ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TH2A004

DEVELOPING MODERN HIGH-LEVEL CONTROLS APIS

B. Urbaniec, L. Burdzanowski, S. Gennaro CERN, Geneva, Switzerland

Abstract

The CERN Accelerator Controls are comprised of various high-level services that work together to provide a highly available, robust, and versatile means of controlling the Accelerator Complex. Each service includes an API (Application Programming Interface) which is used both for service-to-service interactions, as well as by end-user applications. These APIs need to support interactions from heterogeneous clients using a variety of programming languages including Java, Python, C++, or direct HTTP/REST calls. This presents several technical challenges, including aspects such as reliability, availability, and scalability. API usability is another important factor with accents on ease of access and minimizing the exposure to Controls domain complexity. At the same time, there is the requirement to efficiently and safely cater for the inevitable need to evolve the APIs over time. This paper describes concrete technical and design solutions addressing these challenges, based on experience gathered over numerous years. To further support this, the paper presents examples of real-life telemetry data focused on latency and throughput, along with the corresponding analysis. The paper also describes on-going and future API development.

INTRODUCTION

The CERN Accelerator Control System is composed of various high-level services which work together to enable the control of the accelerator complex, experimental areas and across various supporting technical infrastructure. The information provided by these controls services is continuously accessed and modified by different software and processes to ensure optimal operation. Regular evolution of the software and hardware across the complex is needed to meet availability and performance targets. High-level APIs play a crucial role in responding to these demands, enabling developers and experts with greater ease and speed when working with software that needs to configure or interrogate the Accelerator Controls. The API became an integral part of software building blocks, reshaping how software systems are designed, implemented, integrated, and maintained.

This paper presents the journey of continuous evolution of APIs used by CERN Accelerator Controls for the needs of systems configuration and integration, from rudimentary function calls to sophisticated, feature-rich end-user interfaces. High-level APIs have been instrumental in unlocking the potential of cutting-edge technologies by providing intuitive and expressive programmatic interfaces easing the access for the end-users, and equally, when integrating complex systems. The paper describes technologies and concepts used to provide reliable, robust and scalable APIs for the centralised CERN Controls Configuration Service (CCS) [1].

CONTROLS CONFIGURATION SERVICE

The CCS is a core component of CERN's Control system, serving as a central point for the configuration of all Controls sub-domains, in a coherent and consistent way. The CCS is used by diverse user groups, including installation teams (configuring Controls hardware), equipment experts (configuring processes and applications), and accelerator operators. Though CCS downtime does not directly impact on-going beam operation, CCS users rely on being able to interact with the service at any point in time, to verify or define appropriate configurations. As such, extended downtime periods are unacceptable. To provide the highest possible availability and quality of service, the CCS, including its API are realized as a modular system, with redundancy, monitoring, and alerting at its core. Each day, on average, more than 400 different users and processes generate 80 million requests using the CCS API to obtain or modify various configurations (Figure 1). Peak values of user traffic can reach considerably higher levels.

Figure 1: Number of requests per CCS API node.

API REQUIREMENTS AND DESIGN

In addition to the core business requirements of the API, represented by the exposed data structures and operations, a robust API requires careful consideration of architectural, technological, and operational factors from the design stage, including:

- Scalability to meet the demands of a growing user base.
- Availability ensuring minimal downtime.
- High performance.
- Fault tolerance.
- Security and stability.
- Accessibility by non-software-developers and across different programming languages and technologies.

The overall CCS architecture is based on the REST (Representational state transfer) architectural pattern - the de-facto industry standard for HTTP-based, high-level communication between different systems. REST based APIs run on an HTTP server, are language agnostic and natively supported by popular programming languages and web browsers. Scalability and high availability are achieved by deploying multiple instances of the API on redundant physical machines to safeguard against downtime caused by hardware or network failures, therefore ensuring continuous operation. This is facilitated by the server instances being stateless and therefore not depending on any shared information, as well as isolating requests from each

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TH2A004

other. As a result, server nodes can be transparently added or removed in function of traffic and peak demand.

To uniformly distribute incoming requests and mitigate the risk of overloading a single API instance, load-balancing across API instances is implemented on both the API server-side and within dedicated Java & Python CCS client-side API libraries. The load-balancers also help fault tolerance by transparently directing client requests to healthy API server instances in case a server becomes unavailable.

