POLYTECH GRENOBLE

UGA Université Grenoble Alpes

Integration of Software Heritage in Nix package manager

Romain PASDELOUP - Alexandre SALMON

INFO4 Projects 2019-2020 Mr. Olivier RICHARD

Table of Contents

Introduction

As fourth-year engineering students at Polytech Grenoble, we had to take part in a project in order to improve our technical and management skills.

This report has been written to sum up what is our project and what has been implemented during the 13 weeks of development. It can be used as a basis for further improvement and implementation of new functionalities.

Our code, as well as the documentation and our sandbox are available in the following repo: https://gricad-gitlab.univ-grenoble-alpes.fr/Projets-INFO4/19-20/20. If you don't have access to it, you can request it to an administrator.

Our topic

Our project's main goal is to provide an implementation for Software Heritage in Nixpkgs.
The ultimate goal is to make it possible for someone to use Software Heritage as a source to download tarballs for a specific release of the software hosted on the platform.

Context

Persistence of knowledge and its transmission has always been an issue. We lost things like texts, paint and many other cultural elements from the past because people didn't thought about the future. We encounter a similar issue in computer science related subjects. Software receive updates, then become obsolete and finally disappear because they aren't maintained anymore. That's why Software Heritage came to be. It's purpose is "to collect, preserve, and share software that is across cultural heritage, industry, education, science, and research communities"[1].

On the other hand, Nix package manager is "a cross-platform package manager that relies on a functional deployment model where software is installed into unique directories generated through cryptographic hashes"[2]. Thanks to that, it solves dependencies issues that you can encounter (when you need a specific version of a software for example) and allows a modular environment. There is an OS called NixOS that relies on Nix package manager.

Linking those two projects would mean to have a sustainable software distribution and will make a large number of open source projects reproducible over time.

[1] From Wikipedia : https://en.wikipedia.org/wiki/Software_Heritage
[2] From Wikipedia : https://en.wikipedia.org/wiki/Nix_package_manager

Our objective

Following Nix's ideology, our main goal will be to write a specific fetcher for Software Heritage and include it in the Nixpkgs code. This in order to make the git repositories available in Software Heritage accessible to the functional Nix package manager.

Technical Context

Software Heritage API

Software Heritage owns an API available online that allows us to interact with the internal database. It has a lot of endpoints and functionalities.
Every revision of a project is identified by a key named “swhid”. If you know this key, you are able to retrieve the revision in their database and download a tarball from it.

The API also has a section called “The Vault”, where tarballs are generated. This process is called “Cooking” and runs if a revision is requested and no tarball has been generated yet.

Nix package management

As written in the context section, Nix relies on cryptographic hashes to install the software. It means that multiple releases (that are identified by those hashes) of the same software can coexist on the same system.

On the graph below, we can observe an example of Nix hierarchy where two versions of git (1.9.3 and 2.0.0) coexist. It means that even if you update software, you can still keep the old version as a way to not break existing configurations.

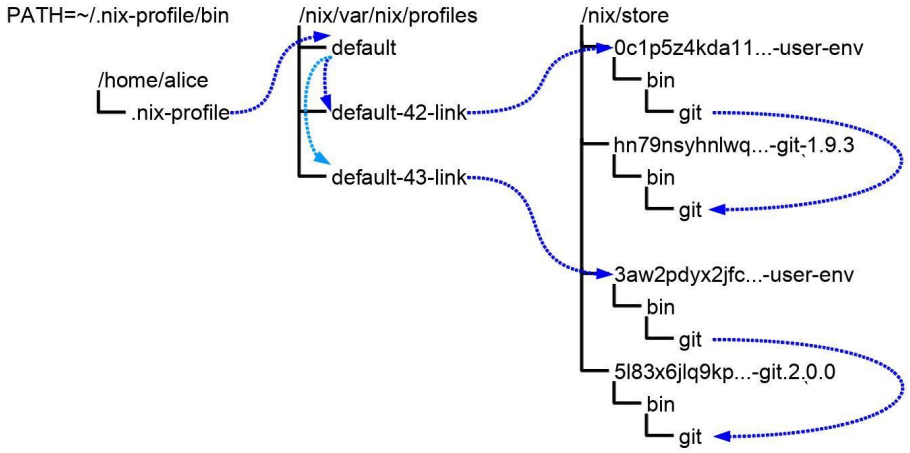

Image 1: *Coexistence of different releases of git*
(Source: https://www.infoq.com/articles/configuration-management-with-nix/)

Package architecture

Nixpkgs is centered around the idea of *nix expressions*. Those expressions are written in the nix language, which is a *functional language*. Adding a new package in nixpkgs means adding a “derivation” to Nix. A derivation could be defined as a package description.

