CS CAPSTONE DESIGN DOCUMENT

DECEMBER 2, 2017

COVERAGEJSON RESPONSE HANDLER FOR OPENDAP

PREPARED FOR

NASA JET PROPULSION LABORATORY

LEWIS JOHN MCGIBBNEY

PREPARED BY

GROUP55

RILEY RIMER
RIVER HENDRIKSEN
COREY HEMPHILL

Abstract

This document covers the high-level design of the components necessary to complete the CoverageJSON Response Handler for OPeNDAP project. It also defines the project's purpose, scope, and stakeholders, as well as the intended audience of the document, and the design rationale of the project developers. The component designs defined here include host architecture, data access protocol, data format, testing, and user interface.

Contents

1 OVERVIEW

1.1 Purpose

The purpose of this document is to describe the initial design plans that will be used to develop a CovJSON Data Response Handler for the OPeNDAP project. This document will cover the following sections from different viewpoints:

- The host architecture that will be used for HTTPS Network Handling and managing client requests.
- OPeNDAP, the protocol that the response handler will be implemented for.
- The CoverageJSON scientific coverage data format.
- The testing framework that will be used to ensure the reliability of the code we create.
- The web application framework that will be used.

1.2 Project Scope

This document will provide information on the design of components that are to be implemented in the finished response handler. The purpose of this project is to integrate a new response handler for the CoverageJSON data format into the currently existing collection of data response handlers within the OPeNDAP network data access protocol. Implementing this will involve interfacing with the OPeNDAP team as well as NASA Jet Propulsion Laboratory (JPL) to gather requirements and acquire a general understanding of the OPeNDAP code base. OPeNDAP does not currently provide a handler for a coverage data format like CoverageJSON, which is a structure that associates positions in space and time with the corresponding data values. The addition of the CoverageJSON format into OPeNDAP will allow for more fluid development of coverage data based web applications by NASA JPL and all other OPeNDAP users. Due to the desire for the new CoverageJSON response handler to be integrated into the OPeNDAP open-source project, the data handler will need to be accompanied by extensive documentation and testing. This project will also include testing current NASA satellite data in OPeNDAP on the proposed response handler and creating a promotional poster for the American Geophysical Union.

1.3 Intended Audience

This document is intended primarily for developers of the OPeNDAP protocol. However, this document may also be useful for NASA JPL, and other developers that intend to contribute to or implement the OPeNDAP protocol within their data server systems. From an OPeNDAP perspective, this document will act as a confirmation that their requirements are being met. For others, it will serve as a high-level view of how we plan to approach the design and implementation of the CovJSON response handler for OPeNDAP. This document may represent requirements that differ from the requirements that the client explicitly requested; if so, they may have been deemed necessary by the developers for the creation of the project. This document has not yet been reviewed by the client and will be revised at a future date.

2 DESIGN DESCRIPTION

2.1 Design Stakeholders

The design stakeholders of this project include both the users and the developers of OPeNDAP, Hyrax, and CovJSON. This includes Lewis John McGibbney and NASA JPL, as well as James Gallagher of OPeNDAP. The completion of the implementation of this design will help further accommodate web development on coverages, and extend the usability of the CoverageJSON format. OPeNDAP is also a stakeholder due to the intended incorporation of this project into OPeNDAP.

2.2 Design Rationale

The design for this project is heavily influenced by the fact that almost all of the work in the project is on systems that were already implemented. This leads the design to take heavy cues from the existing system implementations to avoid conflicting design and to remain consistent with the development standards of the project. This is most prevalent in the coding itself that will take place in OPeNDAP, where this project's implementation will need to align strictly to the current code base for OPeNDAP so it smoothly transitions into the OPeNDAP project upon completion. Due to the scope of the systems being worked on there is also a lot of information relevant to the design for this project that has not been discussed yet, which is why parts of the current design are rather shallow. This document will be periodically updated as more of the design planning is worked out with the current members of OPeNDAP, due to their integral nature in the final implementation of this project.

