Getting started with stack development

This page is deprecated.

This topic has moved to the DM Developer guide at https://developer.lsst.io.

Not all of the content may have moved yet. If you want to help, please contribute to the https://github.com/lsst-dm/dm_dev_guide GitHub repository. For assistance, contact a DM Documentation Engineer in the #dm-square channel.

Under construction

Details at DM-1641 - Jira project doesn't exist or you don't have permission to view it.

This document is an attempt to collect everything that an enthusiastic newcomer needs to know to get up to speed with LSST stack development. So far as possible, we assume little-to-no prior knowledge of how the various tools work. We link out to other information as appropriate. This guide is based on the experiences of the author (you may also read it as "things I found out the hard way"), and may not reflect best practice or received wisdom. See also the LSST Software User Guide and the LSST DM Developer Guide.

Structure & hosting of the stack

The stack is arranged as a series of separate but interdependent packages. An index of the LSST packages is available. Each package corresponds to a separate code repository. See material in the developer guide on LSST Code Repositories for details.

Note that if you are an LSST developer, you can register an SSH key for write access to the Git repositories. If you don't, you can still access the repositories in read-only mode. In both cases, instructions are available.

Building the stack

There are two separate systems for building the complete stack: one based on the EUPS package manager, and the other based on the The LSST Software Build Tool. The latter is fundamentally targeted at supporting the buildbot continuous integration system; while it may sometimes be helpful for end users or developers, we neglect it here in favour of the simpler route using EUPS.

The basic procedure is well covered in in Building the LSST Stack from Source. In short, this is a two-part process: first, we bootstrap the basic tools required to build the stack by downloading and running the `newinstall.sh` script. This installs EUPS itself, as well as ensuring that you have up-to-date versions of Python, git, Doxygen and SCons, all of which are required by the stack. In the second stage, this newly-installed EUPS is used to install the complete LSST stack in the form of the `lsst_distrib` package. **Be aware** that the guide linked above will install a specific version (or 'tag') of the stack, which may or may not correspond to the one you need: see the information on versioning below for more information.

It is worth emphasizing that all of the stack components – including the basic tools like git – bar EUPS itself are installed and managed using EUPS. For this reason, a grasp of basic EUPS usage is helpful.

At this point, the stack is installed and ready to use. If something went wrong, refer to the material on troubleshooting; otherwise, move on to test your installation.

Troubleshooting

A collection of pointers for when things go wrong.

Dealing with test failures

As the packages are being built, their test suites will be run. Any test failures will cause the build to come crashing to a halt with an error message. In general, of course, you want to figure out what the error was and fix it. In some cases, it can be convenient to press on regardless.

Here's an example test failure:

```
[ 40/71 ] cat master-ga9ffcc60e0+13 ...
***** error: from ${LSST_HOME}/EupsBuildDir/DarwinX86/cat-master-ga9ffcc60e0+13/build.log:
{ ... many lines elided for brevity ... }
running tests/timeFuncs.py... failed
1 tests failed
scons: *** [checkTestStatus] Error 1
scons: building terminated because of errors.
```

In this case, the build tests stored in `${LSST_HOME}/EupsBuildDir/DarwinX86/cat-master-ga9ffcc60e0+13/cat-master-ga9ffcc60e0+13/tests/timeFuncs.py` failed. The error itself is logged in `${LSST_HOME}/EupsBuildDir/DarwinX86/cat-master-ga9ffcc60e0+13/cat-master-ga9ffcc60e0+13/tests/.tests/timeFuncs.py.failed`; look there to see if it's fixable. If not, skip the test. Unfortunately, that procedure is a little involved. Our first step is to simply remove the failing test file *and the log of the failure*:

```
$ rm EupsBuildDir/DarwinX86/cat-master-ga9ffcc60e0+13/cat-master-ga9ffcc60e0+13/tests/timeFuncs.py
$ rm EupsBuildDir/DarwinX86/cat-master-ga9ffcc60e0+13/cat-master-ga9ffcc60e0+13/tests/.tests/timeFuncs.py.
failed
```

If we simply re-run the build, EUPS will grab a fresh copy of the package files, re-create timeFuncs.py, and our test will fail again. Instead, we need to adjust the build scripts and run them manually:

```
$ cd EupsBuildDir/DarwinX86/cat-master-ga9ffcc60e0+13/
$ vim build.sh # Comment out the line which unpacks a fresh "eupspkg tarball" (it's line 23 on the author's
system).
$ ./build.sh
```

The above will build and install the package, but now we need to tell EUPS that it's available. We do this as follows:

```
$ cd cat-master-ga9ffcc60e0+13 # Note that this is one directory level deeper than where build.sh was executed
$ eupspkg -er decl
```

Now you can simply start the build of the stack as before: EUPS will pick up where it left off.

