CONTENTS

CHAPTER

ONE

INTRODUCTION

PHEMI Agile is a powerful data storage and analytics tool. It has the ability to ingest huge volumes of data and extract *derived data* from different ingested data sources.

Customers can create Data Processing Functions (DPFs) to extract individual pieces of data and transform them for later analysis. The DPF feature of PHEMI Agile allows users to provide their own instructions for extracting individual data values from documents, utilizing customer domain knowledge. The extracted data is known as *derived data*. Furthermore, the customer may specify time increments over which to aggregate multiple entries from different points in time. Currently, the supported aggregations are count, minimum, maximum, sum, and sum of squares. This is useful for the case that incoming data is arriving periodically and aggregated quantities could help speed up a calculation of the average value over a period of time, for example.

The DPF is like an application plugin that enables automatic derived data extraction at *ingest* time.

1.1 Overview of the PHEMI Agile Ingestion process:

Ingestion of new data into PHEMI Agile is presented to the user as a single seamless process - but can be thought of as multiple steps:

1. Data - typically documents or archives - is selected and uploaded to the system.
2. Once the data has finished uploading, the system processes the data and stores it, each uploaded file becoming a single *logical row* inside the system. During this Ingest process, the system examines the uploaded files, storing both the *raw data* and some extra *metadata*.
3. Optionally, after ingestion, a DPF can be triggered which can then process the *raw data* stored in the logical row, to produce *derived data*.

In order to generate derived data, customers need to complete the following steps:

1. Create a *Data Source* and define the relevant properties of that Data Source.
2. Understand and define what derived data is required and how to extract that data from the source documents using a DPF.

3. Write the DPF, test it and register it with PHEMI Agile using the Data Source screen, specifying when the DPF is triggered.
4. Import documents relevant to the data source defined through PHEMI Agile's data ingestion tab on the Data Source screen.
5. Depending on the DPF trigger setting, a registered DPF can be launched either automatically or manually. After successful execution of the DPF, derived data are available in PHEMI Agile.

CHAPTER

TWO

DATA PROCESSING FUNCTIONS (DPFS)

A Data Processing Function (DPF) extracts individual pieces of data from *raw data* and transforms them for later analysis. The DPF feature of PHEMI Agile allows users to provide their own instructions for extracting individual and aggregated data values from documents, utilizing customer domain knowledge. The extracted data is collectively known as *derived data*, with each individual piece of data being a *Digital Asset*:

Figure 2.1: Schematic overview of data flow in a DPF

2.1 DPF Archive Structure

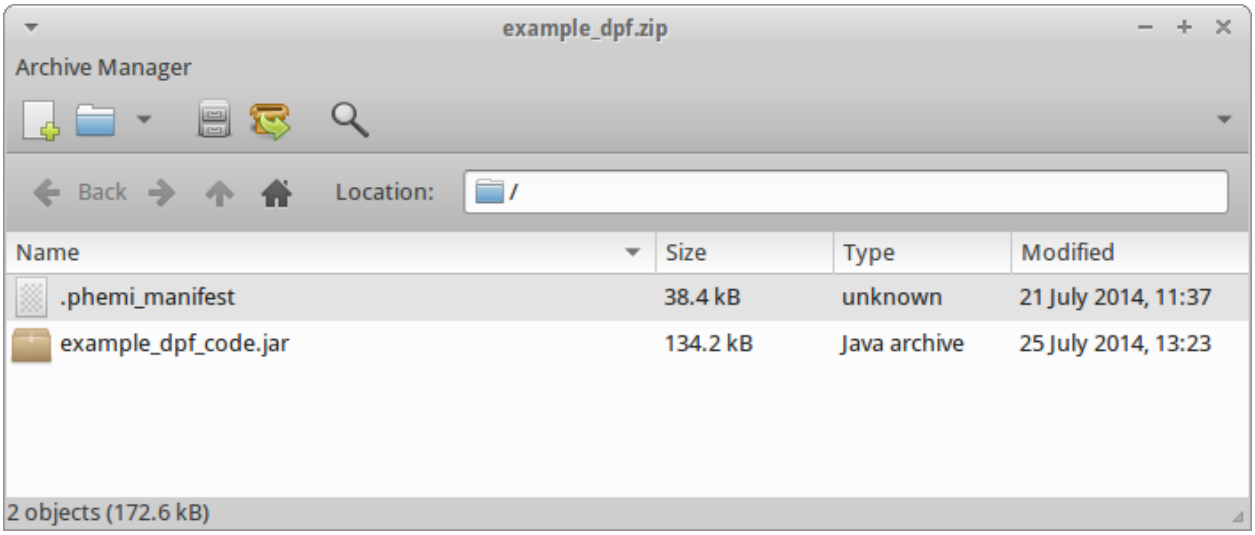
There are two parts to a complete DPF specification: the code that actually processes the data, and a list of the outputs that will be generated.

The code needs to implement the *DPF Programming Interface* and process the raw data, outputting the derived data. Once written, the code is bundled up, along with all its dependencies, into a *Code Library*. See *Creating the Code Library* for more information.

