Hyrax Module Integration

Deploying your module with Hyrax

Nathan Potter
James Gallagher

Version 1.2
April 2016

Contents

1 Terms

besstandalone

A standalone application that runs the entire BES execution stack using file base command input. Configuration file, command file, debug control, and output file are specified with command line options.

2 Overview

This guide is intended to help you distribute your Hyrax module so that others may include it in their server, either by building the source code or installing a binary package. This guide does not explain how to write a module or explain the Hyrax object-oriented software framework. However, it does provide help on the typical configuration management issues module developers face including testing, binary compatibility with Hyrax, and packaging a module as an RPM file for distribution.

Hyrax supports a number of different types of modules:

Data Handlers - Read data from some storage medium (file system, database, etc) and create DAP data objects in the Hyrax memory space.

Transmitters - A transmitter takes in memory DAP Objects and produces alternative serialization encodings such as JSON, CSV, NetCDF, etc.

Functions - Perform numerical and meta operations on DAP data objects in memory and make the results available as DAP objects.

While the specific code in every module is different, the tasks described in this guide apply to all of them. That is, while the parts of the Hyrax OO framework they use may differ, the configuration management issues are generally the same for each kind of module.

3 Tests For Your Module

The importance of making good tests for your module cannot be overstated. When changes get made both in your code or to the underlying dependencies (libdap4, etc.) unexpected things can happen. Automated tests, both unit-tests and integration tests, will allow you to easily check to see if things are still behaving as expected. When bugs are found, tests should be written that trigger the bug in the unpatched software and subsequently demonstrate that the “fix” has in fact been effective. By adding discovered bug tests to the testsuite its efficacy grows over time.

3.1 Unit-Tests

Unit-tests are good. They enable testing with very high coverage levels and provide an excellent way to isolate problems in specific functions/methods in your code. The BES component of Hyrax relies on `cppunit` (version 1.12.1 as of 2/25/2016) for its unit-test framework. The **bes/functions** module has examples of unit-tests although these don't differ from any standard use of CppUnit. However, we have found that there are some tricks to writing a Makefile.am that runs these tests.

In your configure.ac file, test for CppUnit and, if found define a symbol (e.g., CPPUNIT) using AM_CONDITIONAL. This can be used by your Makefile.am to control whether CppUnit is required, so people can build you code when they don't have CppUnit installed. In addition, take some care in how you define the tests. Here are the lines from the `Makefile.am` found in `bes/functions/unit-tests`:

```
# This determines what gets built by make check
check_PROGRAMS = $(UNIT_TESTS)

# This determines what gets run by 'make check.'
TESTS = $(UNIT_TESTS)

...

if CPPUNIT

UNIT_TESTS = CEFunctionsTest GridGeoConstraintTest Dap4_CEFunctionsTest \
TabularFunctionTest BBoxFunctionTest RoiFunctionTest BBoxUnionFunctionTest \
OdometerTest MaskArrayFunctionTest MakeMaskFunctionTest

else

UNIT_TESTS =

endif
```

Note that the automake variables check_PROGRAMS and TESTS are special to automake while UNIT_TESTS is just a name we picked for a make variable to hold the list of all the unit tests to run. Also note that if the automake conditional CPPUNIT is not defined, the UNIT_TESTS Makefile variable is set to null, disabling tests (presumably because CppUnit is not installed).

We use a header generated by autoconf when the unit tests need to read files. This helps ease getting the distcheck target to work correctly. The header (test_config.h) is built from test_config.h.in by a target in the Makefile.am that uses sed to substitue the autoconf variable

`${abs_srcdir}` for the characters `@abs_srcdir@`. Here's what the test_config.h.in file looks like.

```
#ifndef _test_config_h
#define _test_config_h

#define TEST_SRC_DIR "@abs_srcdir@"

#endif
```

This is used in the unit tests like so:

```
float32_array->parse((string)TEST_SRC_DIR +
      "/ce-functions-testsuite/float32_array.dds");
```

where "`/ce-functions-testsuite/float32_array.dds`" is a file in the unit-tests directory. Using grep, look in any of the *Test.cc file in bes/functions/unit-test and you'll be able to see many examples of this.

3.2 Autotest

We use `autotest` (part of GNU's autotools) for integration testing in Hyrax. Modules are typically tested using the *besstandalone* command-line application with *autotest* providing the test harness. This is the preferred test method because *besstandalone* behaves almost identically to the BES itself. The *besstandalone* application is run as a command-line tool, not a daemon process, and so is well suited for this type of script-based testing. The **bes/functions** module has good autotest examples.You can see examples of autotest code and the automake framework around it here.

