Gitinfo Lua package*

Erik Nijenhuis

10th November 2023

This file is maintained by **Xerdi**.
Bug reports can be opened at
`https://github.com/Xerdi/gitinfo-lua`.

Abstract

This project aims to display git project information in PDF documents. It's mostly written in Lua for executing the `git` commands, therefore making this package only applicable for `lualatex` with `shell escape` enabled. If `lualatex` isn't working for you, you could try gitinfo2 instead. For LaTeX it provides a set of standard macros for displaying basic information or setting the project directory, and a set of advanced macros for formatting commits and tags.

Contents

Index

*This document corresponds to package `gitinfo-lua` version 1.0.0 written on 2023-11-10.

1 Usage

For the package to work one should work, and only work, with LuaTeX. Another prerequisite is that there is an available git repository either in the working directory, or somewhere else on your machine (see section 2.2).

1.1 Git

For this package to work at a minimum, there has to be an initialized Git repository, and preferably, at least with one commit. For example, the following minimal example should do the trick already:

```
mkdir my_project
cd my_project
echo "# My Project" > README.md
git init && git commit -am "Init"
```

Listing 1: Minimal Git setup

Then in order for the changelog to work, the project needs to contain either 'lightweight-' or 'annotated' tags. The main difference is that a lightweight tag takes no extra options, for example: `git tag 0.1`. See listing 8 for more examples on authoring and versioning with `git`.

1.2 LuaLaTeX

For generating the document with LaTeX one must make use of `lualatex`. For example, when having the main file '`main.tex`':

```
# Generate once
lualatex -shell-escape main
# Generate and keep watching with LaTeXMK
latexmk -pvc -lualatex -shell-escape main
```

Listing 2: Generating the document with LaTeX

Note that in both cases option `-shell-escape` is required. This is required for issuing `git` via the commandline.

2 LaTeX Interface

2.1 Package Options

`\usepackage[`⟨*opts...*⟩`]{gitinfo-lua}` This package provides some options for default formatting purposes. The author sorting is one of them. If the options contain ⟨*contrib*⟩ the authors will be sorted based on their contributions, otherwise the authors will be sorted alphabetically, which is the default option ⟨*alpha*⟩. Another option is the ⟨*titlepage*⟩ option, which sets the `\author` and `\date` macros

accordingly. By default, it sets the local git author, equivalent to option ⟨*author*⟩. Pass option ⟨*authors*⟩ to set all git authors of the project instead.

2.2 Basic macros

By default the main file's directory is used as git project directory. This directory
\gitdirectory can be manipulated with `\gitdirectory{`⟨*path*⟩`}`. This is only tested with rela-
\gitunsetdirectory tive paths on Linux. To undo this operation and switch back to the main file's directory use `\gitunsetdirectory`.

\gitversion The current version can be display by using `\gitversion` and is equivalent to `git describe --tags --always`, working for both lightweight and annotated tags. For this project `\gitversion` results in 1.0.0. When the version is dirty it will be post fixed with `-<commit count>-<short ref>`. For example, when this paragraph was written, the version was displaying 0.0.1-14-gcc2bc30.

\gitdate The `\gitdate` macro gets the most recent date from the git log. Meaning, the last 'short date' variant is picked from the last commit. This short date is formatted ISO based and is already suitable for use in packages like `isodate` for more advanced date formatting.

The author's name and email can be accessed using `\gitauthor` and `\`
\gitauthor `gitemail`. These values are based on `git config user.name` and `git config`
\gitemail `user.email`.

2.3 Multiple Authors

\dogitauthors When projects having multiple authors this package can help with the
\forgitauthors `\dogitauthors[`⟨*conj*⟩`]` and `\forgitauthors[`⟨*conj*⟩`]{`⟨*csname*⟩`}` macro. Where `\dogitauthors` executes a default formatting implementation of `\git@format@author` and `\forgitauthors` executes the given ⟨*csname*⟩ for every author available. The optional ⟨*conj*⟩ conjunction makes it possible to even integrate it further. For example, when setting the authors in pdfx, the conjunction would be `[\\sep~]`, so that the authors are properly separated in the document properties[1].

Listing 3: Formatting authors

```
1 \newcommand{\myauthorformat}[2]{#1 \
    href{mailto:#2}{#2}}
2 \forgitauthors[\\]{myauthorformat}
3 % Or using standard format
4 \dogitauthors[\\]
```

Results in

Alice ⟨alice@example.com⟩
Bob ⟨bob@example.com⟩

[1] See package documentation of `pdfx`: `https://ctan.org/pkg/pdfx`

This example is generated with the history of the `git-test-project` (see section 3) and is alphabetically sorted with package option `alpha`.

2.4 Commits

For this section the git project of this document is used due to the fact that there are references to revisions. The test project's revisions change for every user, since they get recreated every time `test-scenario.sh` is executed (see section 3).

`\gitcommit` `[`⟨*format*⟩`]{`⟨*csname*⟩`}{`⟨*revision*⟩`}`
For displaying commit data `\gitcommit` can be used. The optional `format` takes variables separated by a comma. The default `format` is `h,an,ae,as,s,b`. The `csname` is a user defined command accepting every variable as argument.

Listing 4: Formatting a commit

```
1 \newcommand{\formatcommit}[3]{#1, by #2 on
      \printdate{#3}}
2
3 \gitcommit[s,an,as]{formatcommit}{75dc036}
```

Results in

Add value escaping, by Erik Nijenhuis on 23rd October 2023[2]

Consult `man git-log` for possible format variables and omit the % for every variable.

`\forgitcommit` `[`⟨*format*⟩`]{`⟨*csname*⟩`}{`⟨*rev_spec*⟩`}`
For displaying multiple commits the `\forgitcommit` is used, which has the same arguments as `\gitcommit`, but only this time the `csname` is executed for every commit. The last argument `rev_spec` this time, however, can have no argument or a sequence.