Concerning high-performance requirements, a caching mechanism is used to maximise response times, as in many situations, the same static or immutable data is read many times by independent clients. To avoid each request to travel from client via server to the persistence layer (Oracle database) a cache mechanism was introduced on both the client-side and server-side. As a result, the response time for accessing cached data was reduced by up to a factor of 10. For example, when extracting a list of CERN accelerators (immutable data), the total response time has been reduced from 40 ms to 4 ms when using the cache in comparison to a direct database call (JDBC generated by Hibernate via Spring JPA).

Fault tolerance is implemented by error handling and a retry mechanism for failed requests. This prevents service disruptions caused by temporary issues (hardware problems, network issues or server unavailability) and ensures that the API handles permanent errors gracefully, responding to the user in a meaningful way (e.g. with specific error codes or clear messages).

to guard the API against DDoS (distributed denial-of-service) attacks and accidental API misuse by unaware end-users, thus minimizing the impact on others. Throttling is essential in the API design, considering that although the API is scalable, the amount of underlying server and network resources cannot be scaled-up indefinitely.

Orthogonal aspects in robust API design include monitoring and logging, which are crucial to track API performance, user activity and abnormal system behaviour, or to identify potential bottlenecks. For the CCS API, off-the-shelf solutions like Prometheus, Grafana, and the ELK stack are used. Combined, well-structured logs, metrics, alerts, and dashboards help system engineers gain insights into API behaviour and its health, both of which need consideration for proactive software system control.

Within the CCS API, attention to the aforementioned aspects help ensure that the expectations are met. The next chapter describes more details of how the implementation is realised.

TECHNICAL IMPLEMENTATION

The selected technology and implementation aim to meet the demands of modern software by providing a cutting-edge, robust, and highly available solution. As a result of an initial design and subsequent prototyping validation phase, it was decided to use Java, Spring Boot and the Netflix Cloud ecosystem for the API service implementation. This technology stack was chosen for its versatility, platform independence, and comprehensive toolset, making it the ideal foundation for developing RESTful APIs. At its

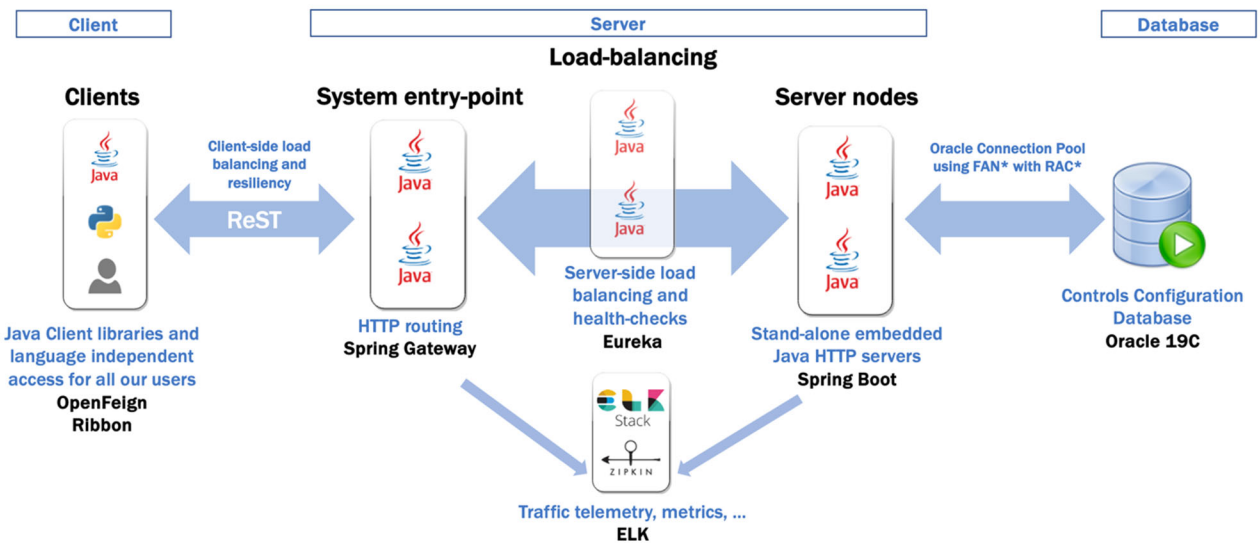

Figure 2: Redundant REST API architecture.

