elAPI: A Powerful, Extensible API Client for eLabFTW

Mahadi Xion [1], Alexander Haller [1], Philipp Kling [1], Martin Baumann [1], Nick Kepper [1]

[1] University Computing Centre, Heidelberg University, Heidelberg, Germany

The adoption of electronic lab notebooks (ELNs) has gained considerable traction, driven by the need for efficient data management, reproducibility, and automation, as well as the broader shift toward digitizing workflows in research environments. At Heidelberg University, the Research Data Unit[1] – a joint initiative of the University Library and the University Computing Centre – has introduced the ELN eLabFTW[2] as a general solution for the entire university[3]. To improve the management of our instance, we initially set out to develop a simple API script, which gradually evolved into elAPI – a full-fledged API client for eLabFTW. This paper outlines the motivation for developing elAPI, its architecture, use cases, and the impact it has had so far.

Keywords: Electronic Lab Notebook (ELN), eLabFTW, RDM-Tool, Automation, API

1 Introduction

Application Programming Interfaces (APIs) are essential for connecting software systems and automating data exchanges, improving efficiency across various domains. elAPI is a specialized API client developed to interact seamlessly with eLabFTW (CARPi, Minges, and Piel 2017), a widely used electronic lab notebook (ELN). ELNs play an important role in modern scientific research by allowing researchers to securely document experiments,

1 https://www.data.uni-heidelberg.de; *Visited on May 24, 2025.*
2 https://www.elabftw.net; *Visited on May 24, 2025.*
3 https://www.urz.uni-heidelberg.de/en/service-catalogue/software-and-applications/elabftw; *Visited on May 24, 2025.*

Published in: Vincent Heuveline, Philipp Kling, Florian Heuschkel, Sophie G. Habinger, and Cora F. Krömer (Hrsg.): E-Science-Tage 2025. Research Data Management: Challenges in a Changing World. Heidelberg: heiBOOKS, 2025. DOI: https://doi.org/10.11588/heibooks.1652.c23952 (CC BY-SA 4.0).

manage data, and ensure regulatory compliance. However, the complexity of laboratory workflows often necessitates advanced data management and automation capabilities. elAPI addresses these needs by simplifying interactions with eLabFTW, enabling users to automate key functions and efficiently manage experimental data.

2 Context and Motivation

In 2023, we initially set out to create a simple automated script to collect user statistics and generate invoices based on user data. At the time, only the official `elabapi-python`[4] library for eLabFTW was available, along with a few limited, single-purpose solutions from third parties or older projects that were based on the now-retired API v1[5].

The official Python library was, at the time, a relatively new, auto-generated client for the new API v2. It exhibited some inconsistencies with the actual API endpoints and, ultimately, we decided to use raw HTTP API calls via the popular HTTPX[6] library in our script. This decision, combined with a desire for abstraction, and better support for our specific automation needs, led to the development of a completely new API client for eLabFTW.

Once we realized that we had the opportunity to elevate our code into a complete API client with relatively little additional effort, we decided to take that chance and create an attractive solution for both researchers and administrators. Our goal was to develop a client that is easy to configure and use, yet flexible, with safe and sensible defaults.

2.1 Feature Overview and Comparison

The eLabFTW project first released their official API V2 client, `elabapi-python` in December, 2022[7], and elAPI was released almost a year later. elAPI, so far, is the only mature API client alternative to `elabapi-python`. While `elabapi-python` works well as a reference client, it has some caveats as automation scripts become increasingly advanced, creating a clear demand for a scalable, extensible, and robust framework. elAPI fills this gap while simplifying code development for both researchers and software developers. Although elAPI offers many additional features to streamline automation; here we will focus on the *structural* differences between elAPI and `elabapi-python`.

4 https://github.com/elabftw/elabapi-python; *Visited on May 24, 2025.*
5 See (a) https://gitlab.com/iam-cms/workflows/extra-nodes/elabapy-cli,(b) https://metacpan.org/pod/ELab::Client and (c) https://github.com/elabftw/elabftw/blob/6079d11f4c9c5695de8c7484f659dd4f26d25529/README.md#deprecated-projects-using-retired-api-v1; *Visited on May 24, 2025.*
6 https://www.python-httpx.org; *Visited on May 24, 2025.*
7 https://github.com/elabftw/elabapi-python/releases/tag/0.1.3; *Visited on May 28, 2025*

1. Simplicity: For small and simple automation tasks, writing a whole script can be cumbersome and counterproductive. elAPI is designed to be both a CLI tool and a library to enable high versatility. E.g., for a task like getting a list of experiments that have been assigned a certain status (e.g., “success”) can be done with elAPI by running the following single line command: `elapi get experiments --query '{"status": "<desired status ID>"}'`. The same task with `elabapi-python` would require at least a dozen lines of code, which include some boilerplate code for server authentication.

