ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA135

PyDM - EXTENSION POINTS

H. H. Slepicka*, M. Gibbs, SLAC National Accelerator Laboratory, Menlo Park, USA

Abstract

PyDM (**Py**thon **D**isplay **M**anager) is a Python and Qt-based framework for building user interfaces for control systems providing a no-code, drag-and-drop system to make simple screens, as well as a straightforward Python framework to build complex applications. PyDM developers and users can easily create complex applications using existing Python packages such as NumPy, SciPy, Scikit-learn and others. With high level interfaces for data plugins and external tools, PyDM can be extended with new widgets, integration with facility-specific tools (electronic log books, data logger viewers, et cetera) as well as new data sources (EPICS, Tango, ModBus, Web Services, etc) without the need to recompile or change the PyDM internal source.

PyDM

In 2015, studies were performed to evaluate software to be used as the next-generation display manager[1] at SLAC.

Based on the output of the study and evaluation of potential candidates, it was concluded that a new framework was required to fulfil the demands from scientists, operators and engineers for user interfaces and application development.

PyDM is an open-source Python-based framework for control system graphical user interfaces (GUIs) intended to span the range from simple displays without any dynamic behavior, to complex high level applications, with the same set of widgets.

It provides a system for the drag-and-drop creation of user interfaces using Qt Designer[2] and it also allows for the creation of displays driven by Python code.

Since PyDM is based on Python, it can be leverage the scientific Python ecosystem (see Fig. 1).

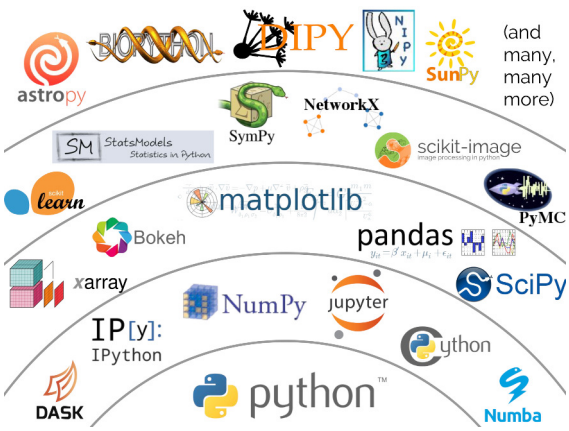

Figure 1: Scientific Python Ecosystem. (Credit: Jake VanderPlas, "The Unexpected Effectiveness of Python in Science", PyCon 2017)

* slepicka@slac.stanford.edu

BEYOND DRAG AND DROP

As many other display managers, PyDM allow users to create synoptic displays via drag-and-drop of widgets at a form in a WYSIWYG (What You See Is What You Get)[3] fashion.

While this is a great solution for synoptic displays in which process variables are presented in a very well defined and static way, it imposes limitations on what can be done when business logic and client-side data processing are desired for more intelligent displays.

To overcome this limitation, PyDM takes advantage of the Qt Framework and also from the Python language to allow users to develop the view using the Qt Designer (Fig.-2) and after that, users can create a Python class (see Listing-1) that allows them to add code to the displays and interface with the widgets created using the Qt Designer through code.

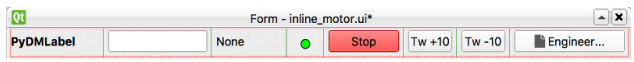

Figure 2: UI developed via drag-and-drop using Qt Designer

```
from pydm import Display

class MyDisplay(Display):
  def __init__(self, parent=None, args=None,
    macros=None):
    super().__init__(...)
    # Interact here with your ui..
    self.ui.form.setTitle('Hello ICALEPCS')

  def ui_filename(self):
    return 'inline_motor.ui'
```

Listing 1: Example of a Display class

Another possibility is to develop the whole display or application using just Python code, which is again possible due to fact that PyDM is based on Python and Qt. This approach imposes a higher learning curve since users must be familiar with the composition of layouts, the instantiating of widgets as well as the configuration of their properties via Python code.

