Sources of this document are available at:
https://github.com/Slookeur/OPEN

From source code to packaging: open source software distribution review and tips

Sébastien LE ROUX sebastien.leroux@ipcms.unistra.fr

INSTITUT DE PHYSIQUE ET DE CHIMIE DES MATÉRIAUX DE STRASBOURG,
DÉPARTEMENT DES MATÉRIAUX ORGANIQUES,
23 RUE DU LOESS, BP43,
F-67034 STRASBOURG CEDEX 2, FRANCE

OCTOBER 7, 2024

Acknowledgments

You are about to read a document I decided to write after packaging my own software atomes for the Fedora and Debian Linux distributions. I wanted to put together the results of these experiences so that it could help others to take advantage of the fantastic opportunities of the open source communities.

But the fact is that I am neither a specialist of building systems, git, GitHub, GitLab, RPM or even Debian packaging. I merely have now a broad perspective of the whole system.

I had help, to say the least, along the way.

Help when I pushed open the doors of the Fedora and the Debian communities where I got lucky to find people who supported atomes, and took the time to teach me to make it happen ... not only that but they also accepted to proof read this document afterwards, crazy ... thank you so, so, very much:

- My Fedora mentor: **Alexander Ploumistos**
- My Debian mentor: **Pierre Gruet**

I also have to thank Sylvain Thery, a colleague and Git specialist, who kindly destroyed the first version of the Git chapter of this manual.

Contents

Introduction

Purpose

The purpose of this tutorial is to introduce the essential steps required to publish an open source software. It is both a personal review of the process and a collection of tips, intended to help the programmer navigates from its source code up to free software distribution.
It is not, and does not aim to be comprehensive, but to be a proper recollection of my own experiences, so that these could help others.

Prerequisites

In the following I will consider that the reader is familiar with using the command line.
To learn more about the command line and Linux check out my Linux tutorial.
To learn more about scripting and advanced command line use check out my Bash tutorial.

Target readers

Any one who want to publish an open source program, in particular if that person is interested in having its code distributed within the Linux open source community.

Software build

Starting from the next chapter I will consider that you already have a piece of code written and ready to be used, even ready to be distributed. The programming language does not matter, I will simply focus on what comes next.

The example I will use through out this tutorial is based on my Atomes software, and will offer various illustrations for the build automation process, packaging and distribution of a code that requires:

- To be built using:
  - The gcc compiler (part of the sources are in C).
  - The gfortran compiler (part of the sources are if Fortran90).
- To be linked with:
  - The GTK3 library.
  - The libxml2 library.
  - The MESA GLu library.
  - The libepoxy library.
  - The FFMPEG libraries (libavcodec, libavformat, libavutil, libswscale)

Our concern will be focused on the software building process, that is the process of converting source code file(s) into standalone software(s) that can be run on a computer (see software build process on Wikipedia), and, the build automation process, that is the automating of the creation of a software build and the associated tasks (compiling, packaging, testing ...) (see build automation on Wikipedia).

CHAPTER 1

BUILDING SYSTEM AND AUTOMATION

In order to distribute your code it is almost mandatory to have a build system in mind, that is an automation tool that will help you to build your software using the sources, link it with used libraries, and even handle the installation process.
Historically build automation was handled using basic Makefile(s), but now more advanced tools are available, the most well known likely being:

- Make
- CMake
- Meson

But there are many others.
In the following I will present examples using:

- The GNU Make implementation of the Make building system. The GNU Make is a standard distribution format, if your are familiar with open source software it is likely that your already stumbled upon a GNU tarball (or archive), either in Gzip of Bzip2 format:
  - **`program.tar.gz`**, to be opened using:

    ```
    user@localhost ~]$ tar -zxf program.tar.gz
    ```

  - **`program.tar.bz2`**, to be opened using:

    ```
    user@localhost ~]$ tar -jxf program.tar.bz2
    ```

  Very often the content of this archive is based on the GNU automation build system. In the following I will provide an example of the preparation of basic GNU tarball, that represents the first step in distributing your software to the open source community.

- The CMake cross-platform open-source software for build automation, testing, packaging and installation of software by using a compiler-independent method.
  CMake is not a build system itself, it generates another system's build files and can invoke native building environments such as Make.

- The modern meson build system written in Python.
  Meson is desinged to provide simple, out-of-the-box support for modern software development tools and practises.

1.1 The GNU Autotools

The GNU Autotools, also known as the GNU Build System, is a suite of programming tools designed to assist in making source code packages portable (see GNU Autotools on Wikipedia).

Autotools consists of the GNU utility programs: Autoconf, Automake, and Libtool.
Other related tools are frequently used alongside it:

- GNU's make program
- GNU gettext
- pkg-config
- GNU Compiler Collection, also called GCC.

Before going further into this guide, and if you want to try to build your own GNU tarball, you will have to install some of these tools:

- For Windows and OSX please refer to the corresponding websites.
- For Linux you can quite conveniently use the command line as follow:
  - Red Hat based Linux (using the **dnf** command):

```
user@localhost ~]$ sudo dnf install autoconf automake libtool
user@localhost ~]$ sudo dnf install pkg-config gcc gcc-gfortran
```

  - Debian based Linux (using the **apt** command):

```
user@localhost:~$ sudo apt install autoconf automake libtool
user@localhost:~$ sudo apt install pkg-config gcc gfortran
```

To learn more about software installation on Linux check out my Linux tutorial.
To learn more about scripting and the usage of the command line check out my Bash tutorial.

1.1.1 The GNU tarball or GNU archive

A GNU tarball (archive) is a common way to distribute open source software built using the GNU Autotools, along with the sources of the program it should contain all the following files:

configure.ac	$\Longrightarrow$	A configuration file that defines the compilation options for the project, and the rules to generate the **configure** script to create the tarball.
configure	$\Longrightarrow$	A script generated using the **configure.ac** file. It is used to configure the compilation and installation options based on the operating system and development environment.
Makefile.am	$\Longrightarrow$	A file that contains the build rules for the project. It is used to generate the **Makefile.in** file.
Makefile.in	$\Longrightarrow$	A file used by the **configure** script to generate the **Makefile**.
README(.md)	$\Longrightarrow$	A file that contains important information about the project, such as installation and usage instructions.
INSTALL(.md)	$\Longrightarrow$	A file that contains installation instructions for the project.
AUTHORS(.md)	$\Longrightarrow$	A file that contains information about the author(s) of the project.
COPYING(.md)	$\Longrightarrow$	A file that contains the terms of the license under which the project is distributed.
ChangeLog	$\Longrightarrow$	A file that contains a list of changes made to the project since the last release.
NEWS(.md)	$\Longrightarrow$	A file that contains information about new features or significant changes made to the project.

In order to prepare a GNU tarball it is mandatory to edit and prepare the following files:

- The **configure.ac** file.
- The **Makefile.am** file(s)
  There is usually one file in the upper directory of the archive. However if the sources are organized in the form of a directory tree you might want to provide multiple **Makefile.am** file(s), one at each significant level in the directory tree, this will help to simply the organization of the build process.

At the build process of the GNU archive it is mandatory to have prepared all **Makefile.am** file(s) before being able to process the **configure.ac** file with **autoconf**.

The next sections will illustrate the creation of these 2 files.

1.1.2 The file: `configure.ac`

To start working on your GNU tarball you will need first to create a directory to work in:

```
user@localhost ~]$ mkdir program
user@localhost ~]$ cd program
```

Then use the **`autoscan`** command to generate à **`configure.scan`** file, and rename this file as **`configure.ac`** (you can also remove the empty **`autoscan.log`** file:

```
user@localhost ~/program]$ autoscan
user@localhost ~/program]$ rm -f autoscan.log
user@localhost ~/program]$ mv configure.scan configure.ac
```

Then by editing the **`configure.ac`** file you will see that it follows the general structure:

```
dnl In a configure.ac file, commented line(s), could start by either:
dnl    - 'dnl' and will not appear in the resulting configure script
dnl    - '#' and will appear in the resulting configure script

# Using the instructions AC_INIT and AC_INIT_AUTOMAKE - see [Sec. 1.1.2.1].
AC_INIT([FULL-PACKAGE-NAME], [VERSION], [BUG-REPORT-ADDRESS])
AM_INIT_AUTOMAKE

# Checks for program(s) - see [Sec. 1.1.2.2].

# Check for compiler(s) - see [Sec. 1.1.2.3].

# Optional: check for operating system(s) - see [Sec. 1.1.2.4].

# Check for libraries - see [Sec. 1.1.2.5].

# Using the instruction AC_CONFIG_FILES and AC_OUTPUT - see [Sec. 1.1.2.7].
AC_CONFIG_FILES
AC_OUTPUT
```

In the following I will provide commented examples of the different parts of a standard **`configure.ac`** file, a complete file can be found in appendix A.1.1.

1.1.2.1 Initialization

The **configure.ac** always start with a mandatory **AC_INIT** instruction.

Everything before **AC_INIT** is optional.

A detailed example is provided thereafter to initialize your **configure.ac** file:

```
dnl Ensure to use a minimum version of Autoconf:
AC_PREREQ(2.59)

dnl Defining local variables for major, minor and patch version(s) of the program:
m4_define(major_version, 1)
m4_define(minor_version, 2)
m4_define(patch_version, 12)
dnl Defining local variable for the global version of the program:
m4_define(version, major_version.minor_version.patch_version)
dnl Defining local variable for the bug report email:
m4_define(bug_email, your.email@host.eu)
dnl Defining local variable for the name of the tarball:
m4_define(tar_name, program)
dnl Defining local variable for the project URL:
m4_define(project_url, https://www.program.com)

dnl AC_INIT: performs initialization and verification.
dnl This will define number of variables used afterwards:
dnl     AC_PACKAGE_NAME
dnl     AC_PACKAGE_VERSION
dnl     AC_PACKAGE_BUGREPORT
dnl     AC_PACKAGE_TARNAME
dnl     AC_PACKAGE_URL
AC_INIT([prog], [version], [bug_email], [tar_name], [project_url])

dnl AM_INIT_AUTOMAKE: initializes the Automake build system
dnl It will set up the build environment, using all previously defined variables:
AM_INIT_AUTOMAKE

dnl Defining pre-processor variables for program version(s), major, minor and patch:
dnl     AC_DEFINE([VAR_NAME], [VALUE], [String])
dnl Then replace all instances of VAR_NAME in the output file(s) by VALUE
dnl     AC_SUBST(VAR_NAME, [VALUE])
AC_DEFINE([MAJOR_VERSION], [major_version], [Program major version])
AC_SUBST(MAJOR_VERSION, major_version)
AC_DEFINE([MINOR_VERSION], [minor_version], [Program minor version])
AC_SUBST(MINOR_VERSION, minor_version)
AC_DEFINE([PATCH_VERSION], [patch_version], [Program patch version])
AC_SUBST(PATCH_VERSION, patch_version)
```

1.1.2.2 Checking for program(s)

At some point you will need to check for the required tools and libraries. The easiest way to do this is by using the **AC_CHECK_*** macros provided by GNU autotools.

The most convenient way to check for library/libraries is to use the command **pkg-config**, however you need to make sure that this command is available on the target system, and therefore you need to check if the program **pkg-config** has been installed:

```
dnl This will check that program is installed.
dnl It will also set the variable to the path of the program executable.
dnl     AC_CHECK_PROG ([variable], [program], [text-if-found], [text-if-not-found])
AC_CHECK_PROG([PKG_CONFIG], [pkg-config], [yes], [no])
dnl Checking for utilities need for Freedesktop Linux integration, see [Sec. 4.1.1.7]:
AC_CHECK_PROG([UP_MIME], [update-mime-database], [yes], [no])
AC_CHECK_PROG([UP_DESKTOP], [update-desktop-database], [yes], [no])
AC_CHECK_PROG([UP_APPSTREAM], [appstream-util], [yes], [no])
```

1.1.2.3 Checking for compiler(s)

The following examples illustrate:

- The macro used to check if compilers flags are suitable to build the program:

```
dnl Compiler(s) flags checking
dnl Definig the macro AX_CHECK_COMPILE_FLAG:
AC_DEFUN([AX_CHECK_COMPILE_FLAG],
    [AC_PREREQ(2.64) dnl for _AC_LANG_PREFIX and AS_VAR_IF
    AS_VAR_PUSHDEF([CACHEVAR], [ax_cv_check_[]_AC_LANG_ABBREV[]flags_$4_$1])
    AC_CACHE_CHECK([whether _AC_LANG compiler accepts $1], CACHEVAR, [
        ax_check_save_flags=$[]_AC_LANG_PREFIX[]FLAGS
        _AC_LANG_PREFIX[]FLAGS="$[]_AC_LANG_PREFIX[]FLAGS $4 $1"
        AC_COMPILE_IFELSE([m4_default([$5],[AC_LANG_PROGRAM()])],
            [AS_VAR_SET(CACHEVAR,[yes])],
            [AS_VAR_SET(CACHEVAR,[no])])
        _AC_LANG_PREFIX[]FLAGS=$ax_check_save_flags])
    AS_VAR_IF(CACHEVAR,yes,
        [m4_default([$2], :)],
        [m4_default([$3], :)])
    AS_VAR_POPDEF([CACHEVAR])
])dnl AX_CHECK_COMPILE_FLAG
```

  this macro is included in the **autoconf-archive** package, providing that this package is installed you can simply call AX_CHECK_COMPILE_FLAG (see aftewards) without inserting the macro in your file **configure.ac**.

- The instructions checking for compiler(s) required to build the program:

```
dnl Searching for a working C compiler:
AC_PROG_CC
dnl Checking for C FLAGS, provided on the configure line, ex: CFLAGS="-O3"
AX_CHECK_COMPILE_FLAG([$CFLAGS])

dnl Checking for a working Fortran 90 compiler
dnl Ensuring C and F90 function names compatibility:
AC_FC_WRAPPERS
dnl Switching to Fortran language:
AC_LANG_PUSH([Fortran])
dnl Searching for a Fortran 90 compiler:
dnl     AC_PROG_FC ([compiler-search-list], [dialect])
AC_PROG_FC([xlf95 fort ifort ifc f95 g95 pgf95 lf95 xlf90 f90 pgf90 gfortran],
           [90])
dnl Specify the Fortran file(s) extension(s) for the compiling test:
AC_FC_SRCEXT(f90, FCFLAGS_f90="$FCFLAGS_f90 $FCFLAGS",
               AC_MSG_ERROR([Err. comp. .f90]))
AC_FC_SRCEXT(F90, FCFLAGS_F90="$FCFLAGS_F90 $FCFLAGS",
               AC_MSG_ERROR([Err. comp. .F90]))
dnl Checking for Fortran 90 FLAGS, provided on the configure line, ex: FCFLAGS="-O3"
AX_CHECK_COMPILE_FLAG([$FCFLAGS])
dnl Find required linker Fortran flags:
AC_FC_LIBRARY_LDFLAGS
dnl Ensure that Free form Fortran is allowed in the code:
AC_FC_FREEFORM
dnl Leaving Fortran language:
AC_LANG_POP([Fortran])
```

1.1.2.4 Checking for the target Operating System

Finally it might be required to test for the target operating system:

```
dnl Checking for the target operating system

# AC_CANONICAL_HOST is needed to access the '$host_os' variable
AC_CANONICAL_HOST

build_linux=no
build_windows=no
build_mac=no

# Detect the target system using the $host_os variable
case "$host_os" in
    linux*)
        build_linux=yes
        echo "Building Linux application"
        ;;
    cygwin*|mingw*)
        build_windows=yes
        echo "Building Windows application"
        ;;
    darwin*)
        build_mac=yes
        echo "Building macOS application"
        ;;
    *)
        AC_MSG_ERROR(["OS $host_os is not supported"])
        ;;
esac

# Pass the conditionals to automake
AM_CONDITIONAL([LINUX], [test "$build_linux" = "yes"])
AM_CONDITIONAL([WINDOWS], [test "$build_windows" = "yes"])
AM_CONDITIONAL([OSX], [test "$build_mac" = "yes"])

if test "$build_windows" = "yes"; then
  AC_CHECK_TOOL([WINDRES], [windres])no
  if test "$WINDRES" = no; then
    AC_MSG_ERROR([*** Could not find an implementation of windres in your PATH.])
  fi
fi
```

1.1.2.5 Checking for libraries

The easiest way to do this is to use **pkg-config**.
pkg-config defines and supports a unified interface for querying installed libraries for the purpose of compiling software that depends on them. It allows programmers and installation scripts to work without explicit knowledge of detailed library path information.
To check the libraries installed on your system, and handled by **pkg-config**, use:

```
user@localhost ~]$ pkg-config --list-all
```

With the list of libraries installed and the associated keywords, it is easy to prepare the testing:

```
dnl To check if a library is present use the command:
dnl     PKG_CHECK_MODULES([Name], [keyword optionally version required])
dnl It calls pkg-config to check for the libraries mentioned using keyword
dnl Name is a keyword used to construct variables related to designated library.
dnl Ex:
dnl     PKG_CHECK_MODULES([LGTK], [gtk4 >= 4.60])
dnl     Will create a LGTK_LIBS variable to store the gtk4 linker flags
dnl     You can then re-use these variables later on, in particular in Makefile.am

dnl Checking for libxml-2.0, version must be > 2.4.0:
PKG_CHECK_MODULES([LIBXML2], [libxml-2.0 >= 2.4.0])

dnl Checking for glu, no version requirement, only Linux and macOS:
if test "$build_windows" = "no"; then
  PKG_CHECK_MODULES([GLU], [glu])
fi

dnl Checking for epoxy, no version requirement:
PKG_CHECK_MODULES([EPOXY], [epoxy])

dnl Checking for libavcodec, and other FFMPEG based libraries, no version requirement:
PKG_CHECK_MODULES([FFMPEG], [libavcodec libavformat libavutil libswscale])
```

As you can understand from this example the testing could actually be performed on a single line for all libraries. However I would recommend to split the test in as many libraries as required.

When running the **configure** script to perform the testing the standard output is:

```
checking for Name... [yes or no]
```

Separate checking allows to obtain direct information on the issue without opening the **config.log** (very detailed output of the configure process) file that contains more details about the process.

1.1.2.6 Creating additional `configure` options

To create new flag(s) for the **`configure`** script command line to test for specific option(s).

```
# To create a configure option ot determine the GTK version to use:

dnl 1) Displaying an information message at the configure stage:
AC_MSG_CHECKING([the GTK version to use])

dnl 2) Test for the presence of the appropriate argument, the synthax is:
dnl AC_ARG_WITH([option_name], [Help message], [if flag], [if no flag])
dnl In the following:
dnl   - To use GTK4: ./configure --with-gtk=4
dnl   - To use GTK3: ./configure  (or ./configure --with-gtk=3)
AC_ARG_WITH([gtk],
  [AS_HELP_STRING([--with-gtk=3|4], [the GTK version to use (default: 3)])],
  [case "$with_gtk" in
       3|4) ;;
       *) AC_MSG_ERROR([invalid GTK version specified]) ;;
         esac],
  [with_gtk=3])

dnl 3) Result in information message at the configure stage:
AC_MSG_RESULT([$with_gtk])

dnl 4) optional: create a GTK3 variable if using GTK version 3, see src/Makefile.am
AM_CONDITIONAL([GTK3], [test "$with_gtk" == "3"])

dnl 5) Prepare the PKG_CONFIG test for the GTK version specified by the optional flag:
case "$with_gtk" in
  3) gtk_version="gtk+-3.0"
     gtk_version_number="3.16"
     ;;
  4) gtk_version="gtk4"
     gtk_version_number="4.6"
     ;;
esac

dnl 5) Testing for the proper library version following the option on the configure line:
PKG_CHECK_MODULES([LGTK], [$gtk_version >= $gtk_version_number])
if test "$with_gtk" = "3"; then
  if test "$build_mac" = "yes"; then
    PKG_CHECK_MODULES([INTEGRATION], [gtk-mac-integration])
  fi
fi
```

```
dnl To create a configure to disable OpenMP
dnl if not disable (default) then determine the appropriate compiler flags:

AC_MSG_CHECKING([the OpenMP specification and flags])
AC_ARG_ENABLE([--disable-openmp],
  [AS_HELP_STRING([--disable-openmp], [disable OpenMP (default: no)])],
  [with_opemp="no"], with_openmp="yes")
AC_MSG_RESULT([$with_openmp])
if test "$with_openmp" = "yes"; then
  AC_OPENMP
  AC_SUBST([OPENMP_CFLAGS])
dnl optional: create an OPENMP variable to be used in src/Makefile.am
  AM_CONDITIONAL([OPENMP], [test "$OPENMP_CFLAGS" != ""])
fi
```

1.1.2.7 Finalization

```
dnl Where to look for Makefile.am file(s) and generate Makefile(s)
AC_CONFIG_FILES([
  Makefile
  src/Makefile
])

dnl AC_CONFIG_HEADERS: optional, tells AC_OUTPUT to creates a preprocessor header.
dnl This header will includes a list of all the variables defined at this stage.
dnl This allows to call these variables in your code using '#include <config.h>'
AC_CONFIG_HEADERS([config.h])

dnl Last instruction in the script
AC_OUTPUT
```

1.1.3 The file(s): `Makefile.am`

Let us consider the following example:

```
user@localhost ~/program]$ ls -lh
-rw-r--r--. 1 user group  481 24 mars  11:24 AUTHORS
-rw-r--r--. 1 user group 2,8K 24 mars  11:24 ChangeLog
-rw-r--r--. 1 user group 3,6K 24 mars  11:24 configure.ac
-rw-r--r--. 1 user group  34K 24 mars  11:24 COPYING
drwxr-xr-x. 2 user group 4,0K 24 mars  11:24 data
-rw-r--r--. 1 user group  16K 24 mars  11:24 INSTALL
-rw-r--r--. 1 user group 4,0K 24 mars  11:24 Makefile.am
drwxr-xr-x. 2 user group 4,0K 24 mars  11:24 metadata
-rw-r--r--. 1 user group  247 24 mars  11:24 NEWS
drwxr-xr-x. 4 user group 4,0K 24 mars  11:24 pixmaps
-rw-r--r--. 1 user group 4,8K 24 mars  11:24 README
drwxr-xr-x. 2 user group 4,0K 24 mars  11:24 src
```

The top directory contains 4 subfolders:

- **`src`**:

```
user@localhost ~/program]$ ls -lh src
-rw-r--r--. 1 user group 13K 24 mars  11:24 file-1.f90
-rw-r--r--. 1 user group 13K 24 mars  11:24 file-2.f90
-rw-r--r--. 1 user group 13K 24 mars  11:24 gui.c
-rw-r--r--. 1 user group 13K 24 mars  11:24 main.c
-rw-r--r--. 1 user group 13K 24 mars  11:24 Makefile.am
-rw-r--r--. 1 user group 13K 24 mars  11:24 mod.f90
```

- **`data`**:

```
user@localhost ~/program]$ ls -lh data
-rw-r--r--. 1 user group 0 24 mars  11:24 file-1.dat
-rw-r--r--. 1 user group 0 24 mars  11:24 file-2.dat
```

- **`pixmaps`**:

```
user@localhost ~/program]$ ls -lh pixmaps/*
-rw-r--r--. 1 user group 5K 24 mars  11:24 pixmaps/pix-3.dat

pixmaps/pix-1:
-rw-r--r--. 1 user group 5K 24 mars  11:24 pix-1-a.dat
-rw-r--r--. 1 user group 5K 24 mars  11:24 pix-1-b.dat

pixmaps/pix-2:
-rw-r--r--. 1 user group 5K 24 mars  11:24 pix-2-a.dat
-rw-r--r--. 1 user group 5K 24 mars  11:24 pix-2-b.dat
```

- **metadata** (this is used for Freedesktop integration, see [Sec. 4.1.1.7]):

```
user@localhost ~/program]$ ls -lh metadata
-rw-r--r--. 1 user group 0 24 mars  11:24 com.program.www.appdata.xml
-rw-r--r--. 1 user group 0 24 mars  11:24 program.desktop
-rw-r--r--. 1 user group 0 24 mars  11:24 program-mime.xml

metadata/icons:
-rw-r--r--. 1 user group 0 24 mars  11:24 program.svg
-rw-r--r--. 1 user group 0 24 mars  11:24 program-project.svg
-rw-r--r--. 1 user group 0 24 mars  11:24 program-workspace.svg
```

First it is required to edit both **Makefile.am** and **src/Makefile.am** to define how to install and build the project.

In this case:

- The **Makefile.am** (in the top directory) will:
  - Reference and call the **src/Makefile.am**, see [Sec. 1.1.3.1].
  - Contain install instructions for the documentation, see [Sec. 1.1.3.1].
  - Contain install instructions for the manual pages, see [Sec. 1.1.3.1].
  - Contain install instructions for architecture independent data, see [Sec. 1.1.3.1].
  - Contain uninstall instructions for all of the above, see [Sec. 1.1.3.1].
- The **src/Makefile.am** (in the **src** directory) will:
  - Contain install instructions for the architecture dependent file(s), see [Sec. 1.1.3.2].
  - Contain everything needed to build the program from the sources, see [Sec. 1.1.3.2].

To understand how a program is built using GNU Make give a look to all **Makefile.am**.

Basically the 2 most important types of instructions in a **Makefile.am** file are:

- Target(s) that determine the action(s) to be performed when you run the **make** command. Targets are defined using the syntax "**target_name:**" example:

```
clean:
```

- Variable(s) that allow you to define and reuse values throughout the process. Variables are defined using the syntax "**variable_name =**" example:

```
SUBDIRS =
```

1.1.3.1 Top directory `Makefile.am`

Initialization

```
# Some variables are predefined, built automatically by the GNU automation tools:
# srcdir the GNU tarball top directory
# datadir the installation directory, often: /usr/share
# docdir the target system documentation directory, often: /usr/share/doc
# mandir the target system man pages directory, often: /usr/share/man
# bindir the target system binary directory, often: /usr/bin
# pkgdatadir the program installation directory, usually named after it, ex: program

# DESTDIR root directory of installation use for packaging (RPM or DEB).
# Empty by default, must be used to construct all path variables for packaging.
# Considering that our purpose is to do just that, we will prepare it now:

prog_datadir = $(DESTDIR)$(datadir)
prog_docdir = $(DESTDIR)$(docdir)
prog_mandir = $(DESTDIR)$(mandir)
prog_pkgdatadir = $(DESTDIR)$(pkgdatadir)
prog_desktopdir = $(prog_pkgdatadir)/applications
prog_metadir = $(prog_pkgdatadir)/metainfo
prog_mimedir = $(prog_pkgdatadir)/mime/packages
prog_iconsdir = $(prog_pkgdatadir)/pixmaps

# Subdirectory(ies) that might contain others Makefile.am files to be processed
# SUBDIRS = dir1 dir2 dir3 ...
# In this example there is one more Makefile.am file to be processed: src/Makefile.am
SUBDIRS = src
```

Installing documentation and manual pages data

```
# If required specify the directory for the documentation, using the naming convention:
#     program_docdir
# Where only the suffix _docdir does matter, and the DESDIR variable is not used:
prog_docdir = $(docdir)
# And the data to be installed, using the naming convention:
#     program_doc_DATA
# Where only the suffix _doc_DATA does matter.
prog_doc_DATA = \
    README.md \
    AUTHORS \
    ChangeLog

# If required specify the directory for the manual page(s), using the naming convention:
#     program_mandir
# Where only the suffix _mandir does matter, and the DESDIR variable is not used:
# And the manual page(s) data to be installed, using the naming convention:
prog_mandir = $(mandir)/man1/
# Note the man1 directory is dedicated to manual pages for applications.
# You will need to adapt this if your distribution a library, or else.
# And the data to be installed, using the naming convention:
#     program_man_DATA
# Where only the suffix _man_DATA does matter.
prog_man_DATA = \
    program.1.gz
```

Installing architecture independent data

```
# To install architecture independent file(s) use the install-data-local target:
install-data-local:

# Example with the installation of the files located in the 'data' directory in the tarball
# The files are to be installed in $(pkgdatadir)/data
# In the following the instructions:
#       'test -d' means 'exists and is a directory'
#       'test -f' means 'exists and is a file'
     if test -d $(srcdir)/data; then \
          $(mkinstalldirs) $(prog_pkgdatadir)/data; \
          for data in $(srcdir)/data/*; do \
               if test -f $$ data; then \
                    $(INSTALL_DATA) $data $(prog_pkgdatadir)/data; \
               fi \
          done \
     fi

# If the files to be installed are located in a directory tree,
# then it is required to re-create this directory tree at install.
# Examples with files located in the 'pixmaps' directory in the tarball
     if test -d $(srcdir)/pixmaps; then \
          $(mkinstalldirs) $(prog_pkgdatadir)/pixmaps; \
          for pixmap in $(srcdir)/pixmaps/*; do \
               if test -f $$pixmap; then \
                    $(INSTALL_DATA) $$pixmap $(prog_pkgdatadir)/pixmaps; \
               else \
                    $(mkinstalldirs) $(prog_pkgdatadir)/$$pixmap ; \
                    for pixma in $$pixmap/*; do \
                         if test -f $$pixma; then \
                              $(INSTALL_DATA) $$pixma $(prog_pkgdatadir)/$$pixmap; \
                         fi \
                    done \
               fi \
          done \
     fi

# Program's icons
     if [ ! -d $(prog_iconsdir) ]; then \
          $(mkinstalldirs) $(prog_iconsdir); \
     fi
     $(INSTALL_DATA) $(srcdir)/metadata/icons/program.svg $(prog_iconsdir)
     $(INSTALL_DATA) $(srcdir)/metadata/icons/program-project.svg $(prog_iconsdir)
     $(INSTALL_DATA) $(srcdir)/metadata/icons/program-workspace.svg $(prog_iconsdir)
# Custom MIME type
     if [ ! -d $(prog_mimedir) ]; then \
          $(mkinstalldirs) $(prog_mimedir); \
     fi
     $(INSTALL_DATA) $(srcdir)/metadata/mime/program-mime.xml $(prog_mimedir)
# Meta info, the following instructions are required for Freedekstop integration, see [Sec. 4.1.1.7].
     if [ ! -d $(prog_metadir) ]; then \
          mkdir -p $(prog_metadir); \
     fi
     $(INSTALL_DATA) $(srcdir)/metadata/com.program.www.appdata.xml $(prog_metadir)
# Desktop file, the following instructions are required for Freedesktop integration, see [Sec. 4.1.1.7].
     if [ ! -d $(prog_desktopdir) ]; then \
          mkdir -p $(prog_desktopdir); \
     fi
# Finalize the Freedesktop integration:
     appstream-util validate-relax --nonet $(prog_metadir)/com.program.www.appdata.xml
     desktop-file-install --vendor="" \
          --dir=$(prog_desktopdir) -m 644 \
          $(prog_desktopdir)/program.desktop
     touch -c $(prog_iconsdir)
     if [ -u `which gtk-update-icon-cache` ]; then \
          gtk-update-icon-cache -q $(prog_iconsdir); \
     fi
# If this is a built from sources not to build a package we need to update desktop and MIME databases:
     if [ -z "$(DESTDIR)" ]; then \
          update-desktop-database $(prog_desktopdir) &> /dev/null || :; \
          update-mime-database $(prog_datadir)/mime &> /dev/null || :; \
     fi
```

Uninstall section

```
# This section should regroup all uninstall instructions:
# - Files or directories created installing architecture independent file(s):
#     - Using the install-data-local target.
#     - Using documentation or manual pages.
#     - Using custom user-designed target(s).
uninstall-local:
    -rm -rf $(prog_pkgdatadir)/data/*
    -rmdir $(prog_pkgdatadir)/data
    -rm -rf $(prog_pkgdatadir)/pixmaps/*
    -rmdir $(prog_pkgdatadir)/pixmaps
    -rmdir $(prog_pkgdatadir)
    -rmdir $(prog_docdir)
    -rm -f $(prog_iconsdir)/program.svg
    -rm -f $(prog_iconsdir)/program-project.svg
    -rm -f $(prog_iconsdir)/program-workspace.svg
    -rm -f $(prog_desktopdir)/program.desktop
    -rm -f $(prog_metadir)/com.program.www.appdata.xml
    -rm -f $(prog_mimedir)/packages/program-mime.xml
    touch -c $(prog_iconsdir)
    if [ -u `which gtk-update-icon-cache` ]; then \
      gtk-update-icon-cache -q $(prog_iconsdir); \
    fi
# If this is a built from sources not to build a package,
# then we need to update desktop and MIME databases:
    if [ -z "$(DESTDIR)" ]; then \
        update-desktop-database $(prog_desktopdir) &> /dev/null || :; \
        update-mime-database $(prog_datadir)/mime &> /dev/null || :; \
    fi
```

In both sections 1.1.3.1 and 1.1.3.1 files located in the folder "**metadata**" are introduced. These files ("***.xml**", "***.desktop**", "***.svg**") are required to integrate properly you program with the Linux ecosystem:

- **program-mime.xml**: use to create the file association with your program, so that a specific icon is used, and that this type of file is to be open using your program, see [Sec. 4.1.1.7] for more information.
- **com.program.www.appdata.xml**: metadata about your program, use for instance to integrate the software manager with proper information, see [Sec. 4.1.1.7] for more information.
- **program.desktop**: the desktop icon for your program, use to create launchers, menus ... see [Sec. 4.1.1.7] for more information.
- ***.svg**: the icons for the program and the associated file formats.

1.1.3.2 src/Makefile.am

```
# Setup the name of the program to be generated, using:
#     bin_PROGRAMS = name
# This instruction also defines the install instruction for the architecture dependent binary file,
# that instruction being to install name in $(bindir)
bin_PROGRAMS = prog
# Other architecture dependent file(s) can be installed using the install-exec-local: target.

# In the following you need to specify the flags used during the compilation process.
# To do that simply use the variables that will be created by the configure script.
# Appropriate variables are created for each library searched for by a Name specified in the configure.ac
# Variables are constructed using the Name_ prefix, as in the example:
#     - In configure.ac: PKG_CHECK_MODULES(LGTK, [gtk+-3.0 >= 3.16])
#     - Creates and setup the following variables to store flags: LGTK_LIBS, LGTK_CFLAGS
# It is mandatory to declare to the linker the libraries to link the program with, using:
#     name_LDADD = $(lib1_LIBS) $(lib2_LIBS)  ...
# Where lib1, lib2, etc, match a Name specified in the configure.ac:
prog_LDADD = $(LGTK_LIBS) $(LIBXML2_LIBS) $(PANGOFT2_LIBS) $(FFMPEG_LIBS) $(GLU_LIBS) $(EPOXY_LIBS)

# Create a variable to store the CFLAGS for all the required libraries:
LIB_CFLAGS = $(LGTK_CFLAGS) $(LIBXML2_CFLAGS) $(PANGOFT2_CFLAGS) $(FFMPEG_CFLAGS) $(GLU_CFLAGS) $(EPOXY_CFLAGS)

# OPENMP is defined at configure stage in OpenMP is avaiable
if OPENMP
  OpenMP_FLAGS = -DOPENMP $(OPENMP_CFLAGS)
else
  OpenMP_FLAGS =
endif
# GTK3 is defined, or not, at the configure stage
# The following implies that code parts for GTK3/4 are separated by preprocessor instructions GTK3 or GTK4
if GTK3
  GTK_VERSION=-DGTK3
else
  GTK_VERSION=-DGTK4
endif

# You can add custom compilation flags to the one already provided by the configure script:

# Use AM_LDFLAGS to define additional linker flags:
AM_LDFLAGS = $(OpenMP_FLAGS)
# Use AM_CPPFLAGS to define additional preprocessor flags:
AM_CPPFLAGS = $(GTK_VERSION) $(OpenMP_FLAGS)
# Use AM_FFLAGS to define additional Fortran 90 flags:
AM_FFLAGS = $(OpenMP_FLAGS)
# Use AM_CFLAGS to define additional C flags, in particular library flags:
AM_CFLAGS = $(LIB_CFLAGS)

# Then declare the sources to build the program:
# Fortran source file(s)
prog_fortran_files = file-1.f90 file-2.f90
# Fortran source module(s)
prog_fortran_modules = mod.f90
# Rules to ensure that Fortran modules are compiled before Fortran files
prog_fortran = $(prog_fortran_modules) $(prog_fortran_files)
$(patsubst %.F90,%.o,$(prog_fortran_files)): $(patsubst %.F90,%.o,$(prog_fortran_modules))

# C source file(s)
prog_c = main.c gui.c

# A target to deal with cleaning properly:
clean:
    -rm -f *.mod
    -rm -f */*.o

# The "prog_SOURCES" instruction lists the files required to build the program:
prog_SOURCES = $(prog_fortran) $(prog_c)
```

1.1.4 Using the GNU Autotools to build the GNU tarball

Now that all the files required to build the GNU tarball have been prepared, you simply need to prepare the **configure** script. At this stage the it is mandatory to have the following files in the top directory of the project:

- **configure.ac**
- **Makefile.am**
- **README**
- **INSTALL**
- **AUTHORS**
- **COPYING**
- **ChangeLog**
- **NEWS**

Then providing that the file **configure.ac** and the file(s) **Makefile.am** have been created, it is possible to build the GNU tarball in 4 steps:

1. Use **aclocal** to generate the macros required by the **configure.ac** script:

```
user@localhost ~/program]$ aclocal
```

2. Use **autoheader** to generate the headers **.h.in** files using the **configure.ac** file:

```
user@localhost ~/program]$ autoheader
```

The **configure** script will create the preprocessor headers as declared in the **configure.ac** file using the **AC_CONFIG_HEADERS()** instruction. To achieve this it requires to work using pre-edited versions of the headers having the **.h.in** extension and created by **autoheader**.

3. Use **autoconf** to generate the **configure** script using the **configure.ac** file:

```
user@localhost ~/program]$ autoconf
```

4. Use **automake** to generate the **Makefile.in** files, that are used to generate **Makefile** when the software package is configured, using the **Makefile.am** files:

```
user@localhost ~/program]$ automake --add-missing --copy
```

When this is done the content of the project top directory looks like:

```
user@localhost ~/program]$ ls -lh
-rw-r--r--.  1 user group  54K 24 mars  17:28 aclocal.m4
-rw-r--r--.  1 user group  481 24 mars  11:24 AUTHORS
drwxr-xr-x.  2 user group 4,0K 24 mars  17:28 autom4te.cache
-rw-r--r--.  1 user group 2,8K 24 mars  11:24 ChangeLog
lrwxrwxrwx.  1 user group   32 24 mars  17:24 compile
lrwxrwxrwx.  1 user group   37 24 mars  17:24 config.guess
lrwxrwxrwx.  1 user group   35 24 mars  17:24 config.sub
-rwxr-xr-x.  1 user group 243K 24 mars  17:28 configure
-rw-r--r--.  1 user group 3,6K 24 mars  11:24 configure.ac
-rw-r--r--.  1 user group  34K 24 mars  11:24 COPYING
drwxr-xr-x.  2 user group 4,0K 24 mars  11:24 data
lrwxrwxrwx.  1 user group   32 24 mars  17:24 depcomp
-rw-r--r--.  1 user group  34K 24 mars  11:24 config.h.in
-rw-r--r--.  1 user group  16K 24 mars  11:24 INSTALL
lrwxrwxrwx.  1 user group   35 24 mars  17:24 install-sh
-rw-r--r--.  1 user group 4,0K 24 mars  17:03 Makefile.am
drwxr-xr-x.  2 user group 4,0K 24 mars  11:24 metadata
lrwxrwxrwx.  1 user group   32 24 mars  17:24 missing
-rw-r--r--.  1 user group  247 24 mars  11:24 NEWS
drwxr-xr-x.  4 user group 4,0K 24 mars  11:24 pixmaps
-rw-r--r--.  1 user group 4,8K 24 mars  11:24 README
drwxr-xr-x.  2 user group 4,0K 24 mars  11:24 src
```

And the project is ready to be archived, to create the GNU tarball:

```
user@localhost ~/program]$ cd ..
user@localhost ~]$ tar -zcf program.tar.gz program
#  user@localhost ~]$ tar -jcf program.tar.bz2 program
```

To obtain a package for distribution:

```
user@localhost ~]$ ls -lh
drw-r-xr-x.  1 user group  4,0K 24 mars  17:28 program
-rw-r--r--.  1 user group  750K 24 mars  17:28 program.tar.gz
# -rw-r--r--.  1 user group  550K 24 mars  17:28 program.tar.bz2
```

1.1.5 Installing a GNU tarball

You can install a program distributed using a GNU tarball in 3 steps:

1. Use the **configure** script to configure the package for your system:

```
user@localhost ~/program]$ ./configure
```

   At this stage you can also provided your own instruction(s) to the **configure** script:

   - Customize the installation directory using the **--prefix** option:

```
user@localhost ~/program]$ ./configure --prefix=/usr/local
```

   - Tweak the compiler flags **CFLAGS/FCFLAGS** options:

```
user@localhost ~/program]$ ./configure CFLAGS="-O3" FCFLAGS="-O3"
```

   - And many more options, to know more:

```
user@localhost ~/program]$ ./configure --help
```

   If the **configure** script ends successfully, then **Makefile**(s) have been created. Otherwise, if the script fails, then you might need to give a look to the **config.log** that contains detailed information about the configuration process.

