Version control with git

Stefan Kurtz

Research Group for Genome Informatics
Center for Bioinformatics Hamburg
University of Hamburg

slide content adapted from
https://www.atlassian.com/git/tutorials/what-is-git (introduction)
http://rogerdudler.github.io/git-guide/ (tutorial)

October 23, 2022

Contents

Benefits of version control

Version control systems (VCS)

- category of software tools
- help software developers manage changes to source code over time
- track every modification to the code in a special kind of database
- enable turning back the clock and comparing earlier versions of the code to help fix a regression

code

- is meant in a general sense
  - program code
  - Makefile(s)
  - test-scripts & test data
  - documentation as text-format (e.g. LaTeX)
  - directory structure of development tree

irrelevant for VCS

- object code & executables
- temp. files created in build-process
- word processor files
- PDF-files

Relevance of VCS in software development

source code (in the general sense)

- is like the crown jewels of any project (of students, professionals, . . .),
- is a precious asset whose value must be protected,
- is a repository of invaluable knowledge and understanding about the problem domain that the developer has collected and refined through careful effort

VCS

- protects source code from both catastrophe (e.g. system crashes) and casual degradation caused by
  - human error and
  - unintended modifications

VCS in collaborative projects

- software developers working in teams continually write new source code and change existing source code
- the code for a project, app or software component is typically organized in a structured directory tree
- one developer of the team may be working on a new feature . . .
- while another developer fixes an unrelated bug by changing code
- each developer may make their changes in several parts of the directory tree
- VCS helps teams track every individual change by each contributor, merge them and help resolve conflicts stemming from concurrent work
- VCS are an essential part of modern software development processes within teams as well as for individual projects
- once accustomed to a powerful VCS, many developers wouldn't consider working without one even for non-software projects

Requirements of VCS

An ideal VCS ...

- supports the developer's preferred workflow without imposing one particular way of working
- works on any platform, rather than dictating what operating system or tool chain developers must use
- facilitates a smooth and continuous flow of changes to the code
- tracks a complete long-term change history of every relevant file including
  - its creation,
  - its deletion,
  - all edits to their contents, and
  - metadata, like notes, dates, name of individuals
- allows reverting to an older version of the software to track down a regression in the code

Main features of Git

- by far the most popular VCS
- a collection of tools, each for a specific purpose
- free and open source
- mature
- actively maintained
- portable (i.e. available for all common platforms)
- very efficient (scales well for large projects)
- secure (no information gets lost)
- flexible (supports many different collaborative software development methods)
- distributed, i.e. each developer works on
  - her/his own copy of a directory tree
  - has the full history of the code available
- well integrated with code-hosting services like *github* or *Bitbucket*

Get started in your local file system: git init

```
$ mkdir myproject.git
$ cd myproject.git
$ git init --bare
  Initialized empty Git repository in
  /Users/stefan/myproject.git/
```

- this creates a new bare git repository
- option `--bare` means that no working files will be stored in `myproject.git`

```
$ ls
  HEAD         config       hooks/       objects/
  branches/    description  info/        refs/
```

- make sure that the directory is backed up and can be accessed from anywhere
- the idea is that you use the directory as a remote directory to store and synchronize the changes of your code from anywhere

