FirecREST: RESTful API on Cray XC systems

Felipe A. Cruz, Maxime Martinasso
Swiss National Supercomputing Centre, ETH Zurich, Lugano, Switzerland
felipe.cruz@cscs.ch

***Abstract*—As science gateways are becoming an increasingly popular digital interface for scientific communities, it is also becoming increasingly important for High-Performance Computing centers to provide a modern Web-enabled APIs. With such interface, science gateways can easily integrate access to HPC center resources. This work presents the FirecREST API, a RESTful Web API infrastructure that allows scientific communities to access the various integrated resources and services available from the Cray XC systems at the Swiss National Supercomputing Centre. FirecREST requirements are derived from use cases described in this work.**

***Keywords*-RESTful API; microservices; science gateways;**

I. Introduction

The recent technological progress on Web Application Programming Interfaces [1] (Web APIs) provides third-party developers with frameworks for building HTTP based services that can be accessed by software applications over a variety of platforms. In this way, developers can envision and implement new business processes, build new client workflows that simplify the user experience or enable them to develop completely new platforms and services.

The core of the current Web API development gravitates towards Representational State Transfer (REST) [2]. RESTful API is a software design pattern, that specifies a uniform and predefined collection of stateless operations. The REST software architecture is well suited for enabling services that work over the Web as this design pattern introduces several desirable properties for web services, such as performance, scalability, and flexibility. As such, RESTful Web APIs provide interoperability between systems and applications over the web. It has become the building blocks for web software development, it reduces the total time required for software development of web-enabled applications and portals, and, it provides improved integration across multiple services and organizations.

Over the past year at the Swiss National Supercomputing Centre (CSCS), we developed FirecREST, a RESTful Web API infrastructure that allows scientific communities to access the integrated resources and services available at CSCS. FirecREST web services allow science gateway [3] developers to integrate their platforms with High-Performance Computing (HPC) resources such as the CSCS flagship Cray supercomputer Piz Daint. In practice, the FirecREST API allows access to two core HPC services: submitting and monitoring jobs on HPC systems, and moving data across multiple filesystems. Moreover, it does so while enforcing integration with the Authorization and Authentication Infrastructure deployed at CSCS.

In this work, we present the architecture and the capabilities of the FirecREST API. Prior to describe the API, we introduce use cases that have driven its requirements. Whereas the work we present targets Cray systems, FirecREST is a generic interface that can be used on other non-Cray HPC systems. It interfaces primarily to the batch scheduler and HPC storage technology.

The key contributions of our work are:

- to present concrete use cases that require the capability for an HPC center to provide resource access through a Web interface;
- to describe the architecture and capability of the FirecREST API.

II. Use cases

FirecREST will improve the accessibility of HPC resources to scientific communities by enabling them to build platforms that target specific scientific goals. From the perspective of CSCS, providing a Web API to Cray system resources will expand the center user base as these scientific platforms develop. Platform developers will benefit from a standard interface, whereas CSCS provides a single technology to satisfy multiple user communities. In this section, we present three use cases for which FirecREST provides key programmable functionalities.

A. Swiss Data Science Center

The Swiss Data Science Center[1] (SDSC) aims to help the academic community and the industrial sector to work on Artificial Intelligence and Machine Learning, and, to facilitate the multidisciplinary exchange of data and knowledge. SDSC has developed a solution called RENKU [4], a software platform for doing reproducible and collaborative data science. Piz Daint with its GPU-enabled nodes is a preferred infrastructure to run data science workload. Therefore, as requirements for FirecREST, users of RENKU should be able to submit jobs to Piz Daint and to move data to and from Piz Daint storage. As a software platform, RENKU requires an infrastructure to be executed on and both requirements need to be possible from any Cloud infrastructure. RENKU

[1] https://datascience.ch

is still in development and its integration with Piz Daint is in progress at the time of this writing.

B. Materials Cloud

The Materials Cloud[2] is a web platform to share resources in computational materials science. One feature of the Materials Cloud is to run computational intensive jobs of well-known HPC-aware scientific application for discovering new properties of materials.

The Materials Cloud has been released and is online. It it is currently running on a Cloud system at CSCS. Computational intensive jobs are submitted to Piz Daint. To enable reproducible workflows and job submissions, the Materials Cloud uses AiiDA [5]. On the technical side, AiiDA interacts with Piz Daint by executing SSH commands.

AiiDA will greatly benefit from a RESTful API. It will ensure a better security, a lower effort of maintenance (as it is CSCS responsibility to provide a working API service) and a simplification of its internal mechanisms to access Piz Daint.