Security is based on industry standards for authentication and authorization mechanism to protect sensitive data by restricting the access to authenticated and authorized users. Request rate limits and API quotas (throttling) are used

core, Java provides the reliability and cross-platform compatibility necessary for a wide range of deployments. Spring Boot streamlines development, minimizing configuration overhead and allows for rapid application setup.

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TH2A004

Incorporating the open-source Netflix Cloud library elevates the API's capabilities, enhancing its resilience and scalability. Eureka which is one of the core elements of Netflix Cloud framework is used as a service discovery to ensure seamless service registration and discovery, facilitating dynamic routing and load-balancing. This feature allows clients to effortlessly find and interact with API instances making it highly available and responsive. The integration of Netflix Ribbon (IPC, the Inter Process Communication client-side library) closely intertwined with Eureka, adds client-side load balancing, evenly distributing incoming requests across multiple API server instances. This load balancing mechanism optimizes performance and crucially, eliminates the single point of failure by ensuring that the system can seamlessly continue serving clients when server instances experience issues (provided that at least one server instance remains on-line). To enhance fault tolerance, Resilience4j, a circuit breaker mechanism, is used. By isolating points of access to remote services, it prevents cascading failures and gracefully handles issues, maintaining the API's stability during adverse conditions. This proactive approach to fault tolerance is crucial for delivering uninterrupted service even when components within the system encounter problems.

Redundancy is a fundamental principle in the architecture of REST APIs when it comes to ensuring high availability. Multiple instances of the CCS API servers are deployed across different bare metal machines and are automatically registered with Eureka for service discovery by clients. This redundancy guarantees that the load balancer can distribute traffic effectively, even in scenarios where one or more API server instances fails or requires maintenance. As a result, the overall API service remains resilient and consistently available. Figure 2 shows an overview of the CCS redundant REST API architecture.

Scalability is another core aspect of the CCS API's architecture. The stateless design allows for easy horizontal scaling of server instances to accommodate varying workloads, and auto-scaling policies are employed to dynamically add or remove instances based on traffic patterns, ensuring optimal resource utilization. In other words, to avoid wasting machine resources, only a minimal number of physical nodes are kept, as needed, to fulfil the client requests. In case of an increase in traffic or when requests are taking more time than the defined thresholds, additional nodes will be started, and Eureka will automatically redirect new clients' requests to them.

Authentication and authorisation are based on standard Spring Security mechanisms, customized, and integrated with CERN-specific infrastructure. Role-based authorization is used, which is a foundational concept in designing secure and controlled REST APIs. It establishes a structured framework where users or clients accessing an API are granted permissions based on their assigned roles. These roles, such as "CCS-DEVICE-EDITOR" or "CCS-HARDWARE-EXPERT" come with predefined sets of privileges that dictate the actions and data a user can access or modify. Role-based authorization offers a range of benefits, including granular access control, scalability for evolving applications, compliance with regulatory standards, simplicity in administration, and enhanced security. The assignment of roles to individual users is managed directly by relevant service or domain experts. This gives the flexibility to grant and revoke permissions based on changing circumstances, for example when expert support team interventions are ongoing and require elevated permissions. Role management is also exposed as part of the CCS REST API, allowing automation of role assignment. An example usage is to assign or grant roles according to the current operational status of the machines within CERN's accelerator complex. The actual implementation of the role-based authorization is based on the standard CERN Role-Based Access Control (RBAC) framework [2]. Integration of CERN's RBAC with Spring Security is seamless from the end-user perspective, with RBAC access tokens translated into OAuth2 (recently replacing the use of SAML).

The main high-level programming languages used within CERN accelerator controls environment are Python and Java. To optimise the usability of the underlying, language independent REST-based CCS API, dedicated Software-Development Kits (client SDKs) have been developed. The dedicated Java and Python SDKs encapsulate intrinsic details of the REST implementation (including details of JSON-based serialization) and allow to leverage language-specific features and 3rd-party libraries. In addition, the end-user developer experience when using these SDKs is enhanced by language-specific IDE code-completion features. The design pattern of fluent interfaces is also used to further help the users (Figure 3).

