Git into Version Control: an Introduction to Git for Scientists

Lib4RI training series · 3 November 2026

How to use this handout

Keep this handout open while following the course: every command shown on screen is here, in the same order, so you can type along without copying from the projector.

- Lines in grey boxes are commands to type in the terminal. Press Enter after each line.
- Text after `#` is a comment for you; you don't need to type it.
- **Observe** notes tell you what to look at after a command. Reading the output is half the learning.
- Most parts start from a ready-made checkpoint repository; each part says where to start, and switching is always `cd ../` plus the checkpoint name, for example `cd ../cp2`. If you get lost, don't panic: the next checkpoint lets you rejoin the group.

The course folder you unzipped before the course looks like this:

```
git-course/             # unzip it wherever you like
├── starter/            # plain files you will copy in during Part 3
├── temperature-study/  # your project, created in Part 3
├── cp1/  cp2/  cp3/    # ready-made checkpoint repositories,
└── cp3b/  cp4/         # right next to your project
```

The running example is a small research project: a script that analyses temperature measurements, a data file and a paper draft. You never need to run the script. It is just text that changes over time.

Reference for everything in this course: the free book Pro Git by Scott Chacon and Ben Straub.

Part 1 — Command-line survival kit

The terminal is a text conversation with your computer: you type a command, it answers, and it always works *in* one folder at a time. On Windows, use **Git Bash**; on macOS and Linux, use **Terminal**.

Command	What it does	Example
`pwd`	Print the folder you are in ("print working directory")	`pwd`
`ls`	List the files in the current folder	`ls`
`ls -a`	List all files, including hidden ones (names starting with `.`)	`ls -a`
`ls -l`	List files with details: size, date, permissions. Many systems also accept the shortcut `ll` (not macOS).	`ls -l`
`cd <folder>`	Change into a folder	`cd git-course`
`cd ..`	Go up one folder	`cd ..`
`cd ~`	Go to your home folder	`cd ~`
`mkdir <name>`	Create a new folder	`mkdir test`
`cp <from>`	Copy a file	`cp`

`<to>`	`../starter/paper.md .`

Paths. A path says where a file is. `data/temps.csv` is *relative*: it starts from the current folder. `..` means the parent folder and `.` means the current folder, so `cp ../starter/paper.md .` means "copy `paper.md` from the `starter` folder next to me into here".

Two habits that save time:

- **Tab completion.** Type the first letters of a file or folder name and press Tab; the terminal completes it. If nothing happens, press Tab twice to see the options.
- **History.** Press the Up arrow to recall previous commands, then edit and press Enter.

Editing a file with nano. `nano README.md` opens the file (or creates it). Type normally. Save with Ctrl+O then Enter, exit with Ctrl+X. The shortcuts are listed at the bottom of the screen (`^` means Ctrl). On recent macOS versions, typing nano opens a similar editor called pico, with the same shortcuts.

Try it now:

```
pwd
ls
cd git-course    # the course folder: adjust the path to where you unzipped it
ls
```

More: Pro Git, section 1.4 "The Command Line".

Part 2 — Configuring Git

Git keeps its settings in a configuration file. You set four of them before the course and add a fifth now, all with `git config --global`, where `--global` means "for all my repositories":

Setting	Why it matters
`user.name`, `user.email`	Stamped into every commit you make: this is how history knows who did what.
`core.editor`	The editor Git opens when it needs a message from you. nano is the friendliest choice.
`init.defaultBranch`	The name of the first branch in a new repository: `main`.
`core.autocrlf`	How line endings are handled. `input` keeps files as they are, which avoids confusing warnings on Windows.

Add the fifth setting now (everyone, on any system):

```
git config --global core.autocrlf input
```

Check that your configuration is in place:

```
git --version
git config --list
```

Observe: the list should contain these five lines, with your own values:

```
user.name=Your Name
user.email=your.name@institute.ch
core.editor=nano
init.defaultbranch=main
core.autocrlf=input
```

If one is missing, set it now (example for the name):

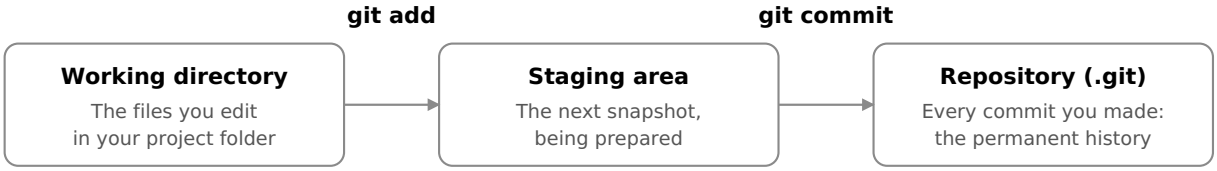
```
git config --global user.name "Your Name"
```

Help is always available: `git help <command>` opens the full manual, `git <command> -h` prints a short summary.

