Table of Contents

Introduction#

GIT offers a wide range of possibilities. As some like to describe it: "Git is a version control Swiss army knife." While this functionality is there to help the developer, it is not always without danger to use it. As we all know, a swiss army knife can leave ugly scars as well.

Depending on how intense you've used git before, your experience level will be different from that of your colleagues. git is very complex in that sense that some of its features can have a impact for others and that it isn't always easy to recover from these situations. The goal of a VCS is to support us (developers) in our day to day work to manage our sources in the most efficient way as possible. That doesn't imply that everyone needs to be a git expert, but everyone should at least understand how to use git to support this goal.

Being used to svn we established a way of working that suits us very well and let's us do our work as optimal as possible. What we will do here is transfer that knowledge into the "git language" and adding some extra GIT flavors to make it even better. This is not a GIT manual replacement. If you need more info, then first read the manual (or ask), but more as a "how-to to do things with GIT that were used to do with svn".

So, below we'll keep a list on how to translate certain actions from svn to git If you think there is something missing: please throw in the group. This thread is considered as open, so throw your suggestion into the group, we'll (all) asses it, check if it has any side effects and then you can add your query here with an explanation how to use, how it can help, and where to watch out for.
Conclusion: use the guide below to make things done using git. Avoid the "stackoverflow copy/paste/execute" pattern at all cost, since most of the solutions presented come at a cost. Unlike subversion, you can make your life (and that of others) difficult pretty fast if you don't know what your doing. If you doubt certain things, have questions or want to do it on a different way: ask first. After that, don't forget to modify or add you experience here.

Setup#

Install git#

First things first, if you haven't got git installed, typing "git" in the terminal returns with command not found:

```
sudo apt-get install git-core
```

You also might want to install a git viewer which you can use to view commits, see diffs of changes etc. "gitg" is such an example;

```
sudo apt-get install gitg
```

Important: some of the configuration options below only work for git 1.8+. Make sure you have at least:

```
koen@koen-HP-EliteBook-8570w:~$ git version
git version 1.8.1.2
```

This should be no problem when on ubuntu 13.04. If you have an older version of ubuntu you can easily compile git yourself:

(taken from: https://www.digitalocean.com/community/articles/how-to-install-git-on-ubuntu-12-04)

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install libcurl4-gnutls-dev libexpat1-dev gettext libz-dev libssl-dev bui
wget https://git-core.googlecode.com/files/git-1.8.1.2.tar.gz
tar -zxf git-1.8.1.2.tar.gz
cd git-1.8.1.2
make prefix=/usr/local all
sudo make prefix=/usr/local install
git version
```

If you're new to git or want to fresh up your memory read the first chapter: http://git-scm.com/book/en/Getting-Started If your already familiar then maybe just read the setup part: http://git-scm.com/book/en/Getting-Started-First-Time-Git-Setup.

Btw; the link above points to the free git book. You can read additional chapters if you want or use them as a reference later on. The git book can be downloaded in PDF and other formats: http://git-scm.com/book

Configure git#

For setup you should have at least done something as:

```
git config --global user.name "Firstname Lastname"
git config --global user.email firstname.lastname@hp.com
git config --global color.ui auto
```

These are global settings, we assume your doing this on your HP developer machine. There are also --local so that you can use different settings for each git repo (eg. you work on multiple projects where your user is different). The user.name is not used for authentication. We are using our private/public keypair together with SSH for that, together with finer grained control using gitlab.

Next also do:

```
git config --global push.default simple
```

More info: https://www.kernel.org/pub/software/scm/git/docs/git-config.html (search for push.default)

With the "push simple" we basically say that on a "git push" git will only push the branch in which you are currently on and that the local branch name must match the remote branch name. These are the sane defaults we want. You normally don't want to push other branches along and there is normally no reason why a local branch should have a different name then its remote.

Generating key pair for ssh access#

git has its own protocol for doing things (the git protocol). While perfect for a closed environment, it's not usable publicly since it has no out of the box authentication support and so on. Because of the git protocol is tunneled in another protocol, most used: http(s) or ssh. We are using ssh. The ssh server is setup using public/private keypair. This means you need to generate a public/private keypair and install the public key on the server so it can allow you access. To do this, generate a keypair first:

```
koen@koen-HP-EliteBook-8570w:~$ ssh-keygen -t rsa -C "firstname.lastname@hp.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/home/koen/.ssh/id_rsa):
```

Default location is OK (hit enter)
Next enter a password, and your private key is stored in ~/.ssh/id_rsa. Your public key in ~/.ssh/id_rsa.pub

Note: your not obligated to enter a password. However, this password is extra security. If someone is able to copy your private key they will gain access to whichever server the public key is installed to without having to enter any password. But don't worry, you'll only have to enter the password the first time you access the remote. After that your key will remain unlocked (unless you restart/logout). Also, your private key together with the password is part of the entire security. Leave your private key in your home folder with only read rights for your own user. Never copy the private key on removable media and such.

Note: if you just have one public/private keypair in the .ssh folder nothing else needs to be done. If you have multiple, you can create a file called "config" in the .ssh directory. Inside you can specify wich key is to be used with which host.

