ISBN: 978-3-95450-227-1 ISSN: 2673-5490 doi:10.18429/JACoW-IPAC2022-TUPOST033

A PYTHON FRAMEWORK FOR HIGH-LEVEL APPLICATIONS IN ACCELERATOR OPERATIONS

J. Chrin, V. Erçağlar, T. Schietinger, Paul Scherrer Institut, 5232 Villigen PSI, Switzerland

Abstract

A Python graphical framework providing reusable components to facilitate the development of accelerator applications, that meet the basic requirements of experts and operators alike, is presented. Such a collective approach serves to bridge the gap between the expert developer and the operational team, resulting in applications that are inherently cohesive, durable and easily navigable. The operational advantages and underlying principles are exemplified in a reference application that provides executable examples of customary practices, and further highlights a number of composite and control system-enabled widgets.

PERSPECTIVE AND MOTIVATION

The development of high-level applications crosses the domain of several groups, each possessing a distinctive skill set and ambition. Typically the 'expert' application is developed by the scientist, engineer, whose primary interest is in providing an interface that details the hardware capabilities of the system. The 'operator' application on the other hand may require only a condensed view and one that offers automated procedures geared towards the everyday operation of the accelerator. Other, beam dynamics, applications will undertake specific measurements crucial to the optimization of the accelerator, and may further require interaction with accelerator models, message reporting capability, and the means to store and retrieve data resulting from their analyses. Many such measurements and procedures are first deployed by the expert in the commissioning phase of the accelerator before ultimately being delegated to machine operators once routine operation is established. The diversity of developers, however, inevitably leads to a variety of frameworks and appearances, with some duplication in that similar functionalities are instigated in different manners and with a non-uniform behaviour. These subtle details of cross-domain application development are acknowledged to add to the tasks and challenges faced by the operator [1, 2].

This inadvertent disparity between applications, however, can be alleviated by supplying a controlled, coordinated, and configurable graphical interface wherein common functionality is provided through predetermined inputs with well defined responses. Applications then become inherently homogeneous, and developers are further relieved of implementing peripheral tasks, releasing time to prioritize their particular area of interest.

In the following, a Python graphical framework is described that provides a base class that can be inherited by the application developer, and adapted to specific needs. The focus is on the components and methodology that constitute the framework, rather than any specific application that is built upon the infrastructure, as applied within the context of SwissFEL and the Swiss Light Source (SLS).

A PYTHON GRAPHICAL FRAMEWORK

The Python programming is presently enjoying a high profile within the accelerator community being the preferred language for application development at facilities of various size [3, 4]. While scripting is regarded as gratifyingly intuitive and powerful, our applications are inherently graphical and this in itself adds a new level of complexity. To this end, PyQt, a Python graphical user interface (GUI) module that connects to the Qt C++ framework [5, 6], is the prevailing choice. Significantly, in this work, Qt modules are imported through the QtPy abstraction layer [7] allowing our framework to be used effortlessly across Qt versions.

The Qt library provides much functionality, with numerous classes. Its specific application domain of windows, widgets, layouts, colours, shapes, and more, however, presents the developer with expansive possibilities to interrogate and discern before converging on a finely-tuned appearance. Furthermore, and rather critically, any complex physics analysis, or any other resource intensive operation having a long running time, needs to be delegated to a separate thread. Here, Qt's dedicated thread support is a fitting option given that our applications interact with other components of the Qt library. In this way, the main thread remains responsive at all times, widgets with read backs continue to be updated, and user interaction, where permitted, is not interrupted. Such considerations are a prerequisite in gaining a satisfactory user experience. Once these challenges are overcome, the programmed solutions may be presented to the developer in a reusable form.

Apps4Ops: A Characteristic Style

The GUI implementation class follows Qt's customary 'main-window-style' approach that offers predefined options for user input in the form of a menu bar, toolbar, status bar, a central widget, and dock windows. The adoption of such a ready-made, integrated approach, at the outset, is an important factor in achieving a polished, intelligible, and navigable user interface. The static visualization of data is accomplished through the comprehensive Matplotlib library [8, 9]. For the display of continuous, real-time data, however, the PyQtGraph [10] library, based on Qt's GraphicsView framework, is preferred for its speedier response time. The principle components that constitute the framework, and other practicalities that play a part in achieving the desired homogeneity among applications, coupled with an optimized user experience, are elaborated.

