ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO4BC002

LESSONS FROM USING PYTHON GraphQL LIBRARIES TO DEVELOP AN EPICS PV SERVER FOR WEB UIs

Rebecca Auger-Williams*, Observatory Sciences Ltd, St Ives, UK
Abigail Alexander, Tom Cobb, Martin Gaughran, Austen Rose, Alexander Wells, Andrew Wilson
Diamond Light Source, Harwell, UK

Abstract

Diamond Light Source is currently developing a web-based EPICS control system User Interface (UI). This will replace the use of EDM and the Eclipse-based CS-Studio at Diamond, and it will integrate with future Acquisition and Analysis software. For interoperability, it will use the Phoebus BOB file format. The architecture consists of a back-end application using EPICS Python libraries to obtain PV data and the query language GraphQL to serve these data to a React-based front end. A prototype was made in 2021, and we are now doing further development from the prototype to meet the first use cases. Our current work focuses on the back-end application, Coniql, and for the query interface we have selected the Strawberry GraphQL implementation from the many GraphQL libraries available. We discuss the reasons for this decision, highlight the issues that arose with GraphQL, and outline our solutions. We also demonstrate how well these libraries perform within the context of the EPICS web UI requirements using a set of performance metrics. Finally, we provide a summary of our development plans.

INTRODUCTION

Diamond Light Source is about to undergo a significant upgrade as part of its Diamond II project, including new beamlines and other accelerator upgrades. Improvements to existing technologies are also being considered as part of this initiative, which stimulated an assessment of the control system UIs currently being used at Diamond. These are predominantly EDM [1] and CS-Studio (Eclipse) [2], the latter of which is now deprecated and has been replaced by Phoebus [2]. Two alternatives are being considered, either Phoebus or a web-based UI, both of which would require a significant amount of effort to move to. A web browser UI has many advantages—there is no installation required, it is truly cross-platform, and it offers the best experience for remote usage—and Diamond therefore developed a prototype version of a web-based UI. This front-end application is built with React [3], one of the most popular JavaScript libraries for building web applications due to its use of components that make it fast, scalable, and simple to use. Redux [4] is used for the data management, which helps maintain a global state across the application. The application itself is written in TypeScript.

A back-end Python application has also been created, named Coniql [5], which uses EPICS [6] Python libraries to access process variable (PV) data and a GraphQL [7] Python library to serve these data to the web UI via WebSockets [8]. A schematic of the application front-end and back-end is shown in Fig. 1.

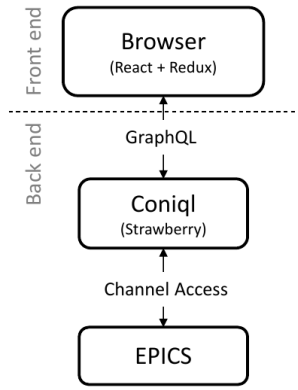

Figure 1: A schematic showing how the front-end web UI receives PV data from EPICS. The web UI uses a WebSocket to connect to Coniql. It uses the GraphQL query language to send requests for data following a defined GraphQL schema. Coniql uses the Python aioca library to subscribe to updates from the requested EPICS PV and returns these to the web UI in a GraphQL query response.

TECHNOLOGIES

This section outlines the technologies that Coniql uses to provide the back-end functionality. Figure 2 shows a diagram of how these fit together in order to supply the front-end web UI with EPICS PV data.

EPICS Python Library

The Python library aioca [9] is used as the EPICS channel access (CA) client to provide access to EPICS PVs running on IOCs. This application is built on top of asyncio [10] to allow asynchronous querying. The API supports three main function calls: `caget`, `caput`, and `camonitor`. These resemble the EPICS CA command line tools.

GraphQL

GraphQL is an open-source querying language and runtime engine that was originally developed by Facebook. It can be used to create fast and stable applications where the client can request only the data needed, with the results returned in a predictable format. This reduces the amount of data sent over the network and hence improves performance.

* rjw@observatorysciences.co.uk

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO4BC002

It supports reading, writing, and subscribing to data changes through queries, mutations, and subscriptions via WebSockets. The original GraphQL JavaScript implementation has been ported to Python3 in the graphQL-core [11] library, used by many higher-level libraries.

An example GraphQL subscription is shown in Fig. 3.

Strawberry GraphQL API

Of the many high-level Python GraphQL libraries available, we selected Strawberry [12] to build our back-end GraphQL server. Strawberry supports a code-first schema, where a schema is generated from a selection of resolver functions and types defined in code. This is in contrast to the schema-first approach where a schema is first defined and then resolver functions are added. The benefit of the code-first approach is that it makes the API much easier to maintain as changes can be incorporated directly into the code and the schema is dynamically created instead of having to maintain compatibility between the two. Strawberry also supports two WebSocket protocols, `graphql-ws` [13] and `graphql-transport-ws` [14], the former of which has now been deprecated. This was an important consideration when choosing a library to build the Coniql application on as we have now moved the web UI to use the new WebSocket protocol. Strawberry also supports both asynchronous and non-asynchronous resolvers, allowing flexibility throughout the code and making the integration with the aioca API, which uses async IO, straightforward. Strawberry has been inspired by dataclasses and uses type hints and Python decorators to offer a cleaner experience for developers.