The setup includes:

- Create a *Makefile.am* file that can compile your autotest file (*MyModuleNameTest.at*) and clean up things like the *bes.conf* and other generated files.
- Create a *bes.conf.in file* and edit it so that your module (and any other module it may need) is loaded and configured.
- Place a copy of the *handler_tests_macros.m4* file in your test directory.
- Create an autotest file for your testsuite, *MyModuleNameTest.at,* and add this line at the beginning of the file:

  `m4_include([handler_tests_macros.m4])`

The general pattern for creating a test is:

- In a subdir (say, *mymodulename/tests/bescmd*) Create a file with a BES command inside, preferably one that causes your module to be utilized.

- Add the test macro (using a relative path name) to the *MyModuleNameTest.at* file. Which test macro you use will depend on what type of output your command will be generating. For a binary response your autotest macro invocation might look like this:

  ```
  AT_BESCMD_BINARYDATA_RESPONSE_TEST(bescmd/mymodule_test.01.dods.bescmd)
  ```

- Test output is captured and is compared to the baseline file (The command file name with a "*.baseline*" suffix).
- At this point running "*make check*" should cause your test to run. Woot!

As is the case with the unit-tests, the *bes/functions* code provides a good example. The files MyModuleNameTest.at, handler_tests_macros.m4, Makefile.am, bes.conf.in (used to build bes.conf from a target in Makefile.am) and the bescmd files in the bescmd subdir form the bulk of this testing scheme.

4 Security

Hyrax modules are run in a server that will typically be exposed to access via the internet. Because of this it is strongly recommended that you perform due diligence with respect to evaluating your module's code from a security standpoint.

4.1 Hyrax Module Security Policy

Enforcing a security policy on code written outside of the OPeNDAP organization is at best impractical and, more realistically, impossible. We offer the following as a list of what we feel are the dos and don'ts of being a well behaved component of the Hyrax server.

- Modules should NEVER accept incoming network connection requests (For example: No listening on a socket).
- Modules should NEVER make undisclosed outgoing calls. If a module needs to retrieve a remote resource:
  - The remote resource must pass a whitelist examination where the whitelist contents are fully available via the modules configuration.
  - The remote access activity must appear in the BES log.
- Modules should use a static code analysis tool. In the following section we describe this in more detail.

4.2 Static Source Code Analysis

Module candidates should be run through the Static Source Code Analysis (SSCA) tool in use by Hyrax. Currently, SSCA for Hyrax and its modules is provided by the Coverity analysis apparatus: https://scan.coverity.com We have found that this free static analysis tool is quite good when compared to commercial tools. Note, however, that Coverity is free only for Open Source projects.

Coverity classifies issues with three levels: High Impact, Medium Impact, and Low Impact. *All High Impact and Medium Impact issues should be resolved before the module is released.*

There are a couple of ways to get Coverity to perform a scan of source code.

1. On way is to integrate Coverity into an existing Travis continuous integration (Travis-CI) tool. Once this is accomplished and the Travis-CI builds are working correctly the linkage to the Coverity scan tool can be made by coordinating a Coverity account with the Travis-CI build through the *.travis.yml* file for the module. And example *.travis.yml* from the BES component of Hyrax can be found here: *https://github.com/OPENDAP/bes/blob/master/.travis.yml*

2. The other way is to submit a build to Coverity manually. This involves downloading the *Coverity Scan Self-Build* bundle from the Coverity website and use to build a *Coverity Scan Self-Build tar-ball* which can be uploaded to Coverity for analysis.

5 Reliable Source Distributions

When developing a module always make sure that you are developing against an official release of Hyrax, either source distribution or using binary (rpm files) from an official release (both the *libdap4* and *bes* have *-devel* packages for developers). This will help ensure that your module works with the current release of Hyrax.

If your module depends on other libraries then you should statically link to them unless you're sure that they will be available where people install your module. To force linking statically with a library, either make sure only the static archive (the archive/library file ends with `.a`) will be found the linker or explicitly name the static archive (for the latter, instead of using `-lnetcdf` use `libnetcdf.a`)

5.1 dist and distcheck

To streamline building a RPM package for your module, it helps to test the autotools *dist* and *distcheck* targets. If you're using a CI tool (e.g. Travis-ci), arrange to run these in addition to *make [all]* and *make check*. The dist target will build a mymodule-version.tar.gz file suitable for distribution – it will have the configure script and Makefile.in files that people need to build your code without having to install autotools themselves. Running *distcheck* determines if the tar.gz file built by *dist* can be built so that the generated files are in a directory structure separate from the sources, something that is very handy when building RPM files.