ISBN: 978-3-95450-227-1 ISSN: 2673-5490 doi:10.18429/JACoW-IPAC2022-TUPOST033

Control System-enabled Widgets The low-level control system is built on the EPICS (Experimental Physics and Industrial Control System) framework [11], which comprises an extensive set of software tools tailored to the needs of particle accelerators and large-scale experiments. Interaction with the EPICS-based control system is established through a proven C++ abstraction layer [12–14], for which high-performance bindings to Python have been provided using the Cython technology [15]. Several control system-enabled Qt widgets, and composites, have been made available. These connect to a gateway module that handles their connectivity management, propagates updates, and ensures their display conforms to the recognized style.

Configuration The provision of an application framework that spans across different accelerators requires a multi-faceted configuration mechanism that allows the generic, the accelerator-specific, and the application's user behaviour to be configured via a hierarchy of configuration files. Their tasks may be numerous, hence only certain representative usage is outlined here.

At the core level, a Qt style sheet file defines the appearance of GUI elements, including fonts, sizes, and colour schemes, in accordance with the stipulated style guide. In addition, a configuration file storing simple data structures in JSON (JavaScript Object Notation) format [16], provides information such as default menu options, and the destination of various output files. The second level configuration file provide handles to several accelerator-specific parameters, e.g., exposing the EPICS channels that constitute the accelerator-specific header widget. The application's user configuration file may be as extensive as the user requires. It can also be used to enact a number of predefined widgets and associated tasks, e.g., checkboxes that control simulated runs, debug options, etc., and also to formulate further GUI components that provide input parameters that dynamically feed into the application and its analysis tasks. A destination subpanel for analysis results, typically in graphical form, can also be entered.

The configuration files, and other small resources files such as icons and help pages, are registered and safe guarded in an accelerator-dependent resources file from which a Python module is generated. The resource module is then imported by the application, from where the files can be accessed directly. No assumptions need then be made about the location of the files, or the application's working directory; with help files also incorporated into the resources module, no external web help pages need be summoned. The default generic and accelerator level configurations can also be replaced by user-supplied files through command line interface flags.

Initialization Applications are expected to initialize as speedily as is feasible. Faster times can be achieved by assigning any time-intensive 'loading' method to a single-shot timer with zero-timeout; the loading method is then executed when the event queue next allows, i.e., once the main window initialization is complete. Another helpful approach is to inform the application at startup of its associated channels. The underlying EPICS client library allows connections to be established in unison with a single call. Having connections established in advance of widget creation allows pertinent information concerning the channels to be readily available to any associated widget and allow its configuration to be determined dynamically and spontaneously. These data are further stored locally and can be recalled in the event of a channel being unavailable at start-up time. The widget is automatically reconfigured once a connection is established. Likewise, other application's settings, such as the window dimensions and positions, will have been previously optimized, remembered locally, and recalled on startup, providing consistency between sessions. Nevertheless, despite these optimizations, where complex applications may still take a few seconds to startup, which is invariably the case, a so-called 'splash screen' is shown ahead of the forthcoming main window display to reassure the user that the application is indeed loading. A progress bar on the splash screen further informs the user of the time remaining for the application to initialize.

Finalization Best practice for a tidy application exit dictates that any pending actions are first completed or at least interrupted gracefully. This may entail checking if any analysis threads are still running and that control system parameters are restored to their pre-measurement values. Explicitly releasing application resources, such as those managing EPICS connectivity, reassures the developer of sound memory management in the underlying libraries.

Menu Bar and Toolbar Menus and toolbars furnish a GUI with a convenient graphical presentation of user commands that perform specific tasks. Placements in the main window's menu bar are grouped into submenus consisting of a list of logically related pull-down items. A dynamically created menu bar is provided for operations on files, as necessitated for the display of recently used files, while other static menu bars are provided for performing peripheral tasks such as:

- initiating output actions, e.g., saving data to HDF5 files [17], reporting to the electronic logbook [18],
- capturing images and screenshots,
- connectivity to printers,
- interrogating the standard output file, clearing entries in the message log window,
- displaying procedural help pages and information that discloses 'behind-the-scenes' details, such as software versions and the application responsible,
- enabling a tidy exit.

