Introduction to Git and GitHub

Daniel Zeman (zeman@ufal.mff.cuni.cz)

June 2024

Platforms

This guide focuses on two platforms which I am most familiar with: Microsoft Windows and Ubuntu. Moreover, the text devoted to Microsoft Windows is much more detailed, while in the Ubuntu sections I focus on things that are done differently; therefore I recommend that you read the Windows sections even if Windows is not your platform.

To some extent, what I say about Ubuntu should generalize to other Linux distributions and even to other systems including MacOS; however, there will be differences, especially when installing stuff.

Git and GitHub in Windows

- Download and install Git for Windows (https://git-scm.com/download/win). This is the core engine that will be responsible for the communication between your computer and GitHub. It can be run from the command line (cmd.exe). However, in Windows we will typically control it through a graphical interface called TortoiseGit (see below). (If you want to see how to work with git in the command line, read the Ubuntu section below. The git commands are the same on both operating systems.)
    - During the installation, your name and e-mail address will be requested. They do not have to match your user data on GitHub. They will only serve as a signature that will be attached to every revision of your data repository. They are *not* used for GitHub to recognize you as a user who has the permission to upload the changes to GitHub's servers.
- Download and install Tortoise Git (https://tortoisegit.org/download/). This is a graphical user interface for Git, accessible via right-mouse click on a file or folder in the Windows Explorer.
- You will also need two programs from the PuTTY suite: PuTTYgen and Pageant. No need to install them separately if you did not have them before – you got them now together with Tortoise Git. If you had them before, you have now two versions at different locations. In theory it does not matter which one you use, except that if you use a version that is too old, it may be using protocols or cipher algorithms that are no longer considered safe and GitHub may not support them.
- You will run PuTTYgen just once to generate a pair of public and private key for your access to GitHub. If for some reason you already have such a pair of keys, you probably can re-use it and do not have to generate a new one. The public key should be uploaded to your profile on GitHub. Rather than a key, you can think of it as a keyhole. Whoever will have a passing private key will be granted rights connected to your GitHub user account, e.g.,

changing the contents of your treebank repository. The private key should be stored in a file on your computer and it will be ciphered with a password so that gaining access to your computer will not automatically give the intruder access to your GitHub account.

- PuTTYgen is just one window. First click on the “Generate” button, then follow the instructions and move your mouse pointer randomly, until the program is satisfied and the key is generated. Then enter and repeat the password to protect the private key. Then save the private key somewhere to your disk in a file with the extension “.ppk”, e.g., “private-key.ppk”. Then copy directly from the window the string with the OpenSSH version of the public key (beware that the button “Save public key” might generate wrong format that GitHub won’t like). Locate your profile on GitHub web, click on Settings (https://github.com/settings/profile), select “SSH and GPG keys” in the menu, select “New SSH key” and paste the text that you copied from the PuTTYgen window.
- The Pageant program on your computer will relieve you from the necessity to enter your password during every interaction with GitHub. When you launch the program, you tell it where to find the file with your private key, upon request you provide the password to decipher the key. Pageant will keep it deciphered in memory and will stay running in background, ready to provide your private key every time when Tortoise Git needs it. As long as Pageant runs, you do not have to provide your password again. If you restart your computer (or manually terminate Pageant), you will have to run Pageant anew, open the file with the key and enter the password.
    - Optionally you can set up Windows so that it launches Pageant automatically after each system restart, and Pageant will load your private key. First you need to locate Pageant on your computer, e.g., C:\Program Files\TortoiseGit\bin\pageant.exe. When you find it, click on it with your right mouse button, select “Copy”, then right-click another location (e.g. Desktop) and select “Create a link”. It will create a new file “pageant.exe – link”, you can rename it if you want to. Right-click it, select “Properties”, in the dialog window select “Link” and in the field “Run in” enter the path to the folder where you saved your private key, e.g., “C:\Users\Dan\Documents\Keys”. Then enter the name of the file with the key to the “Target” field, so that you have there both the path to Pageant and the path to the key, e.g. “"C:\Program Files\TortoiseGit\bin\pageant.exe" "private-key.ppk"”. Save the Properties by clicking on OK. The last step is to copy (or move) this link file to a folder where Windows looks for programs to be launched automatically after restart. Click on the link file, press CTRL+C (or right-click it and select “Copy”), then in the Windows Explorer enter “shell:startup” in the address line and press Enter. In my case it jumps to the folder “C:\Users\Dan\AppData\Roaming\Microsoft\Windows\Start Menu\Programs”, but in your case the path will differ at least in the user name, maybe more. Go to this folder and paste the link file here (CTRL+V). The Pageant should start automatically after the next system restart and it should prompt you to enter the password for your private key.
- Everything is installed, now we can perform the initial unpacking of our local copy of the GitHub repository. Open the browser and go to the repository page on GitHub (e.g.,