C. Interactive CSCS service

The third and last use case drives requirements for an internal CSCS service. CSCS offers an interactive service[3] based on Jupyter notebooks [6]. Jupyter notebooks have become a preferred interface for scientists and many HPC centers provide a similar interactive service. Moreover, science gateways are using Jupyter as a user-facing interface. For instance, the two previous use cases are running their web interface through a JupyterHub[4] service.

The current CSCS interactive service uses a standard Jupyter notebook spawner connected to the batch scheduler used to submit jobs on Piz Daint. In the future, this spawner will be modified to use a RESTful API. One of the main advantages of using a standard API technology is to easily target multiple infrastructures for executing notebooks. Each infrastructure has a cost model associated to, and, it becomes possible for the end users to target either an HPC system for dedicated resources and a higher cost or a Cloud system for shared resources and a lower cost.

D. Summary and requirements

From the above use cases we can identify three major requirements:

- the necessity from the API to integrate with various identity providers external to the center. In the two first use cases, users don't necessarily have accounts at CSCS. This feature depends on the global Identity Management Access policy of the center and the use of standard authentication protocols;
- the capability to manage the execution of workloads on Piz Daint or any other HPC or Cloud systems;
- the possibility to enable external transfers of data to/from the centre filesystems attached to Piz Daint or any other HPC system.

We expect that FirecREST will enable new use cases and further increase the reach of HPC to scientific communities by: enabling the development of more modern and comfortable web interfaces to HPC, and, by providing an interface to the centre resources that is common, stable, secure, and maintainable, thus avoiding scientific community platforms to implement their custom interfaces and integration with infrastructure.

III. FirecREST microservice architecture

FirecREST provides to developers a web-enabled API to Piz Daint that is stable, secure, and maintainable. It allows client applications to access the resources available on the Cray XC system. Internally, FirecREST translates every HTTP request into its appropriate operations on the supercomputer, such as: enforcing authentication and authorization, job management, data mover, and other operations. The operations to perform are loosely coupled and involve different resources. Thus, in order to improve maintainability, test-ability, and to match CSCS organization, we followed a microservice architecture built using open source tools such as Keycloak[5], Kong[6], Flask[7], Paramiko[8], OpenSSH[9], OpenAPI[10], and Redis[11]. Figure 1 describes the FirecREST microservice architecture diagram. In the following subsections, we present the core components and microservices that are part of FirecREST: Identity Access Management, API gateway, compute, storage, utilities, asynchronous tasks execution, delegation, and status.

A. Identity Access Management

The Identity and Access Management (IAM) infrastructure at CSCS ensures that users and web applications have the appropriate permissions to access resources at CSCS by using a secure protocol. From the whole of the IAM infrastructure at CSCS we will only discuss Keycloak, the Identity and Access Management solution deployed at the center. Keycloak allows to secure application and services by providing a mechanism for the authentication and authorization of CSCS users, CSCS services, and third party applications. Among the many features of Keycloak we highlight the following:

- single sign-on solution

[2]https://www.materialscloud.org/
[3]https://jupyter.cscs.ch
[4]https://jupyter.org/hub
[5]https://www.keycloak.org/
[6]https://konghq.com/kong/
[7]http://flask.pocoo.org/
[8]http://www.paramiko.org/
[9]https://www.openssh.com/
[10]https://www.openapis.org/
[11]https://redis.io/