Unfortunately, problems encountered when running *dist* and *distcheck* can be hard to diagnose. Here are some debugging tips for these valuable targets.

If the *dist* target fails to include a needed file, this often does not show up until *distcheck* is run. When *distcheck* fails, the first thing to look for is a missing file in the distribution. The automake part of autotools will include most needed files (anything assigned to a *_SOURCES variable plus anything it know should be part of the build), but may not realize that other files are required for the build. Use the automake (i.e., Makefile.am) variable EXTRA_DIST to list these files. If your code has a directory where every file in the directory should be included, you can add that to EXTRA_DIST as well. Searching for EXTRA_DIST in *bes/functions* will show you some examples of its use.

Another common problem encountered with *distcheck* is that the build part of the target fails during the check-phase. This often happens because the *distcheck* target stores the generated files in a separate source tree from the sources. When *distcheck* works, it erases the source and build trees. However, when it fails they are left in place and this simplifies the debugging process. In the directory where you ran make distcheck, go into the mymodule-*version* directory and you should see both the sources and a *_build* subdir. cd to *_build*, go into the directory where the tests failed and run *make check* there.

A common culprit with *distcheck* failures is using a relative path to specify a file in the Makefile.am (the same problem can occur elsewhere, but it's most often in Makefile.am). This is not an issue during a normal build, because typically people build the code in the same directory as the sources. But distcheck builds the code in a special (*_build*) subdirectory. Instead of using hardcoded relative paths (e.g., *../data/file.txt*) use the Makefile variables *${top_srcdir}* and *${top_builddir}* to name files in the source and build trees, respectively. You can find plenty of examples of this in the *bes/functions* software.

6 Binary Distributions - Packaging with RPM

The nominal deployment target for Hyrax is CentOS (version 6.6 as of Feb 29, 2016). If your module depends on any package not standard to CentOS or easily available via *yum* on the same, then you should statically link your modules binary to these package's libraries. This will increase the size of your RPM somewhat, but it will make your module immune to availability problems with its dependencies at the time of deployment.

Packaging with RPM makes distributing on RedHat-flavored[1] Linux easy. Here we'll look at a simple RPM creation by examining the *.spec* file used to drive the RPM production and the Makefile.am targets to trigger the build.

[1] RedHat-flavored includes CentOS and Fedora, but not Debian/Ubuntu. The OSX packager *brew* seems to be pretty easy to support using source tar.gz distributions built by *make dist* especially with the *distcheck* target works.

- The folks at rpm.org have a great resource on working with RPMs written by E.C. Bailey.
- Here are the Fedora project guidelines for making RPMs.

6.1 Installation

When a Hyrax module is installed from an RPM file:

- Libraries will be installed in **`/usr/lib/bes`**
- Configuration files are placed in **`/etc/bes/modules`**

This should be reflected in the *module.conf.in*, *Makefile.am*, and *module.spec* files.

6.2 Makefile.am

6.2.1 Make Sure The *dist* Target Works.

The *dist* target is a built-in automake target that is used by the RPM process to place source code into a special location to build.

Test the *dist* target like this:

1. Run "*make dist*"
2. Locate the tar file created by the *dist* target. (mymodule-#.#.#.tar.tgz would be something to look for)
3. Unpack the tar file.
4. Change directory into the unpacked distribution.
5. Run "*./configure*"
6. Run "*make*"
7. If it works, proceed. If it fails, fix it!
8. Remember that any files that you need to distribute with your RPM (sample data files, etc.) MUST appear in the *dist* target or they won't be in the RPM. Typically this means working with the *Makefile.am* to get this sorted.

Tip: The *dist* target may have problems compiling if you have failed to identify all of the headers and C(++) files in your *Makefile.am*. See section 4 "*A Binary Version Of You Module That Works With Hyrax*" for more help with the *dist* target.

6.2.2 RPM target

Your Makefile.am will need an rpm target:

```
rpm: dist
```

```
        rpmbuild -tb --clean $(RPM_OPTIONS) @PACKAGE@-@PACKAGE_VERSION@.tar.gz
```

... and it should depend on the automake *dist* target. The `-tb` option tells `rpmbuild` to build a binary (b) using a tarball (t). It will expect to find a **.spec* file in the tarball (so make sure to include the spec file in Makefile.am's EXTRA_DIST variable). The *--clean* option will remove the build tree once the packages are made. Note that, unless you tell it otherwise, *rpmbuild* will build the code in ~/rpmbuild/BUILD and leave the resulting packages in ~/rpmbuild/RPMS/<arch>.

