Encodo Systems AG – Garnmarkt 1 – 8400 Winterthur
Telephone +41 52 511 80 80 – www.encodo.com

Encodo GIT handbook

Introduction, Instructions and Conventions

Abstract

This document is an introduction to using GIT for source code version control at Encodo Systems AG.

Authors

Stephan Hauser
Marco Von Ballmoos

Version 3.0 – 23.07.2015

Table of Contents

1 Introduction 6
1.1 What is Git? 6
1.2 What is distributed version control? 6
1.3 Which tools does Encodo use? 6
2 Basic Concepts 7
2.1 Command-line Syntax 7
2.2 Command Context 7
2.3 Files vs. Changes 7
2.4 States 7
2.5 Branches 7
3 Best Practices 9
3.1 Focused Commits 9
3.2 Snapshots 9
3.3 Developing New Code 9
3.4 Merging vs. Rebasing 10
4 Development Process 11
4.1 Branch Types 11
4.2 Example 11
5 Repositories 14
5.1 Cloning 14
5.2 Tracking branches 14
5.3 Pulling 14
5.4 Pushing 15
6 Commands 16
6.1 Staging 16
6.1.1 Adding 16
6.1.2 Removing 16
6.1.3 Un-staging 16
6.1.4 Reverting 16
6.2 Committing 16
6.3 Stashing 17
6.4 Branching 17
6.4.1 Creating 17
6.4.2 Switching (checking out) 17
6.4.3 Deleting 17
6.5 Merging 18
6.6 Rebasing 18
6.7 Resolving merge conflicts 19
6.8 Tagging 20
6.9 Finalizing 20
6.9.1 Features 20
6.9.2 Releases 20
6.9.3 Hotfixes 20
6.10 The log 21
6.11 Advanced 21
6.11.1 Resetting 21
6.11.2 Reverting changes 21
6.11.3 Interactive rebasing 21
6.11.4 Submodules 22

Table of Figures

Figure 1 – Branching model example 13
Figure 2 – Using a central repository 14
Figure 3 – Two commits on the master branch [1] 18
Figure 4 – Commit on another branch [1] 18
Figure 5 – Fetched new commits on master branch [1] 19
Figure 6 – Merging back to the master branch [1] 19
Figure 7 – Rebasing on the master branch [1] 19

Version History

Version	Datum	Author	Comments
1.0	17.06.2011	sha	Initial Version
1.1	29.09.2011	mvb	Minor changes and enhancements; added "Best Practices" and "Enigma", "Table of Figures", "Terms and Abbreviations" and "Referenced Documents".
1.2	30.09.2011	sha	Minor changes for first public release; added "License"
2.0	25.07.2012	mvb	Complete rewrite of the " Development Process" chapter; removed most references to *Git Flow*; created illustration "Figure 1 – Branching model example"
3.0	23.07.2015	mvb	Remove references to Enigma, Encodo Git Shell and submodules; reorganized chapters

Referenced Documents

Nr./Ref.	Document	Version	Date
[1]	Pro Git <http://progit.org/book/>		
[2]	*SmartGit* <http://www.syntevo.com/smartgit/index.html>		
[3]	Linux Shell <http://linux.org.mt/article/terminal>		

Open Issues

Nr./Ref.	Document	Version	Date

Terms and Abbreviations

Term / Abbreviation	Definition / Explanation
VCS	A Version Control System manages file history for source code; also called *Source Code Control* (SCC). Classically implemented with a central repository/server.
DVCS	Distributed Version Control System: A VCS that works without a central repository.
Change	A modification made to a file comprising one or more changes lines
Commit	A collection of changes added to the repository in one atomic operation
Repository	A database containing all the commits of a project (which comprise the history of the project)
Branch	A named version in the repository
Working Tree	A local copy of a branch of the project, taken from the repository
Index	A staging area for changes to manage moving changes from the working tree to the repository
SHA1	An algorithm for generating a unique key from data

Term / Abbreviation	Definition / Explanation
head	The last commit made to a given branch

1 Introduction

This manual assumes that the reader is a developer and has at least some experience with other source control systems. This manual also refers to processes and tools used at Encodo Systems AG.