The PyDM Tutorial[4] covers all three ways of making displays and walks the user through each step with details.

NEW WIDGETS MADE EASY

PyDM widgets are data source agnostic, this means that they have no knowledge of the origin of the data coming to them.

The widgets provided with PyDM rely on seven key pieces of information from the data-sources: connection sta-

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA135

tus, current value, alarm status, write access status, enumeration strings, engineering units and precision.

From the seven items above, only *connection status* and *current value* are needed for display widgets and additionally *write access status* is needed for input widgets. All the other information is optional and used to enhance the user experience.

Creating New Widgets

In order to help developers with the development of new widgets and establish a uniform interface, PyDM provides a base class hierarchy for widgets (See Fig.-3).

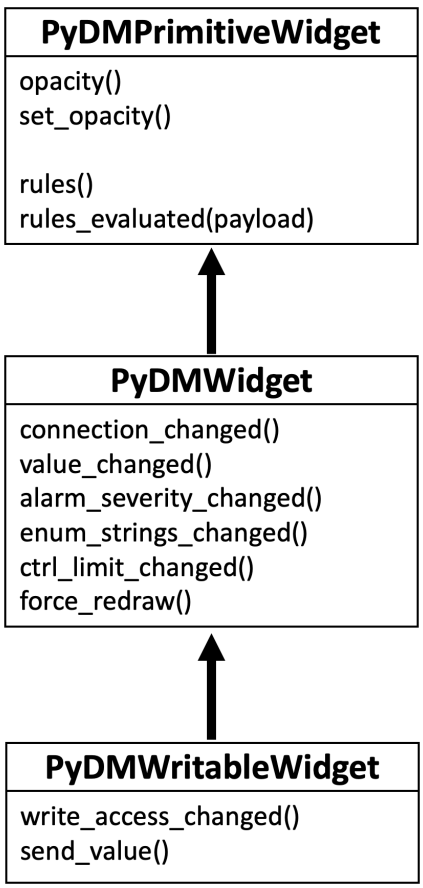

Figure 3: PyDM helper class hierarchy for widgets

PyDMPrimitiveWidget is the base class for all PyDM compatible widgets and it provides the needed pieces to add *Rules* support for custom widgets. Most of the time, custom widgets that require no channels, e.g. containers, embedded displays, auxiliary buttons such as *shell command*, or widgets that require many channnels that would not be configured through a simple text property, e.g. plots, will inherit from *PyDMPrimitiveWidget*.

PyDMWidget is the base class for all display widgets, e.g. as label, image display, scale indicator, tab bar, etc. By inheriting from *PyDMWidget*, developers get the whole machinery in place to interact with a channel. By extending the callback methods, one can easily create a new widget that talks with a data source (See Listing-2 in which we demonstrate how one can create a Progress Bar widget that displays its progress based on the value of a channel data source).

PyDMWritableWidget extends *PyDMWidget* and adds the callback for write access as well as the machinery to disable the widget in case the data source informs that the requested channel is read-only.

```
from qtpy.QtWidgets import QProgressBar
from pydm.widgets.base import PyDMWidget

class MyProgBar(QProgressBar, PyDMWidget):
    def __init__(self, parent=None, init_channel
    =None):
        super().__init__(...)

    def value_changed(self, new_value):
        super().value_changed(new_value)
        self.setValue(new_value)
```

Listing 2: Example of a Progress Bar widget

Widgets and Qt Designer

Since widgets are Python code, as long as they are installed as Python packages or can be reached through the *PYTHONPATH*, PyDM will be able to run with a custom widget without the need to tweak the PyDM package itself.

This approach is valid for Displays that are developed with code and no other work is required, but users won't be able to find this new widget in Qt Designer for a drag-and-drop display unless this widget is transformed into a plugin.

