SmartGit Quickstart Guide

syntevo GmbH, www.syntevo.com

2010

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 applications are software projects, documentation projects or website projects.

Acknowledgments

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

Chapter 2

Git-Concepts

The following section helps you to easily start-up with Git and tries to give you an understanding of the elemental concepts of Git for efficient working.

2.1 Repository, Working Tree, Commit

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

Classical centralized version control systems like CVS or Subversion (SVN) use one central repository and local working copies. CVS/SVN working copies can just refer to parts of the central repository. With Git everything is a repository, even the local "working copy" which always refers to a complete repository, not just to parts of it. Git's *working tree* is the directory where you can edit files. Each working tree has its corresponding repository. So-called *bare repositories* which are used on servers in the meaning of central repositories don't have a working tree.

> **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.

2.2 Typical Project Life-Cycle

As with all version control systems, there typically exists a central repository containing the project files. To create a local repository, you need to *clone* the *remote* central repository. Then the local repository is connected to the remote repository which, from the local repository's perspective, is referred to as *origin*. The cloning-step is comparable with the initial SVN checkout for getting a local working copy.

Having the local repository now containing all project files from origin, you can make changes to the files in the working tree and *commit* the changes. These changes will be stored in your local repository only, you don't even need to have access to a remote repository when committing. Later on, after you have committed a couple of changes, you can *push* (see 3.2.1) them back to the remote repository. Other users which have also their own clone of the origin repository, *pull* (see 3.2.2) from the remote repository to get your pushed changes.

2.3 Branches

Branches can be used to store independent commits in the repository, e.g., to fix bugs for a released software project and continue to develop new features for the next project version.

Git distinguishes between two kinds of branches: *local branches* and *remote branches*. In the local repository, you can create as much local branches as you like. Remote branches refer to local branches of the origin-repository. In other words: cloning a remote repository clones all its local branches which then are stored in your local repository as remote branches. You can't work directly with the remote branches, but have to create local branches for them which are "linked" to the remote branches. From the local branch's point of view, the corresponding remote branch is called *tracking branch*. There can be independent local branches and local branches with tracking branches.

The default local main branch which Git creates is named *master* (the SVN equivalent is the "trunk"). When cloning a remote repository, the master tracks the remote branch *origin/master*.

2.3.1 Working With Branches

After you have pushed changes from your local branch to the origin-repository, the tracking branch will get those changes, too. If you pull changes from the origin-repository, these commits will be stored in the remote tracking branch of the local 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 *Pull* command in SmartGit or later by explicitly invoking the *Merge* command.

2.3.2 Branches Are Just Pointers

Every branch is simply a named pointer to a commit. A special unique pointer for every repository is the *HEAD* which indicates the commit to which the working tree belongs. The HEAD cannot only point to a commit, but also to a local branch which itself points to a commit. Committing changes will create a new commit on top of the commit where the HEAD is pointing to. If the HEAD points to a local branch, the branch pointer will be moved forward to the new commit, so also the HEAD indirectly points to the new commit. If the HEAD points to a commit, the HEAD itself is moved forward to the new commit.

2.4 Commits

A *commit* is the equivalent to an SVN revision, a set of changes which are stored in the repository with a commit message. The Commit command is used to store working tree changes in the local repository creating a new commit.

2.4.1 It's All About Commits

Let's 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 will set the HEAD and working tree to the commit. Then you may alter the working tree and commit your changes which will create a new commit to the repository which will have the previously checked out commit as its parent. Newly created commits are called *heads*, because there are no other commits descending from them. This way you are extending the commit graph.

2.4.2 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 a 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 to the 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 the commit `004`, too.
This was the completely Push-Pull-Merge cycle when working with remote repositories.

2.5 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's content from your working tree into 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 working tree 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 or 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.

2.6 Merging and Rebasing

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 offers different types of merges. If there are no local changes in the local branch and you pull remote changes, it is not necessary to create a merge commit. Instead it is sufficient for Git to just move the local branch pointer forward (*fast-forward* merge).

Usually, when merging from a different branch and both, the current branch and the branch to merge contain changes, there are two options to merge: the normal merge and the *squash merge*. In case of the normal merge a merge commit with at least two parent commits (the last from your current branch and the last from the merged branch) is created. The squash merge will drop the information about the merged branch, just as you would have modified the working tree manually, and hence create a normal commit. This is very useful to merge changes from local (features) branches where you don't want all your feature branch commits be pushed into the remote repository.

Note	Merging is a fundamental concept in Git and SmartGit performs merges automatically in situations where you might not expect it at a first glance. For example, if you are working on 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 merge conflict.