https://github.com/UniversalDependencies/UD_Spanish-GSD).[1] Slightly to the right from the center there is a conspicuous green button "<> Code". Click on it, a tab named "Local" will appear, with sub-tabs "HTTPS", "SSH", and "GitHub CLI", select the middle one, i.e., "SSH". You will see the address that Tortoise Git on your computer needs to synchronize with your GitHub repository: "git@github.com:UniversalDependencis/UD_Spanish-GSD.git". Copy it to the clipboard. Then in Windows Explorer on your computer, go to the folder where you want to have a copy of the repository as a sub-folder, for example C:\Users\Dan\Documents. Right-click the folder and select "Git Clone…" from the context menu. A window will pop up, the important field in the window is URL, which should already have pre-filled the address from your clipboard, that is, "git@github.com:UniversalDependencies/UD_Spanish-GSD.git". Click OK. A local copy of the repository will be downloaded and unpacked. In Windows Explorer, the icon of the folder will be accompanied by a green circle with a check mark, signaling that there are no unsynchronized local changes.

- If the repository holds a UD treebank, it has two branches called "master" and "dev". The master branch has been opened by default but you must work with dev. Switch to dev now and never return to master (master can be modified only as a result of UD release process). Right-click the new folder and select "TortoiseGit / Switch/Checkout…" from the context menu. Select branch "dev" (or, if it does not exist yet, select "remotes/origin/dev") and press OK.
- Right-click the new folder and select "TortoiseGit / Pull…" from the context menu. The Pull operation contacts the remote server (GitHub) and inquires about any changes uploaded by other people, that you do not have in your local copy yet. If so, the changes will be downloaded and integrated into your local version. Now, right after cloning the repository, there is probably nothing new, but in the future, Pull will be one of the most important operations you will be doing.
- If you edit and save a versioned file in your local copy of the repository, the icon of the file (as well as all superordinate folders that are part of the repository) will change and a red circle will warn you that there are local changes that have not yet been saved as a new revision in the repository's history. Two more steps are needed: Commit and Push. Commit will save a new revision to the history of the repository; if needed, you will later be able to return to this revision. However, Commit does not communicate with the GitHub server: Only the local copy of the history is modified. To commit new revision, right-click the local copy of the repository, select "Git Commit → "dev"…", enter a short description of the changes in the "Message" field, then press the "Commit" button. Once Commit is done, you will be offered the second step, "Push"; you need to be online to perform it. TortoiseGit will ask GitHub whether there are any changes by other users in the meantime. If so, Push will fail and you will be asked to first synchronize your local copy using Pull, then retry Push. If there are no new changes on the server, Push will be performed right away.

1 This tutorial assumes that you already have access to an existing repository on GitHub. It does not explain how to create a repository.

