TOWARDS A FLEXIBLE AND SECURE PYTHON PACKAGE REPOSITORY SERVICE

I. Sinkarenko[†], P. Elson, F. Iannaccone, W. Koorn, B. Copy, CERN, Geneva, Switzerland

Abstract

The use of third-party and internal software packages has become a crucial part of modern software development. Not only does it enable faster development, but it also facilitates sharing of common components, which is often necessary for ensuring correctness and robustness of developed software. To enable this workflow, a package repository is needed to store internal packages and provide a proxy to third-party repository services. This is particularly important for systems that operate in constrained networks, as is common for accelerator control systems.

Despite its benefits, installing arbitrary software from a third-party package repository poses security and operational risks. Therefore, it is crucial to implement effective security measures, such as usage logging, package moderation and security scanning. However, experience at CERN has shown off-the-shelf tools for running a flexible repository service for Python packages not to be satisfactory. For instance, the dependency confusion attack first published in 2021 has still not been fully addressed by the main open-source repository services.

An in-house development was conducted to address this, using a modular approach to building a Python package repository that enables the creation of a powerful and security-friendly repository service using small components. This paper describes the components that exist, demonstrates their capabilities within CERN and discusses future plans. The solution is not CERN-specific and is likely to be relevant to other institutes facing comparable challenges.

INTRODUCTION

In recent years, the use of Python within CERN's accelerator control system has seen significant growth. The adoption of Python as a supported language beside Java triggered the rise of new software, spanning from operational and expert GUIs based on PyQt [1], high level control system APIs, offline data analysis based on PySpark, to more recent online optimisation of operations using Machine Learning [2].

Now with a community of over 500 users, it is essential to provide effective support and ensure a stable and smooth user experience, based on practices and tools discussed in [3]. Such a service is undertaken by a dedicated centralised team, where small efficiency improvements can have a significant cumulative effect due to the large number of beneficiaries. One of the encouraged practices is for users to develop Python packages, rather than scripts, (for the sake of versioning, testability, and reuse), and the wide use of virtual environments (to avoid dependency collisions), both of which stimulate frequent installation and publishing of Python packages to a package repository. For this reason, the "Acc-Py Package Repository" is one of the most crucial services maintained by the team.

As an organisation with mission-critical, expensive, and sensitive hardware, CERN operates distinct networks separated physically and by firewall. A general-purpose network with Internet access supports everyday use, and a more restricted network without external access connects accelerator-related hardware. There is a need to install Python packages in both networks, meaning that the Acc-Py Package Repository must provide access to third-party packages from the Python Package Index (PyPI) at pypi.org.

The use-cases of CERN's accelerator control system are surely not unique, and similar scenarios probably exist in other laboratories, and in wider industry. Therefore, this paper aims to share the ideas and implemented solutions and invites readers to contact the authors, especially in the case of an interest to contribute to further developments under an open-source license. The functionality has been presented at the Europython conference [4], which confirmed the assumption of shared challenges, and initial versions of the software have been published in [5].

ACC-PY PACKAGE REPOSITORY

The Acc-Py Python Package Repository runs as a small set of microservices based on the FastAPI web framework. The most important ones are "simple-repository" that provides the index of packages for a package installer to consider and retrieve, and "simple-repository-upload" responsible for receiving uploaded in-house packages. A Web User Interface (Web UI) or "simple-repository-browser" provides a familiar user interface to discover packages and their metadata.

Many requirements for the API microservices are driven by the client-side de-facto tools of the Python Packaging community, namely pip as the package installer, and twine as the package uploader. In turn, this implies wheels as the standard binary package format, which is the preferred installation format, with source distributions, sdists, for other cases and fallbacks. Quality of service is continuously being improved by following the global evolution of Python packaging standards developed through the Python Enhancement Proposal (PEP) process, in a way that is compatible with both current and future versions of these client-side tools.

Despite there being a single endpoint for package downloads, internally "simple-repository" is a representation of two distinct repositories: a local one for in-house packages, and the public PyPI repository. The *pip* installer is pre-configured to communicate directly with the Acc-Py Package Repository service instead of attempting to contact PyPI, as explained in [3]. On the other hand, *twine* does not receive such automatic configuration because information, such as username and password, is different for each user. Instead, guidance is offered to use Continuous Integration (via Gitlab CI) with build-test-publish pipelines: The publishing job configures and uses *twine*. Within the pipeline, the Gitlab Job Token is encoded into the password sent by *twine*, and the "simple-repository-upload" then verifies the authentication via the dedicated Gitlab API.

