point in the history. The fork would be revealed when we issued git remote update, but we want `git pull` do the dirty work for us. Now what needs to be done is to merge the two forks by creating a new commit, a merge commit, on top of the two commit paths. The merge is done with git merge origin/master, but as with the previous command we let `git pull` do the dirty work. Now that we know what happens behind the scenes we can issue our synchronise command and thereby update the information about the remote repository and merge our local branch with the remote branch. The command is written as follows.

```
$ git pull origin master
```

The remote and branch names can be omitted if the local branch is configured as a tracking branch. I don't want to explain this further, as it is safer to just supply the parameters when doing a pull. The same also applies for push by the way.

To better understand what the commit graph looks like, a picture has been included. The commit numbers are in fact sha-1 hashes in git, so that they don't conflict when they are created on different machines.

At some point you will want to share your changes with the other developers. Our starting point in this case might be a situation as it can be seen in the third graph of the picture. The situation can be described as follows. The local repository has the latest changes from a remote repository and local commits exist which haven't been uploaded. It thereby doesn't matter, whether there is a complicated merged graph like in the third graph or a simple commit path as in the first graph. You only have to execute the following command to upload you commits.

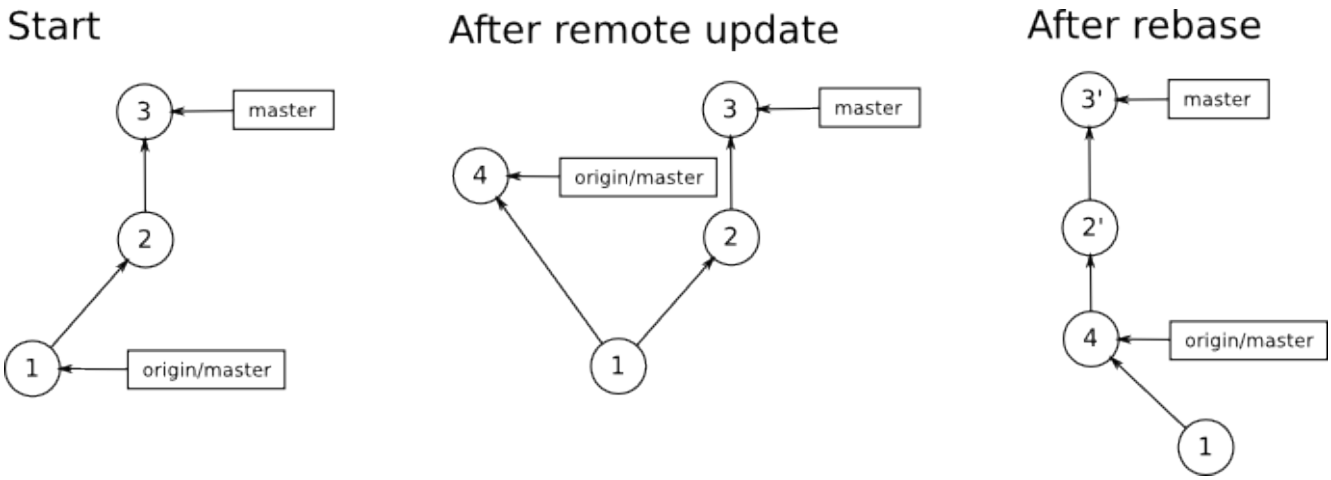



```
$ git push origin master
```

6 Rebase

So far only concepts which, to some extent, also exist in other version control systems. The rebase command is something not present in a centralised VCS. It lets you rewrite the existing history. As it turns out, this is quite helpful in a lot of cases. The most important use case is to prevent merge commits when combining local and remote branches as in the example above.

When going over the workflow of push again, the merge step can be replaced with rebase. As the name rebase states, it rebases the commit history onto another commit than the one it was previously rebased. To do that, all commits after a certain fork point need to be rewritten. A picture can say more than a thousand words, so let it speak for itself.



As a result of the rebase command the local commits are now based on the commit (4) from the other developer and the commit history is linear which has the advantage of being much simpler to understand than one which has lots of different paths. The commits are rewritten and therefore get new commit numbers, marked with ' in this example.

