ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-THPV015

NEW TIMING SEQUENCER APPLICATION IN PYTHON WITH Qt DEVELOPMENT WORKFLOW AND LESSONS LEARNT

Z. Kovari, G. Kruk, CERN, Geneva, Switzerland

Abstract

PyQt is a Python binding for the popular Qt framework for the development of desktop applications. By using PyQt one can leverage Qt's aspects to implement modern, intuitive, and cross-platform applications while benefiting from Python's flexibility. Recently, we successfully used PyQt 5 to renovate the Graphical User Interface (GUI) used to control the CERN accelerator timing system. The GUI application interfaces with a Java-based service behind the scenes. In this paper we introduce the generic architecture used for this project, our development workflow as well as the challenges and lessons we learnt from using Python with Qt. We present our approach to delivering an operational application with a particular focus on testing, quality assurance, and continuous integration.

TIMING CONTROL APPLICATION

Accelerator Timing System

CERN continuously delivers particle beams to a range of physics experiments (end-users), each posing strict, detailed requirements with respect to the physical characteristics of the particle beam to be delivered. Hence, particle beams traverse a number of accelerators while being manipulated in various ways. For this to happen, the accelerators repeatedly "play" pre-defined cycles, which usually consist of an injection-acceleration-ejection sequence. This in turn involves many concurrent beam manipulations including particle production, bunching, cooling, steering, acceleration and beam transfer – all of which must occur at precise moments in time, often with microsecond or even nanosecond precision. The role of the Timing system is an orchestration of all these activities, ensuring that the accelerator complex behaves as expected as a function of time.

Each accelerator at CERN is associated with a Central Timing system (CT) which, based on the configuration provided by the operators and dynamic input such as external conditions, calculates in real time so-called *General Machine Timing (GMT)* events that define key moments in the accelerator cycles such as beginning of the cycle, injection, ramp, extraction, etc.

GMT events are then transmitted to *Front-End Computers (FECs)* via a dedicated cabled network known as the *GMT network*.

On the FEC side, the GMT cables are connected to *Central Timing Receiver* (CTR) modules, which decode the received GMT events and allow generation of derived local events (with optional delay) in the form of software interrupts and physical pulses for the accelerator hardware.

Timing App Suite

The control of the timing system is done via a dedicated GUI application that allows operation crews to define a collection of cycles composing a so-called *beam* (see Fig. 1). A Beam is executed by the central timing to transfer and accelerate a particular particle beam from the source, through the intermediate accelerators, up to the final experiment. Beams are then used to build a *Beam Coordination Diagram (BCD)* that defines sequencing of different particle beams sent to different destinations as illustrated in Fig. 2.

Following the renovation of the central timing itself, it was decided to also renovate the 20-year-old application used to control it, modernizing its architecture, improving usability aspects and taking advantage of the new features provided by the central timing.

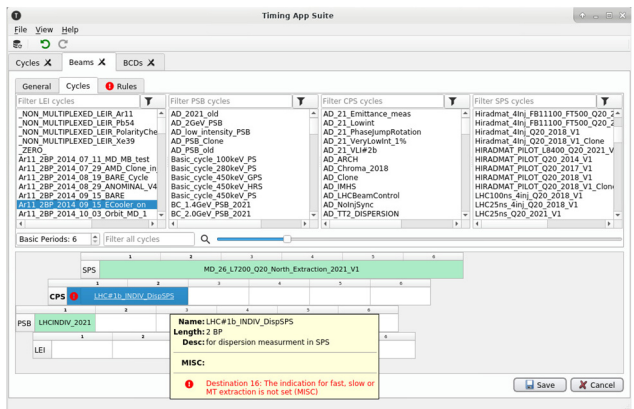

Figure 1: Timing *Beam* editor.

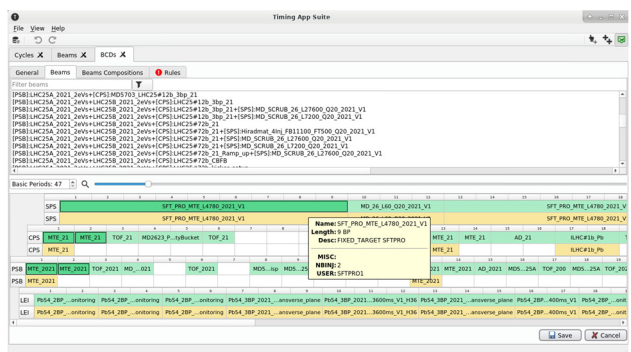

Figure 2: *Beam Coordination Diagram* editor.

