ISBN: 978-3-95450-237-0 ISSN: 2673-5512 doi:10.18429/JACoW-PCaPAC2022-THPP9

SIMPLE PYTHON INTERFACE TO FACILITY-SPECIFIC INFRASTRUCTURE

J. Gethmann*, E. Blomley, P. Schreiber, M. Schuh, W. Mexner, A.-S. Müller
Karlsruhe Institute of Technology, Karlsruhe, Germany
S. Marsching, aquenos GmbH, Baden-Baden, Germany

Abstract

The various particle accelerators hosted at KIT represent a complex infrastructure with a live control system interface, a data archive, measurement routines, and storage and management of metadata, among other aspects. The "KIT Accelerator Python tools" were created to provide a unified interface to all aspects of the accelerator infrastructure for both short-term student projects and basic accelerator operations. Instead of creating another custom framework, these sets of tools focus on bridging the gap between well-established libraries, our facility and accelerator specific needs. External and accelerator specific libraries are glued together to provide an interface in order to minimise the technical knowledge of the accelerator infrastructure needed by the end user. Best practice software engineering workflows of continuous integrationn were implemented to provide automatic testing, packaging, API documentation and release management. This paper discusses the general motivation and approach taken to create and maintain such a set of Python modules.

INTRODUCTION

At KIT, we operate multiple kinds of accelerators (KARA, FLUTE) and further compact accelerators are in the planning and building phase. KARA, a 2.5 GeV storage ring, a part of the KIT Light Source, with a circumference of 110 m. FLUTE [1] includes a linear accelerator. Further accelerators are planned, e. g. a plasma accelerator and a compact storage ring within the project cSTART [2]. This paper first describes the setup of the controls infrastructure, then introduce the requirements for accelerator physics' experiments and the control of the machines. Then, the strategies to solve issues and the decisions made are described. Lastly, we present some lessons that we learned in the process.

SETUP

The controls infrastructure has to integrate the required different operation modes of the different KIT accelerators with a quick and safe access by e. g. non-experienced students via users of synchrotron radiation to experts with very demanding accelerator physics and technology experiments.

The data flow starts by providing live data and control for machine parameters from magnet power supplies to diagnostics instrumentation. Relevant parameters are archived in databases that are flat for performance reasons and are furthermore highly available. Snapshots of machine settings can be saved to and restored from relational databases for selections of groups, e. g. the pre-accelerator synchrotron, in short booster, settings can be restored independently from the storage ring's settings. Furthermore, there exist electronic lab notebooks for routine operation and certain typical experiments. For machine optimisation purposes and typical measurements needed for beam dynamics simulations, MATLAB® routines write their outputs to a central file storage.

Though the controls architecture for the different accelerators is similar, there are logical stand-alone systems for each accelerator requiring different addresses and namespaces.

REQUIREMENTS

For different stakeholders, the requirements differ. Especially for students and visiting researchers, it is necessary to get easy and quick access to certain sub-systems without in-depth knowledge of how exactly these systems are set up or interact with each other. They do not need to know from which control system component the data originates nor which protocol that systems speaks. Another crucial point is a quick setup, possibly with limited permissions granted. Facility scientists need access to a broad variety of data and therefore have to interact with many systems at the same time. However, the knowledge of the specifics of each particular system is typically not required. Finally, there are the developers of such systems who want to have maintainable and flexible code, which is reusable and actually used. This includes to adopt FAIR principles. We need straightforward access to all systems without the domain knowledge of the individual sub-systems or the infrastructure's topology.

In the past, we struggled with these different user groups trying to solve similar issues with their own individual scripts. Such scattered scripts were often written in different languages, with different levels of code quality and focusing on different aspects of the same task. This resulted in making these scripts very hard to share, maintain and re-use, especially across different user groups. The user experience of systems with different interfaces for similar behaviour is not good and can lead to errors in the worst case. Lastly, substantial inside knowledge is required to be aware of such scattered scripts and where they can be found.

APPROACH

Instead of creating another framework or a generic scientific library, our approach bundles existing ones for convenient ways to use them for the KIT accelerators according to our needs.

* gethmann@kit.edu

ISBN: 978-3-95450-237-0 ISSN: 2673-5512 doi:10.18429/JACoW-PCaPAC2022-THPP9

In general, our packages provide an abstraction layer towards the various libraries. We initialise them with site-specific settings in advance, like server addresses and configuration settings, because the user does not need to know this information. In addition, changing of backend libraries can be done transparently for the user with this approach. Furthermore, the user does not need to know if some parts of the infrastructure moved to a different location, e. g. because of a new URL for a new version of a REST-API.

