Long term reproducibility

Contents

Background

One of the main goals for Apertis is to provide teams the tools to support their products for long life cycles needed in many industries, from civil infrastructure to automotive.

This document discusses some of the challenges related to long-term support and how Apertis addresses them, with particular interest in reliably reproducing builds over a long time span.

Apertis addresses that need by providing stable release channels as a platform for products with a clear trade-off between leading-edge functionality and stability. Apertis encourages products to track these channels closely to deploy updates on a regular basis to ensure important fixes reach devices in a timely manner.

Stable release channels are supported for at least two years, and product teams

have three quarters of overlap to rebase to the next release before the old one reaches end of life. Depending on the demand, Apertis may extend the support period for specific release channels.

However, for debugging purposes it is useful to be able to reproduce old builds as closely as possible. This document describes the approach chosen by Apertis to address this use case.

For our purposes bit-by-bit reproducibility is not a goal, but the aim is to be able to reproduce builds closely enough that one can reasonably expect that no regressions are introduced. For instance some non essential variations involve things like timestamps or items being listed differently in places where order is not significant, cause builds to not be bit-by-bit identical while the runtime behavior is not affected.

Apertis artifacts and release channels

As described in the release flow[1] document, at any given time Apertis has multiple active release channels to both provide a stable foundation for product teams and also give them full visibility on the latest developments.

Each release channel has its own artifacts, the main one being the deployable images[2] targeting the reference hardware platforms[3], which get built by mixing:

- reproducible build environments
- build recipes
- packages
- external artifacts

These inputs are also artifacts themselves in moderately complex ways:

- build environments are built by mixing dedicated recipes and packages
- packages are themselves built using dedicated reproducible build environments

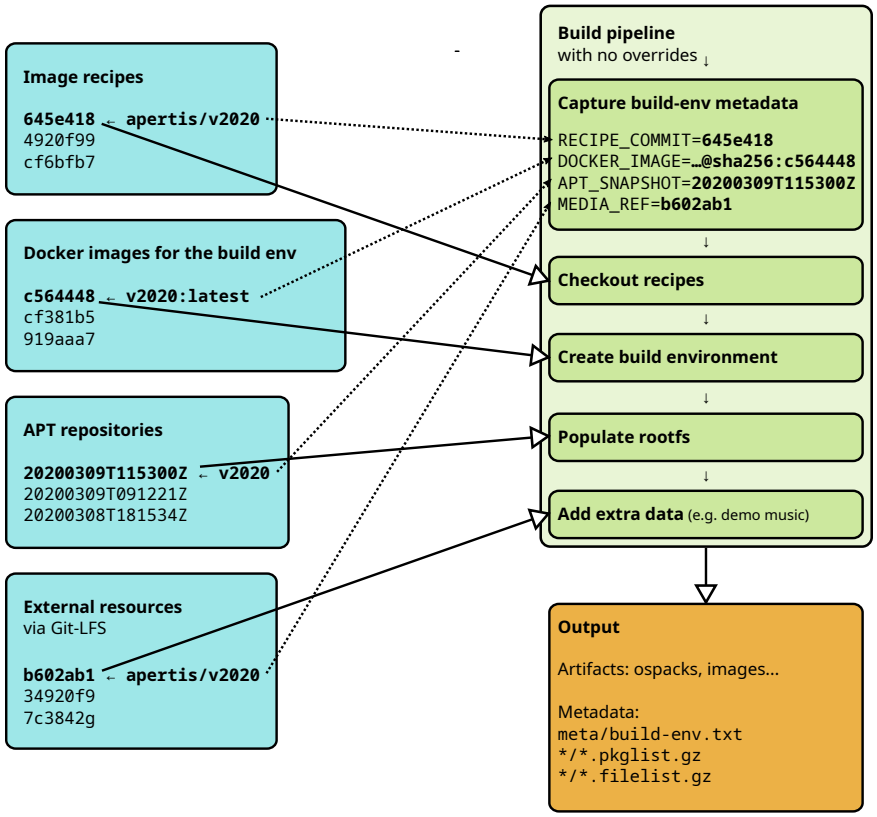
However, the core principle for maintaining multiple concurrent release channels is that each channel should have its own set of inputs, so that changes in a channel do not impact other channels.