6.2.3 Static linking

The idea behind all the linux packaging systems is to provide a well-managed collection of binaries for different distributions of the operating system. The RPM package 'ecosystem' contains a number of official sites that provide packages, with some more 'official' than others. For OS core technologies, software packages are readily available and system administrators can use the RPM/Yum tools to install them without any customization. However, most 'scientific software' falls outside the realm of core OS technology and so its binary packages are typically in 'other' distributions.

The existence of different package collections matters because many system administrators will not use packages from anything but the official package collections for the Linux distribution they are supporting. The most prominent of these other distributions is EPEL (Extra Packages for Enterprise Linux).[2] It comes from the Fedora project and its RPM packages technically will work on RedHat and CentOS Linux distributions as well as Fedora. However, since many SAs won't 'mix' package sources, if you are building a RPM to support CentOS, for example, expect that only packages from the official CentOS sources will be available and not packages in the EPEL collection.

To support third-party code your module needs but cannot get using a package system (because it is not available from the target OSs official package source), we suggest *statically linking* your module to that software. To statically link on Linux, you will need to build a static version of the dependency software and ensure that it is used by your module at link time. This can be fairly tricky, so spend some time making sure that wherever you build *your* package, you don't have a shared/dynamic version of the *dependency* where the linker will find it. Often C/C++ code that uses autotools will support this by having you build the dependency using *./configure* with the option *--disable-shared*.[3] In addition to linking with a static library, you need to omit that dependency from the RPM .spec file. If you don't, installers will still be required to have the dependency! With those two caveats, you can build a RPM package that SAs can deploy without requiring they get packages from 'unofficial sources.'

[2] https://fedoraproject.org/wiki/EPEL

[3] If your dependency uses cmake, use *-DBUILD_SHARED_LIBS:BOOL=OFF*.

What about supporting 'packages,' people that build RPM packages for general use? For this case you should have a second .spec file that lists all of you package's dependencies. It's pretty hard to test both of these spec files on the same host, but you should be able to write one of these spec files from the other. There is, however, one last trick. When you build a RPM using *rpmbuild -tb* you can only have one file in the source dist tar.gz file that ends in *.spec*. Yes, really. To work around this issue, and support both a nice statically-linked package for you to distribute and a .spec file for packagers to use, name your static-linking spec file *<module>.spec.static* and use *<module>.spec* for the packagers.

The only remaining work is to write your Makefile so it supports two targets to use with the two different RPM spec files is the set of Makefile targets. Here are the one we use with the BES software:

```
# Build linux RPMs

srpm: dist
	rpmbuild -ts --clean $(RPM_OPTIONS) @PACKAGE@-@PACKAGE_VERSION@.tar.gz

# NB: 'dist' builds a tar.gz package using automake in the CWD.
rpm: dist
	rpmbuild -tb --clean $(RPM_OPTIONS) @PACKAGE@-@PACKAGE_VERSION@.tar.gz

# This will link with everything it can find in the local deps dir,
# removing the need for EPEL
all-static-rpm: dist
	cp @PACKAGE@-@PACKAGE_VERSION@.tar.gz ~/rpmbuild/SOURCES
	rpmbuild -bb $(RPM_OPTIONS) bes.spec.static
```

The *all-static-rpm* Makefile target first copies the tar.gz source dist built by the *dist* target to the builder's *rpmbuild/SOURCES* directory. Then it uses rpmbuild with the *-bb* option that enables providing the command with the spec file (*bes.spec.static*). The source code tar.gz file to use is named in the spec file.

6.3 The .spec File

What goes in your RPM is governed by the content of the *.spec* file that is used by the *rpmbuild* application to build the binary RPM file. The *.spec* file defines the

- The name of the module package
- The version number of the module package
- The locations of the modules library files.
- The locations of configuration file, any other required files
- The names and versions of other required packages - libdap and bes should be listed

Rather than try to discuss all of the details about the *.spec* I'll leave that to Baily. Here we will illuminate by way of annotation a suitable *.spec* file for the Hyrax module example.

While reading a *.spec* file keep in mind the following::

- The leading # character is used to denote a comment.
  ```
  # This is a comment
  ```

- Data objects known as tags serve as key value pairs and are denoted by the : character.
  ```
  Summary:   This string is the value of the tag named "Summary"
  ```

- The % character denotes the beginning of a script, macro, list of files, directive, macro, or name. Which of these is decided by a combination of the characters following the % and its relative position in the *.spec* file. It's magic!
  ```
  %description
  The description script allows you describe your module package.
  ```

Bailey has an excellent breakdown of the *.spec* file content here.

The beginning of our example *.spec* file contains a series of tags that hold meta information about the RPM and the *.spec* file. The tags section is followed by the invocations of the various scripts, macros, directives and names needed to produce the RPM.