ARCHITECTURE

The application was designed and implemented in a 3-tier architecture (see Fig. 3): a GUI, implemented in Python using PyQt toolkit, communicating via HTTP with RESTful services implemented in Java, and with the Cen-

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-THPV015

tral Timing process (C++) via PyJAPC [1] a Python wrapper over the Controls Middleware (CMW) [2] communication library.

The Java server relies on Spring Boot and several features provided by the Spring framework such as REST controllers or JPA database access, along with dependency injection and configuration services.

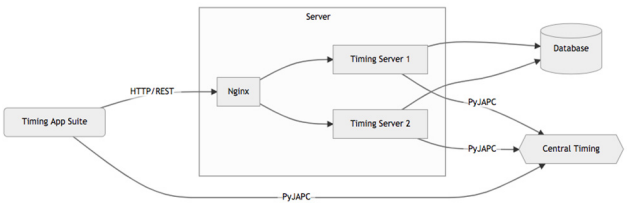

Figure 3: Timing App Suite architecture.

The server is completely stateless allowing simple deployment of two instances with an Nginx [3] proxy server in front, and enabling rolling updates.

GRAPHICAL USER INTERFACE

The GUI application is split into four different layers:

- ***Data***: Qt independent CERN-related domain classes representing the Timing ecosystem.
- ***Model***: Qt model layer responsible for mapping Data to the View [4].
- ***View***: the User Interface widgets [4].
- ***Controller***: optional intermediate layer between view and model, typically handling signal and slot connections that need more sophisticated business logic and cannot be connected directly between the view and the model.

Model

The model layer in Qt's architecture has two responsibilities. On the one hand, it maps the Data objects to the View, basically saying which data item should be displayed at which index (*QModelIndex* [5]). On the other hand, it also tells the View component how that data should be rendered, e.g., in which font, background color, and with what tooltip or icon.

View

In most cases, the user interface was designed in Qt's built-in editor, *Qt Designer,* which is also easily accessible through the pyqt5-tools [6] Python package. *Qt Designer* is a simple-to-use editor that allows dragging and dropping Qt widgets and creating user interfaces without any programming.

Such created panels are then saved to *.ui* files, which are re-used in the Timing Control Python application. There are two ways to do that:

- Using PyQt's *uic* module to load the generated UI files [7].
- Generating Python code from the UI files [8].

The 2nd solution was used, as this approach provides content assists in the *PyCharm* IDE, and in general, helped to better integrate the View components into the source code (e.g. finding usages throughout the entire source code).

The drawback of such an approach was that an additional step was needed between editing the UI in Qt Designer and then using it in the application. The *pyqt5ac* 3rd party library was used for code generation to automatically convert multiple UI files into Python [9].

Communication Between Model and View

The model layer in Qt handles many aspects of what is displayed, and how, from *Data* to *View*. Due to this close connection, it is also simple to directly handle user interactions between view and model by connecting the necessary signals from the view to the model's slots, and vice-versa. That being said, an additional controller layer was introduced to handle some of the more complex signal-slot connections. This also allowed to further modularize the application and re-use some of the view, model, or even controller components in other panels.

Testing

One of the great benefits of using Qt/PyQt was automated testing. Using the *pytest-qt* Python library [10], over 500 test cases were implemented that verify the user interface. The *pytest-qt* package provides a *qtbot* object that can be used to spawn and interact with the GUI application, e.g., clicking buttons, selecting, or even dragging and dropping elements. Behind the scenes, the package uses Qt's *QTest* namespace [11] with some additional features extended (e.g., "stop" method [12]).

TIMING JAVA SERVER

The server side follows the classical pattern with three layers [Fig4]: (1) the data access layer consisting of JPA repositories and corresponding entities mapped to the database tables, (2) the service layer that uses one or more repositories to perform CRUD operations and does the translation between entities and Data Transfer Objects (DTOs), and (3) the REST controllers layer that handles HTTP requests, relying on services.

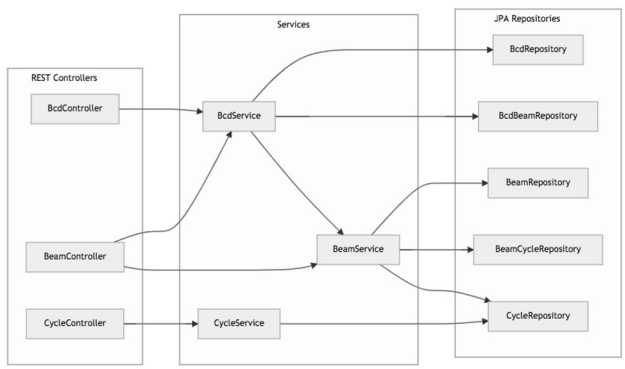

Figure 4: Simplified diagram of Timing server components.