1.1 What is Git?

Git is a distributed version control system (DVCS) designed to handle anything from small to very large projects with speed and efficiency. Development was started in 2005 by Linus Torvalds for the Linux kernel development and it is available under an open-source license and is free to download and use commercially.

1.2 What is distributed version control?

A DVCS keeps track of software revisions and allows many developers to work on a given project without necessarily being connected to a common network.

DVCS takes a peer-to-peer approach, as opposed to the client-server approach of centralized systems. Rather than a single, central repository to which clients must synchronize, each peer's working copy also includes the entire repository. Peers synchronize their repositories by exchanging patches. This leads to a few very important differences from centralized systems:

- There is no canonical reference copy of the code base by default (there *can* be one, but it's only by convention and up to the project maintainer)
- Common operations like commit, history viewing and revert are executed against and local repository and are therefore very fast
- Synchronization with other team members may be more complicated, as merge conflicts will occur more often when pushing changes is not required for each and every commit, as with central repositories.

There are many more differences, but these are the main ones that come up when working with a DVCS.

1.3 Which tools does Encodo use?

Encodo uses SmartGit as its primary tool. For command line, Encodo uses PoshGit (an application that integrates Git into PowerShell). Please see the Git Setup in our Wiki for instructions on how to configure your user with these tools and Encodo's repository server.

2 Basic Concepts

2.1 Command-line Syntax

All git commands have the following syntax:

```
git command arguments
```

If you need help with the syntax of a command, you can always type `git command --help`.

2.2 Command Context

This chapter is very important, as using Git will be much easier to use if you understand the basic concepts behind it. You should try to forget any assumptions you may have from other source control systems to avoid confusion with subtle differences between more classic VCSs and Git.

Except for `push`, `pull`, `fetch` and a few others, every operation in Git is executed locally. This gives you very responsive commands and allows you to do almost everything offline.

Every object in Git is referenced and stored with an *SHA1* hash. This makes it possible to refer to any object in a Git repository and verify its integrity.

With the exception of a few very advanced commands and `gc` (garbage collection), Git only adds information to your repository. Whenever something is changed, a new copy is made and this copy is changed instead. This makes it very hard for the user to lose his work by accident. All commands which have the potential to destroy anything issue a warning.

2.3 Files vs. Changes

Though you will use files and directories to work with Git, the repository actually works with *changes* instead of files. A commit contains a list of changes made to certain files instead of versions of those files. The difference is important; it is what allows Git to have certain changes to a file in the index (staged) while also tracking other changes in the working tree.

2.4 States

States are the most important concept to understand, as you will use them all the time. Git recognizes three stages in which changes can be: *committed*, *modified* and *staged*.

- *committed* means that the change is safely stored in your local repository. When you push to the server, it is these changes that will be pushed.
- *staged* is a set of files that will be committed on your next commit. You can freely move your changes between *modified* and *staged* until you are satisfied with the results.
- *modified* means that you have local changes to a file, but they have not yet been staged or committed. These changes can be lost if make further changes or delete them locally.

Note that, since Git works with *changes*, it is possible to have some changes to a file staged in the index and other changes in the working tree.

2.5 Branches

Unlike other VCSs, Git has very lightweight branches, which can be created, changed and removed very quickly. Git branches are essentially nothing more than a pointer to a specific commit.

One important thing to remember is that commits do not know on which branch they were committed. This allows for removing branches but also makes it impossible to give a branch centered view on the repository. Though this makes visualization of branches more difficult, all in all, it allows for much greater flexibility in using branches.