Even within channels sometimes it is desirable to reproduce a past build as closely as possible, for instance to deliver a hotfix to an existing product while minimizing the chance of introducing regressions due to unrelated changes. The Apertis goal of reliable, reproducible builds does not only help developers in their day-to-day activities, but also gives them the tools to address this specific use-case.

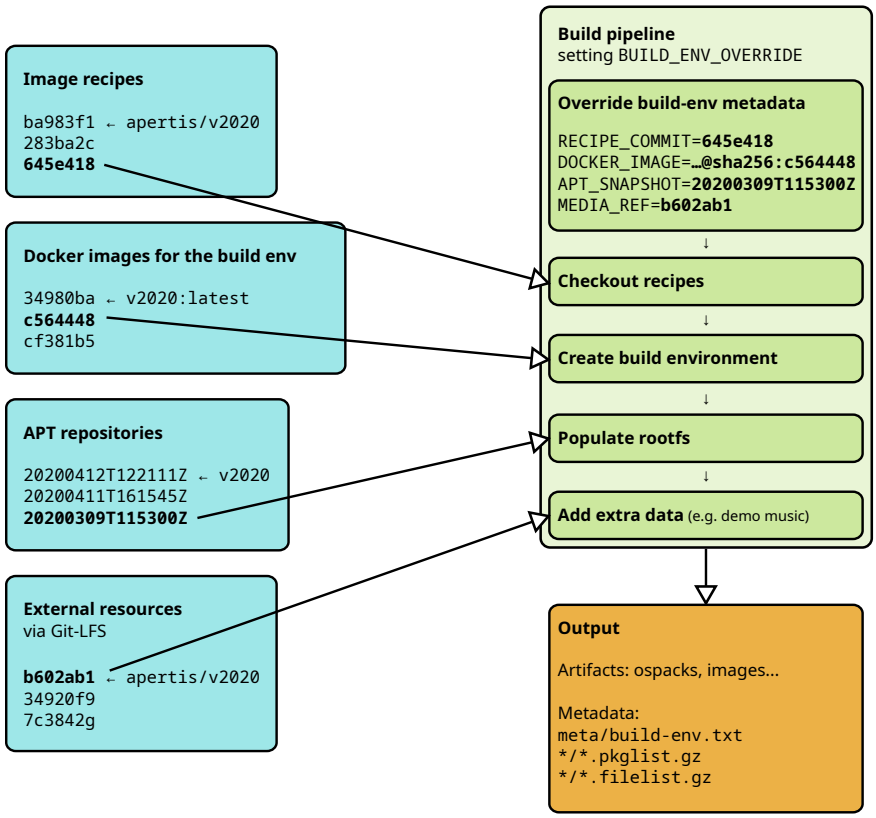
[1] https://apertis-website-0b3586.pages.apertis.org/policies/release-flow/
[2] https://apertis-website-0b3586.pages.apertis.org/policies/images/
[3] https://www.apertis.org/reference_hardware/

The first step is to ensure that all the inputs to the build pipeline are version-controlled, from the pipeline definition itself to the package repositories and to any external data.

To track which input got used during the build process the pipeline stores an identifier for each of them to uniquely identify them. For instance, the pipeline saves all the Git commit hashes, Docker image hashes, and package versions in the output metadata.



While the pipeline defaults to using the latest version available in a specific channel for each input, it is possible to pin specific version to closely reproduce a past build using the identifiers saved in its metadata.



Reproducible build environments

A key challenge in the long term maintenance of a complex project is the ability to reproduce its build environment in a consistent way. Failing to do so means that undetected differences across build environments may introduce hard to debug issues or that builds may fail entirely depending on where/when they get triggered.

In some cases, losing access to the build environment effectively means that a project can't be maintained anymore, as no new build can be made.

To be able to avoid these issues as much as possible, Apertis makes heavy use of isolated containers based on Docker images[4]

All the Apertis build pipelines run in containers with minimal access to external resources to keep the impact of the environment as low as possible.

For the most critical components, even the container images themselves are

[4] https://gitlab.apertis.org/infrastructure/apertis-image-recipes/#building-in-docker

created using Apertis resources, minimizing the reliance on any external service and artifacts.

For instance, the `apertis-v2020-image-builder` container image provides the reproducible environment to run the pipelines building the reference image artifacts for the v2020 release, and the `apertis-v2020-package-source-builder` container image is used to convert the source code stored in GitLab in a format suitable for building on OBS.