For the reasons discussed the writers of this document have decided to forgo some standards of the IEEE design documentation because they do not apply to the work that will come from this design. As such viewpoints have been removed in favor of components, as those are the parts of the design that must be considered for this project to be successful. To not completely remove the standards set in place the document still discusses the viewpoints for each component, this is so the developers can consider the constraints relegated to those views.

3 Components

3.1 Component: Host Architecture

3.1.1 Purpose

The purpose of having a dedicated host architecture is twofold. One, it allows for developers of this project to follow a standard that is already implemented by the architecture. And two it saves time in integration into OPeNDAP which already uses this host architecture. Having an HTTPS handler already in place is imperative for the creation and testing of a response handler.

3.1.2 Viewpoint

The view point for this component is logical since it is a system that is already in place and the development of the CovJSON handler will use libraries and methods already created in the Hyrax data server [1]. Besides the back-end conversion from the NASA JPL scientific data format to the CovJSON standard, all calls and functions to any networking system will be done with methods from the Hyrax data server.

3.1.3 Concerns

There are minor concerns with integration into the Hyrax data server. One limit is time, and the ability for the developers on this project to fully understand a complicated system like Hyrax within a limited time set may be an issue. However, the developers have resources for Hyrax such as employees and founders that may make this problem a non-issue.

3.1.4 Functionality

Hyrax's main purpose is broken into two parts, the OLFS (OLFS) and the BES (BES). Functionally the Hyrax data server is a means for data to be requested by a client from a server using OPeNDAP protocols [2]. The design of this project is entirely based around the use of OPeNDAP and thus must be based on the Hyrax implementation. Figure 1 shows how the clients system interacts with the Server [3].