2. Use the **make** command to build the package:

```
user@localhost ~/program]$ make
```

   - For better efficiency use the **-j num** option, to build using **num** CPU cores:

```
user@localhost ~/program]$ make -j 12
```

   The **make** command is looking for a **Makefile** in the active directory.
   If the **make** command is successful then the program has been built using this **Makefile**.

3.a Use the **make** command with the **install** target to install the package:

```
user@localhost ~/program]$ make install
```

3.b Use the **make** command with the **uninstall** target to uninstall the package:

```
user@localhost ~/program]$ make uninstall
```

1.2 CMake

CMake is a tool to manage the software build process. Originally, CMake was designed as a generator for various dialects of Makefile, today CMake generates modern buildsystems such as Ninja as well as project files for IDEs such as Visual Studio and Xcode.
Most of the information bellow is adapted from the main documentation:

https://cmake.org/cmake/help/latest/

Also a GUI tool called **cmake-gui** is available, but its usage is not covered in this manual.
Finally please note that in the following I will only consider basic, targeted, usage of CMake, that is a complex tool with much more capabilities.

As for the GNU Autotools, pkg-config will be used extensively thereafter. Before going further into this guide, and if you want to try to build and package your program using CMake, you will have to install the following tools:

- For Windows and OSX please refer to the corresponding websites.
- For Linux you can quite conveniently use the command line as follow:
  - Red Hat based Linux (using the **dnf** command):

```
user@localhost ~]$ sudo dnf install cmake
user@localhost ~]$ sudo dnf install pkg-config gcc gcc-gfortran
```

  - Debian based Linux (using the **apt** command):

```
user@localhost:~$ sudo apt install cmake
user@localhost:~$ sudo apt install pkg-config gcc gfortran
```

To learn more about software installation on Linux check out my Linux tutorial.
To learn more about scripting and the usage of the command line check out my Bash tutorial.

1.2.1 Example project

In the next sections the example project will be similar to the case of the GNU autotools:

```
user@localhost ~/program]$ ls -lh
-rw-r--r--. 1 user group 2,8K 24 mars  11:24 ChangeLog
-rw-r--r--. 1 user group  34K 24 mars  11:24 COPYING
drwxr-xr-x. 2 user group 4,0K 24 mars  11:24 data
drwxr-xr-x. 2 user group 4,0K 24 mars  11:24 metadata
drwxr-xr-x. 4 user group 4,0K 24 mars  11:24 pixmaps
-rw-r--r--. 1 user group 4,8K 24 mars  11:24 README
drwxr-xr-x. 2 user group 4,0K 24 mars  11:24 src
```

The top directory contains 4 subfolders:

- **src**:

```
user@localhost ~/program]$ ls src
file-1.f90    file-2.f90    gui.c    main.c    mod.f90
```

- **data**:

```
user@localhost ~/program]$ ls data
file-1.dat    file-2.dat
```

- **pixmaps**:

```
user@localhost ~/program]$ ls pixmaps/*
pixmaps/pix-3.dat

pixmaps/pix-1:
pix-1-a.dat    pix-1-b.dat

pixmaps/pix-2:
pix-2-a.dat    pix-2-b.dat
```

- **metadata** (this is used for Freedesktop integration, see [Sec. 4.1.1.7]):

```
user@localhost ~/program]$ ls metadata
com.program.www.appdata.xml    program.desktop    program-mime.xml

metadata/icons:
program.svg    program-project.svg    program-workspace.svg
```

1.2.2 The file: `CMakeLists.txt`

The file **`CMakeLists.txt`** is a configuration file that defines:

- The rules to configure your project
- The rules to build your project
- The rules to install your project

In the next pages I will browse briefly the different parts of the file **`CMakeLists.txt`**, however my approach will remain simple. For detailed information check the official user manual

To start working your on CMake package you will need first to create a directory to work in:

```
user@localhost ~]$ mkdir program
user@localhost ~]$ cd program
```

Then simply use a text editor to work on this file and insert instructions regarding configuration, building and installation of your program:

```
user@localhost ~/program]$ vi CMakeLists.txt
```

CMake version information

The file **`CMakeLists.txt`** always starts by the minimum CMake version to use:

```
cmake_minimum_required (VERSION 3.10)
```

Project name, version and programming languages

Describe project information using the **`project`** command:

```
project(prog VERSION 1.2.12)
project(prog DESCRIPTION "A tool box")
project(prog HOMEPAGE_URL "https://www.program.com")
project(prog LANGUAGES C Fortran)
```

Where **`prog`** is the project name, and is the keyword that will used thereafter to associate CMake actions to the project.
CMake will split the **`VERSION`** command in:

```
MAJOR_VERSION.MINOR_VERSION.PATCH_VERSION
```

The **LANGUAGES** option allows to check for compilers, however other dependencies might be needed (see bellow).

For more information see the CMake `project()` command related documentation here.

Dependencies

In this manual I will only focus on the **pkg-config** way of doing things, other possibilities are available and documented in the official CMake documentation:

https://cmake.org/cmake/help/latest/command/find_package.html#find-package

Check for package(s) using the `find_package()` and `pkg_check_modules()` commands:

```
find_package(PkgConfig REQUIRED)

# Checking for gtk+3.0:
pkg_check_modules(GTK3 REQUIRED IMPORTED_TARGET gtk+-3.0)

# Checking for libxml-2.0:
pkg_check_modules(LIBXML2 REQUIRED IMPORTED_TARGET libxml-2.0)

# Checking for libavcodec, and other FFMPEG based libraries:
pkg_check_modules(LIBAVUTIL REQUIRED IMPORTED_TARGET libavutil)
pkg_check_modules(LIBAVCODEC REQUIRED IMPORTED_TARGET libavcodec)
pkg_check_modules(LIBAVFORMAT REQUIRED IMPORTED_TARGET libavformat)
pkg_check_modules(LIBSWSCALE REQUIRED IMPORTED_TARGET libswscale)

# Checking for epoxy:
pkg_check_modules(EPOXY REQUIRED IMPORTED_TARGET epoxy)
```

Where keywords in pink define variable names for the CMake project, eg. GTK3, and keywords in dark green, eg. **gtk3**, are the associated to **pkg-config** libraries (see Sec. 1.1.2.5).
You will also need to link the project to the required libraries, but only after the build target is declared, see page 31.

Compiler options

To add specific compiler flags use the `set()` command with the `CMAKE_<LANG>_FLAGS` argument, `<LANG>` being the target programming language:

```
set(CMAKE_C_FLAGS "-O3")
set(CMAKE_Fortran_FLAGS "-O3")

# The 'OpenMP' keyword is known by CMake
# If the package is found, then it sets variables that include:
#  - OpenMP_FOUND : to let know if the package was found
#  - OpenMP_<LANG>_FLAGS : the OpenMP compiler flags for <LANG>
find_package(OpenMP)
if (OpenMP_FOUND)
  set(CMAKE_C_FLAGS ${OpenMP_C_FLAGS})
  set(CMAKE_Fortran_FLAGS ${OpenMP_Fortran_FLAGS})
endif()
```

Preprocess variable definitions

To define compiler variables use the `add_compile_definitions()` instruction:

```
if (OpenMP_FOUND)
  add_compile_definitions(OPENMP)
endif()

# Creating a variable that contains the installation directory:
set(INSTALL_DIR "${CMAKE_INSTALL_FULLDATADIR}")
add_compile_definitions(LOGO="${INSTALL_DIR}/prog/pixmaps/logo.png")

# CMake defines the following set of variables: LINUX, WINDOWS, APPLE ...
# to help setting OS specific compiler flags or instructions:
if (LINUX)
  add_compile_definitions(LINUX)
elseif (WINDOWS)
  add_compile_definitions(WINDOWS)
elseif (APPLE)
  add_compile_definitions(OSX)
endif()
```

On a GNU compiler command line, and a Linux system, the previous instructions would be translated in:

- With OpenMP:

```
gcc -DLINUX -DOPENMP -DLOGO="/usr/local/share/prog/pixmaps/logo.png"
```

- Without OpenMP:

```
gcc -DLINUX -DLOGO="/usr/local/share/prog/pixmaps/logo.png"
```

Declaring the project sources

In CMake you declare sources in list of file(s), that your refer to using a variable name.

- Recursively, not recommended for a distribution purposes:

```
file(GLOB_RECURSE C_SOURCES RELATIVE ${CMAKE_SOURCE_DIR} "src/*.c")
file(GLOB_RECURSE F_SOURCES RELATIVE ${CMAKE_SOURCE_DIR} "src/*.f90")
```

  C_SOURCES is the name of a variable that will contain the list of files with the extension `".c"` that CMake can find recursively in the source directory `"src"`.
  Similarly **F_SOURCES** is the name of a variable that will contain the list of files with the extension `".f90"` that CMake can find recursively in the source directory `"src"`.

- Folder by folder, then you need to include any sub-folder that contains source file(s):

```
file(GLOB C_SRC RELATIVE ${CMAKE_SOURCE_DIR} "src/*.c")
file(GLOB F_SRC RELATIVE ${CMAKE_SOURCE_DIR} "src/*.f90")
```

  C_SRC is the name of a variable that will contain the list of files with the extension `".c"` that CMake can find in the source directory `"src"` only.
  Similarly **F_SRC** is the name of a variable that will contain the list of files with the extension `".f90"` that CMake can find in the source directory `"src"` only.

Declaring the project building process

How to build the project is declared using the `add_executable()` instruction:

```
add_executable(prog ${C_SRC} ${F_SRC})
```

Where `prog` is the project name (see page 27), followed by the list of sources, here C and Fortran90 sources files declared previously and listed in the `${C_SRC}` and `${F_SRC}` variables.

Declaring project headers include directories

When the building instructions have been specified using the `add_executable()` instruction, then it possible to declare project headers include directory using:

```
target_include_directories(prog PUBLIC src)
```

Where `prog` is the project name (see page 27), followed by a space separated list of directory(ies) to look into to search for header files.

Linking to external libraries

Again only focusing on the **`pkg-config`** way of doing things, but other options are available:

https://cmake.org/cmake/help/latest/command/target_link_libraries.html

To link the project to libraries use the `target_link_libraries()` instructions:

```
target_link_libraries(prog PUBLIC PkgConfig::GTK3)
target_link_libraries(prog PUBLIC PkgConfig::LIBXML2)
target_link_libraries(prog PUBLIC PkgConfig::LIBAVUTIL)
target_link_libraries(prog PUBLIC PkgConfig::LIBAVCODEC)
target_link_libraries(prog PUBLIC PkgConfig::LIBAVFORMAT)
target_link_libraries(prog PUBLIC PkgConfig::LIBSWSCALE)
target_link_libraries(prog PUBLIC PkgConfig::EPOXY)
```

Where `prog` is the project name, see page 27, and the keywords in pink in the `target_link_libraries()` instructions, refers to the keywords used in the `pkg_check_modules()` instructions, eg. `GTK3`, see page 28.

Installation using CMake

To generate installation rules for your project use the `install()` instruction:

```
# First import the GNU installation directory variables
include(GNUInstallDirs)

# To install the main binary file:
install(PROGRAMS prog DESTINATION ${CMAKE_INSTALL_BINDIR})

# To install a directory, including all its content, recursively:
install(DIRECTORY data
        DESTINATION ${CMAKE_INSTALL_DATADIR}/prog
        PATTERN "data/*")
install(DIRECTORY pixmaps
        DESTINATION ${CMAKE_INSTALL_DATADIR}/prog
        PATTERN  "pixmaps/*")
install(DIRECTORY metadata/icons
        DESTINATION ${CMAKE_INSTALL_DATADIR}/pixmaps
        PATTERN "metadata/icons/*.svg")

# To install specific files:
install(FILES metadata/com.program.www.appdata.xml
        DESTINATION ${CMAKE_INSTALL_DATADIR}/metainfo)
install(FILES metadata/program-mime.xml
        DESTINATION ${CMAKE_INSTALL_DATADIR}/mime/packages)
install(FILES metadata/program.desktop
        DESTINATION ${CMAKE_INSTALL_DATADIR}/applications)
# To install ChangeLog
install(FILES ChangeLog DESTINATION ${CMAKE_INSTALL_DOCDIR})
# To install license information
install(FILES COPYING DESTINATION ${CMAKE_INSTALL_DOCDIR})
```

Post-installation script

To use apply post-installation configuration, including system integration, an option is to use a script call by the `install()` command with the **CODE** argument:

```
# Creating a variable that contains the installation directory:
set(INSTALL_DIR "${CMAKE_INSTALL_FULLDATADIR}")

# Running the post installation script:
install(CODE "execute_process(COMMAND ./post-install.sh ${INSTALL_DIR})")
```

This will execute the command `./post-install.sh` which is a script containing the post-installation instructions. It is required to transmit to the script the installation directory selected by CMake, this is done using the variable `${INSTALL_DIR}` in argument of the script. Note that you can define variable, eg. `${INSTALL_DIR}`, using the `set()` instruction.
An example of content for the script `post-install.sh` is provided in appendix A.2.2.

A complete example of the file **CMakeLists.txt** is provided in appendix A.2.1.

1.2.3 Configuring, building and installing a CMake package

To configure, build and install a CMake package use the **cmake** command.
In the following I will consider that the file **CMakeLists.txt** was completed:

```
user@localhost ~/program]$ ls -lh
-rw-r--r--. 1 user group 2,8K 24 mars  12:25 ChangeLog
-rw-r--r--. 1 user group 5,3K 24 mars  12:25 CMakeLists.txt
-rw-r--r--. 1 user group  34K 24 mars  12:25 COPYING
drwxr-xr-x. 2 user group 4,0K 24 mars  12:25 data
drwxr-xr-x. 2 user group 4,0K 24 mars  12:25 metadata
drwxr-xr-x. 4 user group 4,0K 24 mars  12:25 pixmaps
-rw-r--r--. 1 user group 4,8K 24 mars  12:25 README
drwxr-xr-x. 2 user group 4,0K 24 mars  12:25 src
```

The first step of the process is to create a temporary directory to work on to build the package:

```
user@localhost ~/program]$ mkdir build
user@localhost ~/program]$ cd build
```

In CMake tutorials this directory is usually named **build**.

Configuring

To configure use **cmake**:

- Without option
- Followed by the name of the location of the file **CMakeLists.txt**

```
user@localhost ~/program/build]$ cmake ../
```

The single argument of the command is the location of the file **CMakeLists.txt**,
in this example case "**../** = upper directory".
The result of this command is the creation of several files and a directory:

```
user@localhost ~/program/build]$ ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  Makefile
```

The new files and repository are as followed:

- `CMakeCache.txt`: list of environment variables used to build and install your project.
- **CMakeFiles**: a directory that will be used to build your project.

- `cmake_install.cmake`: the installation script generated by CMake for your project.
- **`Makefile`**: the Makefile generated by CMake for your project.

Building

To build use:

- **`cmake`**:
  - With the **`--build`** option
  - Followed by the name of the location of the file **`Makefile`**

```
user@localhost ~/program/build]$ cmake --build . -j 12
```

The first option of the command is to request to build the project, it is followed by the is the location of the file **`Makefile`**, in this example case "`.` = current directory", and optionally by other options, in this example a parallel build (**`-j`**) using `12` threads.

- **`make`**
  - Invoke make in the directory where the **`Makefile`** is located
  - With the project name as target, in this example: **`prog`**

```
user@localhost ~/program/build]$ make prog -j 12
```

Installing

To install use:

- **`cmake`**:
  - With the **`--install`** option
  - Followed by the name of the location of the file **`Makefile`**

```
user@localhost ~/program/build]$ sudo cmake --install . --prefix $HOME
```

The first option of the command is to request to install the project, it is followed by the is the location of the file **`Makefile`**, in this example case "`.` = current directory", and optionally by other options in this example to override the default installation directory using the **`--prefix`** option followed by the new installation directory.

- **make**:
  - Invoke make in the directory where the **Makefile** is located
  - With the **install** option

```
user@localhost ~/program/build]$ sudo make install
```

1.2.4 Packaging with CPack

CPack is a cross-platform software packaging tool distributed with CMake. It can be used with or without CMake, to use CPack with CMake simply add the following instruction in the file **CMakeLists.txt**:

```
include(CPack)
```

Then after the configuration stage, new files are created in the active directory:

```
user@localhost ~/program/build]$ ls
CMakeCache.txt        CMakeFiles                cmake_install.cmake
CPackConfig.cmake     CPackSourceConfig.cmake   Makefile
```

The files **CPackConfig.cmake** and **CPackSourceConfig.cmake** can be used by CPack to easily prepare packages to ship the binary version of your project, or the sources:

```
user@localhost ~/program/build]$ cpack -C CPackConfig.cmake -G TBZ2
Pack: Create package using TBZ2
CPack: Install projects
CPack: - Run preinstall target for: prog
CPack: - Install project: prog [CPackConfig.cmake]
CPack: Create package
CPack: - package: ~/program/build/prog-0.1.1-Linux.tar.bz2 generated.
user@localhost ~/program/build]$ ls *.bz2
prog-0.1.1-Linux.tar.bz2
```

CPack creates a package using:

- A configuration file designated using the option **-C** (or **--config**)
- The type of package using the option **-G**

1.2.5 Difference with the GNU autotools

The main difference between the GNU autotools introduced in the first part of the chapter, and CMake, is that you cannot easily uninstall a CMake package, for example using:

```
user@localhost ~/program]$ sudo cmake --uninstall
```

In the CMake case you will need to remove files manually.

However this does not really matter when the project is prepared to be distributed by the Linux software repositories in the RPM or the DEB format.

1.3 Meson

Meson is an open source build system meant to be both fast, and as user friendly as possible. Most of the information bellow is adapted from the main documentation:

https://mesonbuild.com/Manual.html

As for the other build systems, pkg-config will be used extensively thereafter, but in the case of meson it is not mandatory to call it directly.
Before going further into this guide, and if you want to try to build and package your program using meson, you will have to install the following tools:

- For Windows and OSX please refer to the corresponding websites.
- For Linux you can quite conveniently use the command line as follow:
  - Red Hat based Linux (using the **dnf** command):

```
user@localhost ~]$ sudo dnf install meson
user@localhost ~]$ sudo dnf install pkg-config gcc gcc-gfortran
```

  - Debian based Linux (using the **apt** command):

```
user@localhost:~$ sudo apt install meson
user@localhost:~$ sudo apt install pkg-config gcc gfortran
```

To learn more about software installation on Linux check out my Linux tutorial.
To learn more about scripting and the usage of the command line check out my Bash tutorial.

1.3.1 Example project

Same as for the GNU autotools and CMake, see section 1.2.1 for more information.

1.3.2 The file: `meson.build`

The file **`meson.build`** is a configuration file that defines:

- The rules to configure your project
- The rules to build your project
- The rules to install your project

In the next pages I will browse briefly the different parts of the file **`meson.build`**, however my approach will remain simple. For detailed information check the official user manual

To start working on your meson package you will need first to create a directory to work in:

```
user@localhost ~]$ mkdir program
user@localhost ~]$ cd program
```

Then simply use a text editor to work on this file and insert instructions regarding configuration, building and installation of your program:

```
user@localhost ~/program]$ vi meson.build
```

Note that in the **`meson.build`** file the following rules apply:

- Text strings must delimited by single quotes: '*text here*'
- Double quotes " are not allowed

Project description, version and programming languages

Describe project information using the **`project()`** instruction:

```
project('prog', 'c', 'fortran', license : 'AGPL-3.0-or-later', version : '1.2.12')
```

Where **`prog`** is the project name, followed by programming languages, and other coma separated elements.

Project information must be defined using that command, and there is only one single **`project()`** instruction that can be found in the file **`meson.build`**.

It is possible to split the instruction on several lines for clarity purposes:

```
project('prog', 'c', 'fortran',
                license : 'AGPL-3.0-or-later',
                version : '1.2.12')
```

From that single instruction meson prepare several built-in variables stored in the meson project data structure.

These variables can be accessed using the following syntax:

this_data = meson.*data_name*()

Where the variable **this_data** will be used to store the information obtained from the meson project data structure when searching for *data_name*.

Examples:

```
# Define a variable to store the project name (prog):
project_name = meson.project_name()

# Define a variable to store the project license (AGPL-3.0-or-later):
project_license = meson.project_license()

# Define an array variable to store each part of the version information
# the .split() instruction defines how parts are being separated, in this example by dots
project_version = meson.project_version().split('.')
# project_version[0] = 1
# project_version[1] = 2
# project_version[2] = 12
```

Note that the file **meson.build** contain several others data structures for which information can be accessed similarly:

this_data = data_structure.*data_name*()

It is also to retrieve the value of several built-in, or user defined options using the get_option function, the syntax is the following:

```
this_option = get_option('option_name')
```

Examples:

```
# The complete installation prefix is stored in a built-in variable named 'prefix':
project_prefix = get_option('prefix')

# More illustrations page~42
```

To know more about meson variables and syntax:

https://mesonbuild.com/Syntax.html

To display the content of a variable in the configuration output, use the **message()** instruction with coma separated elements:

```
message('My project name is', project_name)
```

To obtain in the output:

```
Message: My project name is prog
```

Dependencies

In this manual I will only focus on the **pkg-config** way of doing things, other possibilities are available and documented in the official meson documentation:

https://mesonbuild.com/Dependencies.html

Check for package(s) using the **dependency()** instruction, note that meson allows to directly use the **pkg-config** keyword as argument of this instruction:

```
# Checking for libxml-2.0:
xml = dependency('libxml-2.0')

# Checking for libavcodec, and other FFMPEG based libraries:
avutil = dependency('libavutil')
avcodec = dependency('libavcodec')
avformat = dependency('libavformat')
swscale = dependency('libswscale')

# Declaration of a variable to store all FFMPEG dependencies:
ffmpeg = [avutil, avcodec, avformat, swscale]

# Checking for epoxy:
epoxy = dependency('epoxy')

# Declaration of a variable to store, and easily re-use, all dependencies:
all_deps = [gtk, xml, glu, epoxy, ffmpeg]
```

This allows to define variables, eg. **gtk**, to store the information regarding each dependencies, using the keyword in pink, eg. `gtk+-3.0` associated to the **pkg-config** libraries (see Sec. 1.1.2.5). Note that keywords are considered as strings and must therefore be inserted between single quotes, eg. `'gtk+-3.0'`.
You will also need to link the project to the required libraries, but only at the moment the build target is declared, see page 44.

By default dependencies are required, if one is not found the configuration will fail and stop. However it is possible to make a dependency optional setting the **required** option to `false`, (the default value being `true`):

```
omp = dependency('openmp', required : false)
```

Compiler options

Default compiler flags and options are automatically defined by meson at the configure stage depending on the compiler(s) selected or available in the system.
To define additional custom compiler options you can either specify flags on the meson configure command line (see Sec. 1.3.3), or, create a dedicated section in the file **meson.build**:

```
# The type of build is stored in the meson built-in variable 'buildtype'
build_type = get_option('buildtype')

# Then creating variables to store adapted compiler options
if build_type == 'release'
# Release version
  cc_args = ['-O3']
  fc_args = ['-O3']
elif build_type == 'debug'
# Debug version
  cc_args = ['-O0', '-g3', '-fvar-tracking', '-fbounds-check']
  fc_args = ['-O0', '-g3', '-fvar-tracking', '-fbounds-check']
else
# Any other type of build
  cc_args = ['-O2', '-g']
  fc_args = ['-O2', '-g']
endif
```

The variables created at this stage to store the compiler options, **cc_args** and **fc_args** for respectively the C and the Fortran compiler, can be used later in the build process, see page 44.

Preprocessor variable definitions

To define compiler variables use the `add_project_arguments()` instruction:

```
add_project_arguments(FLAG, language : 'target_language')
```

Where `FLAG` is the flag to pass to the compiler, and `'target_language'` belong to the list of programming language used in the project and defined in the `project()` instruction (see Sec 1.3.2), example:

```
# meson handles the creation of file path easily using the / symbol:
package_prefix = get_option('prefix') / get_option('datadir') / 'prog'

# building strings is a little bit more tricky, and requires to use the .join function:
# new_string = '-link-'.join('bla', 'bla', 'bla')
#  -  '-link-' the concatenation motif, can be empty
#  - .join('bla', 'bla', 'bla') : coma separated list of strings to concatenate
# Result, store in new_string: 'bla-link-bla-link-bla'
package_logo = ''.join('-DLOGO="', package_prefix, '"')

add_project_arguments(package_logo, language : 'c')
```

For the GNU C compiler command line, the previous instructions would be translated in:

```
gcc -DLOGO="/usr/local/share/prog/pixmaps/logo.png"
```

Example of applications:

- If OpenMP is available we append the OpenMP information to the dependencies, and we create new pre-processor flags to activate OpenMP code in both C and Fortran:

```
omp = dependency('openmp', required : false)
if omp.found ()
  all_deps = [all_deps, omp]
  add_project_arguments('-DOPENMP', language : 'c')
  add_project_arguments('-DOPENMP', language : 'fortran')
else
  message('OpenMP not found')
endif
```

- Using the the built-int `host_machine` data structure we check the OS, then create flag:

```
system = host_machine.system()
if system == 'linux'
  add_project_arguments('-DLINUX', language : 'c')
elif system == 'windows'
  add_project_arguments('-DWINDOWS', language : 'c')
elif system == 'darwin'
  add_project_arguments('-DOSX', language : 'c')
endif
```

User defined configuration options

To define custom options for the configuration stage requires to use an external file:

- For meson $<$ 1.1.10, the file name must be: **meson_options.txt**
- For meson $\geq$ 1.1.10, the file name must be: **meson.options**

To define an option use the **option()** command, and follow the syntax:

```
option('name', type: 'type', value: default_value, description: 'Blablabla')
```

Examples:

```
option('gtk', type: 'integer', value: 3, description: 'Version of the GTK library')
option('openmp', type: 'boolean', value: true, description: 'Use OpenMP')
```

More information is available on the build options page of the user manual.

Options declared in this file can then be used in the meson configuration line, and in the file **meson.build**.
In the file **meson.build** it is required to retrieve the value of the option using the **get_option()** instruction:

- To trigger build with different versions of the GTK library:

```
gtk_version = get_option('gtk')
if gtk_version == 3
  # Default option: checking for gtk+3.0:
  gtk = dependency('gtk+-3.0')
  add_project_arguments('-DGTK3', language : 'c')
else
  # Using -Dgtk=4 on the meson configuration command line
  gtk = dependency('gtk4')
  add_project_arguments('-DGTK4', language : 'c')
endif
```

- To check if OpenMP is available and adjust the compilation options accordingly:

```
use_openmp = get_option('openmp')
if use_openmp
  omp = dependency('openmp', required : false)
  if omp.found ()
    all_deps = [all_deps, omp]
    add_project_arguments('-DOPENMP', language : 'c')
    add_project_arguments('-DOPENMP', language : 'fortran')
  else
    message('OpenMP not found: building serial version')
  endif
endif
```

Declaring the project sources

In meson you declare sources in list of file(s), that your refer to using a variable name. This variable will contain the list of all files to be used to build the program. Furthermore in meson it is not possible to search for file(s) recursively, therefore all files to be used must be declared in the file **meson.build**:

```
src_c = ['src/main.c', 'src/gui.c']
src_f = ['src/file-1.f90', 'src/file-2.f90']
src_all = [src_c, src_f]
```

src_c is the name of a variable that will contain the list of files in C.

src_f is the name of a variable that will contain the list of files in Fortran.

src_all constructed using the previous variables is the list of all source files.

Note that the complete list of source file(s) must be provided in the file **meson.build**.

Declaring project headers include directories

In a meson project it is required to declare project headers (".h") include directory(ies) before building instructions. This is done using the **include_directories()** instruction followed by a list of coma separated file paths.
Save the result of these command in a variable that will be called when building the project:

```
inc_dir = include_directories('.', 'src')
```

Equivalent to:

```
all_inc = ['.', 'src']
inc_dir = include_directories(all_inc)
```

inc_dir contains the list of all file paths to search for include files.
Note once headers have been listed in the file **meson.build**, they can be included in the code between < and > symbols (no need of the file path):

```
#include <config.h>
```

Declaring the project building process

In meson you declare the project building process using the **executable()** instruction.
The **executable()** uses all information prepared up to this point: sources, include directories, dependencies, flags ...

```
executable(project_name,
           sources : src_all,
           include_directories : inc_dir,
           dependencies : all_deps,
           c_args :  cc_args,
           fortran_args:  fc_args,
           install : true)
```

The first argument of the command is the name of the binary to build, in this case the name of the project. It is followed by a coma separated list of elements, to provide information on the build process, by order of appearance in this example:

- The sources, using the keyword **source**
- The include directory(ies), using the keyword **include_directories**
- The dependencies, using the keyword **dependencies**
- User defined C compiler flags, using the keyword **c_args**
- User defined Fortran compiler flags, using the keyword **fortran_args**
- The installation option, using the keyword **install**

The syntax is similar in each case:

keyword : *value*

Note that variables created in the previous sections of the file **meson.build** are used to provide values for each field.
The last keyword in the **executable()** instruction, tells if the binary is to be installed or not:

install : *true / false*

It is possible to find multiple **executable()** instructions in the file **meson.build**.

Note that objects built using the **executable()** instruction are automatically installed in "**get_option('prefix')**/bin" if you need to install it somewhere else then add the "**install_dir** : *installation_path*" keyword to the **executable()** instruction:

```
package_libexec = get_option('libexecdir')
executable('alternate_bin', sources : alt_src,
           include_directories : alt_inc, dependencies : alt_deps,
           install : true, install_dir : package_libexec)
```

Installation using meson

Binary (or binaries) built using the file **meson.built** are automatically installed provided the "**install** : true" information appear in the **executable()** instruction.

To generate installation rules for additional data for your project use the **install_<FUNC>()** instructions:

- To install file, use the **install_data()** instruction:

```
install_dat('metadata/program-mime.xml',
            install_dir : data_dir / 'mime/packages')
install_dat('metadata/com.program.www.appdata.xml',
            install_dir : data_dir / 'metainfo')
install_dat('metadata/program.desktop',
             install_dir : data_dir / 'applications')
install_dat('ChangeLog', install_dir : data_dir / 'doc/prog')
install_dat('COPYING', install_dir : data_dir / 'doc/prog')
```

- To install a directory recursively, use the **install_subdir()** instruction:

```
install_subdir('data', install_dir : data_dir / 'prog')
install_subdir('pixmaps', install_dir : data_dir / 'prog')
```

Post-installation script

To use apply post-installation configuration, including system integration, an option is to use a script call by the **meson.add_install_script()** instruction:

```
# Post installation using a script, syntax:
# meson.add_install_script(script_name, agr_1, arg_2 ...)
# arguments for the script follow its name, and are separated by coma:
data_prefix = project_prefix / get_option('datadir')
meson.add_install_script('post_install.sh', data_prefix)
```

This will execute the command `./post-install.sh` which is a script containing the post-installation instructions. It is required to transmit to the script the installation directory selected by meson, this is done using the variable **project_prefix** in argument of the script. An example of content for the script `post-install.sh` is provided in appendix A.2.2.

A complete example of the file **meson.build** is provided in appendix A.3.1.

1.3.3 Configuring, building and installing a meson package

To configure, build and install a meson package use the **meson** command.
In the following I will consider that the file **meson.build** was completed:

```
user@localhost ~/program]$ ls -lh
-rw-r--r--. 1 user group 2,8K 24 mars  12:25 ChangeLog
-rw-r--r--. 1 user group 5,3K 24 mars  12:25 meson.build
-rw-r--r--. 1 user group 0,3K 24 mars  12:25 meson.options # Optional
-rw-r--r--. 1 user group  34K 24 mars  12:25 COPYING
drwxr-xr-x. 2 user group 4,0K 24 mars  12:25 data
drwxr-xr-x. 2 user group 4,0K 24 mars  12:25 metadata
drwxr-xr-x. 4 user group 4,0K 24 mars  12:25 pixmaps
-rw-r--r--. 1 user group 4,8K 24 mars  12:25 README
drwxr-xr-x. 2 user group 4,0K 24 mars  12:25 src
```

Configuration

To configure your meson project use the **meson** command:

- With the **setup** argument
- Followed by the name of a directory to be created

```
user@localhost ~/program]$ meson setup buildir
```

The last argument of the command is the new directory where the program will be built,
in this example case **buildir**.
The result of this command is both:

- The creation of several files and directories in **buildir**:

```
user@localhost ~/program]$ ls buildir
prog.p      compile_commands.json   meson-logs
build.ninja  meson-info             meson-private
```

- The configuration of your project that is already ready to be built using default options.

Other options can be added to the command line using the `-Doption` syntax, example:

```
user@localhost ~/program]$ meson setup buildir -Dgtk=4
```

Some options might be built-in but configuring with user defined options requires to create and prepare the file **meson.options** accordingly.

To list available options use:

```
user@localhost ~/program]$ meson configure buildir
```

Building

To build use the **meson** command with the **compile** argument:

- Followed by the option **-C** and the name of the build directory:

```
user@localhost ~/program]$ meson compile -C buildir
```

- Without option inside the build directory:

```
user@localhost ~/program]$ cd buildir
user@localhost ~/program/buildir]$ meson compile
```

Installing

To build use the **meson** command with the **install** argument:

- Followed by the option **-C** and the name of the build directory:

```
user@localhost ~/program]$ sudo meson install -C buildir
```

- Without option inside the build directory:

```
user@localhost ~/program]$ cd buildir
user@localhost ~/program/buildir]$ sudo meson install
```

1.3.4 Comparison with CMake and the GNU autotools

As for CMake it is not possible to easily uninstall a meson package, for example using:

```
user@localhost ~/program]$ sudo meson uninstall
```

In the meson case you will need to remove files manually.

However this does not really matter when the project is prepared to be distributed by the Linux software repositories in the RPM or the DEB format.

1.4 Conclusion

At this point you gained basic knowledge of the preparation of an automated installation package. This was the first prerequisite to be able take advantage of the package delivery system of the open source community to distribute your software.

CHAPTER 2

VERSION MANAGEMENT USING GIT

2.1 Introduction

Git is a distributed version control system that allows you to keep track of changes to your code over time. It is widely used among programmers collaboratively developing source code.

Git was originally developed by Linus Torvald in 2005 to improve the collaboration of coders working on the Linux kernel.

Basically Git offers a simple command line interface, several GUI frontends being available, that allows you keep track easily of the modification(s) on the project file(s) that you are monitoring.
The next sections will introduce a limited number of Git features to present the basics required to manage a project using Git.
For more complete documentation:

- The official Git documentation: https://git-scm.com/doc
- The reference manual: https://git-scm.com/docs
- The Pro Git Book: https://git-scm.com/book/en/v2

2.2 Installing Git

Before to start using Git, it is required to install it on your computer (if not done already). For Linux use your favorite software management utility:

- Red Hat based Linux (using the **dnf** command):

```
user@localhost ~]$ sudo dnf install git
```

- Debian based Linux (using the **apt** command):

```
user@localhost:~$ sudo apt install git
```

To learn more about software installation on Linux check out my Linux tutorial.
To learn more about scripting and the usage of the command line check out my Bash tutorial.

For Windows and Mac download the latest version of Git from the official Git website (https://git-scm.com/downloads) and follow the installation instructions provided on the website for your operating system.

2.3 Configuring Git

Once Git is installed, you need to configure it with your name and email address: Open a terminal and enter the following commands:

```
user@localhost ~]$ git config --global user.name "Your Name"
user@localhost ~]$ git config --global user.email your.email@host.eu
```

Replace **Your Name** and **your.email@host.eu** with your own.
The effect of the "--global" option is that **Your Name** and **your.email@host.eu** will be used as credentials by default for all your Git repositories.
However you might need to work on repositories where name and email could be different, in that case use the same command line without the "--global" option, but only after the initialization of the working repository (see bellow).

2.4 Creating a Git repository

To start using Git, you need to create a Git repository. A Git repository is a folder that contains your source code, Git will keep track of changes to your code within this folder. To create a Git repository, navigate to the folder where you want to store your code, and use:

```
user@localhost ~]$ cd program
user@localhost ~/program]$ git init
```

This will create a new Git repository in the current directory.
You can check that the repository has been created by listing the hidden files in the active folder:

```
user@localhost ~/program]$ ls -hla
drwxr-xr-x.  8 leroux dmo 4,0K 27 mars  21:58 .git
```

This **.git** folder will contain all the versioning information of your project.
If you erase this folder you erase all project history, also called index, and the only information that will remain consist of the current file(s) and folder(s) in the active directory.

2.5 Adding a branch to the project

Git projects can be seen as tree view, made with branches. The "**git init**" command creates a first branch called "master" to allow you to start working immediately. However you will likely need to create your own branches, in particular for web hosting purposes (see chapter 3) where main branches are usually called "Main" or "main".

To create a branch use:

```
user@localhost ~/program]$ git branch -m Main
```

To switch to another branch:

```
user@localhost ~/program]$ git checkout branch
```

Where the last option on the line, in this example "branch", is the name of the target branch to switch to.
You can also create a branch and immediately switch to that branch:

```
user@localhost ~/program]$ git checkout -b branch
```

The latest actually check if "branch" exists, if it does not create it, then switch to the target branch.
In the following the sentence "adding modification(s) to your Git repository" will be used extensively, it will be more accurate to write "adding modification(s) to target branch of your Git repository".

2.6 Adding modifications to a project branch

Now that you have a Git repository, and that you created a branch to work on, you can modify your Git branch, using:

```
user@localhost ~/program]$ git add this.file
```

If the file "this.file" is new then the entire content of the file will be added to the project repository, otherwise only potential modification(s) to this file will be recorded.

For example to add the file **README.md**, or modification(s) to this file, use:

```
user@localhost ~/program]$ git add README.md
```

To add all modification(s) to any file(s) and / or folder(s) in the active repository, use:

```
user@localhost ~/program]$ git add .
```

The **git add** command is used to add modification(s) to the project index, it updates the index using files found in the working directory tree:

- It is required to use the **add** command to append any newly created, modified or deleted file(s) / folder(s) to the index.
- This command can be performed several times before a commit.
- The **git status** command can used to obtain a summary of the files that have been changed and that staged for the next commit:

```
user@localhost ~]$ mkdir program
user@localhost ~]$ cd program
user@localhost ~/program]$ git init
astuce: Utilisation de 'master' comme nom de la branche initiale. Le nom de la branche
astuce: par défaut peut changer. Pour configurer le nom de la branche initiale
astuce: pour tous les nouveaux dépôts, et supprimer cet avertissement, lancez :
astuce:
astuce:  git config --global init.defaultBranch <nom>
astuce:
astuce: Les noms les plus utilisés à la place de 'master' sont 'main', 'trunk' et
astuce: 'development'. La branche nouvellement créée peut être rénommée avec :
astuce:
astuce:  git branch -m <nom>
Dépôt Git vide initialisé dans ~/program/.git/
user@localhost ~/program]$ cp ../README.md .
user@localhost ~/program]$ git add README.md
user@localhost ~/program]$ git status
Sur la branche master

Aucun commit

Modifications qui seront validées :
  (utilisez "git rm --cached <fichier>..." pour désindexer)
nouveau fichier : README.md
```

2.7 Committing changes

Now that you have added modification(s) to your Git repository, you can **commit** the changes. That means apply the changes to the repository and create a commit: a snapshot of the current state of your project/code history recorded by Git.
To commit changes, use the following command:

```
user@localhost ~/program]$ git commit -m "Commit message"
```

Replace `"Commit message"` with a brief description of the changes you made, ex:

```
user@localhost ~/program]$ now=`date`
user@localhost ~/program]$ version="1.0"
user@localhost ~/program]$ git commit -m  "Program [V-$version] $now"
```

2.8 Checking commit history

To view the commit history of a Git repository, use the following command:

```
user@localhost ~/program]$ git log
```

This will display a list of all the commits that have been made to the repository, along with their commit messages and other information.