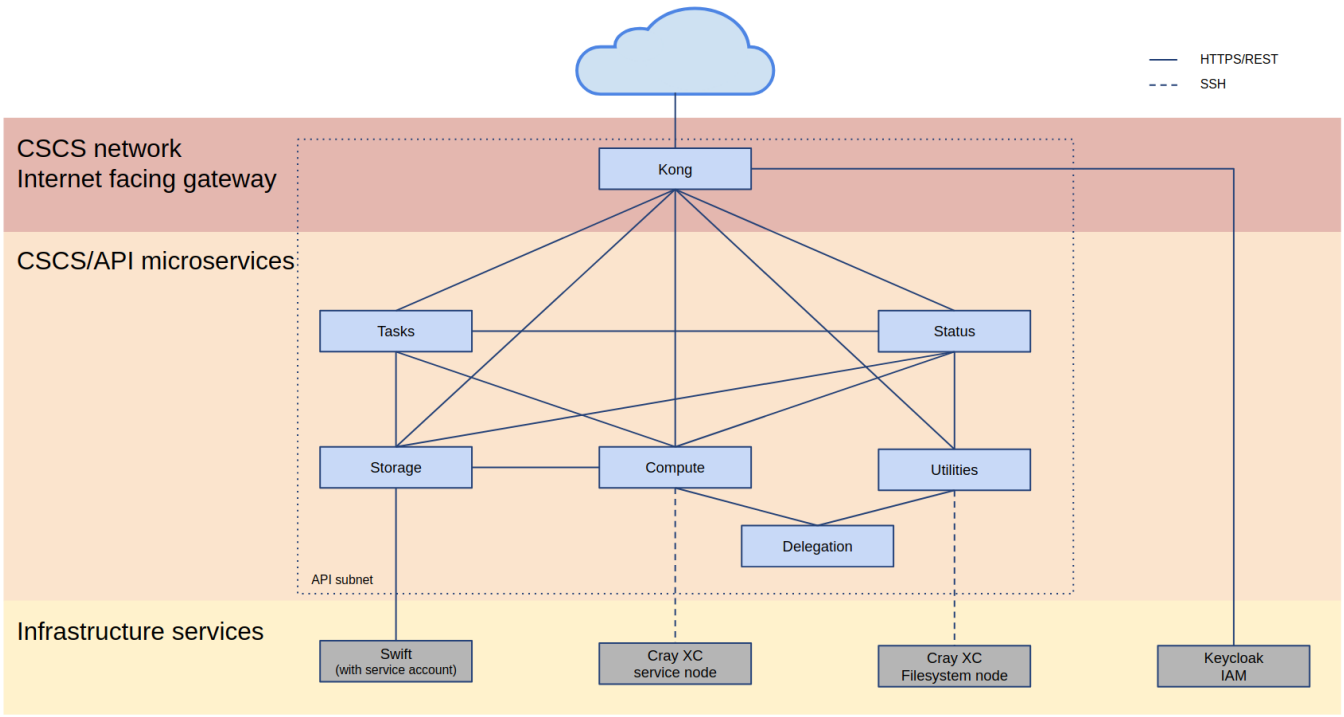

Figure 1. FirecREST microservice architecture.

- integration with Kerberos authentication service
- fine-grained authorization controls for services
- client registration and authorization
- support of OpenID Connect (OIDC) protocol[12]

The integration of FirecREST with Keycloak has been achieved through the use of the OIDC protocol. OIDC is an authentication protocol that extends the OAuth 2.0 specification. OAuth 2.0 is an industry-standard protocol for token-based authorization that is commonly used as a mechanism for users to grant access permission to web application in order to access user-owned resources and services. The extensions provided by OIDC to the OAuth 2.0 protocol add a user authentication layer, providing a mechanism that enables single sign-on to users.

FirecREST leverages on Keycloak and OIDC for authentication and authorization of web applications, enabling the following capabilities:

- enforce that all requests are authenticated
- applications never manipulate user credentials
- only allow requests from registered applications
- user-managed access permissions per application
- stateless security model by use of tokens
- short lifespan for sensitive access tokens
- extended client sessions allowed through refresh tokens

In a nutshell, FirecREST OIDC-based IAM enables the user to login to a registered web application using their CSCS credentials and grant a web application with access to user-owned resource at the Center. Moreover, it does so without the user ever sharing their credentials with the web application. Figure 2 presents a complete description of the OIDC Authorization Code Flow used by FirecREST.

[12]https://openid.net/connect/

B. API Gateway

The API gateway provides an interface to publish, maintain, monitor, and secure all the FirecREST API endpoints. As shown in Figure 1, the gateway is hosted on a machine within CSCS that is facing the internet. All interactions with the FirecREST API are first passed and validated before being redirected to any other FirecREST microservice.

In this way, every request made to the FirecREST API arrives first at the gateway, which will proxy the request towards the requested microservice endpoint. However, before the request is passed on to the microservice, the gateway will enforce that the request are correctly authenticated and authorized by requiring and validating the Access Token that must accompany each API request, as described in Section III-A.

The current implementation of the gateway service is based on the Kong API gateway. Kong is a widely used open source microservice API gateway that implements functionalities such as a variety of authentication and authorization mechanisms, support for OIDC, IP filtering, access control lists, analytics, rate limiting, among many others that have allowed us to configure the gateway to our requirements.