- If you want to add a new file to the repository, put the file in the folder with the local copy of the repository, right-click it and select “TortoiseGit / Add…”. Then do Commit and Push.
- You can now go to your web browser, reload the repository’s page on GitHub and verify that your changes are there. GitHub will show you the commit history here (adjust the URL to your repository’s name): https://github.com/UniversalDependencies/UD_Spanish-GSD/commits/dev. Use the Code tab to browse the repository’s contents. Make sure you are looking at the dev branch (https://github.com/UniversalDependencies/UD_Spanish-GSD/tree/dev)! Thus the Code tab shows the files that are synchronized with your computer via TortoiseGit. In contrast, the Issues tab (https://github.com/UniversalDependencies/UD_Spanish-GSD/issues) holds the discussions about bugs in this repository and they are not part of the repository’s contents, you only see them on the web. (Nevertheless, it is easy to link files in the repository from the issues, as well as to link from one issue to another.)

Git in Ubuntu

If you are not unlucky and git-related packages were not unchecked when the system was installed, you probably already have git. Launch bash in a terminal window and type “which git”. If the system responds with a path, such as “/usr/bin/git”, you’re fine. If not, then there are two possibilities:

1. Git has been installed on your system but your shell environment does not know about it (your $PATH variable does not contain a path in which the git executable can be found). It is difficult to give a general advice how to locate git and fix it, so maybe just try to (re)install it (see below).
2. Git has not been installed on your system. You need superuser privileges to install it using this command: “sudo apt install git-all”.

You will also need the “ssh” program but it should have been installed with the system. Together with ssh, you should also have the utility “ssh-agent” (doing the job of Pageant in Windows).

In contrast to the Windows section above, here we will describe how to work with git in the command line (bash).

However, before we start working with git, we need to set up public and private keys so that we do not have to enter a password every time we want to synchronize our local copy with GitHub. You may also want to consult GitHub’s documentation on this. Use ssh-keygen to generate a pair of public and private key if you don’t already have one:

```
ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
```

When prompted, you should provide a password (passphrase) to secure the file with your private key.

Your public key must be uploaded to your user profile on GitHub (see the Windows section above for details).

Finally, you want to run (or make sure it is already running) the ssh agent with your private key (you may need to adjust the path to the key in the command below).

```
ssh-add ~/.ssh/is_rsa
```

I have created the following script in my home folder (~/bin) and named it `keylogin.sh`:

```
#!/bin/bash

if [[ $_ == $0 ]]; then
  echo "This script needs to be sourced!" >&2
  exit 1
fi

if [ -z "$SSH_AGENT_PID" ]; then
  echo "Launching ssh-agent"
  eval `ssh-agent`
fi

ssh-add ~/.ssh/id_rsa

echo Use ssh -A to forward the ssh agent connection '(provided you trust
superusers on the remote host).'
```

Then I added the following lines to ~/.bashrc:

```
export PATH=$HOME/bin:$PATH
alias key='source keylogin.sh'
```

Then I can just type “key” in my terminal window, it will ask me for the password for my private key, and from that point on I can work with GitHub without having to provide the password again. However, unlike with Pageant in Windows, this works only in the current terminal window and its child processes. If I launch a new terminal session, independent of the previous one, I will have to run “key” again.

Optional: There is a script by Shawn O. Pearce which changes the prompt in your shell so that you immediately see the status of the git folder you are in. You can download the script from https://ufal.mff.cuni.cz/~zeman/git-prompt.sh.txt, rename it (remove the “.txt” extension), put it to your home folder (e.g., `~/bin/git-prompt.sh`), then add the following lines to your `~/.bashrc`:

```
source ~/bin/git-prompt.sh
export GIT_PS1_SHOWDIRTYSTATE=1
export GIT_PS1_SHOWSTASHSTATE=1
export PS1='[\t]\[\033[1;31m\]\h:\w\[\033[01;95m\]$(__git_ps1 "(%s)")\
[\033[1;36m\]>\[\033[0m\] '
```

The Git Workflow in the Command Line

Everything is installed and configured, so we can create a local copy of an existing GitHub repository and start working with it. See the Windows section above on how to figure out the URL

of your repository. When you have the address, put it in the git clone command below. Switch to the dev branch immediately after cloning the repository.

```
git clone git@github.com:UniversalDependencies/UD_Spanish-GSD.git
git checkout dev
```