Each piece of derived data output is known as a *Digital Asset*, and needs to be specified explicitly to indicate the name and data type to PHEMI Agile. This is done by creating a *Manifest File*. See *Creating the Manifest File* for more information.

Once you've written the Code Library and the Manifest File, you bundle them both up into a *DPF Archive*, which you can then upload to PHEMI Agile to register your new DPF.

Here's an example of a DPF Archive, containing a Java .jar file containing the code and the manifest file to go with it:


example_dpf.zip
Archive Manager
Back
Location:
/
Name
Size
Type
Modified
.phemi_manifest
38.4 kB
unknown
21 July 2014, 11:37
example_dpf_code.jar
134.2 kB
Java archive
25 July 2014, 13:23
2 objects (172.6 kB)

CHAPTER

THREE

CREATING THE MANIFEST FILE

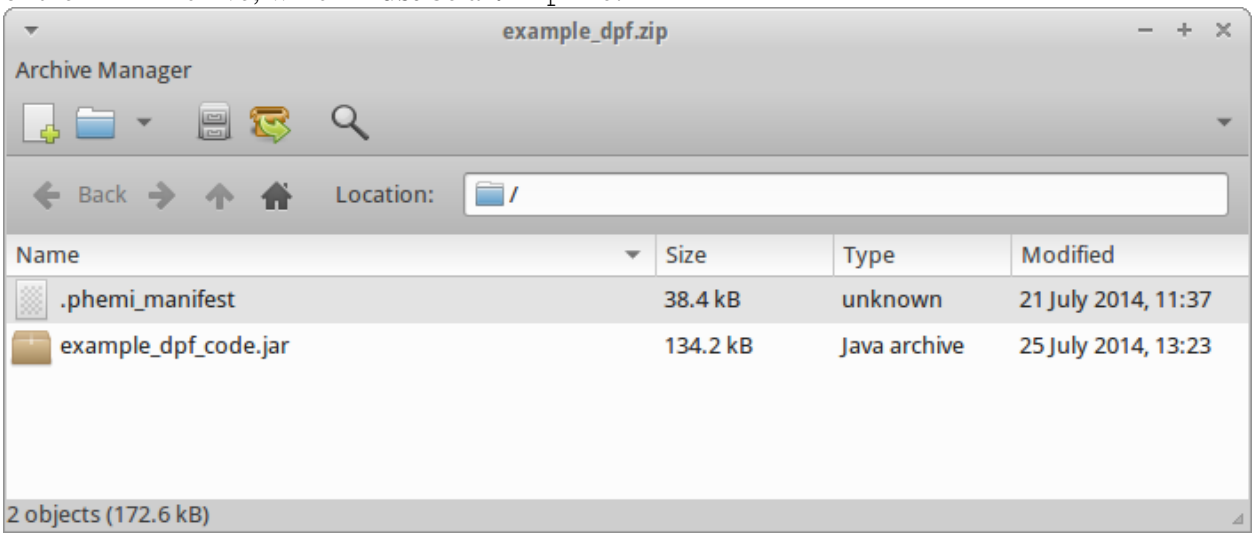
A DPF and its outputs are specified in a Manifest file. The Manifest file is formatted as a JSON document – see the following example:

```
{
    "name": "Echo '98",
    "status": "DRAFT",
    "scope": "INGEST",
    "command": "/home/hduser/venv/phemi-agile/bin/python -m phemi.udf.echo98_docx",
    "description": "This is a test code library for Echo 98 docx records.",

    "output_assets" : [
        {
            "name" : "echocardiograms|echo_date",
            "attributes" : [
                "NON_IDENTIFIED"
            ],
            "data_type" : "DATE",
            "definition" : "Date of ECHO",
            "constraint" : "Date in YYYY-MM-DD format"
        },
        {
            "name" : "echocardiograms|aorta_mm",
            "attributes" : [
                "NON_IDENTIFIED"
            ],
            "data_type" : "INT",
            "unit" : "mm",
            "definition" : "Aorta thickness",
            "constraint" : "Between 1mm and 10mm"
        }
    ]
}
```

PHEMI Agile will extract the information from the manifest file during the code library selection process. The manifest consists of two major parts: DPF & Code Library Metadata and Output

Assets.

The manifest file **must** be named `.phemi_manifest` and **must** be located in the root directory of the DPF Archive, which **must** be a `.zip` file:



3.1 DPF & Code Library Metadata

The first section of the manifest contains configuration for the DPF as a whole and for the code library. The following values are supported:

3.1.1 name

Name of the code library.

3.1.2 status

Controls writing DPF processed results to PHEMI Agile. Only DPFs that have the `AVAILABLE` status are run. The `DRAFT` status will allow the DPF to be run without saving the processed results - this is useful for testing.

Supported values	Meaning
DRAFT	Do not write processed results to PHEMI Agile. This is the default.
AVAILABLE	Write processed results to PHEMI Agile.

3.1.3 scope

The event that triggers the DPF. DPFs can run immediately after a data ingest (when data arrives) or be scheduled to run periodically at set intervals - or started manually.