The microservices implementation is simplified due to the creation of reusable components that represent the abstraction of an upstream repository. This allows components to be chained together in the form of a directed acyclic graph, which progressively enhances repository data or performs operations such as caching or logging. The result is a repository that conforms to the minimal interface understood by all clients, also known as the Simple Repository API, defined in PEP-503 [6]. This least common denominator can be progressively enhanced for clients that support newer features. For instance, it can support the newer JSON-based communication protocol (PEP-691 [7]), as opposed to a more verbose and harder to parse HTML-based format. The same component architecture allowed extracting and serving metadata files separately from the package itself (PEP-658 [8]). The relative simplicity of the architecture allowed the introduction of this functionality in the "simple-repository" even before it had been implemented in PyPI. Encapsulation of well-defined functionality in a small, testable upstream repository abstraction component also permits elegant reusability. For instance, the aforementioned component to extract metadata can be used in both "simple-repository" and the Web UI.

The initial strategy for the Acc-Py Package Repository service was to take an off-the-shelf solution to standardise and simplify the operation as much as possible. Sonatype Nexus [9] was chosen as the best candidate at the time, following an evaluation alongside JFrog Artifactory [10] and devpi [11] projects. Over time, and as functional limitations and severe security shortcomings became apparent, small web services were put in front of the Nexus instance to work around those issues. This gradually constrained the use of Nexus to providing storage for in-house packages and a local cache of packages used from PyPI. Recently, the remaining functionality was introduced directly into the "simple-repository", thus eliminating the need to run Nexus and offering a more maintainable solution for a small Python team to manage.

SECURITY

Security is especially important when running in a mission critical environment such as an accelerator control system. The CERN operating model requires accessing third-party packages from PyPI within the restricted network. Several options exist to mitigate security risks posed by the installation of malign external software, including: the potential to manually curate an "allow-list" of packages which may be installed; automatic "deny-listing" packages based on code scanning services; or snapshotting the external repository to a specific instant to mitigate exposure to zero-day attacks. No matter the policies implemented in the future, strengthening the service's monitoring capabilities, as well as introducing additional mechanisms such as "yanking" (PEP-592 [12]), are seen as valuable enhancements. This chapter discusses the security-related mechanisms in place so far.

Dependency Confusion

In 2021, a study [13] demonstrated the vulnerability of companies to an exploit known as "dependency confusion". This is a type of supply chain attack, whereby "a software installer script is tricked into pulling a malicious code file from a public repository instead of the intended file with the same name from an internal repository" [14]. Thus, depending on both public and private repositories, poses a security risk.

In Python, this vulnerability could be exploited in multiple ways. It could install a package that runs malicious code at runtime, but it is also possible to trick "pip" to install a source distribution instead of a binary package and execute malicious code at installation time, inside setup.py. All that is needed for such a malicious package to be delivered is to upload a package to PyPI with the same name as one found in an internal repository, ensuring that the version of the malicious package is greater than that of the internal one.

In response, Microsoft released a whitepaper [15] that suggests reducing upstream options to a single private feed as a mitigation option. More recently, the Python packaging community has started searching for ways to mitigate the problem, materialised in PEP-708 [16]. While it prevents dependency confusion, it does so by removing the ability to host in-house packages without the risk of "pip" denying the download if the same package name appears on PyPI at some later point. It is believed that the solution for scenarios requiring the installer to make the decision, involves a proper name-spacing mechanism, such as groupId in Maven or scopes in NPM. To date, Nexus and possibly other repository implementations remain vulnerable, and the issue reported to Nexus was closed in 2020 with a "won't-fix" resolution status.

"simple-repository", which started as a Nexus façade, implements [15]'s recommendation by merging public and private repositories on the server side, removing the decision from the installer client. The defined merging rules can be summarised as: prefer in-house packages over public packages, no matter their version.