Each version of each image is identified by a hash, and possibly by some tags. As an example the `:latest` tag points to the image which gets used by default for new builds. However, it is possible to retrieve arbitrary old images by specifying the actual image hash, providing the ability to reliably reproduce arbitrarily old build environments.

To prevent space consumption to grow unboundedly, images that are not pointed by any tag are periodically garbage-collected and removed. To ensure that the needed images are preserved, product teams must ensure that there's **at least one tag** pointing to them.

Each container image build should be tagged with its build id, for instance `:build-20200103.0112` at build time; at release time, the container image used to build the artifacts should be additionally tagged with a release tag, for instance `:v2020.3` for the v2020.3 release.

Cleanup policies[5] must be set up to make the build tags expire after some time, to ensure that the unused container images can be reclaimed during garbage-collection.

To further make build environments more reproducible, care can be taken to make their own build process as reproducible as possible. The same concerns affecting the main build recipes affect the recipes for the Docker images, from storing pipelines in Git, to relying only on snapshotted package archives, to taking extra care on third-party downloads, and the following sections address those concerns for both the build environments and the main build process.

Build recipes

The process to the reference images is described by textual, YAML-based Debos recipes[6] Git repository, with a different branch for each release channel.

The textual, YAML-based GitLab-CI pipeline definitions then control how the recipes are invoked and combined.

Relying on Git for the definition of the build pipelines make preserving old versions and tracking changes over time trivial.

[5] https://docs.gitlab.com/ee/user/packages/container_registry/reduce_container_registry_storage.html#cleanup-policy

[6] https://gitlab.apertis.org/infrastructure/apertis-image-recipes/

Rebuilding the `v2020` artifacts locally is then a matter of checking out the recipes in the `apertis/v2020` branch and launching `debos` from a container based on the `apertis-v2020-image-builder` container image.

By forking the repository on GitLab the whole build pipeline can be reproduced easily with any desired customization under the control of the developer.

Packages and repositories

The large majority of the software components shipped in Apertis are packaged using the Debian packaging format, with the source code stored in GitLab that OBS uses to generate prebuilt binaries to be published in a APT-compatible repository.

Separate Git branches and OBS projects are used to track packages and versions across different parallel releases, see therelease flow[7] document for more details.

For instance, for the v2020 stable release:

- the `apertis/v2020` Git branch tracks the source revisions to be landed in the main OBS project
- the `apertis:v2020:{target,development,sdk}` projects build the stable packages
- the `deb https://repositories.apertis.org/apertis/ v2020 target development sdk` entry points `apt` to the published packages

For most of the time the stable channel is frozen and updates are exclusively delivered through the dedicated channels described below.

Updates are split between small security fixes with low chance of regressions and updates that also address important but non security-related issues which usually benefit from more testing.

For security updates:

- the Git branch is `apertis/v2020-security`
- the OBS projects are `apertis:v2020:security:{target,development,sdk}`
- `deb https://repositories.apertis.org/apertis/ v2020-security target development sdk` is the APT repository

Similarly, for the general updates:

- the Git branch is `apertis/v2020-updates`
- the OBS projects are `apertis:v2020:updates:{target,development,sdk}`
- `deb https://repositories.apertis.org/apertis/ v2020-updates target development sdk` is the APT repository

On a quarterly basis the stable channel get unfrozen and all the updates get rolled in it, while the `security` and `updates` channel get emptied.

[7] https://apertis-website-0b3586.pages.apertis.org/policies/release-flow/

This approach provides to downstreams and product teams a stable basis to build their product without hard to control changes. Products are recommended to also track the security channel for timely fixes, enabling product teams to easily identify and review the changes shipped through it.

The updates channel is not directly meant for production, but it offers to product teams a preview of the pending changes to let them proactively detect issues before they reach the stable channel and thus their products.

While the stability of the release channels is suitable for most use-cases, sometimes it is desirable to reproduce an old build as close to the original as possible, ignoring any update regardless of their importance.

To accomplish that goal the package archives are snapshotted regularly, storing their full history. The image build pipeline accepts an optional parameter to use a specific snapshot rather than the latest contents. This results in the execution installing exactly the same packages and versions as the original run, regardless of any changes that landed in the archive in the meantime.