Nowadays, distributable well-structured Python packages can be created comparably easily with bootstrapping tools like PyScaffold. This enables one to use packages instead of a bunch of more or less sorted Python scripts where you cannot handle dependencies on each other or on third party packages well, unless you have only a single repository, which includes everything. The latter is not what we want, especially as some packages depend on large libraries like *scikit-learn* or software that is either hard to set up and configure or serves only a few special needs. Installing such software should neither be mandatory for users nor required on embedded systems. With our approach, we can use these packages for different use-cases such as pre-processing for machine learning pipelines on the KIT computer clusters, for the daily needs on one's office workstation, as well as for developments inside the accelerator control system. Especially the latter will be much more important in the near future with the increasing usage of Python SoftIOCs [3].

For office and data science applications, distributable Python packages are the preferred format for installation as the installation procedure of such is operating system-independent. It is already known to many users and usable without additional permissions beyond network access. Nevertheless, for the internal accelerator networks we prefer to build our own Debian packages.

Style guides can help to get on board as a person unfamiliar with the respective code base. However, they are arbitrary in some way and may result in so-called "bike shedding" discussions, so we took *black*'s approach [4] and used a set of well-established linters, formatters and style guide checkers. Providing feedback on the formatting, linting and API documentation already at an early stage helps us to maintain readable and documented code. Good and common tooling provides automatic formatting. Code review of consistently formatted code with documentation is far more enjoyable and lets one focus on the logic rather than the style.

We try to have a high test coverage to ensure reliable productive code and to avoid re-introducing bugs, which have already been fixed in the past. Furthermore, tests can provide examples of the intended use, if no example code exist. We use *pytest* [5] with the *coverage* plug-in for testing as it makes testing more convenient than the build-in *unittests* and might nudge students, scientists and other users into writing tests for their personal code.

We host the packages and documentation on a website as they are the interface for most of the end users, who use the libraries but do not develop them. We plan to integrate this functionality into our GitLab setup.

DECISIONS

Python Packages

Nowadays, Python is the eco-system of choice both for (data-)scientists and for many software developers. The scientific libraries eco-system is extensive, thanks to its proper C-API. As it is open source and provides multiple programming paradigms, including object orientation, it is very popular among software developers. Hence Python is a good language to wrap existing APIs to control and archiving systems. We developed two kinds of packages: data aggregation wrappers and helper libraries. The first one is for

- archived data,
- electronic lab notebooks,
- live data,
- our settings database, and
- for general data formats and access of different kinds.

They wrap the different data sources and provide coherent and simple interfaces with sane default settings for our facility.

The second one, the helper libraries take care of the interaction between the other libraries and selection of the correct settings for the accelerator of choice.

Besides the Python packages, we provide Debian packages for the in-use Ubuntu LTS systems. These systems are part of our accelerator network and thus are well defined, have no connection to the internet, and no packages are meant to be installed by local users, consequently *pip* is not installed. That is why, we build our Python packages with all their dependencies that are not part of existing Ubuntu repositories.

Development Workflow

Changes to the tooling are made in a central management repository, and an automated workflow takes care of synchronising these changes across all packages. Each project can be separated into a tooling part, which is very similar for each project, and the actual code part. For creating a new package, we have got an extension for *PyScaffold*, which takes care of setting up the tooling part. In the case the tooling is adjusted, a central management repository automatically takes care of keeping it synchronised across the packages.

For existing projects, people can contribute in various ways, depending on their experience and the effort they want to invest; filing bugs or requesting features in the issue tracker, fixing small bugs via a web-UI or working on a locally cloned project. In the latter case, it is advised to install *pre-commit* [6]. This tool activates a set of Git pre-commit hooks that enforce our style guide. The pre-commit hooks run on the client side and already fix the code, or at least warn the developer about issues, which would lead to a failure in the continuous integration test pipeline.

On the server side we use GitLab, which is hosted on premise with access to our internal data sources, so we can run tests that can rely on these internal APIs. GitLab provides an issue tracker, continuous integration (CI) infras-

ISBN: 978-3-95450-237-0 ISSN: 2673-5512 doi:10.18429/JACoW-PCaPAC2022-THPP9

tructure, a Docker registry, a Python repository, a proper web-editor for small changes and previewing Markdown or ReStructuredText, and might be used for serving documentation and Debian packages in the future. This ensures that both, users and maintainers, only have to deal with a single platform.

