

02 Version Control

Version control, also known as revision control and source control, is the management of changes to information. A version control system keeps track of software versions throughout development. A distributed version control system allows many developers to work on a given project without requiring that they maintain a connection to a common network.

Introduction

We use [git](#) for distributed LabVIEW code version control. Our code repositories are managed externally on a [GitLab](#) server and internally on a Git server.

In order to be able to work on our LabVIEW code you need to follow the steps described here.

How to setup

1. Create a GitLab User

Go to gitlab.com/users/sign_up and create a free account. Then let us know about your username so we can grant you access to our repositories.

2. Install Git

Install git on your development machine.



There's an in-depth description of [Working with Git on Windows](#) available at beanstalk. It covers the installation of Git for Windows as well as installing SSH keys.

Git itself is a collection of tools that are operated from the command line (or bash). However, there are GUI clients available for all major operating systems:

- [Tower](#) (the tool of our choice, for Mac OS X and Windows)
- [SourceTree](#) (a free alternative to the Tower app, for Windows and Mac)

3. Configure git

If you're using a GUI client, some of these settings may be done via its settings dialog.

Git checks 4 places for a configuration file, cascading settings in the following order:

- Your machine's system `.gitconfig` file.
- Your user `.gitconfig` file located at `~/.gitconfig`.
- A second user-specific configuration file located at `$XDG_CONFIG_HOME/git/config` or `$HOME/.config/git/config`.
- The local repo's config file `.git/config`.

If you haven't been asked for your name and your email address during installation, you can manually set this information:

```
git config --global user.name "Your Name"
git config --global user.email "your@email.address"
```

Example `.gitconfig` file

```
[user]
  name = Your Name
  email = your@email.address
[core]
  excludesfile = /some/path/to/.gitignore_global
  filemode = false
```

Ignoring files

The `excludesfile` mentioned in the `.gitconfig` file (the *global `.gitignore` file*) defines which files to ignore (exclude from versioning) for all repositories on the computer its stored on. You can set the path of the global `excludesfile` to the actual user's home directory with

```
git config --global core.excludesfile '~/.gitignore'
```

Each repository and each directory in it can have its own `.gitignore` file.

Common `.gitignore` example

```
# Generic: Temporary Files and Logs
*~
.*_*
.DS_Store
Thumbs.db
~*.docx
~*.xlsx
*.log*
```

Project-specific `.gitignore` example

```
# LabVIEW Metadata
```

```
*.aliases
*.lv\ps
.cache/
# LabVIEW Binaries/Builds
artifacts/
builds/
*.lvlibp
```

Windows Specifics

If you're using msysgit on Windows:

- the machine's system `.gitconfig` file resides at `%PROGRAMFILES(X86)%\Git\etc\gitconfig`.
- your user `~/.gitconfig` file resides at your `%HOMEPATH%`

Use `echo %PROGRAMFILES(X86)%` and `echo %HOMEPATH%` from a Windows command prompt to find the specific locations.

Global `.gitignore` file

Set the global `.gitignore` file location to `C:/users/{myusername}/.gitignore`:

```
git config --global core.excludesfile "%USERPROFILE%\.gitignore"
```

The above command will only set the location of the ignore file that git will use. The file has to still be manually created in that location and populated with the ignore list.

Environment Variables

Open up My Computer → Advanced System Settings → Environment Variables. Then:

1. Add the git binary path (eg `C:\Program Files (X86)\Git\bin`) to the PATH environment variable
2. Add `msys` to the TERM environment variable

File Permissions

If your repository is on a filesystem whose executable bits are unreliable (like FAT), git should be configured to ignore differences in file modes recorded in the index and the file mode on the filesystem if they differ only on executable bit:

```
git config --local core.filemode false
```

and/or

```
git config --global core.filemode false
```

4. Get an SSL Certificate

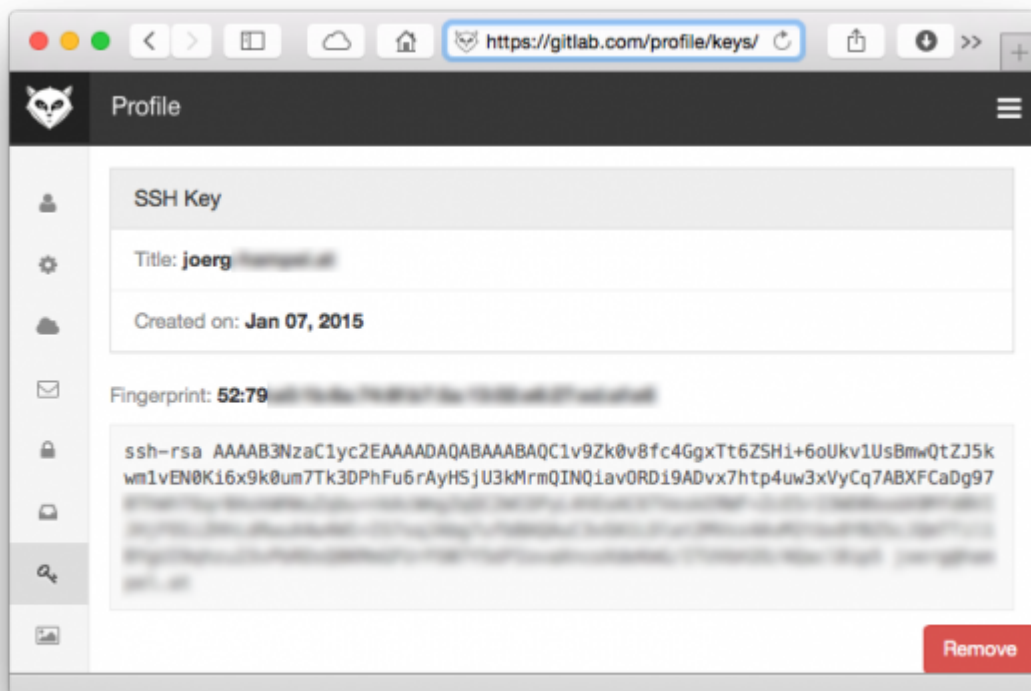
To establish a secure connection between your computer and GitLab via Secure Shell (SSH) you will need to create and install SSH keys. The private key resides on your computer, the public key is stored at the GitLab server.