To use a snapshot it is sufficient to change the APT mirror address, for instance going from `https://repositories.apertis.org/apertis/` to `https://repositories.apertis.org/apertis/20200305T132100Z` and similarly for product-specific repositories.

Every time an update is published from OBS a snapshot is created, tracking the full history of each archive. More advanced use-cases can be addressed using the optional Aptly HTTP API[8].

External artifacts

While the packaging pipeline effectively forbids any reliance on external artifacts, the other pipelines in some case include components not under the previously mentioned systems to track per-release resources.

For instance, the recipes for the HMI-enabled images include a set of example media files retrieved from a `multimedia-demo.tar.gz` file hosted on an Apertis web server.

Another example is given by the `apertis-image-builder` recipe checking out Debos directly from the master branch on GitHub.

In both cases, any change on the external resources impacts directly all the release channels when building the affected artifacts.

A minimal solution for `multimedia-demo.tar.gz` would be to put a version in its URL, so that recipes can be updated to download new versions without affecting older recipes. Even better, its contents could be put in a version tracking tool, for instance using the Git LFS support available on GitLab.

[8] https://www.aptly.info/doc/api/

In the Debos case it would be sufficient to encode in the recipe a specific revision to be checked out. A more robust solution would be to use the packaged version shipped in the Apertis repositories.

Main artifacts and metadata

The purpose of the previously described software items is to generate a set of artifacts, such as those described on the images[9] page. With the artifacts themselves a few metadata entries are generated to help tracking what has been used during the build.

In particular, the `pkglist` files capture the full list of packages installed on each artifacts along their version. The `filelist` files instead provide basic information about the actual files in each artifacts.

With the information contained in the `pkglist` files it is possible to find the exact binary package version installed and from there find the corresponding commit for the sources stored in GitLab by looking at the matching Git tag.

The `build-env.txt` file instead captures metadata about the build environment.

For instance, here's a sample from the pipeline that built the `v2021dev3.0` release[10]:

```
PIPELINE_VERSION=20200921.1223
DOCKER_IMAGE=registry.gitlab.apertis.org/infrastructure/apertis-docker-
images/v2021dev3-image-builder@sha256:50724ec3105f9ea840fa70b536768148722ae59e09b7861a9051ad1397b57f64
RECIPES_COMMIT=b4f1c5c85bd4603f2d9158f513c142a77a3c65c3
RECIPES_URL=https://gitlab.apertis.org/infrastructure/apertis-image-recipes/
PIPELINE_URL=https://gitlab.apertis.org/infrastructure/apertis-image-
recipes/-/pipelines/157555
UPLOAD_ROOT=/srv/images/public
IMAGE_URL_PREFIX=https://images.apertis.org
```

With the `RECIPES_URL` and `RECIPES_COMMIT` variables it is possible to find the exact revision of the recipes in the `apertis-image-recipes` project[11]

The `DOCKER_IMAGE` variable captures the exact revision of the Docker image by explicitly using the digest syntax, to ensure the build environment can be reproduced perfectly. Care must be taken to ensure the retention policy of the container registry preserves the used image for long enough. For the Apertis reference image recipes we currently use a rather aggressive cleanup policy, only preserving images built during the past week but this can be easily customized from the GitLab UI[12]. Improving the preservation of the images used for each release is under discussion.

[9] https://apertis-website-0b3586.pages.apertis.org/policies/images/

[10] https://images.apertis.org/release/v2021dev3/v2021dev3.0/meta/build-env.txt

[11] https://gitlab.apertis.org/infrastructure/apertis-image-recipes/commit/b4f1c5c85bd4603f2d9158f513c142a77a3c65c3

[12] https://docs.gitlab.com/ce/user/packages/container_registry/#cleanup-policy

The metadata above can then be used to reproduce the build.

The implementation plan section defines the remaining planned improvements.

Package builds

Package builds happen on OBS which does not have snapshotting capabilities and always builds every package on a clean, isolated environment built using the latest package versions for each channel.

Since the purposes taken in account in this document do not involve large scale package rebuilds, it is recommended to use the SDK images and the deviants in combination with the snapshotted APT archives to rebuild packages in an environment closely matching a past build.