Supported values	Meaning
INGEST	The DPF is run when data arrives at data ingestion. This is the default.
MANUALLY	The DPF is manually invoked.
PERIODIC	The DPF is run periodically at the the time intervals specified by 'periodicity'.

3.1.4 periodicity

The time interval at which the DPF is run.

Supported values	Meaning
ONE_MIN:	Every minute.
FIVE_MIN	Every 5 minutes. This is the default.
TEN_MIN	Every 10 minutes.
THIRTY_MIN	Every 30 minutes.
SIXTY_MIN	Every 60 minutes.

Todo

Why are these fixed times: why can't the user just enter an integer value for the number of minutes?

3.1.5 command

The command line used to run the DPF. E.g. `python -m /path/to/xmlDPF.py` or `java -cp /path/to/example.jar <Dpf class name>`.

3.1.6 java commands invoking multiple Jar files

When specifying a Java command on the DPF specification screen that requires multiple jar files, take care

```
java -cp build/libs/sample.jar:phemi/libs/client.jar
com.company.SampleDpf
```

the jar file string of `build/libs/sample.jar:phemi/libs/client.jar` must **not** be surrounded by double quotes even though that is required for running the same command on the command line.

3.1.7 description

Description of the DPF. Max. Length 2048 chars.

3.2 Output Assets

The `output_assets` section contains configuration for the digital assets output by the DPF:

3.2.1 name

Name of the digital asset. Our current recommendation is to structure names like this:

```
<data source category>|<digital asset name>
```

however, this isn't mandatory, and any string can be used (the string has to be converted to lowercase, and spaces should be replaced with underscores). Max. Length: 256 chars.

3.2.2 attributes

The *Privacy Level Attributes* for the digital asset.

Supported values	Meaning
NON_IDENTIFIED	The digital asset does not present any *Personally Identifiable Information*.
DE_IDENITIFIED	The digital asset presents de-identified information. Either the source data is already de-identified, or has been de-identified by the DPF.
IDENTIFIED	The digital asset presents *Personally Identifiable Information*

3.2.3 data_type

The Data Type of the Digital Asset.

Supported values	Meaning
LONG	A Long Integer.
DOUBLE	A floating point number.
STRING	String data.

Note: Other types may be added in the future, but only these three are currently supported. More complex data types can currently be saved as STRING for later use, or further processing outside PHEMI Agile.

See *Setting Data Types for Derived Data* for more on PHEMI Agile data types.

3.2.4 unit

The unit of measure for the Digital Asset. This is currently for information only and isn't enforced by PHEMI Agile. Max length 32 chars.

3.2.5 definition

A short description of what the contents of a digital asset represent. Max. length 256 chars.

3.2.6 constraint

A short description of the constraints that should apply to the allowed values of the Digital Asset. This is currently for information only and isn't enforced by PHEMI Agile. Max length 256 chars.

CHAPTER

FOUR

DPF PROGRAMMING INTERFACE

PHEMI Agile uses the Apache Thrift Framework to generate interface files that support communicating with multiple programming languages. PHEMI Agile's Data Processing Function Framework currently supports DPFs that are written in Python or Java.

Note: We currently only have documentation for the Java DPF Interface. Python documentation will follow.

PHEMI Agile's DPF support framework is implemented using the classic Visitor or "Double Dispatch" pattern. This pattern allows a DPF to be integrated with the support framework seamlessly during runtime but allows these processes to be loosely coupled, easing maintenance & dependencies. This implementation is intended to insulate a customer DPF from most framework changes.

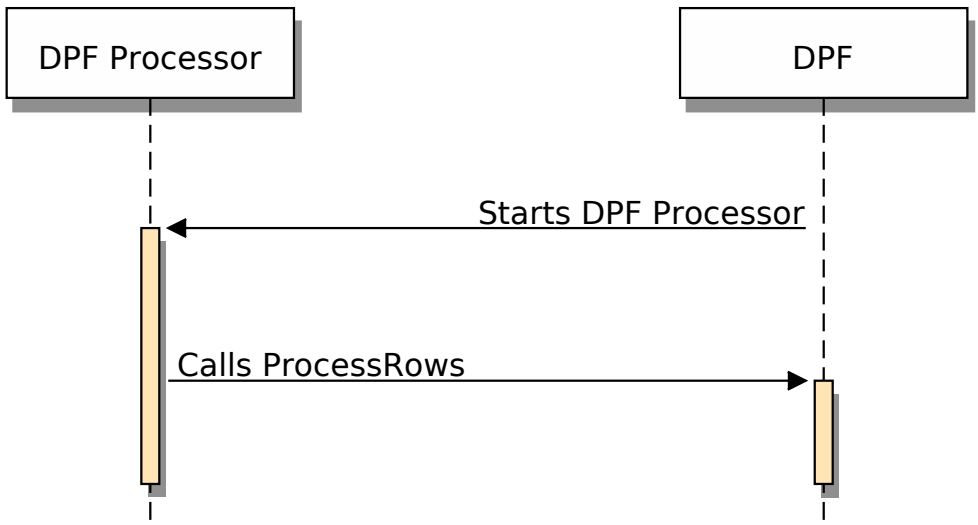


Figure 4.1: DPF is the visitor process that starts the processor.