The basic workflow goes something like this:

1. You modify a file in the local file system (the *working tree*)
2. You stage the file, adding snapshots of them to your staging area (the *index*)
3. You make a *commit*, which moves the snapshots from the index to your repository

3 Best Practices

3.1 Focused Commits

Focused commits are required; small commits are highly recommended. Keeping the number of changes per commit tightly focused on a single task helps in many cases.

- They are easier to resolve when merge conflicts occur
- They can be more easily merged/rebased by Git
- If a commit addresses only one issue, it is easier for a reviewer or reader to decide whether it should be examined.

For example, if you are working on a bug fix and discover that you need to refactor a file as well, or clean up the documentation or formatting, you should finish the bug fix first, commit it and then reformat, document or refactor in a separate commit.

Even if you have made a lot of changes all at once, you can still separate changes into multiple commits to keep those commits focused. Git even allows you to split changes from a single file over multiple commits (the *Git Gui* provides this functionality as does the index editor in *SmartGit*).

3.2 Snapshots

Use the staging area to make quick snapshots without committing changes but still being able to compare them against more recent changes.

For example, suppose you want to refactor the implementation of a class.

- Make some changes and run the tests; if everything's ok, stage those changes
- Make more changes; now you can diff these new changes not only against the version in the repository but also against the version in the index (that you staged).
- If the new version is broken, you can revert to the staged version or at least more easily figure out where you went wrong (because there are fewer changes to examine than if you had to diff against the original)
- If the new version is ok, you can stage it and continue working

3.3 Developing New Code

Where you develop new code depends entirely on the project release plan.

- Code for releases should be committed to the release branch (if there is one) or to the develop branch if there is no release branch for that release
- If the new code is a larger feature, then use a feature branch. If you are developing a feature in a *hotfix* or *release* branch, you can use the optional `base` parameter to base the feature on that branch instead of the *develop* branch, which is the default.

3.4 Merging vs. Rebasing

Follow these rules for which command to use to combine two branches:

- If both branches have already been pushed, then `merge`. There is no way around this, as you won't be able to push a non-merged result back to the origin.
- If you work with branches that are part of the standard branching model (e.g. release, feature, etc.), then `merge`.
- If both you and someone else made changes to the same branch (e.g. develop), then `rebase`. This will be the default behavior during development

4 Development Process

A *branching model* is required in order to successfully manage a non-trivial project.

Whereas a trivial project generally has a single branch and few or no tags, a non-trivial project has a stable release—with tags and possible hotfix branches—as well as a development branch—with possible feature branches.

A common branching model in the Git world is called *Git Flow*. Previous versions of this manual included more specific instructions for using the *Git Flow*-plugin for Git but experience has shown that a less complex branching model is sufficient and that using standard Git commands is more transparent.

However, since *Git Flow* is a very widely used branching model, retaining the naming conventions helps new developers more easily understand how a repository is organized.

4.1 Branch Types

The following list shows the branch types as well as the naming convention for each type:

- **master** is the main development branch. All other branches should be merged back to this branch (unless the work is to be discarded). Developers may apply commits and create tags directly on this branch.
- **feature/name** is a feature branch. Feature branches are for changes that require multiple commits or coordination between multiple developers. When the feature is completed and stable, it is merged to the *master* branch after which it should be removed. Multiple simultaneous feature branches are allowed.
- **release/vX.X.X** is a release branch. Although a project can be released (and tagged) directly from the *master* branch, some projects require a longer stabilization and testing phase before a release is ready. Using a release branch allows development on the develop branch to continue normally without affecting the release candidate. Multiple simultaneous release branches are strongly discouraged.
- **hotfix/vX.X.X** is a hotfix branch. Hotfix branches are always created from the release tag for the version in which the hotfix is required. These branches are generally very short-lived. If a hotfix is needed in a feature or release branch, it can be merged there as well (see the optional arrow in the following diagram).

The main difference from the Git Flow branching model is that there is no explicit stable branch. Instead, the last version tag serves the purpose just as well and is less work to maintain. For more information on where to develop code, see "3.3 – Developing New Code".