Recommendations for product teams

Builds for production should:

1. pick a specific stable channel (for instance, `v2020`)
2. version control the build pipelines using branches specific to a stable channel
3. in the build pipeline, use the latest Docker image for that specific channel, for instance `v2020-image-builder` or a product-specific downstream image based on that
4. use the main OBS projects for the release channel, for instance `apertis:v2020:target`, with the security fixes from `apertis:v2020:security:target` layered on top
5. store the product-specific packages in OBS projects targeting a specific release channel, layered on top of the projects mentioned in the previous point
6. use the matching APT archives during the image build process
7. deploy fixes from the stable channels as often as possible

Development builds are encouraged to also use the contents from the non-security updates (for instance, `apertis:v2020:updates:target`) to get a preview of non time-critical updates that will folded in the main archive on a quarterly basis.

The assumption is that products will use custom build pipelines tailored to the specific hardware and software needs of the product. However, product teams are strongly encouraged to reuse as much as possible from the reference Apertis build pipelines using the GitLab CI and Debos include mechanisms, and to follow the same best-practices about metadata tracking and build reproducibility described in this document.

Implementation plan

Snapshot the package archive

To ensure that build can be reproduced, it is fundamental to make the same contents available from the package archive.

The most common approach, also employed in Debian upstream, is to take snapshots of the archive contents so that subsequent builds can point to the snapshotted version and retrieve the exact package versions originally used.

To provide the needed server-side support, the archive manager need to be switched to the `aptly` archive manager as it provides explicit support for snapshots. The build recipes then need to be updated to capture the current snapshot version and to be able to optionally specify one when initiating the build.

Due to the way APT works, the increase in storage costs for the snapshot is small, as the duplication is limited to the index files, while the package contents are deduplicated.

Version control external artifacts

External artifacts like the sample multimedia files need to be versioned just like all the other components. Using Git LFS and Git tags would give fine control to the build recipe over what gets downloaded.

Link to the tagged sources

The package name and package version as captured in the `pkglist` files are sufficient to identify the exact sources used to generate the packages installed on each artifacts, as they can be used to identify an exact commit.

However, the process can be further automated by providing explicit hyperlinks to the tagged revision on GitLab.

How to reproduce a release build and customize a package

Reproduce the build

1. Open the folder containing the build artifacts, for instance `v2021dev3.0/`[13]
2. Find the `build-env.txt` metadata, for instance `meta/build-env.txt`[14]
3. Find the project hosting the recipes with the `RECIPES_URL` variable in `build-env.txt`

[13] https://images.apertis.org/release/v2021dev3/v2021dev3.0/
[14] https://images.apertis.org/release/v2021dev3/v2021dev3.0/meta/build-env.txt

4. On GitLab, fork[15] the recipes project
5. Create a new branch[16] in the recipes repository pointing to the commit saved in the `RECIPES_COMMIT` field of `build-env.txt`, for instance commit `b4f1c5c85bd4603f2d9158f513c142a77a3c65c3`[17]
6. Go to `Pipelines` → `Run Pipeline` page on GitLab to execute a CI pipeline[18]
7. Configure a variable[19] of type `File` named `BUILD_ENV_OVERRIDE`
8. Paste the contents of `build-env.txt` there
9. Be careful with `PIPELINE_VERSION`: to avoid overwriting an existing build it is recommended to set a custom one
10. Run the pipeline

When the pipeline completes, the produced artifacts should closely match the original ones, albeit not being bit-by-bit identical.

Customizing the build

On the newly created branch in the forked recipe repository, changes can be committed just like on the main repository.

For instance, to install a custom package:

1. Check out the forked repository
2. Edit the relevant ospack recipe to install the custom package, either by adding a custom APT archive in the `/etc/apt/sources.list.d` folder if available, or retrieving and installing it with `wget` and `dpkg` (small packages can even be committed as part of the repository to run quick experiments during development)
3. Commit the results and push the branch
4. Execute the pipeline as described in the previous section

Example 1: OpenSSL security fix 2 years after release v1.0.0

Today a product team makes the official release of version 1.0.0 of their software that is based on Apertis. Two years from now a critical security vulnerability will be found and fixed in OpenSSL. How can the product team issue a new release two years from now with the only change being the fix to OpenSSL?

