SmartGit Quickstart Guide

syntevo GmbH, www.syntevo.com

2009

Contents

Chapter 1

Introduction

SmartGit is a graphical Git client which runs on all major platforms. Git is a distributed version control system (DVCS). SmartGit's target audience are users who need to manage a number of related files in a directory structure, to co-ordinate access to them in a multi-user environment and to track changes to these files. Typical areas of application are software projects, documentation projects or website projects.

Acknowledgments

We want to thank all users, who have given feed-back to the early-access-builds of SmartGit and in this way helped to improve it by reporting bugs and making feature suggestions.

Chapter 2

Basic Concepts

First, we need to define some Git-specific names which might differ in their meaning with, for example, those from Subversion.

In contrast with classical version control systems like CVS or Subversion (SVN), in Git there is not only a single *repository*, but each user has his own repository (for simplicity we will further refer to the user as a 'he'). Of course, you can set up Git also with one *central repository* (similar to the one from CVS or SVN).

Usually, every repository has a *working tree* attached. The working tree is a simple file system tree and contains a set of files which you can edit. At the same time the working tree has an assigned *commit* (what is the equivalent to an SVN revision). Hence for every file in a working tree a state can be defined in relation to the repository. This is the foundation for all *local* Git commands: adding, committing, reverting changes, etc.

Example
Let's assume you have all your project related files in a directory `D:\my-project`. Then this directory represents the working tree containing all files to edit. The attached repository (or more precise: the repository's meta data) is located in the `D:\my-project\.git` directory.

On the *central repository*, there usually doesn't exist a working tree, because files would not be edited on the server directly.

2.1 Typical Project Life-Cycle

As with all version control systems, you first have to *init*ialize a repository and store your project files in this repository. Then (other) users *clone* their repository from the *origin*-repository. Now they can make changes to the files in the working tree of their cloned repositories and *commit* the changes. The changes will be stored in their repository only, they don't even need to have access to the origin-repository when committing. At some later time, after a user has committed a couple of changes, he can push (see 3.2.1) them back to the origin-repository. Other users which also have their own clone of the origin-repository, fetch (see 3.2.2) (resp. *pull*) changes from the origin-repository.

2.2 Branches

Git distinguishes between two kinds of branches, *local branches* and *remote branches*. Remote branches refer to local branches of the origin-repository. In other words: if you clone from a repository, the clone will contain the local branches of the origin-repository as remote branches.

You can't commit to remote branches in your repository directly, but instead have to commit to the corresponding local branch. From the perspective of the local branch, the corresponding remote branch is called the *tracking* branch. After you have pushed changes from your local branch to the origin-repository, the remote branch will get your changes. If you fetch changes from the origin-repository, these commits will be stored in the remote branch in your repository. To get the remote branch changes into your local branch, the remote changes have to be *merged from the tracking branch*. This can be done directly when invoking the *Fetch* command in SmartGit or later by explicitly invoking the *Merge* command.

The default local branch which Git creates is named `master`. When cloning a remote repository, the `master` tracks the remote branch `origin/master`.

2.3 Excursion to The Basics

Note	At least for the authors keeping in mind the fundamental concepts of this section was the mental break-through in understanding branches, merging and rebasing and all other great features which Git offers.

2.3.1 It's All About Commits

Commits are what really matters in Git, so we will take a short trip to theory, namely *commit graphs*.

Every repository starts with the *initial commit*. Every subsequent commit is directly based on one or more *parent* commits. In this way a repository is a *commit graph* (or more technically speaking: a directed, acyclic graph of commit-nodes) with every commit being a descendant from the initial commit. This is the reason why a commit is not just a set of changes, but due to its fixed location in the commit graph, also specifies a unique repository state.

Each commit can be identified by its unique *SHA*-ID. Git allows to *check out* every commit using its SHA (SmartGit does not require you to enter such hard to remember SHAs, but instead lets you select commits). Checking out a commit will set the working tree to the corresponding repository state. Then you may alter the working tree and commit your changes which will add a new commit to the repository which will have the previously checked out commit as its parent. This way you are extending the commit graph.

Newly created commits are called *heads*, because there are no other commits descending from them. For every repository there is one distinct *HEAD* pointing to the "current" commit, the commit to which the working tree belongs.

2.3.2 Branches are Just Pointers

Every branch is simply a named *pointer* to a commit. The *HEAD* may point to a local branch instead of a commit. In this case committing changes will not only create a new commit, but also move the branch pointer (and hence *HEAD*) to the new commit.

2.3.3 How Things Play Together

The following example shows, how commits, branches, pushing, fetching and (basic) merging play together.