4.2 Example

To get a better picture of how these branches are created and merged, the following diagram depicts many of the situations outlined above.

The diagram tells the following story:

- Development began on the master branch
- v1.0 was released directly from the master branch
- Development on feature "B" began
- A bug was discovered in v1.0 and the v1.0.1 hotfix branch was created to address it

- Development on feature “A” began
- The bug was fixed, v1.0.1 was released and the fix was merged back to the master branch
- Development continued on master as well as features “A” and “B”
- Changes from master were merged to feature “A” (optional merge)
- Release branch v1.1 was created
- Development on feature “A” completed and was merged to the master branch
- v1.1 was released (without feature “A”), tagged and merged back to the master branch
- Changes from master were merged to feature “B” (optional merge)
- Development continued on both the master branch and feature “B”
- v1.2 was released (with feature “A”) directly from the master branch

Legend:

- Circles depict *commits*
- Blue balloons are the first commit in a *branch*
- Grey balloons are a *tag*
- Solid arrows are a *required* merge
- Dashed arrows are an *optional* merge

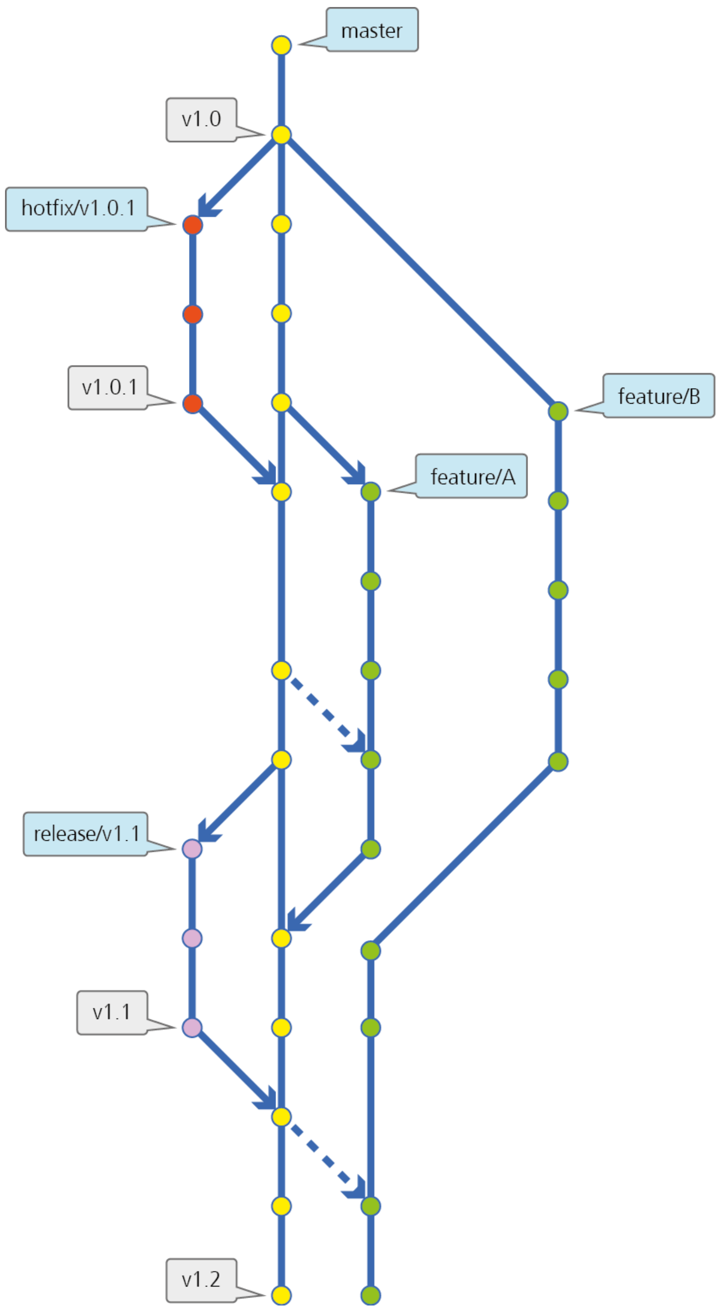

Figure 1 – Branching model example

5 Repositories

The nature of DVCS is very different from normal VCS when working with other developers. Work is synchronized between clients by pushing & pulling.