2.9 Using Git to work on a remote, on-line, repository

The main interest of Git is to help you manage collaborative work(s) on remote, on-line, repositories, hosted on platforms like GitHub or GitLab. Furthermore it is almost mandatory to host your project on such platforms if you intent to distribute it using the pipelines of the open source community.
The next chapter will present how to host a project on these websites, then it will introduce the basic Git commands to use to manage such a project, see [Sec. 3.6] for details.

2.10 Conclusion

At this point you should have gained basic knowledge of Git a powerful tool for managing source codes essential to any developer. This was the second prerequisite to be able take advantage of the package delivery system of the open source community to distribute your software.

CHAPTER 3

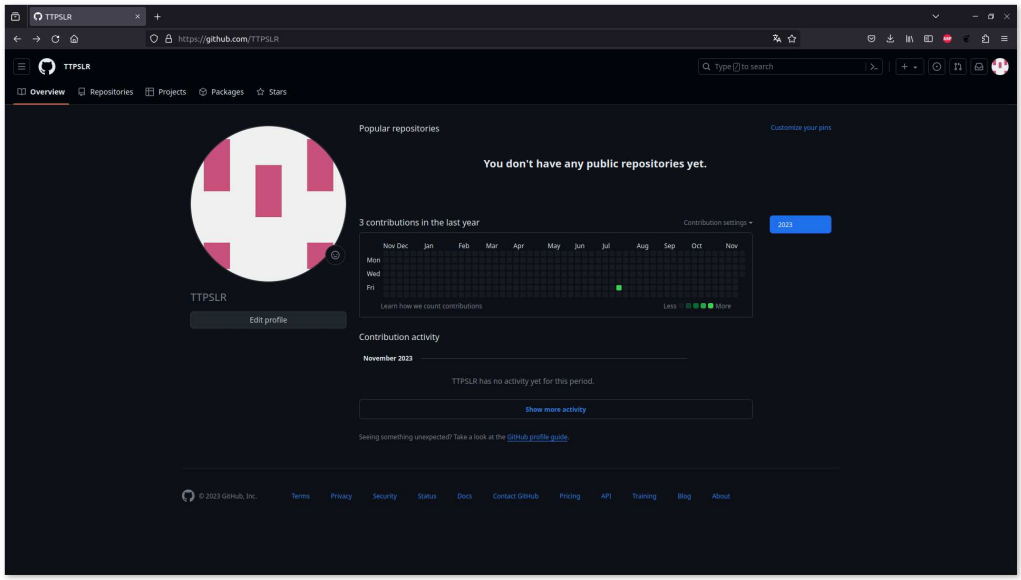
PROJECT HOST ON GITHUB / GITLAB

The previous chapter introduced Git a powerful tool for managing code changes.
Its real interest comes to light if you want your project to be collaborative, and/or if you want to publish your source code.
Indeed in that case you need to think about hosting your project, that is your Git repository on-line, or at least in a remote location where every contributor, member of your community, can access the source code.
The 2 most famous dedicated websites for that purpose are GitHub and GitLab.

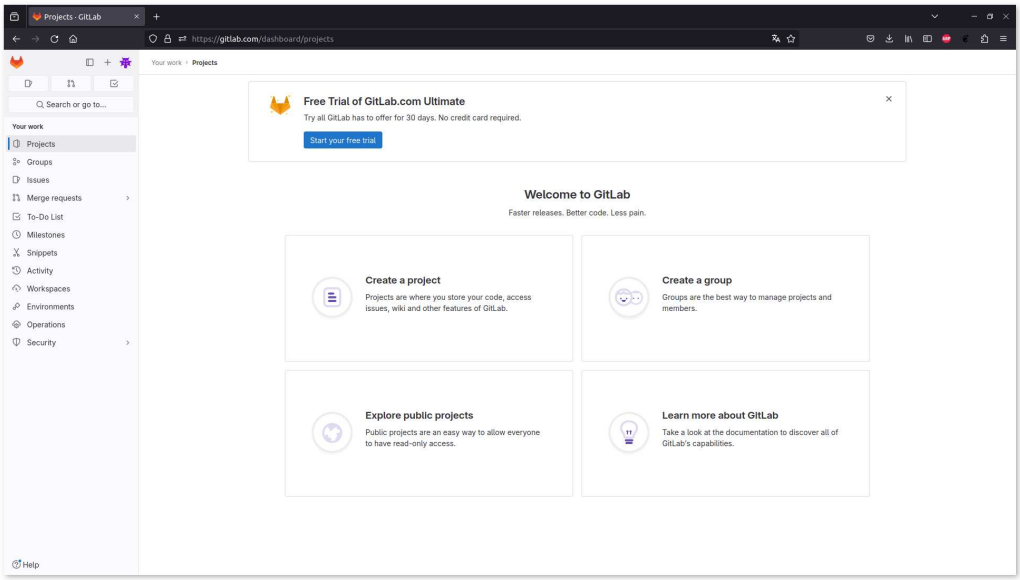
3.1 Introduction and user setup

The first step is to create a user account on the platform of you choice.
I will skip details on the registration process, however when the account is created you will see something that looks like:

- GitHub:



- GitLab:



In the following examples for both GitHub and GitLab:

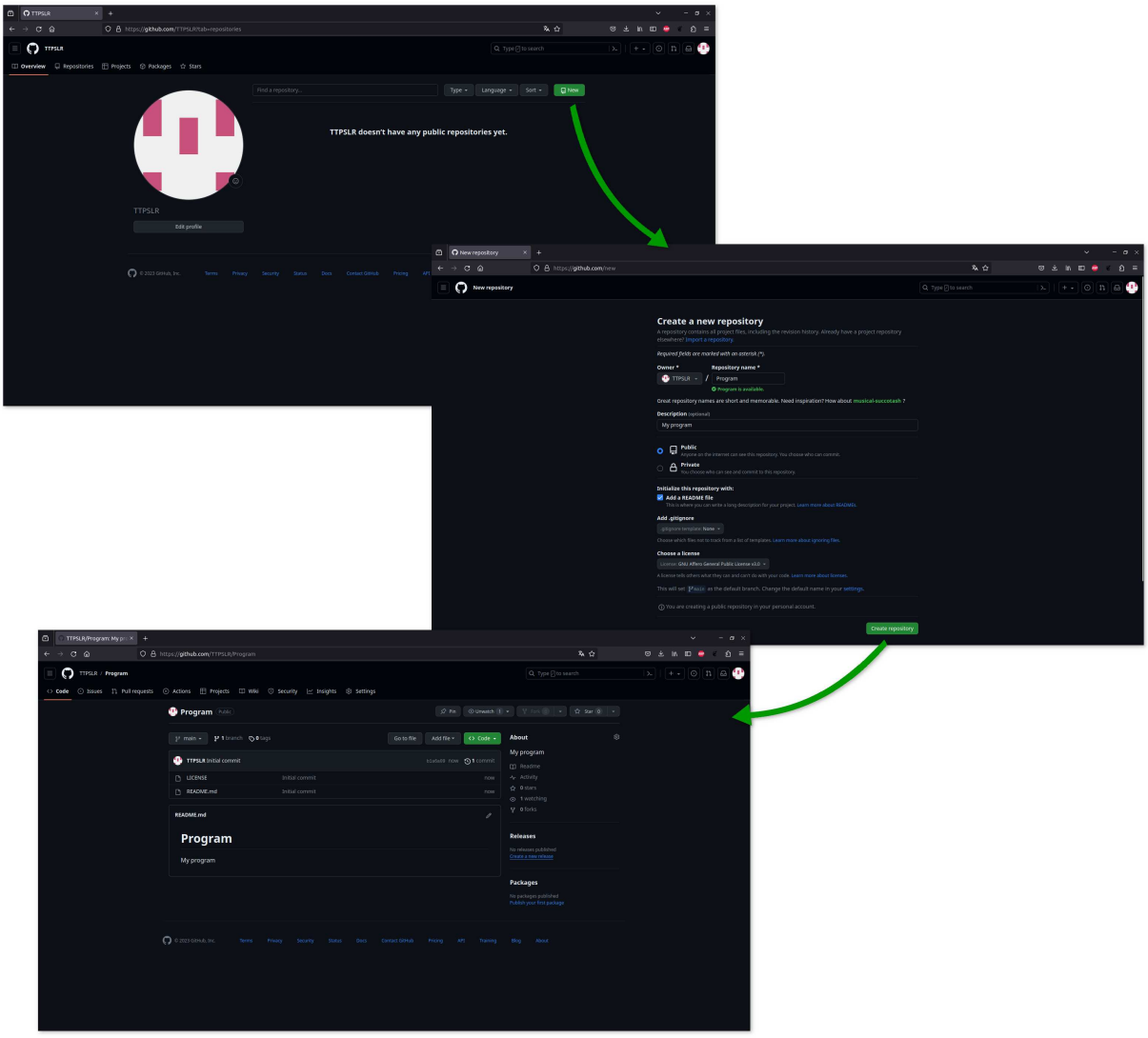
- the account/group name is: "`TTPSLR`"
- the project name is: "`Program`"

3.2 Creating a repository / project

The procedure is quite similar, only the denomination changes:

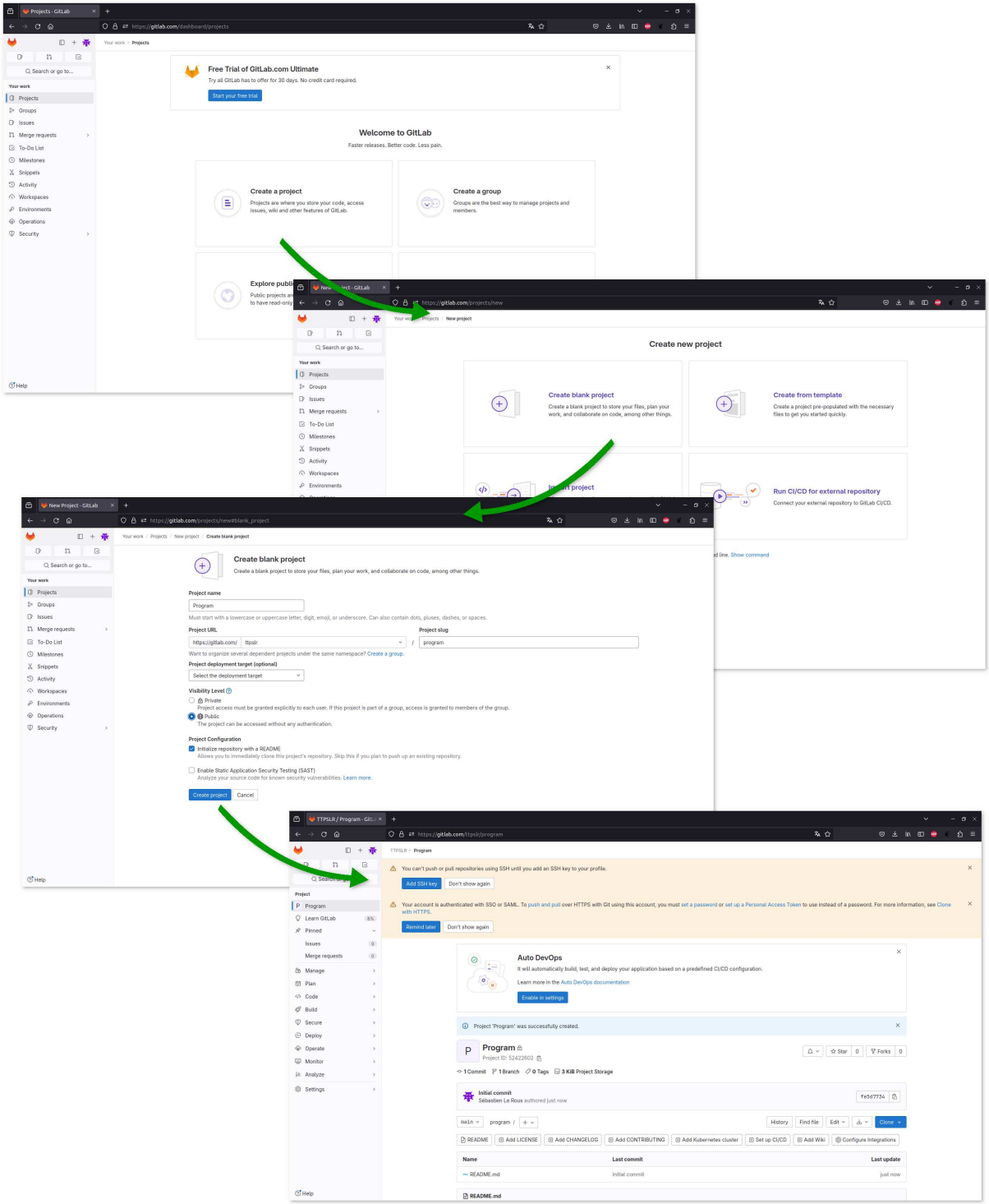
- On GitHub the space to store and share your sources is called a "`repository`"
- On GitLab the space to store and share your sources is called a "`project`"

- GitHub:



- Open the "`Repositories`" tab and click on "`New`"
- Fill the new project information
- Click on "`Create repository`"

- GitLab:


Free Trial of GitLab.com Ultimate
Welcome to GitLab
Create a project
Create a group
Create new project
Create blank project
Create from template
Run CI/CD for external repository
Project name
Project URL
Project slug
Visibility Level
Create project
Cancel
Auto DevOps
Program
Initial commit
README.md

- Click on "`Create project`"
- Click on "`Create blank project`"
- Fill the new project information
- Click on "`Create project`"

3.3 SSH encryption keys

Immediately after creating your first project, setup the SSH encryption keys. These keys are digital tokens that prove your identity when performing actions to update the on line repository using Git.

3.3.1 Generating SSH encryption keys

If not done already you need first to create SSH encryption keys, this is done using a command line utility:

```
user@localhost ~]$ ssh-keygen -t ed25519
```

"**`ssh-keygen`**" generates the key, the "**`-t`**" option is used to define the type of encoding, in the example "**`ed25519`**" that stands for Edwards-curve Digital Signature Algorithm which is the most secured standardization these days.
2 files, to be stored in "`~/.ssh`", will be created by the previous command:

- A private key, that decipher (act like a key), and that must remains on your computer:

  ```
  ~/.ssh/id_ed25519
  ```

- A matching public key, that encrypts (acts like a door), used on remote system(s):

  ```
  ~/.ssh/id_ed25519.pub
  ```

  Which content looks like:

  ```
  user@localhost ~]$ cat ~/.ssh/id_ed25519.pub
  ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIG1TIYyaRZIlFU40NH8QAxXK8SSgl07Thop6CGzcVg0j user@host
  ```

For more about asymmetric keys algorithms: https://en.wikipedia.org/wiki/Public-key_cryptography

3.3.2 Adding the keys to GitHub / GitLab

Now you need to add the public key to your GitHub / GitLab repository:

- GitHub (see figure 3.1):
  - Click on the profile logo and click on "`Settings`"
  - Click on the "`SSH and GPG keys`"
  - Click on "`Create repository`"
  - Click on "`New SSH key`"
  - Adjust the key information:
    - Give the key a title name
    - Select the key type: "`Authentication key`"
    - Copy / paste the content of public key file in the text box
  - Finally click on "`Add SSH key`"
- GitLab (see figure 3.2):
  - Click on shortcut button "`Add SSH key`"
    Alternatively click on your public avatar (squared in red in figure 3.2),
    click on "`Preferences`", then click on "`SSH Keys`"
  - Click on the "`Add new key`"
  - Adjust the key information:
    - Copy / paste the content of public key file in the text box
    - Give the key a title name
    - Select the key type: "`Authentication`" or "`Authentication & Signing`"
  - Finally click on "`Add key`"
  - Optionally go back the "`User Settings` $\Longrightarrow$ `SSH Keys`" tab to visualize that the key is properly stored

As soon as the keys have been installed you are ready to work with your on line repository.

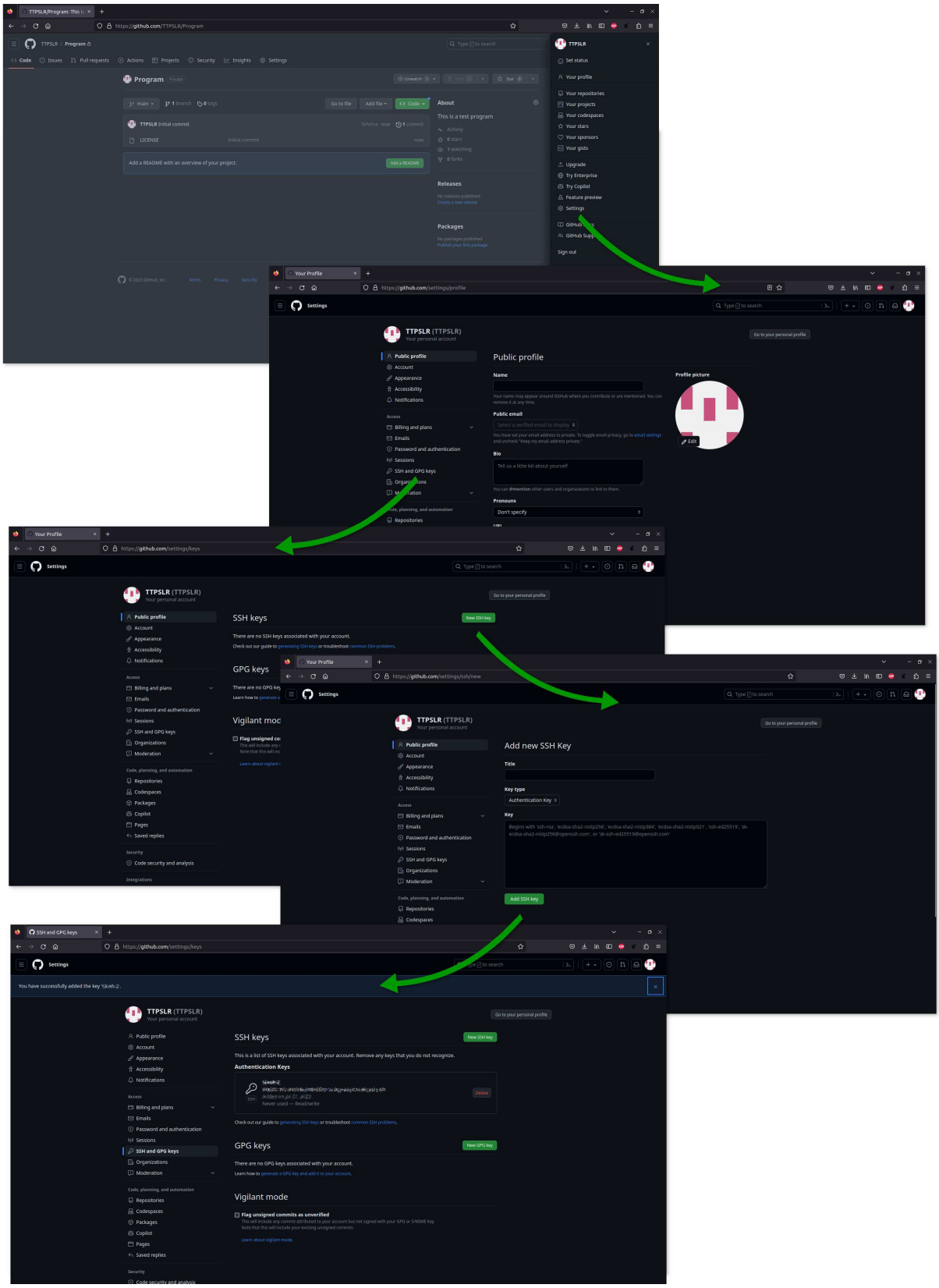


Figure 3.1: Adding a SSH key on GitHub

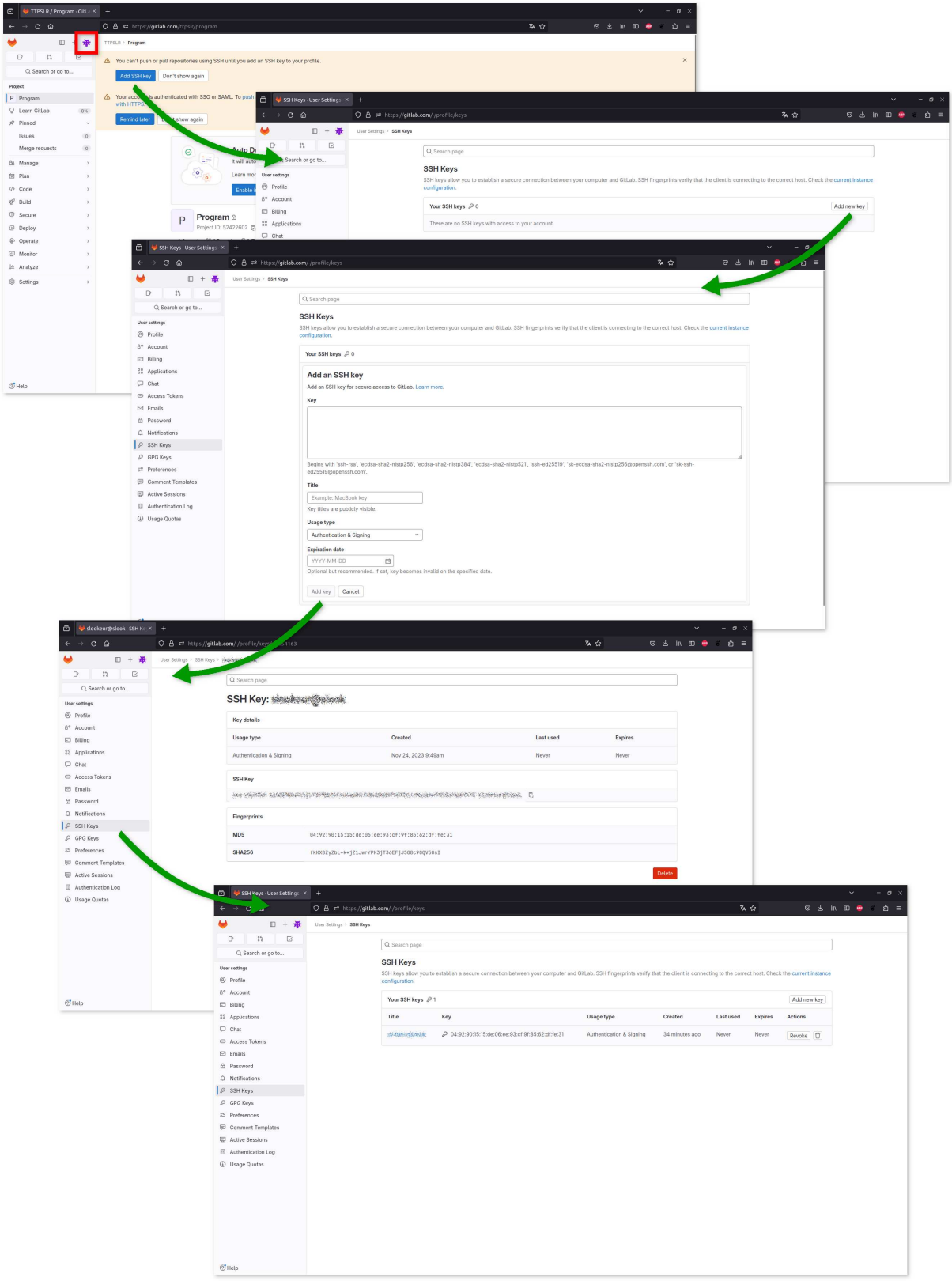


Figure 3.2: Adding a SSH key on GitLab

3.4 Branch protection

If you are managing collaborative project, with several developers being able to push commits to the Git (and GitHub or GitLab) repository branch(es), then you need to ensure that branch(es) of your project is (are) being protected against errors, ensuring that changes are handled by the project manager, or designated and selected developers.
To do that you need to enable, and adjust, branch(es) protection:

- GitHub (see figure 3.3):
  - Click on "`Settings`"
  - Click on "`Branches`"
  - Click on "`Add branch protection rule`"
  - Choose branch name and adjust the protection level for this target branch of your project
  - Scroll down and click on "`Create`"

  Note that on GitHub no branch protection is in place at the beginning to this step is really important.
- GitLab:
  - Default main branch protection for all your projects (see figure 3.4):
    - Click on "`Settings`" $\Longrightarrow$ "`Repository`"
    - In front of "`Default branch`" click on "`Expand`"
    - Ensure that "`Fully protected`"
  - Other branches protection (see figure 3.5):
    - Click on "`Settings`" $\Longrightarrow$ "`Repository`"
    - In front of "`Protected branches`" click on "`Expand`"
    - Adjust the protection level for the target branch of your project

  On GitLab however branch protection should already be in place for the main branch.

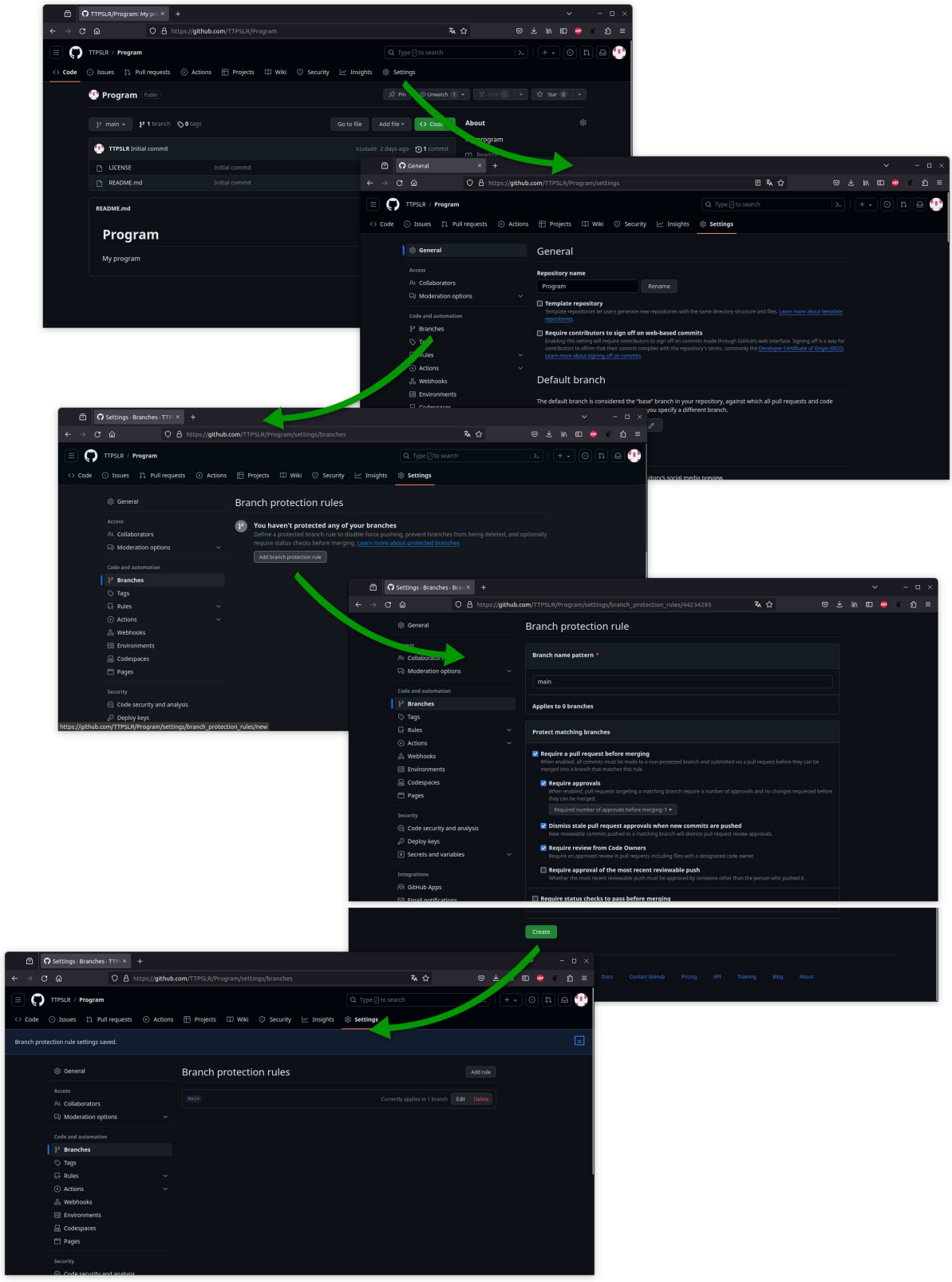


Figure 3.3: Project branch projection on GitHub

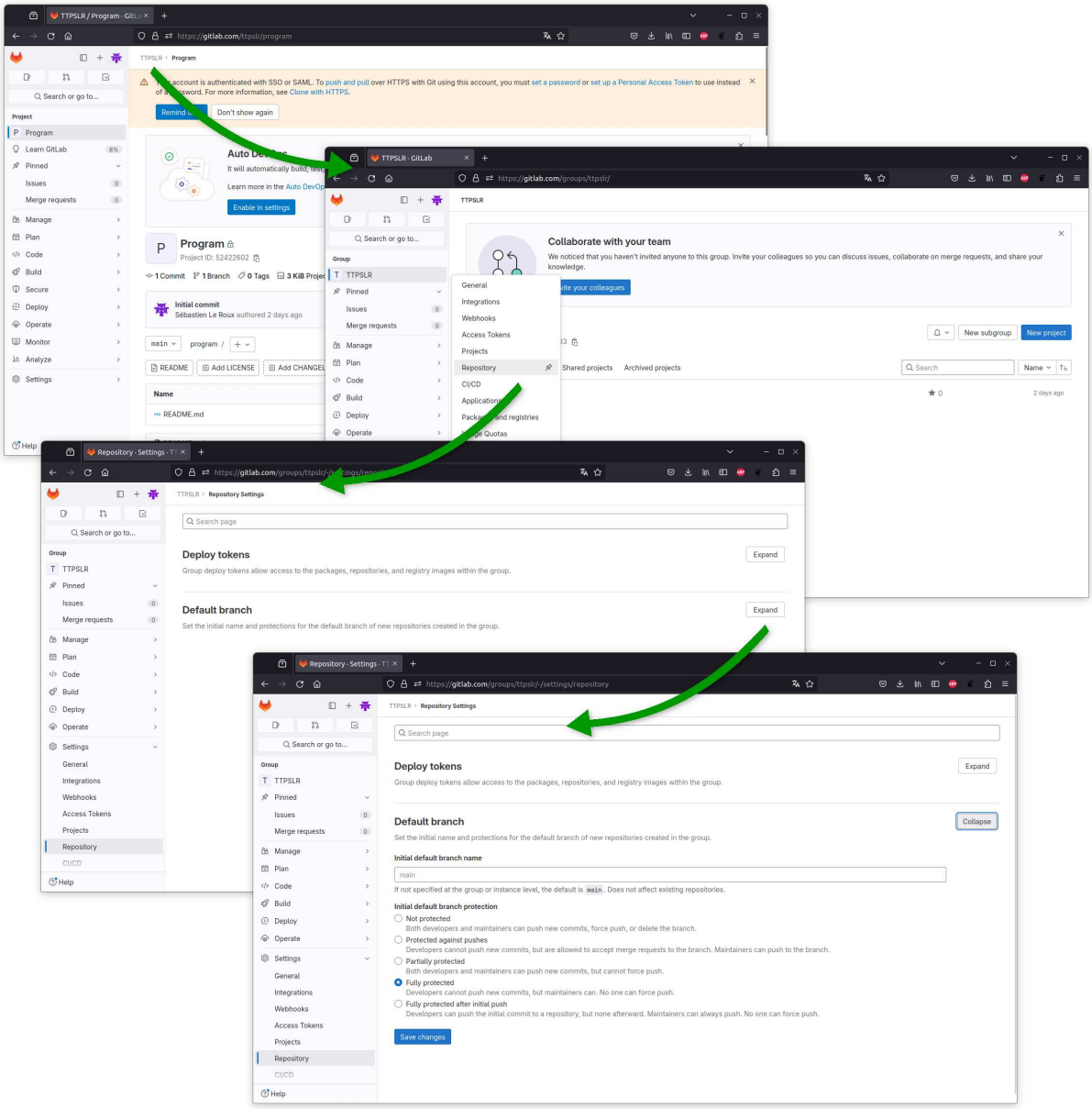


Figure 3.4: Default main branch projection on GitLab

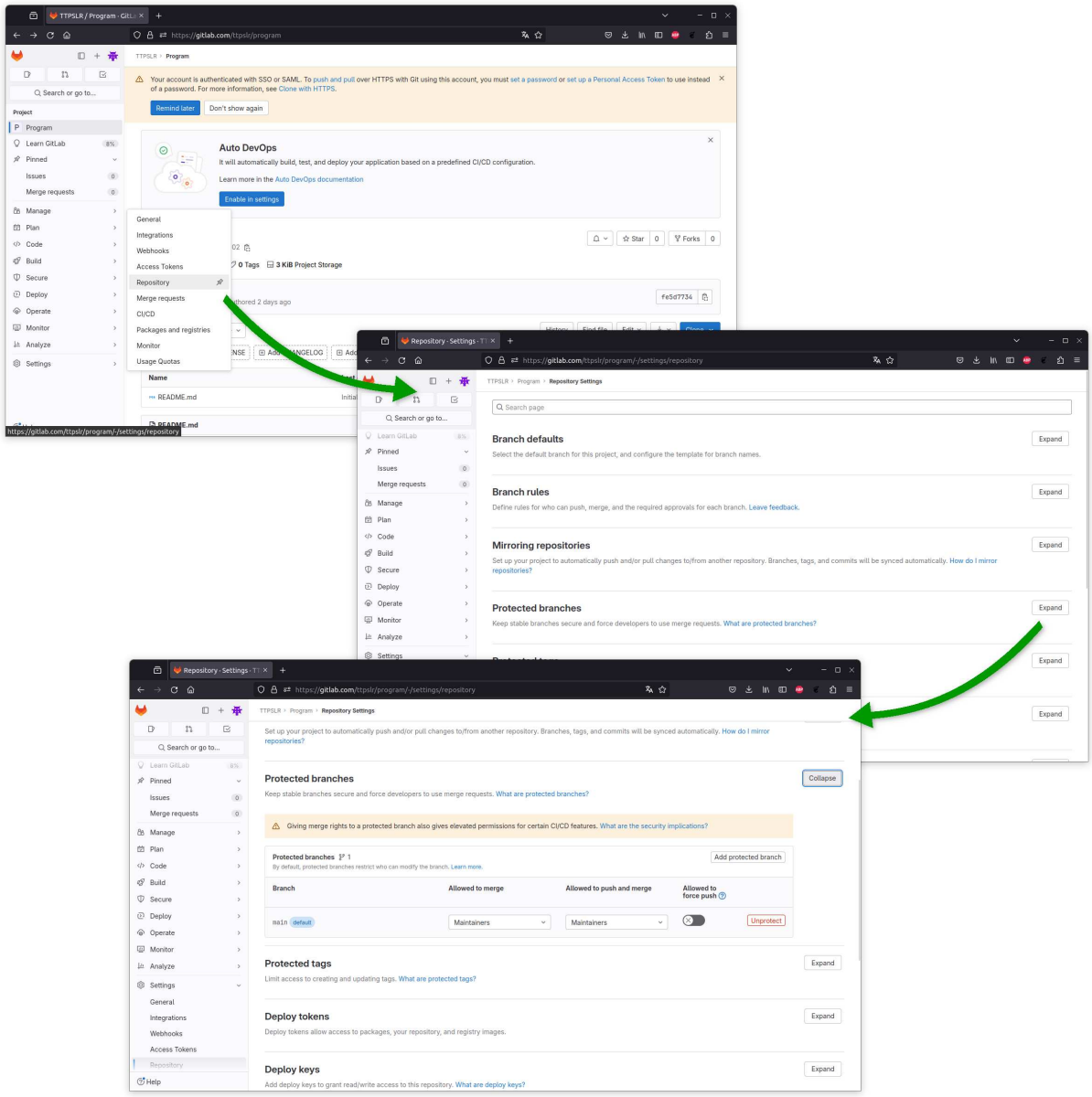


Figure 3.5: Project branch projection on GitLab

3.5 Releases and Tags

3.5.1 Tags

On GitHub and GitLab tags are images of the Git repository pointing towards archives that never change, like snapshots of the repository.
In both GitHub and GitLab you can create tags whenever you think it is appropriate, however tags being required to produce a release, and a specific tag being associated with a specific release I will only focus thereafter in creating tag whenever creating release is required.

