BES Introduction

Table of Contents

1. Introduction

This document describes the Back End Server (BES) used by Hyrax to read data, build response, perform data subsetting operations, and run small compute tasks that are proximate to data. The BES makes up the lower two tiers of the Hyrax data server.

The BES software was initially designed and written at the High Altitude Observatory (HAO) of the University Corporation for Atmosphere Research (UCAR) in the early 2000s by Jose Garcia, Patrick West, and Peter Fox.

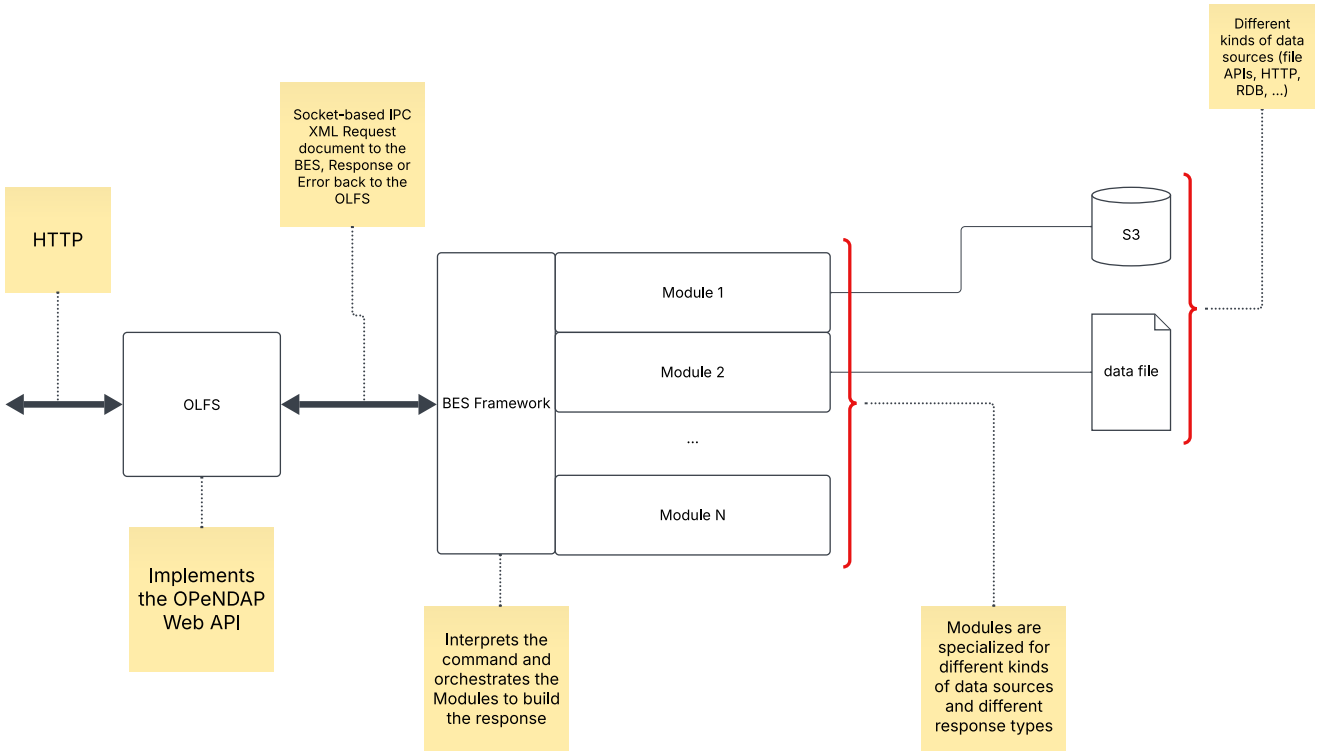

Figure 1. The Hyrax Data Server Architecture

The BES Framework and modules shown in Figure 1 are the subject of this document. The *Module* components of the BES are where the data- and response-specific behavior of the BES are encoded. The *Framework* is the orchestration engine of the BES. It interprets a command from the OLFS, plans how to build the requested response from the named data and how data are passed from one module to another.