More: Pro Git, sections 1.6 "First-Time Git Setup" and 1.7 "Getting Help".

Part 3 — Your first repository

Git keeps your work in three areas. You edit files in the **working directory**, choose what goes into the next snapshot in the **staging area**, and record that snapshot permanently in the **repository** with a commit.



The three areas of Git · git add and git commit

`git status` always tells you which area each changed file is in.

Step 0 — Create the repository

```
pwd                 # make sure you are in the course folder git-course
mkdir temperature-study
cd temperature-study
git init
ls -a
git status
```

Observe: `ls -a` shows a hidden `.git` folder. That folder *is* the repository: the whole history lives there.

Step 1 — Your first commit

```
nano README.md
```

Type these two lines, then save (Ctrl+O, Enter) and exit (Ctrl+X):

```
# Temperature study
Analysis of temperature measurements from a sensor network.
```

```
git status            # README.md is "untracked"
git add README.md
git status            # README.md is "to be committed" (staged)
git commit -m "Add README"
git status            # "nothing to commit, working tree clean"
git log
```

Observe: the commit prints a line like `[main (root-commit) 049619b] Add README`: the branch, `root-commit` (the very first commit, so it has no parent), the short hash (yours will be different) and your message. Below it, `create mode 100644 README.md` means a new file was added; `100644` is Git's code for an ordinary file (an executable script would be `100755`).

Observe: run `git status` after every step and watch the file move from working directory to staging area to repository.

Now change the README once more and ask Git what changed:

```
nano README.md        # add a line, e.g. "Data: temperature sensors S1 and S2"
git status            # README.md is "modified"
git diff              # exactly what changed since the last commit
git add README.md
git commit -m "Describe the data"
```

Observe: `git diff` shows the changes you haven't staged yet: added lines start with `+`, removed lines with `-`. Brand-new files that Git doesn't track yet don't appear in it.

Step 2 — Add the analysis script

```
cp ../starter/analysis.py .
git status
git add analysis.py
git commit -m "Add analysis script"
```

Step 3 — Data, outputs and .gitignore

```
mkdir data results
cp ../starter/temps.csv data/
cp ../starter/summary.txt results/
cp ../starter/notes.tmp .
git status
git status -u         # also lists the files inside untracked folders
```

Observe: Git offers everything, including the `results/` folder (with an output file inside) and a temporary file, neither of which should be versioned. For untracked folders, `git status` shows only the folder name; `git status -u` lists the files inside. Tell Git to ignore them:

```
nano .gitignore
```

Type these two lines, save and exit:

```
results/
*.tmp
```

```
git status            # only data/ and .gitignore are left
git add data .gitignore
git commit -m "Add measurement data and .gitignore"
```

Good to know: Git tracks files, not folders, so an empty folder never appears in `git status`.

Step 4 — Add the paper draft

```
cp ../starter/paper.md .
git add paper.md
git commit -m "Add paper draft"
```

Step 5 — Why the staging area exists

Make two unrelated changes with nano:

1. In `analysis.py`, add this line at the end: `print(f"Number of measurements: {len(temps)}")`
2. In `paper.md`, change "Temperatures above 20 °C were frequent." to "Temperatures often exceeded 20 °C."

```
git status            # both files modified
git diff              # shows exactly what changed
git add analysis.py
git commit -m "Report number of measurements"
git add paper.md
git commit -m "Clarify abstract"
git log --oneline
```

Observe: two logical changes became two separate commits. The staging area is how you compose each snapshot. You should now have 7 commits.

To rename or delete a tracked file, use `git mv old new` and `git rm file` so that Git records the change.

More: Pro Git, sections 2.1 "Getting a Git Repository" and 2.2 "Recording Changes to the Repository".