The Strawberry API [15] is open source and uses GitHub [16] for version control. It is actively maintained with constant development and fast response times to issues and GitHub pull requests (PRs). It is a community-driven project that welcomes contributions and involvement. This is important, given that Strawberry is still in its development phase, as it ensures that we have support for any issues uncovered while developing Coniql. Strawberry also benefits from detailed documentation.

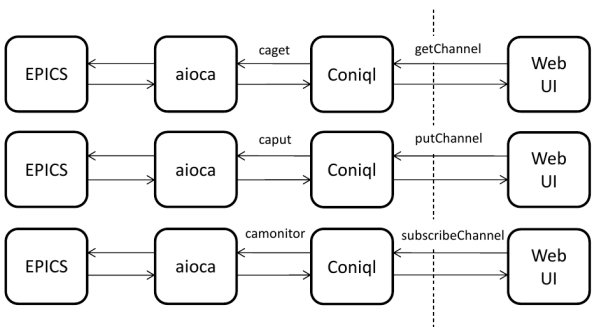

Figure 2: A diagram showing how the GraphQL functions relate to EPICS PV calls. Top shows the calls for a `getChannel` GraphQL query, which aims to get information back about a PV. Coniql calls the `caget` function in aioca, which uses the C libca library under the hood to get the EPICS PV data through CA. A similar process is used for `putChannel` and `subscribeChannel` GraphQL queries, shown middle and bottom, respectively.

```
subscription sub1() {
  subscribeChannel(id: pvname) {
    id
    time {
      datetime
    }
    value{
      float
    }
  }
}
```

Figure 3: A GraphQL subscription request sent by the web UI to Coniql. This example requests two fields: the `time` of the last PV update, formatted as a human readable `datetime`, and the `value` of that PV, formatted as a `float`. The Coniql GraphQL schema supports further fields, such as the status of the PV and display units. The client can define which fields to include in the query.

ISSUES AND SOLUTIONS

We began development of Coniql using a GraphQL library called Tartiflette [17], which is a schema-first GraphQL implementation built on top of Python asyncio. While running tests with our prototype web UI we came across a few issues with this library, including a suspected memory leak. In addition, we wanted to upgrade to use the latest WebSocket protocol because the older version had been deprecated. However, this was not possible at the time with Tartiflette. Furthermore, there seemed to be little recent activity on the GitHub repository, which led to doubts that we would be able to get a fix for the memory leak or that there would be sufficient future development to properly support the latest WebSocket protocol. Therefore, we searched for a new GraphQL API that was actively maintained that supported the new WebSocket protocol, which led us to Strawberry. This was followed by a large effort to refactor the Coniql application to use Strawberry. This highlights one of the risks of relying on third-party technologies that may not continue to be maintained in the future.

In the early stages of refactoring Coniql to use the Strawberry API we discovered a few issues and immediately made use of the active Strawberry community. The majority of these issues were discovered when testing the new WebSocket protocol (`graphql-transport-ws`) and highlighted bugs in the underlying Strawberry implementation. These bugs were likely present because few developers have refactored their client applications to use the new WebSocket and so this part of the Strawberry code had yet to be thoroughly tested. We initially discovered a problem with WebSocket connections not being properly closed down, and this resulted in a memory leak from connections that were left open and consuming resources until they were eventually

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO4BC002

Table 1: Table showing the results from running the performance tests with a single GraphQL client collecting 36,000 samples from *N* PVs that are updating at 10 Hz. The table shows how the averaged CPU usage and number of dropped results vary with the number of PVs subscribed to from the client. The second and third columns show the results from the first set of performance tests run on Coniql before any performance improvements were made. The results show that no updates are missed when the CPU is lower than the maximum of 100 % but this value dramatically increases as soon as the CPU is at its maximum, which occurs at ≈ 200 PV subscriptions from a single client. At 500 PV subscriptions we receive approximately one in four of the updates, dropping the other three. The fourth and fifth columns show the results after making performance improvements to Coniql (described in detail in the Performance Improvements subsection) and show a decrease in CPU usage when CPU is below 100 % as well as fewer dropped updates when it is near 100 %.