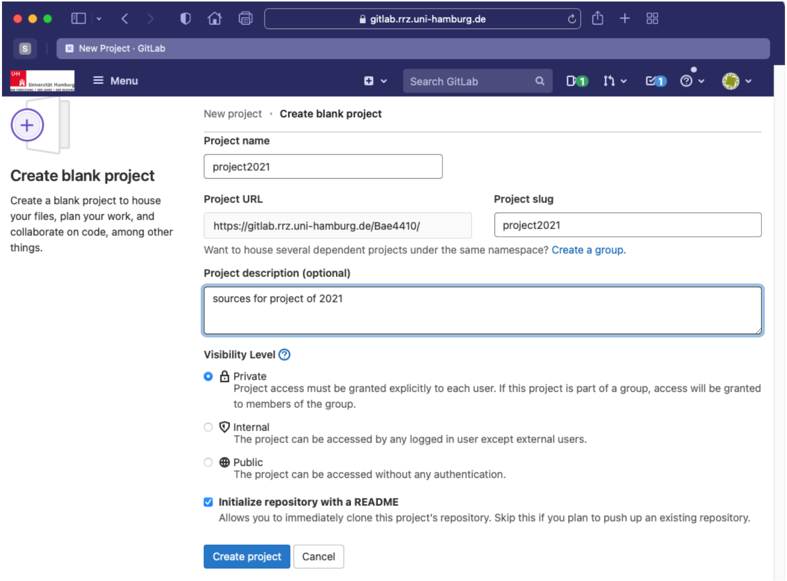
Get started using gitlab@rrz

- open `https://gitlab.rrz.uni-hamburg.de` in your browser
- sign in with your B-account and password
- click on the +-sign, which is likely to the left of the *search GitLab* input field
- choose New project/repository
- enter project name, like *myproject*
- enter a short project description, like *sources for science project*
- keep the standard settings *Private*
- check box *Initialize Repository with README* (important), see slide *Create a GitLab Project* below
- click on *Create Project*
- the browser now shows the new project, see slide *View the project page*

Get started using gitlab@rrz (cont.)

- if you are working on a collaborative project, mouse over *Project information* in the upper left and choose *Members*
- the creator of the project is already listed, see slide *Access Project information/Members* below
- choose *invite member* and click into the field *GitLab member or Email address*
- type in B-Kennung or Name of your co-workers and click the the corresponding line listing the Member (make sure that it is the right person, so use B-Kennung as this is unique)
- select a role, which is likely *Maintainer* (this gives the full right)
- if you only want others to have read access choose *Reporter*
- click on invite, see slide *Invite a Member* below
- the invitee will be informed via E-mail about the invitation as a Member of the project

Create a gitlab project



New project › Create blank project

Create blank project

Create a blank project to house your files, plan your work, and collaborate on code, among other things.

Project name

project2021

Project URL

https://gitlab.rrz.uni-hamburg.de/Bae4410/

Project slug

project2021

Want to house several dependent projects under the same namespace? Create a group.

Project description (optional)

sources for project of 2021

Visibility Level

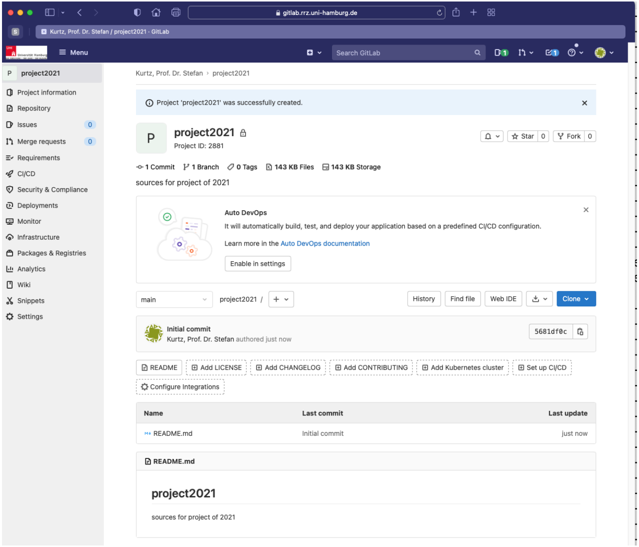
● Private
Project access must be granted explicitly to each user. If this project is part of a group, access will be granted to members of the group.

○ Internal
The project can be accessed by any logged in user except external users.

○ Public
The project can be accessed without any authentication.

☑ Initialize repository with a README
Allows you to immediately clone this project's repository. Skip this if you plan to push up an existing repository.

View the project page



Kurtz, Prof. Dr. Stefan › project2021

Project 'project2021' was successfully created.

project2021

Project ID: 2881

1 Commit 1 Branch 0 Tags 143 KB Files 143 KB Storage

sources for project of 2021

Auto DevOps

It will automatically build, test, and deploy your application based on a predefined CI/CD configuration.

Learn more in the Auto DevOps documentation

Initial commit
Kurtz, Prof. Dr. Stefan authored just now

5681df0c

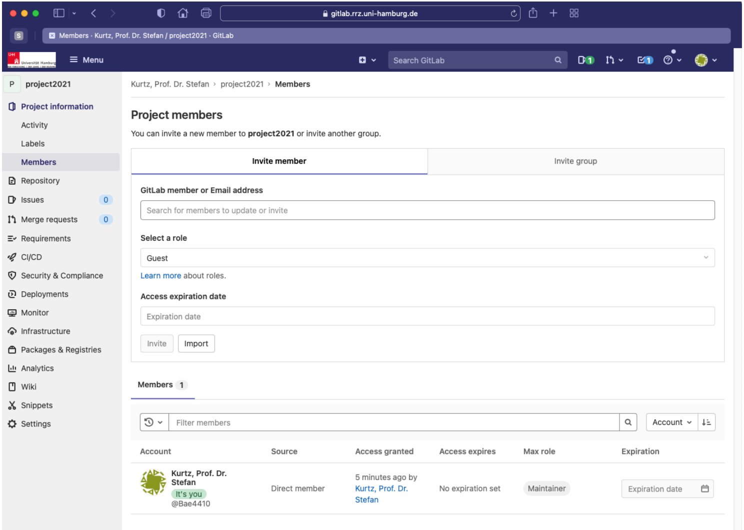
Name	Last commit	Last update
README.md	Initial commit	just now