A peek inside .git

A commit is not a list of changes: it is a full snapshot, stored as three kinds of objects. Watch the demo; you can try the commands yourself afterwards.

```
git cat-file -p HEAD            # the commit: tree, parent, author, message
git cat-file -p HEAD^{tree}     # its tree: file names and their blob hashes
git cat-file -p HEAD~1^{tree}   # the tree of the commit before
cat .git/HEAD                   # "ref: refs/heads/main"
cat .git/refs/heads/main        # just a commit hash
```

Observe: compare the two trees. Only the file you changed in the last commit has a new hash; all the others point to exactly the same blobs. Git stores full snapshots, but unchanged files are never copied.

Object	What it stores
Blob	The content of one file
Tree	A folder: names pointing to blobs and other trees

Commit	One tree (the snapshot), the parent commit, author, date and message

Every object is named by the hash of its content: change one byte and the name changes, which is why a commit can never be modified, only replaced. The key takeaway: **a branch is a tiny file containing a commit hash**, and `HEAD` says which branch you are on. Keep this in mind for undoing and branching later.

More: Pro Git, sections 10.2 "Git Objects" and 10.3 "Git References".

Part 4 — Exploring history

Your colleague Sam worked on the project for a few more days and handed you the whole folder. The history travels inside `.git`, so you get every change Sam made, with the reasons. Everyone switches to this checkpoint:

```
cd ../cp1
git log
git log --oneline
```

Naming a commit. The first 7 characters of its hash are enough (`git show 40c74c1`). `HEAD` is the commit you are on, `HEAD~1` the one before it, `HEAD~2` two before, and so on.

Command	What it shows
`git show <commit>`	One commit: message and changes
`git log <file>`	Only the commits that changed this file
`git log -p <file>`	Those commits with their changes
`git show <commit>:<file>`	The file as it was in that commit
`git diff`	Changes not yet staged
`git diff --staged`	Changes staged for the next commit
`git diff <commit> <commit>`	Differences between two commits

Detective questions

1. **Why is the threshold 25 °C and not 20 °C?** Hint: `git log --oneline analysis.py`, then `git show` the right commit. Read the whole message.
2. **The paper once reported a mean of −105.55 °C. Which commit introduced it, and which commit fixed the cause?** Hint: `git log -p paper.md` (the output opens in a viewer: scroll with the arrow keys or Space, press b to go back a page, /word to search, q to quit).
3. **What did the abstract look like in the very first draft?** Hint: `git show HEAD~7:paper.md`
4. **The three diffs.** Try this sequence and compare the outputs:

```
nano README.md          # add any line, save and exit
git diff                # shows your change
git add README.md
git diff                # empty! the change is staged now
git diff --staged       # here it is
```

Finished early? Bonus questions:

5. What changed in the paper between the first draft and now? Hint: `git diff <first-draft-commit> HEAD -- paper.md`
6. Which commits changed a line containing the word THRESHOLD? Hint: `git log -G THRESHOLD --`

`oneline`

7. The abstract still says 20 °C, the script uses 25 °C. Since when do they disagree?

More: Pro Git, section 2.3 "Viewing the Commit History".

Part 5 — Undoing things

A few days later, the project is in a mess: the abstract fix is incomplete, an older change broke the results, and there are leftovers in your working directory. Everyone switches to this checkpoint:

```
cd ../cp2
git status
git log --oneline -3
```

Observe: one file is staged by accident (`scratch.py`), one file has an unwanted change (`analysis.py`). We clean up the present first, then fix the past.

Step 1 — Clean up the present

```
git diff                          # THRESHOLD = 2.5? a leftover experiment
git restore --staged scratch.py # unstage it (the file stays on disk)
git restore analysis.py         # discard the change
git status
```

Careful: `git restore <file>` throws your changes away for good. There is no commit to fall back on.

Step 2 — Fix the last commit

The last commit fixed the abstract but forgot the Methods section, and its message has a typo.

```
git log --oneline -1            # note the hash
nano paper.md                   # Methods: "threshold of 20 °C" → "25 °C"
git add paper.md
git commit --amend -m "Fix threshold in abstract and methods"
git log --oneline -1            # a different hash!
```

Observe: commits never change. `--amend` creates a new commit and moves the branch to it. Where did the old one go? Keep the question in mind until step 4.