	Initial Performance		Performance after Improvements	
Number of PVs	Average CPU	Average number of dropped updates	Average CPU	Average number of dropped updates
10	20.52 %	0	13.69 %	0
50	53.32 %	0	34.53 %	0
100	74.55 %	0	54.91 %	0
200	100.00 %	3,811	92.60 %	16
500	100.00 %	101,829	100.00 %	66,929

cleaned up by the garbage collector. We brought this to the attention of the Strawberry community and received immediate support. We proposed a solution that was subsequently reviewed by the Strawberry code owners and promptly accepted and merged, triggering an immediate release of a new version.

Another issue we came across was related to the performance of GraphQL subscriptions using the new WebSocket protocol (`graphql-transport-ws`) when compared to using the old protocol (`graphql-ws`). We ran a set of performance tests on the Coniql application (see the following section for full details) and found that the CPU usage was consistently higher when using the new WebSocket protocol over the old. Again, this had likely not been detected previously due to slow adoption of the new WebSocket protocol by the broader community. Nevertheless, we were able to determine where there was a slow down in the underlying Strawberry code and proposed a solution, which, after some discussion, was accepted and merged.

There are downsides to adopting libraries so early in their development. New releases frequently break compatibility, which has resulted in several patches to our own code. Fortunately, extensive automated test suites and lint checks allow us to catch these issues promptly. To mitigate these issues, we pin the Strawberry dependency in our Coniql installation to use a specific version. Before updating to a newer version, we run our test suites and lint checks while also checking for any performance degradation. We expect the volatility of updates to diminish in the future as the library becomes more stable and more widely adopted.

In summary, we have found that the Strawberry GraphQL library provides all of the functionality we require and is straightforward to learn and integrate within applications. Our experience with the Strawberry GraphQL community has so far been very positive, with quick response times to issues and PRs allowing efficient development.

PERFORMANCE

The demands on the Coniql application are very high. Coniql must be able to serve thousands of PVs updating at rates of up to 10 Hz and must also be able to handle hundreds of client connections. In order to verify that Coniql would be able to achieve this we developed a set of performance tests to measure how well Coniql behaves under such conditions. Our performance tests aim to simulate the real end-to-end use case as closely as possible. To do this we use an EPICS IOC with a number of individual PVs incrementing by one at a rate of 10 Hz. The performance test itself runs in Python and creates a GraphQL client that connects to Coniql via a WebSocket and initiates a subscription request for each PV running in the EPICS IOC, the number of which is configurable in the performance test parameters. We run the test until we have received a defined number of updates (samples) from each subscribed PV and then analyse the collected results to determine how many updates have been missed, i.e. how many events Coniql did not send to the client. From this we can estimate the average number of updates missed per subscription. This scenario could occur if updates are coming in too quickly for Coniql to process and return and so they get dropped in favour of the latest update. Monitoring the number of dropped updates allows us to ensure that Coniql is processing all of the PV updates correctly. We also measure the CPU and memory usage of Coniql while all of the subscriptions are running and compute the average usage at the end of the test. We have also made the number of GraphQL clients configurable, allowing us to simulate multiple clients, each setting up a single WebSocket connection, and measure how the associated load affects Coniql.

After running an initial set of tests we discovered that the performance of Coniql was far from optimal. The second and third columns in Table 1 show a summary of the initial results for a single client. We find that we can comfortably support 100 subscriptions to PVs updating at 10 Hz with

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO4BC002

Table 2: Table showing the results from the performance test where the number of GraphQL clients is varied. Each client is collecting 36,000 subscription samples from *N* PVs that are updating at 10 Hz. The second and third columns show the results from tests run on Coniql before any performance improvements were made. The table shows how the average CPU usage varies with the number of clients and the number of PV subscriptions. The fourth and fifth columns show the results after making performance improvements to Coniql (described in detail in the Performance Improvements subsection) and demonstrate a ~30 % decrease in CPU usage.

		Initial Performance		Performance after Improvements	
Number of clients	Number of PVs	Average CPU	Average number of dropped results	Average CPU	Average number of dropped updates
1	100	74.55 %	0	54.91 %	0
2	50	73.70 %	0	51.88 %	0
10	10	75.47 %	0	51.73 %	0

zero dropped updates, however the CPU usage is already quite high. At 200 PV subscriptions we have reached 100 % CPU, the maximum CPU the application can use, and the number of dropped updates starts to increase.

Table 2 shows the results for the performance test runs where we increased the number of clients. For these tests we kept the total number of PV subscriptions constant and varied the number of separate clients. The results show that the CPU use is relatively constant (for a fixed total number of PVs) and the additional WebSocket connections that the Coniql GraphQL server has to respond to do not have a significant impact on performance.