3.5.2 Releases

On GitHub and GitLab releases are deployable software versions made available for users to download. Releases are based on tags targeting specific points in the history of the repository.
To create a release:

- On GitHub (see figure 3.6):
  - Click on "`Create a new release`"
  - Click on "**`Choose a tag`**" to open the corresponding combo box:
    - Input the new tag name, ex: "`1.0.0`"
    - If the tag is new it is immediately proposed to create this tag
    - Click on "**`Create new tag: 1.0.0`** `on publish`"
      (or any other tag name of your choice)
  - Fill the description of the release, including release notes
  - Scroll down and click on "`Publish release`"

  On GitHub releases packages have the form: "**`tag.tar.gz`**"
  Extracting the archive on your hard drive will create a folder named: "**`repository-tag`**"

- On GitLab (see figure 3.7):
  - On the left side menu, click on "`Deploy`" $\Longrightarrow$ "`Releases`"
  - Then click on "`Create a new release`"
  - Click on "**`Tag name`** `(required)`" to open the corresponding combo box:
    - Input the new tag name, ex: "`1.0.0`"
    - If the tag is new it is immediately proposed to create this tag
    - Click on "`Create tag` **`1.0.0`**" (or any other tag name of your choice)
    - Click on "`Save`"
  - Fill the description of the release, including release notes
  - Scroll down and click on "`Create release`"

  On GitLab releases packages have the form: "**`project-tag.tar.gz`**"
  Extracting the archive on your hard drive will create a folder named: "**`project-tag`**"

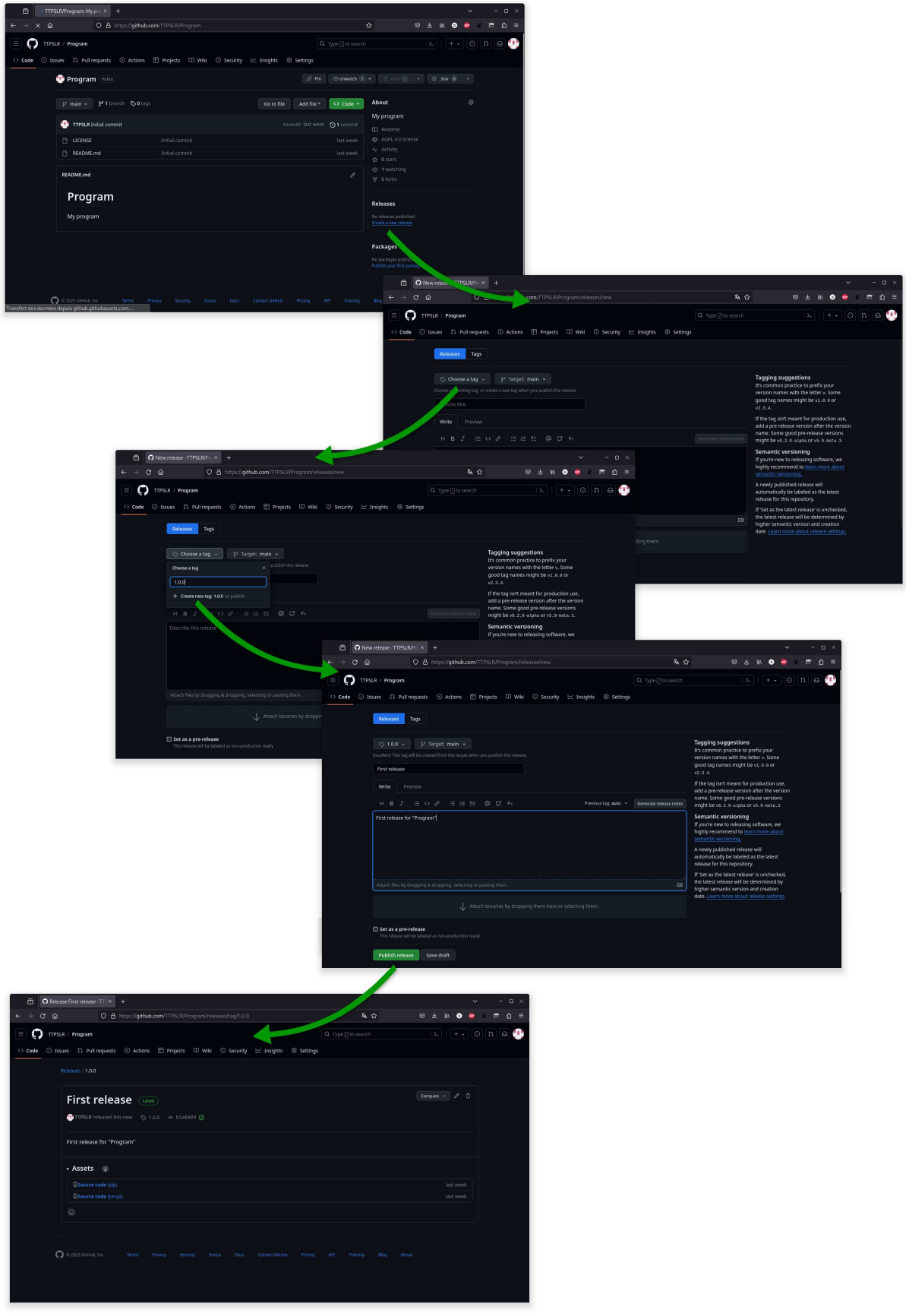


Figure 3.6: Creating a release on GitHub

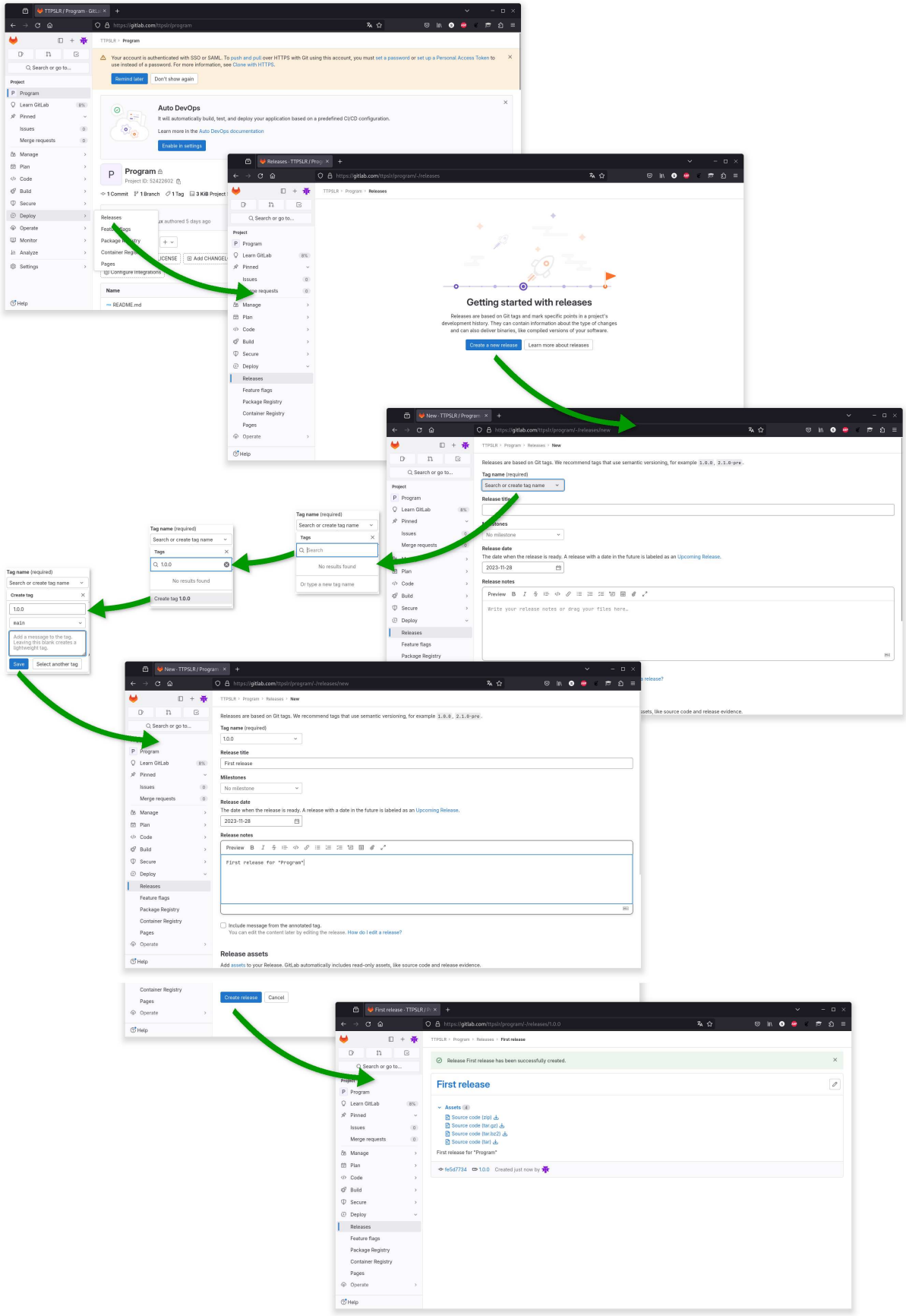


Figure 3.7: Creating a release on GitLab

3.6 Using Git to manage your GitHub / GitLab project

3.6.1 Setting up work on you local computer

To start working on a remote repository using Git you can:

- **Clone** the distant repository:
  - GitHub:

```
user@localhost ~/program]$ git clone git@github.com:Author/Program
```

  - GitLab:

```
user@localhost ~/program]$ git clone git@gitlab.com:Group/Program
```

    Be careful **NOT** to clone the distant repository using the "`https://git***`" instruction (both options being offered by GitHub and GitLab), this would modify the access to the repository via the Git command line and require additional security considerations to setup developer access tokens, not considered in this manual.

- Alternatively set up manually the local folder to work remotely:

```
user@localhost ~]$ mkdir program
user@localhost ~]$ cd program
user@localhost ~/program]$ git init
```

  - GitHub:

```
user@localhost ~/program]$ git remote add origin git@github.com:Author/Program
```

    Replace:
    * **`Name`** by the GitHub account that owns the repository.
    * **`Program`** by the name of the repository.

  - GitLab:

```
user@localhost ~/program]$ git remote add origin git@gitlab.com:Group/Program
```

    Replace:
    * **`Group`** by the group Id for the GitLab account that owns the project.
    * **`Program`** by the name of the project.

This should be enough to get you started, providing that you check that the git user name and email are set properly (see [Sec. 2.3]).

Finally if needed setup the user information:

```
user@localhost ~]$ git config user.name "Your Name"
user@localhost ~]$ git config user.email your.email@host.eu
```

Replace:

- **Your Name** by your user name.
- **your.email@host.eu** by your email.

Update the local folder with the content of the remote repository:

```
user@localhost ~/program]$ git pull origin branch
```

Replace:

- **branch** by the branch of the project your are working on.

When this is done you can verify the link between your local folder and the on-line repository using:

```
user@localhost ~/program]$ git remote -v
origin git@github.com:Author/Program (fetch)
origin git@github.com:Author/Program (push)
```

Where:

- **Author** is the GitHub account that owns the repository.
- **Program** is the name of the repository.

3.6.2 Contributing to other project(s) and collaborative work

If you want to work on project hosted on GitHub or GitLab, and that you are not managing the project you intend to work on, then you need first to **fork** this project. That means to create your own personnal copy of this project, independent of the original repository.
Afterwards to submit your modification(s) to the main repository you will need to create a dedicated branch and request to merge your branch to the main project. Modification(s) will be checked by a project manager, and if appropriate will be merged to the main branch of the repository.

To fork an exisiting project:

- On GitHub (see figure 3.8):
  - Navigate to the GitHub project page of the repository you want to fork
  - Click on "`Fork`" to open the scrolling menu, and select "`Create a new fork`"
  - Follow the dialog to create a fork on the project in your own repository, you can select a specific branch if required, then when ready click on "`Create fork`"
  - It's done your own fork of the target project is now ready for you to work on in your own GitHub repository.
- On GitLab (see figure 3.9):
  - Navigate to the GitLab project page of the repository you want to fork
  - Click on "`Fork`"
  - Follow the dialog to create a fork on the project in your own repository, remember to specify the project URL. You can select a specific branch if required, then when ready click on "`Fork project`"
  - It's done your own fork of the target project is now ready for you to work on in your own GitLab repository.

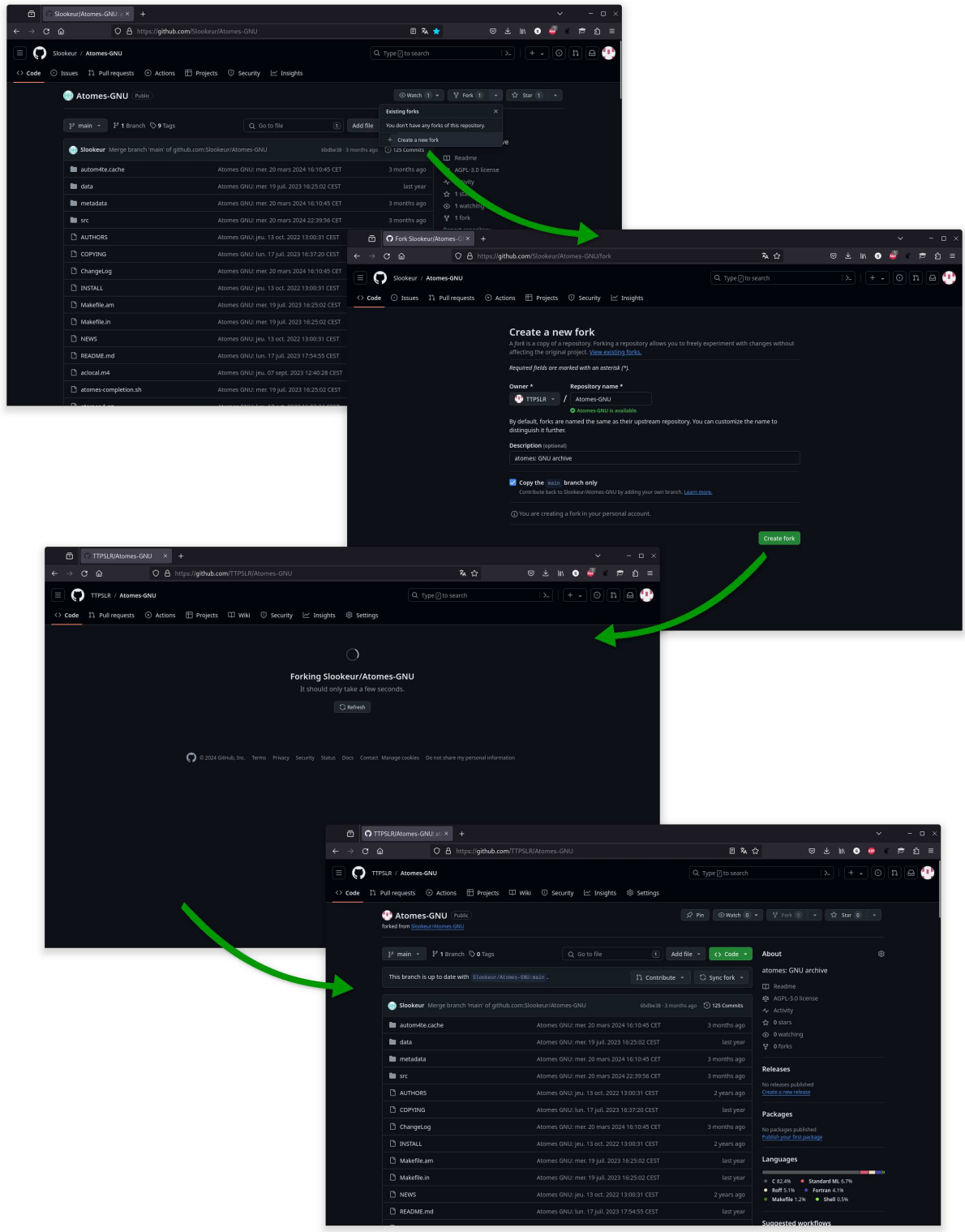


Figure 3.8: Creating a fork of a repository on GitHub

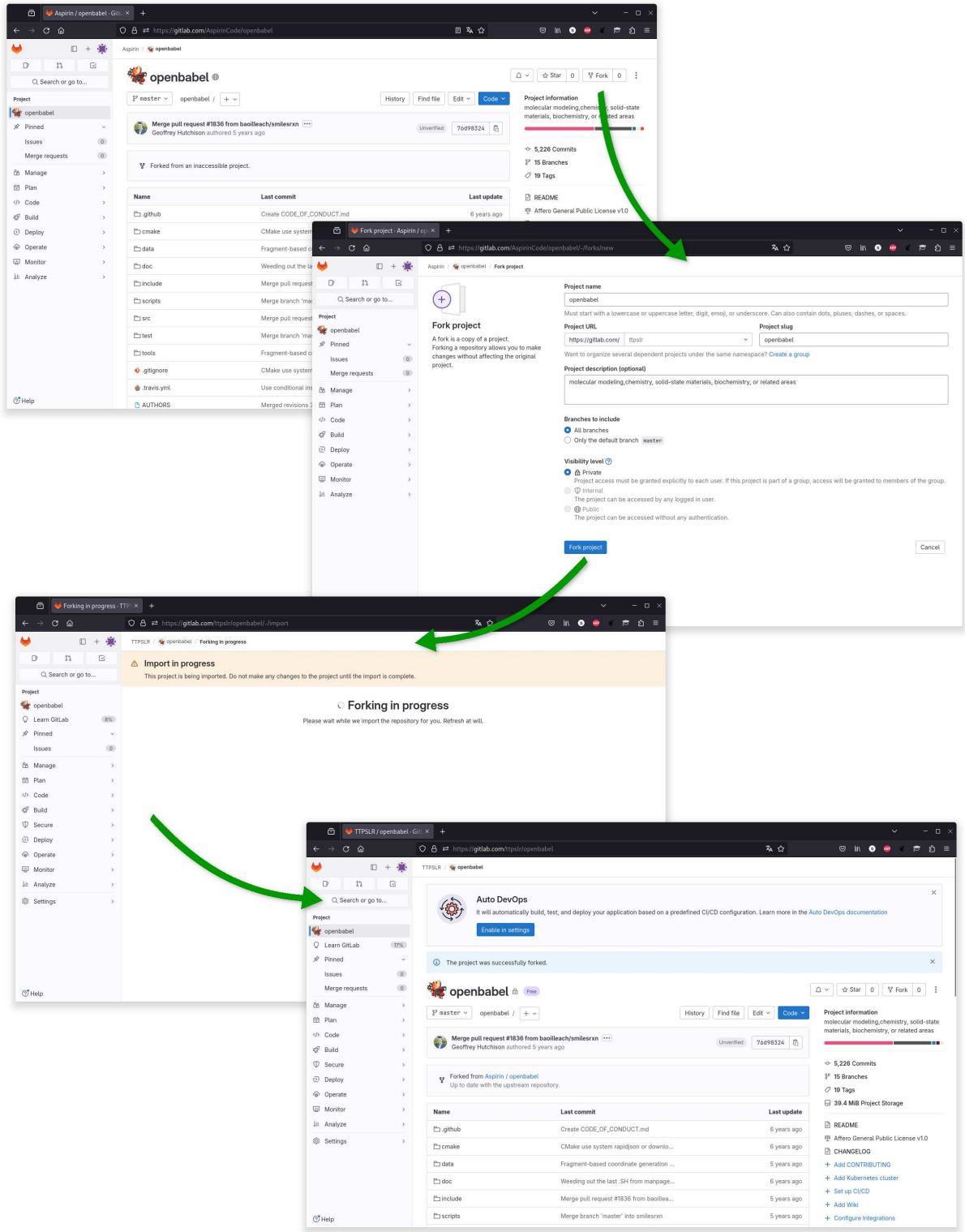


Figure 3.9: Creating a fork of a repository on GitLab

3.7 Project pages and documentation

This section will illustrates how to use a GitHub / GitLab repository and web pages to host a static web documentation for your project.

3.7.1 prerequisites

It is required to install some tools to handle the publication of static webpages that will be used thereafter:

- Prepare the "`~/.bashrc`" file to install Ruby, insert the following lines:

```
user@localhost ~]$ vi .bashrc
# Install Ruby Gems to ~/gems
export GEM_HOME="$HOME/gems"
export PATH="$HOME/gems/bin:$PATH"
export PATH="$HOME/.rbenv/bin:$PATH"
eval "$(rbenv init -)"
export PATH="$HOME/.rbenv/plugins/ruby-build/bin:$PATH"
```

- Install the Ruby dependencies (if needed)
  - Fedora:

```
~]$ sudo dnf install git-core gcc rust patch make bzip2 openssl-devel \
                     libyaml-devel libffi-devel readline-devel zlib-devel \
                     gdbm-devel ncurses-devel perl-FindBin perl-lib \
                     perl-File-Compare
```

  - Debian:

```
~]$ sudo apt install postgresql libpq-dev nodejs yarnpkg git zlib1g-dev \
                     build-essential libssl-dev libreadline-dev libyaml-dev \
                     libsqlite3-dev sqlite3 libxml2-dev libxslt1-dev \
                     libcurl4-openssl-dev software-properties-common \
                     libffi-dev
```

- Install "`rbenv`":

```
~]$ git clone https://github.com/rbenv/rbenv.git ~/.rbenv
~]$ git clone https://github.com/rbenv/ruby-build.git ~/.rbenv/plugins/ruby-build
~]$ rbenv install 2.7.4
~]$ rbenv global 2.7.4
~]$ ruby -v
```

GitHub pages are best compatible with version 2.7.x so do not install more recent release.

To list available stable versions:

```
~]$ rbenv install -l
```

- Update "rubygem":

```
~]$ gem update --system
```

- Install Jekyll and the Ruby bundler:

```
user@localhost ~]$ gem install bundler jekyll
```

From this point forward the following steps are required to publish your documentation on GitHub or GitLab pages:

1. Build the documentation of your project in Markdown and/or HTML language
2. Use Jekyll to convert your documentation in a static website
3. Create a repository on GitHub or GitLab to host the documentation
4. Push your documentation to the web pages of the associated GitHub or GitLab repository.

3.7.2 Building the documentation in Markdown or HTML language

It is up to you to decide how to do this, a good idea is to write your documentation in LaTeX format, so that you can produce clean PDF documents. Then convert the LaTeX files to HTML using pandoc.
Also it allows to use your bibliography in BibTeX format and makes it easy to handle references on web pages.

See the next section to know more about the data structure to prepare.

Few things to take care of to use LaTeX and pandoc to prepare your documentation:

- To insert figures use PNG, or other graphic format, and not EPS as in standard LaTeX documents.
- Be careful with the location of the images, ensure that the location, ideally in a separate and dedicated folder, matches the link in the HTML page.

- Internal references for objects not on the same HTML page will be lost when rendering the website.
  This means that if your refer to a figure from the first chapter in the second chapter, likely on separate pages, you will have to correct the internal link to the proper web page.

- Pandoc conversion is not perfectly clean so few errors will likely require to be corrected afterwards, the best way to do that, is to understand the issue and to automate the correction process.
  Among known errors to take care of:

  - Check the table and figure captions
  - Some LaTeX commands, in particular from peculiar custom packages, might not be understood by pandoc, and should be test proofed.

    For example the "`\Ctrl`" command from the "`keystroke`" package (to render a drawing of the keyboard control key) cannot be processed by pandoc and should be replace by a command having the proper effect in HTML format, in this case "`\newcommand{\Ctrl}{<kbd>Ctrl<kbd>}`"

- Finally to render LaTeX math and equations in your HTML page insert the following code at the top of the HTML page:

  - To render math using mathjax use:

```
<script src="https://polyfill.io/v3/polyfill.min.js?features=es6"></script>
<script id="MathJax-script" async src="https://cdn.jsdelivr.net/npm/mathjax@3.0.1/es5/tex-mml-chtml.js"></script>
```

  - To render math using katex use:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/KaTeX/0.11.1/katex.min.js"></script>
<script>document.addEventListener("DOMContentLoaded", function ()
   var mathElements = document.getElementsByClassName("math");
   var macros = [];
   for (var i = 0; i < mathElements.length; i++)
      var texText = mathElements[i].firstChild;
      if (mathElements[i].tagName == "SPAN")
       katex.render(texText.data, mathElements[i],
        displayMode: mathElements[i].classList.contains('display'),
        throwOnError: false,
        macros: macros,
        fleqn: false
        );
      );
</script>
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/KaTeX/0.11.1/katex.min.css" />
<!--[if lt IE 9]>
  <script src="//cdnjs.cloudflare.com/ajax/libs/html5shiv/3.7.3/html5shiv-printshiv.min.js"></script>
<![endif]-->
```

Overall this method requires some additional work to properly convert the documents to a clean simple combination of Markdown and HTML structure, but once this work is automated you will only write your documentation using LaTeX and then, in a single command, convert it and publish it to GitHub or GitLab pages.

3.7.3 Using Jekyll to build a static website

Jekyll is a simple static site generator.
Think of it like a file-based CMS, without all the complexity. Jekyll takes your content, renders Markdown and Liquid templates, and creates a complete, static website ready to be served by any web server.
Jekyll is the engine behind GitHub Pages, which you can use to host sites right from your GitHub repositories.

Jekyll expects a document structure with Markdown files, but can also handles HTML files as includes files. In the following I will illustrate the document structure required by Jekyll to build a static web site:

- For LaTeX converted to HTML documentation
- Using bibliography references in the BibTeX format

Example repository to create the documentation:

```
user@localhost ~/Program-doc]$ ls
_bibliography  chap-1  intro  _config.yml  Gemfile  _includes  README.md
```

With:

- Files:
  - The home page of your documentation website: "README.md"
  - The list of Ruby extensions required to build your website: "Gemfile"

```
user@localhost ~/Program-doc]$ vi Gemfile
source "https://rubygems.org"

gem 'jekyll-rtd-theme'
gem 'github-pages', group: :jekyll_plugins
gem 'jekyll-scholar', group: :jekyll_plugins
group :jekyll_plugins do
  gem "jekyll", "~> 3.9.0"
  gem "jekyll-feed", "~> 0.12"
  gem "jekyll-paginate"
  gem "jekyll-seo-tag"
  gem "jekyll-sitemap"
  gem "jekyll-archives"
  gem "jekyll-redirect-from"
end

# Windows and JRuby does not include zoneinfo files,
# so bundle the tzinfo-data gem and associated library.
platforms :mingw, :x64_mingw, :mswin, :jruby do
  gem "tzinfo", "~> 1.2"
  gem "tzinfo-data"
end

#  Performance-booster for watching directories on Windows
gem "wdm", "~> 0.1.1", :platforms => [:mingw, :x64_mingw, :mswin]
```

– The Jekkyl configuration file to build the website: "`_config.yml`"

```
user@localhost ~/Program-doc]$ vi _config.yml
title: Program's documentation
baseurl: "/Program-doc"
url: ""
# GitHub or GtiLab repository:
repository: TTPSLR/Program-doc
lang: en
description: This site presents the documentation for "Program"
markdown: kramdown
highlighter: rouge
gist:
  noscript: false
kramdown:
  math_engine: mathjax
  syntax_highlighter: rouge
markdown_extensions:
  - toc:
    permalink: "#"
# Define the depth of the menu
    baselevel: 2
banner: "/assets/img/banner.png"
favicon: "/assets/img/favicon.ico"
remote_theme: rundocs/jekyll-rtd-theme
readme_index:
  with_frontmatter: true

plugins:
  - jekyll-coffeescript
  - jekyll-default-layout
  - jekyll-gist
  - jekyll-github-metadata
  - jekyll-optional-front-matter
  - jekyll-paginate
  - jekyll-readme-index
  - jekyll-titles-from-headings
  - jekyll-relative-links
  - jekyll-scholar
  - jekyll-feed
  - jekyll-paginate
  - jekyll-seo-tag
  - jekyll-sitemap
  - jekyll-archives
  - jekyll-redirect-from
  - jekyll-remote-theme

scholar:
  locale: en
  source: ./_bibliography
  style: _bibliography/my-ieee.cls
  bibliography: references.bib
  bibliography_template: ""
  replace_strings: true
  join_strings:    true
  use_raw_bibtex_entry: false
  details_dir:    bibliography
  details_layout: bibtex.html
  details_link:   Details
# Ensure that details are not printed twice
  query: "@*"

exclude:
  - CNAME
  - Gemfile
  - Gemfile.lock
```

- Directories:
  - Each chapter or section in a separate directory, in this case "**intro**" and "**chap-1**":

```
user@localhost ~/program-doc]$ ls intro
README.md

user@localhost ~/program-doc]$ ls -R chap-1

chap-1:
README.md  section-1  section-2

chap-1/section-1:
README.md

chap-1/section-2:
README.md
```

    Including sub-directories for chapter sections if required.
    Note that each directory contains a single "README.md" file, dedicated to a single page of your site.

  - Include files in the "**_includes**" directory:

```
user@localhost ~/program-doc]$ ls -R _includes
.:
intro  chap-1

./intro:
intro.html

./chap-1:
chap-1.html  section-1  section-2

./chap-1/section-1:
section-1.html

./chap-1/section-2:
section-2.html
user@localhost ~/program-doc]$
```

    including sub-directories for appendix chapters and sections if required.

- The bibliography in the "**_bibliography**" directory:

```
user@localhost ~/program-doc]$ ls _bibliography
my-ieee.cls  references.bib
```

With:

* The "*.ieee.cls" file contains the bibliography style to apply
* The "*.bib" file contains references in BibTeX format
  When you convert LaTeX to HTML the keywords for the bibliography remain in the HTML document.

The "README.md" files, in Markdown format, follow the structure:

```
user@localhost ~/program-doc]$ vi chap-1/README.md
<!-- This is a HTML comment in Markdown -->
<!-- It will show up in the converted HTML document -->

---
<!-- Position of this file in the website menu. -->
<!-- To be compared will all other README.md file(s) at the same level,-->
<!-- in this case all "~/program-doc/*/README.md" files -->
sort: 2
<!-- Creation date: -->
date: 2023-11-26
<!-- Math formula on this pages ? -->
<!-- This is used to enable math javascript: -->
maths: 1
---

<!-- # is the Markdown command for a heading -->
<!-- Therefore the next line is the title of the chapter -->
# How to use "Program"
<!-- Note if you do that you need to remove the heading line -->
<!-- from the included HTML file bellow: -->

<!-- Jekyll instruction to read an include file,-->
<!-- and search for corresponding file(s) in the "_include" folder -->
{% include chap-1/chap-1.html %}

<!-- Jekyll instruction to process all sub-directories, if any: -->
{% include list.liquid all=true %}

<!-- Jekyll instruction to add a bibliography section on this page: -->
{% bibliography --cited %}
<!-- Then Jekyll search the "_bibliography" folder to insert references -->
```

Jekyll will search for all "README.md" files in the directory tree, starting from the top directory. and build the website according to the "sort" instructions, in this example:

- Top level, homepage: "README.md"
  - First level:
    - In "intro\README.md" $\Longrightarrow$ "sort: 1"
    - In "chap-1\README.md" $\Longrightarrow$ "sort: 2"
  - Second level (chapter sections):
    - In "chap-1\section-1\README.md" $\Longrightarrow$ "sort: 1"
    - In "chap-1\section-2\README.md" $\Longrightarrow$ "sort: 2"

Jekyll will create a HTML page for each "README.md" file found in the directory tree.
For your documentation to be easy to browse and read take time to organize it properly and split the pages accordingly.
The directories "_includes" and "_bibliography" are ignored by Jekyll when searching for "README.md" files, however their content is used to produce the website.

To install the packages required by the "Gemfile" to build your site use:

```
user@localhost ~/Program-doc]$ bundle install
```

Then to build the site use:

```
user@localhost ~/Program-doc]$ bundle exec jekyll build
```

You will see that this command creates a new directory name "_site" that contains your website.

```
user@localhost ~/Program-doc]$ ls _site
```

You can give a try to this website on your computer using:

```
user@localhost ~/Program-doc]$ bundle exec jekyll serve
```

And simply follow the instructions.

3.7.4 Hosting the website on GitHub

To use GitHub to host your documentation:

- Create a dedicated repository, in the following example: "TTPSLR/Program-doc"
- After properly configuring the connection to the remote repository, upload the contents of the "~/Program-doc" but the folder "_site" to the main branch, separate branches allows to keep track of both the source material and the final content of the website:

```
user@localhost ~/Program-doc]$ mv _site ../
user@localhost ~/Program-doc]$ git checkout -b Main
user@localhost ~/Program-doc]$ git add .
user@localhost ~/Program-doc]$ git commit -m "To build the doc"
user@localhost ~/Program-doc]$ git push origin Main
```

- After properly configuring the connection to the remote repository, upload the contents of the "_site" to the GitHub pages, the "gh-pages" branch:

```
user@localhost ~/_site]$ git checkout -b gh-pages
user@localhost ~/_site]$ git add .
user@localhost ~/_site]$ git commit -m "Documentation"
user@localhost ~/_site]$ git push origin gh-pages
```

After a short while the result is updated, and as is illustrated on figure 3.10, the website is alive.
Open the "Pages" tab in the "Settings" menu, to find out the web address of your site.
In this example: https://ttpslr.github.io/Program-doc

The GitHub pages address have the form: https://**Author**.github.io/**Program**

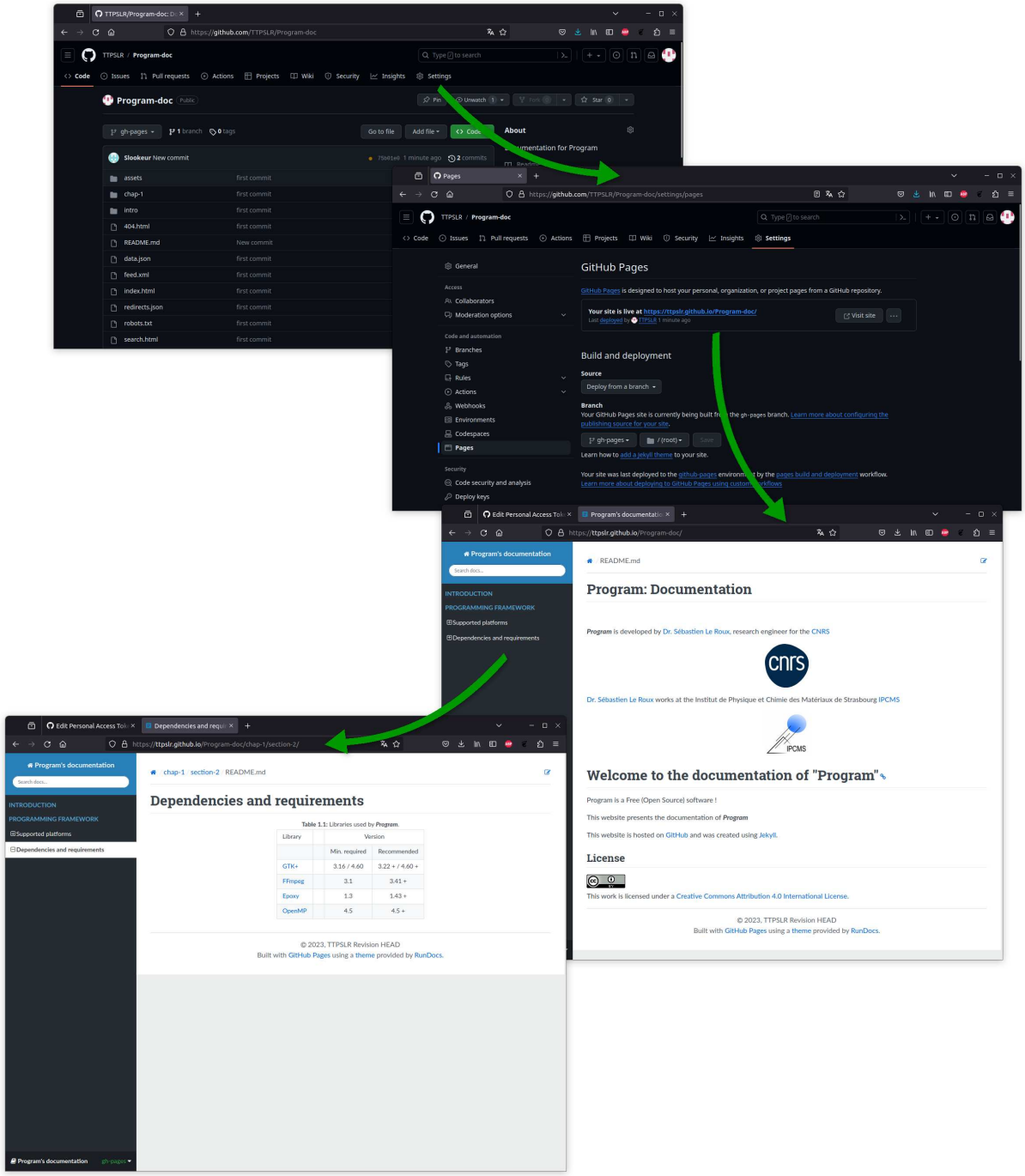


Figure 3.10: Hosting the documentation on GitHub

CHAPTER 4

LINUX DISTRIBUTION

In the Linux world two distributions became widely famous, mostly because of the package distribution systems they implemented:

- The Red Hat Linux distribution that created and maintain the RPM Package-Manager "Red Hat Package Manager" system, that handles the installation/removal of , files with the `.rpm` extension.
- The Debian Linux distribution that created and maintain the dpkg package management system, that handles the installation/removal of DEB "Debian Packages", files with `.deb` extension.

In 2023 there are thousands of Linux distributions, most of them are either Red Hat based and integrate the RPM Package-Manager, or, Debian based and integrate the dpkg package management system.
Integrating the package management systems of both Red Hat and Debian distributions is the most natural way to efficiently distribute your program to the entire open source / Linux community.
In the next sections I will review the requirements to achieve it providing that you already built a proper GNU tarball, or for that matter that you are already using any other build system.

4.1 RPM

I the next pages I will introduce the proper way to build a RPM file suitable to be distributed by Fedora Linux which is the upstream source for Red Hat Enterprise Linux.

Maybe more importantly I will assume that you are working on Fedora Linux, that would make a lot of sense since we are talking about building Fedora RPMs here. Some of the commands I will use thereafter can only be found on Fedora, therefore I would recommend to download and install the latest Fedora version either on your computer or in a virtual machine.

To prepare a RPM file you need first to install the appropriate tools:

```
user@localhost ~]$ sudo dnf install fedora-packager fedpkg rpmdevtools
```

If I intend to provide tips and trick to help you prepare your RPM, I strongly recommend that you give a look to:

- The RPM Packaging Guide.
- The official Fedora packaging guidelines.
- The Fedora packaging tutorial.

Indeed this guide is not meant to be a thorough review of RPM packaging, that is actually much more complicated than for the simple desktop application used afterward to illustrate the process.

4.1.1 The "`.spec`" file

The "`.spec`" file is a fundamental element in the packaging workflow. This file can be thought of as the "recipe" that the rpmbuild utility uses to actually build an RPM. It tells the build system what to do by defining instructions in a series of sections.
The next pages will introduce the basics required to create a simple "`.spec`" file, the complete example is provided in appendix B.1.

Please consider that each section of the "`.spec`" file as detailed between sections 4.1.1.1] and 4.1.1.9 can be completed by your own set of specific instructions.

To create a new "`.spec`" file, use:

```
user@localhost ~]$ rpmdev-newspec program
```

This will create a ready to be filled "**program.spec**" file:

```
# Initialization section - see [Sec.~4.1.1.1].
Name:           program
Version:
Release:        1%?dist
Summary:
License:
URL:
Source0:

# Information section - see [Sec.~4.1.1.2].
Provides:

%description

# Requirement sections - see [Sec.~4.1.1.3].
BuildRequires:
Requires:

# Preparation to build section - see [Sec.~4.1.1.4].
%prep
%autosetup

# Build instructions section - see [Sec.~4.1.1.5].
%build
%configure
%make_build

# Install instructions section - see [Sec.~4.1.1.6].
%install
%make_install

# Post-installation Linux integration section - see [Sec.~4.1.1.7].
%check

# Package file paths section - see [Sec.~4.1.1.8].
%files
%license add-license-file-here
%doc add-docs-here

# ChangeLog section - see [Sec.~4.1.1.9].
%changelog
* Wed Mar 29 2023 Your Name <your.email@host.eu>
-
```

In the next pages we will browse and complete this file section by section.

4.1.1.1 Initialization section

```
# In a spec file, commented lines:
#     Start by the # symbol.
#     When commenting a % symbol, require to double it using another %.
# Ex: This line is a commented line, %%{Name} is the first variable defined bellow.

# The name of the package, MUST be identical to the prefix name of the ".spec" file:
Name:       program
# Optional, to define a variable using the %%global instruction.
# A variable upname is created, and used bellow as the name of the GitHub repository,
%global upname Program
# Version of the program:
Version:    1.2.12
# Release number of the RPM package, independent of the program's version.
#    - Each time you provide a new release of program set this value to 1
#    - Each time you modify something else in the package increment this number.
# Here the value is set to 3, meaning that it is the second modification
# of the package since release of version 1.2.12 of the program.
# %{?dist} is a mandatory Fedora tag: .fc37, .fc38 ...
Release:    3%{?dist}
# Summary information about the program:
Summary:    An nice program
# License information, should use the SPDX identifier, more information here:
#     https://spdx.org/licenses/
#     https://docs.fedoraproject.org/en-US/legal/license-field/
License:    AGPL-3.0-or-later
# Source tarball that uses a known build system.
# GitHub repository of the program, specify the archive to download using its tag:
# In this example the GitHub tag is "v1.2.12"
Source0:    https://github.com/Author/%{upname}/archive/refs/tags/v%{version}.tar.gz
# To sign the RPM package with a GPG key uncomment the next 2 lines:
# Source1:    ./v%%{version}.tar.gz.asc
# Source2:    %%{name}.gpg
# The webpage of the project:
URL:        https://www.%{name}.com/
```

4.1.1.2 Information section

```
# Optional, to inform the package manager about your package.
# This allows your package to be required by another one,
# using the Requires: instruction - see [Sec.~4.1.1.3]
Provides: %{name} = %{version}-%{release}

# A short description of the program capabilities, up to the next %:
%description
This program is amazing:
- It uses GTK3 for the graphical user interface.
- It uses OpenGL to render 3D stuff
- It uses FFMPEG to encode videos
```

4.1.1.3 Requirement sections

This section regroups requirements, to build and to run the program:

```
# To set a requirement to build the program use the BuildRequires: instruction:

# If the GNU Autotools are required to build it:
BuildRequires: make
BuildRequires: automake
BuildRequires: autoconf
```

```
# If CMake is required to build it:
BuildRequires: cmake
```

```
# If meson is required to build it:
BuildRequires: meson
```

```
BuildRequires: pkgconf-pkg-config
BuildRequires: gcc
BuildRequires: gcc-gfortran
BuildRequires: libgfortran

# Library required to build the program, test performed by pkg-config
BuildRequires: pkgconfig(gtk+-3.0)
BuildRequires: pkgconfig(libxml-2.0)
BuildRequires: pkgconfig(glu)
BuildRequires: pkgconfig(epoxy)
BuildRequires: pkgconfig(libavutil)
BuildRequires: pkgconfig(libavcodec)
BuildRequires: pkgconfig(libavformat)
BuildRequires: pkgconfig(libswscale)

# Requirements for FreeDesktop integration - see [Sec.~4.1.1.7]:
BuildRequires: desktop-file-utils
BuildRequires: libappstream-glib

# To sign the RPM package with a GPG key uncomment the next line:
# BuildRequires: gnupg2

# To set a requirement to run the program use the Requires: instruction:
# Requirements, packages that are needed, at runtime.
# The only ways to be sure what to add here:
# - Use rpmlint to verify your rpm after the build - see [Sec.~4.1.3]
# - Try installing your RPM on a clean system.
Requires: gtk3
Requires: mesa-libGLU
```

4.1.1.4 Preparation section

This section prepare everything required to build the package, not only the program itself.

```
%prep
# To sign the RPM package with a GPG key uncomment the next line:
# %%{gpgverify} --keyring='%%{SOURCE2}' --signature='%%{SOURCE1}' --data='%%{SOURCE0}'

# Preparing the build.
# The autosetup instruction automates the setup process.
# The "-n" option specifies the name of the folder created
# by extracting the GNU tarball:
%autosetup -n %{upname}-%{version}
# Note that autosetup is independent of the build system.
# For information regarding options:
# https://rpm-software-management.github.io/rpm/manual/autosetup.html
```

4.1.1.5 Build instructions section

This section describes the instructions requires to build the program.