A Git-specific alternative to merging is *rebasing* (see Section 3.4.5). It can be used to keep the history linear.

For example, if a user has done local commits and performs a pull with merge, a merge commit with two parent commits (his last commit and the last commit from the tracking branch) is created. When using rebase instead of merge, Git applies the local commits on top of the commits from the tracking branch, thus avoiding a merge commit. As for merge, this only works if no conflict occurs.

2.7 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.

2.8 Submodules

Often, software projects are not completely selfcontained, but they usually share common parts with other software projects. Git offers a feature called *submodules* which allows to integrate directory structures into another directory structure (similar to SVN's "externals" feature).

A submodule is a nested repository which is embedded in a dedicated subdirectory of the working tree (of its parent repository). It is always pointing at a particular commit of the embedded repository. The definition of the submodule is stored as a separate entry in the git object database of the parent repository.

The link between working tree entry and foreign repository is stored in the `.gitmodules` files of the parent repository. The `.gitmodules` file is usually versioned, so it can be maintained by all users respectively changes are propagated to all users.

Submodule repositories are not automatically cloned, but need to be initialized first. The initialization arranges necessary entries in the `.git/config` file which can be edited later by the user, e.g., to fix SSH login names.

2.9 Git-SVN

Git allows not only to communicate with other Git repositories, but also with SVN repositories. This means that you can use SmartGit also as a simple SVN client:

- Cloning from a SVN repository is similar to checking out a SVN working copy.
- Pulling from the SVN repository is similar to updating the SVN working copy.
- Pushing to the SVN repository is similar to committing from the SVN working copy to the SVN server.

In addition to a normal SVN client, you can use all (local) Git features like local commits and branching. SmartGit integrates all SVN operations transparently, so you almost never need to care which server VCS is hosting your main repository.

Chapter 3

Important Commands

This chapter gives you an overview of the SmartGit 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 working tree.

You need to specify the local directory which you want to open. If the specified directory is not a 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

The following 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 2.3).

3.2.2 Pull

Use this command to pull commits from a remote repository.

After successfully fetching the commits of the remote repository, they are stored in the local branches corresponding to the remote branches. The changes have to be merged into those 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 any conflicts 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 remote tracking branch to the local branch.

Note	When you fetch *submodules* 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`).

3.2.3 Synchronize

Use this command to store local commits to a remote repository and pull commits at the same time.

Usually, you have to use the Push (see 3.2.1) and Pull (see 3.2.2) commands to keep your repository synchronized with a remote repository. Using **Synchronize** simplifies the task a little bit.

In the case of local and remote commits, the invoked push command fails. But the pull command is performed nevertheless, so at least the commits from the remote repository are available in the tracking branch and are ready to merge or rebase. If the remote changes were applied to the local branch, you can invoke the **Synchronize** command again.

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 2.5) 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 not to 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) manually.

3.3.4 Commit

Use this command to save local changes in the local repository.

If the working tree is in merging state (see Section 2.6), you can only 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 2.5).

Warning! It is strongly recommende not to 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 2.5) 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 local 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, so restoring the content might be impossible!

3.4 Branch Handling

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 remote tracking 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 command (see 3.4.3) 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 branch 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 remote tracking 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 2.6). 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 it.

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 of all branches or to delete some of the branches.

Chapter 4

System properties/VM options

Some very fundamental options, which have to be known early at startup time or which typically need not to be changed are specified by Java VM options instead of SmartGit preferences.

Options suppied to the VM are either actual *standard* or *non-standard* options, like `-Xmx` to set the maximum memory limit, or *system properties*, typically prefixed by `-D`. This chapter is mainly about SmartGit-specific system properties.

4.1 General properties

Following general purpose properties are supported by SmartGit.

smartgit.home

This propery specifies the directory into which SmartGit will put its configuration files; refer to Section 5 for details. The value of `smartgit.home` may also contain other default Java system properties, like `user.home`. It may also contain the special `smartgit.installation` property, which refers to the installation directory of SmartGit.

> **Example**
> To store all settings into the subdirectory `.settings` of SmartGit's installation directory, you can set `smartgit.home=${smartgit.installation}\.settings`.

4.2 User interface properties

smartgit.splashScreen.show

This property specifies whether to show the splash screen on startup or not. It defaults to `true`.

Example
Use `smartgit.splashScreen.show=false` to disable the splash screen.