Performance Improvements

We have made some initial attempts to improve the performance of Coniql. We used the sampling profiler py-spy [18] to identify where most of the time is being spent within the code. We found that most of the time spent processing results is done using `async` processing. As a result we have now minimised the number of `async` calls made within the code meaning that all of our resolver functions are now synchronous. This led to a 10 – 20% improvement in performance. We also improved the performance for two or more clients subscribing to the same PVs by ensuring Coniql only maintains a single EPICS `camonitor` for each PV instead of creating a new one for each client. When running 20 clients requesting the same PV information this led to a 6 – 8 % CPU improvement. The current main users of our Coniql instance are the machine status displays, which provide information on the current state and operating conditions of the machine. These all display similar information and therefore benefit from this improvement. The fourth and fifth columns of Table 1 and Table 2 show the results from the performance tests after making these updates. They show that CPU usage has decreased as has the number of dropped updates when nearing 100 %.

After these improvements approximately 1 % of the time is spent in the Coniql code, with the majority of time now spent in high-level Strawberry code that calls into the lower-level graphql-core library. This code first parses and validates the GraphQL query before executing it. Next, in the execution phase the executor calls the resolve functions, starting from the top level of the query, and then waits until all resolvers have returned a value before formatting and returning the result. It is not yet clear why this process should take so long and any improvements here may require collaboration with the graphql-core project.

We have significantly improved the performance of Coniql with the changes described above, yet the performance of a single instance of Coniql still cannot support multiple screens without dropping a large number of updates. However, we have been able to mitigate the performance issues using Kubernetes [19] to deploy the application. We currently deploy eight Coniql replicas to a Kubernetes cluster with load balancing. Each client establishes one WebSocket connection with one of the replicas, therefore the clients are load balanced in a sensible way. We have live monitors monitoring the Kubernetes pods to verify that all Coniql instances are running at less than 100 % CPU, meaning that they are functioning correctly and should not be dropping PV updates to connected web clients. Eight Coniql replicas are currently sufficient to serve PV data to the ≈ 50 machine status screens that have now been ported to the web UI application.

FUTURE PLANS

The performance issues of Coniql detailed in the previous section are a problem for Diamond going forward. It implies that we will need many more Coniql replicas deployed in Kubernetes to support the number of PV updates and the number of clients expected from a control system UI made up of machine status screens and operator screens across multiple beamlines. This would result in using a large amount of resources, which is not ideal. Whilst we have made significant improvements to the performance of Coniql within our code and some in the Strawberry library, we will investigate further changes including whether altering the shape of our subscription schema will improve the speed at which requests can be processed in the low level graphql-core code.

We will also consider alternatives to GraphQL if it cannot meet our requirements. One possible alternative is PV WebSocket (pvws) [20], which provides a Websocket for EPICS CA and PV access. However, we have yet to evaluate the

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO4BC002

performance of this or any other alternatives. Finally, we plan to add support for the EPICS PVAccess protocol (PVA) to Coniql.

CONCLUSION

We are working to improve and update our back-end Coniql application to make it suitable for our requirements to serve EPICS PV data to web UI clients. We have had a positive experience working with the Strawberry code owners and have managed to get issues fixed quickly, whilst also giving back to the community. However, Strawberry is still in early development and this can lead to issues with compatibility. We are also aware that, as with any third-party library, there is a risk that it may not continue to be maintained in the future thereby making any arising issues difficult to fix.

We have verified that Coniql has the potential to fulfill the requirements of a back-end to a web UI, however the performance of the application in terms of CPU usage and number of dropped PV updates is not as good as expected. At the moment it appears that we are constrained by the performance in the graphql-core library. We have been able to mitigate the issue with performance in production by deploying multiple instances of Coniql to Kubernetes, but there is a limit to how much we can scale in this way due to resources usage. Future work will go into further improving the performance of the Coniql application and also investigating other alternatives so that our application has the ability to support the future control system web based UIs.

REFERENCES

[1] EDM, https://www.slac.stanford.edu/grp/cd/soft/epics/extensions/edm/edm.html

[2] CS-Studio, https://controlsystemstudio.org

[3] React, https://react.dev

[4] Redux, https://redux.js.org/

[5] Coniql, https://github.com/DiamondLightSource/coniql

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

[7] GraphQL, https://graphql.org

[8] WebSockets, https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

[9] aioca PyPI, https://pypi.org/project/aioca

[10] asyncio PyPI, https://pypi.org/project/asyncio

[11] graphQL-core, https://pypi.org/project/graphql-core

[12] Strawberry GraphQL, httbps://strawberry.rocks

[13] graphql-ws library, https://www.npmjs.com/package/subscriptions-transport-ws

[14] grahql-transport-ws library, https://www.npmjs.com/package/graphql-ws

[15] Strawberry GraphQL GitHub, https://github.com/strawberry-graphql/strawberry

[16] GitHub, https://github.com/

[17] Tartiflette, https://tartiflette.io

[18] pyspy, https://pypi.org/project/py-spy

[19] Kubernetes, https://kubernetes.io

[20] pvws, https://github.com/ornl-epics/pvws