2. Scalability: `elabapi-python` uses the Python built-in `threading` module to handle asynchronous connections[8]. Multi-threading is fraught with perils (Smith 2017). elAPI handles asynchronous connections with `asyncio` and `async/await` (co-routines) via HTTPX, which provides better documentation and control over the connections.

3. Configuration management: Any script written with `elabapi-python` requires a number of boilerplate code to manage authentication with the eLab server. elAPI offers a local and global configuration management system along with strong validation methods where boilerplate server authentication information can be punted off. This makes the script written with elAPI clutter-free and allows the developer to focus on the core logic.

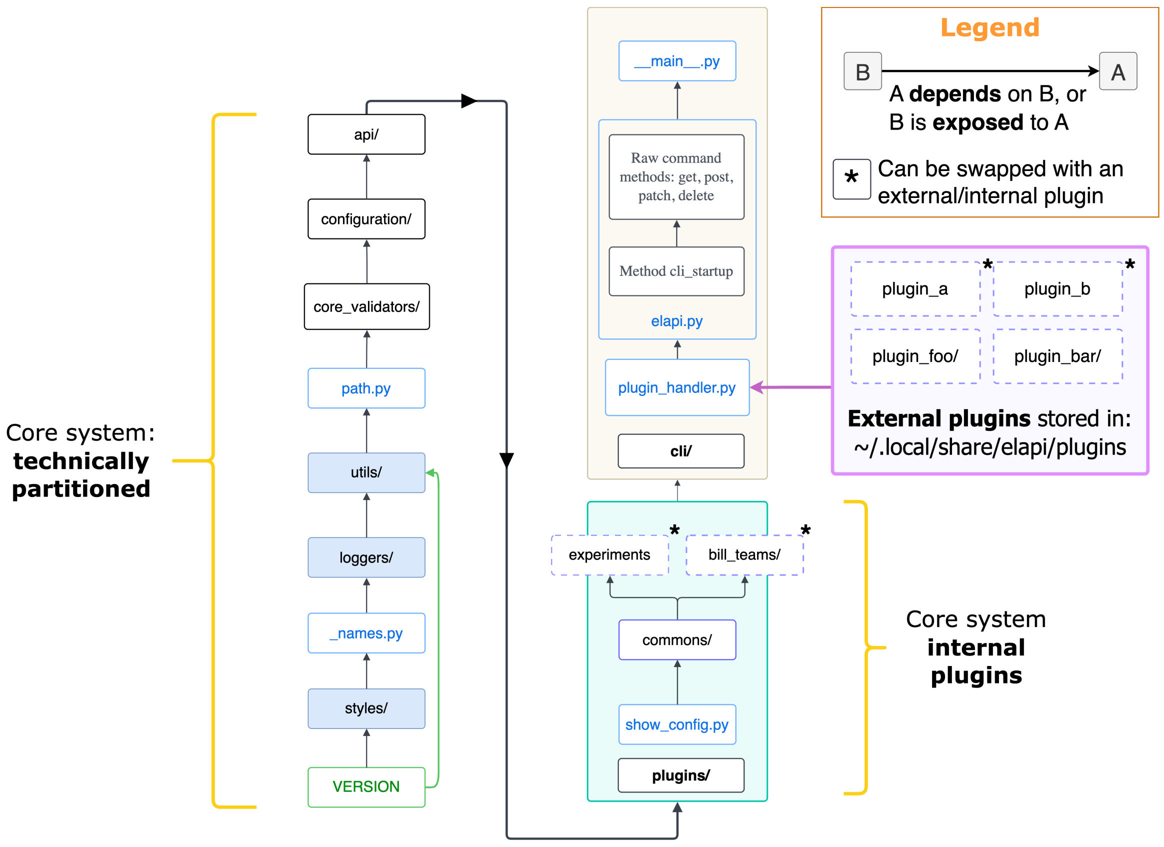
4. Extensibility: elAPI was designed to enable unification of various eLab automation solutions, hence a robust third-party plugin system has been built into elAPI. Reusing an `elabapi-python` script that was developed by a different institute involves a completely different setup and environment to be able to run the script, depending on the script’s requirements. elAPI’s third-party plugin system promotes a shared ecosystem of eLab automation solutions where various institutes can easily share, distribute, and manage each other’s solutions. Existing production-ready plugins support billing, data obfuscation, and user/team management.