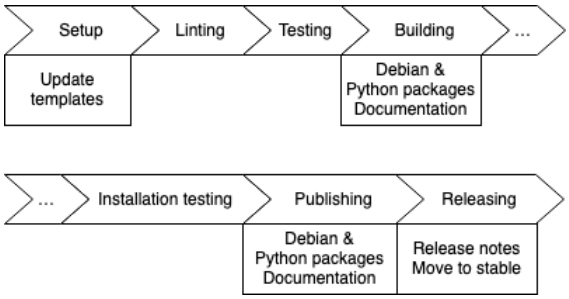

Figure 1: GitLab continuous integration. Besides the usual linting, testing, building, releasing, there is a setup, installation testing and publishing step. Besides some steps act on Python and Debian packages as well as documentation.

For the continuous integration we make use of custom Docker containers with pre-installed environments for the individual steps of the pipeline. The containers are managed in a dedicated project alongside the actual packages. Figure 1 shows the workflow. In the setup step we take care of updating the configuration and the template files of the project. As it is done for every new merge request, the different projects do not diverge in their overall structure and style. In the linting and testing part, checks are done again to enforce that code on the main branch is as easy to understand as tooling can guarantee. The build step not only builds the Python packages, but also the Debian packages and artifacts such as the documentation in HTML. In that way the code reviewer can study such tangible by-products and artifacts. As we build Debian packages, which might have dependencies on further packages, we have a dedicated "installation testing" step as part of the pipeline. During that stage the Debian packages are installed and tested as done during an update and as done in a green field, to ensure that conflicting dependencies do not disrupt or stop the processes on the production computers in either of these scenarios. The last step of the merge pipeline is to publish the packages and documentation to our internal repositories.
We try to create merge requests and code reviews for all of our code, so that more people are familiar with the code base. Best practices are taught and releases of malfunctioning code are minimised.

The release step is a dedicated manual step including a major, minor or patch release decision. Compiling the release notes, tagging the commits and moving the repositories to the stable branch is then done automatically. As an additional security measure, our control-system computers use an Apt repository that does not automatically receive newly published packages. Instead, a separate process has to be triggered manually to update the packages in that repository. This completes the release cycle.

LESSONS LEARNED

It turned out to be good to have one single source of truth for templates, configuration files, etc. for our tools and to invest some time into automation. Thereby, files can be automatically updated in all projects when a new merge requests is opened, so that new conventions can easily be propagated. Also having a dedicated testing project turned out to be helpful to test changes of the CI configurations.

Firstly, building own Docker images for the CI saves requests to Docker Hub and, makes a big difference in CI pipeline run time. Secondly, the images come with the correct versions of the checking tools, which have to be kept synchronised with the local checks (*pre-commit*) to avoid frustration of code contributers and error tracking in the CI pipelines. Therefore, version pinning of the tools is necessary and can also be checked automatically.

Though we only enforce API documentation, investing additional time in writing easy accessible examples and "how-to-use" documentation increases the user-friendliness and adoption rate, eventually leading to a re-usable and sustainable code base.

Last but not least, it makes sense to automate the release process to enable new releases often, especially when new projects with many new features and more mature packages with minor fixes co-exist.

SUMMARY

We developed a set of Python packages to provide consistent interfaces to the KIT accelerators and facility infrastructure, making it useful for all levels of users—may it be for student projects or expert tasks in the control system itself. The libraries are developed and maintained by using a modern software development cycle in form of continuous integration with automated code checks, tests, and package releases. This makes the code maintainable, extendable and easy to use for all stakeholders.
We adopted GitLab, because it provides a single platform of this complex setup without the need to manage and maintain many individual solutions.

ACKNOWLEDGEMENTS

We thank E. Bründermann proof reading and value comments and recommendations.

REFERENCES

[1] M. J. Nasse *et al.*, "FLUTE: A versatile linac-based THz source", *Rev. Sci. Instrum.*, vol. 84, p. 022705, 2013. doi:10.1063/1.4790431

[2] A. Papash, E. Bründermann, and A.-S. Müller, "An Optimized Lattice for a very Large Acceptance Compact Storage Ring", in *Proc. IPAC'17*, Copenhagen, Denmark, May 2017, pp. 1402–1405. doi:10.18429/JACoW-IPAC2017-TUPAB037

[3] SoftIOC, https://github.com/dls-controls/pythonSoftIOC

[4] black https://github.com/psf/black

[5] *pytest* https://pytest.org

[6] *pre-commit* https://pre-commit.com