Use consistent compilers

When building the stack with eups distrib, your C++ compiler will be invoked by running c++. When running scons directly, it may execute a different compiler by default. For example, on the author's system, three different compilers are available:

```
$ c++ --version
Apple LLVM version 6.0 (clang-600.0.56) (based on LLVM 3.5svn)
Target: x86_64-apple-darwin14.0.0
Thread model: posix

$ clang++ --version
clang version 3.5.0 (tags/RELEASE_350/final)
Target: x86_64-apple-darwin14.0.0
Thread model: posix

$ g++ --version
g++ (MacPorts gcc49 4.9.2_1) 4.9.2
Copyright (C) 2014 Free Software Foundation, Inc.
```

Take care that you use the same compiler to build individual packages as you used to build the full stack. The notes on SCons may be helpful.

Testing the stack

You already have some indication that things are working, since the build ran the test suite for each component. Testing the Installation describes a larger-scale test you can perform to check that everything is working properly. See also the examples of Using the LSST Stack.

Test results

Be aware that a stack build on Mac OS X will *not* pass the test described above: the values produced do not exactly match the expected results.

This is being tracked as DM-1086 - Jira project doesn't exist or you don't have permission to view it.

Basic EUPS usage

EUPS, Extended Unix Packaging System, is a tool which makes it convenient to juggle complex, interdependent pieces of software like the LSST stack and its dependencies. It provides a system for installing software packages, managing dependencies between them, and ensuring that the user is presented with a consistent set of tools by manipulating their environment appropriately. A complete guide to EUPS is outside the scope of this page: refer to its manual for documentation, and see also this collection of handy tips.

Installing software with eups distrib

newinstall.sh installs git by executing the equivalent of:

```
$ source "$LSST_HOME/eups/bin/setups.sh" # configure the environment to use the newly installed eups
$ eups distrib install --repository=http://sw.lsstcorp.org/eupspkg git
```

You are at liberty to use the same technique to install whichever packages you like from the LSST repository. Get a list as follows:

```
$ eups distrib list --repository=http://sw.lsstcorp.org/eupspkg/
```

(NB, the repository is also read from the `${EUPS_PKGROOT}` environment variable. From now on we assume it's set there, rather than setting it explicitly.)

Setting up a package

Once a package has been installed, you can add it to your environment using setup. For example, here we manipulate the git package:

```
$ which git
/usr/bin/git
$ git --version
git version 1.7.1
$ eups list git
   1.8.5.2     b345 b5 (... other tags elided ...)
$ setup git
$ which git
/ssd/swinbank/stack/Linux64/git/1.8.5.2/bin/git
$ git --version
git version 1.8.5.2
```

(`eups list` will list all the tags which refer to a particular version – that list is trimmed here for clarity. What's a tag? See the next section.)

Note that the package has been installed in a version and architecture specific location: it's easy to install more than one version of a package and juggle back and forth between them as required. You can remove a package from your environment by simply running `unsetup ${PACKAGENAME}`.

Tags

Above we installed "the current version" of the LSST stack without worrying too much about what that actually means. In fact, EUPS uses a system of "tags" to identify a coherent set of packages. You can see the tags your system knows about by running:

```
$ eups tags
```

Initially, these correspond directly to the files ending in .list at https://sw.lsstcorp.org/eupspkg/tags/, but note that it's also possible to define local tags which aren't stored on the server. In theory, one could see the contents of a particular tag by running:

```
$ eups distrib list -t ${tag}
```

Unfortunately, at time of writing, a bug in EUPS will cause that to list everything, regardless of tag. However, you can see the contents of individual tags by grabbing the corresponding files directly from the web (ie, take a look at https://sw.lsstcorp.org/eupspkg/tags/b449.list and see exactly what's inside it).

By default, if you simply run `eups distrib install ${package}` it will figure out which version to install by according to its version resolution order, or VRO. Display this as follows:

```
$ eups vro
type:exact commandLine version versionExpr current
```

For the details of what this actually means, refer to the EUPS manual. In brief, this will install the version specified in the "current" tag unless another version is specified on the command line. An alternative is to specify a particular tag to install with a `-t` option:

```
$ eups distrib install -t ${tag} ${package}
```

Look at the material on version numbers, below, to help figure out which tag you actually want to install.