In a broad sense, there are two kinds of modules: those that read from a data source (called a *BESContainer*) and those that build responses. The BES natively supports the OPeNDAP *DAP2* and *DAP4* data protocols as well as an extensible set of file formats that includes NetCDF3 and 4, JPEG2000, GeoTiff, ASCII/CSV and XML. Modules can not only read from files on a POSIX file system, but they can read from SQL databases, S3 and similar Web Object Stores (WOS) and other kinds of remote servers.

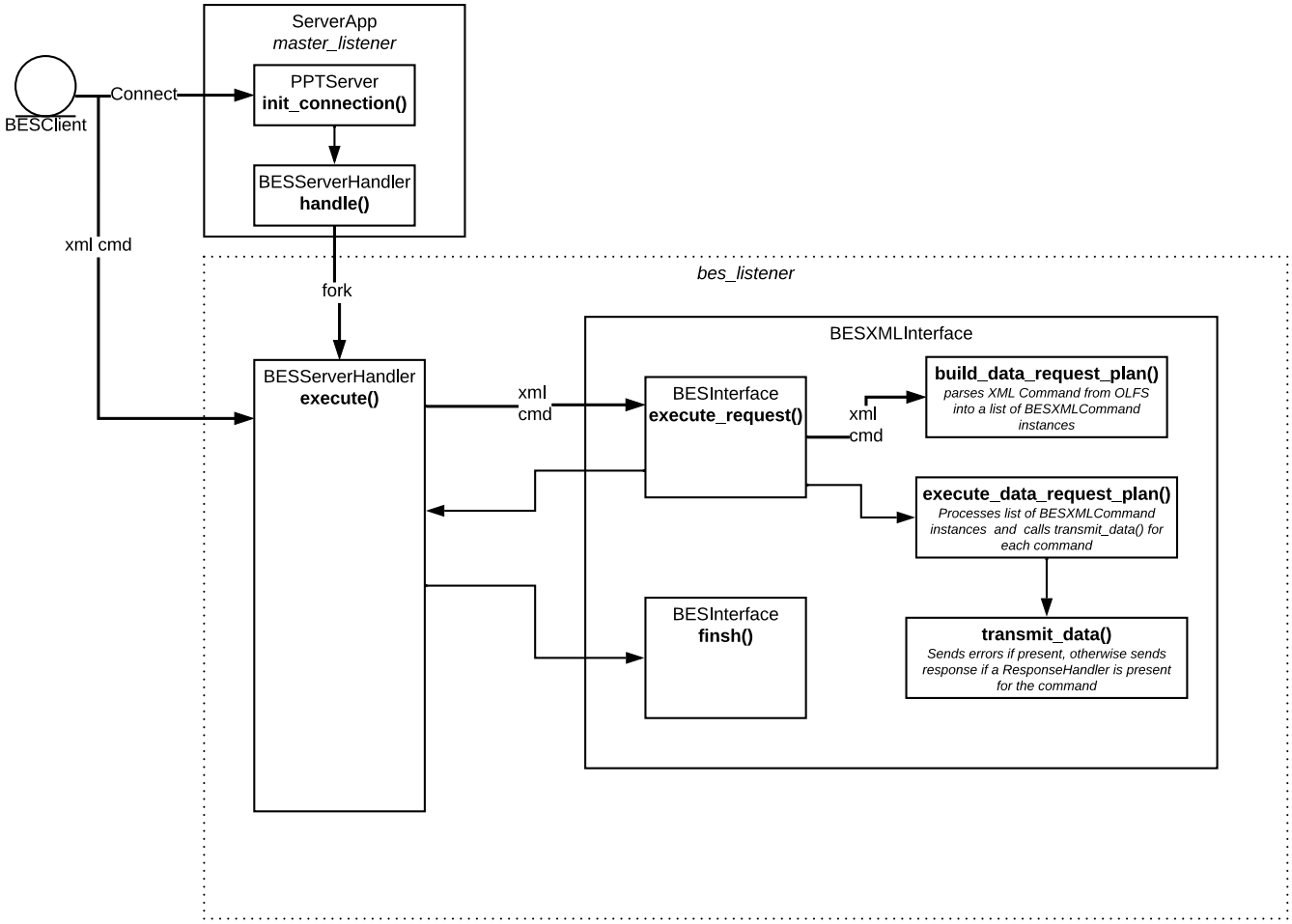

Figure 2. BES Framework Overview

More detail of the BES Framework is shown in Figure 2.