For a *hello-world* package, it could be defined like this:

```
1 { stdenv, fetchurl, perl }:
2
3 stdenv.mkDerivation {
4   name = "hello-2.1.1";
5   builder = ./builder.sh;
6   src = fetchurl {
7     url = ftp://ftp.nluug.nl/pub/gnu/hello/hello-2.1.1.tar.gz;
8     sha256 = "1md7jsfd8pa45z73bz1kszpp01yw6x5ljkjk2hx7wl800any6465";
9   };
10  inherit perl;
11 }
```

The first line announces the required packages for our new package to be fetched, built, and run. Then, from line 3 to line 11, the function *mkDerivation* is called. This function call will create a new derivation for the package named “hello-2.1.1”, source code is available for download at this URL “ftp://ftp.nluug.nl/pub/gnu/hello/hello-2.1.1.tar.gz” and can be built using the script “builder.sh”.

Line 6 is quite interesting, it is a function call wich downloads source code from a given URL and verifies it’s hash. This part is the one we have to replace to get *nixpkgs* to download from *software heritage*. The *fetchurl* package allows us to download from a given URL or its mirrors, however, we can’t use it to fetch from *software heritage* because a tarball of the source code might not be available at the moment.

With an appropriate fetcher, the same “hello” package fetcher from *software heritage* could look like this.

```
1 { stdenv, fetchswh, perl }:
2
3 stdenv.mkDerivation {
4   name = "hello-2.1.1";
5   builder = ./builder.sh;
6   src = fetchswh {
7     swhid = 1234567890;
8     sha256 = "1md7jsfd8pa45z73bz1kszpp01yw6x5ljkjk2hx7wl800any6465";
9   };
10  inherit perl;
11 }
```

with *swhid* being the package id for the software heritage API.

Our work

During this project, we wanted to develop with Incremental releases. Indeed, with this method, we are able to offer a working version of each new feature.
We planned to have three releases:

- A first version, fetching only tarballs that are already cooked;
- A second one, where we warn the user if the tarball isn't generated and request cooking;
- A last one, where the tarball is cooked and downloaded asynchronously.

Simple version

As a first step, we created a fetcher based on *fetchTarball*. *FetchTarball* is a fetcher provided by Nix builtins tools that download a tarball from an URL. The main idea behind this fetcher was to better understand the mechanics behind *nixpkgs*. This release assumes that the package is already cooked and available via the *software heritage* API.

Ignoring the cooking status, fetching source code from *software heritage* is quite straight forward.

```
fetchTarball "http://archive.softwareheritage.org/api/1/vault/directory/${swhid}/raw"
```

This line is the main part of the fetcher. It composes the fetch URL by completing the template given by the API with the package id. And fetches it using fetchTarball.

This first version being quite primitive, it has some limitations. The most obvious one being that it requires the package to be cooked. This fetcher doesn't require any hash so it doesn't check the integrity of the downloaded package.

Work in progress: request a non generated tarball

After creating the first version, we wanted to add a feature to request the cooking of tarball if it hasn't been generated yet. Our idea was to use a custom script as a builder, the way fetchurl fetcher already does it.

This script will do the necessary API call to The Vault, to determine if the tarball is already generated or not. If it exists, it will download it and store it. Otherwise, it will do another request to The Vault that will start the cooking. The user will be warned that their download isn't available at the moment.