3 Architecture and Design

We draw our inspiration from the microkernel architecture (Richards and Ford 2020; Thomas 2025) for the overall architecture principle and layered architecture for architecture partitioning. There was no strong rationale for choosing a microkernel architecture over alternatives such as a microservice architecture, other than the desire to start with something simple that still allows for extensibility. We intend elAPI to serve as the foundation for all eLabFTW automation solutions. In other words, eLabFTW users should

8 HTTP APIs can be found in `api_client.py`. Source: https://pypi.org/project/elabapi-python/5.2.0/#files; *Visited on July 4, 2025.*

be able to package their automation scripts as plugins, thereby extending elAPI's functionality. We refer to this architecture as the Simple Microkernel Architecture (SMA). In the very beginning, SMA was designed for elAPI only. Later we have improved and generalized it enough to be useful for any software design.



Legend
B
A
A depends on B, or B is exposed to A
*
Can be swapped with an external/internal plugin
Core system: technically partitioned
api/
configuration/
core_validators/
path.py
utils/
loggers/
_names.py
styles/
VERSION
__main__.py
Raw command methods: get, post, patch, delete
Method cli_startup
elapi.py
plugin_handler.py
cli/
plugin_a
plugin_b
plugin_foo/
plugin_bar/
External plugins stored in: ~/.local/share/elapi/plugins
experiments
bill_teams/
commons/
show_config.py
plugins/
Core system internal plugins

Figure 1: elAPI SMA architecture overview.

3.1 SMA Principles

SMA follows most of the principles found in microkernel architecture guidelines with a few additions.

1. A core system must provide basic functionalities: The core system principle is adopted straight from the textbook microkernel architecture principles. In SMA, the core system must provide the basic features. Here, features that would be considered as basic would depend on the software domain. At a minimum, the core system must enable the plugin system. The core system in elAPI offers interface style guides, logging functionality, helper functions, OS path handlers, basic validation abstractions, configuration manager, extended HTTPX API interfaces to communicate with eLabFTW endpoints, and plugin handlers that enable the entire plugin system. elAPI's core system can be extracted out and used as the foundation for the SMA implementation for any standalone program.

2. Interfaces can be flexible across internal plugins: Typically, a microkernel architecture separates plugins from the core system entirely. SMA modifies this approach by introducing internal plugins, which reside inside the core codebase, and external plugins, which can exist anywhere. Both types of plugins can share the same plugin *interface*. This is also the default with elAPI's SMA implementation. Depending on flexibility preference, however, having a unique interface or distinct interfaces for internal plugins is also allowed where a unique interface is enforced upon external plugins.

3. Internal plugins can be part of the core system: In SMA, internal plugins can be part of the core system such that they are meant to be used by external plugins in the same way core system functionalities are used by all plugins in a typical microkernel architecture. In elAPI, public APIs offered by the internal plugin *"experiments"* are meant to be leveraged by external plugins to make updates or advanced modifications to eLabFTW experiments, among other features. Note, if external plugins are disabled for a SMA implementation, then internal plugins should not be part of the core system.

4. Hybrid partitioning is allowed: SMA recommends technical partitioning for the basic functionality layers. If domain partitioning is desired, then it is recommended to apply domain partitioning to internal and external plugins. The combined partitioning helps enforce a design based on strict separation of concerns.

5. A strict unidirectional dependency must be maintained between technically partitioned layers: SMA imposes a dependency invariance between technically partitioned layers. In Figure 1, we see a top layer always "depends on" a bottom layer, or in other words, a bottom layer's APIs are always exposed to a top layer. This is denoted with an upward arrow. A layer that sits at the most bottom is an independent layer, as it doesn't depend on any other layer. The third-party libraries (typically whose code do not reside in the codebase) do no take part in the invariance, and are considered floating layers. In that sense, an independent layer is also a floating layer. In elAPI, the styles layer is an independent layer. styles layer designate user interface definitions and methods, e.g., the color scheme used by elAPI CLI. Styles being the independent layer indicates that all dependent layers must adhere to the same UI guidelines.

6. Plugins need not be independent: This principle is in line with the third principle. Microkernel architecture imposes that plugins must be independent, and for good reasons, i.e., to avoid “Big Ball of Mud” (Foote and Yoder 1999). SMA still lets the developer decide the rules internal and external plugins should follow. For example, external plugins can depend on internal plugins (as they are part of the core system). An external plugin vendor is free to decide that their plugin would depend on another external plugin.