Development workflow

Now that the stack is installed, you will want to start contributing your own code. You can use EUPS to replace one (or more) components in the active stack with the component you are currently working on. For example, let's assume we want to work on the afw component. First we check it out from the git repository, then we tell EUPS we want to set up our newly-checked-out version as follows:

```
$ git clone git@github.com:LSST/afw.git
$ cd afw
$ setup -j -r .
```

The arguments to setup tell EUPS to just setup this particular package (ie, `afw`) without worrying about its dependencies (they are already handled through the rest of the stack), and to use the version in the current directory. We can now go ahead and build it with SCons:

```
$ scons
```

See also the `discussion of using SCons`. This newly compiled version of afw will now replace the afw component in the installed stack, but the rest of the stack will be unchanged. You can now work on this package, making whatever changes you like, and rebuilding with SCons whenever necessary.

Using SCons

SCons is the tool which actually coordinates and runs the build; it's somewhat analogous to a tool like `make`. SCons is a popular, freely available tool, and you are encouraged to familiarize yourself with its documentation. In addition, LSST builds upon SCons to provide a standard set of utilities, `sconsUtils`, which are used in building LSST packages; some of the features available are therefore unique to LSST. Useful options include:

`-j ${N}`	Run up to `N` concurrent processes when building.
`opt=${N}`	Set the optimization level to `N`.
`cc={gcc, clang,cc}`	Invoke the C++ compiler as `g++`, `clang++` or `c++` respectively. Note that eups distrib install will use `c++`, and see the notes on using consistent compilers, above.

In addition, the user may supply one or more targets for the build on the command line. Predefined targets for every package, provided by `sconsUtils`, are `doc`, `tests`, `lib`, `python`, `examples`, `include` and `version`; it is also possible to define additional targets for the package if required. By default, invoking scons without specifying a target is equivalent to building the `lib`, `python`, `tests`, `examples` and `doc` targets. Note that building `tests` will not only build but also execute the test suite for the package. Targets may depend upon each other (so that, for example, specifying `tests` will automatically cause the `lib` and `python` packages to be built, if necessary to run the tests). The system is smart enough to only rebuild files when required (e.g. when the code used to generate them has changed).

Note that `version` is a special target which generates Python code embedding information about the version of the package being built. This can then be imported as `lsst.${packagename}.version`.

Providing afwdata

When attempting to test `afw` (the "application framework" page), you will likely find the test suite does nothing:

```
$ scons tests
{... elided for clarity ...}
Warning: afwdata is not set up; not running the tests!
scons: done reading SConscript files.
scons: Building targets ...
scons: Nothing to be done for `tests'.
scons: done building targets.
```

The required `afwdata` package is large (over 6 GB at time of writing) and is not therefore included in the standard stack build, or, indeed, made installable using `eups distrib`. However, you can simply clone and setup the afwdata repository directly:

```
$ cd .. && git clone git@git.lsstcorp.org:LSST/DMS/testdata/afwdata.git && cd afwdata
$ setup -j -r .
$ cd ../afw
$ scons tests
{... now run...}
```

(You may also wish to `declare` the `afwdata` package to `eups` so that it's easy to refer to in future).

Understanding version numbers

There's a confusing panoply of version numbers attached to the DM stack and its various components & releases.

First, individual components:

- Each repository is versioned by git, and hence has an associated SHA1 for every commit (ie, a 40 character hexadecimal string like `ae20e2b580e6d2a9e785dc0d4f471b7ebed2fb45`).
- For distribution, this can be transformed into something of the format `${branchname}-g${first part of SHA1}`. For example, the above becomes `master-gae20e2b580`.
- If the same code is repackaged for some reason (say, the versions of its dependencies change), a `+N` tag may be be appended to the name. For example: `master-gae20e2b580+3`.
- When a specific version of the code is tagged in git (because it is about to become part of an official release, for example) then it can also be referred to by that git tag: `10.0`.

A coherent collection of packages can be tagged in EUPS, as discussed above. Installing a tag of this form enables one to install the complete stack.

- Tags of the form `bNNN` are generated by the continuous integration (Buildbot) system. These tags are occasionally and on no fixed schedule published to the outside world.
- Periodically through the 6 monthly development cycle, numbered releases are made. The first is tagged `vN_0`, the next `vN_1`, etc. At the end of a development cycle, the final `vN_M` is also tagged with the name of the cycle.

A list of releases is available in the Software User Guide. A couple of points may be worth noting:

- The release process has been being heavily worked on in late 2014, meaning that the final Summer 2014 release and the initial Winter 2015 release (`v10_0`) are only reaching the point of completion in December 2014. The work done now should ensure this is more streamlined in future.
- Those who are familiar with LSST planning will be aware of LDM-240, the Software Development Roadmap. This document also defines a per-cycle release, but uses version numbers which are *not* the same as those which are actually being used to tag the code.

In summary, a new developer starting to work with the stack is probably best off starting with the most recent release version, unless that is very old, in which case choosing a more recent Buildbot tag may be more appropriate.