This document has been superseded. Please see Module Maturity Levels and Maintainer Role

Release Process and Module Life Cycle

Introduction
- Objectives
- The how…

From there…

Goals

Identify addons maintainers

Who gets write access to the repository?

Quality Levels
- Level 1 - Beta / Uncertified
- Level 2 - Stable / Approved (Current level)
- Level 3 - Mature / Certified

Merging

Release / Distribution
- Pypi

Deprecation

Introduction

The OCA attempts to apply rules and processes and be an all-encompassing source of Odoo community modules. Moreover, those divisions are often arbitrary and those put in charge (PSC) often only have limited interest in a large number of the modules they are ‘responsible’ for.

Additionally, due to the effect of self interest, only modules of larger integrators or of wide interest are regularly reviewed and merged, and usually by their own staff. As a result, more rules are then put in place which suit them (as after all they are the contributors) just further exacerbating the issues.

This leads to less and less people joining us every year. Even strong community-minded partners have their own public repo because it is too costly to follow OCA processes for every single module they want to publish.

Objectives

- Remain Open Source minded and favor collaboration
- Continue to grow the community in contributors and contributions
- Keep quality, but encourage innovation (by accepting duplicates at a beta stage, natural selection instead of reviewers decision)

The how…

So the proposal is, OCA moves from being all things to all people to essentially the source for the most critical and/or popular modules (maintainer-tools, server-tools, TBD...). For everything else, OCA offers tools and audit assurance services.

So basically with this idea, anyone can submit a repo under the OCA umbrella following some rules[1]. It doesn't need to be "approved" by anyone if rules are followed.
Their benefits:

- OCA has a copy of copyrights of any submitted code (i.e CLA)
- CI Tools with state of the art checks
- Runbot to allow people to test and discover OCA work
- Visibility for their modules (through common channels)
- Packaging (pypi)
- Manage/deploy their stuff as they want

From that point, people can define their own rules among their repos and manage them as they want (review process, etc..). The only remaining constraints is a green CI.

On top of that, we need a clear way to get a certification for a given modules, where our current process is used (at least 2 reviews, x days before merging, etc…).

Our benefits:

- We open the door to anyone that want to share his work
- We welcome more contributors, ideas, diversity and modules
- The best will win over the others → meritocracy is back again
- We're not responsible anymore for all changes to be made on all repos
- Our workload will be lowered
- Decouple development and deployment

[1] Use the provided template for the repo (with proper readme, etc..), follow OCA guidelines for the structure and sign the CLA, provide a clear description of the repo content and goals, commit to the OCA values and objectives (mostly be Open Source minded).

The only exceptions is for our mandatory tools and modules (TBD, but maintainer-tools is one of them, may be some of server-tools, etc..). Here the actual policy still applies.

From there…

“Old” Document about the release process.

Goals

The OCA Board would like to encourage integration of new contributors without compromising on quality and overloading current contributors.
In the following document, we listed different options for the different steps of the process.

Identify addons maintainers

In large repositories it is sometimes difficult to identify who are the active maintainers of a given addon. This creates the following problems:

- the maintainer does not easily notice PR and Issues concerning his/her addons;
- when addons have no active maintainers, PSC members must take care of it, but they may be too busy or may not feel concerned by all addons in their repositories;
- it is difficult for contributors to detect addons which have no active maintainer.

In small, focused repositories (eg management-system, operating-unit, business-requirement, ...), this issue does not arise since the repo maintainers and addons maintainers are the same. In larger repos (server-tools, stock-logistics-*, financial-tools, bank-payment, ...), it happens that some or all PSC members are not interested in all addons and the repo suffers of the above drawbacks, therefore giving a negative impression on the whole repo.

Splitting large repositories so they all become focused with a homogeneous content seems to be impractical, due to the difficulty to manage a large amount of repositories.

A solution to alleviate these issues is to explicitly identify maintainers of each addon, with the following benefits:

- The possibility for the bot to ping maintainers when a PR touches their code;
- TO INVESTIGATE: is the code owner github feature useful for that?
- (TBD how to do something similar for issues?)
- The possibility for contributors to easily identify the best persons to ping to have its PR reviewed or discuss issues.
- The possibility to identify addons that lack active maintainer, and possibly run “adopt an addon” campaign.
- Have adapted policies for maintainers, eg to let identified addons maintainer do maintenance operations with little or no review (eg simple forward/back ports, etc).

- Increase the sense of ownership for identified maintainers, with expected shorter review cycles.

Who gets write access to the repository?

This question must be answered to continue with this topic.

Current *status quo* is that only “repo maintainers” have write access. In general these are the OCA core maintainer team, plus the PSC members. For particular repos, additional people may be named maintainers and granted write access.

Wider write access, keeping controlled merge criteria, could be achieved through a **merge bot**. It would allow other people, without actual write access, to merge PRs through commands in a PR comment. The bot could even implement validations, such as enforcing for criteria to be met according to the module’s maturity level.