The REST API endpoints are documented using The OpenAPI Specification (OAS, previously Swagger) [9] such that as shown in Fig. 4, clients wishing to use the API directly (without a client SDK) can easily identify:

- Which end points they need to interact with.
- The JSON data formats.

```
String search = SearchBuilder.search() PredicateBuilder
        .predicate(DeviceQueryField.name, SearchOperator.EQUAL, ...arguments: "DEVICE.*") AfterPredicateBuilder
        .and() PredicateBuilder
        .predicate(DeviceQueryField.accelerator, SearchOperator.EQUAL, ...arguments: "LHC") AfterPredicateBuilder
        .build();

CcdaPage<Device> devices = deviceService.search(search, page: 0);
```

Figure 3: Fluent style Java client SDK example.

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TH2A004

- The semantics of the available HTTP methods.

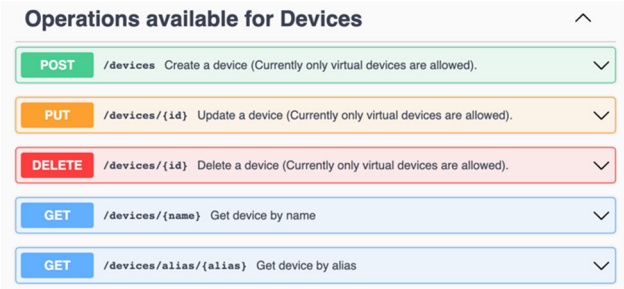

Figure 4: OpenAPI documentation.

API DEVELOPMENT PROCESS

To optimise the API development process, efficient and collaborative tools and practices are indispensable. For CCS API development, GitLab together with a Merge Request (MR) workflow is used [8]. This workflow streamlines how team members work on code changes and helps play a crucial role in orchestrating modern software development. Developers use feature branches, making incremental commits, and then propose their changes through MRs (Figure 5).

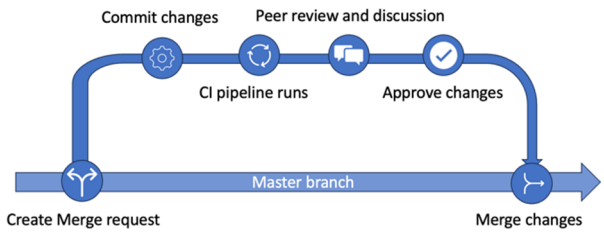

Figure 5: Merge request workflow.

This approach facilitates agile development of new features by avoiding developers working on distinct features disturbing one another. Each change can be designed, implemented, and tested at its own pace. To automate and facilitate code development and deployment CI/CD pipelines are vital. For the CCS API, Jenkins automates the CI process, continuously building, testing, and validating code changes, triggered by both MRs and scheduled nightly jobs. This automation provides developer feedback, guarantees that new code meets quality standards, helps ensure things work as expected and avoids regression.

Once the integration phase (CI) has successfully completed, Ansible [5] is used for the deployment phase (CD). Ansible playbooks define the deployment process, offering a robust and consistent method for delivering software to various environments. This approach eliminates the need for developers to perform manual operations to build and deploy the REST API services. It also guarantees that deployments are reproducible and error-free, across the development team machines. Additionally, Jenkins and Ansible provide monitoring, logging, and alerting capabilities, allowing for quick responses to any build or deployment issues that may arise during the pipeline execution.

Combining GitLab, Jenkins, and Ansible allows to establish an agile, robust, efficient, and reliable software delivery process. Overall, the integration facilitates team collaboration, code quality and deployment automation which in turn help accelerate software delivery with confidence. Since the complete and successful adoption of CI/CD for the CCS, a reduction in end-user support requests has been observed, which can be attributed to the deterministic nature of the development, release and management process of the API services.

Testing is an integral part of software development and helps ensure that code works as expected, performs efficiently, and fulfil standards and user requirements. In addition to standard unit, integration, and regression testing, the CCS API development process uses Gatling [6] and ArchUnit tests [7]. Gatling is an open-source framework designed for performance and stress testing of REST based services and applications. It allows service providers to simulate different API usage scenarios and observe how services behave in terms of resource consumption and stability under various loads. It helps to detect undesired or unexpected behaviours during high and continues load. By incorporating Gatling framework in CI/CD pipelines, potential performance issues can be detected before rolling out a new API version in production, thus mitigating risks for end-users (Figure 6).

```
> request count                                       12577 (OK=12577  KO=0     )
> min response time                                       4 (OK=4      KO=-     )
> max response time                                    1215 (OK=1215   KO=-     )
> mean response time                                     86 (OK=86     KO=-     )
> std deviation                                         168 (OK=168    KO=-     )
> response time 50th percentile                          36 (OK=36     KO=-     )
> response time 75th percentile                          75 (OK=75     KO=-     )
> response time 95th percentile                         278 (OK=278    KO=-     )
> response time 99th percentile                         973 (OK=973    KO=-     )
> mean requests/sec                                78.118 (OK=78.118 KO=-     )
---- Response Time Distribution ------------------------------------------------
> t < 500 ms                                         12197 ( 97%)
> 500 ms < t < 2000 ms                                 380 (  3%)
> t > 2000 ms                                            0 (  0%)
> failed                                                 0 (  0%)
```

