THE HEP SOFTWARE FOUNDATION (HSF)

HSF-TN-2020-01 v2.0
10.5281/zenodo.1469634
Original Date: 4 October 2016
Last Update: 12 October 2023

# Proposal for HSF Project Best Practices

Benedikt Hegner[1], Benjamin Morgan[2], Graeme A Stewart[1]

[1] *CERN*
[2] *University of Warwick*

# 1 Proposal for HSF Project Best Practices

## 1.1 Introduction

This technical note is a proposed list of best practices for HSF and other HEP open source projects. The main motivation is to ensure interoperability and usability of a given project by other projects and being able to build consistent software stacks. In addition, it should make it easier for other developers to contribute to existing projects. In the following we discuss different practices and conventions that ease the life of

- developers and new contributors,
- end-users and other client projects.

Afterwards we provide a checklist of the proposals you may want to use for your project. The proposals are mainly based on experience with the LCG software projects and releases. You may find many of the points discussed here trivial. However, people usually differ in what they consider trivial. The technical recommendations in this document are tailored towards C++-based projects, but can easily be mapped onto, e.g., Python-based projects.

## 1.2 Project Scope, Name and Visibility

On starting a project, make sure you have an idea of the project's scope and goals. Try to pick a name that is suitable for that. You need to ensure uniqueness, as the name will be used to name software artifacts, like libraries, code namespaces, error messages, etc. In addition, have a look around to see if it conflicts with pre-existing trademarks for software products or services.

Though it sounds like a triviality, your project should be made known to the community. For this, having a *dedicated project website* or another entry point for information is essential. It should concentrate all the information useful for users and developers. If possible, it should point at all the other information listed in this document. It is important to find the right place to put information. Try not to repeat yourself, as duplicated documentation can easily get out of sync. Access to all sources of project information should be granted to search engine spiders. Furthermore, the HSF working groups and software forums allow you to present your project or ideas at any stage in its project lifecycle.

There is an excellent open source general guide that covers community interactions, getting your project known and helping find new contributors, particularly helpful for the sociology of successful open source projects.

## 1.3 Supporting Developers and Contributors

The following sections discuss points mainly relevant for project developers and potential new contributors.

### 1.3.1 Code repository

The first requirement for an open-source project is fully versioned code in a *public repository.* The code should be accessible in anonymous read-only mode by everybody. Services like GitHub or GitLab provide it for free. In addition efforts like hepforge or labs like CERN or DESY may host HEP-specific packages. Services supporting a clone plus *merge-request/pull-request workflow* can be extremely helpful to attract new contributors, as it is the current de-facto standard for open software development. Try to make sure there are no barriers to contributing in this way, e.g., lab hosted services may be more difficult for users without accounts to use for merge requests.

### 1.3.2 License and Copyright

The ownership and copyright of the code has to be well defined and understood. Host labs for experiments will often hold copyright on behalf of experiments and projects that are not legal entities and this can simplify future life enormously. In a second step, the code and software provided should be properly licensed in order to be able to use code provided by others, and to allow people to re-use, update, or improve the software you provide. The HSF technical note *HSF-TN-2016-01* (*Software Licence Agreements HSF Policy Guidelines*) discusses various options. This is one of the topics that is *typically ignored at the beginning of a project and hard to fix afterwards.*

### 1.3.3 Compilation and other commands

Compiling, installing and testing should each be - if possible - single-command actions. In particular, making testing easy is important. A good place to put the necessary information is a *README* file in the repository. Relying on community standards like *CMake* make it easier for others to use and understand the setup.

### 1.3.4 Testing

To improve on the quality of software, unit and integration testing are essential. Having well-documented tests makes it as well easier for contributors to participate. They can check whether they break old features and can, with new tests, document what their addition is supposed to do. Testing can also assist in the triage and fixing of bugs. Tests can be written to reproduce reported issues and fail when they occur, with subsequent fixes validated by the tests passing.

For *unit tests* plenty of software packages exist, of which *gtest* and *catch* are two good choices for C++ projects. Integration tests running a software project in a certain setup can take advantage of *CTest* (supplied as part of *CMake*) and *CDash* or be driven by shell scripts. Ease of use is again important here, otherwise tests tend not to be run. For example, *CMake*/*CTest* add dedicated *test* targets to buildscripts so that running the tests is a simple matter of “building” the target, e.g. `make test` when using Makefiles.

**Continuous integration** As well as being available from the command line, tests of the code should run on all pull/merge requests so that reviewers can immediately see if the proposed changes break any known functionality. There are many options to do this, well integrated with modern code repositories.

### 1.3.5 Communication and Reporting

A mailing list to contact developers is always useful, even as issue trackers become also more capable of supporting discussions. It is better to have publicly and anonymously accessible archives and to be open for subscription and posting by the public.

### 1.3.6 Issue tracking

It is useful to provide an issue (bug) tracker for users and developers to interact with, allowing a view of both open and closed tickets anonymously by the public. Optimal solutions here are the issue tracking capabilities the code repository itself (such as GitLab or GitHub), as solutions which integrate directly with the code repository are much easier to use for both users and developers. CERN's JIRA service is an alternative.

