APERTIS

Moving to Gitlab issues

# Contents



Apertis is an Open Source project which has been growing sustainable during the past years. This growth also made it spread across different projects and teams, requiring Apertis to improve the tools it uses. In this regard, one issue that prevents Apertis to be really open is the fact that the bug tracking system is only open to maintainers making it hard for the community to report new bugs or to keep track of them.

Since everyone can have access to Gitlab, the most reasonable approach would be to use Gitlab issues for the bug tracking. The current document describes the plan for the migration.

# Current status

Nowadays Apertis uses Phabricator[1] to track bugs but it is only accessible to users with privileges, such as maintainers or developers. One additional drawback is that Phabricator is an open source tool which is no longer supported upstream[2].

[1] https://phabricator.apertis.org
[2] https://www.phacility.com/phabricator/

## Organization

Currently bugs are treated as Phabricator **tasks** with a bug related tag such as **bug**, **bug (warranty)**, **bug (not-warranty)**. The meaning of these tags are only important for the management team inside Apertis, and has no value for the community.

Bugs can also have additional tags which allows:

- Release: Bugs can be tagged with a release such as **v2021** or **v2022** to describe that the issue is present in specific releases.
- Test-failure: This tag is used to denote bugs that are reported during the QA process, either from an automated test in LAVA or from a manual test.
- Topic: Additional tags such as **infrastructure**, **licensing** can be added to allow easy filtering of possible connected issues.

The current workflow relies mostly on the warranty/not-warranty, test-failure and on the release tags.

## Fields in Phabricator

Phabricator supports special fields that help both developers and the management team to keep track of the progress. The list of fields are:

- Priority: Defines a priority for the issue, possible values: **highest**, **high**, **needs triage**, **normal**, **low**, default value **needs triage**
- Status: Keeps track of the progress of the work, possible values: **unconfirmed**, **confirmed**, **in progress**, **proposed**, **submitted**, **resolved**, **verified**, **closed**, **wont fix**, default value **unconfirmed**
- Visible to: Used to configure some tasks only visible to a group of people
- Tags: Project defined tags
- Subscribers: List of people that get a notification on changes, members of the management team.

## Templates in Phabricator

Currently Phabricator presents a pre-formated bug form[3] page to create a new bug task. This helps developers to keep a consistent format across bugs, as well as avoiding missing important information.

## QA Report App

Quality Assurance in Apertis is managed through QA Report App web interface[4] which summarizes the report for all the supported releases. These report are the result of running Apertis test cases[5] on images either from an automated

[3] https://phabricator.apertis.org/maniphest/task/edit/form/8/
[4] https://qa.apertis.org/
[5] https://qa.apertis.org/

or manual test.

Automated tests are run on image creation using LAVA and results are reported back to QA Report App. On test failures this application automatically generates either a new bug or append new entries if a previous open bug for the same test case is found.

Since some test cannot be completely automated a set of them are run manually. The results of these tests are submitted to QA Report App in order to have a common source of information. In case of failure, following the same approach as for the automated tests, either a new bug or a new comment in an open bug is introduced.

## Apertimes

Management team inside Apertis uses **apertimes** to track progress of the project. This tool retrieves task information from Phabricator to generate reports.

## Workflow with Phabricator

The current workflow used for bug tracking can be described through these steps:

- Bug report: A Phabricator task is created when a new bug is found, there can be three situations:
  - Manual report base on user work: In some cases a user performing some kind of work might notice some unexpected behavior. In this case the user is encourage to create a new bug using the bug form[6].
  - Manual report based on test suite: Apertis provides a test suite[7] for its releases, some of which need to be run manually. When the QA process is run a failure is reported through the bug form[8].
  - Automatic report based on LAVA jobs: As mentioned the test suite[9] includes automated tests that are run on LAVA. QA Report App automatically creates a bug on failures.

On bug reporting it is important to try to include as much accurate information as possible. In this regard, a proper title, description and tags can help on later steps.

- Bug triage: On weekly basis the list of bugs is checked and new bugs are triaged with a priority as mentioned in Fields in Phabricator. Additionally, during this process for bugs with **highest** or **high** priority that require urgent attention, a developer is assigned. In other cases, developers claim a

[6] https://phabricator.apertis.org/maniphest/task/edit/form/8/
[7] https://qa.apertis.org/
[8] https://phabricator.apertis.org/maniphest/task/edit/form/8/
[9] https://qa.apertis.org/

bug based on the priorities and their skills. Additionally, tags can be added to make it easier to connect related bugs and to help the management team to keep track of the task in progress.