4 Use Cases at Heidelberg University

1. Internal billing: The need for automation initially arose from the requirement of a billing system for our eLabFTW instance. This necessitated the collection of monthly user statistics, cost calculations, and the generation of invoices. These objectives have now been successfully achieved through the implementation of the internal elAPI plugin ‘bill-teams’[9] (see Figure 2).

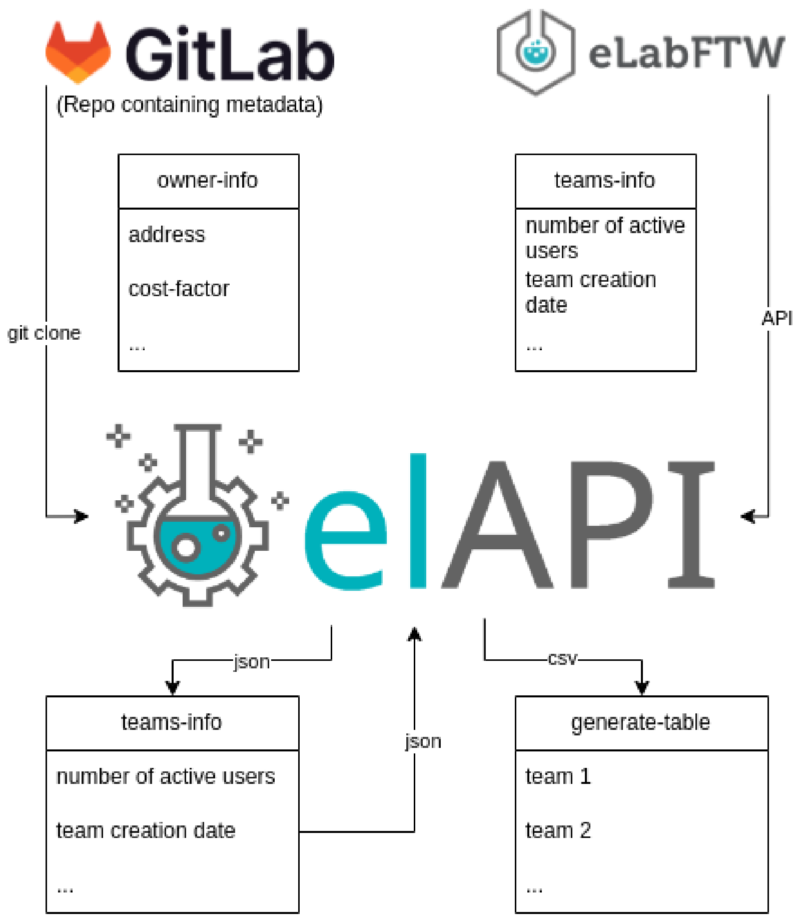


Figure 2: elAPI bill-teams workflow.

9 https://github.com/uhd-urz/elAPI/tree/ad1caad24c1cceaf9970c96093303a589118d351/src/elapi/plugins/bill_teams; *Visited on May 24, 2025.*

2. Stress testing: We utilized elAPI to generate thousands of users and hundreds of teams. Subsequently, we retrieved all relevant metadata and measured the responses to ensure that our instance can scale effectively for the foreseeable future.

3. Mass expiration of users in a team: We utilized a section of the elAPI code to create a custom script that efficiently expires all users for a given team.[10]

4. Rapid creation of plugins: An institute made an urgent request involving a fallback plan for the mass renaming of all experiments and database items across multiple teams. Using elAPI, we were able to quickly develop a custom plugin (called "Obfuscate")[11] in less than two days. This plugin saved the original title in extra fields, renamed the title, and allowed for later reversion if needed. Without a robust framework like elAPI, this task would have required significantly more time.

5. Use in teaching: In a course on research data management, students were tasked with planning and creating templates and experiments in eLabFTW. We used elAPI to demonstrate how we could later retrieve all the tags used by the team and identify frequently used terms.

5 Impact in the Community

elAPI is featured in the official eLabFTW GitHub repository[12] under the "Related Projects" section and was presented at the eLabFTW Community Meeting III[13]. We have received positive feedback from users; however, since we do not collect usage statistics, we are uncertain about the broader adoption of elAPI outside of Heidelberg University. Some usage statistics can be inferred from PyPI download counts (e.g., via pepy.tech[14]). However, these figures are often inflated by automated systems such as CI/CD pipelines and mirrors, and therefore do not accurately reflect the actual number of human users. Conversely, download statistics fail to account for other types of usage, such as users cloning the repository directly from GitHub or those who still continue to use older versions of the package.