5.3.1 Annotated RPM .spec file

Here simple meta information about the module is provided along with the version number, build/install location and versioned dependencies.

```
Summary:         CSV module for the OPeNDAP Data server
Name:            csv_handler
#
# This is the software verion of the module. This is independent of the
# version information in configure.ac
Version:         1.0.4
#
# The Release is like the version number of the .spec file and
# should be incremented when the .spec  file is changed
Release:         2
#
License:         LGPLv2+
Group:           System Environment/Daemons
Source0:         http://www.opendap.org/pub/source/%{name}-%{version}.tar.gz
URL:             http://www.opendap.org/
#
# In these Requires tags we inform the RPM that the target system must have
# libdap-3.13.3 or newer and bes-3.13.2 or newer. This is used by YUM/RPM at
# install time, and will cause YUM to try to find and load  the right libdap
# and BES libraries for the module. These are the run-time packages your
```

```
# module needs. If there are others besides, libdap and bes, add them but
# see the discussion about static linking further down.
Requires:         libdap >= 3.13.3
Requires:         bes >= 3.13.2
#
# The BuildRoot is used to define an alternate build root. The name is a
# bit misleading, as the build root is actually used when the software is
# installed during the build process.
BuildRoot: %{_tmppath}/%{name}-%{version}-%{release}-root-%(%{__id_u} -n)
#
# The BuildRequires element lists the packages needed to build the source
# code. Note that these are the *-devel versions of the libdap and bes
# packages. Your code may require others.
BuildRequires:    libdap-devel >= 3.13.3
BuildRequires:    bes-devel >= 3.13.2
```

6.3.2 Scripts, Macros, %files list and directives, %package, Conditionals

```
# A textual description of the contents of the RPM
%description
This is the CSV module for our data server. It serves data stored in CSV-formatted files.

# The %prep script does the first part of the magic of making the build.
# Don't try this without it.
%prep

# The %setup macro does more startup magic by unpacking files into their correct locations.
%setup -q

# The %build script builds the module, configure and make. Remember that at this point the
# module software is built, these flags are for conditioning the RPM machinery.
%build
%configure --disable-static --disable-dependency-tracking
make %{?_smp_mflags}

# The install script installs the built module files and cleans up
# the libtool mess (*.la files)
%install
rm -rf $RPM_BUILD_ROOT
make DESTDIR=$RPM_BUILD_ROOT install INSTALL="install -p"
rm -f $RPM_BUILD_ROOT%{_libdir}/bes/libcsv_module.la

# Cleans up the build mess.
%clean
rm -rf $RPM_BUILD_ROOT

# After installation this %post script Runs the ldconfig script to add the
# newly installed module library to the list of available shared libraries.
%post -p /sbin/ldconfig
```

```
# After an uninstall this %postun script Runs the ldconfig script to remove
# the removed module library to the list of available shared libraries.
%postun -p /sbin/ldconfig

# The %files list indicates to RPM which files on the build system are to
# be packaged. The list consists of one file per line. The file may have
# one or more directives preceding it. These directives give RPM additional
# information about the file.
%files
# These %dir directives create the place for the conf file to live
%dir %{_sysconfdir}/bes/
%dir %{_sysconfdir}/bes/modules
# The %config directive moves the configuration file into place in the RPM way.
%config(noreplace) %{_sysconfdir}/bes/modules/csv.conf
# Specifies the software library will be installed.
%{_libdir}/bes/libcsv_module.so
# Specifies where the data will be installed.
%{_datadir}/hyrax/

# Specifies the documentation files to be delivered the RPM way.
%doc COPYING COPYRIGHT NEWS README

# Change log entries for the .spec file
%changelog
```

6.3.3 Example .spec file (Not annotated).

```
Summary:         CSV module for the OPeNDAP Data server
Name:            csv_handler
Version:         1.0.4
Release:         2
License:         LGPLv2+
Group:           System Environment/Daemons
Source0:         http://www.opendap.org/pub/source/%{name}-%{version}.tar.gz
URL:             http://www.opendap.org/
Requires:        libdap >= 3.13.3
Requires:        bes >= 3.13.2

BuildRoot:       %{_tmppath}/%{name}-%{version}-%{release}-root-%(%{__id_u} -n)
BuildRequires:   libdap-devel >= 3.13.3
BuildRequires:   bes-devel >= 3.13.2

%description
This is the CSV module for our data server. It serves data stored in CSV-formatted files.

%prep
%setup -q

%build
%configure --disable-static --disable-dependency-tracking
make %{?_smp_mflags}
```

```
%install
rm -rf $RPM_BUILD_ROOT
make DESTDIR=$RPM_BUILD_ROOT install INSTALL="install -p"

rm -f $RPM_BUILD_ROOT%{_libdir}/bes/libcsv_module.la

%clean
rm -rf $RPM_BUILD_ROOT

%post -p /sbin/ldconfig

%postun -p /sbin/ldconfig

%files
%dir %{_sysconfdir}/bes/
%dir %{_sysconfdir}/bes/modules
%config(noreplace) %{_sysconfdir}/bes/modules/csv.conf
%{_libdir}/bes/libcsv_module.so
%{_datadir}/hyrax/
%doc COPYING COPYRIGHT NEWS README

%changelog
```

