ODM Tools Python

Architecture

ODM Tools is a desktop front-end for the ODM scientific data architecture. The schema for ODM version 1.1.1 is found at: http://his.cuahsi.org/images/ODM1_1SchemaDiagram_md.jpg

ODM Tools has a three-level architecture: data access layer, services layer, and user interface layer.

Data Layer

The data layer is built on SQLAlchemy 7.9 (http://pypi.python.org/pypi/SQLAlchemy/0.7.9) which is an ORM library for Python. ODM Tools supports MS SQL, MySQL, and PostgreSQL. All data layer classes can be found in the odmdata package. (there is an issue using a mac to connect to a MS SQL db because of a driver)

Aside from the Base class, which comes from SQLAlchemy, all the classes in the odmdata package map one-to-one onto the ODM schema's tables (with one exception below) . Each class's attributes correspond with table columns (for more see http://en.wikipedia.org/wiki/Object-relational_mapping).

The central object in the ODM model is a Series, which represents a time-indexed set of data. A Series is uniquely identified by the combination of Site, Variable, Method, Source, and QualityControlLevel. A Series has many DataValues.

The MemoryDatabase object is one of two odmdata classes that isn't an ORM object onto a table. It's used to create an in-memory SQLite DB for editing time series. Edits are done on the memory database and then reflected to the live database when editing is complete. There's an open GitHub issue for rewriting the MemoryDatabase to use SQLAlchemy instead of hand-written queries. (see below)