Figure 6: Example Gatling tests summary.

ArchUnit enables architecture testing and provides the possibility to define and enforce architectural constraints and standards, and to verify that code meets predefined rules. For example, such rules can check dependencies between classes, packages, different layers of the software, naming conventions and more.

Incorporating both test frameworks into the overall development process helps ensure that software follows best architectural practices and performs well under demanding conditions. For the CCS API, software quality has been improved as a result, with a reduced number of runtime issues and bugs, including otherwise difficult to detect issues of an asynchronous nature.

FUTURE IMPROVEMENTS

The current implementation and architecture of the CCS API provides a high availability and reliable service, that meets the current performance expectations. Nevertheless, a prototype deployment of the CCS API on a Kubernetes platform highlighted that an evolution of the system to adopt containerisation and orchestration would bring further improvements. Currently, CSS API server instances are deployed on bare-metal machines, coupling them to specific hardware and Operating System. A move towards orchestrated container images deployed on a Kubernetes

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TH2A004

platform will further facilitate automatic deployment, scaling, and fault tolerance, independently of hardware upgrades, etc. This work is expected to take place later in 2024.

A REST API, by definition, should provide immediate responses and not keep state on the server. More CCS use cases are emerging whereby a request involves interactions with many controls sub-systems and therefore, requires time to gather and process results, then provide a response. To avoid keeping clients connected and therefore blocking resources, an asynchronous model of communication is being planned. In the new approach, each operation will be composed of two phases:

1. A request from the client process.
2. A notification upon completed of the operation.

The implementation will be based on Kafka [3,4], an open-source, distributed event streaming platform which is currently part of the CCS API cache eviction mechanism. Kafka facilitates integration of various clients and thanks to its persistence, guarantees messages delivery to clients, even if clients are temporarily available. Kafka supports redundant instances with auto-balancing of traffic, high-availability guarantees, and very low latency.

CONCLUSIONS

When it came to selecting a technology for implementing the CCS API, the primary goal was to offer a solution that was not only user-friendly but also highly reliable and robust, destined to become an integral component of the Controls Configuration Service. Five years since the initial implementation of the REST API, all requirements have been met or even exceeded.

Presently, the CSS API is used by a diverse community of over 400 users and processes, easily accommodating more than 80 million requests per day. Remarkably, apart from the scheduled maintenance periods during CERN Technical Stops, only three incidents of unavailability were recorded. These rare occurrences were swiftly mitigated thanks to the comprehensive monitoring and alerting system, resulting in minimal disruptions to the operation of the particle accelerator complex.

The CCS API's ease of use has not only fostered its widespread adoption but has also inspired other developers to readily incorporate REST APIs into their controls software services as the preferred method of communication between various applications and sub-systems.

REFERENCES

[1] L. Burdzanowski *et al.*, "CERN Controls Configuration Service - a Challenge in Usability", in *Proc. ICALEPCS'17*, Barcelona, Spain, Oct. 2017, pp. 159-165. doi:10.18429/JACoW-ICALEPCS2017-TUBPL01

[2] 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

[3] B. Urbaniec and L. Burdzanowski, "CERN Controls Configuration Service - Event-Based Processing of Controls Changes", in *Proc. ICALEPCS'21*, Shanghai, China, Oct. 2021, pp. 253-256. doi:10.18429/JACoW-ICALEPCS2021-MOPV043

[4] Kafka, https://kafka.apache.org/

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

[6] Gatling, https://gatling.io/

[7] ArchUnit, https://www.archunit.org/

[8] Gitlab MR workflow, https://docs.gitlab.com/ee/development/contributing/merge_request_workflow.html

[9] Swagger, https://swagger.io/specification/