PyDM provides a simple method to create plugins that can be found by the Qt Designer. Using as an example the widget from Listing-2, developers would need to utilize the *qtplugin_factory* in order to add the needed interfaces for Designer (See Listing-3)

```
from pydm.widgets.qt_plugin_base import
    qtplugin_factory
from my_widgets import MyProgBar

# ProgressBar plugin
MyProgBarPlugin = qtplugin_factory(MyProgBar,
group="My Custom Widgets")
```

Listing 3: Example of a Qt Designer Plugin

Once the plugin is available, Qt Designer inspects the paths defined in the *PYQTDESIGNERPATH* environment variable and loads the PyQt plugins for each file with a filename ending in *_plugin.py*, e.g. *my_custom_plugin.py*.

The *_plugin.py* file must import the plugin created at Listing-3.

The end result will be a new category at the Qt Designer's Widget Box called "My Custom Widgets" with the "ProgBar" widget inside which can now be used to create displays via drag-and-drop.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA135

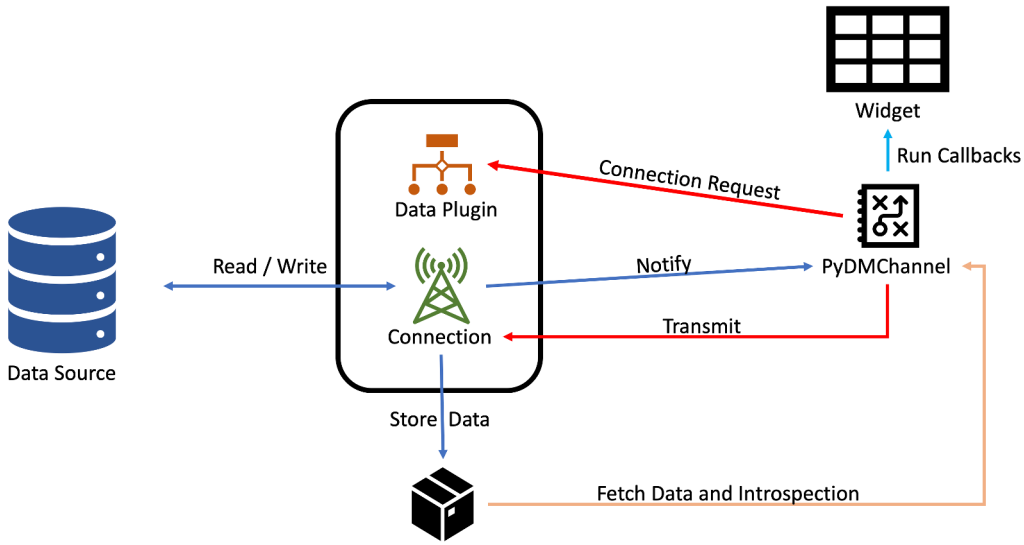

Figure 4: With the new PyDM 2.x Data Architecture, a widget with a PyDMChannel instance requests a connection to a data plugin. When new data is available from a data source (typically a control system), the Connection instance adds data to the DataStore and notifies the channel instance, which then runs the subscribed callbacks on the Widget. For writing operations, the channel instance transmit the payload created by the widget to the connection, which writes the data to the data source.

CONNECTING WIDGETS TO DATA

As discussed above, PyDM base widgets are data source agnostic and rely on seven basic pieces of information.

PyDM widgets connect to data sources via PyDMChannel instances that are linked to a certain Data Plugins.

Data Plugins are Python classes that provide data from a specific type of data source to one or more PyDMChannel instances. A Data Plugin must have a unique identifier for its protocol (e.g. "ca" for EPICS Channel Access, "archiver" for Archiver Appliance, "modbus" for ModBus plugins) and also a PyDMConnection class definition so connections to the data source can be established using this Data Plugin.