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite:///:memory:', echo=True)
```

The other class that doesn't map to a table is the SessionFactory. It uses an SQLAlchemy connection string to create and return Session objects for the current database. See the SQLAlchemy tutorial linked below for examples of how to use a Session.

SQLAlchemy

ODM Tools uses SQLAlchemy version 7, docs here http://docs.sqlalchemy.org/en/rel_0_7/

Specifically, it uses the ORM component to define objects that map to (correspond with) the tables in the data schema. You'll get a good feel for how this works by going through

http://docs.sqlalchemy.org/en/rel_0_7/orm/tutorial.html After you finish the tutorial, look through the class files for any data layer object (especially Series and DataValue), and then look at the get methods for the SeriesService.

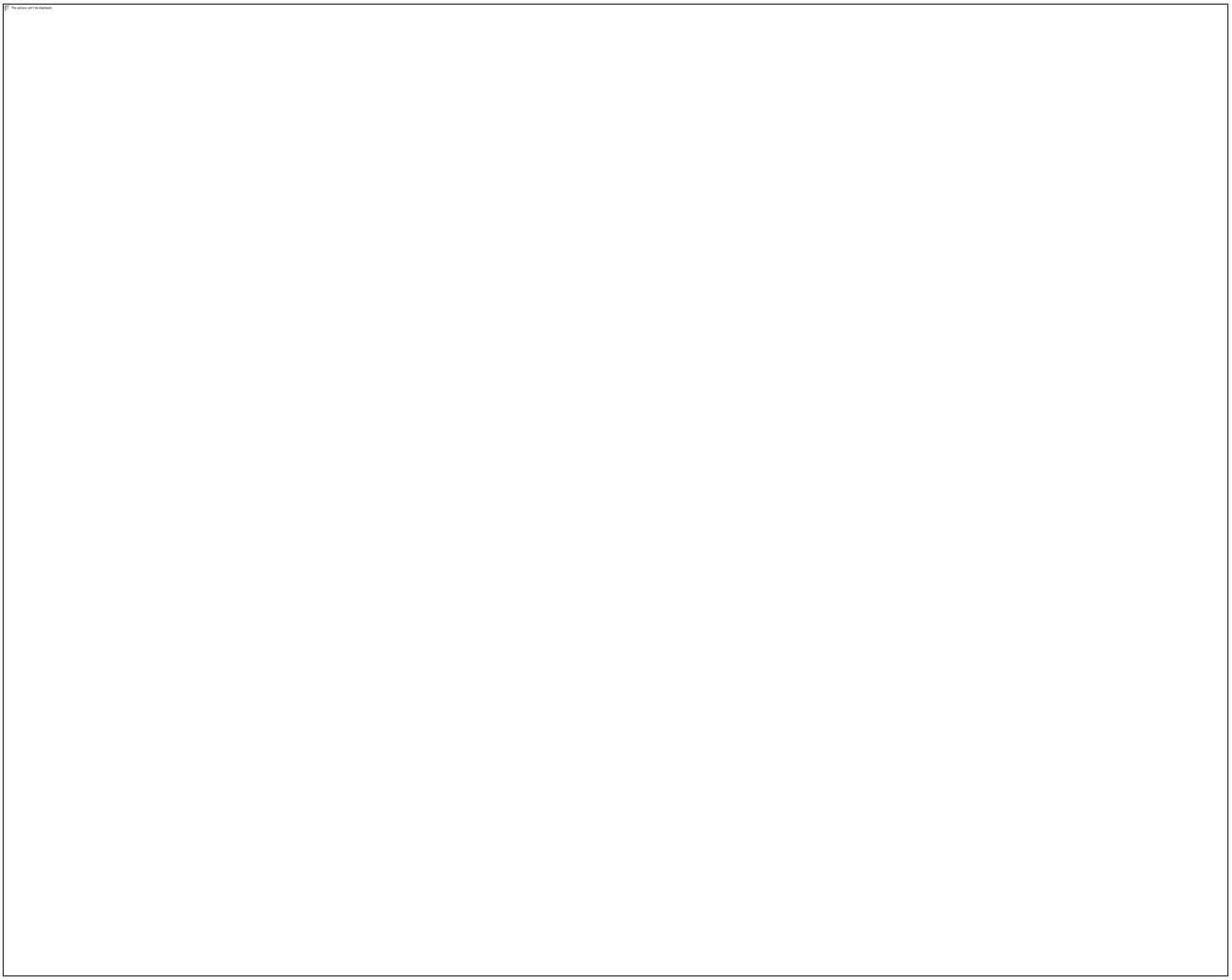
When an ORM object is returned from SQLAlchemy, it's bound to the session that retrieved it.

The bulk of ODM Tools behavioral functionality lies in the services layer.

Services Layer

The interface front-end interacts with the data layer through the services layer. A *service* is an object that gives and receives the data ORM objects, or standard Python types. Primarily this means getting data or arrays of data from the database, and saving data to the database. Service objects are divided along logical groups based on usage. The services are: SeriesService, CVService, EditService, ExportService, and RecordService. The odmservices package also contains a Utilities class for various boilerplate functions (path conversion so far).

All services are created through the ServiceManager class. This class manages the current “open” database connection and the connection.cfg file. When first created, the ServiceManager reads the connection.cfg and takes the **last** connection in the file as the startup default. The below screenshot illustrates using the ServiceManager to create and use a SeriesService. All other services work



similiarly.

The SeriesService contains methods for interacting with Series, DataValues, and Site, Variable, Source, Method, and QualityControlLevel information.

The CVService interacts with all CV tables in the ODM model.

When a series is being edited, a temporary copy is created for editing. The EditService contains all the methods needed for doing this.

The RecordService is largely a wrapper to the EditService that also outputs script lines to the ODM Tools script editor when recording in edit mode. The RecordService is only useful inside the full ODM Tools application because it relies on the Script Editor form to send lines to. When creating a new editing function in the EditService, you need to create a counterpart function in the RecordService as well as in the ConsoleTools class in the odmconsole package (described below).