```
Host xyz
    IdentityFile ~/.ssh/some_other_private_key
```

Next you'll have to upload your (PUBLIC!) key to the server. You do this via gitlab. Logon to gitlab, click on the icon that has "your profile" as caption (left from search box on the top right). On the right bottom you'll notice the button: "Add public key". Open the public key file using a text editor (~/.ssh/id_rsa.pub) and copy paste the entire contents in the field marked with "key". Give it a name and save it. Next you'll be able to access the git repo's. Try to clone one of the projects (for the url look in gitlab). Eg;

```
git clone git@hostname:repo.git
```

If that works, you're set.

For your understanding, the clone url is simply a ssh url. Your connecting with ssh to "hostname" with user "git" and with path "repo.git". The username "git" is for everyone the same. This is a normal linux user on server "hostname" with its own home directory and such. What happens is that gitlab copied your public key you've entered to the .ssh folder of the git user on the server "hostname". This basically would allow you to ssh as the git user to that box. However, the git user has a "special shell" assigned so that it just accepts git commands (you won't be able to login and just run bash or

something). So everyone has access via ssh with the same user given that the public key is installed on the server (managed by GitLab). Besides the authentication/encryption that ssh gives us, gitlab adds extra more fine grained access control by coupling users to projects. Based on the public key you saved with your profile, GitLab can determine that it is user x who is actually logged in via the git user and not user y (although both have ssh access using the git user). Although we are all logged in via the same user, each user has its own unique key pair. The special shell that is executed upon login gets a different parameter for each public key granted access. The parameter is passed on to the shell identifying the actual user. This parameter is a gitlab id of the user within gitlab. So, although your git client maybe able to login using ssh, gitlab can still block a request if it sees that the actual user does not have the required access rights. (and these rights are given via the Gitlab webapp)

Eclipse#

In eclipse you use the Egit plugin. Normally you can do this using the eclipse "market place" and search for egit. If the installation gives an error, it might be that your eclispe is too old. You can use direct installation URL here: http://wiki.eclipse.org/EGit/FAQ#Where_can_I_find_older_releases_of_EGit.3F, check **"What versions of Eclipse does EGit target?"** You can then use the URL as an update URL in "Help>Install new software...".

By default the plugin adds a ">" for each resource that is dirty. This can be changed to that the resource is also prefixed with the asterisk in black (as with SVN). To do this, go to preferences->git and check the "dirty resources" by labels.

Sandbox#

Another thing we would like to address is how easy it is to perform local tests using git. Sooner or later you'll want to test something, verify that your understanding is correct or simply simulate something before actually doing it. testing/trying out is essential with every technology to get a feel on things or to get deeper understanding. You should be able to test something quickly without hassle or otherwise it becomes frustrating and it will be skipped eventually, leaving you with incomplete understanding and a lot of assumptions.

Fortunately, the beauty of a DVCS (Distributed VCS) is that you don't need a central server. If you want to test things, it can be easy as this:

```
koen@koen-HP-EliteBook-8570w:/tmp$ mkdir git-test
koen@koen-HP-EliteBook-8570w:/tmp$ cd git-test/
koen@koen-HP-EliteBook-8570w:/tmp/git-test$ git init
Initialized empty Git repository in /tmp/git-test/.git/
```

And you're done. You have a fully initialized GIT repo which has exactly the same functionality as our "real" development repo. In our real development environment we work with a central bare git repo. This repo serves as a master to which everyone pushes there changes. This doesn't mean that it's no longer distributed: everyone still has the complete repo locally and interaction with the central server is only required to share things with other developers. The central repo makes sharing easier. For example, another way of working would be that everyone synchronizes there repository with every other team member. This is of course much more of a hassle.

In the example above you only created a local repo, but you don't have a remote repo. While you don't need a remote repo as explained (just for sharing, but you don't want to share your tests) it can be sometimes useful to simulate the entire chain as it would work on the "real" development repository. So, to solve this you can easily create a second repo and consider it to be the central remote repository in your testing story. The remote repo can be another repo on your machine, the

fact that it's really remote (as on another machine reached via the network) is 100% transparent and will not influence your tests in any way.The only thing to watch out for is that the second repository you're going to create to serve as the remote repository is a bare" repository :

https://www.kernel.org/pub/software/scm/git/docs/gitglossary.html

A bare repository is normally an appropriately named directory with a .git suffix that does not have a locally checked-out copy of any of the files under revision control. That is, all of the git administrative and control files that would normally be present in the hidden .git sub-directory are directly present in the repository.git directory instead, and no other files are present and checked out. Usually publishers of public repositories make bare repositories available.

Why do I need a bare repository to simulate this? A non-bare repository has by default a branch checked out. You will not be able to safely push a branch that is the same as the checked out branch on the non-bare repo. You can push other branches without problems, but you don't want to make things harder as they are, so with a bare repo you are always safe and it matches with the "real" development scenario where the central repo is a bare git repo. The reason why this causes problems is that if git pushes changes to a non-bare repo which is currently checked out, it will update stuff behind the scenes that is not directly reflected in the checkout version. If the checked out version is not updated with these changes before there are new changes committed data might get lost.

So, if you want to do testing simulating a remote repo, do it like this:

```
koen@koen-HP-EliteBook-8570w:/tmp$ mkdir git-test-central
koen@koen-HP-EliteBook-8570w:/tmp$ cd git-test-central/
koen@koen-HP-EliteBook-8570w:/tmp/git-test-central$ git init --bare
Initialized empty Git repository in /tmp/git-test-central/
koen@koen-HP-EliteBook-8570w:/tmp/git-test-central$ cd ..
koen@koen-HP-EliteBook-8570w:/tmp$ git clone git-test-central/ git-test
Cloning into 'git-test'...
warning: You appear to have cloned an empty repository.
done.
koen@koen-HP-EliteBook-8570w:/tmp$
```

And you're done. Your checked out repository is now in /tmp/git-test and your central repo is now in /tmp/git-test-central. You can now add something in your local checked out repository and push it:

```
koen@koen-HP-EliteBook-8570w:/tmp/git-test$ echo "testing" > testing.txt
koen@koen-HP-EliteBook-8570w:/tmp/git-test$ git add testing.txt
koen@koen-HP-EliteBook-8570w:/tmp/git-test$ git commit -am "testing"
[master (root-commit) 080d289] testing
 1 file changed, 1 insertion(+)
 create mode 100644 testing.txt
koen@koen-HP-EliteBook-8570w:/tmp/git-test$ git push -u origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 221 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To /tmp/git-test-central/
 * [new branch]      master -> master
koen@koen-HP-EliteBook-8570w:/tmp/git-test$
```

You might notice two things. First, when creating the central repo in the previous part we got a warning; warning: You appear to have cloned an empty repository. This is no big deal, but as there is nothing in the repo, the clients that checkout the repo don't get any branch information. Directly after cloning the repo to git-test (before committing anything) you'll notice that there is no branch info whatsoever:

```
koen@koen-HP-EliteBook-8570w:/tmp/git-test$ git branch -a
koen@koen-HP-EliteBook-8570w:/tmp/git-test$
```

After committing your first change, git automatically created the "master" branch (if you'd run "git branch -a" again, you'll notice it prints "* master") Now, if you would do an ordinary push:

```
No refs in common and none specified; doing nothing.
Perhaps you should specify a branch such as 'master'.
fatal: The remote end hung up unexpectedly
error: failed to push some refs to '/tmp/git-test-central/'
koen@koen-HP-EliteBook-8570w:/tmp/git-test$
```

The master branch now only exists locally and not on the remote. You'll have to tell git that where it should track this branch on the remote repo. The best way to do this is specify it together with your first push. That's why we did "git push -u origin master". "Origin" denotes the remote we want to push to and "master" the branch we want to push. You can also give it another name on the remote by doing something like: "origin something else:master", which you normally don't want. Git takes the name of the local repo as the default for the remote repo.

The -u specifies that we want to set this remote tracking repo as the default upstream. This makes sure that you don't need to specify "origin master" the next time you pull or push.

```
koen@koen-HP-EliteBook-8570w:/tmp/git-test$ git branch -a
* master
  remotes/origin/master
```

Note that when you clone a repository containing branches, the remote tracking branches are created automatically. Let's say you've created a test branch:

```
koen@koen-HP-EliteBook-8570w:/tmp/git-test$ git branch
* master
```

We are currently on the only existing branch ?master?. We'll create a branch from our master branch at the current revision. Add a file to it, commit it, and then push the new branch to our remote specifying that we want to track it as a remote branch as well (using the -u parameter) for the current repository (as we'll illustrate next, this is not needed for other repositories that will clone our repository with the new branch afterwards):