Such rules introduce a reverse of the vulnerability, as it is now possible to upload an in-house package with a publicly reserved name, e.g., *numpy*, and break everything that relies on it. This is seen more as lack of fool-proofness, rather than lack of protection against malicious intent. Nevertheless, the solution is a façade for the package upload, "simple-repository-upload", that can verify whether the name of the uploaded package collides with the public namespace and, if necessary, reject the package with a clear message outlining the reason. If such a name has already been used internally, this collision is permitted to allow uploading new versions of the in-house package.

"Yank" Support

As explained in PEP-592 [12], yanking allows deletion of a version of a package distribution for all, except those who have pinned their dependency to that specific version. This is an elegant, non-invasive mechanism for reverting releases in case of a critical problem or vulnerability discovered post-factum.

Nexus support for mirroring the "yanked" attribute from remote repositories arrived at the end of 2022, much later than when it became standard practice in PyPI. In addition, "yanking" locally hosted packages is still not possible. Both facts led to the implementation of a workaround using existing helper services.

The custom repository entry point brings added flexibility to also support additional "yanks" for public packages, should the need arise, e.g., as a rapid response to a new vulnerability or incompatibility specific to the accelerator control system. This information is stored in a database on the server, and the same mechanism may be used for "yanking" in-house packages.

Authentication and Authorization

A key feature for enterprise environments is custom authorization, bound to the organisation's authentication service, typically Single Sign-On (SSO). CERN widely uses SSO, integrated with an internal Role-Based Access Control (RBAC) implementation [17] to manage authorization. RBAC features fall-back authentication for the accelerator-specific network and integrates with E-groups [18] to facilitate authorization across user groups.

While implementing authorization may be possible with Nexus using its groovy-based plugin system, it is costly for Python developers, and long-term support for the plugin system is not assured. Initially, a weak authorization mechanism was used for a small number of registered users. This was superseded by a Gitlab API-based mechanism linked to the recommended use of CI/CD as the primary means for package upload. Having "simple-repository-upload" in place has opened the door to implement custom authorization mechanisms, with the Gitlab API as the first candidate.

Package ownership validation is also being worked on, thus introducing per-package authorization into the service for the first time. Such ownership will likely be based on a "first to claim the name" basis, as is the case on PyPI, with the ability to manage additional users and e-groups. Although improving security, this introduces a possibility for a package to become orphaned, when the responsible changes role or leaves the organization, and is thus a topic for future work.

Telemetry

Besides supporting security-related investigations, good tracing brings the ability to analyse service usage to plan further improvements.

Telemetry in the CERN accelerator control system is based on the common ELK stack (Elasticsearch (OpenSearch in practice), Logstash, Kibana), shared across multiple services. Tracing download and upload events and correlating them with access and authentication helps to build an image of a sequence of events, while Kibana dashboards provide visual analytics capabilities. Rotated log files are also stored on the server for redundancy. It is expected that events being traced, and the composition of dashboards will continue evolving with experience and best practices.

WEB UI

The Web UI provided by the "simple-repository-browser" software is the common entry point for users to find packages. It has been built such that it can be used to display information from any Python package repository that adheres to the Simple Repository API [6], and may therefore be interesting to the reader even if core package repository needs are met by other services. The UI plays an important role in software discoverability and creates links to supporting documentation and source code.

The Web UI service crawls and indexes the in-house repository, as well as common packages from PyPI, and offers a powerful search engine for package discovery. The search results UI clearly indicates in which repository a package has been found, allowing easy discovery of internal software that may be reused.

For projects not previously indexed by the Web UI, metadata is fetched and cached on demand. As a result, the project details UI can render asynchronously. First, the page header is rendered, then the package metadata download is attempted from the upstream repository; if metadata is unavailable, the remote package is downloaded and unpacked to retrieve core metadata; metadata is then cached, and finally the page UI is replaced with the complete package information. The reusability of the repository abstraction components is useful here, sharing the metadata extraction implementation with "simple-repository". In this way, the Web UI can leverage PEP-658 [8] metadata served by an upstream repository, such as PyPI, but can fall back to generating metadata on the fly in case it is presenting results from a repository that follows the Simple Repository API [6] standard but which doesn't provide PEP-658 metadata.

To improve user experience, within the trademark constraints of the Python Software Foundation, the PyPI look and feel has been replicated to provide an interface familiar to every Python developer. Small modifications have been made to display additional information, for instance, the source repository of the package, whether it is an in-house, or a public one. The list of package dependencies is also displayed in the Web UI, something not currently available on PyPI, but common in other places, such as npmjs.org. The dependency information has already proven its utility on several occasions when identifying issues related to breaking changes that sometimes get introduced by minor updates of underlying libraries.