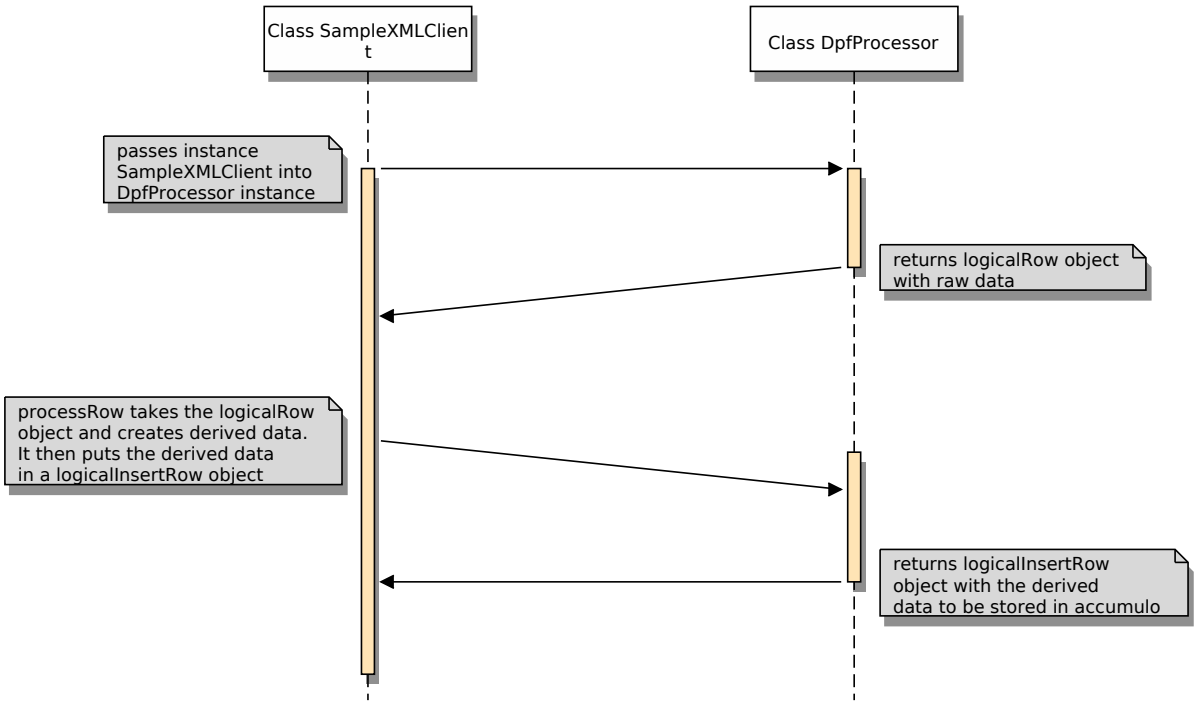
4.1 Java Support

The PHEMI Agile DPF Java API is very simple - only a single class with a single method needs to be implemented. This method should parse a document and extract derived data.

A customer written Java DPF class can register itself with a DpfProcessor class by instantiating it using an instance of itself, like this:

```
public class SampleXmlClient implements DpfClient {

    ...

    @Override
    public LogicalInsertRow processRow(LogicalRow row) {
            ...
    }

    ...

    public static void main(String argv[])
    {
        SampleXmlClient sample = new SampleXmlClient();
        DpfProcessor processor = new DpfProcessor(sample);
        processor.run();
    }
}
```

In the above code, a sample DPF class instantiates the `DpfProcessor` class, passing in an instance of itself. Once the processor starts to run, it will send *key value pairs* to the customer DPF's `ProcessRow` method.



A customer DPF must implement the `com.phemi.agile.udf.client.spi.DpfClient` interface. This interface requires a concrete implementation of the `ProcessRow` method that will take a `LogicalRow` object as input and is expected to return a `LogicalInsertRow` object. A customer DPF must return a `LogicalInsertRow` object even if processing fails and an exception was raised.

In case of failure, the customer DPF must return an empty `LogicalInsertRow` object, catch, log and discard any exceptions; exceptions should be logged to a flat file.

The customer DPF and its support framework communicate through `stdin` and `stdout`, so the customer DPF must not directly write to or read from these channels.

4.1.1 Relevant Classes: LogicalRow, LogicalInsertRow, InsertEntry, and InsertAggregatedEntry

Once a customer document is ingested, it is stored in a row and assigned a row id. This row id is also assigned to all its related derived data. To handle the writing of data into PHEMI Central, there are a number of classes to be aware of including LogicalRow, LogicalInsertRow, InsertEntry, InsertAggregatedEntry.