```
koen@koen-HP-EliteBook-8570w:/tmp/git-test$ git branch test_branch
koen@koen-HP-EliteBook-8570w:/tmp/git-test$ git checkout test_branch
Switched to branch 'test_branch'
koen@koen-HP-EliteBook-8570w:/tmp/git-test$ echo "testing on test_branch" > testing_on
koen@koen-HP-EliteBook-8570w:/tmp/git-test$ git add testing_on_test_branch.txt
koen@koen-HP-EliteBook-8570w:/tmp/git-test$ git commit -am "testing on test_branch"
[test_branch a0d8572] testing on test_branch
 1 file changed, 1 insertion(+)
 create mode 100644 testing_on_test_branch.txt
koen@koen-HP-EliteBook-8570w:/tmp/git-test$ git push -u origin test_branch
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 313 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To /tmp/git-test-central/
 * [new branch]      test_branch -> test_branch
Branch test_branch set up to track remote branch test_branch from origin.
```

Next we clone git-test-central again in a second directory (simulating a new client cloning our repository containing the new branch) to illustrate the default remote tracking:

```
koen@koen-HP-EliteBook-8570w:/tmp/git-test$ cd ..
koen@koen-HP-EliteBook-8570w:/tmp$ git clone git-test-central/ git-test-2
Cloning into 'git-test-2'...
done.
koen@koen-HP-EliteBook-8570w:/tmp$ cd git-test-2
koen@koen-HP-EliteBook-8570w:/tmp/git-test-2$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/test_branch
```

The branch "test_branch" is there, but we need to check it out locally:

```
koen@koen-HP-EliteBook-8570w:/tmp/git-test-2$ git checkout test_branch
Branch test_branch set up to track remote branch test_branch from origin.
Switched to a new branch 'test_branch'
```

You can already see it reports that it will be tracking it automatically. If we now perform push or pull we won't get any errors:

```
koen@koen-HP-EliteBook-8570w:/tmp/git-test-2$ git push
Everything up-to-date
koen@koen-HP-EliteBook-8570w:/tmp/git-test-2$ git pull
Already up-to-date.
koen@koen-HP-EliteBook-8570w:/tmp/git-test-2$
```

Conventions & rules#

- Stable development branch: **master**
- Release branches: **release/xx**
- Feature branches: **features/JAVA-XYZ**
- tags: tags are recognized by git as a special type, so they don't require a prefix

A tag is created for every release. From the moment there is a hotfix required on a release, you can create a release branch from the tag of the release (basically you checkout the tag and then create a release branch). On the release branch you can then create additional tag for each point release.

Feature branching is the key for being able to work properly with git. As you will notice, creating branches and merging is painless using git and has no overhead what so ever. Rule of thumb: each change that isn't fixing a typo, code cleanup, code refactor or other small work is done on a feature branch. Even small features that you will finish in a couple of hours are to be done on a feature branch.

Conclusion: by default create a feature branch for every change you make. Unless it is a very small refactor such as a typo, style cleanup, removal of an obsolete file, ...

When creating a feature branch you are therefore not obliged to make that feature branch public. You can keep it locally and after merging back the feature in the master delete it. The rule of thumb is that if you finish work the same day and you're the only one working on it; keep it local. Does it take longer then one day, or do you need another developer to work together on the feature, push it to central.

A lot of work with git can be done locally. In contrast with subversion, most things you do are reflected directly on the central repo. With git, creating branches, tags, commits etc are always local

first. In case of branches you can even keep them local without ever sharing them via the central repo. While this a good thing in general, we follow a rule that work that you have done should be shared as fast as possible:

- Enables others to already follow what you are doing bit by bit
- While you have a local repository remember that if your laptop crashes your data will be lost if it's not available on the central repo
- If you are sick or on holiday others will not be able to continue your work
- If you are working together on a feature, the faster you share the faster conflicts can be solved (if any)

This rule falls together with the feature branch rule. By using a feature branch most of time you can safely commit without breaking the master development branch. You should make a habit to also push the feature branch to central from the moment your work exceeds one day. Remember; it is your responsibility to make sure that your changes are pushed to central before you go home!

Conclusion: **push** early and frequently.

commit messages; if you are working on a specific issue, you commit with "JAVA-XYZ". Not any personal derivative such as: "Javaxyz" or not "JaVa-XYZ". You capitalize "JAVA" followed by a "-" followed by the number. If you are working on a feature (within a feature branch) you can commit using a message that actually tells what you've done "fixing this" "adding that", ... It doesn't make sense to have 20 commits saying 'JAVA-XYZ'. From the moment you merge your work back in the master you use commit your work with a single commit "JAVA-XYZ". The small work directly done on master not related to a JAVA can also have a free format text. Like "Cleaning up javadoc", "Code review", "Fixing typo", ..

Conclusion: commit messages on the main development branch are in the form of JAVA-XYZ unless they are small refactorings not done on a feature branch and not related to a JAVA issue in Jira.

"Merge from master" commits: maybe you can re-read this after having gone through to the basic commands below. Using "git pull" can sometimes create cluttering merge commits. For example, you are going to work on master and do a "git pull" before starting your work. If the master has changes on central they will be pulled in. Since you don't have any changes on your local master, git is able to do a fast forward merge. This basically means it's not merging anything, but able to in-line changes and move the pointer after the last change retrieved from the remote. The result is comparable to an "svn up". You pulled in the changes and nothing else was done (no commit, no normal merge, ...). However, from the moment your local repository is "dirty" (you have committed changes) git is not able to do a fast forward. It will have to merge the changes from the remote into your local repo. Doing this git will generate a commit. In the history this will show up as:

Merge branch 'master' of git.host:some.repo

Suppose we didn't have the feature branch rule and Davy and Bram are working on master:

```
T1: Bram starts it work on master and does a "git pull" first. Git updates some files
<bram starts developing>
T2: Davy finishes his work and commits ("JAVA-001")
T3: Davy pushes to the central master
T4: Bram finishes his work, and commits to the local repo ("JAVA-002").
T5: Bram pushes to the central master
<error: git cannot push since the remote master has changed in the mean time>
T6: Bram pull's
<Git will now merge the changes into the local repo, it cannot fast forward since loca
T7: Bram pushes again and succeeds
```

The resulting history will look like this:

```
JAVA-001 (Author: Davy)
JAVA-002 (Author: Bram)
__Merge branch 'master' of git.host:some.repo (Author: Bram)__
```

The merge commit is the "problem". The changes recorded in the merge commit are the same as in the JAVA-001 commit from Davy. It is polluting the history with information that's already there. On a busy branch where a lot of changes are pushed and a lot of pulls happen, you can imagine that there could be dozens of these messages appearing per day.

Solution: feature branches. Some sites/individuals will advice to use "git rebase" but don't do that. The way to go is still using "git pull" and NOT to rebase. Merging is the natural way git works. When you work with a feature branch you will automatically circumvent this issue.

How? When you do your work on a feature branch you only need a (very) short period of time to reintegrate your branch into the master. Suppose you have a feature branch open for several days. After finishing you reintegrate it. Your master is still clean (since all work was done on the feature). Your way or working would be:

```
git checkout master
git pull
git checkout fb
git merge master
<commit mergeback, first fix conflicts if they appear>
git checkout master
git pull
git merge --no-commit --squash fb
git commit -am "JAVA-XYZ"
git push
```

All preparing work has been done on the feature. Once you're ready to push, there will only be some seconds between the 2nd "git checkout master" and the "git push". Chances that someone pushed in the mean time to the master are very low. So doing this you avoided any additional merge commits. Note that the commit "JAVA-XYZ" is also a merge commit, but this is of course the "real" merge commit containing information we really need.

On the other hand: if you have a public feature branch, updating it from the central repo can also create additional merge commits;

```
git checkout fb
git pull
<do some work>
git commit -am "Some work"
git push
<error: git cannot push since the remote has changed in the mean time>
git pull
-> pulls in some new changes committed by another developer working on fb, creating a
```

This merge commit will only exist on the feature branch. Once you've re-integrated the feature branch into the master your changes will be squeezed in one single commit (the merge commit). The master will only see a single commit containing all your changes. All individual commits remain on the feature branch, including the additional merge commits.

Binary files: avoid committing/pushing any binary files in git. For jar/war/zip artifacts we user our nexus. Even self maintained artifacts can be uploaded in nexus, there is no need in storing them in git. Documentation is best kept here on the wiki. Preferably in wiki format or otherwise as attached

documents. Small binaries that are required for the application itself are allowed when there is no other solution at hand. For example; keystores for tests or resources for web-projects (images, ...).

Git usages#

Checking out#

To check out, or in git terms "clone" the repository from the remote, you simply do:

```
git clone <repo>
```

You can find the repo URL using the GitLab web frontend. Once this is done you have the *entire* remote repository on your disk. No matter if you want to checkout branches, create branches, create tags, check history, revert a file, ... you don't need the remote anymore (you can literally disconnect your network). The only reason you'll need to contact the remote is to share your work or to fetch changes from others.

Committing#

This is more or less the same as with svn. Before you commit you'll have to add the directories or files to your local repo. By adding you'll place them in the staging zone. Committing makes the "permanent" in your local repo:

```
git add <directory or file>
git commit -am "commit message"
```

- -a denotes you want to commit all outstanding changes (all files/directories that have been added or changes)
- -m denotes the commit message (if not specified git will open your configured editor to enter the commit message)

Important: committing is, unlike to svn, only to your LOCAL repository. To share your work you'll have to synchronize your local commits with the remote repository (see pushing to remote). Again, work that is not pushed is NOT shared with others!

Reverting working copy#

Read: https://www.atlassian.com/git/tutorial/undoing-changes
Brief conclusion:

Revert **newly** added (but not yet committed) file(s):

```
git reset HEAD <file>
```

The file is then unstaged, but still appears. You can then remove it with "rm".
You modified existing files, which are not committed, with or without having staged them, and want to revert:

```
git checkout HEAD <file>
```

Clean **every** local change including working directory and staging:

```
git reset --hard HEAD
```

Reverting commits#

Read: https://www.atlassian.com/git/tutorial/undoing-changes
Brief conclusion:

Revert a commit that has not yet been pushed and of which you don't want any history traces of:

```
git reset --hard HEAD~x
or
git reset --hard <sha1rev>
```

Example, you've committed and directly realize that some extra files have slipped through that weren't supposed to be committed:

```
git reset --hard HEAD~1
```

This will reset to the previous commit. ~2 will reset to the second previous commit etc. If you need to reset multiple commits, you can find the revision sha1 of the commit you want to reset to using **"git reflog"**. then "git reset --hard <sha1_from_reflog>". Warning: remember that the revision you enter means: "remove all commits between your entered sha1 revision and HEAD". It is not only that revision that is removed.

If you want to remove a commit you've pushed, you use the revert command. This will undo the changes by creating an new commit rather then removing history. This will revert the last commit you've pushed:

```
git revert HEAD
```

You can also use the sha1rev instead of HEAD. Remember that there is a usage difference with reset; with reset you point to the PREVIOUS commit, since you want git to reset the HEAD pointer to there. With revert you want to undo the changes of a particular commit by pointing to the commit you want to revert. So if you want to undo a local commit using git reset, you would point to the previous revision (HEAD~1). If you want to revert (a pushed) commit, you would issue revert HEAD since you want to undo the changes of the current commit.

Updating from remote#

After you've cloned you probably want to synchronize your local repo with the central repo now and then. The easiest way to do this is to use "git pull". This is a shorthand command for "git fetch" and "git merge". It will first update your remote tracking branch with the latest changes from the remote using the fetch command.Then it will merge the changes from the remote tracking branch into the local branch (the branch you're actually working on).

If you don't changed anything in the branch you want to update, there is nothing special to be noticed. In this case the pull will work as an "svn up". It will simply bring in the changes using a fast forward merge and nothing special will happen.

Note that "git pull" will also pull in new branches, tags or other artifacts that are new on the central repo. However, it will NOT automatically merge branches with the new changes other then the branch you are on. Suppose you are on master, someone committed a change to feature branch "fb". Before you go home (assuming you cannot access the central repo from home) you do a "git pull" while on master. This will retrieve all changes from all known branches, but it will only merge from the remote tracking branch of master to your local master repo. Changes from feature fb are only staged on the remote tracking branch, but not yet merged on your local branch.

```
koen@koen-HP-EliteBook-8570w:/tmp/test$ git checkout master
Switched to branch 'master'
```

```
koen@koen-HP-EliteBook-8570w:/tmp/test$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 2 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (2/2), done.
From /tmp/central
   3232249..cda0e98  fb         -> origin/fb
Already up-to-date.
```

Our master branch was up-to-date, but there were changes on our feature branch fb. When we checkout our feature branch at home, we would see:

```
koen@koen-HP-EliteBook-8570w:/tmp/test$ git checkout fb
Switched to branch 'fb'
Your branch is behind 'origin/fb' by 1 commit, and can be fast-forwarded.
  (use "git pull" to update your local branch)
```

Using git pull won't work since it will do a fetch that tries to access the central repo. However, all changes are already in the remote, so we can skip the fetch part and just do the merge:

```
koen@koen-HP-EliteBook-8570w:/tmp/test$ git merge origin/fb
Updating 3232249..cda0e98
Fast-forward
 test2 | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 test2
```

Pushing to remote#

When you're ready to synchronize your local repo with the remote you can simply "push" the branch you want to synchronize. Before you push, you better always pull first to make sure you have the latest changes. If you don't, no big deal, but your push will fail telling you to pull first.