All pulled remote branches can be displayed using `git branch -a`. Remote branches are always marked with an `origin/` prefix.[1]

5.1 Cloning

When starting work on a project, you'll want to download an existing repository most of the time. To do this, you must use `git clone url`. This will download the whole history from the remote server / client and allow you to work locally on the repository.

Because every developer follows this workflow, someone will probably make changes they want to share with everyone. To do that, he pushes his changes to the central repository and every other developer can get those changes into their local repository when they `pull` or `fetch` from that repository.

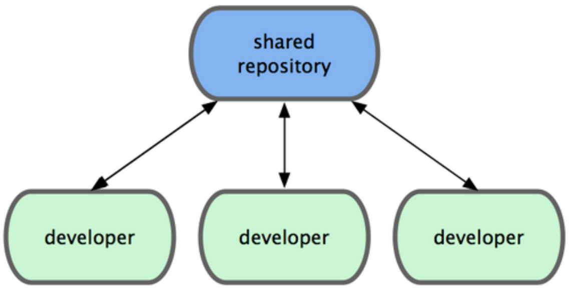

Figure 2 – Using a central repository

5.2 Tracking branches

A tracking branch is the default branch to pull from / push to. It can be configured in the `.git/config` file. When you create a branch you can make set a tracking branch using the `git branch --track name` where `origin/name` is an existing remote branch.

5.3 Pulling

To fetch all the changes from a remote repository into your repository, you can use `git fetch [url [branch]]`. If the repository was created using `git clone`, the URL can be omitted. This will download all new commits and branches from the remote repository and include them into your repository. It will not do anything with your local commits.

After downloading all the data from the remote repository, the new commits will diverge from your new commits. We already saw in "6.6 – Rebasing" how rebase works and this is exactly what we need to do to move our work to be on top of the new commits. See "3 – Best Practices" for the basic workflows to use.

There is another command to get the latest changes from a remote repository: `git pull`. The difference to `git fetch` is that `git pull` will also merge the latest changes into your current branch. This will create a merge commit if both you and someone other made changes, which is

[1] The default name of the first remote repository is `origin`; this name can be changed. Also, if there are multiple remote repositories, the other repositories will have a different prefix.

not desired. It is therefore recommended to only use `git pull` if you don't have any local commits.

5.4 Pushing

To push my own changes to the remote repository, use `git push [url [branch[:remotebranch]]]`. If the repository was created using `git clone`, the URL may be omitted or replaced by origin instead. If the current branch has a tracking branch, the `push` command will use that as the target branch by default.

6 Commands

6.1 Staging

The staging area is internally referred to as the "index". You may find this name in the output of some commands.

If you want to see the current state of your staging area and any local modifications, use `git status`. Please note that the status output will also list the commands which are useful for moving files between the different states.

6.1.1 Adding

Changes are added to the staging area using `git add filename`. This will add the given file / change to the staging area. The command actually means to "add the changes" for the given file. If this is a new file, then the entire content of the file is added; otherwise the changes versus the index are added to the index. This command can be executed at any time to make a snapshot without committing the changes. See "3.2 – Snapshots" for more information.

If you have renamed a file, Git will automatically recognize that fact as long as the content has not changed too much at the same time. For this recognition to work, both the removal of the old file and the inclusion of the new file need to be added to the staging area. You can verify this using `git status`.