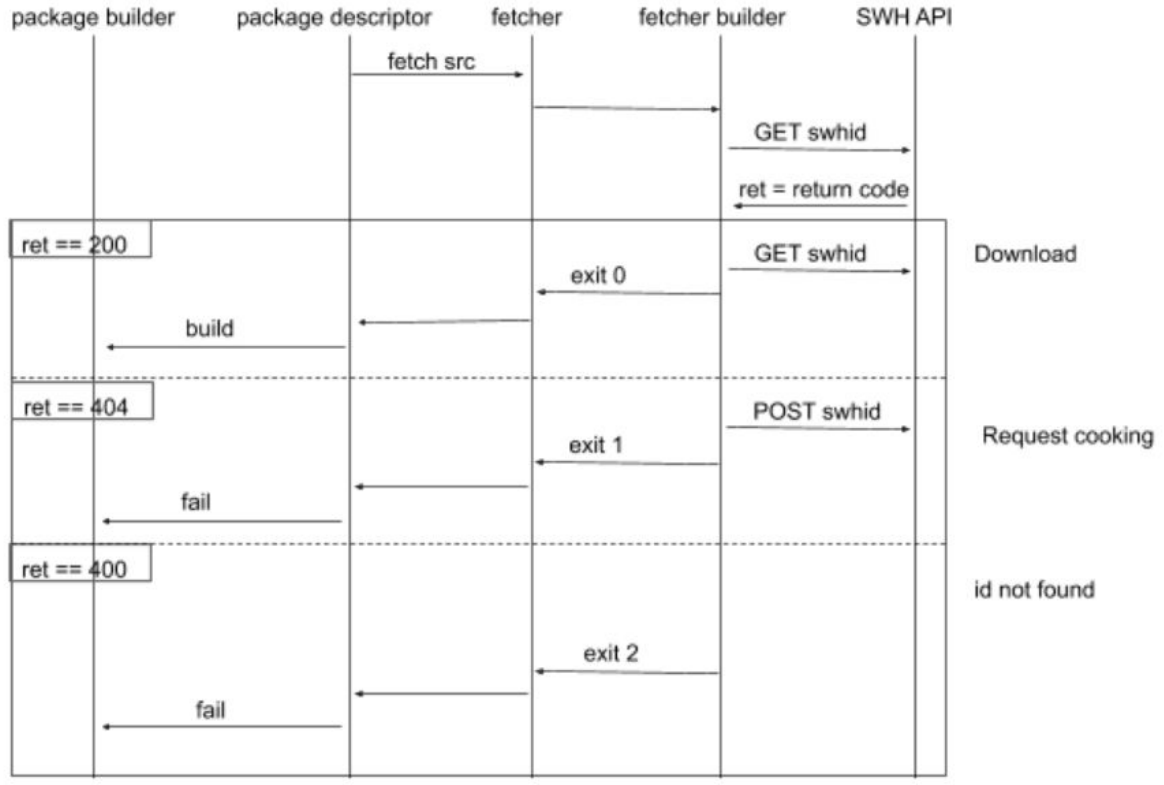

Image 2: Schema showing interactions and different API calls made

Our script to fetch the tarball resort to the cURL command much like fetchurl. First, we do a call to the API to determine if the tarball is available or not.
Then depending on the result code, we will either:

- Make an api call to start the cooking of the tarball;
- Download the tarball;
- Handle errors

This release fixes some issues that the previous one had. Indeed, now we take in consideration the cooking status of the tarball. The main issue now is that you still can't download the uncooked tarball and will have to try again in a few moments.

Planned release: asynchronous download

The last release we planned to do was an improvement for the previous one. Indeed, we wanted to be able to download the tarball asynchronously if it's not available when we request the download. To do so, we would inspire ourselves from the work done in Guix (http://guix.gnu.org/), a project similar to Nix but led by GNU and the Free Software Foundation. Their fetcher for Software Heritage can be found here: https://github.com/dongcarl/guix/blob/master/guix/swh.scm.

Results

Our first release is working. We can download an archive that is already generated. The default builder will try to build the software which can fail sometimes. But it can be disabled by editing the buildPhase.

Our second release isn't working at the moment. We are struggling with the use of the cURL command. We tried to reproduce what has been done with fetchurl (minus the multiple mirrors that are useless in our case), but we encountered a bug: the cURL command can't seem to reach any website (curl: (6) could not resolve host). We are still investigating the issue but haven't found a solution yet.

We didn't start the third release but were expecting to use code from the second one.

Conclusion

Working on such an unusual topic has been quite interesting. Most of the tools used in this project were unknown to us, so it has been quite rewarding to acquire competences. However, it has it's cost, the project being technically very different from anything we worked on so far. As such, we spend most of our time studying to better understand Nix language and Nixpkg mechanics. Nix is a powerful technology and project, knowing more about it can only be an asset for our future. Also working with a major project like Software Heritage which promotes important values like open-sourcing and sharing of knowledge is enriching for future engineers like us.

The biggest difficulty we encountered was Nix and SWH API documentation. Both can be quite hard to understand. Nix being for us a whole new world, we missed practical examples and punctual explanations which would have been very useful during the creation of our fetcher. Software heritage has an interesting way of dealing with all these sources, so it's API isn't what we expected it to be. Its documentation lacked examples too. Even if those documentations were a bit complicated to understand, it only encouraged us to learn by trying. When it comes to our researches, most of our results come from experimentations with curl or experimental packages and fetchers.

The things that are left to be done with the project are fixing the cURL issue for the second release and making the third release. Some improvement we can think of for our project are:

- Selecting the release version for the software in the fetcher
- Adding some sort of checking for the tarball (like the hash in other fetchers)