



FNAL Spack / SpackDev status: the MVP

Chris Green, FNAL, 3 Oct 2018



Description

- MVP: Minimum Viable Product.
- “First look” for experimenters to try, vet overall direction, *etc.*
- Software stack for SL7 / GCC (C++17) only (optimized for profiling).
- Keep track of issues along the way but achieve the narrow goal first.
- “Pinnacle” of the software stack: the art suite.
- Everything built “our way” to maximize realism & compatibility for experiments.
- Use system-available packages where possible via **packages.yaml**.
- First demonstration of **cetmodules**.
- NOT “release”-oriented.
- NOT a solution to every problem.
- NOT a guarantee that every remaining problem *can* be solved.
- NOT a collection of every piece of software every experiment will need.

Detail

- Project wiki at <https://cdcvs.fnal.gov/redmine/projects/spack-planning/wiki/Wiki>.
- Instructions for MVP at <https://cdcvs.fnal.gov/redmine/projects/spack-planning/repository/raw/MVP/README.pdf>.
- Released 2018-08-29.
- Procedure outline:
 - Obtain and execute the bootstrap script.
 - Source the generated setup script.
 - Initialize a SpackDev area to develop (e.g.) the cetlib product from the art suite.
 - Develop, build and test the checked-out products within the correct environment for each package.
- Instructions include an heuristic for creating recipes for your own packages, and one for creating your own dependency tree.
- Comments on **everything** welcome, including instructions, wiki, scripts, *etc.*

Detail: anatomy of an MVP installation area

- `<spack-tools>`, an independent Spack installation containing the compiler and one or two other tools such as git.
- `spack/`, containing the Spack application ([FNAL-specific fork](#)), its accompanying code, all its package recipes, and some configuration.
- `fnal-art/`, containing recipes for the art suite, and into which you would place your own recipes.
- `<spack-data>`, an area containing all the other products to be built.
- `spackdev/`, containing the [SpackDev](#) application and its accompanying code.
- `spack_glue/MVP/`, containing the bootstrap script, these instructions [and more](#).
- `setup.sh`, a Bash setup script which, when sourced, will get your environment ready to initialize a SpackDev area and start developing.

[`<...>`: configurable, may be located outside the MVP tree.]

Detail: anatomy of a SpackDev development area

- `build/`, where the build takes place.
- `install/`, where products are installed.
- `srcs/`, all checked-out sources appear here. This also contains the generated `CMakeLists.txt` file.
- `spackdev-aux/`, containing tool wrappers and environment information for each package.

Detail: anatomy of a SpackDev CMakeLists.txt

- Front matter (project declaration, path variables, external project setup).
- One `ExternalProject_Add()` call per product to be developed, including:
 - Generator specification.
 - Full set of CMake arguments for project.
 - Build command (to match generator choice and ensure nested `make` commands share parallelism).
 - Dependencies on other external projects.

Detail: using a SpackDev development area

- Global build (no tests):
`spack load cmake`
`cmake --build build/`
- Per-package development (`--prompt` is a Bash-only enhancement):
`spackdev env --cd --prompt art`
`make ...`

Observations

- Currently source-only, so make sure you have space (and time) for the initial build of the external dependencies.
- Even without the dependency build, the initialization process is time-consuming due to the need for multiple concretization steps for non-trivial dependency trees.
- Relies on the existence of a pre-calculated dependency tree ([one is provided for the art suite](#)) including versions and variants appropriate for the particular release being developed.
- Global multi-package builds are available with coordinated package-level parallelism. Develop and test one package at a time using “`spackdev env`”.
- Support for Bash prompt labels to keep track of environment (contrib. for Zsh?).
- Support for IDEs via CMake’s secondary generator options (guinea pigs needed).

Observations

- Packages to be developed cannot be build-only dependencies of the other packages (e.g. cetmodules).
- Packages to be developed must be CMake-based (cetmodules is not required) at this time. `spackdev init` could conceivably handle other build systems if it were straightforward to extract the necessary build instructions from the recipe for insertion in `CMakeLists.txt`. Parallelism issues might be a concern, however.

Next steps

- Support the MVP and collect and incorporate feedback therefrom.
- Reformulate SpackDev as a Spack extension (facility available via [PR #8612](#)) to reduce the number of required concretization steps to (hopefully) one.
- Expand the repertoire of Spack recipes to include those required for LArSoft.
- Investigate how to incorporate Spack Chains (feature originally from Jim Amundson currently under development as a [PR](#)) and BuildCache into a coherent operational system.

Next steps

- Investigate and implement changes to Spack necessary to support:
 - Multiple releases and compilers.
 - Data-only packages.
 - No-source (“umbrella”) packages.
 - Development of product sets including build-only dependencies (e.g. cetmodules).
 - Layered releases (already-installed packages from prior releases, or central installations of the current release) in the face of evolved recipes (and likely different checksums for the same package).
 - Externally-based build system base classes such as (say) `CetPackage`.
- Develop release management tools.