Example
Let's assume we have commits `000`, `001` and `002`. `master` and `origin/master` both point to `002`. `HEAD` points to `master`.
Now, let's commit a set of changes which results in commit `003`. Commit `003` is a child of `002`. `master` will now point to `003`, hence it is one commit *ahead of the tracking branch* (`origin/master`).
When performing *Push*, Git uses this information and sends `003` to the origin-repository, moving its `master` to `003`, too. Because a remote branch always refers to the branch in the remote repository, `origin/master` of our repository will also be set commit `003`.
Now, let's assume someone else has further modified the remote repository and committed `004` as a child of `003`, i.e. the `master` in the origin-repository points now to `004`. When fetching from the origin-repository, we will receive commit `004` and our repository's `origin/master` will be moved to `004`.
Finally, we will now *Merge* our local `master` from its tracking branch (`origin/master`). This will simply move `master` to commit `004`, too.
This was the completely Push-Fetch-Merge cycle when working with remote repositories.

2.4 The Index

The *Index* is an intermediate "storage" for preparing a commit. Depending on your personal preferences, SmartGit allows you to make heavy use of the Index or to ignore its presence at all.

With the *Stage* command you can save a file *content* from your working tree in the Index. If you stage a previously version-controlled, but in the working tree missing file, it will be marked for removal. You can do that explicitly using the *Remove* command, just as you are accustomed from CVS or SVN. Right-clicking the project root in SmartGit and selecting *Commit* will give you the option to commit all staged changes.

If you have staged some file changes and later modified the working tree file again, you can use the *Revert* command to either revert the file content to the staged changes stored in the Index or to the file content stored in the repository (HEAD). The **Changes** preview of the SmartGit project window can show the changes between the HEAD and Index, the HEAD and working tree and the Index and the working tree state of the selected file.

When *unstaging* previously staged changes before committing them, the staged changes will be "moved" back to the working tree, if the working tree is not modified. The Index will get the HEAD file content.

2.5 Merging

A normal commit has just one parent commit (or none in case of the initial commit). A merge commit has two (or more) parent commits.

Git uses different kind of merges, the most important one is the *fast-forward* merge. If, for example, you don't have done any local commits in your repository and you pull remote changes, they have to be merged into your local branch. But because a branch in Git is just a pointer to a commit, Git does not need to create a merge commit (with the parent commit from the local and remote branch). Simply moving the branch-pointer forward is sufficient and the most effective way, because no separate file content has to be stored in the repository - like it would be necessary, for example, when merging with CVS.

Note	Merging is a fundamental concept in Git and SmartGit performs merges automatically in situations where you might not expect it. For example, if you are working in the `master` branch and want to switch to the `release-1` branch, SmartGit merges changes from the tracking branch `origin/release-1`. So be aware that a plain switch to a different branch can result in a conflict.

2.6 Working Tree States

Usually, you can commit individual file changes. But there are some situations where this is not possible, e.g., if a merge has failed with a conflict. In this case you either have to finish the merge by solving the conflict, staging the file changes and performing the commit on the working tree root or by reverting the whole working tree.

Chapter 3

Important Commands

3.1 Project-Related

A SmartGit project is a named entity which usually has one assigned working tree and makes working with it easier by remembering a couple of, especially GUI related options. Depending on the selected directory, when cloning or opening a working tree, SmartGit allows to create a new project, open an existing one for the directory or to add the working tree to the currently open project.

To group the projects, use **Project**|**Project Manager**. To remove a working tree from a SmartGit project, use **Project**|**Remove Working Tree**. If you have moved a working tree on your hard disk to a new location, SmartGit will let you know when opening the project that it could not find the working tree. In this case, select the missing working tree and use **Project**|**Edit** to tell SmartGit the new location.

3.1.1 Open Working Tree

Use this command to either open an existing local working tree (e.g. initialized or cloned with the Git command line client) or initialize a new (personal) working tree.

You need to specify the local directory which you want to open. If the specified directory is no Git working tree, you have the option to initialize it.

3.1.2 Cloning a Repository

Use this command to clone a repository.

Specify the repository to clone either as a remote URL (e.g. ssh://user@server:port/path) or, if the repository is locally available in your file system, the file path. In the next step you have to provide a local path where the clone should be located.

3.2 Synchronizing with a Remote Repository

These commands can be found in the **Remote** menu:

3.2.1 Push

Use this command to store local commits to a remote repository.

In case multiple repositories are assigned to your local repository, select the target repository where you want to store the commits to. Select the local branch(es) for which you want to push commits. If you try to push commits from a new local branch, you will be asked whether to create the necessary tracking branch. In most cases it's recommended to create the tracking branch, so you will also be able to receive changes from the remote repository and have Git's branch synchronization mechanism working here (see Section 2.2).

3.2.2 Fetch

Use this command to fetch commits from a remote repository.

After successful fetching the commits of the remote repository are stored in remote branches of the local repository. They have to be merged into the corresponding local branches either automatically or manually. If the option **Merge fetched remote changes** is selected, the merge will happen immediately after fetching. If the merge worked without conflict and the option **Commit merged remote changes** is selected, a merge commit is created automatically, otherwise the working tree remains in merging state.

Alternatively, you can use the Merge (see 3.4.3) command to merge the remote changes from the tracking (remote) branch to the local branch.

