oi-userland - best practices

> ⚠ **Deprecation warning**
>
> The page has been migrated to https://github.com/OpenIndiana/oi-userland/blob/oi/hipster/doc/best-practices.md . Avoid touching this wiki page.

oi-userland build system is constantly evolving. So we tried to gather some 'best practices' for component's Makefiles and manifests.

1) **Ugly is wrong. Complex is likely wrong.**
If your Makefile becomes too complex or ugly, likely you are doing something wrong. Committer will more likely accept component, which logic it can understand,
looking at Makefile without actually running build.

2) **Try to use existing rule sets.**
Each Makefile usually is based on one of the rule sets in make-rules - for example justmake.mk for simple Makefile-based software,
configure.mk for autoconf-based components (sometimes it's used for autoconf-**like** components, for example components/web/nginx), cmake.mk for cmake-based components. If you see that rule set which you are using is not rather adequate, check if there's better rule set. Using cmake.mk based rules will likely allow you to create simpler Makefile for software, using cmake, than configure.mk. See rule 1.

3) **Try to set common macros in Makefile, rather than in component's manifest.**
Look for examples at templates/<ruleset>.mk. Almost all header field values
can be set in Makefile. For simple components sample manifest, generated by 'gmake sample-manifest', and your p5m file will likely differ only in copyright line
and perhaps some transform rules.

4) **Hide complex software-specific installation logic in patches and artificial Makefiles.**
Some software has installation logic, which is not suitable for us. Or just completely lacks it (for example, ships just some scripts).
Usually it's better to correct this logic in patches. When it's completely missing, common pattern is to ship simple Makefile in $(COMPONENT_DIR)/files/Makefile and copy it to unpacked component archive in COMPONENT_PREP_ACTION.
If it's easier to achieve desired layout by some transforms in manifests or specifying alternative file/link path in manifest than patch software to do 'the right thing', do it in manifest.

5) **Prefer simplicity to generality.**
Don't over-engineer. You create component, which will be used as part of oi-userland and only on OpenIndiana. Don't create knobs and whistles.
Component is delivered to user in compiled form. Nobody will touch your knobs except the next person who would like to update component.
The only exception to this rule concerns upstreamable patches, which possibly should be rather generic to be accepted.

6) **Specify REQUIRED_PACKAGES in your component's Makefile.**
For this you can do 'gmake REQUIRED_PACKAGES'. This allows other users to check that they have sane build environment.

7) **Place component in corresponding category.**
When in doubt, look at packages FMRIs.

8) **If you can, provide COMPONENT_SIG_URL .**
This allows everyone to check that tarball was not compromised. Also this helps with updating component - you can update COMPONENT_VERSION and tarball will not be removed because of wrong checksum. Nonetheless, update component's archive's checksums when necessary even when component has COMPONENT_SIG_URL set.

9) **Provide sample-manifest, if it's not empty.**
Always include manifests/sample-manifest.p5m in your PR. This simplifies component's updates (you can easily find out which files (dis)appeared and correct your p5m manifests just using 'git diff manifests/sample-manifest.p5m').

10) **Format manifests similar to one, generated by sample-manifest.**
Usually it includes items in the following order:

- copyright header (don't forget to update it, <contributor> can't be a copyright holder);
- basic set actions (pkg.fmri, classification and so on, usually you don't touch generated part - see 3rd rule);
- license action;
- necessary transforms;
- group and user actions;
- file,link and other actions, which don't match those in sample-manifest (e.g. deliver configuration files from $(COMPONENT_DIR)/files directory);
- necessary depend actions (if you can, use TBD 'fmri=__TBD' depend actions);
- contents, generated by gmake sample-manifest.

11) **Rule of three publishes.**
Your component should be rebuilt and republished correctly, when its Makefile is touched. And almost always only when its Makefile is touched.
So, when you touch Makefile, it should be reinstalled and republished on 'gmake publish'. If you run 'gmake publish' twice, it could be republished due to some
peculiarities in publishing logic. If you run 'gmake publish' three times and component is republished, something is wrong with your Makefile.

12) **Bump COMPONENT_REVISION when you change component.**
It's the easiest way to trigger component's rebuild (see 11), and also it is reflected in packages' branch ids, so you can clearly find out which package version is installed.
Usually initially you don't set it and it defaults to 0. On subsequent component changes you bump it. If COMPONENT_VERSION is updated, you should remove COMPONENT_REVISION.

13) **Do not use spaces in Makefile's COMPONENT_SUMMARY**

It is due "known oi-userland issue - it wrongly escapes strings with spaces while handling PYV files"

14) **Provide system integration when necessary.**
The last thing everybody needs is to realize that after they install the package, they have no way how to run the service because it is missing (**valid!)** SMF manifest, RBAC profiles, desktop files, etc.
To provide a certain quality, packages should be integrated with the system at least minimally.

15) **Use test suites.**
If component has check target, usually you can use its test suite automatically, adding test target to your component. If component has $(COMPONENT_DIR)/test/results-$(BITS).master, test output is compared
to output in this file. You can perform sed/awk operations on test results, setting necessary operations in COMPONENT_TEST_TRANSFORMS. This allows to find regressions when component
is modified. If test files do not exist already, create empty test files $(COMPONENT_DIR)/test/results-$(BITS).master to trigger the test comparison stage. If tests are identical for 32-bit and 64-bit builds, you can create a unique $(COMPONENT_DIR)/test/results-all.master.

16) **Check if library updates are ABI compatible.**
You can check for instance on https://abi-laboratory.pro/tracker/ for a selection of common libraries.
Trigger rebuild of dependent components if necessary.
List them, assuming that you are updating 'library/foo', with pkg search -r -o pkg.name 'depend:require:library/foo'.

17) **When in doubt, look at existing components.**

18) **When in doubt and your question wasn't answered by taking a look at existing components, ask on IRC.**