It is important for product teams to consider their future requirements at the point they make a release. To ensure bug and security fixes can be deployed

[15] https://docs.gitlab.com/ee/user/project/repository/forking_workflow.html#creating-a-fork

[16] https://docs.gitlab.com/ee/gitlab-basics/create-branch.html

[17] https://gitlab.apertis.org/infrastructure/apertis-image-recipes/commit/b4f1c5c85bd4603f2d9158f513c142a77a3c65c3

[18] https://docs.gitlab.com/ee/ci/pipelines.html#manually-executing-pipelines

[19] https://docs.gitlab.com/ee/ci/variables/README.html#create-a-custom-variable-in-the-ui

with minimal impact on users a number of artifacts need to be preserved from the initial release:

1. The image recipes
2. The Docker images used as build environment
3. The APT repositories
4. External artifacts

Getting started with Apertis: one year before release 1.0.0

Good news! A product team has decided to use Apertis as platform for their product. At this stage there are a few recommendations on how to get started that will make it easier to use Apertis long term reproducibility features.

The product team needs control over their software releases, and is important to decouple their releases from Apertis. One important objective is to give the product team control over importing changes from Apertis, such as package updates. We recommend using release channels for that.

A product team can have multiple release channels, each reflecting what is deployed for a specific product. And because release channels are independent and parallel deliveries, a single product may even have multiple release channels, for instance a stable channel and a development one.

In turn each product release channel is based on an Apertis release channel. As an hypothetical example the `automotive` product team may have an `automotive/cluster-v1` release channel for delivering stable updates to their `cluster` product, and an `automotive/cluster-v2` release channel for development purposes, both based on the same `apertis/v2020` release channel.

Git repositories need to use a different branch for each release channel, and each release channel has its own set of projects on OBS. However only the components that the product team need to customize have to be branched or forked. To maximize reuse, it is expected that the bulk of packages used by every product team will come directly from the main Apertis release channels.

1. **What**: Create a dedicated release channel
2. **Where**: GitLab and OBS
3. **How**: Create release channel branches in each Git repository that diverges from the ones provided by Apertis; set up OBS projects matching those release channels to build the packages

In this way the product team has complete control on the components used to build their products:

- Source code for all packages is stored on GitLab with full development history
- Compiled binary packages are tracked by the APT archive snapshotting system for both the product-specific packages and the packages in the main Apertis archive.

The previous step took care of the Apertis layer of the software stack, but there is one important set of components missing: the product team software. We suggest that product teams use one of Apertis recommended ways for shipping software which consists of using .deb packages or Flatpaks. For this example we are going to use .deb packages.

While there are multiple ways of handling product team specific software, for this example we are going to recommend the product team to create a new APT suite and a few APT components, and host them on the Apertis infrastructure. We will call the new suite cluster-v1. The list of APT repositories will then be:

```
deb https://repositories.apertis.org/apertis/ v2020 target development sdk
deb https://repositories.apertis.org/automotive/ cluster-v1 target
```

For reference, in APT terminology[20] both `v2020` and `cluster-v1` are suites or distributions, and `target`, `development`, and `sdk` are components.

The steps are:

1. **What**: Create new APT suite and APT components for the product team
2. **Where to host**: Apertis infrastructure

Creating the list of golden components: the day of the release 1.0.0

As we mentioned earlier each component is identified by a hash, and it is also possible to create tags. We recommend using hashes for identification of specific revisions because hashes are immutable. Tags can also be used, but we recommend careful evaluation as most tools allow tags to be modified after creation. Modifying tags can lead to problems that are difficult to debug.

The image recipe is usually a small set of files that are stored in a single Git repository. Collect the hash of the latest commit of the recipe repository.

1. **What**: Image recipe
2. **Where**: Apertis GitLab
3. **How**: Collect the Git hash of the latest commit of the recipe files

The Docker containers used for building are stored in GitLab Container Registry. The Registry also allow to identify containers by hashes.

There are expiration policies and clean-up tools for deleting old versions of containers. Make sure the golden containers are protected against clean-up and expiration.

1. **What**: Docker containers used for building: `apertis-v2020-image-builder` and `apertis-v2020-package-source-builder`
2. **Where**: GitLab Container Registry

[20] https://manpages.debian.org/testing/apt/sources.list.5.en.html

3. **How**: On the GitLab Container Registry collect the hash for each container used for building
4. **Do not forget**: Make sure the expiration policy and clean-up routines will not delete the golden containers

From the perspective of APT clients, such as the tools used to create Apertis images, APT repositories are simply a collection of static files served through the web. The recommended method for creating the golden set of APT repositories is to create snapshots using `aptly`. Aptly is used by Debian upstream and is capable of making efficient use of disk space for snapshots. aptly snapshots are identified by tags. Something along the lines of:

```
aptly snapshot create v1.0.0 from mirror target
```

Repeat the command for `target`, `development`, `sdk`, and `cluster-v1`.