```
# With the GNU Autotools:
%build
# The first step is to call the configure script:
%configure
# Then build the program using the make command:
%make_build
# This will force to use parallel building.
# If you want to ajdust the number of CPU cores, use:
# %%make %%smp_mflags -j20
```

```
# With CMake:
%build
# To configure use:
%cmake
# Then to build use:
%cmake_build
# This will force to use parallel building.
```

```
# With meson:
%build
# To configure use:
%meson
# Then to build use:
%meson_build
# This will force to use parallel building.
```

4.1.1.6 Install instructions section

This section describes the instructions requires to install the program.

```
# With the GNU Autotools:
%install
# Install the program using the make command:
%make_install
```

```
# With CMake:
%install
# Install the program using:
%cmake_install
```

```
# With meson:
%install
# Install the program using:
%meson_install
```

4.1.1.7 Post-installation Linux and FreeDesktop integration section

The **check** section allows to input commands required after the installation process.
If your are building an application then 2 files must be delivered with your package:

- The custom MIME association "**-mime.xml**" file

  MIME types are used to create and define file association(s) with a program.
  An example of custom MIME association file is provided in the appendix B.3.1.

- The desktop entry or "**.desktop**" file

  Desktop entries are desktop configuration files describing how a particular program is to be launched, how it appears in menus, etc.
  An example of desktop entry for a desktop application is provided in the appendix B.3.2.

- The AppStream metadata or "**.appdata.xml**" file

  AppStream metadata allows upstream projects to define metadata about the components they provide using small XML files, metainfo files, which get installed into locations on the client system and are used by distributors to enhance their metadata.
  The name of this file should follow the structure:

  web.adress.reversed.appdata.xml

  - The website of the program is: https://www.program.com
  - The file should be named: **com.program.www.appdata.xml**

An example of AppStream metadata file for a desktop application is provided in the appendix B.3.3.

Both cases require actions to be performed so that the system will know new data has been installed, these actions, standard Linux system post-installation and integration commands, are to be described in the **check** section:

```
# You can add in the check section any action required after install:
%check
# For the .desktop file:
desktop-file-validate %{buildroot}/%{_datadir}/applications/%{name}.desktop
# For the .appdata.xml file:
appstream-util validate-relax --nonet \
    %{buildroot}%{_metainfodir}/com.%{name}.www.appdata.xml
```

4.1.1.8 Package file paths section

The **files** section describes the directory created, and the files installed, by the package.

```
%files
# License information, this is mandatory
%license COPYING
# The binary, executable command installed:
%{_bindir}/%{name}
# The package home directory, usually "/usr/share/prog"
# that contains the data required by the program.
# Note there is no need to specify all files.
# The name of the directory to be removed is enough:
%{_datadir}/%{name}
# The package documentation directory:
%{_datadir}/doc/%{name}
# The manual page for the program:
%{_mandir}/man1/%{name}.1*
# The package desktop entry:
%{_datadir}/applications/%{name}.desktop
# The package metadata:
%{_metainfodir}/com.%{name}.www.appdata.xml
# The package custom MIME type:
%{_datadir}/mime/packages/%{name}.xml
# The package icons:
%{_datadir}/pixmaps/%{name}.svg
%{_datadir}/pixmaps/%{name}-project.svg
%{_datadir}/pixmaps/%{name}-workspace.svg
```

4.1.1.9 ChangeLog section

Resumes the changes made to the project for each release of the program or the package.

In the initialization section:

```
# Version of the program:
Version:   1.2.12
# Release number of the RPM package:
Release:   3%{?dist}
```

Then the ChangeLog section could look like:

```
%changelog
* Thu Mar 30 2023 Your Name <your.email@host.eu> - 1.2.12-3
- Revised package: what you did here.

* Wen Mar 29 2023 Your Name <your.email@host.eu> - 1.2.12-2
- Revised package: what you did here.

* Tue Mar 28 2023 Your Name <your.email@host.eu> - 1.2.12-1
- New release: bug corrections

* Fri Feb 24 2023 Your Name <your.email@host.eu> - 1.2.11-2
- Revised package: what you did here.

* Mon Feb 06 2023 Your Name <your.email@host.eu> - 1.2.11-1
- New release: bug corrections
```

4.1.2 Building the RPM

Now that you prepared a "**.spec**" for your program your are ready to build the RPM.
This manual will not only focus on the options required to build a RPM using a "**.spec**" file.

You have several ways to build a RPM using a "**.spec**" file for the Fedora platform:

- Local build using **rpmbuild**: on your computer.
- Local build using **fedpkg**: on your computer.
- Mock build using **mock** or **fedpkg**: on your computer in a virtual, clean, environment. It allows to check if you declared properly all dependencies to build your program.
- Copr build: remotely on the Fedora Copr project hosting server: to full-proof your RPM. The Copr server is the closest thing to the official Fedora build system named Koji. Also Copr has all the testing tools required to full-proof your RPM package before submitting it to Koji.
- Koji build: remotely on the official Fedora Koji build system: for official distribution.

4.1.2.1 Local build

The first and most simple build is the local build using **rpmbuild**.

To build the source RPM use:

```
user@localhost ~]$ rpmbuild -bs program.spec
```

You will find the results of the build in `$HOME/rpmbuild/SRPMS/`

To build the binary RPM, use:

```
user@localhost ~]$ rpmbuild -bb program.spec
```

You will find the results of the build in `$HOME/rpmbuild/RPMS/`

Note that the previous command will build the RPM for your version of Fedora.

This way of building the RPMs does not ensure that your RPM will be full proofed in terms of required dependencies, indeed your are using the libraries installed on your computer, likely used to develop your software. Everything you need is available, therefore it is unlikely for the build to failed because of missing dependency(ies) in the "**.spec**" file.
The best way to test your "**.spec**" file is to do a mock build, see [Sec. 4.1.2.2].

Another simple way to build your RPM locally is to use **fedpkg**.
fedpkg is used to interact with Fedora infrastructure and deals for you with the workflow of building RPMs for the Fedora Linux distribution. It is a powerful tool that offers a number of commands and options, check out this link for more information about **fedpkg** and the options available.

To build your RPM navigate to the folder that contains your "**.spec**" file and:

1. Download the source of the program as specified in the "**.spec**" file:

   ```
   user@localhost ~]$ spectool -g program.spec
   ```

   spectool is used to download the sources of the program using the information provided in the "**.spec**" file, **-g** option gets the sources/patches that are listed with an URL.

2. Use **fedpkg** to build the RPM:

   ```
   user@localhost ~]$ fedpkg local
   ```

   fedpkg is intended to work with the Fedora distribution system, and can only work locally by providing the sources of the "**.spec**" file in the active repository.
   The default option for **fedpkg** is to build the RPM for the latest Fedora release to date.
   You will find the results of the build in the current directory in "result_program".

   Optionally you can provide the target Fedora release version:

   ```
   user@localhost ~]$ fedpkg --release f36 local
   ```

4.1.2.2 Mock build

A mock, or mock build, is a build from scratch in a clean environment.
It requires to use the mock command:

```
user@localhost ~]$ mock -r fedora-39-x86_64 ~/rpmbuild/SRPMS/program.*.src.rpm
```

Where:

- The **-r** option followed by a keyword describes the target architecture of the RPM to build. Available keywords, or architectures, can be found in /etc/mock, this directory contains a list of "**.cfg**" files, use the name of the file without the "**.cfg**" extension.
- The mock command takes the source RPM (SRPM) as final argument.

Alternatively, and focusing only of the Fedora distribution, it is possible to use the **fedpkg** command in the directory that contains the "**.spec**" file:

```
user@localhost ~]$ spectool -g program.spec
user@localhost ~]$ fedpkg --release f38 mockbuild
```

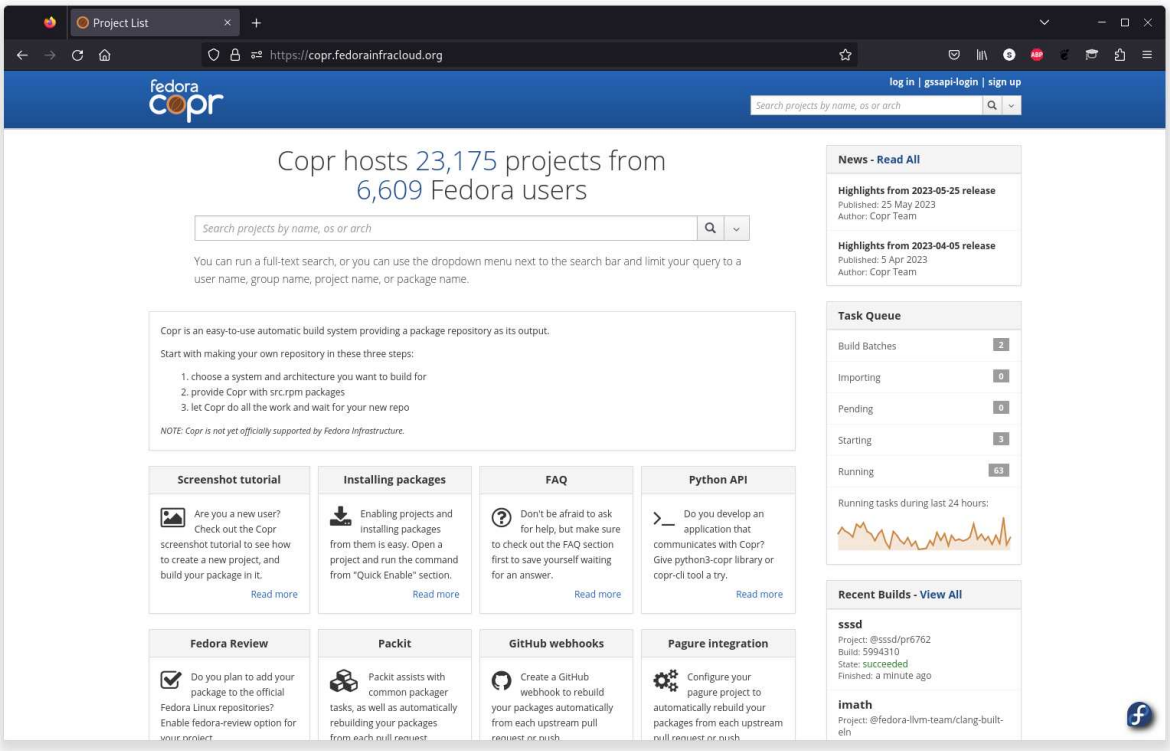
This will build the package for the architecture of your operating system.
Change the target architecture using the same keyword syntax as for the **mock** command:

```
user@localhost ~]$ fedpkg --release f38-aarch64 mockbuild
```

You will find the results of the build in the current directory in `result_program`.

4.1.2.3 Copr build

Fedora Copr is the Fedora project hosting server:



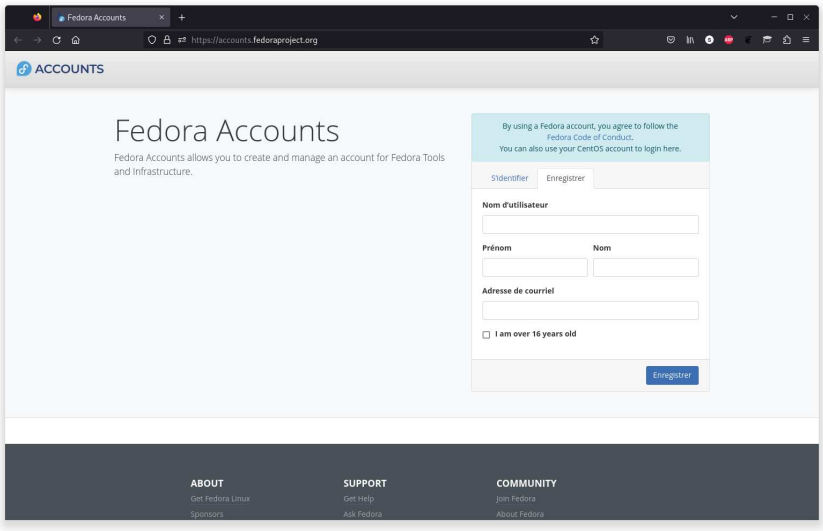
The Copr server is the closest thing to the official Fedora build system named Koji.
Also Copr has all the testing tools required to full-proof your RPM package before submitting it to Koji.

The Copr user documentation is available at:

https://docs.pagure.org/copr.copr/user_documentation.html

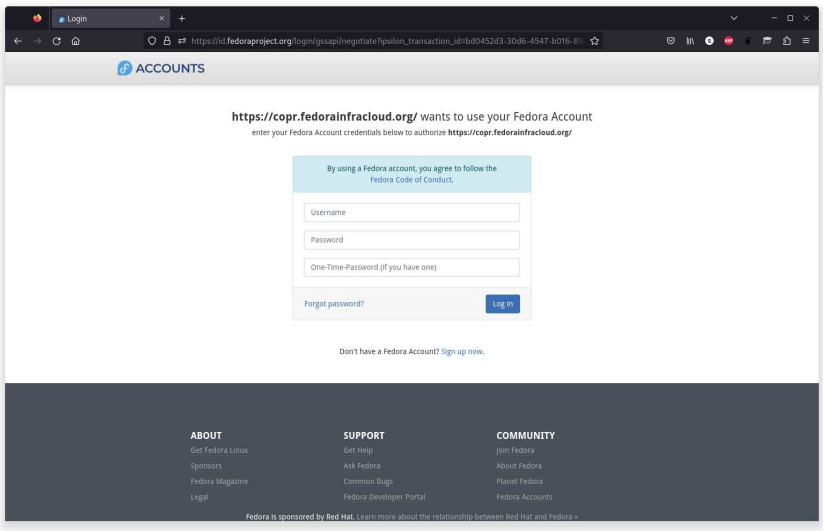
Using Copr requires first of all to create a Fedora account at: https://accounts.fedoraproject.org/ This Fedora account will be used for all Fedora resources, in particular Koji (see [Sec. 4.1.2.4]).

1. Register at https://accounts.fedoraproject.org/



2. After the registration sign the Fedora project Contributor Agreement.

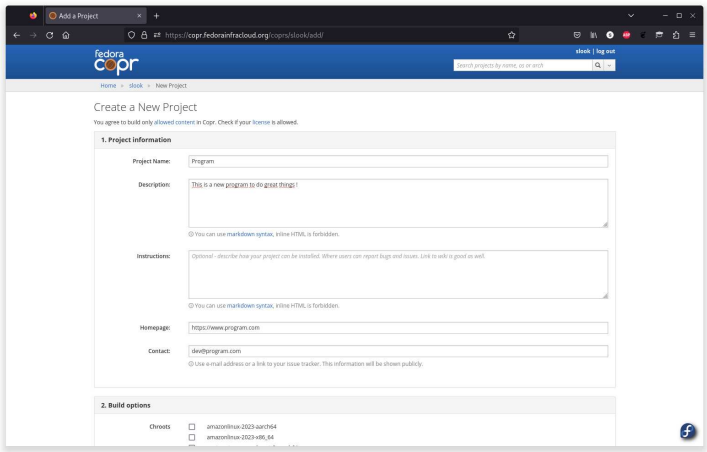
3. Navigate to https://copr.fedorainfracloud.org and login using your Fedora account.



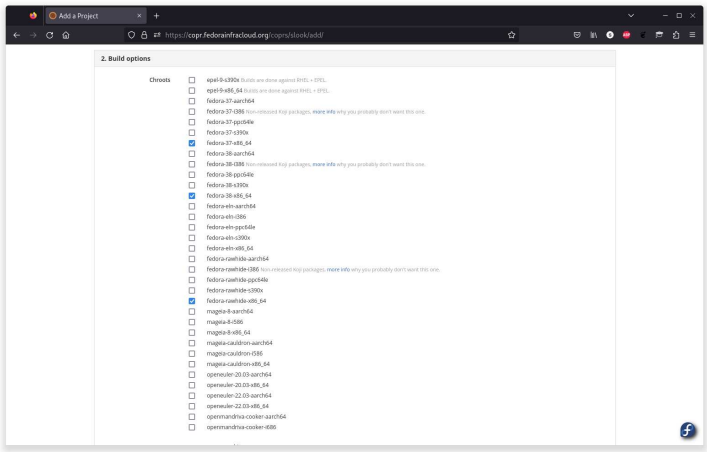
Remember, and store, the Fedora username and password that you created at this stage since they will be used on regular basis if you integrate the Fedora network and building tools (Koji, see [Sec. 4.1.2.4]).

4. Create a new project to host your work in Copr:

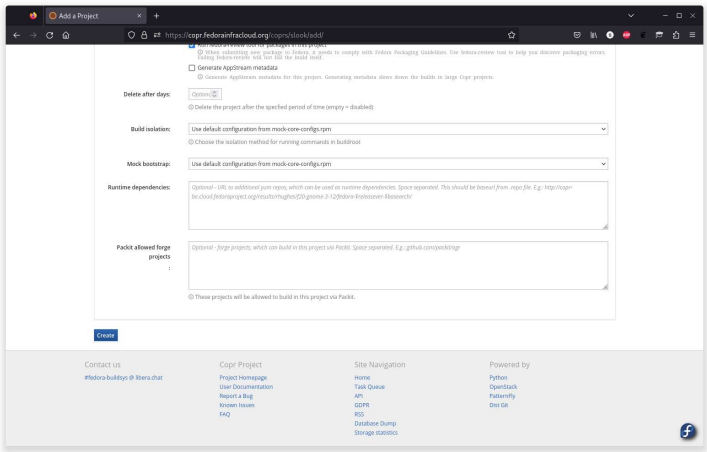
   - Describe the project (you can use Markdown):



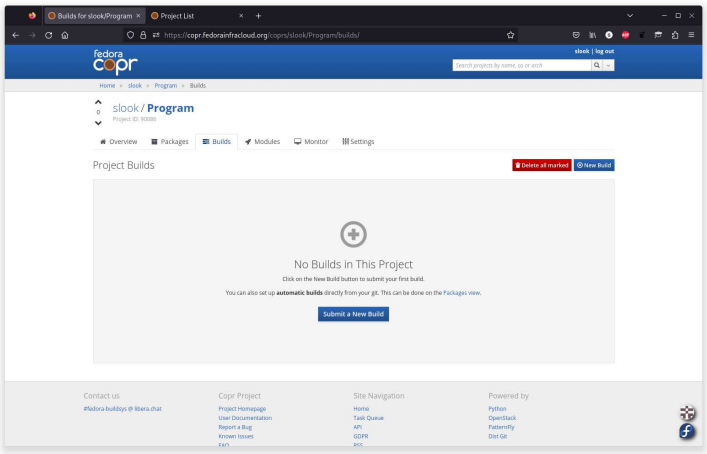
   - Select the target architecture(s):



   - Scroll down and click on "Create":

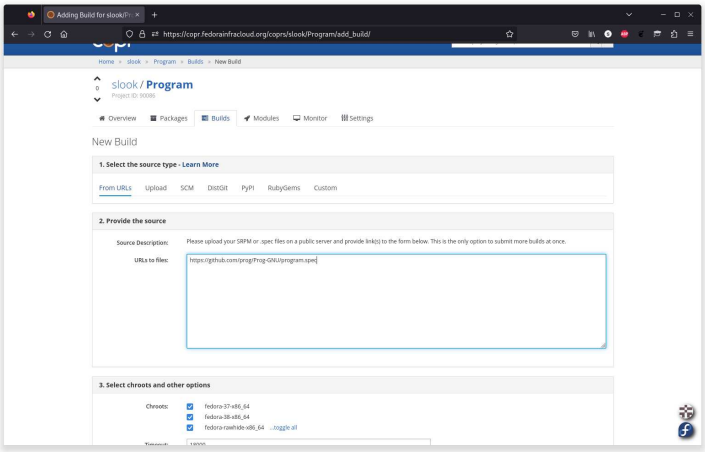


5. Then "Submit a New Build" for the project:

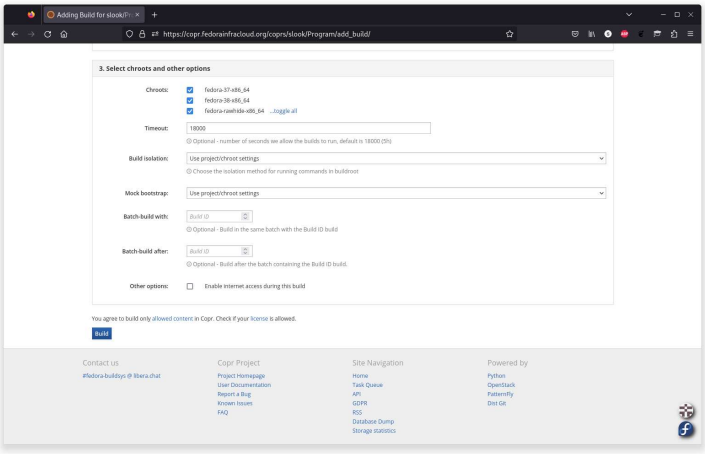


6. Build the project:

   - Build from URL, and specify the URL of your "`.spec`" file:



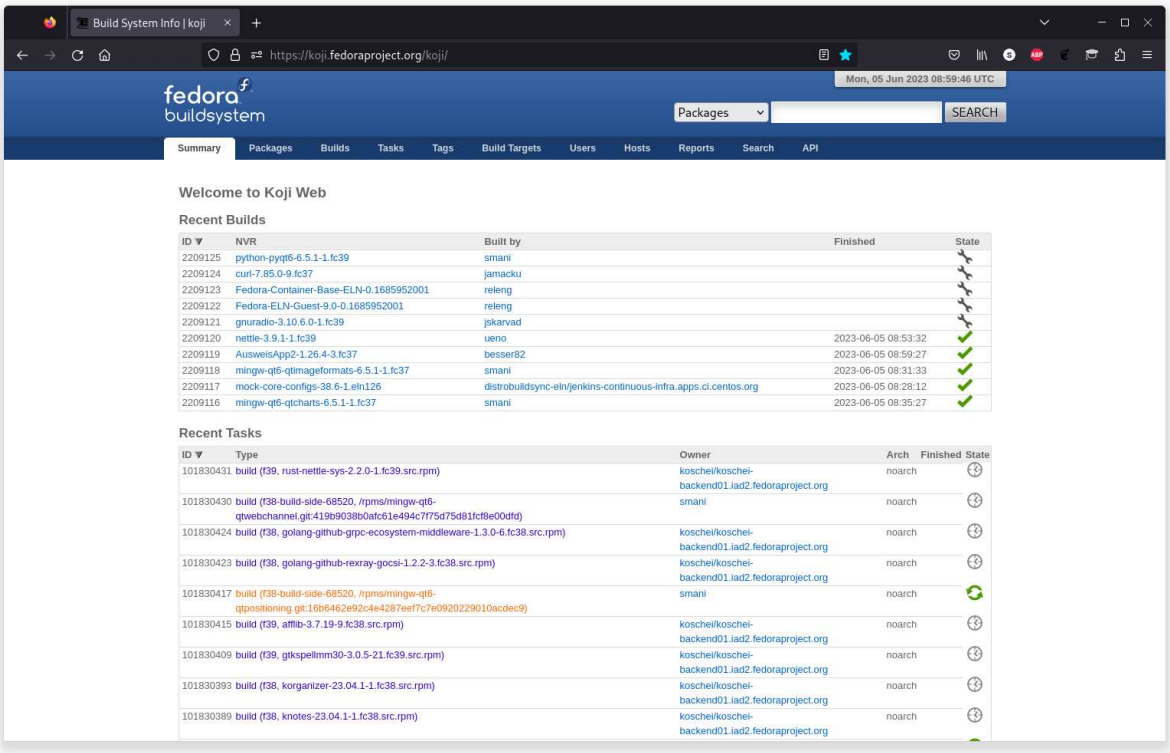
   - Scroll down and simply click on "Build":



At this point Copr is going to use the specified "`.spec`" file to build the RPMs of your project. Providing that the "`.spec`" file properly describes the sources location and the actions needed to build the RPM, then nothing else is required.

4.1.2.4 Koji build

Koji is the official RPM build system of the Fedora project: https://koji.fedoraproject.org/koji



Even if your RPM is not distributed by the Fedora project, it possible, if not recommended, to test build your RPM in the Koji environment.
To access and use Koji requires to be registered with a Fedora account (see [Sec. 4.1.2.3]). Also to use Koji requires to use the command line.

First of all to start using Koji you need to request a temporary access to the servers by initiating a new Kerberos ticket using your Fedora username and password:

```
user@localhost ~]$ fkinit -u username
Enter your password and OTP concatenated.
(Ignore that the prompt is for only the token)
Password for username@FEDORAPROJECT.ORG:
user@localhost ~]$
```

This step is mandatory to allow you to request remote builds on the Koji servers.

To build your RPM using Koji, create the source RPM (if not done already):

```
user@localhost ~]$ rpmbuild -bs program.spec
```

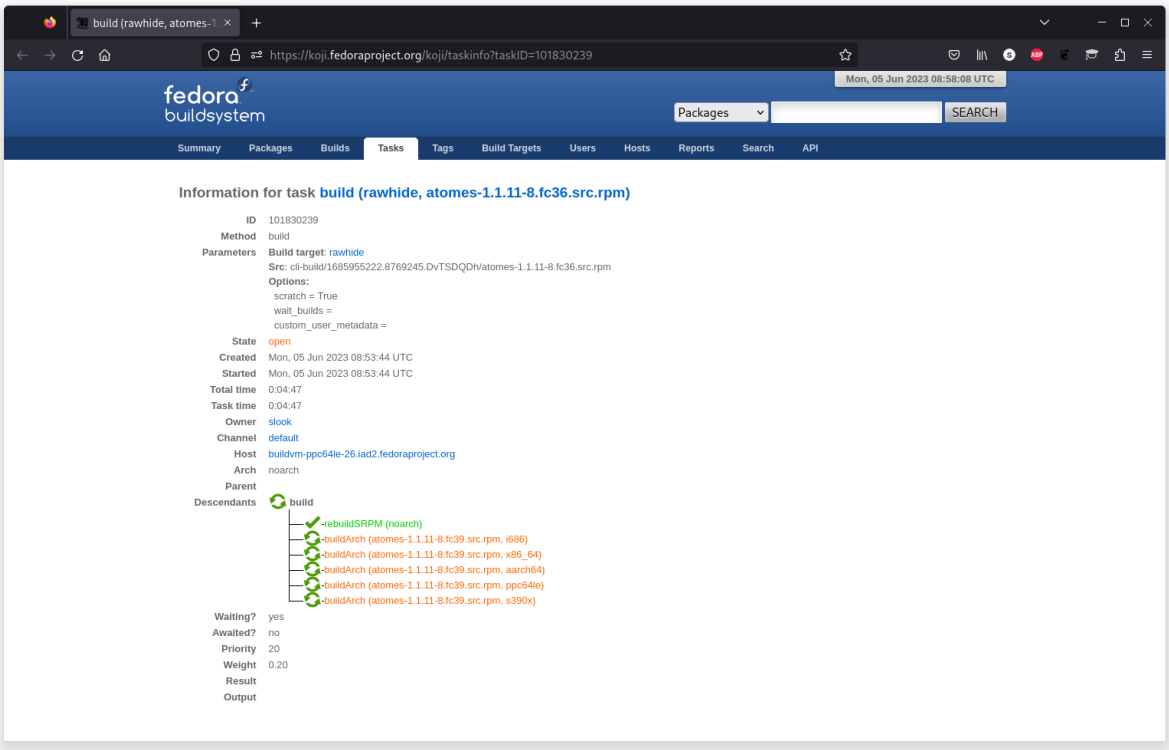
Then build the RPMs using:

```
user@localhost ~]$ koji build --sractch rawhide program-***.fc36.src.rpm
```

Replace "program-***.fc36.src.rpm" by the name of your SRPM, then press Enter.

```
user@localhost ~]$ koji build --sractch rawhide program-***.fc36.src.rpm
Uploading srpm:  program-***.src.rpm
[==================================] 100% 00:00:01   3.14 MiB   2.54 MiB/sec
Created task: 101830239
Task info: https://koji.fedoraproject.org/koji/taskinfo?taskID=101830239
Watching tasks (this may be safely interrupted)...
101830239 build (rawhide, program-***.fc36.src.rpm): free
```

The link provided in the output of the commands provides you with a webpage to follow the build process:



The information is also provided and refreshed in the terminal:

```
user@localhost ~]$ koji build --sractch rawhide program-***.fc36.src.rpm
Uploading srpm:  program-***.src.rpm
[==================================] 100% 00:00:01   3.14 MiB   2.54 MiB/sec
Created task: 101830239
Task info: https://koji.fedoraproject.org/koji/taskinfo?taskID=101830239
Watching tasks (this may be safely interrupted)...
101830239 build (rawhide, program-***.fc36.src.rpm): free
101830239 build (rawhide, program-***.fc36.src.rpm): free -> open (buildvm-ppc64le-26.iad2.fedoraproject.org)
  101830240 rebuildSRPM (noarch): open (buildvm-ppc64le-02.iad2.fedoraproject.org)
  101830240 rebuildSRPM (noarch): open (buildvm-ppc64le-02.iad2.fedoraproject.org) -> closed
  0 free  1 open  1 done  0 failed
  101830324 buildArch (program-***.src.rpm, i686): free
  101830325 buildArch (program-***.src.rpm, x86_64): free
```

Up to the end of the task:

```
user@localhost ~]$ koji build --sractch rawhide program-***.fc36.src.rpm
Uploading srpm:  program-***.src.rpm
[==================================] 100% 00:00:01   3.14 MiB   2.54 MiB/sec
Created task: 101830239
Task info: https://koji.fedoraproject.org/koji/taskinfo?taskID=101830239
Watching tasks (this may be safely interrupted)...
101830239 build (rawhide, program-***.fc36.src.rpm): free
101830239 build (rawhide, program-***.fc36.src.rpm): free -> open (buildvm-ppc64le-26.iad2.fedoraproject.org)
  101830240 rebuildSRPM (noarch): open (buildvm-ppc64le-02.iad2.fedoraproject.org)
  101830240 rebuildSRPM (noarch): open (buildvm-ppc64le-02.iad2.fedoraproject.org) -> closed
  0 free  1 open  1 done  0 failed
  101830324 buildArch (program-***.src.rpm, i686): free
  101830325 buildArch (program-***.src.rpm, x86_64): free
  101830328 buildArch (program-***.src.rpm, s390x): free
  101830326 buildArch (program-***.src.rpm, aarch64): open (buildvm-a64-31.iad2.fedoraproject.org)
  101830327 buildArch (program-***.src.rpm, ppc64le): free
  101830324 buildArch (program-***.src.rpm, i686): free -> open (buildvm-x86-19.iad2.fedoraproject.org)
  101830328 buildArch (program-***.src.rpm, s390x): free -> open (buildvm-s390x-21.s390.fedoraproject.org)
  101830325 buildArch (program-***.src.rpm, x86_64): free -> open (buildvm-x86-12.iad2.fedoraproject.org)
  101830327 buildArch (program-***.src.rpm, ppc64le): free -> open (buildvm-ppc64le-17.iad2.fedoraproject.org)
  101830326 buildArch (program-***.src.rpm, aarch64): open (buildvm-a64-31.iad2.fedoraproject.org) -> closed
  0 free  5 open  2 done  0 failed
  101830328 buildArch (program-***.src.rpm, s390x): open (buildvm-s390x-21.s390.fedoraproject.org) -> closed
  0 free  4 open  3 done  0 failed
  101830324 buildArch (program-***.src.rpm, i686): open (buildvm-x86-19.iad2.fedoraproject.org) -> closed
  0 free  3 open  4 done  0 failed
  101830325 buildArch (program-***.src.rpm, x86_64): open (buildvm-x86-12.iad2.fedoraproject.org) -> closed
  0 free  2 open  5 done  0 failed
  101830327 buildArch (program-***.src.rpm, ppc64le): open (buildvm-ppc64le-17.iad2.fedoraproject.org) -> closed
  0 free  1 open  6 done  0 failed
101830239 build (rawhide, program-***.src.rpm): open (buildvm-ppc64le-26.iad2.fedoraproject.org) -> closed
  0 free  0 open  7 done  0 failed

101830239 build (rawhide, program-***.src.rpm) completed successfully
```

4.1.3 Testing the RPM

Up to this point I introduced the different ways to build a RPM, leaving apart possible failures and errors. However if you intend to distribute your RPM, even if not by the official Fedora software repositories, it is important to know how to check for possible issues. In particular because even a successful build can contain errors. Also, not surprisingly, the first step in getting your RPM distributed by Red Hat is to get rid of a comprehensive list of possible errors.

4.1.3.1 Locally using the command line

To test your RPM via the command line use:

- Check the RPM using the **`rpmlint`** command:
  - On the "**`.spec`**" file:

```
user@localhost ~]$ rpmlint -v program.spec
```

  - On the SRPM:

```
user@localhost ~]$ rpmlint -v program*.src.rpm
```

  - On the RPM:

```
user@localhost ~]$ rpmlint -v program*.rpm
```

- Check the file permissions in the RPM using the **`rpmls`** command:

```
user@localhost ~]$ rpmls program*.rpm
```

- Mock build and review of the RPM using the **`fedora-review`** tool.

```
user@localhost ~]$ fedora-review -n program
```

  Where `program` is the name of the "**`.spec`**" file, without the `.spec` extension.
  Both `program.spec` and the source RPM must be located in the active directory.

Equivalent to:

```
user@localhost ~]$ fedora-review --rpm-spec -n program*.src.rpm
```

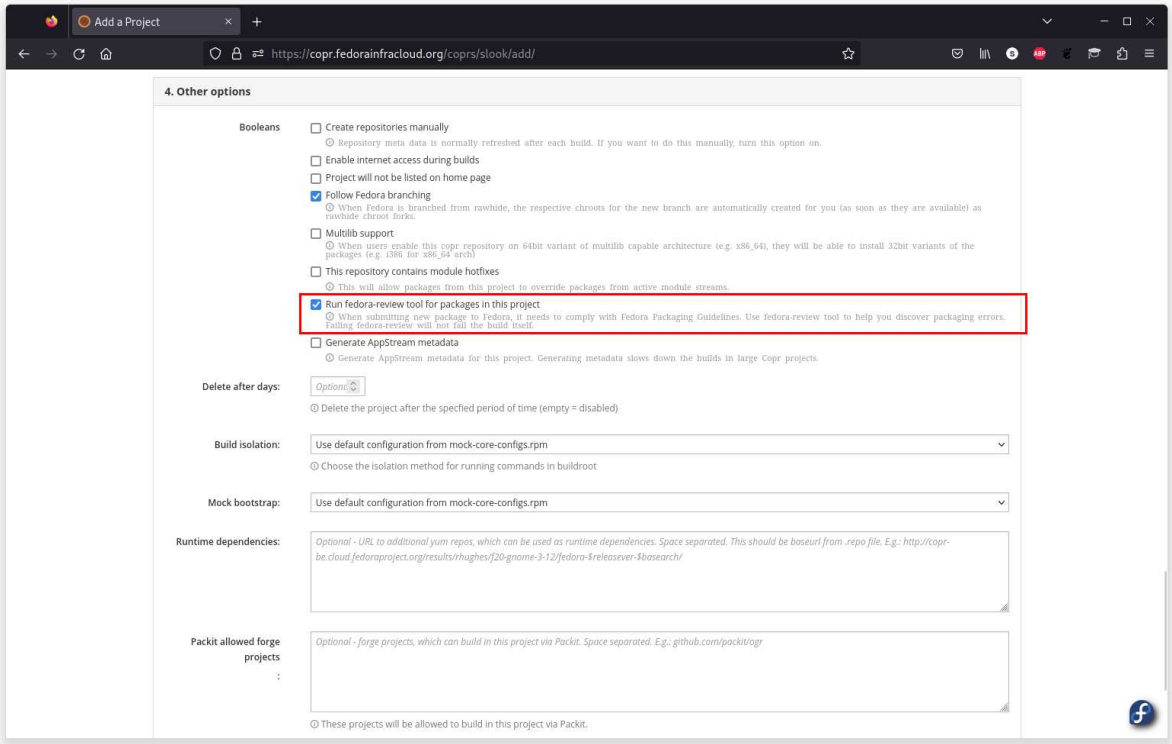
Where is `program*.src.rpm` is the source rpm, note that in this case `fedora-review` uses the spec file inside the source rpm.

`fedora-review` will output the results of the analysis in a directory with the program name.

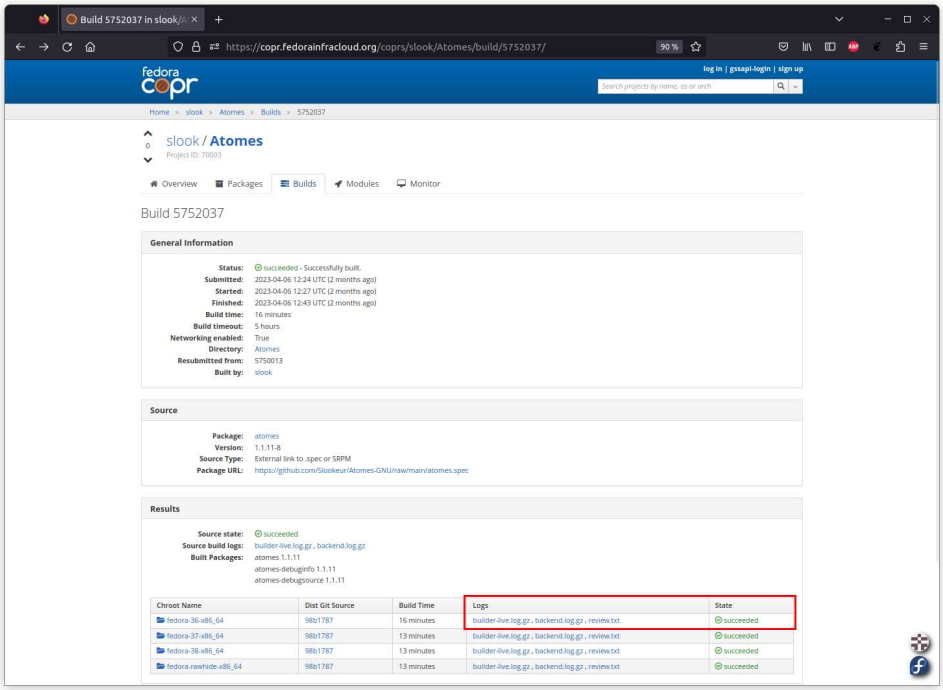
These commands produce both errors and warnings.
It is mandatory to fix all errors to distribute your RPM, while some warning can be safely ignored.

4.1.3.2 Remotely on Copr

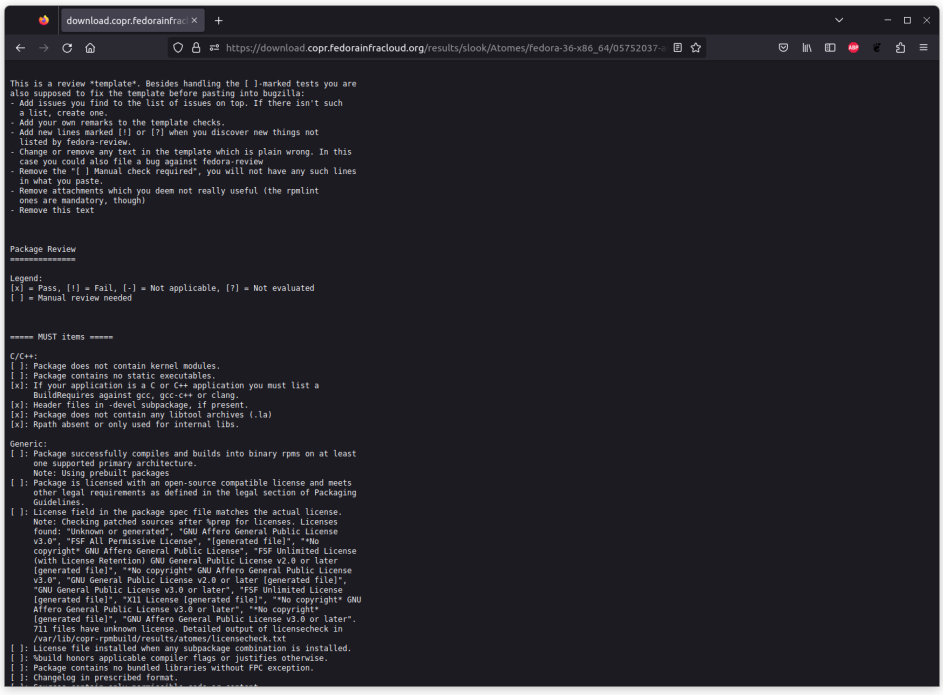
You can activate review tools on Copr to get a lot of information on the build, when creating a new project or editing the project settings:



After the building the RPM with the proper option, check out the results and open the corresponding information page:



Scroll down if needed, and in the bottom section check the "Logs" section, several files outputs of the build are provided, including a file "`review.txt`", click on the link to access the results of the Fedora review:



4.1.4 Submitting your RPM to the Fedora project

At this point I will consider that you already prepared you own version of the RPM package, ideally following the guidelines provided in this manual. If not you should really take the time to prepare a first, raw, version of your RPM package. Doing so will at least tell the Fedora packagers, members of the Fedora community in charge of packaging applications, that you are willing to be part of the process, as you should.

Ideally you should already have test build this package in Copr and / or Koji:

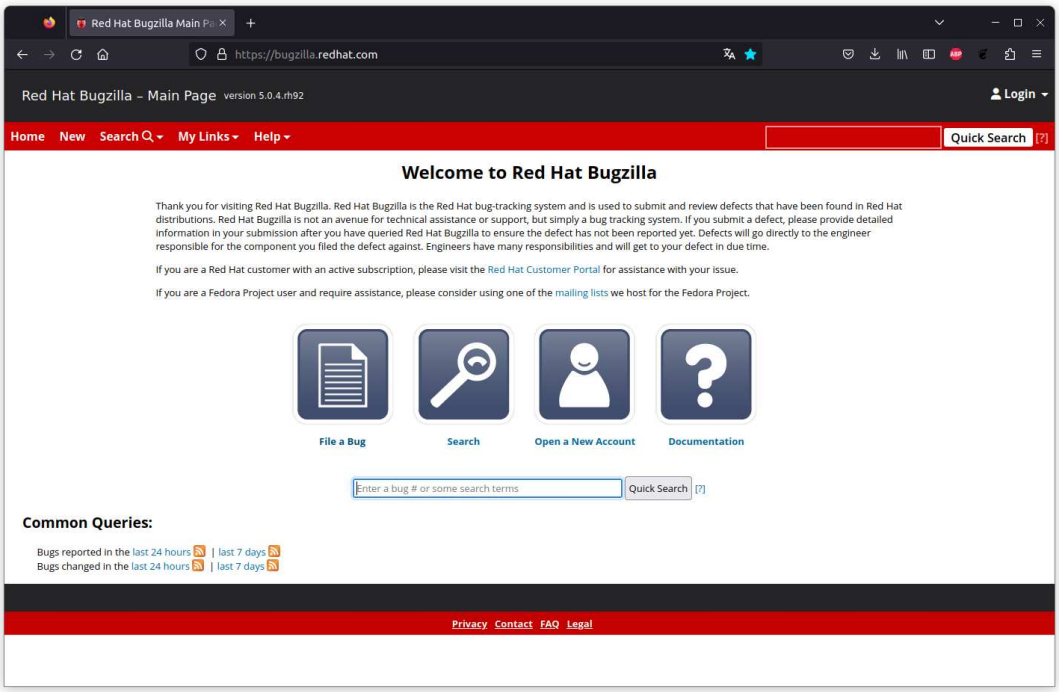
- For Copr copy the link to the package review information (see [Sec. 4.1.3.2]).
- For Koji copy the address of the webpage with the build information (see [Sec. 4.1.2.4]).

To submit your RPM package to the Fedora project:

1. Create a Fedora account (see [Sec. 4.1.2.3]).
2. Create a Red Hat Bugzilla account, with the same email used to create the Fedora account.
3. Create a message in https://bugzilla.redhat.com/ to request the review of your package.
4. Create a message to introduce yourself and ask to join the Fedora Package Maintainers.

4.1.4.1 The Red Hat Bugzilla message to request a new package review

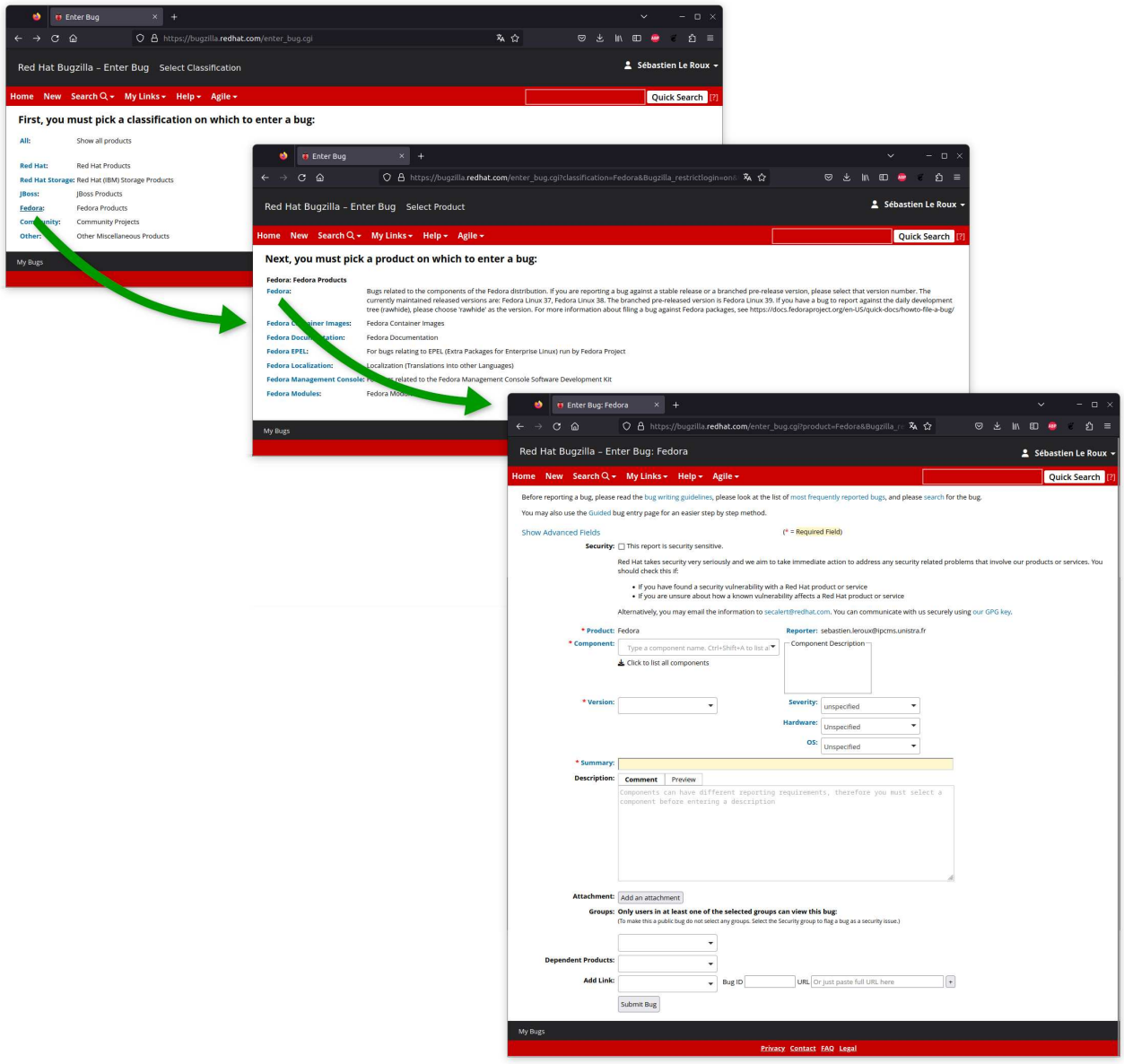
1. Login to Red Hat Bugzilla



The click of "File a Bug", please note that depending if you logged in using your "Fedora Account System" or a "Red Hat Bugzilla Account" the page dedicated to the preparation of the bug message might be slightly different.
For the sake of consistency with this tutorial I would recommend to login using the "Red Hat Bugzilla" account you just created.

2. Then "File a Bug" and create a new message:

    - Click of "Fedora", then click of "Fedora" again:



You are now of the interface dedicated to the preparation of the message to ask for your package to be reviewed by members of the Fedora Package Maintainers.

- In Component select (or enter) "`Package Review`"

- Summary should start with:

  "Review request: program - short description"

  - program is the name of your program / package.

    It is the name that will be used to create the Fedora package, and it is case sensitive, so choose it carefully, basically this is the keyword Fedora, and Red Hat based Linux, users will use to search for and install your program using:

    ```
    user@localhost ~]$ sudo dnf install prog
    ```

  - short description describes its purpose in as few words as possible

- Priority, Hardware and OS should be set to "Unspecified"
- Severity should be set to "Medium"
- In the message section, you can delete the "Bug Description" section since your are not reporting a bug, instead you are asking for help to review your new package:
  - In the introduction part of the message:
    * Introduce yourself:
      · Real name
      · Fedora Account System username (useful for future exchanges)
      · Tell more about your work.

        Doing this professionally it is a good idea to tell Package Maintainers about your job, indeed it is likely to help you find people in the same area of expertise, the best way to get help and to find a sponsor (see below).

      · Introduce briefly your software

    * Provide a link to the repository of the ".spec" file used to create the RPM.
    * Provide a link to the last SRPM (source RPM).
    * Ideally provide a link to the last successful Koji (or Copr) build.
    * State that you need a sponsor: this is mandatory for your first package.

      A sponsor is an official member of the Fedora Package Maintainers experienced enough to ensure that your package is properly prepared, and who can validate its integration to the Fedora ecosystem. Your sponsor will also register you as an official Fedora Package Maintainer for your package.

  - In the Description part of the message:
    * Describe your program with details.
    * Describe the interest of the program for the community.

3. Test the information your provided in your bug "Package review" message:

   Once the email has been sent you will receive a link to a webpage hosting the entire discussion on Red Hat Bugzilla at the address:

   `https://bugzilla.redhat.com/show_bug.cgi?id=???????`

   Where `???????` is the bug message id number that you will find on the email you received and on the webpage hosting the discussion.
   The information provided in your bug message can be tested by other Fedora packagers using the **`fedora-review`** command:

   ```
   user@localhost ~]$ fedora-review -b ???????
   ```

   `fedora-review` then retrieves the "`.spec`" file from the address provided in the bug message to build and test your package. Make sure that your package can be built like this because it is likely how official Fedora packagers will try to build and test it.
   Note that you can edit your messages, including the first one, on this webpage to correct any information if required.

4.1.4.2 Joining the Fedora Package Maintainers

In the same time you need to join the Fedora Package Maintainers, so that you would later on be able to manage your package from the official Fedora repository. The next lines basically follow the dedicated tutorial: Join the Package Maintainers

1. Join the following Fedora mailing lists:

   - The Fedora devel list
   - The Fedora devel-announce list
   - The Fedora packaging list

2. Send an email to devel@lists.fedoraproject.org to introduce yourself and your package.

   You should now introduce yourself to the community, the primary purpose of this is to begin the process of building trust by allowing the Fedora community members to get to know you a bit more. In order to establish a level of trust between yourself and the other members of the project use your real name, describe your motivations, and a description of the software your are submitted for review.

   The email subject should be: `Self Introduction:` **`Your Name`**

4.1.4.3 After that: the next steps

The next steps of the process are not up to you, or not entirely anyway.
As soon as both personal introduction and bug message have been sent, the community feedback is required for your package to go further. Packager(s) will look into your bug message to test your package, they will likely use the **`fedora-review`** tool to build and test it, as introduced previously, so make sure that the build works like this.
The more your ensured that your package follows the official Fedora packaging guidelines (check the output provided by **`fedora-review`** or Copr), the more easily your package will be accepted.

The most complicated part is likely to find a sponsor, an official Fedora package maintainer experienced enough to be allowed to register new package and their maintainer. This is done via exchanges with the Fedora package maintainer community, here are some advise:

- The process can take time: be patient !
- Search for people that could be interested by your software in the community: remember that a good introduction is the best starting point !
- When you have someone to talk to, ask what to do to help: be part of the process all the way through !

4.1.5 Managing your official RPM package for the Fedora project

At this stage I will assume that:

- Your RPM package has been approved by your mentor
- You are now an official member of the Fedora Package Maintainers

At first you need to get familiar with the Fedora tools at your disposal to help you update and distribute your package:

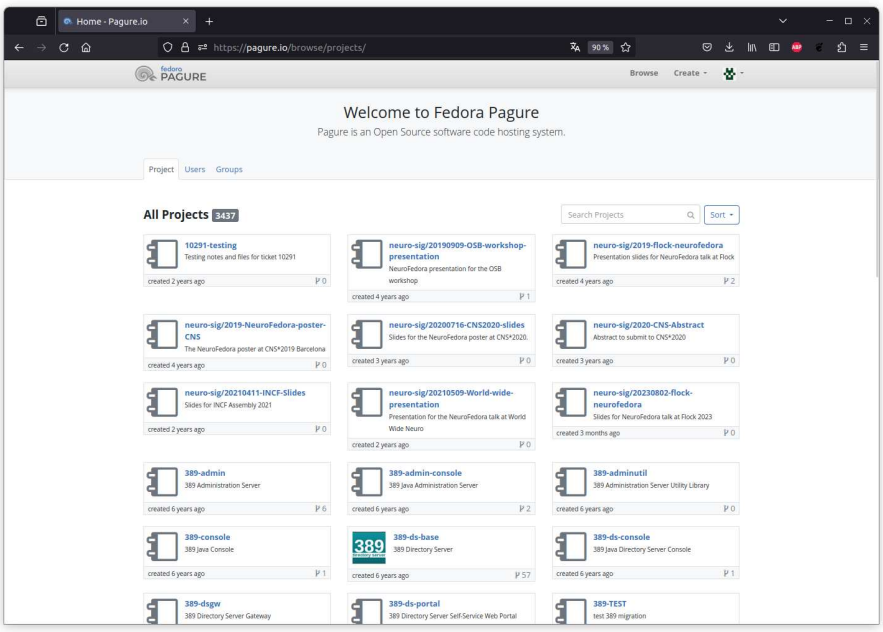
- The source repository of all Fedora packages: https://src.fedoraproject.org
- The update system for Fedora packages, Bodhi: https://bodhi.fedoraproject.org

The update process of a Fedora package can be decomposed in 3 steps:

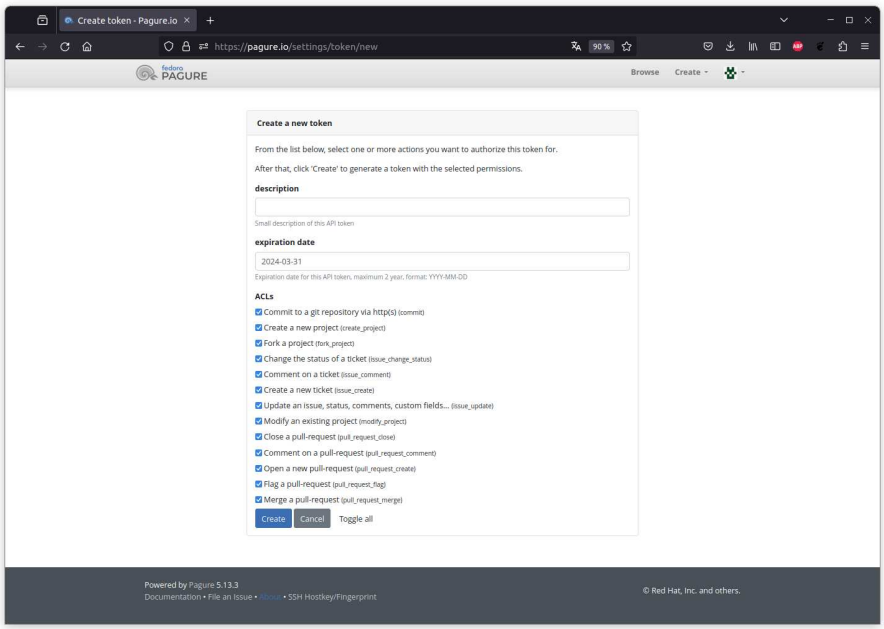
1. Request an official Pagure.io api token on https://pagure.io
2. Creating the official repository for your package on https://src.fedoraproject.org
3. Uploading new package sources and / or "`.spec`" file to the official repository
4. Building the new package using the update(s)
5. Requesting for the new built(s) to update the package in the official repositories

4.1.5.1 Request an official Pagure.io api token on Pagure

When your package passes the review comes the time to request an official Fedora Git repository for it. Before doing so, you will need a Pagure.io api token token that you can obtain from the Pagure website.
Pagure is an Open Source software code hosting system for the Fedora project:

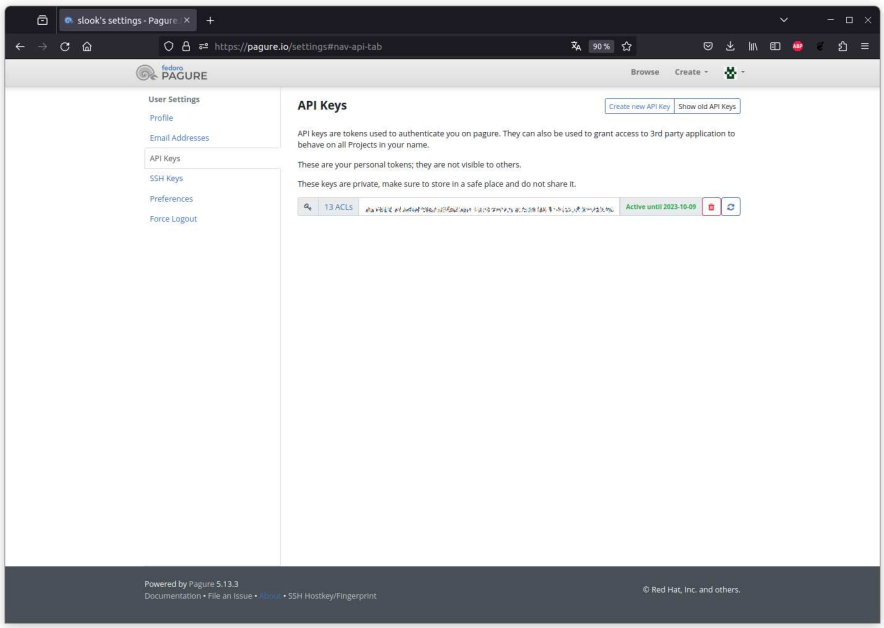


1. On Pagure login using your Fedora Account credentials
2. Create a new ticket ACL: https://pagure.io/settings/token/new



You can select all available options, basically the system allows you create and work with set(s) of API Keys with different purposes.

3. You can check the newly created API key in "`My Settings`" and then "`API Keys`"



4. Copy the content of the new key into: `~/.config/rpkg/fedpkg.conf`

```
user@localhost ~]$ cat .config/rpkg/fedpkg.conf
[fedpkg.pagure]
url = https://pagure.io/
token = <generated-code-to-be-inserted-here>
```

Note that the API key has an expiration date, so you will likely have to reproduce these few steps at some point in the future.

This will ensure that the **fedpkg** command, a Fedora frontend to the `git` command, has the proper authorization access to handle the official repository of your package.

4.1.5.2 Creating the official repository for your package on https://src.fedoraproject.org

Then it becomes possible to create the official source repository for your package.
The repository will be on the server https://src.fedoraproject.org, with the address:

https://src.fedoraproject.org/rpms/`program`

This is where you will maintain you Fedora package, upload new source(s) and / or new package version(s).

The repository creation is done using the **fedpkg** command:

```
user@localhost ~]$ fedpkg request-repo program ???????
```

Where:

- `program` is the package name as provided in the bug message
- `???????` is the bug message id number

4.1.5.3 Uploading the package sources and / or "**.spec**" file to the official repository

This is done using the **fedpkg** command, a Fedora frontend to the `git` command:

1. First, if not done already, configure your Git for Fedora packaging:

```
user@localhost ~]$ git config --global user.name "Your Name"
user@localhost ~]$ git config --global user.email your.email@host.eu
```

2. Then start by cloning the official source repository:

```
user@localhost ~]$ fedpkg clone prog
```

This will copy the official Fedora source repository of "program" to your hard drive:

```
user@localhost ~]$ ls prog
program.spec README.md sources
user@localhost ~]$
```

Note that you must use the name of the official Fedora repository as created in section 4.1.5.2, in this case: "program"

3. Download the sources from the official repository:

```
user@localhost ~]$ cd prog
user@localhost ~/program]$ fedpkg sources
user@localhost ~/program]$ ls
program.spec program-v1.1.tar.gz README.md sources
user@localhost ~/program]$
```

4. Update the sources to the latest version:

```
user@localhost ~/program]$ cp ~/program-v1.2.tar.gz .
user@localhost ~/program]$ fedpkg new-sources program-v1.2.tar.gz
```

Before uploading new sources for your package to the official repository I strongly suggest that you already test-proofed the Koji build for this new package, see [Sec. 4.1.2.4].

5. Modify the ".spec" file accordingly, including the **%changelog** section:

```
user@localhost ~/program]$ vi program.spec
```

6. Commit and push the changes:

```
user@localhost ~/program]$ fedpkg commit -p -c
```

This will upload the new file(s) to: https://src.fedoraproject.org/rpms/program

4.1.5.4 Building the new package using the update(s)

This is also done using the **fedpkg** command:

1. Request a build for Fedora branch(es), first of all for the rawhide branch:

```
user@localhost ~/program]$ fedpkg request-branch --repo program rawhide
```

The rawhide branch is the development version of Fedora, usually the latest released version + 1. If the latest released is version 39 (f39), then rawhide will released as f40, then new rawhide branch will be created and later released as f41, and so on.

To request a build for another Fedora branch, for instance the Fedora 39 branch, use:

```
user@localhost ~/program]$ fedpkg request-branch --repo program f39
```

2. To build the rawhide branch:

```
user@localhost ~/program]$ fedpkg switch-branch rawhide
user@localhost ~/program]$ fedpkg build --nowait
```

To build the Fedora 39 branch:

```
user@localhost ~/program]$ fedpkg switch-branch f39
user@localhost ~/program]$ git merge rawhide:f39
user@localhost ~/program]$ git push origin rawhide:f39
user@localhost ~/program]$ fedpkg build --nowait
```

For any other branch simply replace the **f39** by the appropriate version number.

At this stage you simply need to wait, the package will be built on Koji for the requested branches. If successful the builds will be available shortly to update the versions of the package in the Fedora software repository using the Bodhi website.

4.1.5.5 Requesting for the new build(s) to update the package in the official repositories

This is done via the web interface called Bodhi: https://bodhi.fedoraproject.org.
Bodhi is a web-based system that facilitates the process of publishing package updates for Fedora.

To update the Fedora repositories with the new builds:

1. Login on Bodhi using your Fedora Account
2. Click on "`Updates`"
3. Search for your package name, here "`program`"
4. Need to create a new update to see what's next with more details

4.2 DEB

The next pages will focus on building a DEB (".deb") file suitable for distribution by Debian Linux.

Maybe more importantly I will assume that you are working on Debian Linux, that would make a lot of sense since we are talking about building Debian DEBs here. Some of the commands I will use thereafter can only be found on Debian based Linux, therefore I would recommend to download and install the latest Debian version either on your computer or in a virtual machine.

To prepare a DEB file you need first to install the appropriate tools:

```
user@localhost:~$ sudo apt install build-essential dh-make devscripts
user@localhost:~$ sudo apt install debhelper sbuild schroot debootstrap
user@localhost:~$ sudo apt install debmake debmake-doc reportbug
user@localhost:~$ sudo apt install piuparts piuparts-master piuparts-slave
```

dpkg (for **D**ebian **p**ac**k**a**g**e) is the tool managing package distribution on Debian based Linux distributions. If I intend to provide tips and trick to help you prepare your DEB, I strongly recommend that you give a look to:

- Introduction to Debian Packaging
- How to package for Debian
- Debian policy
- Tutoriel: la construction de paquets Debian
- Simple packaging tutorial
- debmake documentation
- https://wiki.debian.org/PackagingWithGit/

Indeed this guide is not meant to be a thorough review of DEB packaging, that is actually much more complicated than for the simple desktop application used afterward to illustrate the process.

4.2.1 Prerequisites

We will briefly introduce hereafter the basics of Debian package construction.
Before starting few considerations are in order:

- It is required to setup properly some information and aliases.
  To do that edit the "`~/.bashrc`" file and add the following lines:

```
# Debian packager identification
export DEBEMAIL='your.email@host.eu'
export DEBFULLNAME='Your Name'

# Improved lintian command for package verification
alias lintian='lintian -EviIL +pedantic --color auto'
```

- For Debian packaging to be successful the folder with the sources **MUST** be of the form:

```
name-version
```

  With:

  - `name`: the package name, in example afterwards: "`program`"
  - `version`: the version number(s), in example afterwards: "`1.2.12`"

```
user@localhost:~$ mkdir program-1.2.12
user@localhost:~$ cd program-1.2.12
```

  And copy the sources in this directory.

The process of preparing a Debian package for your program is called "Debianization", so let's go and Debianize away !

4.2.2 The "debian" directory

The Debian package will be built using information in the "**debian**" directory that must be located with your sources:

```
user@localhost:~/program-1.2.12$ ls -lh
-rw-r--r--.  1 user group  54K 24 mars  17:28 aclocal.m4
-rw-r--r--.  1 user group  481 24 mars  11:24 AUTHORS
drwxr-xr-x.  2 user group 4,0K 24 mars  17:28 autom4te.cache
-rw-r--r--.  1 user group 2,8K 24 mars  11:24 ChangeLog
lrwxrwxrwx.  1 user group   32 24 mars  17:24 compile
lrwxrwxrwx.  1 user group   37 24 mars  17:24 config.guess
lrwxrwxrwx.  1 user group   35 24 mars  17:24 config.sub
-rwxr-xr-x.  1 user group 243K 24 mars  17:28 configure
drwxr-xr-x   5 user group 4,0K 19 sept. 17:20 debian
-rw-r--r--.  1 user group 3,6K 24 mars  11:24 configure.ac
-rw-r--r--.  1 user group  34K 24 mars  11:24 COPYING
drwxr-xr-x.  2 user group 4,0K 24 mars  11:24 data
lrwxrwxrwx.  1 user group   32 24 mars  17:24 depcomp
-rw-r--r--.  1 user group  34K 24 mars  11:24 config.h.in
-rw-r--r--.  1 user group  16K 24 mars  11:24 INSTALL
lrwxrwxrwx.  1 user group   35 24 mars  17:24 install-sh
-rw-r--r--.  1 user group 4,0K 24 mars  17:03 Makefile.am
drwxr-xr-x.  2 user group 4,0K 24 mars  11:24 metadata
lrwxrwxrwx.  1 user group   32 24 mars  17:24 missing
-rw-r--r--.  1 user group  247 24 mars  11:24 NEWS
drwxr-xr-x.  4 user group 4,0K 24 mars  11:24 pixmaps
-rw-r--r--.  1 user group 4,8K 24 mars  11:24 README
drwxr-xr-x.  2 user group 4,0K 24 mars  11:24 src
```

You can generate a ready to use "**debian**" directory using:

```
user@localhost:~/program-1.2.12$ debmake
```

If no "**debian**" directory is present then this command will create it, including sample files to work on. However this sample "**debian**" directory will contain some files likely unnecessary to build your first simple Debian package.

I recommend to start working with the following files and directories:

```
user@localhost:~/program-1.2.12$ ls -lh debian
-rw-r--r-- 1 user group  586 19 sept. 17:22 changelog
-rw-r--r-- 1 user group 3,2K 19 sept. 17:22 control
-rw-r--r-- 1 user group  39K 19 sept. 17:22 copyright
drwxr-xr-x 1 user group 4,0K 19 sept. 17:22 patches
-rw-r--r-- 1 user group  137 19 sept. 17:22 program.install
-rw-r--r-- 1 user group   46 19 sept. 17:22 program-data.install
-rwxr-xr-x 1 user group  103 19 sept. 17:22 rules
drwxr-xr-x 2 user group 4,0K 19 sept. 17:22 source
drwxr-xr-x 2 user group 4,0K 19 sept. 17:22 upstream
-rw-r--r-- 1 user group  183 19 sept. 17:22 watch
```

Note that the files "program.install" and "program-data.install" are not created by the **debmake** command.

4.2.2.1 The "changelog" file

The "changelog" file is to be updated for each new release of the program or the package, it describes briefly the changes in the new version:

```
user@localhost:~/program-1.2.12$ cat debian/changelog
program (1.2.12-2) unstable; urgency=medium

  * New package version

 -- Your Name <your.email@host.eu>  Tue, 19 Sep 2023 15:45:00 +0200

program (1.2.12-1) unstable; urgency=medium

  * Bug corrections and improvements.
  * See: https://github.com/Author/Program/releases/tag/v1.2.12

 -- Your Name <your.email@host.eu>  Mon, 17 Jul 2023 17:26:00 +0200
```

4.2.2.2 The "control" file

The "control" file contains build and runtime dependencies for your program.
To prepare this file requires to find the appropriate package name for each dependencies of your program:

- For building the program: in the section "Build-Depends"
- For running the program: in the section(s) "Depends"

Dependencies are presented using coma separated lists, parenthesis can be used for specific requirement.
Note that the following example illustrates a "control" file with 2 "Package:" instructions, therefore 2 packages will be created:

- A package, architecture dependent, for the binary: "Package: program"
- A package for all non-architecture dependent data: "Package: program-data"

This is a standard way of doing things in the Debian world.

```
user@localhost:~/program-1.2.12$ vi debian/control
Source: program
Section: science
Priority: optional
Maintainer: Your Name <your.email@host.eu>
Uploaders: Your Name <your.email@host.eu>
Build-Depends: debhelper-compat (= 13),
               automake,
               autoconf,
               pkg-config,
               gfortran,
               libgfortran5,
               libgtk-3-dev,
               libxml2-dev,
               libpango1.0-dev,
               libglu1-mesa-dev,
               libepoxy-dev,
               libavutil-dev,
               libavcodec-dev,
               libavformat-dev,
               libswscale-dev,
               desktop-file-utils,
               appstream-util
Standards-Version: 4.6.2
Homepage: https://www.prog.com
Vcs-Browser: https://github.com/Author/Program
Vcs-Git: https://github.com/Author/Program.git
Rules-Requires-Root: no

Package: program
Architecture: any
Depends: program-data (= ${source:Version}),
         ${shlibs:Depends}, ${misc:Depends},
         libgtk-3-0,
         libglu1-mesa,
         bash-completion
Description: program is doing something
 Program is a tool box to analyze many things
 This package provides the binaries.

Package: program-data
Architecture: all
Multi-Arch: foreign
Depends: ${misc:Depends}
Enhances: program
Suggests: program
Description: program is doing something (data)
 Program is a tool box to analyze many things
 .
 This package contains data files for program.
```

In the **Build-Depends** section if you are CMake, or meson, replace `automake` and `autoconf` by `cmake` or `meson` respectively.

Few tips to help you prepare this file:

- The "`Section:`" keyword allows to select where the program will appear in the applications menu
- The value for "`Standards-Version:`" changes with the Debian version:
  - For Debian 11: `4.5.1`
  - For Debian 12: `4.6.2`
- The short description follows the instruction "`Description`" on the same line
- The long description start on the next line:
  - Every line of the long description must start by a space character
  - Every blank line is specified using a space followed by a dot character: " ."

4.2.2.3 The "copyright" file

The "`copyright`" file is likely the most complicated file to prepare for Debian packaging. This file describes the licensing of every file in your package:

- The software license for your program
- The software license for every third-party library you ship together with the program
- The software licence for every file in your source code with specific copyright attribution

In each case license information must includes:

- The name of the software license
- A short, if available, otherwise complete text description of the license:
  - Every line of the license text must start by a space character
  - Every blank line in the license text is specified by a space followed by a dot: " ."

The first instructions in the "`copyright`" file are providing contact information, afterwards the "`copyright`" file is organized as follow, with a section listing the files and the associated license:

```
Files:     file(s) name(s)
Copyright: year(s) copyright owner
License:   License-Keyword
```

Follow later on by the associated license text information:

```
License: License-Keyword
 Text of the license
```

`License-keyword` information, should use the SPDX identifier: https://spdx.org/licenses/

Example:

```
user@localhost:~/program-1.2.12$ vi debian/copyright
Format: https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: program
Upstream-Contact: your.email@host.eu
                  Your Name <your.email@host.eu>
Source: https://github.com/Author/Program/

Files:     *
Copyright: 2023-2024 Your Name
License:   AGPL-3.0-or-later

Files:     Makefile.in
           src/Makefile.in
Copyright: 1994-2021 Free Software Foundation, Inc.
License:   FSFULLR

Files:     aclocal.m4
Copyright: 1996-2020 Free Software Foundation, Inc.
           2004 Scott James Remnant <scott@netsplit.com>.
           2012-2015 Dan Nicholson <dbn.lists@gmail.com>
License:   FSFULLR and GPL-2.0-or-later
```

... All other files with a license in your package must be listed as well ...

```
Files:     metadata/com.program.www.appdata.xml
Copyright: 2023-2024 Your Name
License:   FSFAP

Files:     debian/*
Copyright: 2023-2024 your Name
License:   AGPL-3.0-or-later
```

... Then all license texts are to be inserted after the file list:

```
License: AGPL-3.0-or-later
 Text of the AGPL v3.0 or later

License: FSFULLR
 Text of the FSULLR

License: GPL-2.0-or-later
 Text of the GPL v2.0 or later
```

A complete example is provided in appendix B.2.1.

Note that some tools can be used to help you out in preparing the "copyright" file:

- **licensecheck** is command line tool to search licensed file(s) in your sources:

```
user@localhost:~/program-1.2.12$ licensecheck -r --copyright .
```

- **scan-copyrights** helps you create a "copyright" file from scratch:

```
user@localhost:~/program-1.2.12$ scan-copyrights
```

4.2.2.4 The "patches" directory

This optional directory contains patch instructions required to build the Debian package, if any. In that case, meaning if source patches are required then the "patches" directory should contain

- A raw text file named "series" containing the list of all patches to apply.
- As many ".patch" files as described in "series", example:

```
user@localhost:~/program-1.2.12$ ls debian/patches
file-1.patch  file-2.patch  series
user@localhost:~/program-1.2.12$
```

With:

```
user@localhost:~/program-1.2.12$ cat debian/patches/series
file-1.patch
file-2.patch
user@localhost:~/program-1.2.12$
```

And patch file, ex: "file-1.patch", should have the format:

```
user@localhost:~/program-1.2.12$ vi debian/patches/file-1.patch
Description: this patch is doing this.
Author: Your Name <your.email@host.eu>
Forwarded: not-needed
Last-Update: 2023-09-19

--- a/src/file.c
+++ b/src/file.c
@@ -214,5 +214,5 @@
   # The next lines to print compilers and flags:
-  printf ("FC    Compiler         : %s\n", FC);
+   /*printf ("FC    Compiler         : %s\n", FC);
   printf ("FC    Compiler flags   : %s\n", FCFLAGS);
   printf ("C     Compiler         : %s\n", CC);
-  printf ("C     Compiler flags   : %s\n", CFLAGS);
+   printf ("C     Compiler flags   : %s\n", CFLAGS);*/
```

This patch will comment the lines that display compiler flags: when building the package path information is added to the compiler flags and you do no want that information to appear in the binary.

To create a patch file use the "`diff`" command:

```
user@localhost:~/program-1.2.12$ diff -u file.old file.new > file.patch
```

4.2.2.5 The "`program.install`" file

The "`program.install`" file lists the locations in the system directory tree where files are going to be installed. The content of this file depends on the installation process described using in your building system, for the examples listed in this manual:

```
user@localhost:~/program-1.2.12$ cat debian/program.install
usr/bin
usr/share/applications
usr/share/doc
usr/share/man
usr/share/metainfo
usr/share/mime
usr/share/pixmaps
```

For the examples in this manual:

`usr/bin`	Executable	`prog`
`usr/share/applications`	Desktop entry	`program.desktop`
`usr/share/doc`	Documentation	`README.md`
		`AUTHORS`
		`ChangeLog`
`usr/share/man`	Manual pages	`man1/program.1.gz`
`usr/share/metainfo`	AppStream metadata	`com.program.www.appdata.xml`
`usr/share/mime`	File association(s)	`package/program-mime.xml`
`usr/share/pixmaps`	Icons and images	`program.svg`
		`program-project.svg`
		`program-workspace.svg`

You can also have other "`.install`" file(s):

```
user@localhost:~/program-1.2.12$ cat debian/program-data.install
usr/share/program
```

The architecture non-dependent data to be installed with the program, if any.

4.2.2.6 The "rules" file

The "rules" file is an executable makefile that contains specific construction rules for your package, keep this file as simple as possible. Actually to package a simple application it is unlikely that you will need to modify this file at all.