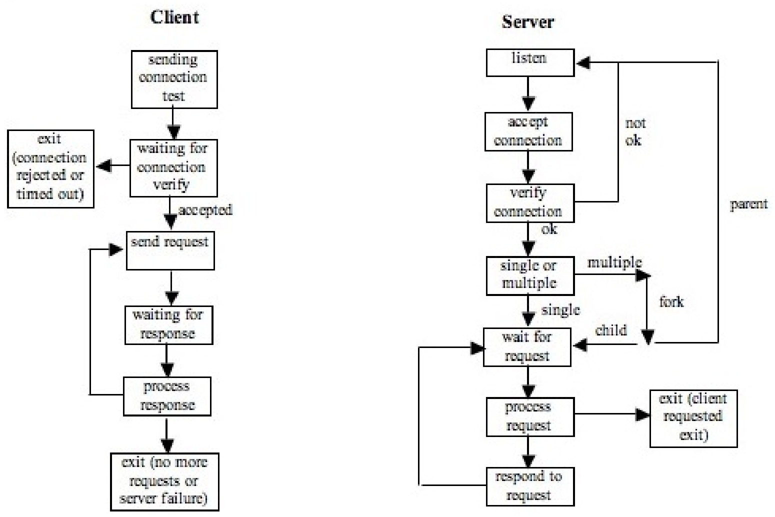


Fig. 1: Client Server relationship Hyrax

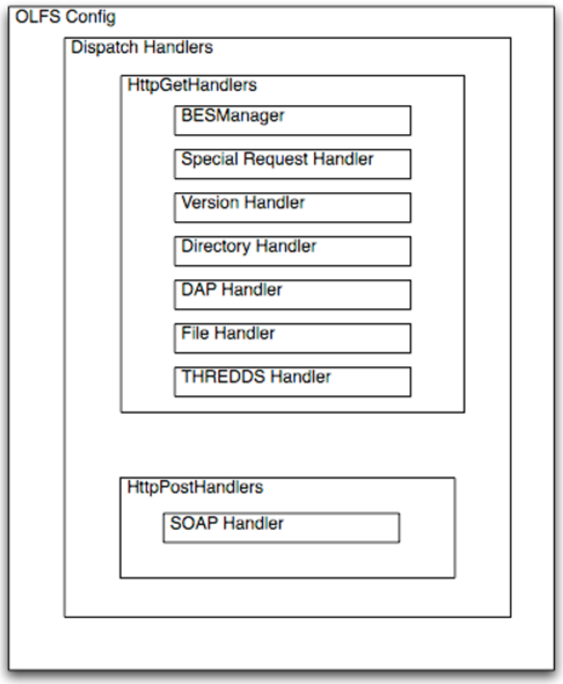


Fig. 2: OLFS Handler

3.1.5 OLFS

The OLFS works as a receiver for requests from clients, authenticating requests and ensuring DAP2 compliance. OLFS is implemented in the Java programming language, which contrasts with the BES which is primarily implemented in C++. The BES is the prime target for this project, however understanding and setting up the OLFS is a must to effectively implement and test the CovJSON response handler.
Figure 2 shows the dispatch handler for the Hyrax server. The dispatch is an ordered list that specifies what BES calls must be made to handle an incoming request. This list is broken into GET and POST HTTPS requests and configuration for response handlers can be applied for both.Figure 4 represents a basic implementation of the Hyrax OLFS. Going through the code quickly it can be seen that BESMANAGER sets the configurations for BES. While the dispatch handler goes through both the GET and POST requests as mentioned before, the OLFS will send BES commands to the BES which will then run conversion commands and return the data to the OLFS again which will then return it to the client. The project's engineers will have to know how to initially set up the Hyrax front-end and send the correct BES commands to run CovJSON request handler. [4]

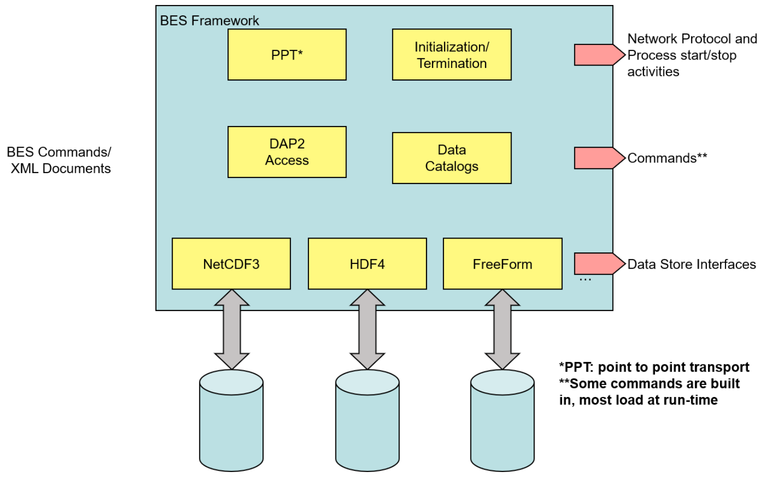

BES Framework
PPT*
Initialization/ Termination
DAP2 Access
Data Catalogs
NetCDF3
HDF4
FreeForm
...
BES Commands/ XML Documents
Network Protocol and Process start/stop activities
Commands**
Data Store Interfaces
*PPT: point to point transport
**Some commands are built in, most load at run-time

Fig. 3: BES Framework

3.1.6 BES

Figure 3 shows the general representation for the BES server within the standard Hyrax server. [5] From this figure there is a hierarchy to the process of the BES. The first is the Network Protocols with the examples of PPT and Initialization/Termination, these are not the only options but are not something developers will likely need to deal with. The second is the BES commands which define how data is to be manipulated and handled. The third and final level is the interface which connects the BES to the data servers that house the data. For the sake of NASA JPL this will be their data servers, see a later section for information on their data types. The focus of the project will be on the BES command and the pull from the source, with more focus on commands. The commands will take the data returned from the server and translate it into the CovJSON standard. Figure 5 is an example of this which can already be found in the BES file, "FoDapJsonTransform.cc," which transforms a DAP object into JSON. [6] To go into too much detail this function takes in the return information from a call and turns it into a JSON Object. The code being developed from CovJSON will be similar to this example, although its base system will be from pycovjson rather than just this file.