- Once a developer starts to work on a bug they change the **status** to **in progress** and updates the task regularly with the progress.
- After debugging, a solution is designed and proposed, usually by submitting a Draft Merge Request[10] to discuss the approach. When doing this **status** is updated to **proposed**, to show that there is a proposed possible fix for the issue.
- During the review process different things can happen
  - The proposed solution is accepted by a developer with experience in the field in which case it can be merged. This is usually the case for small fixes.
  - The proposed solution is accepted but additional work needs to be done to fix the issue. This goes from additional work on the MR or a new MR with additional changes. In this case when additional MR are submitted the **status** can be changed to **submitted**.
  - In some cases the root cause of the problem is in an upstream issue, from which information or feedback is required. Under such circumstances, the **status** can be updated to show that the developer is waiting for upstream by setting it as **upstream**.
- When all the MR are merged and no additional work needs to be done the **status** should be updated to **resolved**.
- Finally, the reporter can check if the issue is solved in which case the **status** should be changed to **verified**. If the reporter has doubts about the issue, he can request additional information and set the **status** to **need info**.
- In some cases, after investigating a bug, evaluate its impact and the possible solutions the best action is not to fix it. A good example of this case are packages no longer supported. In order to make this clear the **status** should be updated to **wont fix**.

During all the steps described any developer can add comments to help to fix the issue. It is also possible that members of the management team queries the the status of the bugs.

# Proposal

As previously mentioned, given all the Apertis related projects are already hosted in Gitlab, the use of Gitlab issues is natural. For this, Gitlab provides

[10] https://gitlab.apertis.org/dashboard/merge_requests?scope=all&utf8=%E2%9C%93&state=opened&assignee_id=None

a per project issue tracking system which is the best approach to handle development. Gitlab also support **incidents**, but those are focused on service disruption.

However, since Apertis is an Open Source Distribution, the mindset is different, each individual upstream project usually has its own bug tracking system, while Apertis focus on tracking issues from a distribution's integration point of view.

Also, requiring contributors to first find the right project to report the issue will not be user friendly, and as consequence some users will avoid reporting the problem or possibly report it to the wrong project.

For these reasons, the recommendation is to create a new single project for bug tracking, which will also help the management team, since there will be a new different issue id for every new issue. This is the same approach Debian uses in its bug tracking system[11].

This approach has also the advantage of making transition from Phabricator easier as issues will be migrated to only one project.

During the migration open bugs in Phabricator will be added to Gitlab with title, description, priority, status and tags set. In the description, a link to the Phabricator task will be added to more easily check the history. Importing the history is not recommended since there could be sensitive data, additionally comments also wouldn't have a correct datetime set. For closed bugs, since there is no value on importing them without the history the recommendation is to keep them only in Phabricator.

After the import, which is one time operation, Gitlab issues will be the main repository for open or new bugs.

The use of tags in Phabricator can be emulated in Gitlab with labels, making the transition almost transparent for end users.

However the use of tags for private management is discouraged since all information is made public. This topic will be covered in more detail in the Management data and view section.

## Fields in Gitlab

Gitlab issues does not provide the same fields, however, similar functionality can be implemented using labels[12]. To do so, for each field a set of labels should be created for every possible value, providing also additional flexibility.

For example, to implement the **priority** field, a set of labels **priority: highest**, **priority: high**, **priority: normal** and **priority: low** should be created. The same approach can be used for **status** since it also has a defined set of possible values.

[11] https://bugs.debian.org
[12] https://docs.gitlab.com/ee/user/project/labels.html

However the visibility field does not make much sense on this approach since the plan behind this task is to openly show the bug tracking system to the community, in the same way other Open Source projects do. In case of an exception, Gitlab supports the use of confidential issues[13] which has visibility set to only team members.

Tags can be easily implemented by labels since they have the same behavior, providing a very flexible way of connecting issues.

Finally, subscribers can be handled using the Gitlab notifications[14]. Issues templates can be configured to notify people about new activity by sending an email allowing all team members to be informed. Gitlab supports different [notification levels] providing a way for users to configure them according to their needs. Since the template will issue a mention to a default list of subscribers, everyone in that list who have not disabled notification will receive an e-mail on bug report and in subsequent comments.

## Templates in Gitlab

Gitlab templates[15] are used to have a common pattern to report bugs. Gitlab supports creating templates using markdown and thus allowing an easy way to import the current Phabricator templates.

The management of templates is straightforward since they are file located in the **.gitlab/issue_templates** folder, so the creation, modification and deletion can be handled thorough **merge requests**, which also helps traceability.

## Permissions

As described in permissions[16], Gitlab supports different roles in a project and since bugs will be a new one, permissions can be easily adjusted as needed.

In order to provide public access to reported issues, contributors need to be assigned the role of **guest** to the project. In this way they can create issues but won't be able to change labels or assign them after creation.

## Management data and view

Gitlab issues is a powerful tool to keep track of different tasks, in this case bugs. However, due to the private nature of some aspects of these tasks some information should be placed in a different location.