When you are inside the local copy of the repository, you can run git pull to get new changes (if any) from GitHub.

```
git pull
```

Sometimes git annoys me with opening an editor and asking for a comment when I run git pull, so I have created an alias, called “gp”, which does the pulling and does not open the editor. I added the following line to my ~/.bashrc, then I can call “gp” instead of “git pull”:

```
alias gp='git pull --no-edit'
```

You can use “git status” to see if the working copy contains changes that have not been committed (but if you installed the optional git-prompt.sh above, an asterisk in the prompt will tell you about such changes). You can use “git diff” to see what the changes are:

```
[12:57:44]zen:/net/work/people/zeman/unidep/UD_Spanish-GSD(dev *)> git status
On branch dev
Your branch is up to date with 'origin/dev'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
[13:01:31]zen:/net/work/people/zeman/unidep/UD_Spanish-GSD(dev *)> git diff
diff --git a/README.md b/README.md
index 3e218de..51a7143 100644
--- a/README.md
+++ b/README.md
@@ -1,4 +1,4 @@
-# Summary
+d# Summary

 The Spanish UD is converted from the content head version of the [universal
 dependency treebank v2.0 (legacy)](https://github.com/ryanmcd/uni-dep-tb).
[13:01:33]zen:/net/work/people/zeman/unidep/UD_Spanish-GSD(dev *)>
```

Finally, use “git commit -a” and “git push” to save a new revision in the history and to upload the revision to GitHub. Do not forget to add a short description of the changes after the -m switch:

```
git commit -a -m "Messed with the first character of the README :-)"
git push
```

Github Issues

Each repository in Github has its own “issue tracker”. You can access it in your browser when you go to the page of the repository and click on the “Issues” tab. In the case of Spanish GSD treebank repository, which we used as an example above, the issue tracker is at https://github.com/UniversalDependencies/UD_Spanish-GSD/issues. Note that in the specific case of Universal Dependencies, the issue tracker of a treebank repository should be used only for issues (errors) in the given treebank. Any guidelines-related questions (even those that are specific to Spanish) should be discussed in the issue tracker of the docs repository, https://github.com/UniversalDependencies/docs/issues.

Issues are discussions about particular problems in the repository. When the discussion reaches a solution / consensus / result, the issue can be “closed”. It is not removed, it can still be viewed, but it will not be offered in the list of issues by default. It can be reopened if it turns out that more discussion is needed. Any Github user can create an issue for our repository, including users who do not have push access to the repo. But some other actions can only be done by users who have the required permissions. New issue is created by pressing the green button “New issue”, filling out the title and the first comment for the discussion, and pressing “Submit new issue”.

If you submit an issue and then decide to rephrase a part of it, you can edit the already submitted comment. (But people will only get one e-mail notification with the first version of the comment.)

To add a new comment when viewing an issue, write the comment in the edit window, then click on the green “Comment” button. If the discussion has reached an end, you can close the issue by pressing the “Close issue” button (without comment), or enter a comment and press the same button, which now reads “Close with comment”.

The text of the issue (comment) can be formatted using MarkDown syntax; for your comfort, the edit window has a few buttons to help you with commonly used constructs, and it also has a “Preview” tab which shows what it will look like once submitted. There can be links to other issues, to files in the repository, or to any other page on the web. There can be even images (simply drag the file with the image and drop it to the edit window).