PyDM's dynamic data plugin loading system looks for data plugins in folders specified in the *PYDM_DATA_PLUGINS_PATH* environment variable. The requirements for the autoloader are:

- The filename must end with *_plugin.py*;
- The **protocol** identifier must be unique;
- The plugin class must inherit from *PyDMPlugin*;
- The plugin class must provide a class for connection that inherits from PyDMConnection;

In the PyDM 1.x series, only scalar and scalar arrays were allowed to be transferred as values to Channels via the Qt signals, which imposed a limitation on the type and amount of data that could be handled. Moreover, this design also imposed a copy of the data for each connection tied to a data plugin connection since each piece of data was transferred using an individual Qt Signal which was sent to the proper slot at each of the widgets sharing the same connection. With the PyDM 2.x series, the data transferring mechanism was refactored in such a way that data plugins now store a data payload dictionary in a central storage area called the *DataStore* (See Fig.-4).

The DataStore class contains two dictionaries, data and introspection. Both are keyed on the channel address, and the values are the data payload and a introspection lookup table respectively. This new design allows PyDM to store structured and complex data to be consumed by the widgets. The introspection lookup guides PyDM base widgets on which fields in the payload are used for the seven key data items needed for a widget to work.

Data Plugin specific widgets can take advantage of additional metadata stored in the DataStore to enhance the user experience and provide capabilities that were not possible with the previous design of PyDM 1.x series.

A PyDMChannel is a class that connects a widget to a specific data plugin through the specific protocol identifier along with a connection string according to the format defined by the data plugin.

When new data is available at the PyDMConnection, it stores the data in the DataStore and emits a signal to all channels connected to it notifying them that new data is available. The PyDMChannel instances fetch data from the DataStore for the specific channel address and execute the subscribed callbacks for the widget with the new data and introspection information.

Creating a New Data Plugin

PyDM provides two data plugins out of the box, EPICS Channel Access and Archiver Appliance due to the fact that

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA135

the SLAC particle accelerators use EPICS for their control system. As described above, one can create new data plug-ins and use them with PyDM and its widgets.

```
from pydm.data_store import DataKeys
from pydm.data_plugins.plugin import PyDMPlugin,
     PyDMConnection

class Connection(PyDMConnection):
    def __init__(self, ...):
        super().__init__(...)
        self._value = 0
        self.data[DataKeys.CONNECTION] = True
        self.data[DataKeys.WRITE_ACCESS] = True
        self.send_new_value()

    def send_new_value(self):
        if self._value is not None:
            value = self._value
            self.data[DataKeys.VALUE] = value

        self.send_to_channel()

    def receive_from_channel(self, payload):
        new_val = payload.get(DataKeys.VALUE,
    None)
        self._value = new_val
        self.send_new_value()

class NumberEchoPlugin(PyDMPlugin):
    protocol = "echo"
    connection_class = Connection
```

Listing 4: Example of a PyDM Plugin that echoes a number value using the default introspection keys

Widgets can connect to the data plugin at Listing-4 using ”echo://test” in which ”echo” is the protocol and ”test” is the address identifier.

BRING YOUR EXTERNAL TOOLS

Displays and applications often need to be integrated with facility-specific electronic logbooks, control system utility software and more.

One of the PyDM design principles is to not enforce SLAC-specific tools as part of the framework, but instead offer a well-defined and flexible interface for users and developers to extend it and easily connect to their own tools.

PyDM refer to those tools as *ExternalTools*, which are dynamically loaded into PyDM following the same concept as the data plugins (with some small differences):

– The filename must end with *_tool.py*;

– The tool class must inherit from *ExternalTool* (See Listing-5);

– The external tool file must be reachable through the *PYDM_TOOLS_PATH* or loaded via the PyDM Main Window menu ”Tools, Load...”;

Developers can define if an external tool is to be used with widgets and/or without widgets. If a tool is to be used with a widget, it will be rendered in the Context Menu (Right Click) for a widget if the tool is compatible with that particular widget, otherwise it won't be added to the menu to avoid confusion and mistake by users. If the tool is to be used without widgets, it will be rendered in the PyDM main window Tools menu.