Create SSH keys

- On a Mac, follow the steps in the [GitLab SSH keys help](#)
- On Windows, see the [beanstalk article](#) mentioned above

Set public key in gitlab.com

Add your key by navigating to the 'SSH Keys' section in your user profile, selecting 'Add SSH Key' and copy-pasting the *public* key to the 'key' field. Please copy the complete key (a long string starting with ssh- and ending with the comment you specified, usually your email address).



Server certificate verification failure

For some setups, the SSL certificate verification will fail. Error messages might be:

```
error: SSL certificate problem, verify that the CA cert is OK. Details:
error:14090086:SSL routines:SSL3_GET_SERVER_CERTIFICATE:certificate verify
failed while accessing (repo)
```

or

```
Cloning into 'c:\repo-path'...
ssh_exchange_identification: Connection closed by remote host
fatal: Could not read from remote repository
```

If it is acceptable to turn off the SSL validation instead of actually solving the issue this will turn off validation for the current repo

```
git config --local http.sslVerify false
```

If you would rather have this as a default behaviour for git then the following will do it for all repos

```
git global --local http.sslVerify false
```

and for those that would rather add to the .git/config file directly the entry looks like

```
[http]
    sslVerify = false
```

How to use

We created a test repository to let you check your installation and make your first steps.

1. Get code

To get the code you need to *clone a remote repository*:

```
git clone git@gitlab.com:hampel-soft/hse/hampelsoft-test.git
```

Beware: SSH clients use the `known_hosts` file to authenticate the server, to check that it isn't connecting to an impersonator. The first time you connect to a new server, you have to acknowledge the authenticity of the server by accepting the ECDSA key fingerprint. Some GUI clients hide the appropriate message, so if you get permission errors, try cloning from the command line.

Git submodule

A submodule is a repository embedded inside another repository. We provide a detailed description and other resources on our [Best Practices page on git submodules](https://dokuwiki.hampel-soft.com/).

The `hampelsoft-test` repository contains the `hampelsoft-test-sub` repo as a submodule. After cloning the repository, the submodule directory is empty (as it is uninitialized). The following command pulls the contents of the submodule repository into the local submodule folder:

```
cd hampelsoft-test
git submodule update --init --remote --recursive
```

2. Work locally

To propose files for version controlling (to make git work with your files) you need to *stage* the files:

```
git add README.txt
```

To actually add new or changed files to the repository, you need to *commit* them:

```
git commit -m "first commit"
```

3. Share with others

To send your changes to the central, remote repository, you need to *push* your local repository:

```
git push -u origin master
```



Be aware that you **cannot** push to the origin of our `hampelsoft-test` and `hampelsoft-test-sub` repositories as you do not have sufficient access rights. Please read up on [workflows](#) and the [project forking workflow](#) in particular for more information.

SCC Best Practices

You can find more information on how to work with(in) Source Code Control in our [Best Practices](#) section:

- [Source Code Control](#)
 - [Rules & Guides](#)
 - [Workflow Model](#)
 - [Submodules](#)

Version Numbering

Please read [Version Numbers](#) for more information on our version number formats and policy.

Git Tags

Git does not give perfect integer revision numbers that increment on each commit. Instead, git hashes the commit data and uses that as a unique identifier of the commit. That means one commit could be 8d844d8d... and the next f1728db4..., which is incomprehensible AND unusable as (part of) our [version numbering convention](#).

However, git allows you to create descriptive labels for marking specific points in history as being important, i.e. creating pointers to specific commits. These pointers are called *tags*. Typically, tagging is used for marking release points (v1.2.3, and so on), and that's exactly what we do, as well.

Read the [Git Documentation](#) for detailed info on how to use tags, and the [Tower Documentation](#) on how to create tags with Tower.

Further Resources

General

- [Official Git Tutorial](#)
- [GitLab Documentation](#)
- [Simple git how-to in german](#)

Password Management

By courtesy of Fabiola De La Cueva

pageant is a tool that runs at startup so you only have to enter your password once:

<http://blog.shvetsov.com/2010/03/making-pageant-automatically-load-keys.html>

Compare / Merge VIs

By courtesy of Fabiola De La Cueva

Instructions on how to configure your client tool to use LabVIEW Compare and LabVIEW Merge towards the end of this article:

<http://delacor.com/configuring-hg-or-git-to-use-labview-compare-and-labview-merge/>

Internal

HSE-internal resources

- [Infrastructure - SCC](#)

From:

<https://dokuwiki.hampel-soft.com/> - **HAMPEL SOFTWARE ENGINEERING**

Permanent link:

<https://dokuwiki.hampel-soft.com/processes/version-control?rev=1579173572>

Last update: **2020/01/16 11:19**