### 1.3.7 Reference Guide

For developers it is important to have a good overview of provided interfaces, existing classes, and implementation details. For this a reference guide is a helpful tool. The de-facto standard for creating reference guides in C++ projects is *Doxygen*.

### 1.3.8 Conventions and Workflows

Every project choses certain (coding) conventions and integration workflows. While there is a plethora of possibilities, the concretely chosen conventions and workflows should be documented visibly. A *How to contribute* document is good practice. This is, as well, a nice place to add information where contributions by others would be possible and desired.

### 1.3.9 Be prepared for using external (cloud) services

The project should be careful in its assumptions about the environment available for development and testing, like access to extra storage or connections. Make it easy to integrate your software into e.g. a container. Containers also help greatly in deploying a continuous integration system, e.g., testing the build on different Linux flavours.

## 1.4 End-users and client projects

### 1.4.1 Documentation

In addition to the already mentioned documentation, end-user focused documentation is important. A little checklist further below summarizes the most important information to be given as part of the documentation.

### 1.4.2 Release Information

While developers (most of the time) know the changes between various releases, it is important to document changes between releases for end-users. It turned out to be a good policy to have multiple categories of releases, like production releases, development releases, bug fix releases, etc. While each project may have different conventions here, the chosen convention should be explained, including its meaning in terms of changes to the project's *API* and *ABI*. A clear numbering scheme like "major.minor.patch" can support this (also known as semantic versioning). For each release the *supported compilers*, *supported operating systems* and *required dependencies* should be listed. This helps avoiding frustrations on the user side.

### 1.4.3 Interaction with developers

To be able to interact with developers, both the already mentioned *mailing list* and *issue tracker* are important and helpful. The required permissions to post there should be as low as possible. Make it easy for people to give feedback and to contribute.

### 1.4.4 Relocatability and co-existence of versions

Often a project has to be integrated into bigger software stacks. Being relocatable, i.e. having no hard-coded absolute paths in any build artifact, is often a necessity to deploy and distribute these stacks. To enable your project to become part of such a software stack, try to make it relocatable. In addition your software should not make too strong assumptions about its own location.

### 1.4.5 Usability and run-time settings

It should be straight forward for a user to set up and run your project. This can for example be ensured by providing environment setup scripts, but the number of environment variables required should be limited as far as possible.

### 1.4.6 Be prepared for using external (cloud) services

The project should be careful in its assumptions about the user environment, like having access to extra storage or network connections. Like for the development process, make it easy to integrate your software into e.g. a container.

### 1.4.7 Publications and References

Users of your software should be able to give credit to your work. Try to publish your work in conference proceedings or journals such as *Computing and Software for Big Science* so that it can be properly cited.

To help users cite your software make sure that you list the recommended citation where it can easily be found in your documentation. One highly recommended standard is now to add a `CITATION.cff` file to your repository. This is a human and machine readable file that tells users exactly how to write a citation. More information is on the Citation File Format website.

## 1.5 Making best practices easier - the HSF Template Project

Many of the points mentioned are per se trivial, but need some infrastructure to be set up. To assist new projects, an HSF project template was created. It covers many of the technical points and provides some canonical or example implementation for many of the issues. It is meant as open collection point of ideas and proposals by the community.

## 1.6 Checklist

A little checklist of topics to consider is given here. Not every point applies to every project, but it may give you a handle in improving the quality of the software you provide.

### 1.6.1 Repository and code checklist

Table 1: Repository and code checklist

| **Topic** | **Possible solution(s)** | **Template** |
|---|---|---|
| Public repository | github, gitlab | - |
| License + file | MIT, Apache2 | MIT, Apache2 |
| README file | Markdown, reStructuredText | Yes |
| Reference guide | Doxygen | Doxygen |
| build scripts | CMake | CMake |
| Unit testing | gtest, catch | catch |
| Integration testing | CTest, scripts | CTest |
| version file | headers | headers |
| Relocatability | strict policy | Yes |
| environment setup | (c)sh script | - |

### 1.6.2 Procedure and release checklist

The following list contains mostly "nice-to-have" points. Having them well-defined and documented helps both developers as well as potential volunteer contributors.

Table 2: Procedure and release checklist

| Topic | Possible solution(s) | Template |
|---|---|---|
| Defined workflow | plenty | |
| Automatic testing | Github Actions, gitlab CI | - |
| Test run+reporting | CTest,CDash | CTest |
| Static Analysis | clang-analyzer | - |

### 1.6.3 Website and information checklist

Table 3: Website and information checklist

| Topic | Possible solution(s) |
|---|---|
| Website | jekyll, github pages, Hugo |
| How to contribute | - |
| User manual | markdown, doxygen |
| Reference manual | doxygen |
| Bug tracker | github, gitlab, jira |
| Mailing list | google groups, e-groups |
| Link to repository | - |
| List of releases | - |
| List of supported OS+compilers | - |
| List of pre-requisites | - |
| CITATION.cff file | CFF website |

## 1.7 Summary and Outlook

This document described a few best practices, and potential implementations. Updates, additional points or corrections are very welcome.