Listing 5: Formatting commits

```
1 \newcommand{\formatcommits}[2]{\item
      #1\\\quad -#2}
2
3 \begin{itemize}
4     \forgitcommit[s,an]{formatcommits
          }{75dc036...e51c481}
5 \end{itemize}
```

Results in

- Add value escaping —Erik Nijenhuis
- Reimplement for_commit —Erik Nijenhuis

[2] `\printdate` from `isodate`: `https://www.ctan.org/pkg/isodate`

2.5 Tags

In this section the `git-test-project` is used.

`\forgittag` The tags are mostly useful for generating changelogs. For formatting tags, there's a `\forgittag[`⟨*format*⟩`]{`⟨*csname*⟩`}`. Again, like `\forgitcommit` it takes a format, however, this time more complex, since the formatting options differ between `git log` and `git for-each-ref`. For more info regarding these formatting options consult the man page of `git-for-each-ref`.

Listing 6: Formatting tags

```
1 \newcommand{\formattags}[2]{\item Version
    #1\\type: #2}
2
3 \begin{itemize}
4     \forgittag[refname:short,objecttype]{
          formattags}
5 \end{itemize}
```

Results in

- Version 0.0.1
  type: commit
- Version 0.1.0
  type: tag

This example shows that the versions used are mixed. This is, of course, a horrible way to manage a project's version, though, we'll continue on with this hard objective. For example, if we wish to display the author of the lightweight and annotated tag, we can do so by specifying a format using the if-then-else feature of the format specification. The format would be: `(taggername)(taggername)(authorname)`. Here the `taggername` will show up, or if not present, the `authorname` will be shown instead.

The default format specification is like the `\forgitcommit` format, but then again, some bit more complex:
`refname:short,(taggername)(taggername,taggeremail,taggerdate:short)(authorname,authoremail,authordate:short),subject,body`. This is a robust example of getting all information, being it a lightweight- or annotated tag.

`\forgittagseq` For displaying commits in between tags, there's a `\forgittagseq{`⟨*csname*⟩`}`. The ⟨*csname*⟩ takes exactly three arguments, namely, the ⟨*current*⟩, ⟨*next tag*⟩ and ⟨*rev spec*⟩. The last iteration gives an empty value for ⟨*next tag*⟩ and the ⟨*rev spec*⟩ is identical to ⟨*current*⟩.

`\gittag` Afterward tag info get be fetched using the `\gittag[`⟨*format*⟩`]{`⟨*csname*⟩`}{`⟨*tag*⟩`}`. This macro takes the same formatting specification as `\fotgittag`. Beware of using `\gittag` for the ⟨*next tag*⟩ parameter in `\forgittagseq`.

All these macros put together are demonstrated in listing 7 (see next page).

2.6 Changelog

This example demonstrates the generation of a changelog. For simplicity's sake, every tag is displayed in a `description` environment's item and within an `enumerate` environment displaying commits in between.

Listing 7: Formatting a changelog

```
1 \section*{Changelog}
2 \newcommand{\commitline}[1]{\item #1}
3 \newcommand{\formatversion}[3]{%
4     \item[#1]
5     \gittag[(taggerdate)(taggerdate:short)(authordate:short
          )]{printdate}{#1}
6     \begin{itemize}
7         \forgitcommit[s]{commitline}{#3}
8     \end{itemize}
9 }%
10 \begin{description}
11     \forgittagseq{formatversion}
12 \end{description}
```

Results in

Changelog

0.1.0 6th August 2017

- Add gitignore

0.0.1 5th August 2017

- Add intro (README.md)
- Add readme

For displaying the tagline (see line 5) we use the existing `\printdate` macro of package `isodate`, which also takes exactly one argument For every version sequence the commits in between are displayed (see line 7), where the last sequence having the initial commit as second argument plays well with the `\forgitcommit` macro and makes it possible to show the whole sequence of history.

3 Project Example

This documentation uses an example `project` which gets created by the `git-scenario.sh` script (see listing 8). It creates some commits having dates in the past and different authors set. Lastly it creates a 'lightweight-' and 'annotated' tag.

To set up this scenario either do `make scenario` or `bash scenario.sh`.

Listing 8: git-scenario.sh

```
#!/bin/bash

set -e

PROJECT_DIR="${1:-../../git-test-project}"

set_author() {
  git config user.name $1
  git config user.email $2
  git config committer.name $1
  git config committer.email $2
  git config author.name $1
  git config author.email $2
}

alice() {
  set_author 'Alice' 'alice@example.com'
}
bob() {
  set_author 'Bob' 'bob@example.com'
}
charlie() {
  set_author 'Charlie' 'charlie@example.com'
}

if [[ -d "${PROJECT_DIR}" ]]; then
  rm -rf "${PROJECT_DIR}"
fi

mkdir "${PROJECT_DIR}"
cd "${PROJECT_DIR}"

git init

alice

echo "# My project" > README.md
```

```
git add README.md
git commit -m "Add readme" --date="2017-08-04 10:32"

bob

echo "
Another project by Alice and Bob." >> README.md
git add README.md
git commit -m "Add intro (README.md)" --date="2017-08-05
    06:12"

alice

GIT_COMMITTER_DATE="2017-08-05 07:11" git tag 0.0.1

bob

curl https://raw.githubusercontent.com/github/gitignore/
    main/TeX.gitignore > .gitignore
git add .gitignore
git commit -m "Add gitignore" --date="2017-08-06 12:03"

charlie

export GIT_COMMITTER_DATE="2017-08-06 08:41"
git tag -a 0.1.0 -m "Version 0.1.0"
```