Who made this commit? Every commit records two people: the **author**, who wrote the change, and the **committer**, who created this exact commit. `--amend` keeps the original author, so `git log` still shows Sam, but you are now the committer. To see both, with their dates: `git log --format=fuller`

Step 3 — Undo an older commit safely

A commit from a few days ago rounded the mean to whole degrees, contradicting the paper.

```
git log --oneline               # find "Round mean to whole degrees"
git revert <hash>               # nano opens with a message: save and exit
git log --oneline -3
git show HEAD
```

Observe: the mistake is still in the history, followed by a new commit that undoes it. `revert` adds history instead of erasing it, which makes it the safe choice for anything older than the last commit.

Step 4 — The accident and the rescue

`git reset` moves the branch pointer to another commit. What happens to your files depends on the mode:

Mode	Branch pointer	Staging area	Working directory
`--soft`	moved	kept	kept
`--mixed` (default)	moved	reset	kept
`--hard`	moved	reset	reset

Now make a deliberate accident:

```
git reset --hard HEAD~3
git log --oneline              # three commits gone!
git reflog                     # everywhere HEAD has been
git reset --hard HEAD@{1}      # back to where we were before the reset
git log --oneline              # everything is back
```

Observe: the reflog also lists the commit from before your amend in step 2. Almost nothing in Git is ever really lost.

> **If you committed a secret** (a password, a token, a key)
>
> 1. **Change or revoke the secret first.** Removing it from Git cannot reach copies that already exist elsewhere: a clone, a zip, a backup.
> 2. **Only in the last commit, nothing shared yet:** remove it from the file, then `git add` and `git commit --amend`. The old commit still sits in the reflog, so delete it for good with `git reflog expire --expire=now --all` and then `git gc --prune=now`.
> 3. **Deeper in the history:** use the open-source tool git filter-repo (`pip install git-filter-repo`), on a fresh copy of the repository. Write `my-secret==>***REMOVED***` into a file `replacements.txt` and run `git filter-repo --replace-text replacements.txt`. All commits from the leak onwards get new hashes.
> 4. **Check:** `git log --all --oneline -S 'my-secret'` must print nothing.
> 5. **Prevention:** keep secrets outside the project folder, in environment variables or in a file listed in `.gitignore`.

More: Pro Git, sections 2.4 "Undoing Things" and 7.7 "Reset Demystified".

Part 6 — Branching and merging

A branch is a movable pointer to a commit; `HEAD` tells Git which branch you are on. Continue in your cp2 folder from Part 5. If you got lost there, start from the checkpoint instead: `cd ../cp3`.

Exercise A — A branch is just a pointer

Task: add usage instructions to the README on a separate branch.

```
git branch                          # only main, marked with *
git branch readme-usage
git log --oneline --graph --all     # two names on the same commit
cat .git/refs/heads/readme-usage    # the same hash as main
git switch readme-usage
nano README.md                      # add the line: Run with: python analysis.py
git commit -am "Add usage instructions"
git log --oneline --graph --all
git switch main
cat README.md                       # the new line is gone!
git merge readme-usage              # "Fast-forward"
git branch -d readme-usage
```

Observe: switching branches changes the files on disk. "Fast-forward" means there was nothing to combine: Git just moved the `main` pointer forward.

New shortcut: `git commit -am "..."` stages all changes to tracked files and commits in one step. It never adds new, untracked files.

Exercise B — Two lines of work, one file

Task: a colleague suggests the median as a more robust estimate. Try it on a branch, and meanwhile continue the paper on `main`.

```
git switch -c median-method       # create the branch and switch to it
nano analysis.py
```

Add `import statistics` as the second line, and these two lines at the end:

```
median = statistics.median(temps)
print(f"Median temperature: {median:.2f} °C")
```

```
nano paper.md                     # Methods: "We computed the mean" → "We computed the mean and median"
git commit -am "Add median as robust estimate"
git switch main
nano paper.md                     # add at the very end (see below)
git commit -am "Start discussion section"
git log --oneline --graph --all   # the history has diverged
git merge median-method           # nano opens with a merge message: save and exit
git log --oneline --graph --all
```

Text to add at the end of `paper.md` on `main`:

```

## Discussion
The two sensors show a consistent daily cycle.
```

Observe: both branches changed `paper.md`, but in different places, so Git combined them automatically into a merge commit.