3.1.7 Conclusion

The functionality of the Hyrax server is instrumental for this project as most, if not all, components will have to eventually be implemented within it. While most coding of the project is dedicated to the BES, it is mandatory for developers to be able to understand and work with all pieces of Hyrax in order to make a complete and functional project.

```
<?xml version="1.0" encoding="UTF-8"?>
<OLFSConfig>

    <BESManager>
        <BES>
            <prefix>/</prefix>
            <host>localhost</host>
            <port>10022</port>

            <timeOut>300</timeOut>

            <adminPort>11002</adminPort>

            <maxResponseSize>0</maxResponseSize>
            <ClientPool maximum="200" maxCmds="2000" />
        </BES>
    </BESManager>
    <DispatchHandlers>
        <HttpGetHandlers>
            <Handler className="opendap.bes.VersionDispatchHandler" />

            <Handler className="opendap.coreServlet.BotBlocker">
                <<IpMatch>65\.55\.[012]?\d?\d\.[012]?\d?\d</IpMatch>
            </Handler>

            <Handler className="opendap.ncml.NcmlDatasetDispatcher" />

            <Handler className="opendap.threddsHandler.StaticCatalogDispatch">
                <prefix>thredds</prefix>
                <useMemoryCache>true</useMemoryCache>
            </Handler>

            <Handler className="opendap.gateway.DispatchHandler">
                <prefix>gateway</prefix>
            </Handler>

            <Handler className="opendap.bes.BesDapDispatcher" >
                <UseDAP2ResourceUrlResponse />
            </Handler>

            <Handler className="opendap.bes.DirectoryDispatchHandler">
            </Handler>

            <Handler className="opendap.bes.BESThreddsDispatchHandler"/>
            <Handler className="opendap.bes.FileDispatchHandler" />
        </HttpGetHandlers>

    </DispatchHandlers>

</OLFSConfig>
```

Fig. 4: Initial setup for generic OLFS

2

```
/**
 * String version of json_simple_type_array(). This version exists because of the differing
 * type signatures of the libdap::Vector::value() methods for numeric and c++ string types.
 *
 * @param strm Write to this stream
 * @param a Source Array - write out data or metadata from or about this Array
 * @param indent Indent the output so humans can make sense of it
 * @param sendData True: send data; false: send metadata
 */
void FoDapJsonTransform::json_string_array(std::ostream *strm, libdap::Array *a, string indent, bool
    sendData)
{
    *strm << indent << "{" << endl;\
    string childindent = indent + _indent_increment;

    writeLeafMetadata(strm, a, childindent);

    int numDim = a->dimensions(true);
    vector<unsigned int> shape(numDim);
    long length = fojson::computeConstrainedShape(a, &shape);

    *strm << childindent << "\"shape\": [";

    for (std::vector<unsigned int>::size_type i = 0; i < shape.size(); i++) {
        if (i > 0) *strm << ",";
        *strm << shape[i];
    }
    *strm << "]";

    if (sendData) {
        *strm << "," << endl;

        // Data
        *strm << childindent << "\"data\": ";
        unsigned int indx;

        // The string type utilizes a specialized version of libdap:Array.value()
        vector<std::string> sourceValues;
        a->value(sourceValues);
        indx = json_simple_type_array_worker(strm, (std::string *) (&sourceValues[0]), 0, &shape, 0);

        if (length != indx)
            BESDEBUG(FoDapJsonTransform_debug_key,
                "json_string_array() - indx NOT equal to content length! indx:  " << indx << "
                    length: " << length << endl);

    }

    *strm << endl << indent << "}";
}
```

Fig. 5: Function for translating scientific data formats into JSON

3.2 Component: Data Access Protocol

3.2.1 Purpose

The Data Access Protocol (DAP) is a protocol for accessing highly structured metadata and data organized as name-datatype-value tuples. OPeNDAP is well suited for allowing a client computer access to scientific data stored on a server computer that is networked with the client computer. The OPeNDAP protocol is one of the primary components of the project, as the team will be implementing a CovJSON data response handler module for the protocol.