Links have the MarkDown syntax

```
[text of the link](https://the.address/folder/file)
```

When referring to a MarkDown file in the repository, you can either refer to the whole file (e.g., https://github.com/UniversalDependencies/UD_Spanish-GSD/blob/dev/README.md – simply locate the file in your browser and copy the URL), or to a particular section (e.g., https://github.com/UniversalDependencies/UD_Spanish-GSD/blob/dev/README.md#changelog – locate the heading of the section, move your mouse to it, click on the link icon that appears to the left of the heading, the URL in the browser’s address line changes, copy it). You can also refer to a particular line or range of lines in the source file. To do that, view the file, then on the top switch from “Preview” to “Code”, then click on the line number. If you want a range of lines, press SHIFT and click on the last line of the range. Then copy the URL from the address line (e.g., https://github.com/UniversalDependencies/UD_Spanish-GSD/blob/dev/README.md?plain=1#L30-L35. Now you may ask what happens when somebody edits the file and pushes the changes to GitHub? The line numbers may now point to different section of the file than at the time

you were creating the link. Fortunately, this is a git repository and it remembers all previous versions (commits). If you want (you do!) your link to always lead to the version that was current when writing the comment, you should use a permalink. After selecting the line range, click on the three dots that appear next to the first line of the range, and click on “Copy permalink”. Now you have the permalink URL in your clipboard (the address line of the browser has not changed, so do not go there), and the URL is different from the above: instead of “main” (the branch name), it contains the commit identification number (e.g., https://github.com/UniversalDependencies/UD_Spanish-GSD/blob/2faf8da780765bc82910157ac6e8c23aaed10745/README.md?plain=1#L30-L35).

Like in e-mails, you can quote other people by inserting “>” before each line with a quoted text. It will then be formatted differently to distinguish the quote from your own reaction. The fastest way to insert a quote is to go up in the discussion thread, locate the comment you want to quote, click on the three dots in the upper right corner of that comment, and select “Quote reply”.

You can also refer to other issues in the same repository, simply by “#” + their number; GitHub will resolve it to a URL that your browser understands. For example, if you put “#3” in your MarkDown source, the browser will see a link to https://github.com/UniversalDependencies/UD_Spanish-GSD/issues/3. You can also link to individual comments in the other issues: in the initial line, e.g., “dan-zeman commented 11 hours ago”, click on “commented”, then take the new URL from the address line (e.g., https://github.com/UniversalDependencies/UD_Spanish-GSD/issues/12#issuecomment-954032290). Whenever you link from a comment in issue A to issue B (or to a comment in it), the page of issue B will also inform the reader that the issue has been referenced, and it will provide a back link. This works across the whole Github, so if you are in your private repository issue tracker and reference, say, an issue in the repository of the UD guidelines (e.g., https://github.com/UniversalDependencies/docs/issues/16), the readers of the target issue will see that you mentioned it.

You can refer to other users by “@” + their user name. Of course, you do not have to do it, you can write “as Dan said above, …”. But if you instead write “as @dan-zeman said above, …”, Github will know that you are talking about a Github user. This may affect whether or not that user gets an e-mail notification, so you should use the “@” notation if you want the reader to pay attention to your comment and maybe reply. (Be careful. If you misspell the user name but the result is an existing user name, that user will now get notifications although they are probably not interested in our repository.) There are four levels of e-mail notifications for each user-repository pair:

- Watching All Activity: The user gets a notification for each new issue in the repository, and each new comment in existing issues in the repository, unless this user is the author of that comment. If you want to know who is watching your repository, see https://github.com/UniversalDependencies/UD_Spanish-GSD/watchers. If you are watching the repository, you probably see an “Unwatch” button at its home page (I don’t see it but I’m the owner of the repository, so I cannot unwatch it).
- Participating and @mentions: You will get notifications for issues in which you added a comment (you are participating) or where someone mentioned you using your @user name. If you are in this state, the button where you can change it is labeled “Watch”.

- Ignore. No notifications even if mentioned using your @user name.
- Custom. You can select types of events for which you want to be notified.

The default notification behavior for a new repository to which you have push access depends on the settings in your user profile. Go to https://github.com/settings/notifications and see if you want to change it.

The e-mail with the notification contains the text of the new comment, although not so nicely formatted as on the Github web. There is a link that will take you to the Github page and display it in your browser. But if you just want to quickly answer in text mode, you can reply to the notification e-mail and it will post your answer as a new comment. (If you do so, you should make sure that the e-mail does not contain unnecessary quotation of the notification e-mail, long signatures etc.)