The conflict

Two co-authors, Anna and Marc, each improved the same sentence of the abstract on their own branch. Everyone switches to this checkpoint:

```
cd ../cp3b
git log --oneline --graph --all
```

Before you start: if anything goes wrong during a merge, `git merge --abort` takes you back to before it.

```
git merge coauthor-anna           # fast-forward
git merge coauthor-marc           # CONFLICT
git status                        # "both modified: paper.md"
nano paper.md
```

In the file, Git marks both versions:

```
<<<<<<< HEAD
Temperatures exceeded 25 °C in 3 of 10 measurements.
=======
Afternoon temperatures regularly exceeded 25 °C.
>>>>>>> coauthor-marc
```

The top part is what `main` has, the bottom part what Marc's branch has. Decide what the sentence should say (one version, the other, or a combination), write it, and delete the three marker lines. Then:

```
git add paper.md                  # "I resolved it"
git commit                        # nano opens with a merge message: save and exit
git log --oneline --graph --all
```

Observe: Marc's keywords line was merged without any conflict. Git combines everything it can and asks you only about the lines both sides changed.

More: Pro Git, sections 3.1 "Branches in a Nutshell" and 3.2 "Basic Branching and Merging".

Part 7 — Advanced branching

You are halfway through adding a figure to the paper, on the branch `new-figure`. Everyone switches to this checkpoint:

```
cd ../cp4
git status                        # on new-figure, two files modified
git log --oneline --graph --all
```

Stash — park unfinished work

Your supervisor wants the draft today, but the author list is missing. Your figure work isn't ready to commit.

```
git switch main                    # refused: your changes would be overwritten
git stash                          # park the changes
git status                         # clean
git stash list
git switch main
nano paper.md                      # add an author line under the title
git commit -am "Add author list"
git switch new-figure
git stash pop                      # your unfinished work is back
git status
git commit -am "Finish figure"
```

Rebase — replay commits on a new base (demo)

Watch the demo first; you can try it yourself afterwards.

```
git log --oneline --graph --all   # new-figure started before "Add author list"
git rebase main
git log --oneline --graph --all   # now a straight line, and all hashes changed
```

Observe: rebase takes your commits and replays them on top of `main`. The changes are the same, but they are new commits.

The branch history is messy: "wip", "fix", "fix again", "typo". Tidy it up before integrating:

```
git rebase -i main
```

nano opens a list of your commits. Keep `pick` on the first line and change `pick` to `squash` on all others, then save and exit. A second editor opens with all the messages: replace them with one line, `Add temperature figure`, save and exit.

```
git log --oneline --graph --all   # one clean commit on top of main
git switch main
git merge new-figure              # fast-forward
git branch -d new-figure
```

> **Golden rule: never rebase commits that someone else already has.**
>
> Rebasing replaces commits with new ones (new hashes). On your own machine that's harmless, but if others already have the old commits, their history and yours no longer match.

Tags — mark a version forever

The paper is submitted today. The tag must go on main, so make sure you are there first.

```
git switch main
git merge new-figure              # only if you skipped the rebase above: save and exit nano
git tag -a submitted-v1 -m "Version submitted to the journal"
git tag
git show submitted-v1
nano paper.md                     # a reviewer asks you to reword a sentence
git commit -am "Address reviewer comment"
git diff submitted-v1             # what changed since submission?
```

Tagged the wrong commit? Delete the tag with `git tag -d submitted-v1` and create it again on the right one.

Observe: the tag will always point to the exact state you submitted, however much the project changes afterwards.

More: Pro Git, sections 7.3 "Stashing and Cleaning", 3.6 "Rebasing", 7.6 "Rewriting History" and 2.6 "Tagging".

Part 8 — Where to go next

A first look at sharing

```
cd ..                             # back to the course folder
git clone cp4 my-copy
cd my-copy
git log --oneline                 # the complete history came along
git remote -v                     # the copy knows where it came from: "origin"
```

This connection between repositories is the basis of working with others: remotes, `push` and `pull`. It is the natural next step after this course: Pro Git, section 2.5 "Working with Remotes".

Command line or GUI?

Everything you did today also works in a graphical interface, and you can switch between the two at any time: they work on the same repository. Git for Windows includes `git gui` (good for staging and committing) and `gitk` (good for viewing history and branches). Many others exist: Pro Git, section A1.1 "Graphical Interfaces".