3.2.2 Viewpoint

OPeNDAP has a number of different data response handlers previously built into it, and the design of the project's CovJSON response handler will follow the same design as these previously built handlers. Therefore, the dependency design viewpoint makes the most sense as it will be a part of an integrated system. [1]

3.2.3 Concerns

The major concern regarding the data access protocol is that the CovJSON response handler will need to be adequately test and debugged before it can be fully integrated into the OPeNDAP project. As always, time is a concerning factor, and it will be imperative for the developers to utilize project resources effectively to remain on schedule.

3.2.4 Functionality

The OPeNDAP CovJSON response handler will define a C++ based API that Hyrax's BES service will use to retrieve coverage data from a database. The response handler functionally defines the data types, data structures, objects, and function prototypes necessary for BES to convert and transmit requested data back to the client from the Hyrax server. Fundamental protocol functionality includes transmission of data, conversion of data, and receiving of data. Figure 6 shows the data request/retrieval flow for OPeNDAP; OPeNDAP provides the BES an interface that it can use to request and move scientifically formatted data from a database. All of this functionality already exists within the currently implemented scientific data handlers in OPeNDAP, therefore, the design of the DAP/BES interface and functions will rely heavily upon previously designed implementations to avoid having to reinvent the wheel, so to speak.

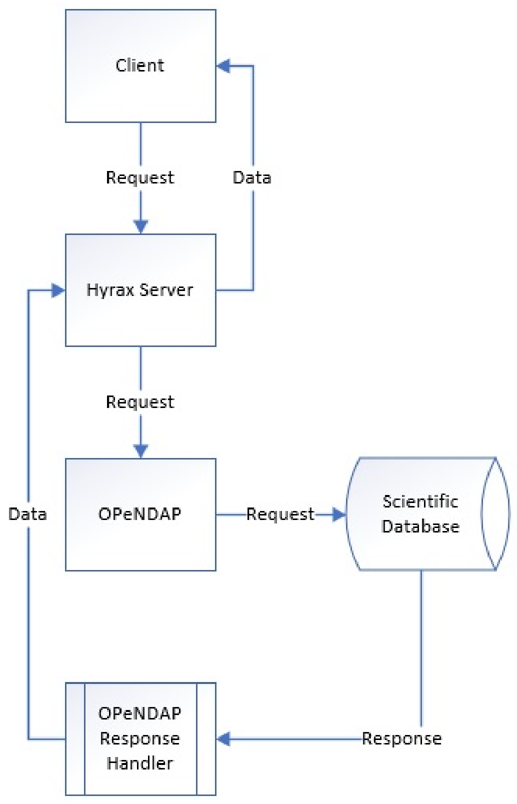


Fig. 6: OPeNDAP Flow Chart

3.3 Component: Data Format

3.3.1 Purpose

The purpose of adding CoverageJSON to the available data formats in OPeNDAP is to accommodate web development needs based on the data held in OPeNDAP. CoverageJSON allows for richer application development due to it being an extension of JSON language, which is tailored for conveying data distinguished by its space-time coordinates. JSON is one of the more common standards for retrieving data from a source and makes CoverageJSON a popular choice for those unfamiliar with scientific data formats.

3.3.2 Viewpoint

The view point for this component is information due to it being directly related and composed of the information handling and formatting portions of the project. [1]

3.3.3 Concerns

The primary concern regarding the data format is the possibility of an incorrect CoverageJSON encoding, since the requirement calls for multiple different data types to be changed into CoverageJSON this may present a problem in conversion methods. This will also require more knowledge of BES commands and managing the data correctly.