Note	When you fetch *submodules* (nested repositories) the first time, you need to invoke `git submodule init` manually for your working tree. Currently, SmartGit can only fetch submodules after they have been initialized (`git submodule update`).

3.3 Local Operations on the Working Tree

These commands can be found in the **Local** menu.

3.3.1 Stage

Use this command to prepare a commit by saving the current file content state in the Index (see 2.4), by scheduling an untracked file for adding or a missing file for removing from the repository.

To commit staged changes, invoke the commit (see 3.3.4) command on the working tree root.

3.3.2 Unstage

Use this command to undo a previous stage (see 3.3.1).

If the file content in the Index is the same as in the working copy, the indexed content will be restored to the working tree, otherwise the indexed content will be lost.

3.3.3 Ignore

Use this command to mark untracked files as to be ignored. This is very useful for files which should not be stored in the repository and ignoring them helps to not forget to add files which should be stored in the repository. If the menu option **View|Ignored Files** is selected, selected files will be shown.

Ignoring a file will write an entry to the `.gitignore` file in the same directory. Git supports various options to ignore files, e.g. patterns that apply to files in subdirectories, too. Using the SmartGit Ignore command only ignores the files in the same directory. To use the more advanced Git ignore options, you may edit the `.gitignore` file(s) yourself.

3.3.4 Commit

Use this command to save local changes in the repository, to create a commit.

If the working tree is in merging state (see Section 2.5), you only can commit the whole working tree. Otherwise, you can select the files to commit (previously tracked, now missing files will be removed from the repository, untracked new files will be added). If you have staged (see 3.3.1) changes in the Index, you can commit only these Index changes by selecting the working tree root before invoking the commit command.

Note	If you commit one or more individual files which have both staged and unstaged changes, all changes will be committed.

While entering the commit message, you can use <*Ctrl*>+<*Space*>-keystroke to complete file names or file paths. Use **Select from Log** to pick a previous commit message from the log.

If **Amend foregoring commit instead of creating a new one** is selected, you can update the commit message and files of the previous commit, e.g. to fix a typo or add a forgotten file.

3.3.5 Undo Last Commit

Use this command to undo the last commit. The committed file contents will be stored in the Index (see 2.4).

Warning! Don't undo a commit which has already been pushed!

3.3.6 Revert

Use this command to revert the file content either back to their Index (see 2.4) or repository state (HEAD). If the working copy is in merging state, use this command on the root of the working copy to get out of the merging state.

3.3.7 Remove

Use this command to remove files from the repository and optionally delete them in the working tree.

If the local file in the working tree is already missing, staging (see 3.3.1) will have the same effect, but the Remove command also allows to remove files from the repository and still keeping them locally.

3.3.8 Delete

Use this command to delete local files (or directories) from the working tree.

> **Warning!** Note, that the files will *not* be deleted into the system's trash and hence restoring the content might not be possible!

3.4 Branch Handling

Branch-handling commands are located in the **Branch** menu.

3.4.1 Switch

Use this command to switch your working tree to a different branch.

If you select a remote branch, you can optionally create a new local branch. Not creating the local branch will not allow to commit changes afterwards.

Switching to a local branch which has a tracking (remote) branch, will try to merge changes from the tracking branch after the switch if the option **Merge changes from tracking branch** is selected. If this option is not selected, you can later use the merge (see 3.4.3) command to merge changes from the tracking branch.

3.4.2 Checkout

Use this command to switch the working tree to a certain commit.

First select the **Root** which contains the desired commit, then select the commit.

3.4.3 Merge

Use this command to merge changes from another branch to the current branch.

If the current branch has a tracking (remote) branch, you simply can select **Tracking branch** to merge those changes. To merge from any other branch, select **Other branch or commit** and pick the commit or branch.

With **Fast-forward** merge, SmartGit will only update the branch-pointer, if this is possible (for details refer to Section 2.5). If not possible, this option behaves like **Record sources to prepare real merge commit** which will perform the merge, record the source commits and leave the workspace in merging state. You may then review the merge results, tweak the merge (if necessary) and finally commit the merge.

With **Don't record sources to prepare simple commit** set, the content will be merged in the usual way, but the merge sources won't be recorded. When committing the result, the merge will show up as a simple commit in the log, i.e. it will have no reference to the merge source. In this way the merged commits have been condensed into a single commit.

Tip	**Don't record sources to prepare simple commit** can be useful to condense a series of intermediate/temporary commits e.g. after having finished a larger feature.

3.4.4 Cherry Pick

Use this command to "merge" certain commits to the current branch (actually, cherry-picking is no real merge as it does not record the source commits.

Commits displayed in grey already belong to the current branch, commits displayed in black are mergable.

3.4.5 Rebase

Use this command to "apply" (or "rebase") certain commits from one branch to another.

Tip	This command is in particular useful to keep the history of a repository linear.

3.4.6 Add Branch

Use this command to create a branch at the current commit.

3.4.7 Add Tag

Use this command to create a tag at the current commit.

3.4.8 Branch Manager

Use this dialog to get an overview over all branches or to delete some of the branches.