6.1.2 Removing

If you removed a file from the local file system and want that change to be committed, you can add this change to the staging area using `git rm filename`.

6.1.3 Un-staging

If you added a file with `git add` or `git rm` and want to move this change back to your local modifications, use `git reset HEAD filename`.

6.1.4 Reverting

If you have local modifications you don't want to keep, you can use `git checkout -- file` to revert them to the state in the staging area.

If you want to undo all local modifications, see "6.11.1 – Resetting".

6.2 Committing

The most important step of working with the staging area is the actual commit to the local database. This can be done using a simple `git commit`. The editor will come up to let you edit the commit message. If you don't specify any message, the commit will be aborted.

If you want to make changes to the last commit, you can use `git commit --amend`. This will add all changes from the staging area to your last commit and allow you to edit the commit message.

6.3 Stashing

The stash is a useful utility to temporarily put changes away without committing them to the local repository. It can be very useful if you want to do operations which need a clean index, like merging or rebasing.

- To move all current changes to a new stash frame use `git stash`
- To list all available stashes use `git stash list`
- To apply a previously created stash use `git stash apply stash@{n}`. You can also apply the last created stash by omitting the `stash@{n}` argument.

Most of the time you don't need a stash any more after you applied it. You can use `git stash pop` instead of `apply` to remove it right after successfully applying it.

6.4 Branching

6.4.1 Creating

To start a new branch—feature "A", for example—switch to the desired source branch (usually *master*) and then execute the following command:

```
git checkout –b feature/A
```

Instead of just checking out an existing branch named "feature/A", the –b flag tells Git to create the branch and switch to it. That's all there is to it.

6.4.2 Switching (checking out)

To switch the currently selected branch, you can use `git checkout <branch-name>`. This will change all working files to the state in the requested branch or commit.

NOTE: Switching the branch will only work if you have no changes to any of the files that differ between the current and requested branch. If you have local changes, use the stash (see 6.1) to put your changes away while switching branches. This is in order to protect you from destroying your work in a bad merge.

6.4.3 Deleting

Once a branch is no longer needed, it can be deleted locally and must also be deleted from the remote repository.

6.4.3.1 Deleting local

To delete a local branch called "feature/A", use:

```
git branch –D feature/A
```

To be a bit more careful, substitute –d for –D to prevent Git from deleting branches that haven't been synced with the origin yet. A standard workflow, however, is to make a final commit to branch *A*, merge to the *master* branch and then delete branch *A*. In this case, the –d option will insist that you push branch *A* to the server first, which makes little sense here.

6.4.3.2 Deleting remote

To delete the corresponding branch on the server, use:

```
git push origin :feature/A
```

The command to delete a remote branch uses an extended syntax for the *push* command, which indicates the local and remote references to use, separated by a colon. Since the local reference is empty, the remote reference is removed instead of updated.

6.5 Merging

To merge changes from another branch into the current branch use `git merge <branch-name>`.

NOTE: Please double-check the direction in which the merge is executed, because Git doesn't know which direction is the "right" one. You'll want to have checked out the target branch and then merge the branch you want to replay on the current branch. See "3.4 – Merging vs. Rebasing" for information about the preferred direction for merge-operations.

If you have a merge conflict, please follow the instructions in "6.7 – Resolving merge conflicts" and commit the merge.

6.6 Rebasing

Rebasing is an operation that is a bit more advanced than merging, but allows you to achieve pretty much the same result. Make sure you understand the differences between rebase and merge before using rebase (see "3.4 – Merging vs. Rebasing" for more information).

WARNING: Never ever use rebase on a commit that has already been pushed. Rebasing will change your commit IDs and thus make other people's work based on yours invalid. Nothing will be irrevocably broken, but other users will not be able to push until they rebuild their local repositories to match yours.

Rebasing can be explained pretty easily using an example:

Note: the graphics for this sample are from the ProGit [1] book and available used under the Creative Commons License.