```
user@localhost:~/program-1.2.12$ vi debian/rules
#!/usr/bin/make -f

# Uncomment the next line to enable deb-helper verbose mode
# export DH_VERBOSE = 1

export DEB_BUILD_MAINT_OPTIONS = hardening=+all

%:
        dh $@
```

4.2.2.7 The "source" directory

The "upstream" directory contains a single file:

```
user@localhost:~/program-1.2.12$ ls debian/sources
format
user@localhost:~/program-1.2.12$ cat debian/sources/format
3.0 (quilt)
user@localhost:~/program-1.2.12$
```

4.2.2.8 The "upstream" directory

The "upstream" directory contains a file in YAML format describing the upstream project being packaged:

```
user@localhost:~/program-1.2.12$ ls debian/upstream
metadata
user@localhost:~/program-1.2.12$ cat debian/upstream/metadata
Bug-Database: https://github.com/Author/Program/issues
Bug-Submit: https://github.com/Author/Program/issues/new
Repository: https://github.com/Author/Program.git
Repository-Browse: https://github.com/Author/Program
Contact: your.email@host.eu
user@localhost:~/program-1.2.12$
```

4.2.2.9 The "watch" file

"watch" file is used to check for newer versions of upstream software and to download it if necessary. For a GitHub project the syntax is as follow:

```
user@localhost:~/program-1.2.12$ cat debian/watch
version=4

opts="filenamemangle=s%(?:.*?)?v?(\d[\d.]*)\.tar\.gz%program-$1.tar.gz%" \
   https://github.com/Author/Program/tags \
   (?:.*?/)?v?(\d[\d.]*)\.tar\.gz debian uupdate
user@localhost:~/program-1.2.12$
```

4.2.3 Building the ".deb" file(s)

Few recommendations to build the ".deb" file(s):

- The commands "**dh_make**" and "**dpkg-buildpackage**" to be used to build the package are sensitive to already existing files in the upper directory, therefore before each build I recommend to clean the working directory:

```
user@localhost:~$ rm -f *_1.2.12_*
user@localhost:~$ ls
program-1.2.12
```

- The commands to build the package should be run directly above the "debian" directory

```
user@localhost:~$ cd program-1.2.12
user@localhost:~/program-1.2.12$
```

- The commands to build the package must be able to find a README file:

```
user@localhost:~/program-1.2.12$ cp README.md README
```

Then to build the package:

1. Use "**dh_make**" to create the Debian origin tarball using the active source directory:

```
user@localhost:~/program-1.2.12$ dh_make --createorig -s -y
Maintainer Name     : Your Name
Email-Address       : your.email@host.eu
Date                : Thu, 02 Nov 2023 11:37:21 +0100
Package Name        : program
Version             : 1.2.12
License             : blank
Package Type        : single
You already have a debian/ subdirectory in the source tree.
dh_make will not try to overwrite anything.
user@localhost:~$ ls ../
program-1.2.12  program_1.2.12.orig.tar.xz
```

   Note that this command must be performed on a clean source folder, without any object, or already compiled file(s), or any files created by the build system when preparing the software.
   In the example of this manual such files could be:

   - Compiled Fortran modules, if any "`.mod`"
   - Compiled object files "`*.o`"
   - Autotools generated files

   To avoid any issue simply remember to prepare each build using a clean source tarball.

2. Use "**dpkg-buildpackage**" to create the package(s) using the Debian origin tarball:

```
user@localhost:~/program-1.2.12$ dpkg-buildpackage >& ../dpkg-build.log
user@localhost:~/program-1.2.12$ ls -lh ../
total 7,0M
-rw-r--r-- 1 user group 1,1M  2 nov.  11:46 dpkg-build.log
drwxr-xr-x 7 user group 4,0K  2 nov.  11:46 program-1.2.12
-rw-r--r-- 1 user group  16K  2 nov.  11:46 program_1.2.12-2_amd64.buildinfo
-rw-r--r-- 1 user group 2,2K  2 nov.  11:46 program_1.2.12-2_amd64.changes
-rw-r--r-- 1 user group 941K  2 nov.  11:46 program_1.2.12-2_amd64.deb
-rw-r--r-- 1 user group  14K  2 nov.  11:46 program_1.2.12-2.debian.tar.xz
-rw-r--r-- 1 user group 1,3K  2 nov.  11:46 program_1.2.12-2.dsc
-rw-r--r-- 1 user group 2,2M  2 nov.  11:45 program_1.2.12.orig.tar.xz
-rw-r--r-- 1 user group 1,1M  2 nov.  11:46 program-data_1.2.12-2_all.deb
-rw-r--r-- 1 user group 1,7M  2 nov.  11:46 program-dbgsym_1.2.12-2_amd64.deb
```

   The "**dpkg-buildpackage**" builds the program and then prepare the Debian package using the result of this build.
   Note that in the example above both results and potential errors of the commands are redirected in a log file in the top directory.

After that, and if the build is successful, package(s) can then simply be installed using:

```
user@localhost:~$ sudo apt install ./program_1.2.12-2_amd64.deb \
                                   ./program-data_1.2.12-2_all.deb
```

It is also possible to list the content of the package using:

```
user@localhost:~$ dpkg -c ./program_1.2.12-2_amd64.deb
```

4.2.4 Testing the ".deb" package and the associated files

After building the package, and before entering the process of having your package distributed by Debian, I strongly recommend to test it thoroughly.

4.2.4.1 lintian

Use the "**lintian**" command to check the ".changes" file:

```
user@localhost:~$ lintian program_*changes
```

Note that the "**lintian**" command, that refers to the alias defined in section 4.2.1, is likely to produce a verbose output.
If errors appear, and they likely will, some might be corrected easily while other will be more complicated to deal with.
In any case, all errors should be fixed in order to have the package approved as an official Debian package. If that is what you intent to do, then I suggest to have a list ready for the up-coming discussions with your Debian mentor during the packaging process.

A complete local building and testing script is provided in appendix B.2.2

4.2.4.2 piuparts

You can also test the ".deb" files using the "**piuparts**" command:

```
user@localhost:~$ sudo piuparts ./program_*amd64.deb
```

"**piuparts**" tests that Debian packages handle installation, upgrading, and removal correctly. It does this by creating a minimal Debian installation in a chroot, and installing, upgrading, and removing packages in that environment, and comparing the state of the directory tree before and after. "**piuparts**" reports any files that have been added, removed, or modified during this process.
Note that you need to be in the sudoers to use "**piuparts**".

4.2.5 Submitting you DEB to Debian

At this point I will consider that you already prepared you own version of the DEB package, ideally following the guidelines provided in this manual. If not you should really take the time to prepare a first, raw, version of your DEB. Doing so will at least tell the Debian packagers, members of the Debian community in charge of packaging applications, that you are willing to be part of the process, as you should. Ideally you should already have tested this package (see Sec. 4.2.4).
Also your package should meet the Debian standards, or Debian free software guidelines, and requirements as listed in the Debian policy manual.
To submit your DEB package to the Debian community:

1. Create a Salsa account to host a version of your package.
2. Search for a suitable Debian packaging team at https://wiki.debian.org/Teams.
3. Post an ITP message on WNPP Debian Official Work-Needing and Prospective Packages.
4. Create a message to introduce yourself to the Debian community and search for a mentor and sponsor to review your package.

4.2.5.1 Create a Salsa account

Salsa is a collaborative development server for Debian based on the GitLab software. Salsa is supposed to provide the necessary tools for package maintainers, packaging teams and other Debian related individuals and groups for collaborative development (see [Fig. 4.1]).

4.2.5.2 Search for suitable Debian packaging team

In order to find a mentor and a sponsor to help you to get your package approved, search for suitable teams on https://wiki.debian.org/Teams.
In particular browse the parts dedicated to:

- Blends teams: https://wiki.debian.org/Teams#Blends_teams
- Packaging teams: https://wiki.debian.org/Teams#Packaging_teams

This is where you are likely to get in touch with people interested by your software.
For example to package "**atomes**" I got in touch with the Debichem team.

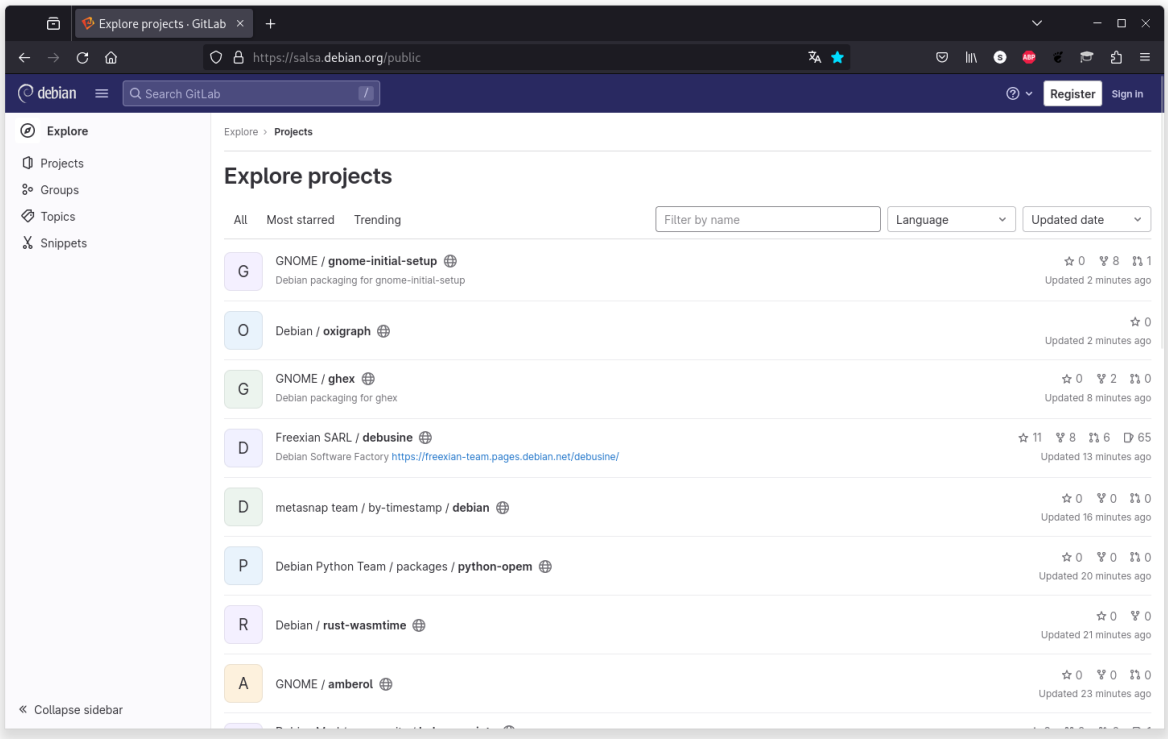


Figure 4.1: The Debian Salsa web site

4.2.5.3 Post an ITP message that will appear on WNPP

WNPP Debian Official Work-Needing and Prospective Packages, is the official Debian website that holds, among other things, the list of the prospective packages for the Debian community:

- Packages being worked on (**I**ntent **T**o **P**ackage)
- Requested packages (**R**equest **F**or **P**ackaging)

By posting an ITP message on Debian's WNPP you are announcing that you intend to do the packaging yourself. If your package meets Debian standards, then you can look for a Mentor and Sponsor to review your package and upload it.

To post the ITP message requires to use the command line and the "`reportbug`" utility:

1. Prepare a complete text message that describe your package, including:
   - Short description
   - Long description
   - Demonstration of the interest for the community
   - Request for sponsorship

   A detailed example is provided in appendix B.2.3.

2. Configure "`reportbug`":

   (a) Run "`reportbug -configure`" to create a "`~/.reportbugrc`" configuration file.

   (b) Follow the instructions and when asked:
   "`Do you have a 'mail transport agent' (MTA) ... configured ?`"
   choose **`No`**

   (c) Then enter the SMTP host for your mail server

   (d) For the user name enter your email address

   (e) For the question:
   "`Does your SMTP host require TLS authentication ?`"
   choose **`Yes`**

   You can edit the file "`~/.reportbugrc`" in particular to add your password:

```
user@localhost:~$ vi .reportbugrc
#  reportbug preferences file
#  character encoding: UTF-8
#  Version of reportbug this preferences file was written by
reportbug_version "7.10.3+deb11u1"
#  default operating mode: one of: novice, standard, advanced, expert
mode novice
#  default user interface
ui text
#  offline disables querying information over the network
# offline
#  name and email setting (if non-default)
realname "Your Name"
email "your.email@host.eu"
#  Send all outgoing mail via the following host
smtphost "www.webserver.eu"
smtpuser "your.email@host.eu"
smtppasswd "Your-Password-Here"
#  Require STARTTLS for the SMTP host connection
smtptls
#  If nothing else works, remove the # at the beginning
#  of the following three lines:
# no-cc
# list-cc-me
# smtphost reportbug.debian.org
#  You can add other settings after this line.  See
#  /etc/reportbug.conf for a full listing of options.
```

3. Use "**`reportbug wnpp`**" to send your ITP message:

```
user@localhost:~$ reportbug wnpp
```

   Then simply follow the prompt:

(a) This is an ITP

(b) Enter the proposed package name: "`program`"

(c) Enter the short description

(d) Press "`s`" to skip he bug message search

(e) Complete and send the message, note that "`reportbug`" uses a vi interface:

- To insert text press [i], then input your text.
  To navigate the document use the keyboard arrows [←], [→], [↑] and [↓].
- To save the message and to send it:
  - Press [Esc]
  - Press [:]
  - Input:

    ```
    x!
    ```

  - Press [Enter]

Note that you can modify each line from the "`Subject`" first line to end of the message, including the package name and the short description if required.

When the message has been sent you will receive an email carbon copy of your message, as well as another message acknowledging your request and including a bug number as well as a link to follow the discussion on-line.
This message will have the following subject:

```
Bug#???????: Acknowledgment (ITP: program -- short description)
```

where "`???????`" is the bug number associated with your ITP.
The link to follow the discussion on-line will be like:

```
https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=???????
```

Now what's left is to introduce your self to the Debian packaging community.

4.2.5.4 Introduce yourself to the Debian community

Simply send an email to the team you identify in the previous stage, to introduce your self, your work, and of course your program, also do not forget to mention the ITP bug number on WNPP. Remember that Debian puts a lot of focus on quality, thus the better your package is prepared before submitting the ITP the easier it will be to find sponsorship. Sponsors might be hard to find and busy, helping them out to do the job is helping your package to be accepted more easily. You can always get help from:

- Other members of a packaging team: https://wiki.debian.org/Teams

- The Debian Mentors group (if your package does not fit in a team):
  - https://wiki.debian.org/DebianMentorsFaq
  - http://mentors.debian.net/
  - The mentors mailing list: debian-mentors@lists.debian.org
- Documentation: http://mentors.debian.net/intro-maintainers
- Localized mailing list (in your own language) "debian-devel-{language}@lists.d.o":
  - debian-devel-french@lists.d.o
  - debian-devel-spanish@lists.d.o

4.2.5.5 After that: the next steps

As for the RPM side, the next steps of the process are not up to you, or not entirely anyway. As soon as both personal introduction and bug message have been sent, the community feedback is required for your package to go further. Debian Mentors and Packager(s) will look into your bug message to test your package. The more your ensured that your package follows the official Debian packaging guidelines the more easily your package will be accepted. The most complicated part is likely to find a Mentor, an official Debian package maintainer experienced enough to be allowed to register new package and their maintainer. This is done via exchanges with the Debian packaging community, here are some advise:

- The process can take time: be patient !
- Search for people that could be interested by your software in the community: remember that a good introduction is the best starting point !
- When you have someone to talk to, ask what to do to help: be part of the process all the way through !

4.2.6 Managing your official DEB package for Debian

At this point I will assume that your DEB package has been approved by your mentor.
When done she or he, will create the official repository on Salsa.
Togheter you will have decided on the most suitable packaging team for your program.
Each teams has a group on Salsa, for example the Debichem team regroups chemistry related packages: https://salsa.debian.org/debichem-team.

You can search for teams and the corresponding group on Salsa at:

https://wiki.debian.org/Teams#Packaging_teams

Then "Request Access" to the packaging team for your package.

Your mentor will create a repository for your package in the selected packaging team, once the membership granted you will be able to upload data in your repository on Salsa at the address:

https://salsa.debian.org/PACKAGING-team/Program

The official respository for your package on Salsa will contain 3 branches:

- "upstream": the sources of your program
- "pristine-tar": the "orig" archive for your package, that contains both sources and the "debian" directory to build the DEB package.
- "master": the sources of your program and the "debian" directory to build the DEB package

Debian offers a tool to hanlde packaging using the **git** command:

```
user@localhost:~$ sudo apt install git-buildpackage
```

The Debian package "git-buildpackage" provides acces to the **gbp** command that you can use to maintain your package with Git:

- Upload the new sources on the "master" branch using standard Git commands:

```
user@localhost:~/program-1.2.12$ git checkout master
user@localhost:~/program-1.2.12$ git add .
user@localhost:~/program-1.2.12$ git commit -a
```

- Then use the "**gbp**" for **g**it **b**uild **p**ackage to build the package:

```
user@localhost:~/program-1.2.12$ gbp buildpackage
```

  This will trigger the build of the source and binary packages from your Salsa repository.

- When you have produced a release ready package, tag it using:

```
user@localhost:~/program-1.2.12$ gbp buildpackage --git-tag
```

  This creates a tag, as illustrated in the chapter dedicated to hosting platforms (Sec. 3.5.1), simply using Git, the tag is extracted from the file "debian/changelog" and will have the form: debian/version", where "version" is the last version number in the file "debian/changelog", in this example "1.2.12",

For more information on **gbp**: https://wiki.debian.org/PackagingWithGit

At this point your package is almost ready to integrate the official Debian respositories, the remaining tasks are up to an official Debian developper. Contact your Debian mentor or the packaging team that accepted your software to let them know so that they can finish up the packaging process.

4.3 Flatpak

Flatpak is a utility for software deployment and package management for Linux.
It offers a sandbox environment in which users can run application software in isolation from the rest of the system.

4.3.1 Prerequesites

Give a look to the following tutorials:

- https://flatpak.org/setup
- https://docs.flatpak.org/en/latest/first-build.html

To install Flatpak and the tools required to build your program:

- Install Flatpak:
  - Fedora:

```
user@localhost ~]$ sudo dnf install flatpak
```

  - Debian:

```
user@localhost:~$ sudo apt install flatpak
user@localhost:~$ sudo apt install gnome-software-plugin-flatpak
```

- Install the Flathub repository:

```
flatpak remote-add --if-not-exists flathub https://dl.flathub.org/repo/flathub.flatpakrepo
```

- Install the suitable Flatpak runtime:

```
flatpak install flathub org.freedesktop.Platform//22.08
```

- Install the corresponding Flatpak Software Development Kit "SDK":

```
flatpak install flathub org.freedesktop.Sdk//22.08
```

- Install the Flatpak builder:

```
flatpak install flathub org.flatpak.Builder
```

4.3.2 To build the Flatpak

1. Prepare the YAML file that describe your project: "`org.flatpak.prog.yml`"
2. Init a Git repository and install the shared modules repository
   - This is required to use glu in the example
   - You might need it for other purposes later on
3. Build the Flatpak

4.3.2.1 The file "`org.flatpak.prog.yml`"

This file describes the contruction protocole to build the Flatpak for your project:

```
app-id: org.flatpak.prog
runtime: org.freedesktop.Platform
runtime-version: '22.08'
sdk: org.freedesktop.Sdk
command: prog
modules:
  - shared-modules/glu/glu-9.json
  - name: prog
    buildsystem: autotools
    no-autogen: true
    sources:
      - type: archive
        url: https//github.com/AUTHOR/RESPOSITORY/archive/refs/tags/v1.2.12.tar.gz
        sha256: "c9a540d0492f8c6b4d829b7cb8df5fc2fa5cf6374ee4e3c13f90865e9303d143"
```

This file points to a release in your GitHub or GitLab repository.
The value following "`sha256`" is the corresponding SHA256 check sum obtained using:

```
sha256sum v1.2.12.tar.gz
```

The builder will download the package from your repository, and control that it is a match.

4.3.2.2 Init a Git repository and install the shared modules repository

```
git init
git submodule add https://github.com/flathub/shared-modules.git
```

4.3.2.3 Building the Flatpak

```
ls
org.flatpak.prog.yml
flatpak-builder prog org.flatpak.prog.yml --force-clean
```

4.3.3 Running the Flatpak

If your Flatpak is not in the official repositories your need to install it manually:

```
flatpak-builder --user --install --force-clean prog org.flatpak.prog.yml
```

Then to run the Flatpak use:

```
flatpak-builder --socket=session-bus --nosocket=fallback-x11 --socket=x11 org.flatpak.prog.yml
```

4.4 Appimage

AppImage is an open-source format for distributing portable software on Linux. It aims to allow the installation of programs independently of specific Linux distributions, a concept often referred to as upstream packaging.
As a result, an AppImage can be installed and run across Ubuntu, Arch Linux, and Red Hat Enterprise Linux without needing to use specific files. The AppImage is a format that's self-contained, rootless, and independent of the underlying Linux distribution.

4.4.1 Prerequesites

Give a look to the following tutorials:

- https://docs.appimage.org/
- https://docs.appimage.org/packaging-guide/index.html
- https://appimage-builder.readthedocs.io/en/latest/
- Creating appimage
- https://docs.appimage.org/packaging-guide/converting-binary-packages/pkg2appimage.html

To create an appimage of your program, install **`appimage-builder`**:

```
wget -O appimage-builder-x86_64.AppImage \
https://github.com/AppImageCrafters/appimage-builder/releases/download/v1.1.0/appimage-builder-1.1.0-x86_64.AppImage

# make executable
chmod +x appimage-builder-x86_64.AppImage

# install in PATH to make it available as command (optional)
sudo mv appimage-builder-x86_64.AppImage /usr/local/bin/appimage-builder
```

For RedHat based Linux install **`apt`**:

```
user@localhost ~]$ sudo dnf install apt
```

4.4.2 appimage-builder

To build an appimage using **appimage-builder** requires to prepare a recipe in a JSON file. Several methods are available to let **appimage-builder** know how to retreive your program's data to build the appimage.

Recipe using a local build of your program

- Configure and build the program with the installation prefix set to "/usr"

```
user@localhost ~/program-1.2.12]$ ./configure --prefix=/usr
user@localhost ~/program-1.2.12]$ make
```

- Create an "AppDir" directory to install the program using the **DESTDIR** variable:

```
user@localhost ~/program-1.2.12]$ mkdir AppDir
user@localhost ~/program-1.2.12]$ make install DESTDIR=$PWD"/AppDir"
```

- Prepare you **appimage-builder** recipe, in a file named "program.yml"

```
# appimage-builder recipe, for details see:
# https://appimage-builder.readthedocs.io
version: 1
AppDir:
  path: ./AppDir
  app_info:
    id: program     # Name of the .desktop file
    name: program   # Name of the program
    icon: program   # Name of the icon file, no extention required
    version: 1.2.12
    exec: usr/bin/program
    exec_args: $@
  test:
    fedora:
      image: appimagecrafters/tests-envfedora-30
      command: ./AppRun
      use_host_x: True
    debian:
      image: appimagecrafters/tests-envdebian-stable
      command: ./AppRun
      use_host_x: True
AppImage:
  arch: x86_64
  update-information: guess
```

Recipe using packages from official Debian repositories

- Prepare you **appimage-builder** recipe, in a file named "program.yml"

```
# appimage-builder recipe, for details see:
# https://appimage-builder.readthedocs.io
version: 1
AppDir:
  path: ./AppDir
  app_info:
    id: program     # Name of the .desktop file
    name: program   # Name of the program
    icon: program   # Name of the icon file, no extention required
    version: 1.2.12
    exec: usr/bin/program
    exec_args: $@
  apt:
    arch: amd64
    sources:
      # One sourceline instruction for each repository to use to search for packages:
      # In this example official Debian repositories:
      #   - bookworkm: the official Debian 12 main repository
      #   - bookworkm-backports: the official Debian 12 backports main repository
      # Each repository requires to GPG signed, for some reasons the script main fails,
      # and it is declare the GPG key to use using the 'signed-by=/file-path/key-file.gpg' option:
      - sourceline: 'deb [arch=amd64, signed-by=/usr/share/keyrings/debian-archive-bookworm-automatic.gpg]
                     http://deb.debian.org/debian bookworm main'
      - sourceline: 'deb [arch=amd64, signed-by=/usr/share/keyrings/debian-archive-bookworm-automatic.gpg]
                     http://deb.debian.org/debian bookworm-backports main'
    # Package and dependencies to be included:
    include:
      - program
      - program-data
      - libgtk-3-0
      - libxml2
      - libpangoft2-1.0-0
      - libglu1-mesa
      - libepoxy0
      - libavutil57
      - libavcodec59
      - libavformat59
      - libswscale6
AppImage:
  arch: x86_64
  update-information: guess
```

The `apt:` section describes packages to download using the **apt** command, if you are not using a Debian based Linux.
Repositories must be GPG signed, however **appimage-builder** has issues to download and install keys during the process. It is safer to install the key(s) before hand manualy and then declare the key to use in the recipe file using the `signed-by=/file-path/key.gpg` instruction.

Build the appimage using appimage-builder and the recipe

To build the appimpage use:

```
user@localhost ~/program-1.2.12]$ appimage-builder --recipe program.yml
```

appimage-builder will download and install all required packages in a new "AppDir" directory, and will create an appimage in the active directory with the name "**program**-**version**-**arch**.AppImage" where the values for **program**, **version** and **arch** come from the corresponding fields in the recipe file.

Conclusion

I have been a Linux user for almost 25 year, and a developer for about the same period.
Back in my beginner's days what really impressed me the most about the Linux ecosystem was the extremely convenient way to install softwares: using on-line repositories and automated procedures.
It was confidential then, but nowadays thanks to massive usage of smartphones, this has been widely popularised by the words "app store" or "magasin d'applications" (in French). For years, and even as impressed as I was by the whole packaging process, I sadly though that all this was out of my league ... until I decided to give it a try.

I hope that this manual helped you understand that this is not out of your league !

APPENDIX A

BUILD SYSTEM FILES

A.1 The GNU Autotools files

A.1.1 The file: `configure.ac`

```
AC_PREREQ(2.59)
m4_define(major_version, 1)
m4_define(minor_version, 2)
m4_define(patch_version, 12)
m4_define(version, major_version.minor_version.patch_version)
m4_define(bug_email, your.email@host.eu)
m4_define(tar_name, program)
m4_define(project_url, https://www.program.com)
AC_INIT([prog], [version], [bug_email], [tar_name], [project_url])
AM_INIT_AUTOMAKE
AC_DEFINE([MAJOR_VERSION], [major_version], [Program major version])
AC_SUBST(MAJOR_VERSION, major_version)
AC_DEFINE([MINOR_VERSION], [minor_version], [Program minor version])
AC_SUBST(MINOR_VERSION, minor_version)
AC_DEFINE([PATCH_VERSION], [patch_version], [Program patch version])
AC_SUBST(PATCH_VERSION, patch_version)
AC_CHECK_PROG([PKG_CONFIG], [pkg-config], [yes], [no])
AC_CHECK_PROG([UP_MIME], [update-mime-database], [yes], [no])
AC_CHECK_PROG([UP_DESKTOP], [update-desktop-database], [yes], [no])
AC_CHECK_PROG([UP_APPSTREAM], [appstream-util], [yes], [no])

dnl Definig the macro AX_CHECK_COMPILE_FLAG:
AC_DEFUN([AX_CHECK_COMPILE_FLAG],
     [AC_PREREQ(2.64) dnl for _AC_LANG_PREFIX and AS_VAR_IF
     AS_VAR_PUSHDEF([CACHEVAR], [ax_cv_check_[]_AC_LANG_ABBREV[]flags_$4_$1])
     AC_CACHE_CHECK([whether _AC_LANG compiler accepts $1], CACHEVAR, [
          ax_check_save_flags=$[]_AC_LANG_PREFIX[]FLAGS
          _AC_LANG_PREFIX[]FLAGS="$[]_AC_LANG_PREFIX[]FLAGS $4 $1"
          AC_COMPILE_IFELSE([m4_default([$5],[AC_LANG_PROGRAM()])],
               [AS_VAR_SET(CACHEVAR,[yes])],
               [AS_VAR_SET(CACHEVAR,[no])])
          _AC_LANG_PREFIX[]FLAGS=$ax_check_save_flags])
     AS_VAR_IF(CACHEVAR,yes,
          [m4_default([$2], :)],
          [m4_default([$3], :)])
     AS_VAR_POPDEF([CACHEVAR])
])dnl AX_CHECK_COMPILE_FLAG
AC_PROG_CC
AX_CHECK_COMPILE_FLAG([$CFLAGS])
AC_FC_WRAPPERS
AC_LANG_PUSH([Fortran])
AC_PROG_FC([xlf95 fort ifort ifc f95 g95 pgf95 lf95 xlf90 f90 pgf90 gfortran],
           [90])
AC_FC_SRCEXT(f90, FCFLAGS_f90="$FCFLAGS_f90 $FCFLAGS",
             AC_MSG_ERROR([Err. comp. .f90]))
AC_FC_SRCEXT(F90, FCFLAGS_F90="$FCFLAGS_F90 $FCFLAGS",
             AC_MSG_ERROR([Err. comp. .F90]))
AX_CHECK_COMPILE_FLAG([$FCFLAGS])
AC_FC_LIBRARY_LDFLAGS
AC_FC_FREEFORM
AC_LANG_POP([Fortran])

PKG_CHECK_MODULES([LIBXML2], [libxml-2.0 >= 2.4.0])
PKG_CHECK_MODULES([GLU], [glu])
PKG_CHECK_MODULES([EPOXY], [epoxy])
PKG_CHECK_MODULES([FFMPEG], [libavcodec libavformat libavutil libswscale])
AC_MSG_CHECKING([the GTK version to use])
AC_ARG_WITH([gtk],
  [AS_HELP_STRING([--with-gtk=3|4], [the GTK version to use (default: 3)])],
  [case "$with_gtk" in
       3|4) ;;
       *) AC_MSG_ERROR([invalid GTK version specified]) ;;
         esac],
  [with_gtk=3])
AC_MSG_RESULT([$with_gtk])
AM_CONDITIONAL([GTK3], [test "$with_gtk" == "3"])
case "$with_gtk" in
  3) gtk_version="gtk+-3.0"
     gtk_version_number="3.16"
     ;;
  4) gtk_version="gtk4"
     gtk_version_number="4.6"
     ;;
esac
PKG_CHECK_MODULES([LGTK], [$gtk_version >= $gtk_version_number])

AC_CONFIG_FILES([
  Makefile
  src/Makefile
])

AC_OUTPUT
```

A.1.2 The file: `Makefile.am`

```
prog_datadir = $(DESTDIR)$(datadir)
prog_docdir = $(DESTDIR)$(docdir)
prog_mandir = $(DESTDIR)$(mandir)
prog_pkgdatadir = $(DESTDIR)$(pkgdatadir)
prog_desktopdir = $(prog_pkgdatadir)/applications
prog_metadir = $(prog_pkgdatadir)/metainfo
prog_mimedir = $(prog_pkgdatadir)/mime/packages
prog_iconsdir = $(prog_pkgdatadir)/pixmaps
SUBDIRS = src
prog_docdir = $(docdir)
prog_doc_DATA = README.md AUTHORS ChangeLog
prog_mandir = $(mandir)/man1/
prog_man_DATA = program.1.gz
install-data-local:
     if test -d $(srcdir)/data; then \
          $(mkinstalldirs) $(prog_pkgdatadir)/data; \
          for data in $(srcdir)/data/*; do \
               if test -f $$data; then \
                    $(INSTALL_DATA) $data $(prog_pkgdatadir)/data; \
               fi \
          done \
     fi
     if test -d $(srcdir)/pickups; then \
          $(mkinstalldirs) $(prog_pkgdatadir)/pixmaps; \
          for pixmap in $(srcdir)/pixmaps/*; do \
               if test -f $$pixmap; then \
                    $(INSTALL_DATA) $$pixmap $(prog_pkgdatadir)/pixmaps; \
               else \
                    $(mkinstalldirs) $(prog_pkgdatadir)/$$pixmap ; \
                    for pixma in $$pixmap/*; do \
                         if test -f $$pixma; then \
                              $(INSTALL_DATA) $$pixma $(prog_pkgdatadir)/$$pixmap; \
                         fi \
                    done \
               fi \
          done \
     fi
     if [ ! -d $(prog_iconsdir) ]; then \
          $(mkinstalldirs) $(prog_iconsdir); \
     fi
     $(INSTALL_DATA) $(srcdir)/metadata/icons/program.svg $(prog_iconsdir)
     $(INSTALL_DATA) $(srcdir)/metadata/icons/program-project.svg $(prog_iconsdir)
     $(INSTALL_DATA) $(srcdir)/metadata/icons/program-workspace.svg $(prog_iconsdir)
     if [ ! -d $(prog_mimedir) ]; then \
          $(mkinstalldirs) $(prog_mimedir); \
     fi
     $(INSTALL_DATA) $(srcdir)/metadata/mime/program.xml $(prog_mimedir)
     if [ ! -d $(prog_metadir) ]; then \
          mkdir -p $(prog_metadir); \
     fi
     $(INSTALL_DATA) $(srcdir)/metadata/com.program.www.appdata.xml $(prog_metadir)
     if [ ! -d $(prog_desktopdir) ]; then \
          mkdir -p $(prog_desktopdir); \
     fi
     appstream-util validate-relax --nonet $(prog_metadir)/com.program.www.appdata.xml
     desktop-file-install --vendor="" \
          --dir=$(prog_desktopdir) -m 644 \
          $(prog_desktopdir)/program.desktop
     touch -c $(prog_iconsdir)
     if [ -u `which gtk-update-icon-cache` ]; then \
          gtk-update-icon-cache -q $(prog_iconsdir); \
     fi
     if [ -z "$(DESTDIR)" ]; then \
          update-desktop-database $(prog_desktopdir) &> /dev/null || :; \
          update-mime-database $(prog_datadir)/mime &> /dev/null || :; \
     fi
uninstall-local:
     -rm -rf $(prog_pkgdatadir)/data/*
     -rmdir $(prog_pkgdatadir)/data
     -rm -rf $(prog_pkgdatadir)/pixmaps/*
     -rmdir $(prog_pkgdatadir)/pixmaps
     -rm -rf $(prog_pkgdatadir)/pixmaps/*
     -rmdir $(prog_pkgdatadir)
     -rmdir $(prog_docdir)
     -rm -rf $(prog_iconsdir)/program.svg
     -rm -rf $(prog_iconsdir)/program-project.svg
     -rm -rf $(prog_iconsdir)/program-workspace.svg
     -rm -f $(prog_desktopdir)/program.desktop
     -rm -f $(prog_metadir)/com.program.www.appdata.xml
     touch -c $(prog_iconsdir)
     if [ -u `which gtk-update-icon-cache` ]; then \
          gtk-update-icon-cache -q $(prog_iconsdir); \
     fi
     if [ -z "$(DESTDIR)" ]; then \
          update-desktop-database $(prog_desktopdir) &> /dev/null || :; \
          update-mime-database $(prog_datadir)/mime &> /dev/null || :; \
     fi
```

A.1.3 The file: `src/Makefile.am`

```
bin_PROGRAMS = prog

prog_LDADD = $(LGTK_LIBS) $(LIBXML2_LIBS) $(PANGOFT2_LIBS) $(FFMPEG_LIBS) $(GLU_LIBS) $(EPOXY_LIBS)

LIB_CFLAGS = $(LGTK_CFLAGS) $(LIBXML2_CFLAGS) $(PANGOFT2_CFLAGS) $(FFMPEG_CFLAGS) $(GLU_CFLAGS) $(EPOXY_CFLAGS)

if GTK3
  GTK_VERSION=-DGTK3
else
  GTK_VERSION=-DGTK4
endi

if OPENMP
  OpenMP_FLAGS=-DOPENMP $(OPENMP_CFLAGS)
else
  OpenMP_FLAGS=
endiff

if LINUX
  OS=-DOPENMP
else
  if WINDOWS
    OS=-DWINDOWS
  else
    OS=-DOSX
  endif
endif

AM_LDFLAGS = $(OpenMP_FLAGS)
AM_CPPFLAGS = $(GTK_VERSION) $(OpenMP_FLAGS) $(OS)
AM_FFLAGS = $(OpenMP_FLAGS)
AM_CFLAGS = $(LIB_CFLAGS) $(OpenMP_FLAGS)

prog_fortran_files = file-1.f90 file-2.f90
prog_fortran_modules = mod.f90

prog_fortran = $(prog_fortran_modules) $(prog_fortran_files)
$(patsubst %.F90,%.o,$(prog_fortran_files)): $(patsubst %.F90,%.o,$(prog_fortran_modules))

prog_c = main.c gui.c

clean:
    -rm -f *.mod
    -rm -f */*.o

prog_SOURCES = $(prog_fortran) $(prog_c)
```

A.2 The CMake file(s)

A.2.1 The file: `CMakeList.txt`

```
# CMake minimum configuration
cmake_minimum_required (VERSION 3.10)

# Project version and details
project(prog VERSION 1.2.12)
project(prog DESCRIPTION "A tool box")
project(prog HOMEPAGE_URL "https://www.program.com")
project(prog LANGUAGES C Fortran)

# Checking for pkg-config
find_package(PkgConfig REQUIRED)
# Checking for gtk+3.0:
pkg_check_modules(GTK3 REQUIRED IMPORTED_TARGET gtk3)
# Checking for libxml-2.0:
pkg_check_modules(LIBXML2 REQUIRED IMPORTED_TARGET libxml-2.0)
# Checking for libavcodec, and other FFMPEG based libraries:
pkg_check_modules(LIBAVUTIL REQUIRED IMPORTED_TARGET libavutil)
pkg_check_modules(LIBAVCODEC REQUIRED IMPORTED_TARGET libavcodec)
pkg_check_modules(LIBAVFORMAT REQUIRED IMPORTED_TARGET libavformat)
pkg_check_modules(LIBSWSCALE REQUIRED IMPORTED_TARGET libswscale)
# Checking for epoxy:
pkg_check_modules(EPOXY REQUIRED IMPORTED_TARGET epoxy)

# Custom compiler flags
set(CMAKE_C_FLAGS "-O3")
set(CMAKE_Fortran_FLAGS "-O3")
# Compiler variables
find_package(OpenMP)
if (OpenMP_FOUND)
  set(CMAKE_C_FLAGS ${OpenMP_C_FLAGS})
  set(CMAKE_Fortran_FLAGS ${OpenMP_Fortran_FLAGS})
endif()
# Operating system:
if (LINUX)
  add_compile_definitions(LINUX)
elseif (WINDOWS)
  add_compile_definitions(WINDOWS)
elseif (APPLE)
  add_compile_definitions(OSX)
endif()

# Sources
file(GLOB C_SRC RELATIVE ${CMAKE_SOURCE_DIR} "src/*.c")
file(GLOB F_SRC RELATIVE ${CMAKE_SOURCE_DIR} "src/*.f90")
# Build rules
add_executable(prog ${C_SRC} ${F_SRC})
# Header files include directories
target_include_directories(prog PUBLIC src)
# Link with dependencies
target_link_libraries(prog PUBLIC PkgConfig::GTK3)
target_link_libraries(prog PUBLIC PkgConfig::LIBXML2)
target_link_libraries(prog PUBLIC PkgConfig::LIBAVUTIL)
target_link_libraries(prog PUBLIC PkgConfig::LIBAVCODEC)
target_link_libraries(prog PUBLIC PkgConfig::LIBAVFORMAT)
target_link_libraries(prog PUBLIC PkgConfig::LIBSWSCALE)
target_link_libraries(prog PUBLIC PkgConfig::EPOXY)

# First import the GNU installation directory variables
include(GNUInstallDirs)
# To install the main binary file:
install(PROGRAMS prog DESTINATION ${CMAKE_INSTALL_BINDIR})
# To install a directory, including all its content, recursively:
install(DIRECTORY data DESTINATION ${CMAKE_INSTALL_DATADIR}/prog PATTERN "data/*")
install(DIRECTORY pixmaps DESTINATION ${CMAKE_INSTALL_DATADIR}/prog PATTERN  "pixmaps/*")
install(DIRECTORY metadata/icons DESTINATION ${CMAKE_INSTALL_DATADIR}/pixmaps
                                 PATTERN "metadata/icons/*.svg")
# To install specific files:
install(FILES metadata/com.program.www.appdata.xml DESTINATION ${CMAKE_INSTALL_DATADIR}/metainfo)
install(FILES metadata/program-mime.xml DESTINATION ${CMAKE_INSTALL_DATADIR}/mime/packages)
install(FILES metadata/program.desktop DESTINATION ${CMAKE_INSTALL_DATADIR}/applications)

# Running the post installation script:
set(INSTALL_DIR "${CMAKE_INSTALL_FULLDATADIR}")
install(CODE "execute_process(COMMAND ./post-install.sh ${INSTALL_DIR})")
```

A.2.2 The script: `post-install.sh`

```
#!/bin/bash

# Retrieving the installation directory as first argument:
INSTALL_DIR=$1

# Setting up variables
prog_metadir=${INSTALL_DIR}/metainfo
prog_desktopdir=${INSTALL_DIR}/applications
prog_iconsdir=${INSTALL_DIR}/pixmaps

# Updating metadata information
appstream-util validate-relax --nonet ${prog_metadir}/com.program.www.appdata.xml

# Installing desktop file
desktop-file-install --vendor=""  \
                     --dir=${prog_desktopdir} \
                      -m 644 ${prog_desktopdir}/program.desktop

# Updating icon file cache
touch --no-create ${prog_iconsdir}
if [ -u 'which gtk-update-icon-cache' ]; then
  gtk-update-icon-cache -q ${prog_iconsdir}
fi

# Updating desktop database information
update-desktop-database ${prog_desktopdir} &> /dev/null || :;

# Updating MIME database
update-mime-database ${prog_datadir}/mime &> /dev/null || :;
```

A.3 The meson file(s)

A.3.1 The file: `meson.build`

```
project('prog', 'c', 'fortran', license : 'AGPL-3.0-or-later', version : '1.2.12')

# Checking for libxml-2.0:
xml = dependency('libxml-2.0')
# Checking for libavcodec, and other FFMPEG based libraries:
avutil = dependency('libavutil')
avcodec = dependency('libavcodec')
avformat = dependency('libavformat')
swscale = dependency('libswscale')
# Declaration of a variable to store all FFMPEG dependencies:
ffmpeg = [avutil, avcodec, avformat, swscale]
# Checking for epoxy:
epoxy = dependency('epoxy')

gtk_version = get_option('gtk')
if gtk_version == 3
  # Default option: checking for gtk+3.0:
  gtk = dependency('gtk+-3.0')
  add_project_arguments('-DGTK3', language : 'c')
else
  # Using -Dgtk=4 on the meson configuration command line
  gtk = dependency('gtk4')
  add_project_arguments('-DGTK4', language : 'c')
endif

# Declaration of a variable to store, and easily re-use, all dependencies:
all_deps = [gtk, xml, glu, epoxy, ffmpeg]

use_openmp = get_option('openmp')
if use_openmp
  omp = dependency('openmp', required : false)
  if omp.found ()
    all_deps = [all_deps, omp]
    add_project_arguments('-DOPENMP', language : 'c')
    add_project_arguments('-DOPENMP', language : 'fortran')
  else
    message('OpenMP not found: building serial version')
  endif
endif

system = host_machine.system()
if system == 'linux'
  add_project_arguments('-DLINUX', language : 'c')
elif system == 'windows'
  add_project_arguments('-DWINDOWS', language : 'c')
elif system == 'darwin'
  add_project_arguments('-DOSX', language : 'c')
endif

# Sources
src_c = ['src/main.c', 'src/gui.c']
src_f = ['src/file-1.f90', 'src/file-2.f90']
src_all = [src_c, src_f]

# Header files include directories
inc_dir = include_directories('.', 'src')

# Build rules
project_name = meson.project_name()
executable(project_name, sources : src_all, include_directories : inc_dir, dependencies : all_deps, install : true)

# To install specific files:
install_dat('metadata/program-mime.xml', install_dir : data_dir / 'mime/packages')
install_dat('metadata/com.program.www.appdata.xml', install_dir : data_dir / 'metainfo')
install_dat('metadata/program.desktop', install_dir : data_dir / 'applications')
install_dat('ChangeLog', install_dir : data_dir / 'doc/prog')
install_dat('COPYING', install_dir : data_dir / 'doc/prog')

# To install a directory, including all its content, recursively:
install_subdir('data', install_dir : data_dir / 'prog')
install_subdir('pixmaps', install_dir : data_dir / 'prog')

# Running the post installation script:
project_prefix = get_option('prefix')
meson.add_install_script('post_install.sh', project_prefix)
```

A.3.2 The associated file: `meson.options` (or `meson_options.txt`)

```
option('gtk', type: 'integer', value: 3, description: 'Version of the GTK library')
option('openmp', type: 'boolean', value: true, description: 'Use OpenMP')
```

APPENDIX B

THE PACKAGING FILES

B.1 ". spec" file for RPM packaging

```
Name:      prog
%global upname Program
Version:   1.2.12
Release:   3%{?dist}
Summary:   An nice program
License:   AGPL-3.0-or-later
Source0:   https://github.com/Author/%{upname}/archive/refs/tags/v%{version}.tar.gz
# Source1:   ./v%%{version}.tar.gz.asc
# Source2:   %%{name}.gpg
URL:       https://www.%{name}.com/

BuildRequires: make
BuildRequires: automake
BuildRequires: autoconf
BuildRequires: pkgconf-pkg-config
BuildRequires: gcc
BuildRequires: gcc-gfortran
BuildRequires: libgfortran
BuildRequires: pkgconfig(gtk+-3.0)
BuildRequires: pkgconfig(libxml-2.0)
BuildRequires: pkgconfig(glu)
BuildRequires: pkgconfig(epoxy)
BuildRequires: pkgconfig(libavutil)
BuildRequires: pkgconfig(libavcodec)
BuildRequires: pkgconfig(libavformat)
BuildRequires: pkgconfig(libswscale)
BuildRequires: desktop-file-utils
BuildRequires: libappstream-glib
# BuildRequires: gnupg2
Requires: gtk3
Requires: mesa-libGLU

%prep
# %%{gpgverify} --keyring='%%{SOURCE2}' --signature='%%{SOURCE1}' --data='%%{SOURCE0}'
%autosetup -n %{upname}-%{version}

%build
%configure
%make_build

%install
%make_install

%check
desktop-file-validate %{buildroot}/%{_datadir}/applications/%{name}.desktop
appstream-util validate-relax --nonet %{buildroot}%{_metainfodir}/com.%{name}.www.appdata.xml

%files
%license COPYING
%{_bindir}/%{name}
%{_datadir}/%{name}
%{_datadir}/doc/%{name}
%{_mandir}/man1/%{name}.1*
%{_datadir}/applications/%{name}.desktop
%{_metainfodir}/com.%{name}.www.appdata.xml
%{_datadir}/mime/packages/%{name}.xml
%{_datadir}/pixmaps/%{name}.svg
%{_datadir}/pixmaps/%{name}-program.svg
%{_datadir}/pixmaps/%{name}-workspace.svg

%changelog
* Thu Mar 30 2023 Your Name <your.email@host.eu> - 1.2.12-3
- Revised package: what you did here.
```

B.2 "debian" directory for Debian packaging

B.2.1 The "copyright" file

```
Format: https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: program
Upstream-Contact: your.email@host.eu
  Your Name <your.email@host.eu>
Source: https://github.com/Author/Program

Files:     *
Copyright: 2023-2024 Your Name
License:   AGPL-3.0-or-later

Files:     Makefile.in
           src/Makefile.in
Copyright: 1994-2021 Free Software Foundation, Inc.
License:   FSFULLR

Files:     aclocal.m4
Copyright: 1996-2020 Free Software Foundation, Inc.
           2004 Scott James Remnant <scott@netsplit.com>.
           2012-2015 Dan Nicholson <dbn.lists@gmail.com>
License:   FSFULLR and GPL-2.0-or-later

Files:     compile
           depcomp
           missing
Copyright: 1996-2020 Free Software Foundation, Inc.
License:   GPL-2.0-or-later with Autoconf-data exception

Files:     config.guess
           config.sub
Copyright: 1992-2018 Free Software Foundation, Inc.
License:   GPL-3.0-or-later

Files:     configure
           autom4te.cache/*
Copyright: 1992-2021 Free Software Foundation, Inc.
License:   FSFUL

Files:     install-sh
Copyright: 1994 X Consortium
License:   Expat-X
Comment:   FSF changes to this file are in the public domain.