README.md

project2021

sources for project of 2021

Access Project Information/Members



Kurtz, Prof. Dr. Stefan › project2021 › **Members**

Project members

You can invite a new member to **project2021** or invite another group.

GitLab member or Email address

Search for members to update or invite

Select a role

Guest

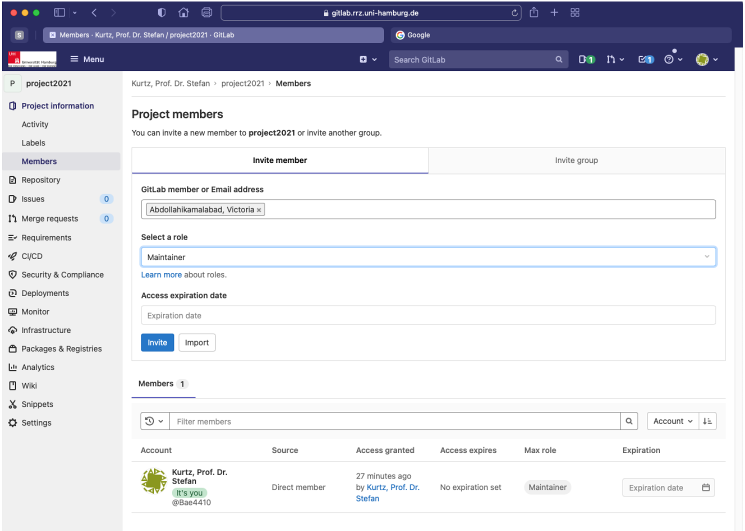
Learn more about roles.

Access expiration date

Expiration date

Account	Source	Access granted	Access expires	Max role	Expiration
Kurtz, Prof. Dr. Stefan It's you @Bae4410	Direct member	5 minutes ago by Kurtz, Prof. Dr. Stefan	No expiration set	Maintainer	Expiration date

Invite a Member



Kurtz, Prof. Dr. Stefan › project2021 › **Members**

Project members

You can invite a new member to **project2021** or invite another group.

Invite member | Invite group

GitLab member or Email address

Abdollahikamalabad, Victoria ×

Select a role

Maintainer

Learn more about roles.

Access expiration date

Expiration date

Invite | Import

Members 1

Filter members | Account

Account	Source	Access granted	Access expires	Max role	Expiration
Kurtz, Prof. Dr. Stefan It's you @Bae4410	Direct member	27 minutes ago by Kurtz, Prof. Dr. Stefan	No expiration set	Maintainer	Expiration date

Invite a Member (cont.)

- to clone the project visit the project page, click on `Clone` and copy the link beginning with `https://`
- follow description of next section
- for more details see `https://www.rrz.uni-hamburg.de/services/datenhaltung/repositories/gitlab/erste-schritte.html`

Checkout a repository: git clone

- you work on the project from anywhere by making a local clone:

```
$ git clone /path/myproject.git myproject
```

- or on a remote computer (e.g. your laptop at home):

```
$ git clone user@host:/path/myproject.git myproject
  Cloning into 'myproject'...
  warning: You appear to have cloned an empty repository.
  done.
```

- this creates a copy of your repository in a local tree named `myproject`
- your local repository consists of three "trees" maintained by git:
  1. your working directory which holds the actual files (if any)
  2. an index which acts as a staging area
  3. the HEAD which points to the last commit you have made
- the last two items are in the `.git` directory, which is basically a copy of the original bare repository

Add files to a project: git add

- copy the files relevant for your project into `myproject` or
- start a new project and create new file there

```
$ ls
  Makefile        dna_sequence.py  command_line_parse.py
$ git status
  On branch master

  Initial commit

  Untracked files:
    (use "git add <file>..." to include in what will be committed)

  Makefile
  dna_sequence.py
  command_line_parse.py
```