1. We're currently working on master and doing our first two commits

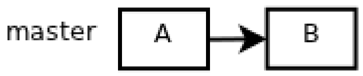

Figure 3 – Two commits on the master branch [1]

2. Now we switch to another branch and make a commit based on B

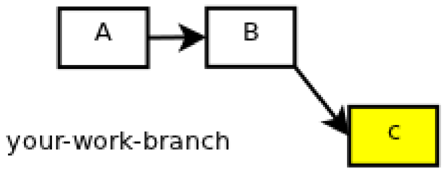

Figure 4 – Commit on another branch [1]

3. Someone else makes two commits in the meantime based on B

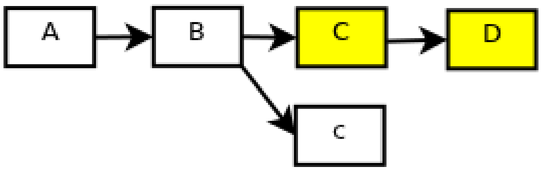

Figure 5 – Fetched new commits on master branch [1]

4. We now want to push our changes upstream but need to include all changes done in the meantime, too. We have two options for this:

   a. Merge the changes C & D into our c

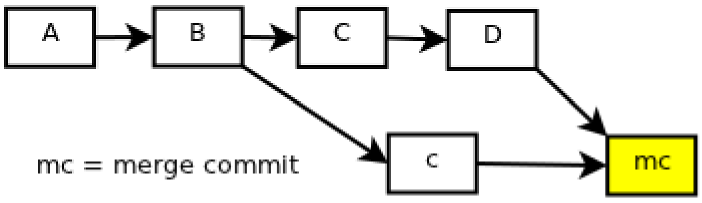

Figure 6 – Merging back to the master branch [1]

   b. Rebase our change c onto D

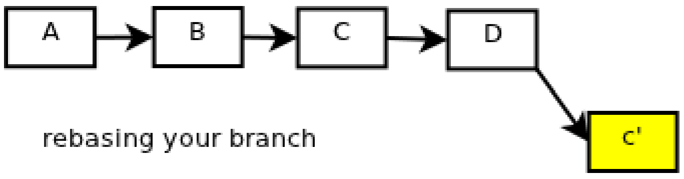

Figure 7 – Rebasing on the master branch [1]

What rebase basically does is to replay all your changes on top of another change you specified. It makes a copy of c on top of D and moves your branch pointer to the new commit.

The main difference is that a merge will be visible with those "merge bubbles" later, while a rebase will not be visible because the commits were reapplied as if they were made on the target commit in the first place.

To rebase the current branch onto a commit use `git rebase branch`.

If there are merge conflicts, proceed as described in "6.7 – Resolving merge conflicts". After resolving the conflicts do not commit, but use `git rebase --continue` instead. If all the changes were applied in both branches, all files will disappear after merging. In this case you can use `git rebase --skip`.

If you want to start over, you can use `git rebase --abort` and redo the rebase or, if the rebase merge proved too difficult to resolve, try merging instead.

6.7 Resolving merge conflicts

A merge conflict can occur as a result of operations like `merge`, `rebase`, `pull`, or `stash`. Whenever a merge conflict occurs, the command will tell you that you need to resolve it and how to continue afterwards.

Whenever a merge conflict occurs, the files are split into two groups:

- The ones that were merged successfully are staged
- The ones that could not be merged automatically will stay modified and have merge markers included

For resolving merge conflicts you can use any visual merging tool with which you feel the most comfortable: *SmartGit* has a merging tool but can also integrate others, like *BeyondCompare* (highly recommended). The merge format is text-based, so you can resolve conflicts in a standard text editor, though it's not very convenient.

After resolving the conflict, stage the file using `git add filename`.

6.8 Tagging

A tag is nothing other than a read only branch. Tags can be used to mark releases or any commit that is important to reference at a different time.