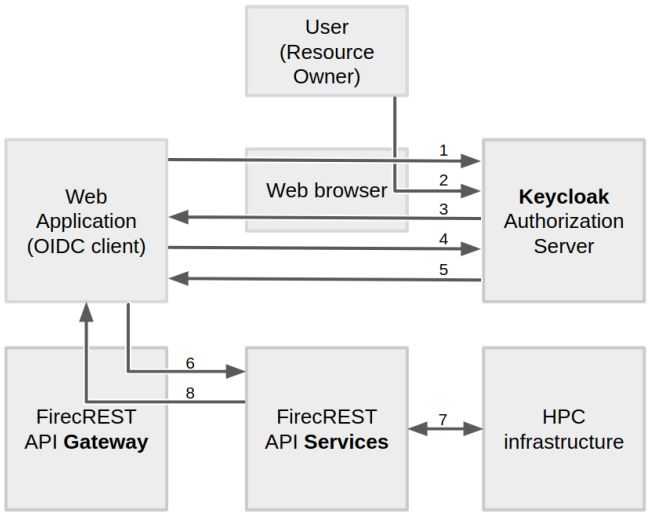

Figure 2. FirecREST authentication and authorization workflow: 1) Client performs a user authentication and access permission request; 2) User authenticates himself and authorizes client application; 3) Keycloak responds to the application with an authorization code; 4) Application uses the authorization code to request for an Access Token; 5) Keycloak grants the Access Token to the application over a secure backchannel; 6) Application request access to user-owned resources via the FirecREST API Gateway, enforcing the Access token permissions and redirecting to the correct service API endpoint; 7) FirecREST services translate the web request into actions on the HPC infrastructure; 8) FirecREST responds to the web application request.

C. Compute

The compute microservice implements the interface to the workload manager, thus allowing applications to submit, stop, and query the status of jobs by using non-blocking asynchronous API calls. This service depends on: the tasks microservice (see section III-H) that provides a temporal resource that tracks the state of each call; and, the delegation microservice (see section III-F) that issues a restricted SHH certificate that allows the execution of operations on behalf of the user. We now describe the integration with the SLURM [7] workload manager used by Piz Daint.

For conciseness, in this section we will only describe the *job submission* workflow, as other operations follow similar or simpler workflows. Let us consider now the *job submission* workflow shown in the sequence diagram in Figure 3. As it can be observed, the job submission starts with the client calling the API endpoint `firecrest/jobs` with a `POST` operation, passing the following parameters: the access token which identifies the user and authorizes the call; the system where the job is submitted to; and, a file part that contains the job definition written in SLURM's sbatch format. Upon receiving a request the compute microservice checks the validity of the parameters passed with the request and then call the tasks microservice, creating a new task which will track the progress of the operation returning a task resource as an immediate response to the client request. The client will use the task microservice to access the task resouce (identified by its `task id`) and use it to track the status of the request in an asynchronous way, meanwhile the compute microservice continuously updates the tasks resource as the job request progresses.

The compute microservice now requests to the delegation microservice an SSH certificate, passing the access token as a parameter. The delegation microservice will respond by retrieving the username from the access token and generating an SSH certificate that will be signed with the *Certificate Authority key* (see section III-F for details on this). Thus, the delegation microservice responds to the compute microservice request with a valid SSH certificate and the related public and private keys used in the process.

Next, the compute microservice makes use of the Paramiko library to establish an SSH session using the certificates and keys obtained from the delegation microservice. Over this SSH session, a unique temporal folder in the users's `$HOME` directory is created, at this location the job information will be stored. The compute microservice then copies the sbatch script into the newly created temporal directory. Finally, the microservice runs the script using an sbatch command over the SSH session and captures its output, updating the task identified by the initial `task id` accordingly. Information in the task resource, such as `SLURM's job id field` can be used by the client to query for the state of the scheduled job.

We now provide an overview of all the compute microservice functionality by API endpoint, operation, and parameters:

1) Endpoint: /Jobs/
   a) Operation: POST
      - Parameter: user, machine, scheduler script file.
      - Description: Submits a job with the scheduler script file that targets the specified machine.
   b) Operation: GET
      - Parameter: user, machine.
      - Description: Retrieve information from all jobs for the user at a specified machine.
2) Endpoint: /Jobs/acct
   a) Operation: GET
      - Parameter: user, machine.
      - Description: Retrieves account information from user at specified machine.
3) Endpoint: /Jobs/jobid
   a) Operation: GET
      - Parameter: user, jobid, machine.
      - Description: Retrieve information from a job with jobid at a specified machine.
   b) Operation: DELETE
      - Parameter: user, jobid, machine.
      - Description: Cancels a job with jobid at a specified machine.