It is important to mention that the product team needs to create a snapshot **every time a package is updated**. This is the only way to keep track the full history of the APT archive.

1. **What**: APT repositories:

   ```
   deb https://repositories.apertis.org/apertis/ v2020 target development sdk
   deb https://repositories.apertis.org/automotive/ cluster-v1 target
   ```

2. **Where**: aptly
3. **How**: create a snapshot for each repository using aptly
4. **Do not forget**: create a snapshot for every package update

External artifacts should be avoided, but some times they are required. An example of external artifacts are the multimedia files Apertis uses for testing. Those files are currently simply hosted on a web server which creates two problems: no versioning information, and no long term guarantee of availability.

To address this issue we recommend creating a repository on GitLab, and copy all external artifacts to it. This gives the benefit of using the well defined processes around versioning and tracking that are already used by the other components. For large files we recommend using Git LFS.

1. **What**: External artifacts: files that are needed during the build but that are not in Git repositories
2. **Where**: A new repository in GitLab
3. **How**: Create a GitLab repository for external artifacts, add files, use Git LFS for large files, and collect the hash pointing to the correct version of files

Notice that the main idea is to collect hashes for the various resources used for building. The partial exception are external resources, but our suggestion is to also create a Git repository for hosting the external artifacts and then collect and use the Git hash as a pointer to the correct version of the content.

At the time of writing there is work planned to automate the collection of relevant hashes that were used to create an image. The outcome of the planned work will be the publication of text files containing all relevant hashes for future use.

Using the golden components two years after release 1.0.0: Creating the new release

We recommend product teams to make constant releases, for example in a quarterly basis, to cover security updates and to minimize the technical debt to Apertis upstream. However in some cases a product team may decide to have a much longer release cycle, and for our example, the product team decided to make the second release two years after the first one.

For our example the product team wants the second release to include a fix for OpenSSL that corrects a security vulnerability, but be as identical as possible otherwise. A note of caution here is that deterministic builds, or the ability to build packages that are byte-by-byte identical in different builds, is not expected to happen naturally and is outside the scope of this guide. A good source of information about this topic is the Debian Reproducible Builds[21] page.

Our aim is to be able to reproduce builds closely enough so that one can reasonably expect that no regressions are introduced. For instance some non essential variations could be caused by different time stamps or different paths for files. These variations cause builds to not be byte-by-byte identical while the runtime behavior is not affected.

For our example the product team will import the updated OpenSSL package from Apertis, build the OpenSSL package, and build images for the new v1.0.1 release.

The first step is to rescue all the hashes that were collected on the day of the build.

Reproduce the build

The `build-env.txt` produced by the build pipeline should capture all the information needed to reproduce it as closely as possible:

1. Retrieve the `build-env.txt` from the golden build
2. On GitLab create a new branch[22] on the previously identified recipe repository. The branch should point to the golden commit which should be captured in the `RECIPES_COMMIT` field.

[21] https://wiki.debian.org/ReproducibleBuilds

[22] https://docs.gitlab.com/ee/user/project/repository/web_editor.html#create-a-new-branch-from-a-projects-dashboard

3. Execute a CI pipeline[23] on the newly created branch, reproducing or customizing the original build environment by creating a variable called `BUILD_ENV_OVERRIDE` into which the contents from `build-env.txt` should be pasted, modifying it as desired.

When the pipeline completes, the produced artifacts should closely match the original ones, albeit not being bit-by-bit identical.

Customizing the build

On the newly created branch in the forked recipe repository, changes can be committed just like on the main repository.

For instance, to install a custom package:

1. Check out the newly-created branch
2. Edit the relevant ospack recipe to install the custom package, either by adding a custom APT archive in the `/etc/apt/sources.list.d` folder if available, or retrieving and installing it with `wget` and `dpkg` (small packages can even be committed as part of the repository to run quick experiments during development)
3. Commit the results and push the branch
4. Execute the pipeline as described in the previous section

[23] https://docs.gitlab.com/ee/ci/pipelines.html#manually-executing-pipelines