Actions to application-specific items can be customized by the user by resetting the base class implementations. The application menu items are selected by the application's user configuration file.

The toolbar, located immediately below the horizontal menu bar, is populated with icons that provide quick access to the most commonly used commands.

ISBN: 978-3-95450-227-1 ISSN: 2673-5490 doi:10.18429/JACoW-IPAC2022-TUPOST033

Central Widget An accelerator-specific header widget takes ownership of the top area of the central widget. It displays the most pertinent information related to the current status of the machine. The remainder of the central widget is partitioned into dedicated subpanels that serve as destination locations for operator/expert input parameters to measurement procedures, the visualization of results, and the display of log messages. The partition of the central widget can be configured to the application's needs through the application-specific configuration file.

Message Reporting The use of a carefully considered message reporting interface ensures a consistent set of data is delivered to the log window. Mandatory information includes the origin and timestamp of the message, coupled with a severity level. The data are supplemented by a number of optional fields that are filled at the discretion of the user. In the event of a fatal condition that prevents the application from continuing, the provision of the developer to supply a meaningful and helpful message, that may also propose a solution, can hasten the return to normal operation.

Analysis Module An example analysis module which inherits Qt's thread class demonstrates the proposed analysis procedures. Input data from the operator or expert panels are automatically propagated to the analysis class. These, together with the analyzed data, are encapsulated into dictionaries for subsequent display and storage. Stored data can be recalled into the application and reanalysed, thereby demonstrating the seamless integration of logged and real-time accelerator data into the same framework. Interthread communication, as required for progress updates and message reporting, is accomplished through Qt's distinctive signal and slots mechanism.

Deployment and Experience

A generic, skeleton application exemplifying features of the graphical framework acts as a reference for developers. An application that makes typical use of the infrastructure is shown in Fig. 1. The emphasis has been on creating fastidious modules to help ensure excellent operational ability. Even the simplest of push buttons must execute its task impeccably and be equipped to act with integrity in the event of an anomaly. The adoption of the framework ensures that applications have recognizable buttons and icons that lead to predictable actions, and accelerates the creation of user interfaces by relieving developers of implementing peripheral tasks, thereby releasing time to concentrate on their particular area of interest. The involvement of both graduate and undergraduate students in research and creative activities is also an integral part of our organization; their use of the framework increases productivity and readies their work for operational use. It further promotes sustainability in that only a single framework need be maintained and extended for basic application needs.

The deployment of the framework has proven to be an iterative process as essential feedback is returned from a focus group of users. The sharing of their experience, the problems encountered, and subsequent proposals, feature in a collective decision on the evolution of an application. The collaborative approach brings flexibility into our section, promotes inter-department activities, and equally broadens our own horizon.

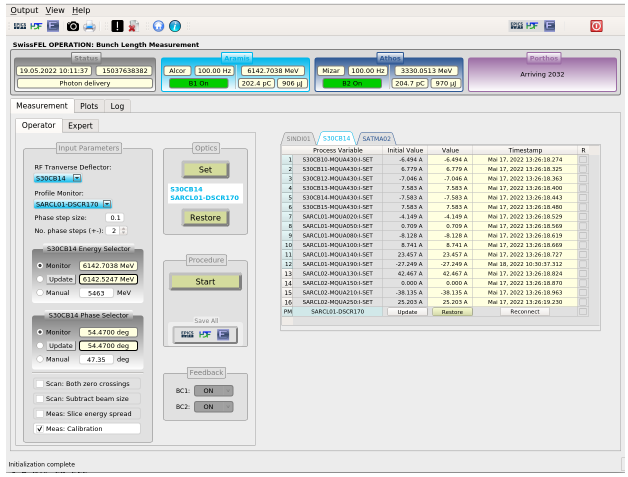

Figure 1: An example of an application using the 'main-window-style' framework. It comprises a menu bar, a toolbar, an accelerator-specific header widget, customized epics-enabled widgets, a status bar, and partitions for operator and expert inputs and procedures, the visualization of data, and the display of log messages.