LogicalRow: A LogicalRow is composed of a row id, a data value (the ingested file), meta-data name/value pairs, and derived-data name/value pairs. So the LogicalRow basically encapsulates information about the document before it has been processed.

```
public final class LogicalRow {

private static final Charset UTF8_CHARSET = Charset.forName("UTF-8");

private String id;
private byte[] value;
private final LinkedHashSet<Entry> meta = new LinkedHashSet<>();
private final LinkedHashSet<Entry> derived = new LinkedHashSet<>();


/**
 * Constructor
 *
 * @param id - the id of the logical row
 */
public LogicalRow(String id) {
    this(id, null);
}
...
}
```

LogicalInsertRow: This class provides instructions for inserting the entire processed document into PHEMI Central. It is composed of the data entry name/value pairs, as well as the aggregation name/value pairs. The aggregations are optional but can be used to speed up calculations with time-series data.

```
public final class LogicalInsertRow {

private static final Charset UTF8_CHARSET = Charset.forName("UTF-8");

private LinkedHashSet<InsertEntry> entries = new LinkedHashSet<>();
private LinkedHashSet<InsertAggregatedEntry> aggregations = new LinkedHashSet<>();

/**
 * Constructor
 */
public LogicalInsertRow() {

}
```

The LogicalInsertRow is just a collection of InsertEntry objects, which provide the detailed-level instructions for inserting each piece of data into the system.

```
public final static class InsertEntry {

    ....

    /**
```

```
     * Creates a new InsertEntry instance.
     *
     * @param name  - the name of the entry being inserted.  Cannot be null.
     * @param value - the value of the entry.  Cannot be null.
     * @param level - the privacy level associated with the entry. Cannot be null.
     * @param type  - the data type associated with the entry value. If not
     *                defined, String is assumed.
     * @return a new InsertEntry instance.
     */
    public static InsertEntry newInstance(String name,
            String value, PrivacyLevel level, LexiCoderTag type) {
        ....

        return new InsertEntry(...);
    }

    .....
}
```

Table 4.1: InsertEntry Object Properties

Attributes	Meaning
name	The Name of the derived data; should match the name specified in the manifest file when the DPF was first created. The value can be extracted from the input document or substituted with a value that is computed based on the extracted value.
value	Value of derived data
privacy	PHEMI Agile currently supports a number of privacy levels or *Authorizations*. The privacy levels that PHEMI Agile supports are: IDENTIFIED, DE-IDENTIFIED, and NON-IDENTIFIED. Setting this field is optional but if it is set, it must match what was recorded in the manifest file so the derived data can be exported later by executing a *Dataset* defined on this data source.
type	String, Long, or Double, please see Data Value encoding section for more details.

Similar to the InsertEntry, there is the InsertAggregatedEntry class, which provides instructions for assigning an individual piece of data to be included in an aggregate calculation. The LogicalInsertRow may include a list of InsertAggregateEntry objects.

```
/**
    * An Aggregated table entry
    */
   public final static class InsertAggregatedEntry {

       private byte[] coarseGrainedBin;
       private byte[] fineGrainedBin;
       private byte[] value;
```

```
    /**
     * Constructor
     *
     * @param coarseGrainedBin - required field
     * @param fineGrainedBin   - optional field
     * @param value            - required field.
     */
    public InsertAggregatedEntry(byte[] coarseGrainedBin, byte[] fineGrainedBin,
       ...
    }
```

When a document is submitted for ingestion, the document and its row id are passed to a DPF inside a LogicalRow object.

The document is passed in as a Byte array. A customer DPF should convert and process it properly to generate derived data. Once generated, a customer DPF should put the derived data into an instance of the InsertEntry object. Then, the InsertEntry object must be placed into a LogicalInsertRow object, which is then passed back to the PHEMI Agile support framework as the return value of the ProcessRow function.

The following code excerpt illustrates these points:

- line 13: a LogicalRow is received by the ProcessRow method.
- line 14-19: the xml document is extracted (UTF-8 BOM is handled) and ready for processing.
- line 16: use System.err.print instead of System.out.print, to prevent java heap space error.
- line 23: the populateRow function is called to process and populate a LogicalInsertRow object.
- line 24 - 32: even if exceptions are raised they are written out and thrown away.
- line 36: a LogicalInsertRow is returned (note that this is empty in case of exception).

```
1  private static final Charset UTF8_CHARSET = Charset.forName("UTF-8");
2  ...
3  /******
4   * The purpose of processRow is to process the input document,
5   * extract data that we want to store and return it in
6   * a LogicalInsertRow structure.
7   *
8   * @param row - a LogicalRow object that contains the row id and
9   * raw document to be processed
10  * @return LogicalInsertRow
11  */
12 @Override
13 public LogicalInsertRow processRow(LogicalRow row) {
14     byte[] xmlByte = row.getValue();
```

```
15      if ((xmlByte[0]==-17)&&(xmlByte[1]==-69)&&(xmlByte[2]==-65)) {
16              System.err.print("Found matching UTF-8 BOM");
17              xmlByte = Arrays.copyOfRange(xmlByte, 3, xmlByte.length);
18      }
19      String xmlString = new String(xmlByte);
20
21      LogicalInsertRow insertRow = new LogicalInsertRow();
22      try{
23          insertRow = populateRow(xmlString);
24      } catch (JDOMException e) {
25          System.err.append("Cannot parse input xml document, actual message: "
26                  + e.getMessage());
27          e.printStackTrace();
28      } catch (IOException e) {
29          System.err.append("Encounter IO error while parsing input xml document, ac
30                  + e.getMessage());
31          e.printStackTrace();
32      }
33      System.err.println("******************/n/n/n"
34          + insertRow.toString()
35          + "/n/n**********************");
36      return  insertRow;
37  }
38
39  /***
40   * We parse the input xml document to extract each field and
41   * process them one by one.
42   *
43   * @param xml - the String representation of the input xml document
44   * @return LogicalInsertRow
45   * @throws JDOMException
46   * @throws IOException
47   */
48  private LogicalInsertRow populateRow(String xml)
49          throws JDOMException, IOException
50  {
51      LogicalInsertRow insertRow = new LogicalInsertRow();
52      SAXBuilder builder = new SAXBuilder();
53      Document doc = builder.build(new StringReader(xml));
54      Element root = doc.getRootElement();
55      List<Element> fields = root.getChild("row").getChildren("field");
56      insertRow.addEntries(getEntries(fields));
57
58      return insertRow;
59  }
```