PERFORMANCE

With a daily average of 30,000 package downloads (including Continuous Integration (CI) pipelines), the package index can quickly run into performance bottlenecks. This is especially noticeable with the rise of Machine Learning (ML) applications since packages such as PyTorch and TensorFlow measure close to a gigabyte in size and frequent downloads are costly. This chapter discusses ways to optimise the performance.

Cache

In a standard interaction with the Acc-Py Python Package Repository, it is common to perform multiple requests to the upstream remote repository, PyPI, to first list the available versions of a package, then fetch the metadata for these versions, and finally download the desired package distribution. This introduces potential delays that can be eliminated by caching; however, cache must be correctly invalidated to avoid hiding recent updates to the original source. The best approach for Acc-Py Package Repository is to benefit from the local cache of clients, such as "pip", that relies on ETags [19]. It is necessary for the service to correctly forward the ETag header between the client and the remote repository. There is also a place for an intermediate cache on the server, to avoid downloading heavy packages when "pip" does not contain any local cache, such as in a fresh CI job. Likewise, care is needed to properly manage its lifetime and properly refresh it whenever an expired ETag is detected.

Currently, among cached artifacts are package distributions (both wheels and sdists), project index responses of PyPI, project details responses of PyPI and extracted PEP-658 metadata.

Metadata Extraction

The original design of the Simple Repository API [6] was indeed simple, sometimes at the cost of efficiency. One example is related to metadata. Core metadata, including package dependencies, is stored in a file inside a packaged distribution [20] and previously there was no mechanism to expose it elsewhere. As a result, while resolving dependencies on the client side, "pip" would need to download the actual package together with its dependencies, sometimes several versions of it, to reconstruct the dependency graph. The answer to the problem was proposed as PEP-658 [8] mentioned multiple times in this paper, which suggests hosting extracted metadata alongside package distributions. This way, "pip" needs to download only lightweight text files instead of full packages to resolve dependencies.

This PEP faced a lot of debate, which delayed its appearance in PyPI. Fortunately, "pip" introduced its support earlier, and it allowed the Acc-Py Package Repository to roll out an on-the-fly metadata extraction that immediately improved user dependency resolution times and reduced bandwidth needs on the server.

FUTURE WORK

The Python Packaging community appears to have found renewed vigour in recent years, with new improvements being proposed frequently, aided by the appearance of PEP-691 [7] that paves the way for future-proof versioning of the communication protocol and content negotiation. What follows covers several proposed future improvements to the Acc-Py Package Repository service, some of which are logical for any package repository, while others add value specifically for CERN's control system environment.

Robustness Improvements

Thanks to server-side cache, it is possible to minimize the need to download packages from PyPI. Additionally, the same mechanism, when reinforced with simple logic, can partially protect the service from the outage of the remote repository, in cases where all required resources have already been cached.

Security Improvements

Good security must be layered, and there is always room for improvement. While more distant goals due to their readiness, these topics should be foreseen.

Vulnerability scanning would be a clear improvement, with dependency scanning for newly uploaded packages, as well as repeated, scheduled dependency scanning of previously uploaded packages. DTrack has been evaluated [21] as an open-source solution and is a candidate for potential integration in the future. Upgrading the Web UI to present known vulnerabilities (e.g., from DTrack or an advisory database), when browsing a package would ensure that known bad software is not unwittingly installed.

Enforcement of code-signing for locally hosted packages would be another step. While currently not widely adopted in public repositories, the technology already exists to validate signed packages in pip, and with a simple user workflow would offer an additional layer of security against man-in-the-middle attacks.

Improvements to Package Authoring

While package discovery functions of PyPI have been met by the current state of the Web UI, authors of in-house packages do not have an interface to manage their products. So far, it has been manageable to perform those actions at the author's request via support channels, but as the number of packages and authors grows, there will be a need to provide an admin UI for managing "yank", setting package ownership and removal of obsolete packages in the future.