Files:     INSTALL
Copyright: 1994-1996, 1999-2002, 2004-2016 Free Software Foundation, Inc.
License:   FSFULLR

Files:     metadata/com.program.www.appdata.xml
Copyright: 2023-2024 Your Name
License:   FSFAP

Files:     debian/*
Copyright: 2023-2024 your Name
License:   AGPL-3.0-or-later
```

```
License: AGPL-3.0-or-later
                    GNU AFFERO GENERAL PUBLIC LICENSE
                       Version 3, 19 November 2007
 .
 Copyright (C) 2007 Free Software Foundation, Inc. <https://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.
 .
                            Preamble
 .
 The GNU Affero General Public License is a free, copyleft license for
 software and other kinds of works, specifically designed to ensure
 cooperation with the community in the case of network server software.
 .
 The licenses for most software and other practical works are designed
 to take away your freedom to share and change the works.  By contrast,
 our General Public Licenses are intended to guarantee your freedom to
 share and change all versions of a program--to make sure it remains free
 software for all its users.
 .
  When we speak of free software, we are referring to freedom, not
 price.  Our General Public Licenses are designed to make sure that you
 have the freedom to distribute copies of free software (and charge for
 them if you wish), that you receive source code or can get it if you
 want it, that you can change the software or use pieces of it in new
 free programs, and that you know you can do these things.
 .
  Developers that use our General Public Licenses protect your rights
 with two steps: (1) assert copyright on the software, and (2) offer
 you this License which gives you legal permission to copy, distribute
 and/or modify the software.
 .
  A secondary benefit of defending all users' freedom is that
 improvements made in alternate versions of the program, if they
 receive widespread use, become available for other developers to
 incorporate.  Many developers of free software are heartened and
 encouraged by the resulting cooperation.  However, in the case of
 software used on network servers, this result may fail to come about.
 The GNU General Public License permits making a modified version and
 letting the public access it on a server without ever releasing its
 source code to the public.
 .
  The GNU Affero General Public License is designed specifically to
 ensure that, in such cases, the modified source code becomes available
 to the community.  It requires the operator of a network server to
 provide the source code of the modified version running there to the
 users of that server.  Therefore, public use of a modified version, on
 a publicly accessible server, gives the public access to the source
 code of the modified version.
 .
  An older license, called the Affero General Public License and
 published by Affero, was designed to accomplish similar goals.  This is
 a different license, not a version of the Affero GPL, but Affero has
 released a new version of the Affero GPL which permits relicensing under
 this license.
 .
  The precise terms and conditions for copying, distribution and
 modification follow.
```

```
.
                       TERMS AND CONDITIONS
.
  0. Definitions.
.
  "This License" refers to version 3 of the GNU Affero General Public License.
.
  "Copyright" also means copyright-like laws that apply to other kinds of
 works, such as semiconductor masks.
.
 "The Program" refers to any copyrightable work licensed under this
License.  Each licensee is addressed as "you".  "Licensees" and
"recipients" may be individuals or organizations.
.
  To "modify" a work means to copy from or adapt all or part of the work
in a fashion requiring copyright permission, other than the making of an
exact copy.  The resulting work is called a "modified version" of the
earlier work or a work "based on" the earlier work.
.
  A "covered work" means either the unmodified Program or a work based
on the Program.
.
  To "propagate" a work means to do anything with it that, without
permission, would make you directly or secondarily liable for
infringement under applicable copyright law, except executing it on a
computer or modifying a private copy.  Propagation includes copying,
distribution (with or without modification), making available to the
public, and in some countries other activities as well.
.
  To "convey" a work means any kind of propagation that enables other
parties to make or receive copies.  Mere interaction with a user through
a computer network, with no transfer of a copy, is not conveying.
.
  An interactive user interface displays "Appropriate Legal Notices"
to the extent that it includes a convenient and prominently visible
feature that (1) displays an appropriate copyright notice, and (2)
tells the user that there is no warranty for the work (except to the
extent that warranties are provided), that licensees may convey the
work under this License, and how to view a copy of this License.  If
the interface presents a list of user commands or options, such as a
menu, a prominent item in the list meets this criterion.
.
  1. Source Code.
.
  The "source code" for a work means the preferred form of the work
for making modifications to it.  "Object code" means any non-source
form of a work.
.
  A "Standard Interface" means an interface that either is an official
standard defined by a recognized standards body, or, in the case of
interfaces specified for a particular programming language, one that
is widely used among developers working in that language.
.
  The "System Libraries" of an executable work include anything, other
than the work as a whole, that (a) is included in the normal form of
packaging a Major Component, but which is not part of that Major
Component, and (b) serves only to enable use of the work with that
Major Component, or to implement a Standard Interface for which an
implementation is available to the public in source code form.  A
"Major Component", in this context, means a major essential component
(kernel, window system, and so on) of the specific operating system
(if any) on which the executable work runs, or a compiler used to
produce the work, or an object code interpreter used to run it.
.
```

```
  The "Corresponding Source" for a work in object code form means all
the source code needed to generate, install, and (for an executable
work) run the object code and to modify the work, including scripts to
control those activities.  However, it does not include the work's
System Libraries, or general-purpose tools or generally available free
programs which are used unmodified in performing those activities but
which are not part of the work.  For example, Corresponding Source
includes interface definition files associated with source files for
the work, and the source code for shared libraries and dynamically
linked subprograms that the work is specifically designed to require,
such as by intimate data communication or control flow between those
subprograms and other parts of the work.
.
  The Corresponding Source need not include anything that users
can regenerate automatically from other parts of the Corresponding
Source.
.
  The Corresponding Source for a work in source code form is that
same work.
.
  2. Basic Permissions.
.
  All rights granted under this License are granted for the term of
copyright on the Program, and are irrevocable provided the stated
conditions are met.  This License explicitly affirms your unlimited
permission to run the unmodified Program.  The output from running a
covered work is covered by this License only if the output, given its
content, constitutes a covered work.  This License acknowledges your
rights of fair use or other equivalent, as provided by copyright law.
.
  You may make, run and propagate covered works that you do not
convey, without conditions so long as your license otherwise remains
in force.  You may convey covered works to others for the sole purpose
of having them make modifications exclusively for you, or provide you
with facilities for running those works, provided that you comply with
the terms of this License in conveying all material for which you do
not control copyright.  Those thus making or running the covered works
for you must do so exclusively on your behalf, under your direction
and control, on terms that prohibit them from making any copies of
your copyrighted material outside their relationship with you.
.
  Conveying under any other circumstances is permitted solely under
the conditions stated below.  Sublicensing is not allowed; section 10
makes it unnecessary.
.
  3. Protecting Users' Legal Rights From Anti-Circumvention Law.
.
  No covered work shall be deemed part of an effective technological
measure under any applicable law fulfilling obligations under article
11 of the WIPO copyright treaty adopted on 20 December 1996, or
similar laws prohibiting or restricting circumvention of such
measures.
.
  When you convey a covered work, you waive any legal power to forbid
circumvention of technological measures to the extent such circumvention
is effected by exercising rights under this License with respect to
the covered work, and you disclaim any intention to limit operation or
modification of the work as a means of enforcing, against the work's
users, your or third parties' legal rights to forbid circumvention of
technological measures.
```

```
.
  4. Conveying Verbatim Copies.
.
  You may convey verbatim copies of the Program's source code as you
receive it, in any medium, provided that you conspicuously and
appropriately publish on each copy an appropriate copyright notice;
keep intact all notices stating that this License and any
non-permissive terms added in accord with section 7 apply to the code;
keep intact all notices of the absence of any warranty; and give all
recipients a copy of this License along with the Program.
.
  You may charge any price or no price for each copy that you convey,
and you may offer support or warranty protection for a fee.
.
  5. Conveying Modified Source Versions.
.
  You may convey a work based on the Program, or the modifications to
produce it from the Program, in the form of source code under the
terms of section 4, provided that you also meet all of these conditions:
.
    a) The work must carry prominent notices stating that you modified
    it, and giving a relevant date.
.
    b) The work must carry prominent notices stating that it is
    released under this License and any conditions added under section
    7.  This requirement modifies the requirement in section 4 to
    "keep intact all notices".
.
    c) You must license the entire work, as a whole, under this
    License to anyone who comes into possession of a copy.  This
    License will therefore apply, along with any applicable section 7
    additional terms, to the whole of the work, and all its parts,
    regardless of how they are packaged.  This License gives no
    permission to license the work in any other way, but it does not
    invalidate such permission if you have separately received it.
.
    d) If the work has interactive user interfaces, each must display
    Appropriate Legal Notices; however, if the Program has interactive
    interfaces that do not display Appropriate Legal Notices, your
    work need not make them do so.
.
  A compilation of a covered work with other separate and independent
works, which are not by their nature extensions of the covered work,
and which are not combined with it such as to form a larger program,
in or on a volume of a storage or distribution medium, is called an
"aggregate" if the compilation and its resulting copyright are not
used to limit the access or legal rights of the compilation's users
beyond what the individual works permit.  Inclusion of a covered work
in an aggregate does not cause this License to apply to the other
parts of the aggregate.
.
  6. Conveying Non-Source Forms.
.
  You may convey a covered work in object code form under the terms
of sections 4 and 5, provided that you also convey the
machine-readable Corresponding Source under the terms of this License,
in one of these ways:
.
    a) Convey the object code in, or embodied in, a physical product
    (including a physical distribution medium), accompanied by the
    Corresponding Source fixed on a durable physical medium
    customarily used for software interchange.
```

```
.
    b) Convey the object code in, or embodied in, a physical product
    (including a physical distribution medium), accompanied by a
    written offer, valid for at least three years and valid for as
    long as you offer spare parts or customer support for that product
    model, to give anyone who possesses the object code either (1) a
    copy of the Corresponding Source for all the software in the
    product that is covered by this License, on a durable physical
    medium customarily used for software interchange, for a price no
    more than your reasonable cost of physically performing this
    conveying of source, or (2) access to copy the
    Corresponding Source from a network server at no charge.
.
    c) Convey individual copies of the object code with a copy of the
    written offer to provide the Corresponding Source.  This
    alternative is allowed only occasionally and noncommercially, and
    only if you received the object code with such an offer, in accord
    with subsection 6b.
.
    d) Convey the object code by offering access from a designated
    place (gratis or for a charge), and offer equivalent access to the
    Corresponding Source in the same way through the same place at no
    further charge.  You need not require recipients to copy the
    Corresponding Source along with the object code.  If the place to
    copy the object code is a network server, the Corresponding Source
    may be on a different server (operated by you or a third party)
    that supports equivalent copying facilities, provided you maintain
    clear directions next to the object code saying where to find the
    Corresponding Source.  Regardless of what server hosts the
    Corresponding Source, you remain obligated to ensure that it is
    available for as long as needed to satisfy these requirements.
.
    e) Convey the object code using peer-to-peer transmission, provided
    you inform other peers where the object code and Corresponding
    Source of the work are being offered to the general public at no
    charge under subsection 6d.
.
  A separable portion of the object code, whose source code is excluded
from the Corresponding Source as a System Library, need not be
included in conveying the object code work.
.
  A "User Product" is either (1) a "consumer product", which means any
tangible personal property which is normally used for personal, family,
or household purposes, or (2) anything designed or sold for incorporation
into a dwelling.  In determining whether a product is a consumer product,
doubtful cases shall be resolved in favor of coverage.  For a particular
product received by a particular user, "normally used" refers to a
typical or common use of that class of product, regardless of the status
of the particular user or of the way in which the particular user
actually uses, or expects or is expected to use, the product.  A product
is a consumer product regardless of whether the product has substantial
commercial, industrial or non-consumer uses, unless such uses represent
the only significant mode of use of the product.
.
  "Installation Information" for a User Product means any methods,
procedures, authorization keys, or other information required to install
and execute modified versions of a covered work in that User Product from
a modified version of its Corresponding Source.  The information must
suffice to ensure that the continued functioning of the modified object
code is in no case prevented or interfered with solely because
modification has been made.
.
```

```
  If you convey an object code work under this section in, or with, or
specifically for use in, a User Product, and the conveying occurs as
part of a transaction in which the right of possession and use of the
User Product is transferred to the recipient in perpetuity or for a
fixed term (regardless of how the transaction is characterized), the
Corresponding Source conveyed under this section must be accompanied
by the Installation Information.  But this requirement does not apply
if neither you nor any third party retains the ability to install
modified object code on the User Product (for example, the work has
been installed in ROM).
.
  The requirement to provide Installation Information does not include a
requirement to continue to provide support service, warranty, or updates
for a work that has been modified or installed by the recipient, or for
the User Product in which it has been modified or installed.  Access to a
network may be denied when the modification itself materially and
adversely affects the operation of the network or violates the rules and
protocols for communication across the network.
.
  Corresponding Source conveyed, and Installation Information provided,
in accord with this section must be in a format that is publicly
documented (and with an implementation available to the public in
source code form), and must require no special password or key for
unpacking, reading or copying.
.
  7. Additional Terms.
.
  "Additional permissions" are terms that supplement the terms of this
License by making exceptions from one or more of its conditions.
Additional permissions that are applicable to the entire Program shall
be treated as though they were included in this License, to the extent
that they are valid under applicable law.  If additional permissions
apply only to part of the Program, that part may be used separately
under those permissions, but the entire Program remains governed by
this License without regard to the additional permissions.
.
  When you convey a copy of a covered work, you may at your option
remove any additional permissions from that copy, or from any part of
it.  (Additional permissions may be written to require their own
removal in certain cases when you modify the work.)  You may place
additional permissions on material, added by you to a covered work,
for which you have or can give appropriate copyright permission.
.
  Notwithstanding any other provision of this License, for material you
add to a covered work, you may (if authorized by the copyright holders of
that material) supplement the terms of this License with terms:
.
    a) Disclaiming warranty or limiting liability differently from the
    terms of sections 15 and 16 of this License; or
.
    b) Requiring preservation of specified reasonable legal notices or
    author attributions in that material or in the Appropriate Legal
    Notices displayed by works containing it; or
.
    c) Prohibiting misrepresentation of the origin of that material, or
    requiring that modified versions of such material be marked in
    reasonable ways as different from the original version; or
.
    d) Limiting the use for publicity purposes of names of licensors or
    authors of the material; or
.
    e) Declining to grant rights under trademark law for use of some
    trade names, trademarks, or service marks; or
.
```

```
    f) Requiring indemnification of licensors and authors of that
    material by anyone who conveys the material (or modified versions of
    it) with contractual assumptions of liability to the recipient, for
    any liability that these contractual assumptions directly impose on
    those licensors and authors.
.
  All other non-permissive additional terms are considered "further
restrictions" within the meaning of section 10.  If the Program as you
received it, or any part of it, contains a notice stating that it is
governed by this License along with a term that is a further
restriction, you may remove that term.  If a license document contains
a further restriction but permits relicensing or conveying under this
License, you may add to a covered work material governed by the terms
of that license document, provided that the further restriction does
not survive such relicensing or conveying.
.
  If you add terms to a covered work in accord with this section, you
must place, in the relevant source files, a statement of the
additional terms that apply to those files, or a notice indicating
where to find the applicable terms.
.
  Additional terms, permissive or non-permissive, may be stated in the
form of a separately written license, or stated as exceptions;
the above requirements apply either way.
.
  8. Termination.
.
  You may not propagate or modify a covered work except as expressly
provided under this License.  Any attempt otherwise to propagate or
modify it is void, and will automatically terminate your rights under
this License (including any patent licenses granted under the third
paragraph of section 11).
.
  However, if you cease all violation of this License, then your
license from a particular copyright holder is reinstated (a)
provisionally, unless and until the copyright holder explicitly and
finally terminates your license, and (b) permanently, if the copyright
holder fails to notify you of the violation by some reasonable means
prior to 60 days after the cessation.
.
  Moreover, your license from a particular copyright holder is
reinstated permanently if the copyright holder notifies you of the
violation by some reasonable means, this is the first time you have
received notice of violation of this License (for any work) from that
copyright holder, and you cure the violation prior to 30 days after
your receipt of the notice.
.
  Termination of your rights under this section does not terminate the
licenses of parties who have received copies or rights from you under
this License.  If your rights have been terminated and not permanently
reinstated, you do not qualify to receive new licenses for the same
material under section 10.
.
  9. Acceptance Not Required for Having Copies.
.
  You are not required to accept this License in order to receive or
run a copy of the Program.  Ancillary propagation of a covered work
occurring solely as a consequence of using peer-to-peer transmission
to receive a copy likewise does not require acceptance.  However,
nothing other than this License grants you permission to propagate or
modify any covered work.  These actions infringe copyright if you do
not accept this License.  Therefore, by modifying or propagating a
covered work, you indicate your acceptance of this License to do so.
.
```

```
  10. Automatic Licensing of Downstream Recipients.
.
  Each time you convey a covered work, the recipient automatically
receives a license from the original licensors, to run, modify and
propagate that work, subject to this License.  You are not responsible
for enforcing compliance by third parties with this License.
.
  An "entity transaction" is a transaction transferring control of an
organization, or substantially all assets of one, or subdividing an
organization, or merging organizations.  If propagation of a covered
work results from an entity transaction, each party to that
transaction who receives a copy of the work also receives whatever
licenses to the work the party's predecessor in interest had or could
give under the previous paragraph, plus a right to possession of the
Corresponding Source of the work from the predecessor in interest, if
the predecessor has it or can get it with reasonable efforts.
.
  You may not impose any further restrictions on the exercise of the
rights granted or affirmed under this License.  For example, you may
not impose a license fee, royalty, or other charge for exercise of
rights granted under this License, and you may not initiate litigation
(including a cross-claim or counterclaim in a lawsuit) alleging that
any patent claim is infringed by making, using, selling, offering for
sale, or importing the Program or any portion of it.
.
  11. Patents.
.
  A "contributor" is a copyright holder who authorizes use under this
License of the Program or a work on which the Program is based.  The
work thus licensed is called the contributor's "contributor version".
.
  A contributor's "essential patent claims" are all patent claims
owned or controlled by the contributor, whether already acquired or
hereafter acquired, that would be infringed by some manner, permitted
by this License, of making, using, or selling its contributor version,
but do not include claims that would be infringed only as a
consequence of further modification of the contributor version.  For
purposes of this definition, "control" includes the right to grant
patent sublicenses in a manner consistent with the requirements of
this License.
.
  Each contributor grants you a non-exclusive, worldwide, royalty-free
patent license under the contributor's essential patent claims, to
make, use, sell, offer for sale, import and otherwise run, modify and
propagate the contents of its contributor version.
.
  In the following three paragraphs, a "patent license" is any express
agreement or commitment, however denominated, not to enforce a patent
(such as an express permission to practice a patent or covenant not to
sue for patent infringement).  To "grant" such a patent license to a
party means to make such an agreement or commitment not to enforce a
patent against the party.
.
```

```
  If you convey a covered work, knowingly relying on a patent license,
and the Corresponding Source of the work is not available for anyone
to copy, free of charge and under the terms of this License, through a
publicly available network server or other readily accessible means,
then you must either (1) cause the Corresponding Source to be so
available, or (2) arrange to deprive yourself of the benefit of the
patent license for this particular work, or (3) arrange, in a manner
consistent with the requirements of this License, to extend the patent
license to downstream recipients.  "Knowingly relying" means you have
actual knowledge that, but for the patent license, your conveying the
covered work in a country, or your recipient's use of the covered work
in a country, would infringe one or more identifiable patents in that
country that you have reason to believe are valid.
.
  If, pursuant to or in connection with a single transaction or
arrangement, you convey, or propagate by procuring conveyance of, a
covered work, and grant a patent license to some of the parties
receiving the covered work authorizing them to use, propagate, modify
or convey a specific copy of the covered work, then the patent license
you grant is automatically extended to all recipients of the covered
work and works based on it.
.
  A patent license is "discriminatory" if it does not include within
the scope of its coverage, prohibits the exercise of, or is
conditioned on the non-exercise of one or more of the rights that are
specifically granted under this License.  You may not convey a covered
work if you are a party to an arrangement with a third party that is
in the business of distributing software, under which you make payment
to the third party based on the extent of your activity of conveying
the work, and under which the third party grants, to any of the
parties who would receive the covered work from you, a discriminatory
patent license (a) in connection with copies of the covered work
conveyed by you (or copies made from those copies), or (b) primarily
for and in connection with specific products or compilations that
contain the covered work, unless you entered into that arrangement,
or that patent license was granted, prior to 28 March 2007.
.
  Nothing in this License shall be construed as excluding or limiting
any implied license or other defenses to infringement that may
otherwise be available to you under applicable patent law.
.
  12. No Surrender of Others' Freedom.
.
  If conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License.  If you cannot convey a
covered work so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you may
not convey it at all.  For example, if you agree to terms that obligate you
to collect a royalty for further conveying from those to whom you convey
the Program, the only way you could satisfy both those terms and this
License would be to refrain entirely from conveying the Program.
.
  13. Remote Network Interaction; Use with the GNU General Public License.
.
  Notwithstanding any other provision of this License, if you modify the
Program, your modified version must prominently offer all users
interacting with it remotely through a computer network (if your version
supports such interaction) an opportunity to receive the Corresponding
Source of your version by providing access to the Corresponding Source
from a network server at no charge, through some standard or customary
means of facilitating copying of software.  This Corresponding Source
shall include the Corresponding Source for any work covered by version 3
of the GNU General Public License that is incorporated pursuant to the
following paragraph.
```

```
.
  Notwithstanding any other provision of this License, you have
permission to link or combine any covered work with a work licensed
under version 3 of the GNU General Public License into a single
combined work, and to convey the resulting work.  The terms of this
License will continue to apply to the part which is the covered work,
but the work with which it is combined will remain governed by version
3 of the GNU General Public License.
.
  14. Revised Versions of this License.
.
  The Free Software Foundation may publish revised and/or new versions of
the GNU Affero General Public License from time to time.  Such new versions
will be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.
.
  Each version is given a distinguishing version number.  If the
Program specifies that a certain numbered version of the GNU Affero General
Public License "or any later version" applies to it, you have the
option of following the terms and conditions either of that numbered
version or of any later version published by the Free Software
Foundation.  If the Program does not specify a version number of the
GNU Affero General Public License, you may choose any version ever published
by the Free Software Foundation.
.
  If the Program specifies that a proxy can decide which future
versions of the GNU Affero General Public License can be used, that proxy's
public statement of acceptance of a version permanently authorizes you
to choose that version for the Program.
.
  Later license versions may give you additional or different
permissions.  However, no additional obligations are imposed on any
author or copyright holder as a result of your choosing to follow a
later version.
.
  15. Disclaimer of Warranty.
.
  THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY
APPLICABLE LAW.  EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT
HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY
OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM
IS WITH YOU.  SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF
ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
.
  16. Limitation of Liability.
.
  IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS
THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY
GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE
USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF
DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD
PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS),
EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGES.
.
  17. Interpretation of Sections 15 and 16.
.
  If the disclaimer of warranty and limitation of liability provided
above cannot be given local legal effect according to their terms,
reviewing courts shall apply local law that most closely approximates
an absolute waiver of all civil liability in connection with the
Program, unless a warranty or assumption of liability accompanies a
copy of the Program in return for a fee.
```

```
 .
                    END OF TERMS AND CONDITIONS

License: FSFUL
 Copyright (C) 1992-1996, 1998-2012 Free Software Foundation, Inc.
 .
 This file is free software; the Free Software Foundation
 gives unlimited permission to copy, distribute and modify it.

License: FSFULLR
 Copyright 1996-2006 Free Software Foundation, Inc.
 .
 This file is free software; the Free Software Foundation gives
 unlimited permission to copy and/or distribute it, with or without
 modifications, as long as this notice is preserved.

License: GPL-2.0-or-later
 This program is free software; you can redistribute it and/or modify
 it under the terms of the GNU General Public License as published by
 the Free Software Foundation; either version 2, or (at your option)
 any later version.
 .
 This program is distributed in the hope that it will be useful,
 but WITHOUT ANY WARRANTY; without even the implied warranty of
 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 GNU General Public License for more details.
 .
 You should have received a copy of the GNU General Public License
 along with this program; if not, write to the Free Software
 Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
 02110-1301, USA.
 .
 On Debian systems, the complete text of version 2 of the GNU General
 Public License can be found in '/usr/share/common-licenses/GPL-2'.
 /usr/share/common-licenses/GPL-2

License: GPL-3.0-or-later
 This program is free software; you can redistribute it and/or modify
 it under the terms of the GNU General Public License as published by
 the Free Software Foundation; either version 3, or (at your option)
 any later version.
 .
 This program is distributed in the hope that it will be useful,
 but WITHOUT ANY WARRANTY; without even the implied warranty of
 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 GNU General Public License for more details.
 .
 You should have received a copy of the GNU General Public License
 along with this program; if not, write to the Free Software
 Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
 02110-1301, USA.
 .
 On Debian systems, the complete text of version 3 of the GNU General
 Public License can be found in '/usr/share/common-licenses/GPL-3'.
 /usr/share/common-licenses/GPL-3
```

```
License: Expat-X
 Permission is hereby granted, free of charge, to any person obtaining
 a copy of this software and associated documentation files (the
 "Software"), to deal in the Software without restriction, including
 without limitation the rights to use, copy, modify, merge, publish,
 distribute, sublicense, and/or sell copies of the Software, and to
 permit persons to whom the Software is furnished to do so, subject to
 the following conditions:
 .
 The above copyright notice and this permission notice shall be included
 in all copies or substantial portions of the Software.
 .
 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
 OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
 IN NO EVENT SHALL THE X CONSORTIUM BE LIABLE FOR ANY CLAIM, DAMAGES OR
 OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
 ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 OTHER DEALINGS IN THE SOFTWARE.
 .
 Except as contained in this notice, the name of the X Consortium shall
 not be used in advertising or otherwise to promote the sale, use or
 other dealings in this Software without prior written authorization
 from the X Consortium.

License: FSFAP
 Copying and distribution of this file, with or without modification,
 are permitted in any medium without royalty provided the copyright
 notice and this notice are preserved.
 This file is offered as-is, without any warranty.

License: GPL-2.0-or-later with Autoconf-data exception
 This program is free software; you can redistribute it and/or modify
 it under the terms of the GNU General Public License as published by
 the Free Software Foundation; either version 2, or (at your option)
 any later version.
 .
 This program is distributed in the hope that it will be useful,
 but WITHOUT ANY WARRANTY; without even the implied warranty of
 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 GNU General Public License for more details.
 .
 You should have received a copy of the GNU General Public License
 along with this program; if not, write to the Free Software
 Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
 02110-1301, USA.
 .
 As a special exception to the GNU General Public License, if you
 distribute this file as part of a program that contains a
 configuration script generated by Autoconf, you may include it under
 the same distribution terms that you use for the rest of that program.
 .
 On Debian systems, the complete text of version 2 of the GNU General
 Public License can be found in '/usr/share/common-licenses/GPL-2'.
```

B.2.2 Example script to build and test locally your Debian package

```
#!/bin/bash

function autoclean {
  rm -f aclocal.m4
  rm -rf auto4te.cache
  rm -f configure~
  aclocal
  autoconf
  automake --add-missing
  rm -f configure~
}

VERSION="1.2.12"

# Ensure that the top directory is clean
rm -f "*_"$VERSION"*"
if [ -d program-$VERSION ]; then
  rm -rf program-$VERSION
fi

DOWN=1
if [ $DOWN -eq 1 ]; then
  # Retrieving the sources of the program from the official repository
  wget https://github.com/Author/Program/archive/refs/tags/v$VERSION.tar.gz
  tar -zxf v$VERSION.tar.gz
  mv Program-$VERSION program-$VERSION
  rm v$VERSION.tar.gz
fi

BUILD=1
if [ $BUILD -eq 1 ]; then
  cd program-$VERSION
  # If the 'debian' directory is not shipped with the official tarball
  # Uncomment and adapt the next line to add the 'debian' directory in the sources directory
  # cp -r ../debian-package-data debian

  # Ensure that there is a README file
  cp README.md README

  # To ensure that the version of the autotools files are suitable for this system
  autoclean
  # Debian packager identification
  export DEBEMAIL="your.email@host.eu"
  export DEBFULLNAME="Your Name"
  # Improved lintian command for package verification
  alias lintian='lintian -EviIL +pedantic --color auto'
  # Creating the Debian origin tarball
  dh_make --createorig -s -y
  # Building the Debian package
  dpkg-buildpackage
  cd ..
fi

TEST=1
if [ $TEST -eq 1 ]; then
  # Linitian on 'program.changes'
  lintian ./program_*changes >& results.lintian
  cd program-$VERSION
  # Licensecheck to check for license
  licensecheck --recursive --copyright . >& ../license.check
  # Scan-copyrights to create a copyrights file from scratch using licensecheck
  scan-copyrights >& ../scan.copy
  cd ..
fi

PIUPARTS=0
if [ $PIUPARTS -eq 1 ]; then
  echo "Piuparts on program.deb" > results.piuparts
  echo " " >> results.piuparts
  sudo piuparts ./program_$VERSION*.deb &>> results.piuparts
  echo " " >> results.piuparts
  echo "Piuparts on program-data.deb" >> results.piuparts
  echo " " >> results.piuparts
  sudo piuparts ./program-data*.deb &>> results.piuparts
fi
```

B.2.3 Example of ITP bug report message

```
Subject: ITP: program -- short description
Package: wnpp
X-Debbugs-Cc: debian-devel@lists.debian.org
Owner: Your Name <your.email@host.eu>
Severity: wishlist

* Package name    : program
  Version         : 1.2.12
  Upstream Author : Your Name <your.email@host.eu>
* URL             : https://www.program.com/
* License         : AfferoGPLv3+
  Programming Lang: C, Fortran90, GLSL
  Description     : short description

"program" is a cross-platform software doing many things,
and I will elaborate about the most significant achievements now ...

Among the many reasons I could think of, to help you understand why you
should consider "program" seriously, I will pick 3:

1) This is the first reason ...

2) This is the second reason ...

3) This is the third reason ...

For the time being "program" is being maintained by ..

Sources are hosted on Github and Salsa:

"program" sources:          https://github.com/Author/Program
"program" sources on Salsa: https://salsa.debian.org/author/program

Finally I am looking for a sponsor.
Indeed as this is my first Debian package, I am looking for all the help I can get.
I am willing to spend time to learn how to do things properly
to get "program" approved by the Debian community and ultimately packaged.
If the proper way to do that is inside a packaging team then after checking
the packaging teams at https://wiki.debian.org/Teams
I think the "Debian *** Team" is likely to be the most appropriate place to start.

Best regards.

Your Name
```

B.3 Metadata for Linux intergation

B.3.1 Custom MIME file(s) setup

This file is required to define file association(s) and uses the XML file format:

```
<?xml? version="1.0" encoding="UTF-8"?>
<mime-info xmlns="http://www.freedesktop.org/standards/shared-mime-info">
  <mime-type type="application/x-ppf">
    <comment>Program Project file</comment>
    <glob pattern="*.ppf"/>
    <generic-icon name="program-project"/>
    <sub-class-of type="text/plain"/>
  </mime-type>
  <mime-type type="application/x-pwf">
    <comment>Program Workspace file</comment>
    <glob pattern="*.pwf"/>
    <generic-icon name="program-workspace"/>
    <sub-class-of type="application/octet-stream"/>
  </mime-type>
</mime-info>
```

Each file association is defined between the **<mime-type>** ... **</mime-type>** tags:

```
...
  <mime-type type="application/x-ppf">
    <comment>Program Project file</comment>
    <glob pattern="*.ppf"/>
    <generic-icon name="program-project"/>
    <sub-class-of type="text/plain"/>
  </mime-type>
...
```

Where the tags are used as follow:

- **<comment>**: to describe the file format.
- **<glob**: to define the file extension.
- **<generic-icon**: to define the icon name, without extension (svg, png).
  In the example providing that the file "program-project.svg" exists and is located in "/usr/share/pixmaps", then it is enough for the system to find it and associate it with the file format.
- **<sub-class-of**: to help the system recognize the format:
  - "text/plain" for text formats.
  - "application/octet-stream" for binary formats.

To get information about the format of this_file:

```
user@localhost ~]$ gio info this_file
```

B.3.2 Desktop entry for desktop application

The following is an example of the content of a installed "**`*.desktop`**" file.

```
[Desktop Entry]
Version=1.0
Name=Program
GenericName=Program
Comment=For more information: https://www.program.com/
Comment[fr]=Pour plus d'information: https://www.program.com/
Comment[es]=Para más información: https://www.program.com/
Keywords=chemistry;physics;
Keywords[fr]=chimie;physique;
Keywords[es]=química;física;
Exec=program %F
# Providing that the file "program.svg" exists and is located in "/usr/share/pixmaps"
# Then it is enough for the system to find it and associate it using:
Icon=propram
Terminal=false
Type=Application
MimeType=application/x-ppf;application/x-pwf;
StartupNotify=true
Categories=Education;Science;Chemistry;Physics;
```

For **`MimeType`** the keywords must match the values defined in the MIME association file by the **`<mime-type type=`**"keyword"**`>`** opening tags, see [Sec. B.3.1].

B.3.3 AppStream metadata for desktop application

```
<?xml? version="1.0" encoding="UTF-8"?>
<!-- This is a commented line for a XML file -->
<!-- Copyright 2023 Your Name -->
<!-- The type of desktop object described -->
<component type="desktop-application">

  <id>com.program.www</id>
  <name>Program</name>
  <summary>An interesting program</summary>
  <!-- The license of your program -->
  <project_license>AGPL-3.0-or-later</project_license>
  <metadata_license>FSFAP</metadata_license>

  <description>
    <p>Program: an utility to do many things !</p>
  </description>
  <!-- The desktop file that describes launch information -->
  <launchable type="desktop-id">program.desktop</launchable>

  <!-- Up to 3 screenshot(s), used in the Linux app store to illustrate your program -->
  <screenshots
    <screenshot>
      <!-- The video cannot be the default screenshot -->
      <!-- Container: mkv, Video codecs allowed: av1 or vp9, Audio codec allowed: opus -->
      <caption>A nice video to illustrate the features of the program</caption>
      <video container="webm" codec="vp9" width="1920" height="1080">https://www.program.com/my-video.webm</video>
    </screenshot
    <screenshot type="default">
      <caption>Overview of the program</caption>
      <image type="source" width="1600" height="900">https://www.program.com/program-overview.png</image>
    </screenshot
    <screenshot>
      <caption>A detail of the program</caption>
      <image type="source" width="1600" height="900">https://www.program.com/program-detail.png</image>
    </screenshot
    <screenshot>
      <caption>Another detail of the program</caption>
      <image type="source" width="1600" height="900">https://www.program.com/program-other-detail.png</image>
    </screenshot
  </screenshots

  <url type="homepage">https://www.program.com/</url>
  <url type="bugtracker">https://github.com/Author/Program/issues/new/choose</url>
  <developer_name>Mr. Your Name</developer_name>
  <update_contact>your.email_AT_host.eu</update_contact>

  <provides>
    <binary>program</binary>
  </provides>

  <keywords>
    <keyword>chemistry</keyword>
    <keyword>physics</keyword>
    <keyword xml:lang="fr">chimie</keyword>
    <keyword xml:lang="fr">physique</keyword>
    <keyword xml:lang="es">química</keyword>
    <keyword xml:lang="es">física</keyword>
  </keywords>

  <content_rating type="oars-1.1" />

  <releases>
    <release version="1.2.12" date="2023-03-29">
      <description>
        <p>Release of version 1.2.12:</p>
        <p>- Bug corrections, for details see: <url>https://github.com/Author/Program/releases/tag/v1.2.12</url></p>
        <p>- Improvements, for details see: <url>https://github.com/Author/Program/releases/tag/v1.2.12</url></p>
      </description>
    </release>
    <release version="1.2.11" date="2023-02-06">
      <description>
        <p>Release of version 1.2.11:</p>
        <p>- Bug corrections, for details see: <url>https://github.com/Author/Program/releases/tag/v1.2.11</url></p>
      </description>
    </release>
  </releases>

</component>
```

This document has been prepared using the Linux operating system and free softwares:

The text editor	"gVim"
The vector graphics editor	Inkscape
The GNU image manipulation program	"GIMP"
And the document preparation system	"LaTeX 2ε".