7 A Version Response

The hyrax server provides version information for modules it loads and this is very useful when testing and debugging, especially when people email us with support questions asking why something doesn't work (or works in ways they don't expect). The version information can be accessed by adding 'version' to the hyrax servlet base URL. For example,

```
http://test.opendap.org/opendap/version
```

This shows all of the modules actually loaded by the server and is produced by the framework asking each loaded module to report its version at the time the 'version' request is made. Thus, this will show if modules that were initially loaded have since been 'unloaded.'

The only downside to this is that it must be explicitly coded into each module. In this section we explain how to do this; reference code is provided in the Appendix (see section A.4).

7.1 Specialize BESRequestHandler

You will need to implement a specialization of the class BESRequestHandler for your module. This class contains the base machinery for you handler to handle 'requests,' specifically the HELP and VERSION requests. A typical class definition looks like:

```
class DapFunctionsRequestHandler: public BESRequestHandler {
```

```
public:
    DapFunctionsRequestHandler(const std::string &name);
    virtual ~DapFunctionsRequestHandler() {}

    virtual void dump(std::ostream &strm) const;

    static bool build_help(BESDataHandlerInterface &dhi);
    static bool build_version(BESDataHandlerInterface &dhi);
};
```

The complete source file is in section A.1. The most important parts of this specialization are the build_help() and build_version() methods, that are used to build the HELP and VERSION responses. The software that implements these two methods is included in the appendix. You will also need to 'wire up' those methods so that they are used to respond to HELP and VERSION requests. This is done by registering them using the BESRequsetHandler::add_handler() method in your specialization's constructor, like this:

```
DapFunctionsRequestHandler::DapFunctionsRequestHandler(const string &name) :
    BESRequestHandler(name)
{
    add_handler( HELP_RESPONSE, DapFunctionsRequestHandler::build_help);
    add_handler( VERS_RESPONSE, DapFunctionsRequestHandler::build_version);
}
```

7.2 Install the Module on the Handlers List

Now the module can respond to HELP and VERSION requests, but for the BES to know that, the module must be added to the list of 'request handlers.' To do this, put the following call at the beginning of your module's initialize() method:

```
void DapFunctions::initialize(const string &modname)
{
    BESDEBUG( "dap_functions", "Initializing DAP Functions:" << endl );

    // Add this module to the Request Handler List so that it can respond
    // to version and help requests. Note the matching code to remove the
    // handler from the list in the terminate() method.
    BESRequestHandlerList::TheList()->add_handler(modname,
                                 new DapFunctionsRequestHandler(modname));
...
```

This will ensure that whenever a HELP or VERSION request is sent to the server, this module is asked for the correct information. There is a corresponding change to the module's terminate() method; see section A.1.

7.3 Modify the Makefile.am

The last change you will need to make is to add support for this in the Makefile.am. The Makefile.am files for the handlers contain the version information and pass it into the handler at compile-time. The appendix includes the Makefile.am from the bes/functions software that is the source for the other example code.

APPENDIX

A.1 Code for the Version and Help Responses

A.1.1 DapFunctionsRequestHandler.h

```
// -*- mode: c++; c-basic-offset:4 -*-
//
// DapFunctionsRequestHandler.h
//
// This file is part of BES DAP Functions handler
//
// Copyright (c) 2016 OPeNDAP, Inc.
// Author: James Gallagher <jgallagher@opendap.org>

#ifndef I_DapFunctionsRequestHandler_H
#define I_DapFunctionsRequestHandler_H 1

#include <string>
#include <ostream>

#include "BESRequestHandler.h"

class BESDataHandlerInterface;

/** @brief A Request Handler for the DAP Functions module
 *
 */
class DapFunctionsRequestHandler: public BESRequestHandler {
public:
    DapFunctionsRequestHandler(const std::string &name);
    virtual ~DapFunctionsRequestHandler() {}

    virtual void dump(std::ostream &strm) const;

    static bool build_help(BESDataHandlerInterface &dhi);
    static bool build_version(BESDataHandlerInterface &dhi);
};

#endif
```