The ExportService exports series data in .csv and .xml.

Console Tools

ODM Tools contains an embedded Python console using the pycrust component of wx (see "Script & Console section of frmODMToolsMain.py). The odmconsole package contains the ConsoleTools utility class to give access to internal ODM Tools functionality in the console. It provides the same behavior available through the gui buttons on the editing ribbon. As mentioned above, any new edit functionality created in the EditService needs to be reflected here as well as the RecordService. ConsoleTools calls RecordService which calls EditService.

Testing

ODM Tools has a suite of unit tests for the data layer and the services layer using pytest (http://pytest.org). Not every class in the data layer has a set of tests for it, only the most complex ones. All services have tests.

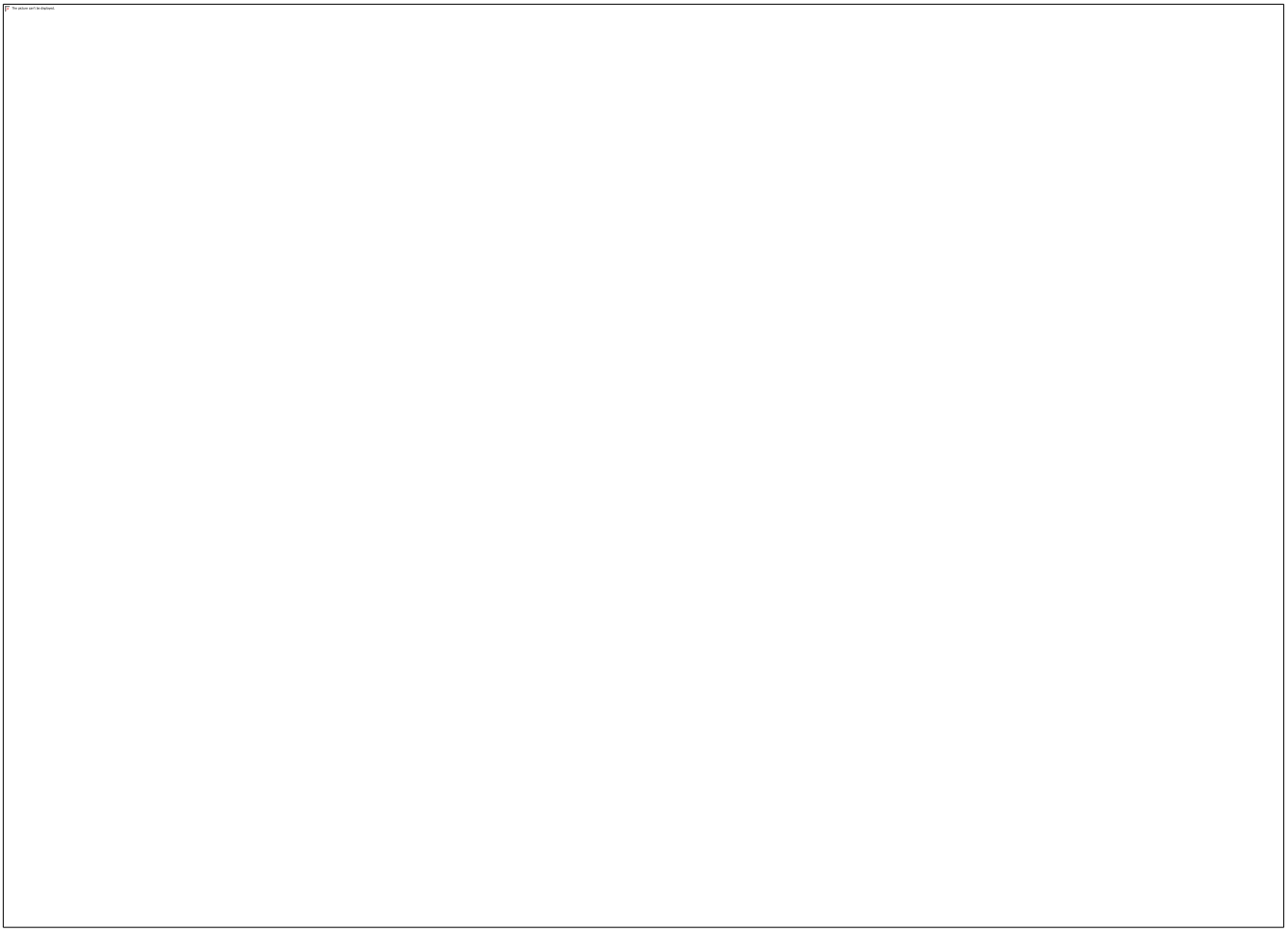
Each group of tests are contained in a test class. These tests are methods of the testing class, and are essentially true/false assertions. Each testing class has a special setup() method that is automatically run during the start of a testing run, and is used to perform any required initialization (such as creating a test database). A single test both documents/describes and verifies a single specific feature of the class or program feature it is for.

For example, the test_data_value.py file in the data_tests directory tests the DataValue ORM object. It's setup method creates a temporary in-memory SQLite database and populates it with sample data using the methods in test_util.py. The subsequent tests act on this prepared sample data.

To run the tests, navigate to the root level of the ODM project and run

```
>>> py.test
OR
>>> py.test path/to/specific_test.py
```

Below is a screenshot showing a successful test run, followed by an intentionally failing one.



For more about testing see

http://en.wikipedia.org/wiki/Unit_testing

http://en.wikipedia.org/wiki/Test-driven_development

Profiling

There are areas of ODM Tools that are in need of optimization. Notably, graphing/editing a large dataset is quite slow. In order to target the biggest wins and get a good return on time invested, profiling the application to measure performance will be needed.

Python provides some profiling tools built right into the language
http://docs.python.org/2/library/profile.html

There has yet to be much if any work done in this area. This guide looks like a great place to start
http://www.huyng.com/posts/python-performance-analysis/