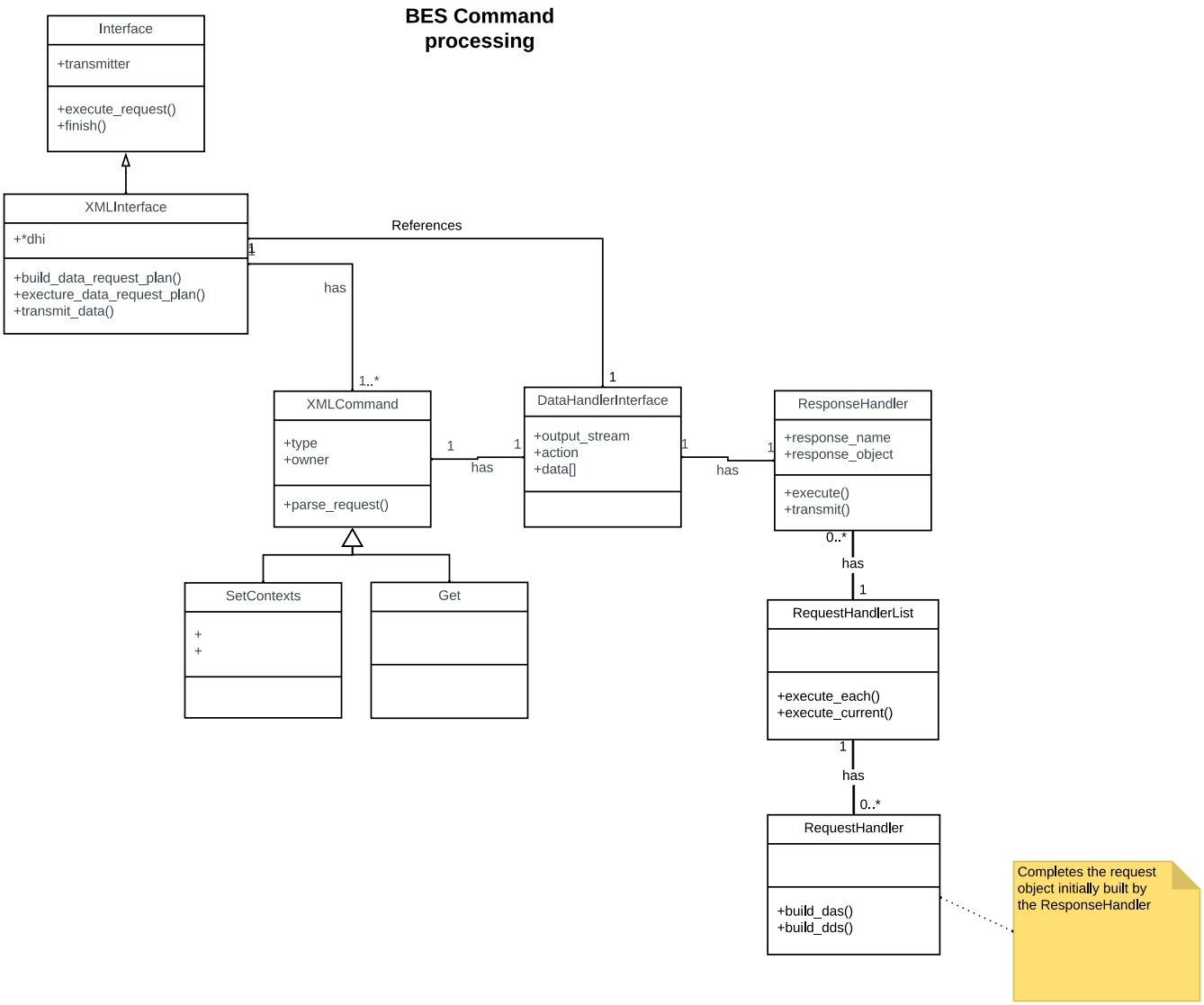

Figure 3. BES Classes used for Command Processing

2. BES Modules and Component Assembly

A module is an instance of BESAbstractModule is linked as a run-time loadable module. For example, Figure 4 shows how the `NCModule` class inherits from the BESAbstractModule class. The NCModule is the base of the module that reads data from the netCDF3 and 4 files, see Source from NCModule.cc.