Large data

Git is made for text files. For large datasets or binary files, look at tools built on top of Git, such as Git LFS, git-annex or DVC.

Reading path in Pro Git

- Chapter 2 "Git Basics" and chapter 3 "Git Branching": consolidate everything from today.
- Chapter 7 "Git Tools": 7.3 "Stashing and Cleaning", 7.6 "Rewriting History", 7.7 "Reset Demystified".
- Chapter 10 "Git Internals", for the curious.
- Section 2.5 "Working with Remotes" and chapter 5 "Distributed Git": remotes and collaboration.

Also worth looking up: `git blame`, who last changed each line (7.10 "Debugging with Git"), and `git cherry-pick`, which copies a single commit to another branch (5.3 "Maintaining a Project").

Cheat sheet

Command	What it does	Pro Git
Basics		
`git init`	Turn the current folder into a repository	2.1
`git status`	Show which files are modified, staged or untracked	2.2
`git add <file>`	Put a file's current content into the staging area	2.2
`git commit -m "msg"`	Record the staged snapshot with a message	2.2
`git commit -am "msg"`	Stage all changes to tracked files and commit	2.2
`git mv <old> <new>` / `git rm <file>`	Rename / delete a tracked file	2.2
History		
`git log --oneline --graph --all`	Compact history of all branches	2.3
`git log -p <file>`	Commits that changed a file, with their changes	2.3
`git log --format=fuller`	Show author and committer, with dates	2.3
`git show <commit>`	One commit: message and changes	7.1
`git show <commit>:<file>`	A file as it was in that commit	7.1
`git diff` / `git diff --staged`	Unstaged / staged changes	2.2
`git diff <a> <b>`	Differences between two commits, branches or tags	2.2
Undoing		
`git restore <file>`	Discard changes in the working directory (permanent!)	2.4
`git restore --staged <file>`	Unstage a file, keeping its changes	2.4
`git commit --amend`	Replace the last commit	2.4
`git revert <commit>`	Add a new commit that undoes an older one	7.8
`git reset --hard <commit>`	Move the branch and reset all files (careful!)	7.7
`git reflog`	Everywhere HEAD has been: your safety net	7.1
Branches		
`git branch`	List branches	3.3
`git switch <branch>` / `git switch -c <branch>`	Switch to / create and switch to a branch	3.2
`git merge <branch>`	Merge a branch into the current one	3.2
`git merge --abort`	Cancel a merge with conflicts	3.2
`git branch -d <branch>`	Delete a merged branch	3.3
Advanced		
`git stash` / `git stash pop`	Park unfinished changes / bring them back	7.3
`git rebase <branch>`	Replay your commits on top of another branch	3.6
`git rebase -i <branch>`	Edit, squash or reorder your commits	7.6
`git tag -a <name> -m "msg"`	Mark the current commit permanently	2.6
Other		
`git cat-file -p <hash>`	Show the content of a Git object	10.2
`git clone <path> <new-folder>`	Make a complete copy of a repository	2.1

`git help <command>`	Open the manual for a command	1.7

`.gitignore`: one pattern per line, for example `results/` (a folder) or `*.tmp` (all files ending in .tmp).

Good commit messages: a short summary line in the imperative ("Add", "Fix", "Clarify") that completes the sentence "If applied, this commit will...".

Tip: aliases turn long commands into short ones. After `git config --global alias.graph "log --oneline --graph --all"`, typing `git graph` does the same as `git log --oneline --graph --all`. More: Pro Git, section 2.7 "Git Aliases".

Resources

Pro Git by Scott Chacon and Ben Straub, the reference for everything in this course, free online:
git-scm.com/book

- **Git reference manual**, every command and option: git-scm.com/docs
- **Official Git cheat sheet**: git-scm.com/cheat-sheet
- **Graphical clients** for Git: git-scm.com/downloads/guis

A complementary course

K295 – Branch into Git: Collaborate Effectively with Version Control. Another introductory course, from a different perspective: it starts from GitHub and working together with Git. Part of the Lead Campus course offer (login.lead-campus.ch).
Course page and details

Questions after the course

- Federico Cantini: federico.cantini@lib4ri.ch
- Frank Hösli: frank.hoesli@lib4ri.ch