```
git checkout <the_branch>
git pull (make sure we're up to date before starting work)
<do some changes, "git commit" them>
git pull (make sure we're up to date before pushing, this is the point were you would
git push
```

Creating branches#

To create a new branch you can simply do:

```
git branch features/JAVA-XYZ
git checkout features/JAVA-XYZ
```

The branch is created from the branch you are currently on. Safer is to explicitly specify the branch you want to branch from.This way you can never accidentally branch from another branch (from which you did not intent to branch) without knowing it.

```
git branch features/JAVA-XYZ master
git checkout features/JAVA-XYZ
```

You can also checkout a branch and create it in a single command:

```
git checkout -b features/JAVA-XYZ
```

Again, better always specify from which branch to branch:

```
git checkout -b features/JAVA-XYZ master
```

Note: checkout -b will not work if the branch already existed, in that case you simply checkout without telling it to create the branch if it does not exist (drop the -b)

Use an existing (feature) branch#

You already cloned from the remote repo so all branches are already available locally. However, they are available in the git "backend" you'll have to explicitly "check out" a branch to be able to do something with it in your local repo:

```
git checkout <branch_name>
```

Since the branch already existed, git automatically tracks the local branch to the remote. This means that if you pull/push changes it will automatically work.

Checking available branches#

```
git branch -a
```

Shows all available branches including remote tracking branches. You can also use your browser and use gitlab to see which branches are there.

Merging#

To bring a feature branch "fb" up to date with its parent branch (eg. master), you simply do:

```
git checkout fb
(git pull: in case the feature branch is pushed to central, to make sure it's up to da
git merge (--no-commit) master  (--no-commit is not required, but it can be smart to u
git commit -am "Mergeback from master"
(git push: in case the feature branch is pushed to central, to make sure your merge is
```

Bringing back a feature branch back to master is the same thing:

```
git checkout master
git pull
git merge --no-commit --squash fb
git commit -am "JAVA-XYZ"
git push
```

Important: always use: **--no-commit --squash**. If you accidentally merged without these flags, but you did use --no-commit, no problem: just rollback with "git reset --hard HEAD". If you didn't use any of these flags and your merge was already committed, no problem, rollback the commit: "git reset --hard HEAD^1". Then simply re-do the merge this time using the correct flags.

The "--no-ff" stands for (no) Fast Forward merging. This is enabled by default but we don't want that:

http://git-scm.com/book/en/Git-Branching-Basic-Branching-and-Merging *when you try to merge one commit with a commit that can be reached by following the first commit?s history, Git simplifies things by moving the pointer forward because there is no divergent work to merge together ? this is called a "fast forward". Note: --no-ff cannot be combined with --squash, so if you got to make a choice between the two, use --squash.*

Basically when you reintegrate a feature, but there were no commits on the master since you've created the feature, git will automatically fast forward. This is normally a good thing, but you'll be missing a single clean merge commit from the feature you reintegrated. The history of the feature will be visible on the master branch (including all your intermediate commits) which will bloat its history.

The "--no-commit" is straight forward: git will not auto-commit if the merge succeeded without conflicts. Giving you a change to review the changes first
The "--squash" makes sure that the merge is committed using a single commit. For features this is generally what we want, as the history will otherwise be (by default at least) polluted wit all the "intermediate" commits on the feature. For example; if you started on feature X, changed some code, commit, and after a couple of days you decide to refactor. The code reviewer is only interested in the end result, the finalized feature going back to master. There could however be scenario's where the link to the history of the branch is desired, in these cases just drop --squash.

Note that there is technically no need to first merge the feature branch with the master before reintegrating the feature branch in the master.However, it is not a bad idea to do this because it will avoid possible merge commits if you have to deal with conflicts

Suppose you have a feature branch standing out for a couple of days and you never merged the feature branch with the master. Bringing it back into the master can possibly trigger conflicts with changes made by others in the mean time on the master. Once you've resolved the merge conflicts and finally commit everything the result will be the same as if you fixed the merge conflicts on the feature (by first merging back from the master). But depending on how long it took you to fix the merge conflicts, the master could have changed in the mean time. Since you have outstanding changes, pulling from the master will create a merge commit.

So the ideal scenario to follow is something like this:

```
git checkout master
git pull
git checkout fb
<git pull: if it was a remote available branch>
git merge master
<sort out merge conflicts>
git commit -am "Fixing merge conflicts"
<git push: if it was a remote available branch>
git checkout master
git pull
git merge --no-commit --squash fb
git commit -am "JAVA-XYZ"
```

Changes on the central master while you resolved conflicts on the feature don't cause a merge commit, since your master remained clean (you fixed the conflicts on the feature). It has to be said that nothing guarantees that the changes on the master branch won't cause NEW merge conflicts when you reintegrate your feature. The assumption is that fixing the conflicts on the feature branch is a job of minutes, maybe hours, but not days. Chances are high that someone committed to the master in the mean time but they will be low that these commits are causing new merge conflicts. If they do, you can still rollback everything on master and perform the operation again.

Undesired auto merge commit#

Auto merge commits on feature branches are not a big issue. They can occur if your branch is public and work with multiple developers together. Feature branches are short lived and when they are re-integrated you do this with a single merge commit coming from your manual merge. There will be no history of the auto merge commits on the master. Auto merge commits can also occur on master in two situations. They are undesired there and should be removed.

Scenario 1: you are working on master, pull first to be up-to-date, change something small and then commit/push. At this stage you get the message that your push failed because master has changed. You pull, but now you get an annoying merge commit. First, normally you wouldn't pull after getting this message in the first place, if you did you will have to reset the merge commit. If you didn't pull, you can skip the first steps and start directly (temporary) undoing your commits.

Reset the merge commit if needed:

```
git reset --hard HEAD^1
```

Ok, first verify that the merge commit is gone by checking the history in "gitg". Next, temporary undo your commits. Note that we are not using --hard here. Leaving out --hard will remove the commit but leave the changes in our current working copy:

```
git reflog  <- find your first commit and take the sha1 rev from the commit before tha
git reset <sha1>
```

Very important: you are throwing away commits (!) the assumption is that you apply this in a standard scenario, like the one given here. You pulled before starting your work, did several commits and then you couldn't push because the master has changed. You know want to push your changes avoiding a merge commit. If your scenario deviates from this be careful about what you are doing.

Your working copy is now dirty with the files you've changed/added before your commits, you can now stash them:

```
git stash
```

Your working copy will be clean now, verify with "git status", so you can pull (again) without introducing an auto merge commit:

```
git pull
```

Next apply the stash, check for conflicts, if none commit/push:

```
git stash apply
git commit -am "changes xyz"
git push
```

By resetting your commits and bringing them in your local working copy, your commit is now squeezed. While you may have had multiple commits you are now left with a single commit containing all your changes. This is not a big deal as changes applied on master are supposed to be small anyway. You can create a summary message describing your changes in a single commit message.

Scenario 2: you are working on a feature branch and want to re-integrate it in the master. You follow the merge procedure explained in the "merge" section. When you merged the feature onto your master, you want to push. However, in the mean time someone changed the remote so the push fails. At this stage you can simply undo your merge, pull to get the updates from remote and re-do the merge.

Reset your merge:

```
git reset --hard HEAD^1
```

If you did "git pull" after getting the message the push failed; a auto merge commit will have been created. In that case you need to rollback TWO commits. The latest commit to rollback is your auto merge commit, and the second commit is the actual merge you performed:

```
git reset --hard HEAD~2
```

Note that we are using "~" and not "^", resetting with ^ refers to the previous commit as allows resetting the specified number of commits .
Next our working copy should be clean. You can verify this with "git status". Finally, you can pull in the changes, re-do the commit and try to push again.

Note: when pulling in the changes and re-doing your merge, you could (very unlikely, but possible) have merge conflicts on your master. In the merge procedure we explained that you should first pull your master and then update the feature with your master. If merge conflicts occur they can be handled on the feature rather then on the master. If pushing your master fails after re-integrating your feature, you should theoretically re-do this by again merging your master in your feature etc. However, this is just a procedure to keep the history clean and not required by git. The assumption is that if you merged your master in your feature before re-integrating it and your push of the master fails, the update you pull in will most likely not create any conflicts (changes are very low). If they do you can simply fix them on the master branch and commit them as this is an exceptional case anyway.

Cherry picking#

Suppose you've created a feature branch from master some days ago, you've committed several changes. However, today you continue work, but you accidentally commit on master unknowingly. You cannot simply re-create the feature from master (including your commits) and remove the commits from master. As re-creating your feature would also throw away your commits from your previous session. A solution is to use cherry picking. Doing so you can select one or more commits to be re-applied on another branch:

```
git checkout <feature>
<gitg or git reflog to find the commit to be transfered>
git cherry-pick <sha1rev>
```

If you want to bring back multiple commits:

```
git checkout <feature>
<gitg or git reflog to find the first and latest commit of the range to be transfered>
git cherry-pick <OLDEST sha1rev> .. <NEWEST sha1rev>
```

Important: you need to put the **OLDEST** revision first and the newest last. If you don't do this cherry pick might fail as newer changes operate on files that have not been created or changed by older commits. Suppose you created a new file in commit x and changed the file in commit y. If you would first apply y it would fail as the file does not yet exist since commit y is not applied.

Delete a TAG#

If you have a tag named '12345' then you would just do this:

```
git tag -d <tagname>
git push origin :refs/tags/<tagname>
```