A.1.2 DapFunctionsRequestHandler.cc

```
// -*- mode: c++; c-basic-offset:4 -*-
```

```
//
// DapFunctionsRequestHandler.cc
//
// This file is part of BES DAP Functions module
//
// Copyright (c) 2016 OPeNDAP, Inc.
// Author: James Gallagher <jgallagher@opendap.org>

#include "config.h"

#include "BESResponseHandler.h"     // Used to access the DataHandlerInterface
#include "BESResponseNames.h"       // for {HELP,VER}_RESPONSE constants
#include "BESDataHandlerInterface.h" // Used to get the Info object
#include "BESVersionInfo.h"         // Includes BESInfo too

#include "TheBESKeys.h"             // A BES Key can be used to supply help info

#include "DapFunctionsRequestHandler.h"

/** @brief Constructor for FileOut NetCDF module
 *
 * This constructor adds functions to add to the build of a help request
 * and a version request to the BES.
 *
 * @param name The name of the request handler being added to the list
 * of request handlers
 */
DapFunctionsRequestHandler::DapFunctionsRequestHandler(const string &name) :
    BESRequestHandler(name)
{
    add_handler( HELP_RESPONSE, DapFunctionsRequestHandler::build_help);
    add_handler( VERS_RESPONSE, DapFunctionsRequestHandler::build_version);
}

/** @brief Provides information for the DAP functions help request
 *
 * @param dhi The data interface containing information for the current
 * request to the BES
 * @throws BESInternalError if the response object is not an
 * informational response object.
 */
bool DapFunctionsRequestHandler::build_help(BESDataHandlerInterface &dhi)
{
    BESResponseObject *response = dhi.response_handler->get_response_object();
    BESInfo *info = dynamic_cast<BESInfo *>(response);
    if (!info) throw BESInternalError("cast error", __FILE__, __LINE__);

    bool found = false;
    string key = "BES.functions.Reference";
    string ref;
    TheBESKeys::TheKeys()->get_value(key, ref, found);
    if (ref.empty()) ref =
"http://docs.opendap.org/index.php/Server_Side_Processing_Functions";

    map<string, string> attrs;
    attrs["name"] = MODULE_NAME;
    attrs["version"] = MODULE_VERSION;
    attrs["reference"] = ref;

    info->begin_tag("module", &attrs);
    info->end_tag("module");

    return true;
}

/** @brief add version information to a version response
 *
 * Adds the version of this module to the version response.
 *
 * @param dhi The data interface containing information for the current
 * request to the BES
 */
bool DapFunctionsRequestHandler::build_version(BESDataHandlerInterface &dhi)
{
    BESResponseObject *response = dhi.response_handler->get_response_object();
    BESVersionInfo *info = dynamic_cast<BESVersionInfo *>(response);
    if (!info) throw BESInternalError("cast error", __FILE__, __LINE__);

    info->add_module(MODULE_NAME, MODULE_VERSION);

    return true;
}

/** @brief dumps information about this object
 *
 * Displays the pointer value of this instance
 *
 * @param strm C++ i/o stream to dump the information to
 */
void DapFunctionsRequestHandler::dump(ostream &strm) const
{
    strm << BESIndent::LMarg << "DapFunctionsRequestHandler::dump - (" << (void *)
this << ")" << endl;
    BESIndent::Indent();
    BESRequestHandler::dump(strm);
    BESIndent::UnIndent();
}
```

A.1.3 DapFunctions.cc

Only the relevant portions of this file are included here - you can find it online at: https://github.com/OPENDAP/bes/blob/master/functions/DapFunctions.cc.
Also note that there are no changes to the DapFunctions.h (header) file when adding support for these responses.

```
// DapFunctions.cc

// This file is part of bes, A C++ back-end server implementation framework
// for the OPeNDAP Data Access Protocol.

// Copyright (c) 2013 OPeNDAP, Inc.
// Author: James Gallagher <jgallagher@opendap.org>

#include "config.h"

#include <iostream>

#include <ServerFunctionsList.h>

#include <BESRequestHandlerList.h>

...

#include "DapFunctions.h"

namespace functions {

void DapFunctions::initialize(const string &modname)
{
    BESDEBUG( "dap_functions", "Initializing DAP Functions:" << endl );

    // Add this module to the Request Handler List so that it can respond
    // to version and help requests. Note the matching code to remove the
    // handler from the list in the terminate() method.
    BESRequestHandlerList::TheList()->add_handler(modname,
             new DapFunctionsRequestHandler(modname));

    libdap::ServerFunctionsList::TheList()->add_function(new GridFunction())

    ...

    BESDEBUG( "dap_functions", "Done initializing DAP Functions" << endl );
}

void DapFunctions::terminate(const string &modname)
{
    BESDEBUG( "dap_functions", "Removing DAP Functions." << endl );

    BESRequestHandler *rh = BESRequestHandlerList::TheList()->remove_handler(modname);
    if (rh) delete rh;

}

/** @brief dumps information about this object
  *
  * Displays the pointer value of this instance
  *
  * @param strm C++ i/o stream to dump the information to
  */
void DapFunctions::dump(ostream &strm) const
{
    strm << BESIndent::LMarg << "DapFunctions::dump - (" << (void *) this
          << ")" << endl;
}

extern "C" {
BESAbstractModule *maker()
{
    return new DapFunctions;
}
}

}
```