As mentioned previously, introducing strict authorization and ownership rules may lead to orphaned packages when maintainers change role or leave the organization. Reassigning rights afterwards requires additional support. This can be automated by integrating a service [18] to re-assign rights to the person's supervisor ahead of their departure.

Scalability

So far, it has been sufficient to run the service on a dedicated bare-metal machine. With time, the number of users and packages is growing and just a dozen of simultaneous CI jobs, each triggering a download of 20 packages, can create a visible load on the service. A prototype implementation to deploy the service using Kubernetes is proposed, as a paradigm that promises scalability under high load, and offers zero-downtime upgrades, disaster recovery and more.

SUMMARY

Package access is a common need in many enterprise environments, and while there are off-the-shelf solutions for the Python ecosystem, they do not always suffice or are not able to keep up with rapidly evolving Python packaging standards.

This paper presented the solution to the evolving needs of the Acc-Py Package Repository in the constrained yet growing environment of the CERN accelerator control system. The solution started as an off-the-shelf product and over time has transformed into a modular, security-oriented service that solves crucial long-standing problems not easily addressable by generic solutions.

The initial prototype has been published in [5] under an MIT licence, and it is hoped that it will trigger interest from other parties that have similar operational needs. With sufficient collaborative interest, there is the potential for the project to be openly developed, and to power Python package repositories across many domains.

REFERENCES

[1] I. Sinkarenko, S. Zanzottera, and V. Baggiolini, “Our Journey from Java to PyQt and Web for CERN Accelerator Control GUIs”, in *Proc. ICALEPCS'19*, New York, NY, USA, Oct. 2019, pp. 808. doi:10.18429/JACoW-ICALEPCS2019-TUCPR03

[2] M. Schenk *et al.*, “Machine learning & optimisation in particle accelerator operation for CERN”, https://indico.cern.ch/event/1145124/contributions/4948834/, 2022.

[3] P. J. Elson, C. Baldi, and I. Sinkarenko, “Introducing Python as a Supported Language for Accelerator Controls at CERN”, in *Proc. ICALEPCS'21*, Shanghai, China, Oct. 2021, pp. 236-241. doi:10.18429/JACoW-ICALEPCS2021-MOPV040

[4] P. J. Elson and I. Sinkarenko, “The Python package repository accelerating software development at CERN,” in *Conf. Europython 2023*, Prague, Czechia, Jul. 2023.

[5] Group of source code projects for the simple-repository, https://github.com/simple-repository/.

[6] PEP 503 – Simple Repository API, https://peps.python.org/pep-0503/

[7] PEP 691 – JSON-based Simple API for Python Package Indexes, https://peps.python.org/pep-0691/

[8] PEP 658 – Serve Distribution Metadata in the Simple Repository API, https://peps.python.org/pep-0658/

[9] Sonatype Nexus Repository product home, https://www.sonatype.com/products/sonatype-nexus-repository/

[10] JFrog Artifactory product home, https://jfrog.com/artifactory/

[11] Devpi source code, https://github.com/devpi/devpi/.

[12] PEP 592 – Adding “Yank” Support to the Simple API, https://peps.python.org/pep-0592/

[13] Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies, https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610/

[14] What is a dependency confusion attack?, https://secureteam.co.uk/2021/02/24/what-is-a-dependency-confusion-attack/

[15] 3 ways to mitigate risk when using private package feeds, https://aka.ms/pkg-sec-wp.

[16] PEP 708 – Extending the Repository API to Mitigate Dependency Confusion Attacks, https://peps.python.org/pep-0708/.

[17] P. Charrue *et al.*, “Role-Based Access Control for the Accelerator Control System at CERN”, in *Proc. ICALEPCS'07*, Oak Ridge, TN, USA, Oct. 2007, paper TPPA04, pp. 90-92.

[18] A. Corman *et al.*, “CERN’s Identity and Access Management: A journey to Open Source”, in *Proc. CHEP'19*, Adelaide, Australia, Nov. 2019, pp. 03012. doi:10.1051/epjconf/202024503012

[19] ETag - HTTP header explained, https://http.dev/etag

[20] PEP 241 – Metadata for Python Software Packages, https://peps.python.org/pep-0241/

[21] B. Copy *et al.*, “Protecting Your Controls Infrastructure Supply Chain”, presented at ICALEPCS'23, Cape Town, South Africa, Oct. 2019, paper MO4BCO03, this conference.