3.3.4 Functionality

A sample of the CoverageJSON format is shown in Figure 7. Since CovJSON is based on JSON, the data is broken down into categories, i.e. type, and domain. Those categories can have subcategories associated with them as a means of relying more data at once. JSON is notable in its ability to be human read, allowing developers to more easily manage the data that they receive.

```
{
  "type" : "Coverage",
  "domain" : {
    "type": "Domain",
    "domainType": "Grid",
    "axes": {
      "x": { "start": -179.5, "stop": 179.5, "num": 360 },
      "y": { "start": -89.5, "stop": 89.5, "num": 180 },
      "t": { "values": ["2013-01-13T00:00:00Z"] }
    },
    "referencing": [{
      "coordinates": ["x","y"],
      "system": {
        "type": "GeographicCRS",
        "id": "http://www.opengis.net/def/crs/OGC/1.3/CRS84"
      }
    }, {
      "coordinates": ["t"],
      "system": {
        "type": "TemporalRS",
        "calendar": "Gregorian"
      }
    }]
  },
  // Some lines omitted due to length
}
```

Fig. 7: CoverageJSON Example

The implementation of CoverageJSON into OPeNDAP will be heavily based off of the current JSON handling in OPeNDAP. The actual logic behind the conversion from other scientific formats into CoverageJSON will be based off of the PyCovJSON python program developed by NASA JPL, which is able to view and perform conversions into CoverageJSON from common scientific data formats. The scientific data formats primary function is to structure and organize the metadata and data for a given format. Design of the response handler will conform to CovJSON language definitions and structure.

3.4 Component: Testing

3.4.1 Purpose

The purpose of having predefined testing designs laid out is to create our implementation with testability in mind during the development process. This project will also have tests created during the implementation itself to minimize the overall level of effort exerted in testing. Functionality is inherently easier to test when it is fresh on the developer's mind, so testing close to the time of implementation will be the goal.

3.4.2 Viewpoint

The viewpoint of testing will be interaction. Interaction defines, "strategies for interaction among entities, regarding why, where, how, and at what level actions occur." [1] The testing that will occur is to ensure correctness among the CovJSON handler and the hyrax server elements.

3.4.3 Concerns

The primary concern for unit testing is ensuring that the quality of tests created is high and that they accurately test the functionality of the implementation. Inadequate testing could lead to problems not being located and issues arising in the future development of the project; the goal will be to catch all errors as soon as possible so that they cannot inhibit further development.

3.4.4 Functionality

The testing that will occur on the implementation will be done through unit tests in the CppUnit testing framework for C++ code. The unit testing will be done with the goal of having 90 percent code coverage when the implementation and testing are complete. The code coverage percentage goal may change due to feedback from the OPeNDAP team, as one of the primary goals in ensuring testing on the implementation is to allow it to be pulled into the OPeNDAP source code. The purpose of using the Cppunit framework is to align the work we do with the work that the OPeNDAP team does. Having consistent frameworks makes it easier for a hand off to occur between the developers and OPeNDAP.

Figure 8 is an example code snippet of a CppUnit unit test. A short rundown of the code and explanation of the usefulness of testing frameworks will now be given. In this given example two values of money are created, and another is added to one of those constants. These values are now compared with CPPUNIT_ASSERT which will either be true or false. If an assertion fails, or is not true, then the program will throw an error. The benefit of having a testing framework allows for quick and easy writing of unit tests to those who know the framework.

This project will likely require manual and randomly created tests to ensure the safety of the code, this is simply because this is to become production code. The usefulness of the random tests is yet to be seen in this project, but will allow for testing beyond the scope of what can be thought of by the developers. However random tests may not be a requirement of OPeNDAP, and this section will be changed to reflect that later.