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-THPV015

CONTINUOUS INTEGRATION

CERN's self-hosted *GitLab* instance was used to host the application's code repository and set up automated pipelines [13] for continuous quality assurance.

Each time a commit is pushed to the Git repository, a *GitLab* pipeline (see Fig. 5) runs and automatically verifies the codebase of that specific commit. Verification means the following:

- GUI tests run automatically on *GitLab*. To run UI tests in a container, Xvfb [14] (a virtual framebuffer X server that can run on machines with no display) was configured.
- Static code analyzer tools (such *flake8* [15], *pylint* [16], and *mypy* [17]) are used to analyze the correctness of the code. These tools can detect non-standard code usages that violate PEP-8, or even potential problems such as missing arguments or missing imports. Based on the development experience, these tools are easy to set up and do help to elevate code quality but don't replace frequent code reviews and testing.

Figure 5: Pipeline to build, test and analyse the code.

CONTINUOUS DEPLOYMENT

On the same *GitLab* pipelines that verify the code, the application can also be deployed to different environments, as illustrated in Fig. 6.

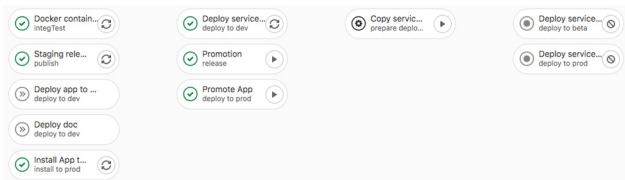

Figure 6: Deployment part of the pipeline.

To track the environments (such as DEV, STAGING, RC, PROD) the *GitLab* built-in environments feature is used [18]. *GitLab's* environments panel allows to track which commits are deployed to which environments. Furthermore, a deployment pipeline can also be quickly triggered from the same panel, and likewise a roll back to a previous version if necessary.

Application deployment is carried out with *Ansible* [19]. The Python application's wheels [20] are built on *GitLab CI*, dependencies are locked, and then with *Ansible*, binaries are transferred to the deployment server where the Python virtual environment is re-created and the application deployed.

The deployment of the backend server is also triggered from the same pipeline. The server and application code are stored in the same repository, which allows to version and deploy them together to production.

Docker images are built which contain the backend server's binaries and then deployed with *Ansible* to the various environments (DEV, STAGING, RC, PROD). One such environment comprises a *Podman* pod [21] with three running containers: two for the server to facilitate rolling updates, and one *Nginx* container to delegate the incoming requests between the two other containers.

CHALLENGES AND LESSONS LEARNT

Python for GUI

From the end of 90's until 2020, the vast majority of CERN Controls desktop applications were implemented in Java, for many years using the Swing toolkit, and more recently JavaFX.

For the new Timing application, it was decided to switch to Python and PyQt. The decision was not easy and was mainly dictated by the announcements made by Oracle for deprecation of Swing and discontinued support for JavaFX. Both frameworks are unlikely to disappear within the next few years (Swing will still remain a part of the JDK at least over the next few years), however their future beyond the time frame of 8-10 years is very uncertain. The lifetime of CERN controls applications (including Timing App Suite) is typically beyond 20 years, therefore for the development of new GUI software, requiring significant investment, it was preferred to switch to a framework that is actively developed and maintained, and whose future seems to be much more reassuring.

Python is a powerful, flexible, easy-to-learn and easy-to-use language. One needs less lines of code to perform the same task when compared to other major languages like C++ or Java, which turns into a better productivity. These advantages apply also to GUI programming. The biggest issue confronted when developing the application comes from the flexibility of the language i.e. dynamic typing and lack of compilation checks.

While the codebase of the application was growing, many internal modifications were applied, such as renaming classes, functions and variables, moving parts of the logic from one place to another, replacing one class with another etc. *PyCharm* is dealing with most of the refactoring quite well, but occasionally was failing to properly identify the type of a variable and not applying the requested change. After realizing this, type hints were consistently used all over the code, paying particular attention to function declarations i.e. parameters and return types.

The static analysis tools came in very useful to catch various issues related to the code refactoring, however the ultimate safety net to protect from these kinds of risks was the high coverage of the code by automated tests.