When tools are invoked the *call* method will be invoked, sending the channels in the case of a widget-compatible tool and the sender, which is generally the widget. In the case of a tool that is not to be used with widgets, both channels and sender will be None.

```
import subprocess
import logging
from pydm.tools import ExternalTool
from pydm.utilities.iconfont import IconFont
from pydm.utilities.remove_protocol import
    remove_protocol

logger = logging.getLogger(__name__)

class ProbeTool(ExternalTool):

    def __init__(self):
        icon = IconFont().icon("cogs")
        name = "Probe"
        group = "EPICS"
        use_with_widgets = True
        kwargs = {"icon":icon, "name":name, "
    group":group, "use_with_widgets":
    use_with_widgets}
        super().__init__(**kwargs)

    def call(self, channels, sender):
        cmd = "probe"
        args = [cmd]
        if not channels:
            channels = []
        for ch in channels:
            args.append(remove_protocol(ch.
    address))
        try:
            subprocess.Popen(args, stdout=
    subprocess.PIPE, stderr=subprocess.PIPE)
        except Exception as e:
            logger.error("Error while invoking
    Probe. Exception was: %s", str(e))
```

Listing 5: Example of an External Tool which opens an external program called Probe and sends as parameter the channel used at the Widget.

CUSTOMIZING THE LOOK AND FEEL

The Qt Framework provides a great feature called Qt Style Sheet[5], with terminology and syntactic rules that are almost identical to CSS (Cascading Style Sheets)[6].

Based on that, PyDM widgets do not impose style sheet configuration to allow for maximum customization of the look and feel by developers and end users to support facility-specific color schemes, typography, etc.

PyDM provides properties on widgets that can be used as selectors when developing custom style sheets (See Listing-6 in which we use custom properties such as *error* and *interlocked*). QSS extends CSS by allowing users to tweak Qt properties via the style sheet rules.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA135

```
PneumaticValve[error="Lost Vacuum"] #icon {
    qproperty-penStyle: "Qt::DotLine";
    qproperty-penWidth: 1;
    qproperty-brush: red;
}
PneumaticValve[interlocked="true"] #interlock {
    border: 2px solid red;
}
PneumaticValve[interlocked="false"] #interlock {
    border: 0px;
}
```

Listing 6: Example of a Qt Style Sheet (QSS) for a Pneumatic Valve widget customization using property selectors and changing Qt properties via the style sheet.

CONCLUSION

PyDM can be extended beyond static displays by leveraging the Python language and its wide ecosystem of packages. Users can create intelligent applications, new data plugins to interface with different kinds of data sources, new widgets, and integrate with third-party external tools with minimal code.

The flexibility and extensibility of the framework opens new avenues for innovative features and wide usage at multi-disciplinary facilities - not only at particle accelerators and science laboratories.

ACKNOWLEDGMENTS

The authors would like to thank all the contributors to the PyDM code as well as users that provided valuable feedback, bug reports and feature requests.

REFERENCES

[1] A. Babbitt, L. Jose, M. Carvalho, M. Gibbs, and E. Williams, "An Evaluation of EDM Replacement Candidates at SLAC," 2015, unpublished.

[2] The Qt Company Ltd. (2019). Qt Designer Manual, https://doc.qt.io/qt-5/qtdesigner-manual.html

[3] Wikipedia contributors, *Wysiwyg*, 2001. https://en.wikipedia.org/wiki/WYSIWYG

[4] M. Gibbs and H. Slepicka. (2018). Pydm tutorial, https://slaclab.github.io/pydm-tutorial/

[5] The Qt Company Ltd. (2019). The style sheet syntax, https://doc.qt.io/qt-5/stylesheet-syntax.html

[6] H. W. Lie and B. Bos, *Cascading Style Sheets: Designing for the Web*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997, ISBN: 0-201-41998-X.