3.5 Component: User Interface

3.5.1 Purpose

The purpose of the user interface portion of this project is to allow users to be able to select data format types they need from the NASA JPL servers. This will include being able to have their data in the CoverageJSON format, which will be the addition made to the already implemented Hyrax user interface.

```
void
MoneyTest::testAdd()
{
  // Set up
  const Money money12FF( 12, "FF" );
  const Money expectedMoney( 135, "FF" );

  // Process
  Money money( 123, "FF" );
  money += money12FF;

  // Check
  CPPUNIT_ASSERT( expectedMoney == money );           // add works
  CPPUNIT_ASSERT( &money == &(money += money12FF) );  // add returns ref. on 'this'.
}
```

Fig. 8: CppUnit Example

3.5.2 Concerns

The primary concern for the user interface implementation is ensuring usability while adding the functionality to make a request for CoverageJSON format. Adding in the new functionality may cause conflicts from an interface standpoint, as it may interfere with the current interface elements.

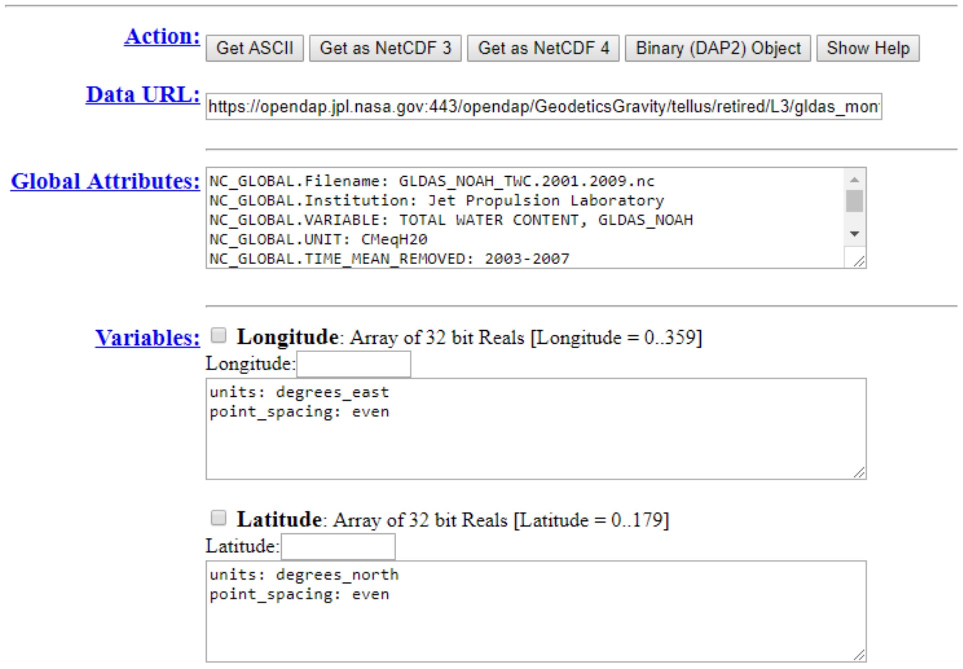
3.5.3 Viewpoint

The view point for this component is logical since it is a system that is already in place and our project just entails making modifications to the current interface implementation. [1]

3.5.4 Functionality

The user interface development portion of this project will involve making modifications to the current implemented interface. The current plan for the modification will be the addition of another button on the OPeNDAP server dataset access form which is shown in Figure 9. This modification may change through feedback from the OPeNDAP team, or the modification to the UI may not happen at all, as it will also be possible to request data in the CoverageJSON format through a corresponding data url, which is currently how JSON formatting is requested.

Figure 9 is a screen shot of the current data selection UI implemented in OPeNDAP. While it may seem bland to some, the purpose of the UI is to enable data scientists to get information that they need quickly.