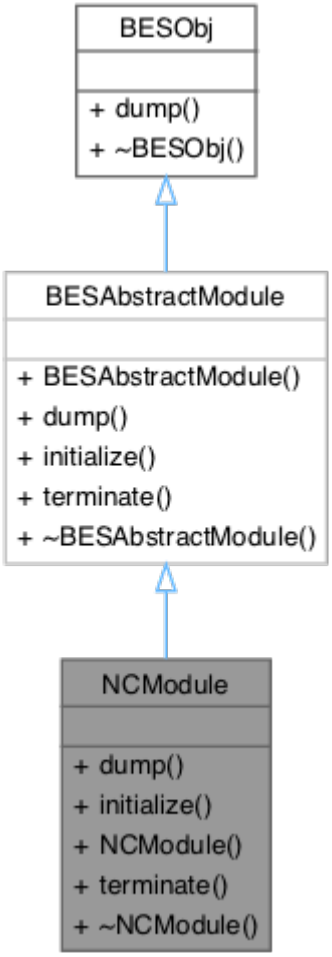

Figure 4. The NCModule Class

The BES daemon uses the code in the implementation of, e.g., NCModule to initialize the model at startup. Each specialization of BESAbstractModule should define `initialize()` and `terminate()`. The initialize() method is used when the module is first loaded. It performs all the startup actions the module needs. In Source from NCModule.cc the initialize() method makes some objects and passes those to methods that will add them to BES List (see Section [sec-bes-lists]) classes. The classes and the Lists are not important right now—know that when a module is loaded, the initialize() method is run.

When the daemon shuts down, each Module is 'unloaded' and its terminate() method is run. That is not shown in the listing, but you can find it in the GitHub repo *OPENDAP/bes* or in the online reference documentation.

2.1. How Modules are Loaded

In the BES daemon, a *master listener* is started, the configuration files are read, the BES Keys (our name for the configuration parameters) are parsed, and all the modules are loaded. The details of the server startup operations are discussed in Section [sec-startup], BES Startup.

One detail you will need to know, common to all modules, is that an extern C function named `maker()` must be included. This function is used when the module is loaded to make an instance of the Module class. Once the instance is made, a pointer to the module is stored in an instance of BESModuleApp. The BESModuleApp also contains the `loadModules()` method that runs each module's initialize() method.

2.2. What You Need to Know About Modules

The process of loading and unloading modules is interesting, but it is performed by parts of the BES framework that are rarely modified. You need to know that the initialize() and terminate() methods and the maker() *virtual constructor* must be included in a Module implementation.

In addition to the code of the module, the correct entries in the BES Keys must be made. To be loaded, a module must have two BES keys set correctly. The first is that the name of the module must be included in the comma separated list of names set to the `BES.modules` key. The module can have any name (assuming valid syntax) so long as it is not already in use. There is one Key called BES.Modules for the entire BES. See Example BES Keys for a Module.

The second BES Key is formed using the module name as a suffix to `BES.module`. The value of this key is full path to the shared object file that holds the module software and that will be loaded by the BESModuleApp using the unix/Linux `dload()` system call.

In the example Example BES Keys for a Module, the module name is `nc` so the BES Key that holds the path of the shared object files is `BES.module.nc`. The one key that lists all the modules is `BES.modules` (plural) while the key that is unique for each module is `BES.module.<mod name>` (singular).

In the Source from NCModule.cc example, the `modname` parameter passed to initialize() is the module name set in the BES Keys BES.modules and BES.module.<mod name>. It is important to be consistent with this name since it is literally the text string that forms the association between various parts of the module components. Follow the convention of always using that name in the calls shown in the example.