On the public project useful information can be included to help the management team, as example new labels based on **bug-type** should be used:

[13] https://docs.gitlab.com/ee/user/project/issues/confidential_issues.html
[14] https://docs.gitlab.com/ee/user/profile/notifications.html
[15] https://docs.gitlab.com/ee/user/project/description_templates.html
[16] https://docs.gitlab.com/ee/user/permissions.html

- bug-type:apertis: To denote a bug that appeared in Apertis due to work done inside Apertis project
- bug-type:upstream: To denote a bug that appeared in Apertis due to a bug also present upstream

Any private information should be placed in a different location, such as a private GitLab project or an external tool, and the mapping between issues and management data should be performed manually.

The easiest way to implement this is to keep private information in Phabricator as it is the tool currently used, reducing the complexity of the solution. With this approach bugs will be handled on Gitlab issues but a Phabricator task will be created and linked to track private management information. To make this possible a tool should be developed to create a Phabricator task based on a Gitlab issue with the same title and a description, including a link, referencing it. The management team will use this task to track only private data. Lastly, since management data will remain available in Phabricator, **apertimes** tool doesn't require any update, making the transition easier.

For public data, to have a high level overview of bugs, Gitlab support boards[17] which provide a kanban style interface. Boards columns can be configured based on labels, so the initial approach could be to have a board which shows bugs grouped by priority, which should help tracking the most important issues. It is also possible to create multiple issue boards per project, to highlight other aspects, such as **bug-type**.

### Workflow

The current workflow will suffer only minimal changes after switching to Gitlab issues. All the steps described in [Workflow with Phabricator] can be followed using Gitlab issues, however, with the new approach task will be automatically created and linked to Phabricator to allow the management team to hold private data.

## Summary

The table below shows a summary for the mapping between Phabricator and Gitlab features/information:

| | Phabricator | Gitlab |
|---|---|---|
| Organization | Tasks with tag `bug` | Gitlab issues on new project |
| Priority field | Internal field | Label with prefix `priority` |
| Status field | Internal field | Label with prefix `status` |
| Asignee field | Internal field | Internal field |

[17] https://docs.gitlab.com/ee/user/project/issue_board.html

| | Phabricator | Gitlab |
|---|---|---|
| Templates | Supported | Supported |
| Permissions | Supported | Supported |
| Management view | Workboard | Gitlab boards |
| Information type | Public and private | Public information only |

# Export/import

A key point during the migration is the export and import task, which should end up with all the open bugs in Phabricator available as Gitlab issues.

The desired data to be migrated is:

- Title
- Description with a link to the original Phabricator task
- Priority
- Status
- Tags

In this regard there are some tools already available such as import from csv[18] and import from Phabricator[19]. These tools only allow to import title and description, so importing additional information requires building a tool based on the Phabricator and Gitlab API. Creating such a tool will likely be rather simple though.

# Migration

The migration should be planned with enough time before a quarter ends in order to have a stable system previous to the release dates. It is a pre-requisite that the QA Report App is updated to support Gitlab issues and a set of tools are developed, one to create and mirror Phabricator tasks based on Gitlab issues, and another to [Export/import] Phabricator tasks into Gitlab issues.

For the actual migration, first a clean up of the current open bugs list should be performed, to avoid importing useless information. After this point, a freeze period should be started to guarantee that no modification to the bugs is made on Phabricator during the migration. This freeze period will be divided in two, a hard freeze and a soft one.

During the hard freeze the export/import operation will be triggered, which will end with a production test to confirm that all bugs have been migrated and tools work as expected. The main purpose of this tests is to spot any blocker for considering Gitlab issues as the main bug repository. For any no-blocker issue a task will be created. The estimation for this hard freeze period is one day.

[18] https://docs.gitlab.com/ee/user/project/issues/csv_import.html
[19] https://docs.gitlab.com/ee/user/project/import/phabricator.html

During the soft freeze period, adding bugs will be enabled only with strict control for just a few people to allow further tests to be run and to fix minor issues. The estimation for this soft freeze period is 3 days. In case that no issues are found the soft freeze period will be reduced.

If during the hard freeze period a blocker issue is found which will be unable to fix during the hard freeze, the migration will be aborted and a new migration schedule will be planned.

## Steps

- Create test environment on Gitlab
- Enhance QA Report App to report failure to Gitlab issues instead of Phabricator
- Create script to export bugs
- Cleanup bug list
- Hard freeze bug report/update on Phabricator
- Export bug list
- Import bugs list
- Perform production test
- Soft freeze bug report/update on Gitlab
- Additional tests and controlled bug report
- Unfreeze bug report/update on Gitlab

[Workflow with Phabricator] #workflow-with-phabricator