Quality Levels

Quality levels are stored in the manifest key “**development_status**” (development status is the term used on PyPi). Possible values should be the same as the ones used by PyPi: alpha, beta, production/stable, mature.

For better visibility of the quality level, an appropriate banner or tag should be placed at the top of the README. To not burden too much contributors with boilerplate tasks, ideally the README should be automatically generated from a template, by MQT, MergeBot, or a nightly script. An implementation suggestion is to have the README content in a “description.md” file, used to then generate the final “README.md” file.

All modules, regardless of the quality level, will be hosted in the same repos and branch, alongside the existing “stable” modules. They will also be published on PyPi and AppStore, just like “stable” modules.

A module’s Quality/Development Status can be different for different versions. A “mature” or “stable” module in a version can start as “stable” or “beta” in the next branch, as a step towards maturity.

Level 1 - Beta / Uncertified

Beta modules allow for the gradual development of OCA stable modules.
The work can be split into several smaller pull requests, that are easier to review and may be managed by several people.
Each pull request is an iteration providing a correct set of features. The corresponding code should pass TravisCI checks and should be peer-reviewed.

Since a “beta” module is a work in progress toward a “stable” module, it may not be suitable for use in production. Without notice the design and implementation may change in an incompatible way, the development work may halt, and it is even possible that it may be abandoned and deleted from the repository.
As such, it should carry a prominent banner warning about the unstable status of the module.

The module incubation workflow looks like this:

a) Create a “WIP” Issue for the module, used to coordinate the work and different PRs related to it. It should state the final goal and describe the next work units to be done.
b) Create PRs to implement work units described in the WIP Issue. They must pass CI tests and peer review.
c) Once contributors feel that the module is complete, promotion to “stable” status can be proposed through a PR removing the beta banner from the README.
d) If the module has not been migrated to version X-1, then the module can be deleted when version X branch is created.

This workflow also supports the functional design stage, prior to the implementation:

- Before the “WIP” Issue, an “RFC” issue (complemented by a Google Doc or similar, if necessary) can be created for an initial discussion of the scope, desired features and the implementation design outline.
- Once a “WIP” Issue is created, the first PR should be to create the skeleton of the new module, including a basic README.rst, for user documentation, and a SPECS.rst file for the specifications and technical design.
- The next PRs should focus on the README and SPECS, to collaborate on the solution.

As a summary, “beta” modules should:
- at all times follow the OCA coding standards and ensure TravisCI green builds.
- carry a prominent banner at the top of the README, warning about the BETA status.
- have a “WIP” Issue to list the pending tasks and coordinate the work around them.
- Runbot can install it and has no conflicts with other installed modules, so that people can try it.

Level 2 - Stable / Approved (Current level)

Stable modules meet the current OCA quality requirements: green CI build passing all current mandatory checks, and peer-reviewed by at least two other people.
They have an “API stable” policy similar to the Odoo policy for stable branches, but more permissive allowing for feature additions and improvements.

- Level 1 criteria
- Module respects the current OCA Conventions (2 reviews, etc..)
-

Level 3 - Mature / Certified

Mature modules comply with the stable requirements plus other requirements. They are proven to be:

- Actively maintained: Module exists for at least one previous Odoo version.
- Meet top quality criteria: they have tests with at least 80% of code coverage, and have no lint beta message warnings.
- Stable across Odoo versions: in case that significant changes are made to the data model, automatic migration OpenUpgrade scripts are provided. API breakages must be documented clearly, and be accompanied with a change in the major version, 3rd digit)
- Well documented and have a changelog
- At least 2 contributors

Merging

Quality Level	**Conditions to merge**	**Green CI means...**
1 (automatic merge)	● Pull request is linked to an issue ● Green CI	● Module is documented ● Module can be tested on Runbot ● Module can be packaged and distributed
2	● Pull request is linked to an issue ● Green CI ● 1 review from a Contributor ● 1 review from Maintainer / PSC Member ● 5 calendar days or less if we have another review from a Contributor	● Same as Level 1 ● PEP8 ● Lint tests (deprecated code) ● (Scope of the current tests)
3	● Pull request is linked to an issue ● Green CI ● Test coverage > 80% ● 2 reviews from Contributors ● 1 review from Maintainer / PSC Member	● Same as Level 2 ●

	• 5 calendar days or less if we have another review from a Contributor	

Release / Distribution

Modules of all Quality Levels are pushed to Odoo Apps and Pypi.

Module Quality Level	**Pypi Development Status**
1	Beta
2	Production/Stable
3	Mature

Pypi

For reference, the Development statuses available on Pypi are:

- Pre-alpha
- Alpha
- Beta
- Production/Stable
- Mature
- Inactive

Deprecation

With the release of a new version of Odoo, the new branch is created without any module.