Example BES Keys for a Module

```
BES.modules += nc
BES.module.nc = /Users/jimg/src/opendap/hyrax_git/build/lib/bes/libnc_module.so
```

Source from NCModule.cc

```
#define NC_CATALOG "catalog"

void NCModule::initialize(const string & modname)
{
    BESDEBUG("nc", "Initializing NC module " << modname << endl);

    BESRequestHandler *handler = new NCRequestHandler(modname);
    BESRequestHandlerList::TheList()->add_handler(modname, handler);

    BESDapService::handle_dap_service(modname);

    if (!BESCatalogList::TheCatalogList()->ref_catalog( NC_CATALOG)) {
        BESCatalogList::TheCatalogList()->add_catalog(new BESCatalogDirectory(
NC_CATALOG));
```

```
    }
    else {
        BESDEBUG("nc", "    catalog already exists, skipping" << endl);
    }

    if (!BESContainerStorageList::TheList()->ref_persistence( NC_CATALOG)) {
        BESContainerStorageList::TheList()->add_persistence(new
BESFileContainerStorage( NC_CATALOG));
    }
    else {
        BESDEBUG("nc", "    storage already exists, skipping" << endl);
    }

    BESDebug::Register("nc");

    BESDEBUG("nc", "Done Initializing NC module " << modname << endl);
}

...

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

3. BES Containers and Container Storage