Data URL: https://opendap.jpl.nasa.gov:443/opendap/GeodeticsGravity/tellus/retired/L3/gldas_mon

Global Attributes:

```
NC_GLOBAL.Filename: GLDAS_NOAH_TWC.2001.2009.nc
NC_GLOBAL.Institution: Jet Propulsion Laboratory
NC_GLOBAL.VARIABLE: TOTAL WATER CONTENT, GLDAS_NOAH
NC_GLOBAL.UNIT: CMeqH20
NC_GLOBAL.TIME_MEAN_REMOVED: 2003-2007
```

Variables: **Longitude**: Array of 32 bit Reals [Longitude = 0..359]

Longitude:

```
units: degrees_east
point_spacing: even
```

Latitude: Array of 32 bit Reals [Latitude = 0..179]

Latitude:

```
units: degrees_north
point_spacing: even
```

Fig. 9: OPeNDAP Server Dataset Access Form

3.5.5 Conclusion

This section may be a little light, but this is due to the fact that this project is not targeted at the creation of a user interface. However, adding to the current implementation of the user interface is requirement of this project, it just does not entail much change by the development team. As stated previously, if this were to become a requirement or became a higher priority by the team then this section will change to accommodate those new requirements.

4 Glossary

BES	back end, in C++, for the Hyrax data server.
CovJSON	An extension of the JSON (JavaScript Object Notation) language made to transmit geotemporal data. JSON is one a few standard means by which data is transmitted through networks. It is notable for being human readable, therefore easier to parse and manage.
CppUnit	A unit testing framework that is available for c++..
GET	A request from a client to a server to receive data that the server has.
HTTPS	Hypertext Transfer Protocol Secure. A means of secure communication over a computer network.
Hyrax	A data server that receives requests via OPeNDAP. Hyrax is mainly used for the gathering and processing data then returning it to the client.
NASA JPL	National Aeronautics and Space Administration Jet Propulsion Laboratory.
OLFS	Front end, in Java, for the Hyrax data server.
OPeNDAP	Open-source Project for a Network Data Access Protocol. An HTTP protocol/architecture that transfers data from a source to a client.
POST	A request from a client to a server to modify/recieve data that the server has.

REFERENCES

[1] S. . S. E. S. Committee, *IEEE Standard for Information Technology - System Design*, 2009.

[2] "Hyrax data server," https://www.opendap.org/software/hyrax-data-server, OPeNDAP, [Online; accessed 30-November-2017].

[3] "Hyrax - bes ppt," http://docs.opendap.org/index.php/Hyrax_-_BES_PPT, OPeNDAP, [Online; accessed 30-November-2017].

[4] "Hyrax - olfs configuration," http://docs.opendap.org/index.php/Hyrax_-_OLFS_Configuration, OPeNDAP, [Online; accessed 30-November-2017].

[5] "Hyrax architecture, ppt," https://www.opendap.org/pdf/bom_saw/SAW_2r3_HyraxArchitecture.ppt, OPeNDAP, [Online; accessed 30-November-2017].

[6] "odapjsontransform," https://github.com/OPENDAP/bes/blob/master/modules/fileout_json/FoDapJsonTransform.cc, OPeNDAP, [Online; accessed 30-November-2017].

[7] "Cppunit.cpp," http://cppunit.sourceforge.net/doc/cvs/money_example.html, cppunit, [Online; accessed 30-November-2017].

[8] "opendapui.png," https://opendap.jpl.nasa.gov/opendap/GeodeticsGravity/tellus/retired/L3/gldas_monthly/netcdf/GLDAS_NOAH_TWC.2001.2009.nc.html, OPeNDAP, [Online; accessed 30-November-2017].

[9] "Fodapjsontransmitter.cc," https://github.com/OPENDAP/bes/blob/master/modules/fileout_json/FoDapJsonTransmitter.cc, OPeNDAP, [Online; accessed 30-November-2017].

[10] R. Love, *Linux Kernel Development*. Pearson Education International, 2010.