See *sample_xml_client* for a full listing of this sample DPF program and *sample_xml_file* for a

sample XML document that it processes.

4.2 Setting Data Types for Derived Data

PHEMI Agile stores all raw data as bytes. To sort and query derived data, the data type of the derived data must be specified when it is created by the DPF.

All data not explicitly assigned a data type will default to a STRING data type. Numeric values sorted as strings will produce erroneous sort and query results. For example, the String "100" will be sorted before the String "2". This dictionary ordering is referred to as lexicographic ordering. To ensure the correct lexicographic ordering and get proper query results on numeric values, the correct LexiCoderTag's enum value must be specified. The following code fragment illustrates this point:

```
1  private LexiCoderTag getEncodedType(String inType){
2      LexiCoderTag dataType = LexiCoderTag.STRING;
3      if (inType.equalsIgnoreCase("int") || inType.equalsIgnoreCase("long") || inTyp
4          dataType = LexiCoderTag.LONG;
5      }
6      else if (inType.equalsIgnoreCase("double") || inType.equalsIgnoreCase("float")
7          dataType = LexiCoderTag.DOUBLE;
8      }
9      return dataType;
10 }
```

- line 1: an input data type is passed in.
- lines 3 - 7: the input data type is mapped to a PHEMI Agile supported data type.
- line 9: the appropriate PHEMI Agile data type is passed back.

Currently, PHEMI Agile supports the following data types:

PHEMI Agile Type	Meaning	Matching Java Types	Matching MySQL Types
LONG	A Long Integer.	int, long, bigint	INTEGER, INT, SMALLINT, TINYINT, MEDIUMINT, BIGINT, TIMESTAMP
DOUBLE	A floating point number.	double, float	FLOAT, DOUBLE
STRING	String data.	string	CHAR, VARCHAR, BINARY, VARBINARY, BLOB, TEXT

These correspond to various different data types supported by different languages and data stores (e.g. Oracle, SQL server, MySql, MongoDB) and should be mapped as shown in the code above.

4.3 Integrating with the PHEMI Agile support framework

When authoring a customer DPF, a user will need to use and interact with PHEMI Agile's various Java classes. These classes are contained in the PHEMI Agile's udf (user defined function) client jar file. A PHEMI Agile DPF developer needs to make this jar file accessible to the DPF by putting it on the DPF's class path.

CHAPTER

FIVE

CREATING THE CODE LIBRARY

5.1 Java Code Libraries

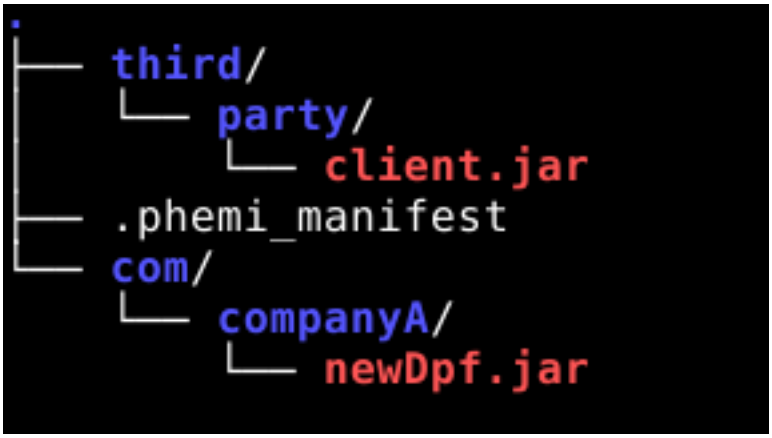
A code library is a zip file that contains Data Processing Function (DPF) classes and a mainfest file that describes the details of DPF generated data items and contains other admin data related to the DPF.

To prepare a code library to process data for PHEMI Agile, a developer has to write a new Java DPF class(es) as described in *DPF Programming Interface*. Once this new class is written and tested with the DPF harness (see *Standalone Debugging of a customer DPF*), it needs to be packaged with the .phemi_manifest (see *Creating the Manifest File*) file into a zip file.