Exactly as merge, rebase is available in `git pull`. You can pull changes and rebase instead of merge with the following command.

```
$ git pull --rebase origin master
```

Just to be complete, the pull command is roughly a combination of the following commands.

```
$ git remote update
$ git rebase origin/master
```

The most important thing about rebase is that it rewrites your history and is therfore potentially dangerous. In the default use case you should however be relatively safe. The reason why it is dangerous is that you can modify commits you received from the remote repository. When you upload those commits again and another developer

wrote new commits on top of the old ones, he gets problems when downloading the new versions of the commits. Just don't forget that.

7 Best practices

Now that the basics have been pointed out, the rules how to work with the new concepts can be defined.

7.1 Branches

Branches are the first topic I want to discuss. The most important point with branches is "use them". The mainline development always happens in the master branch while new features are developed in separate so called feature branches. So always when you want to develop something new, open a branch and do it there. If you want you can push those branches, so that other developers can work with you on the same feature. If your feature is complete you can merge or rebase it (I recommend rebase in most situations) with the master branch and then delete the feature branch. By moving the development away from master you can be sure, that the master branch always contains a version that runs.

The same applies for bug fixes you write. Imagine a a very complicated bug, that propably needs several commits to give them a meaning, then a feature branch is the right place to do it. Now think of that you are writing a bug fix and can't finish it right now. With branches you are free to change the branch and do something else before finishing your work. With rebase you can even add new developments from other branches to your new feature or bug.

7.2 Commits

When creating commits there are a few simple rules to follow.

First of all write expressive messages. Something like "new feature added" really doesn't make its point. I personally like to write a short line of what I did, then leave an empty line and write a short text if the change needs more explanation. If your commit contains more than one change write all in the head line and write several blocks which describe the change.

The next thing is to be sure, that your commit still compiles. Git has a command which can help you find the commit which introduced a bug by walking over your commits. You can mark a commit as good or bad after executing some tests. This obviously only works if your code compiles. But there are other reasons as well why you would want a certain commit to compile.

7.3 Repository content

This point clarifies what belongs to a repository and what not. Version control systems (VCS) are also called source code management (SCM) which makes it pretty clear for what it is intended. When I download a software projects source code, I expect that there is the source code, build scripts and, if it is not possible in any other way, some bundled libraries. I don't expect that there are compiled versions of the source code, architecture documentation or anything else. As a rule of thumb, the repository should contain only files which can't be autogenerated or are modified after their automatic creation, which makes them build from hand.

Some build tools can download needed libraries from the internet, so there is no need to put them into the repository. Maven for example is capable of doing that. No lets say you have a library maven can't download, because its proprietary, then look at other ways to include it into your repository. A simple jar library would be installed into the maven repository by the developer or deployer. Instructions on how to install the library would be written in an `INSTALL` file located at the project root directory.

Back in the days when ant was a popular java build tool, you couldn't load all needed libraries from the internet with your build tool. Downloading everything by hand would be a terrible job, so I came up with a small extension to ant which loaded all libs by itself. What I try to say is, that there is always a way around this problem. The size of your repository will thank you.

As I already stated, build products don't belong into the repository. From time to time you might accidentally check in a class file or something else. To prevent such human failure, you can state that certain files or directories will never be committed. In git you can use an ignore file for this task. Create a file called `.gitignore` at the root of your repository and add the paths you want to exclude. If for example maven creates a target directory with class files, you can write /target into the .gitignore file. For details on what you can exclude look at the git documentation.

Last but not least the project documentation. It surely doesn't belong into the project repository, but a repository can be a good solution to save such things. The simple solution is to create another repository which contains only documentation. With git you can keep your documentation in sync with your project versions. If you delete a file because it isn't needed anymore for the next version, it is still available in older commits and easily reachable with a tag or a branch.

8 Conclusion

There are quite a lot git style guides out there. The most prominent projects which have one are the linux kernel and KDE. If you need inspiration on a better workflow don't hesitate to look at one.

At last I want to present a negative example where the rules from above were not applied correctly. The result is mostly a twisted history, lots of merge commits an commit messages that force you to look at the diff to see what happened, but see for yourself. The code can be found at `http://git.electrologic.org/repo/prongx.git`. If you want to see a good example download the linux kernel (it is not that much as git compresses extremely well) and take a look how they do it.