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-THPV015

PyQt Toolkit

Qt provides all standard components designed following the model/view architecture, therefore their usage did not pose any major problems. The concept of signals and slots for establishing communication among various components permits flexibility when dealing with GUI events and results in a smooth codebase.

PyQt is one of the most used UI frameworks for Python therefore there is a relatively big community behind it. There are many learning resources for Qt and PyQt, however a complete and up-to-date PyQt beginners tutorial, with simple and more advanced code examples was found to be missing. Answers to most specific questions could be found, but this required browsing through different forums and blogs, and often applying solutions given for C++ Qt or PyQt4, to the PyQt5-based application.

There is a certain learning curve, especially when it comes to a bit more advanced usage of UI widgets, although once foundations were established and the concepts became familiar, the Qt API documentation was deemed satisfactory most of the time.

The documentation of the testing library, pytest-qt, is rather modest, so learning it and setting up bases for UI tests required a bit of time to experiment and browse Qt forums.

On several occasions segmentation faults (coming from the Qt widgets) were encountered. At first these were not straightforward to understand, but with time they could be interpreted, and it quickly became possible to identify their source.

Architecture

Implementing the GUI in Python and the backend in Java did not pose any issues. The advantages of both languages and corresponding frameworks could be leveraged to develop the adequate logic in the most efficient way.

What comes as a drawback in this kind of approach is the duplication of domain objects (or Data Transfer Objects) as they are needed in both languages. The Timing API is relatively small and the domain objects quite simple, therefore implementing and maintaining them in Java and Python was a negligible effort.

CONCLUSIONS

After two decades of developing desktop controls applications using Java, it was decided to change the programming language to Python with PyQt selected as the widget toolkit. The choice was not obvious, but the end result is more than satisfactory, and the same approach is now foreseen for some other applications e.g. the CERN developed Front End Software Architecture [22] diagnostic tool (known as the FESA Navigator).

Efficiency and flexibility, while preserving readability of Python-based applications as compared to Java, comes along with the lack of compile time verification of the codebase. Therefore, a rigorous approach to Continuous Integration performing static code analysis, together with high coverage by automated testing is deemed indispensable for any operational application.

REFERENCES

[1] V. Baggiolini *et al.*, "JAPC - the Java API for Parameter Control", in *Proc. ICALEPCS'05*, Geneva, Switzerland, Oct. 2005, paper TH1.5-8O.

[2] J. Lauener and W. Sliwinski, "How to Design & Implement a Modern Communication Middleware Based on ZeroMQ", in *Proc. ICALEPCS'17*, Barcelona, Spain, Oct. 2017, pp. 45-51. doi:10.18429/JACoW-ICALEPCS2017-MOBPL05

[3] Nginx, https://www.nginx.com

[4] Qt5, https://doc.qt.io/qt-5/model-view-programming.html

[5] Qt5, https://doc.qt.io/qt-5/qmodelindex.html

[6] PyQt5 Tools, https://pypi.org/project/pyqt5-tools/

[7] Importing the UI File in Python, https://nitratine.net/blog/post/how-to-import-a-pyqt5-ui-file-in-a-python-gui/#importing-the-ui-file-in-python

[8] First Steps with Qt Designer, https://www.pythonguis.com/tutorials/first-steps-qt-creator

[9] PyQt5 Auto Compiler, https://pypi.org/project/pyqt5ac

[10] PyTest Qt, https://pypi.org/project/pytest-qt

[11] Qt5 Test, https://doc.qt.io/qt-5/qtest.html

[12] QtBot, https://pytest-qt.readthedocs.io/en/latest/reference.html#pytestqt.qtbot.QtBot.stop

[13] CI Pipelines, https://docs.gitlab.com/ee/ci/pipelines

[14] PyTest GitHub Actions, https://pytest-qt.readthedocs.io/en/latest/troubleshooting.html#github-actions

[15] Flake, https://pypi.org/project/flake8

[16] Pylint, https://pylint.org

[17] Mypy, https://mypy.readthedocs.io/en/stable

[18] CI Environments, https://docs.gitlab.com/ee/ci/environments

[19] Ansible, https://www.ansible.com

[20] Python Wheels, https://realpython.com/python-wheels

[21] Podman, https://mohitgoyal.co/2021/04/23/spinning-up-and-managing-pods-with-multiple-containers-with-podman

[22] M.Arruat *et al.*, "Front-End Software Architecture", in *Proc.* in *Proc. ICALEPCS'07*, Knoxville, Tennessee, USA, Oct. 2007, paper WOPA04.