The new zip file should contain the new Data Processing Function related Java class(es) and all its dependent classes (including required 3rd party packages). A developer should package an in house developed jar file along with all dependent ones into the new zip file. He is free to use a number of popular tools to package the jar and zip files (for example ant, maven, gradle, zip, 7zip, win zip, etc.). The zip file must contain all dependent files so PHEMI Agile can run the customer supplied java command standalone. Consider the following example:

```
java -cp com/companyA/newDpf.jar:third/party/client.jar com.companyA.NewDpf
```

where the new DPF java class is `com.companyA.NewDpf`, the new jar file that contains this class is `com/companyA/newDpf.jar` and the `third/party/client.jar` contains all the packages or classes the new DPF class depends on. The following diagram shows the example zip file's internal structure:



5.2 Python Code Libraries

Todo

Write this section.

Note: We currently only have documentation for the Java DPF Interface. Python documentation will follow.

CHAPTER

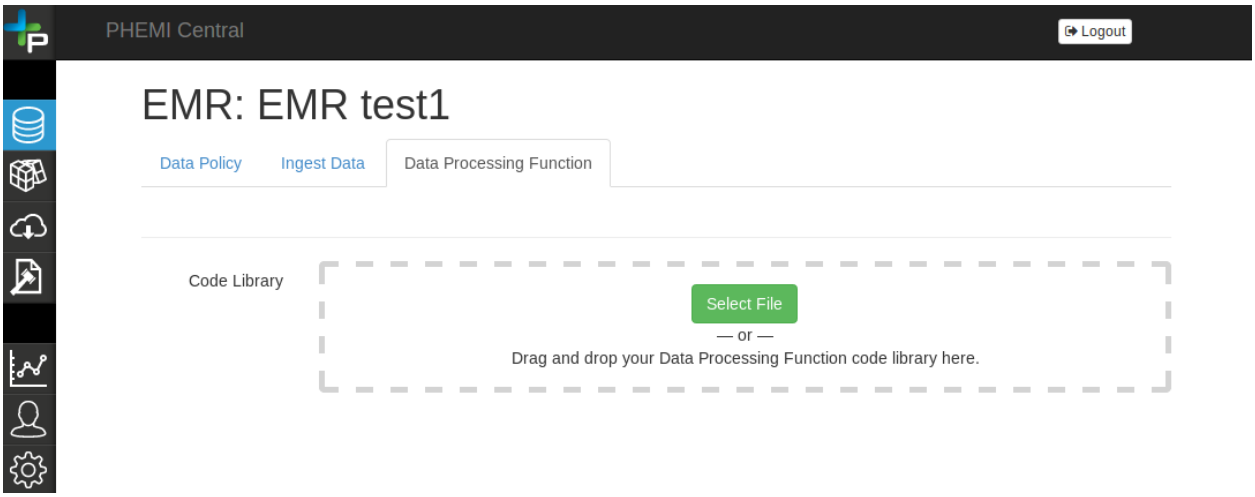
SIX

REGISTERING A NEW DPF WITH PHEMI AGILE

Users can register a DPF on PHEMI Agile's Data Sources screen. Once a data source is defined (see: *Defining a new Data Source*), select the Data Source, then click on the "Data Processing Function" tab.

6.1 Upload a DPF Archive

Once a new Data Source is defined, the "Data Processing Function" tab will ask you to upload a *DPF Archive*. You can drag and drop the target file to the dotted rectangle, or click 'Select File' to browse for the file.



6.2 Save your DPF

Once a valid DPF Archive is uploaded, the DPF is defined. The *Code Library* and the digital assets (e.g. name, attributes, data type) are displayed. Make any required changes, then click Save Changes.

Note: The DPF isn't automatically saved when a manifest or code library is uploaded - you must review the configuration and click Save Changes.