The BES using the abstraction of *Containers* that hold data to enable all the BES modules that read data to work with all the different kinds of data stores (files, S3, etc.) the BES is configured to use. To understand the flow of control the the BES Framework manages, it is important to understand how instances of BESContainer are used to abstract different types of data.

3.1. The components

1. A module (a concrete instance of the BESAbstractModule class)
2. A container storage object (a specialization of BESContainerStorageVolatile)
3. A container object (a concrete instance of BESContainer)
4. BES configuration parameters, usually in a `.conf` test file
5. An XML command document that includes a `<container name="…" space="…">` element

3.2. BESContainerStorage

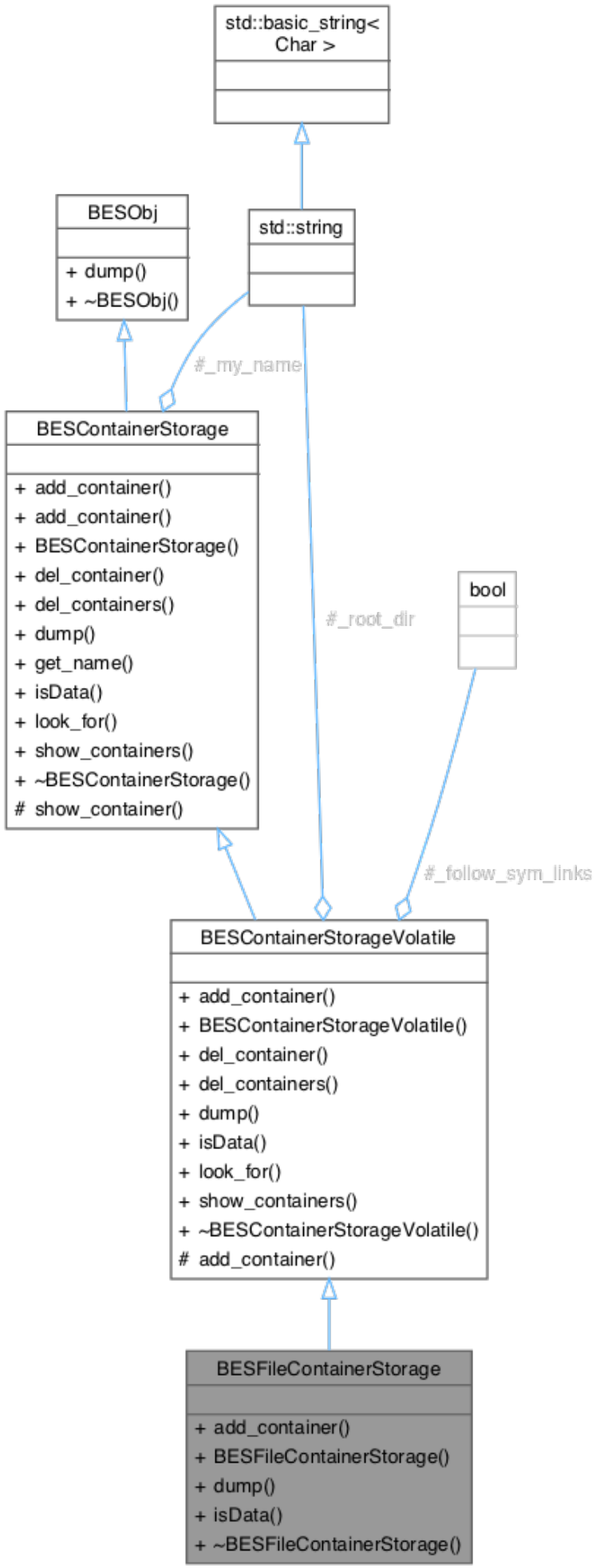

Figure 5. BESContainerStorageVolatile and BESFileContainerStorage

3.3. BESContainer

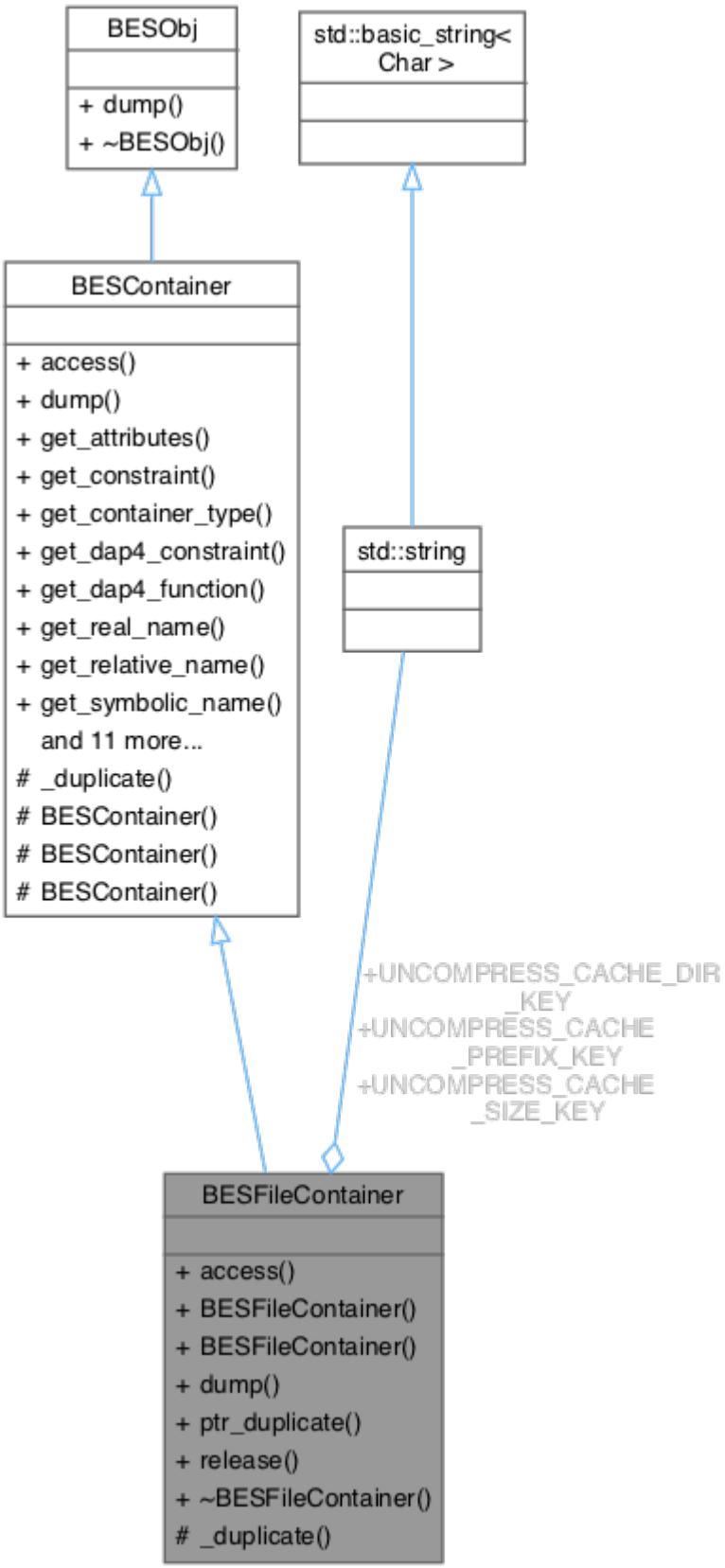

Figure 6. BESContainer and the Concrete BESFileContainer

4. BES Lists

The BES uses a number of singleton `*List` classes to manage named collections of objects that will

be used during command planning and evaluation.

TBD

5. BES Response Handlers

TBD

6. BES Transmitters

TBD

7. BES Server Startup

TBD