4.3 Specifying VM options and system properties

Depending on your operating system, VM options resp. system properties are specified in different ways.

smartgit.properties file

The `smartgit.properties` file is present on all operating systems. It's located in SmartGit's *settings directory*; refer to Section 4.1 for details. All *system properties* can be specified in this file.

Note	*System properties* are VM options which would be specified by the `-D` prefix when directly providing them with the start of the `java` process. All options listed in this chapter are *system properties* and hence can be specified in the `smartgit.properties` file.

Every option is specified on a new line, with its name followed by a "=" and the corresponding value.

Example Add

```
smartgit.splashScreen.show=false
```

to disable the splash screen.

Microsoft Windows

VM options are specified in `bin/smartgit.vmoptions` within the installation directory of SmartGit. You can also specify system properties by adding a new line with the property name, prefixed by `-D`, and appending `=` and the corresponding property value.

Example Add the line

```
-Dsmartgit.splashScreen.show=false
```

to disable the splash screen.

Apple Mac OS X

System properties are specified in the `Info.plist` file. Right click the `SmartGit.app` in the Finder and select **Show Package Contents**, double click the `Contents` directory and there you will find the `Info.plist` file. Open it in a text editor of your choice. Specify the system properties as key-string pairs in the `dict`-tag after the `key` with the `Properties` content.

Example Use the following key-string pairs

```
<key>Properties</key>
<dict>
...
<key>smartgit.splashScreen.show</key>
<string>false</string>
</dict>
```

to disable the splash screen.

Specify a VM option by placing them in the `string`-tag to the `VMOptions` array.

Unix

System properties are specified e.g. in `bin/smartgit.sh` within the installation directory of SmartGit. You can specify a property by adding the property name, prefixed by `-D` and appending `=` and the corresponding property value to the `_VM_PROPERTIES` environment variable. Multiple properties are simply separated by a whitespace; make sure to use quotes when specifying several properties.

Example Add

```
_VM_PROPERTIES="$_VM_PROPERTIES -Dsmartgit.splashScreen.show=false"
```

before the `$_JAVA_EXEC` to disable the splash screen.

Chapter 5

Installation and Files

SmartGit stores its configuration files per-user. The root directory of SmartGit's configuration area contains subdirectories for every major SmartGit version, so you can use multiple versions concurrently. The location of the configuration root directory depends on the operating system.

5.1 Location of SmartGit's settings directory

- **Windows:**: The configuration files are located below `%APPDATA%\syntevo\SmartGit`.
- **Mac OS:**: The configuration files are located below `~/Library/Preferences/SmartGit`.
- **Unix/Other:**: The configuration files are located below `~/.smartgit`.

Tip	You can change the directory where the configuration files are stored by the system property smartgit.home (see 4.1).

5.2 Notable configuration files

- `accelerators.xml` stores the *accelerators* configuration.
- `credentials.xml` stores authentication information, except the corresponding passwords.
- `license` stores your SmartGit's *license key*.
- `log.txt` contains debug log information. It's configured via `log4j.properties`.
- `passwords` is an encrypted file and stores the *passwords* used throughout SmartGit.
- `projects.xml` stores all configured *projects* including their settings.

- `settings.xml` stores the application-wide Preferences of SmartGit.
- `uiSettings.xml` stores the *context menu* configuration.

5.3 Company-wide installation

For company-wide installations, the administrator can install SmartGit on a network share. To make deployment and initial configuration for the users easier, certain configuration files can be prepared and put into the subdirectory `default` (within SmartGit's installation directory).

When a user starts SmartGit for the first time, following files will be copied from the `default` directory to his private configuration area:

- `accelerators.xml`
- `credentials.xml`
- `projects.xml`
- `settings.xml`
- `uiSettings.xml`

The `license` file (only for *Enterprise* licenses and 10+ users *Professional* licenses) can also be placed into the `default` directory. In this case, SmartGit will prefill the **License** field in the **Set Up** wizard when a user starts Smartgit for the first time. When upgrading SmartGit, this `license` file will also be used, so users won't be prompted with a "license expired" message, but can continue working seamlessly.

Note	Typically, you will receive license files from us wrapped into a *ZIP* archive. In this case you have to unzip the contained `license` file into the `default` directory.

5.4 JRE search order (Windows)

On Windows, the `smartgit.exe` launcher will search for an appropriate JRE in the following order (from top to bottom):

- Environment variable SMARTGIT_JAVA_HOME
- Sub-directory `jre` within SmartGit's installation directory
- Environment variable JAVA_HOME

- Environment variable JDK_HOME
- Registry key HKEY_LOCAL_MACHINE\SOFTWARE\JavaSoft\Java Runtime Environment