A tag can be created with the following command:

```
git tag –a tagname
```

Tags are not pushed automatically; use the following command to update the remote repository:

```
git push --tags
```

6.9 Finalizing

The following sub-sections indicate the necessary steps for finalizing the different types of branches documented in "4 – Development Process".

6.9.1 Features

- `git checkout master`: Switch to the *master* branch
- `git merge feature/A`: Merge in the feature branch
- `git push`: Push the merge commit to the server
- Delete the feature branch (see above)
- *Optional*: merge master to any open feature branches

6.9.2 Releases

- Create and push a tag from *release/vX.X.X* (see above)
- `git checkout master`: Switch to the *master* branch
- `git merge release/vX.X.X`: Merge in the release branch
- `git push`: Push the merge commit to the server
- Delete the release branch (see above)
- *Optional*: merge master to any open feature branches

6.9.3 Hotfixes

- Create and push a tag from *hotfix/vX.X.X* (see above)
- `git checkout master`: Switch to the *master* branch
- `git merge hotfix/vX.X.X`: Merge in the hotfix branch
- `git push`: Push the merge commit to the server
- Delete the hotfix branch (see above)

- *Optional*: merge hotfix to any open feature branches

6.10 The log

You can have a look at the changes done in the past using `git log`. If you want to have a more compact output, you can use `git log --oneline`.

To view all changes done in a single commit, use `git show ID`. Note that you don't need to type the full ID. As long as the first n characters are unique, you can just use them. The one-line log only shows the first 8 characters, for example.

6.11 Advanced

6.11.1 Resetting

WARNING: Resetting will affect your current working tree without further warnings. Always check if you have local changes before you reset your branch.

The reset command will reset the current branch to a different state. This can be useful if you made changes (e.g. a commit or a failed merge) and want to undo everything, reverting back to another state. A soft reset, for example, is useful for moving commits on a temporary branch to a main branch (i.e. one that is remotely tracked).

There are two different operation modes available:

- A *Regular reset* will move all committed changes past the given commit to the index. The command fails if there is a conflict with any changes made in the working tree. A regular reset is executed with the command `git reset <commit-ID>`.*Soft reset* will change the branch pointer to a different commit but keep your current index. This operation will move all your local changes and all committed changes that differ from the target commit to your working tree, from where you can work with them. A soft reset is executed with the command `git reset --soft <commit-ID>`.
- A *Hard reset* will reset everything to a specific commit. All local changes will be lost on the way. A hard reset is executed with the command `git reset --hard <commit-ID>`.

6.11.2 Reverting changes

If you want to revert a change that is already committed (and possibly pushed), you can always undo it using `git revert ID`.

It is possible that this operation yields merge conflicts which you'll have to resolve prior to committing the change. See "6.5 – Merging" for details on how to do this.

After all changes have been verified, you'll need to commit these changes using `git commit`.

6.11.3 Interactive rebasing

Interactive rebasing introduces some very interesting and advanced operations. As interactive rebasing is a very advanced and broad topic which could fill another 5 pages, only some of the possibilities are covered here.

Just for a few of the more common examples, interactive rebasing can be used to accomplish the following tasks:

- Squash multiple commits into a single one.
- Edit the commit message of a commit (that is not the most recently applied commit).
- Amend files to a commit (that is not the most recently applied commit).
- Remove a commit altogether.
- Split a commit into multiple atomic commits.

6.11.4 Submodules

Submodules are used to keep the contents of a part of the repository in another repository. This allows a project to declare a dependency on one or more libraries (e.g. third-party code or binaries), but not manage those changes directly. The changes to the library are applied once in its own repository and pulled in to any repository using that library as a submodule. Submodules are configured in the `.gitmodule` file in the root directory of a Git repository.

NOTE: Encodo no longer actively uses submodules, instead using packaging mechanisms like NuGet to resolve external dependencies. Please see a prior version of this manual for more documentation on using Git 1.7 to manage submodules.