10 https://github.com/uhd-urz/elAPI/blob/ad1caad24c1cceaf9970c96093303a589118d351/examples/python/manage_teams/expire_teams.py; *Visited on May 24, 2025.*

11 https://github.com/uhd-urz/elAPI/tree/ad1caad24c1cceaf9970c96093303a589118d351/examples/python/create_plugins/obfuscate; *Visited on May 24, 2025.*

12 https://github.com/elabftw/elabftw; *Visited on May 24, 2025.*

13 https://github.com/elabftw/elabftw/discussions/5152; *Visited on May 24, 2025.*

14 https://pepy.tech/projects/elAPI; *Visited on May 24, 2025.*

6 Limitations and Improvements

Some limitations arise from the general limitations of the eLabFTW API. eLabFTW currently does not provide unrestricted API-Level access to all actions and data. This makes it impractical to provide solutions like a global sync to repositories or archival systems that could be triggered by a sysadmin. While elAPI significantly enhances the user-friendliness of the API, it remains a CLI application and a Python library. This can pose a substantial barrier to entry for many users, especially those unfamiliar with command-line tools or programming. To improve accessibility, introducing a GUI/Web interface or enabling custom code hooks within eLabFTW would be valuable enhancements.

7 Conclusion and Outlook

elAPI has successfully addressed all our immediate needs far better than we initially anticipated. We now plan to gradually introduce new features as needed, primarily through the development of additional plugins. The first commands for the experiment-related plugins serve as an example of this approach.

For internal use cases, we have identified several potential areas for further development, including team management, automated metadata checks, and import/export functionality with heiARCHIVE[15]/heiDATA[16]. These avenues will be explored in future iterations of elAPI. We are advocating for the addition of new features in eLabFTW that would address or alleviate the limitations mentioned previously. To foster reuse and collaboration, we offer to feature community-provided plugins in our GitHub repository[17], serving both as practical tools and as examples for other developers. Additionally, we welcome contributions of general-purpose plugins, which could then be included in future releases of elAPI. We are exploring ways to link or feature elAPI and its plugins, along with examples and other related materials such as templates, in a community repository for ELN/eLabFTW resources, such as the ELN material collection[18]. A long-term goal may be to generalize the architecture of elAPI into a broader framework, with elAPI itself becoming a consumer of that framework.

15 https://heiarchive.uni-heidelberg.de; *Visited on May 24, 2025.*
16 https://heidata.uni-heidelberg.de; *Visited on May 24, 2025.*
17 https://github.com/uhd-urz/elAPI; *Visited on May 24, 2025.*
18 https://eln-resources.gitlab.io/collection; *Visited on May 24, 2025.*

Software Availability Statement

elAPI is freely available under the AGPLv3 license at the following repository: https://github.com/uhd-urz/elAPI[19]. Citation recommendation: Xion, Mahadi, and Alexander Haller. 2025. *elAPI*. Version V2. https://doi.org/10.11588/DATA/E4XHXZ.

Acknowledgments

The authors wish to thank the organizers of the E-Science-Tage conference in Heidelberg for their dedicated efforts, and our reviewers for their valuable and constructive feedback.

Authorship Contributions

- Mahadi Xion: Conceptualization, Original Draft, Review, Editing
- Alexander Haller: Conceptualization, Original Draft, Review, Editing
- Philipp Kling: Review
- Nick Kepper: Review
- Martin Baumann: Review

Conflict of Interest

Philipp Kling is leading the organization of the conference and, thus, is involved with the reviewing of contributions. Evaluation of this contribution was done anonymously by our reviewers and the final decision for the acceptance of this paper was in line with Cora F. Krömer and Sophie G. Habinger and the programme committee of the conference.

Bibliography

CARPi, Nicolas, Alexander Minges, and Matthieu Piel. 2017. “eLabFTW: An open source laboratory notebook for research labs”. *The Journal of Open Source Software* 2 (12): 146. https://doi.org/10.21105/joss.00146.

19 *Visited on July 4, 2025.*

Foote, Brian, and Joseph Yoder. 1999. “Big Ball of Mud”. In *Fourth Conference on Patterns Languages of Programs.* Visited on June 27, 2025. http://www.laputan.org/mud/.

Richards, Mark, and Neal Ford. 2020. *Fundamentals of Software Architecture: An Engineering Approach.* 432. Sebastopol, CA: O’Reilly Media. ISBN: 9781492043454.

Smith, Luke. 2017. *Python: what are the advantages of async over threads?* Visited on May 24, 2025. https://stackoverflow.com/a/48021834/7696241.

Thomas, Richard. 2025. *Microkernel Architecture.* Visited on March 31, 2025. https://csse6400.uqcloud.net/handouts/microkernel.pdf.