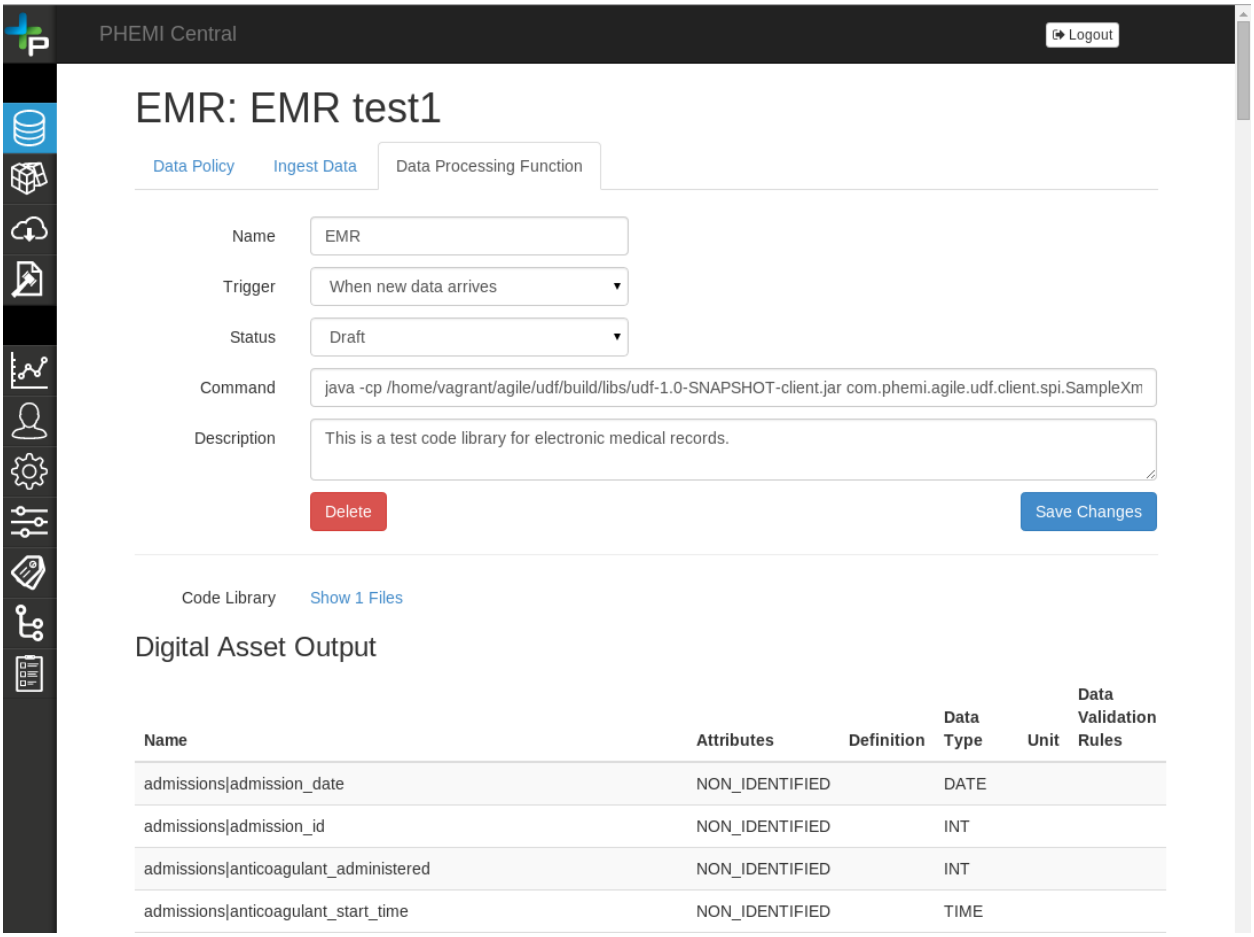


Figure 6.1: These fields are pre-populated from a valid manifest file.

For more details on the configuration options for a DPF, see: *Creating the Manifest File*.

CHAPTER

SEVEN

STANDALONE DEBUGGING OF A CUSTOMER DPF

7.1 DpfHarness Tool

The `DpfHarness` Java tool is used to help debug and test a DpfClient in a standalone environment outside the PHEMI Agile application. The tool is started in the developer's IDE and regular Java debugging is used to step through the DpfClient code.

The `DpfHarness` tool mimics the PHEMI Agile application communication protocol and, as such, verifies whether LogicalInsertRows generated by the DpfClient are correctly serialized/deserialized.

The `DpfHarness` is used with byte arrays or Java File objects to send data to the DpfClient being tested.

7.1.1 Testing/Debugging with a Byte Array

```
...
import com.phemi.agile.udf.client.tools.DpfHarness
...

public class ExampleDpfClientTester {

    ....

    public static void main(String args[]) {

        byte[] somebytes = ....;
        DpfHarness harness = new DpfHarness(new ExampleDpfClient());
        LogicalInsertRow row = harness.processData(somebytes);
        System.out.println(row);
    }
}
```

7.1.2 Testing/Debugging with Multiple Byte Arrays

```
public class ExampleDpfClientTester {

    ....

    public static void main(String args[]) {

        List<byte[]> bytevals = ......;
        DpfHarness harness = new DpfHarness(new ExampleDpfClient());
        List<LogicalInsertRow> rows = harness.processDataInputs(bytevals);
        for(LogicalInsertRow row: rows) {
            System.out.println(row);
        }
    }
}
```

7.1.3 Testing/Debugging with a File Object

```
public class ExampleDpfClientTester {

    ....

    public static void main(String args[]) {

        File file = new File("/home/dpfclient/data/testdata.txt");
        DpfHarness harness = new DpfHarness(new ExampleDpfClient());
        LogicalInsertRow row = harness.processFile(file);
        System.out.println(row);
    }
}
```

7.1.4 Testing/Debugging with Multiple File Objects

```
public class ExampleDpfClientTester {

    ....

    public static void main(String args[]) {

        try {
            File directory = new File("/home/dpfclient/data");
            File[] files = directory.listFiles();
            DpfHarness harness = new DpfHarness(new ExampleDpfClient());
```

```
            List<LogicalInsertRow> rows = harness.processFiles(files);
            for (LogicalInsertRow row : rows) {
                System.out.println(row);
            }
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
}
```