CONCLUDING REMARKS

Several common, peripheral requirements for high-level applications have been identified and incorporated into a dedicated graphical framework that serves as a foundation for the development of applications that are collectively cohesive, functional, durable, and navigable. The *Apps4Ops* software architecture has encompassed several renewed efforts as our understanding of operator requirements, and the capabilities of the underlying software, has advanced with experience. It has now been applied to a number of beam dynamics applications developed by accelerator physicists[1] achieving a level of homogeneity and consistency that is shaping the evolution of high-level applications for operations at the existing SwissFEL and the upcoming SLS 2.0 [19, 20].

ACKNOWLEDGEMENTS

We are grateful to our many colleagues in the Accelerator Operation and Development Dept. of the Large Research Facilities (GFA) Division, for their valuable feedback, especially concerning enhancements to the user experience. The Python environment within GFA is provided by the Electronics and Control Systems Dept. One of us (JC) thanks Ivan Sinkarenko, CERN, for fruitful Pythonic discussions.

[1] "You Are The Music... We're Just The Band", Trapeze, 1972

ISBN: 978-3-95450-227-1 ISSN: 2673-5490 doi:10.18429/JACoW-IPAC2022-TUPOST033

REFERENCES

[1] D. Jacquet, "Breaking the wall between operational and expert tools", in *Proc. 7th Evian Workshop on LHC Beam Operation*, Evian Les Bains, France, Dec. 2016, pp. 157–160. https://cds.cern.ch/record/2293679

[2] S. Deghaye, "How to improve interactions with the control system", in *Proc. 7 th Evian Workshop on LHC Beam Operation*, Evian Les Bains, France, Dec. 2016, pp. 161–166. https://cds.cern.ch/record/2293526

[3] T. Zhang, J. H. Chen, B. Liu, and D. Wang, "Python-based high-level applications development for Shanghai soft X-ray free-electron laser", in *Proc. 12th Int. Computational Accelerator Physics Conf. (ICAP'15)*, Shanghai, China, Oct. 2015, pp. 23–25. doi:10.18429/JACoW-ICAP2015-MODWC4

[4] P. Elson, C. Baldi, and I. Sinkarenko, "Introducing Python as a supported language for accelerator controls at CERN", in *Proc. 18th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'21)*, Shanghai, China, Oct. 2021, pp. 236–241. doi:10.18429/JACoW-ICALEPCS2021-MOPV040

[5] Qt, https://www.qt.io

[6] PyQt, https://riverbankcomputing.com/software/pyqt

[7] QtPy, https://pypi.org/project/QtPy

[8] J. D. Hunter, "Matplotlib: A 2D graphics environment", *Computing in Science and Engineering*, vol. 9, no. 3, pp. 90–95, 2007. doi:10.1109/MCSE.2007.55

[9] Matplotlib, https://matplotlib.org

[10] PyQtGraph, https://www.pyqtgraph.org

[11] EPICS, https://epics-controls.org

[12] CAFE, http://cafe.psi.ch

[13] J. Chrin, "An update on CAFE, a C++ Channel Access client library and its scripting language extensions", in *Proc. 15th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'15)*, Melbourne, Australia, Oct. 2015, pp. 1013–1016. doi:10.18429/JACoW-ICALEPCS2015-WEPGF132

[14] J. Chrin, M. Aiba, and J. Snuverink, "A Channel Access software platform for beam dynamics applications in scripting languages", *J. Phys.: Conf. Ser.*, vol. 1350, p. 012155. doi:10.1088/1742-6596/1350/1/012155

[15] J. Chrin, "A Cython interface to EPICS Channel Access for high-level Python applications", in *Proc. 11th Int. Workshop on Personal Computers and Particle Accelerator Controls (PCaPAC'16)*, Campinas, Brazil, Oct. 2016, pp. 21–24. doi:10.18429/JACoW-PCAPAC2016-WEUIPLCO04

[16] JSON, https://www.json.org

[17] HDF®, https://www.hdfgroup.org

[18] ELOG, https://elog.psi.ch/elog

[19] A. Streun *et al.*, "SLS-2 - the upgrade of the Swiss Light Source", *J. Synchrotron Radiat.*, vol. 25, pp. 631–641, 2018. doi:10.1107/S1600577518002722

[20] A. Streun, "SLS 2.0, the upgrade of the Swiss Light Source", in *Proc. 13th Int. Particle Accelerator Conf. (IPAC'22)*, Bangkok, Thailand, Jun. 2022, paper TUPOST032, this conference.