Commit files to Git: git commit

- propose changes (add it to the index, see Figure 1, left halve) using:

```
$ git add *.py Makefile
$ git status
  On branch master

  Initial commit

  Changes to be committed:
    (use "git rm --cached <file>..." to unstage)

  new file:   Makefile
  new file:   dna_sequence.py
  new file:   command_line_parse.py
```

- now commit the file to the HEAD:

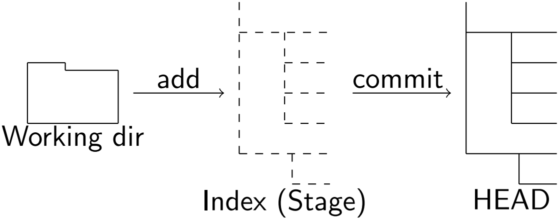
```
$ git commit -m "files for advanced project"
```

Commit files to Git: git commit (cont.)

```
[master (root-commit) 220883b] initial files for advanced project
 4 files changed, 248 insertions(+)
 create mode 100644 Makefile
 create mode 100644 dna_sequence.py
 create mode 100644 command_line_parse.py
```

- this only changes the local copy, not the remote repository

Figure 1: The effects of `git add` and `git commit`.



Push to the bare repository: git push

- as there is no commit in the bare repository yet, one has to execute:

```
$ git push --set-upstream origin master
```

- the options are only required once; later, it suffices to execute:

```
$ git push
```

- so for example, after copying the main file to the repository:

```
$ git status
On branch master
Your branch is up-to-date with ’origin/master’.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

dna_sequence_main.py

nothing added to commit but untracked files present
$ git add dna_sequence_main.py
$ git commit -m "added main file"
```

Push to the bare repository: git push (cont.)

```
$ git status
  On branch master
  Your branch is ahead of 'origin/master' by 1 commit.
    (use "git push" to publish your local commits)
  nothing to commit, working tree clean
$ git push
  Counting objects: 3, done.
  Delta compression using up to 4 threads.
  Compressing objects: 100% (3/3), done.
  Writing objects: 100% (3/3), 558 bytes | 0 bytes/s, done.
  Total 3 (delta 0), reused 0 (delta 0)
  To /Users/stefan/myproject.git/
     9cb5e7a..0da9b1a  master -> master
```

Local update to last commit of bare repository: git pull

- suppose you or a member of your team have worked on another clone of the repository on a different computer
- they had committed the changes and pushed these to the bare repository
- for this example, a python-script `gendist.py` to generate test-data has been added
- to update your local repository to the newest commit, execute:

```
$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /Users/stefan/myproject
   0da9b1a..3a46eab  master     -> origin/master
Updating 0da9b1a..3a46eab
Fast-forward
 gendist.py | 31 +++++++++++++++++++++++++++++++
```

Local update to last commit of bare repository: git pull (cont.)

```
 1 file changed, 31 insertions(+)
 create mode 100755 gendist.py
```

- to view the commits execute:

```
$ git log
commit 3a46eab2f66ba376bfcec93da6c7177a5e1d5758
Author: Joe Smith <joe.smith@uni-hamburg.de>
Date:   Sat May 4 22:47:29 2021 +0200

    Added script to generate testdata

commit 0da9b1a6dd7dbe8f6488f55b5aa332d670d34857
Author: Stefan Kurtz <stefan.kurtz@uni-hamburg.de>
Date:   Sat May 4 22:28:03 2021 +0200

    added main file
...
```

View differences in files: git diff

- to view the changed contents in the last commit execute:

```
$ git diff 0da9b1a6dd7dbe8f6488f55b5aa332d670d34857
diff --git a/gendist.py b/gendist.py
new file mode 100755
index 0000000..2e2441c
--- /dev/null
+++ b/gendist.py
@@ -0,0 +1,31 @@
+#!/usr/bin/env python3
+
+if len(sys.argv) < 3
+  sys.stderr.write(’Usage: {} <numvalues> <list of probabilities>\n’
                    .format(sys.argv[0]))
+  exit(1)
+end
....
```

- suppose you have modified your code to allow for lower or upper case output in the `dna_sequence_show`-function

View differences in files: git diff (cont.)