A.1.4 Makefile.am

```
# Automake file for functions
#
# 01/28/2013 Hacked up by jhrg
#

AM_CXXFLAGS =

AUTOMAKE_OPTIONS = foreign check-news

if DAP_MODULES
AM_CPPFLAGS = $(BES_CPPFLAGS) -I$(top_srcdir)/dispatch -I$(top_srcdir)/dap
$(DAP_CFLAGS)
LIBADD = $(DAP_SERVER_LIBS) $(DAP_CLIENT_LIBS)
else
AM_CPPFLAGS = $(BES_CPPFLAGS)
LIBADD = $(BES_DAP_LIBS)
endif

# These are not used by automake but are often useful for certain types of
# debugging. The best way to use these is to run configure as:
# ./configure --disable-shared CXXFLAGS=...
# or ./configure --enable-developer --disable-shared
# the --disable-shared is not required, but it seems to help with debuggers.
CXXFLAGS_DEBUG = -g3 -O0  -Wall -W -Wcast-align
TEST_COV_FLAGS = -ftest-coverage -fprofile-arcs

if BES_DEVELOPER
AM_CXXFLAGS += $(CXXFLAGS_DEBUG)
endif

# Set the module version here, in the spec file and in configure.ac
M_NAME=functions
M_VER=1.1.0

AM_CPPFLAGS += -DMODULE_NAME=\"$(M_NAME)\" -DMODULE_VERSION=\"$(M_VER)\"

SUBDIRS = . unit-tests tests

[The file continues...]
```

A.2 Hyrax Dependencies February 25, 2016

Hyrax comes bundled with a number of dependency project libraries. Most significantly, the libdap4 library provides the support for the internal data model used in the server. If a module is going to do pretty much anything with data inside of Hyrax, then it will be using the libdap4 to get the data and to transmit it. The libdap4 library requires a recent version of Bison (for parser generation) The current versions for libdap4 and its attendant Bison are:

`libdap4-3.17.0`
`bison-3.0.4`

The BES:

`bes-3.17.0`

These third-party libraries dependencies are used by the various Hyrax modules to read and produce scientific data file formats and provide general support for the construction of the server’s various components and are used by one or more of the modules we distribute in the 'official' Hyrax releases:

`cfitsio3270`
`cmake-2.8.12.2`

```
gdal-1.10.0
gridfields-1.0.5
hdf-4.2.10
hdf5-1.8.16
HDF-EOS2.19v1.00
icu4c-3_6-src
jpegsrc.v6b
netcdf-4.3.3.1
openjpeg-2.0.0
```

Beyond this Hyrax only relies on the basic set of libraries found under a typical CentOS-6.6 system.

If a module requires an additional dependency it MUST be available via YUM, or the module library must be statically linked to the dependency.

Recommendation: Statically link your Module and its dependencies in order to avoid potential version collisions/incompatibilities between common dependencies such as the netcdf, hdf4, and hdf5 libraries.

Modules should be built against the OPeNDAP lease RPMs: libdap4, libdap4-devel, bes, and bes-devel.

The OPeNDAP supplied modules are statically linked because (for most people) the modules work better this way as their dependencies are often not available via YUM distribution channels.

A.3 Packaging Your Module For Debian

We don't currently build packages for Debian Linux distributions, but we'd like to! Here's information on building `.deb` packages. We'll add the caveat that we have tried using alien and fpm to convert a Hyrax/BES/Module RPM to a deb file and not gotten something that end users could get to work. We're including references to those because YMMV. If you do get a Debian package to build and it works, no matter how you get there, please let us know!

- https://www.howtoforge.com/converting_rpm_to_deb_with_alien
- https://www.digitalocean.com/community/tutorials/how-to-use-fpm-to-easily-create-packages-in-multiple-formats
- http://www.debian.org/doc/manuals/maint-guide/
- http://www.debian.org/doc/manuals/developers-reference/