- this required modifying all C-files and header files
- to view all changes, do:

```
$ git diff
```

- or, if you want to limit the output to a specific file:

```
$ git diff dna_sequence.py
```

```
diff --git a/dna_sequence.py b/dna_sequence.py
index fd39842..6a6fd75 100644
--- a/dna_sequence.py
+++ b/dna_sequence.py
@@ -150,9 +150,9 @@ # the following class represents ...

-def __init__(filename):
+def __init__(filename,upper_case):
-  pretty = "ACGTN"
+  pretty = "ACGTN" if upper_case else "acgtn"
```

Ignore files irrelevant for git: .gitignore

- besides the files edited by a developer, the source tree usually contains many other files such as
  - object code files,
  - executables,
  - files created during the build-process (e.g. those with suffix `.d`),
  - temporary files,
  - PDF files generated by LaTeX.
- they are not part of the source ⇒ not to be committed
- to avoid `git status` listing them, enter corresponding regular expressions into the `.gitignore` file:

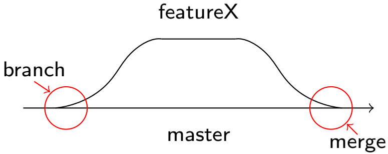
```
*.[oxd]
*.pdf
```

- `.gitignore` is added & committed as part of the source code

Branching of development trees: git branch

- Branches are used to develop features isolated from each other
- Branching means to diverge from the main line of development and continue to do work without disrupting that main line, see Figure 2
- the master branch is the “default” branch when creating a repository
- one may use other branches for development and merge them back into the master branch upon completion

Figure 2: Branching from the master



Branching of development trees: git branch (cont.)

Example

- A student wants to try an improved method for Bwt-based compression, so they switch to a new branch ⇒ bwtimproved
- Student develops the corresponding code in said branch
- Student commits those changes to the branch
- While testing the new method with more data, the student detects an error in the sequence parser
- Student switches back to the master branch, fixes the errors and commits the changes
- Student switches back to the bwtimproved branch, finishes their work, commits it and merges it with master
- all of this can occur without any network access since it takes place in the local repository

Branching of development trees: git branch (cont.)

- the following shows the git command for branching, however, not all steps from the previous example are shown
- create a new branch named “bwtimproved” and switch to it using:

```
$ git checkout -b bwtimproved
  Switched to a new branch "bwtimproved"
```

- after working on it and completing the tests you want to merge it into the master:

```
$ git checkout master
$ git merge bwtimproved
  Merge made by the ’recursive’ strategy.
  bwt.py |     10 ++++++++++
  1 file changed, 1 insertion(+)
```

- as the modified code is now in the master-branch, you can delete developer branch:

```
$ git branch -d bwtimproved
```

Replace local changes: git checkout

- sometimes one recognizes that the idea one is working on was misleading
- all changes to the current file, say `bwt.py` need to be reverted to the previously committed version

⇒ turn back the clock and execute:

```
$ git checkout bwt.py
```

- replaces local changes to the file `bwt.py`, i.e. it restores the previous committed version
- changes already added to the index, as well as new files, will be kept

Search the files in the current subtree: git grep

- you may frequently find yourself in a situation where you want to search all files in the current repository for a regular expression
- e.g. you want to look for all occurrences of the string DNAsequence:

```
$ git grep DNAsequence
dna_sequence.py:class DNAsequence:
dna_sequence_main.py:  sequence = DNAsequence(sys.argv[1])
```

- search is done on all files of the entire subtree which have been added to the index
- git grep has inherited all options of the Unix-tool, grep
- e.g. if you are only interested in the names of the files (which you want to edit), execute:

```
$ git grep -l DNAsequence
dna_sequence.py
command_line_parse.py
dna_sequence_main.py
```

Configuring ssh for git

- remote access to the computers at the ZBH is via port 7373
- as git remote access is tunneled by ssh, the best way is to configure ssh accordingly
- on the external computer (e.g. your laptop) create the file `.ssh` in your home directory
- add the following lines:

```
Host bari
User username
Port 7373
Hostname bari.zbh.uni-hamburg.de
```

- this introduces an abbreviation `bari` which allows the user with name `username` to access the given host
- you can then clone your git repo via ssh with:

```
$ git clone bari:/path/myproject.git
```

Useful hints

- there are built-in GUIs for git, such as `gitk` for Linux
- to see the list of all git commands, each with one-line descriptions, execute:

```
$ git --help
```

- each git command has an own manual which can be displayed with:

```
$ git <command> --help
```

- use colorful git output:

```
$ git config color.ui true
```

- type the commit message in your favorite editor:

```
$ git commit -a
```

- rename files which have been added to the index before:

```
$ git mv oldname.py newname.py
```

- remove files from the index, or from the working tree and the index:

```
$ git rm deprecated.py
```

- the effect of `git mv` and `git rm` with respect to the index become effective with the next commit