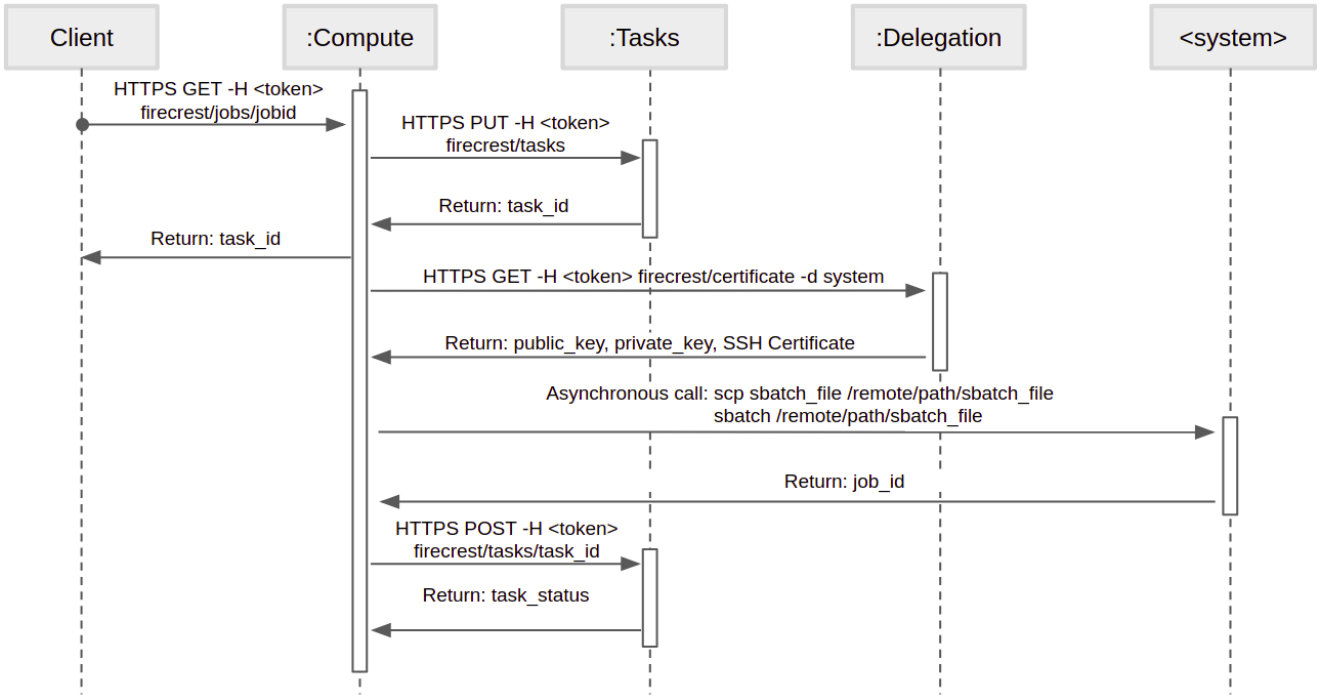

Figure 3. Sequence diagram of the job submission workflow. Please note that for conciseness we have skipped the API gateway step from the description, however, all interactions with the FirecREST API are passed and validated first by the gateway.

D. Data mover

This microservice enable users the upload and download of large files to/from CSCS, while also enabling the movement of data within the different filesystems available on the Cray XC system (Piz Daint). It does so by using non-blocking calls to high-performance storage services while immediately responding with a reference to a resource that tracks the state of the request (see section III-H). A full description of the upload and download workflow is presented in Figures 4 and Figure 5 respectively.

We now provide an overview of all the storage microservice functionality by API endpoint, operation, and parameters:

1) Endpoint: /Storage/xfer-external/upload
   a) Operation: GET
      - Parameter: user, target path.
      - Description: First step of the asynchronous upload workflow, returns the tasks with information on the upload URL at the staging area.
   b) Operation: GET
      - Parameter: user, task-id.
      - Description: Last step of the asynchronous upload workflow, moves data from upload staging area into POSIX filesystem.
2) Endpoint: /Storage/xfer-external/download
   a) Operation: GET
      - Parameter: user, source path.
      - Description: Asynchronous download workflow, returns the tasks with information on the download URL from staging area once the file from the POSIX filesystem has been made available.
3) Endpoint: /Storage/xfer-external/operation
   a) Operation: GET
      - Parameter: user, operation, target path, source path (optional).
      - Description: Asynchronous workflow for executing recursive operations on the POSIX filesystem, returns the tasks with job information scheduled to perform the operation. Operations that are possible: rsync, mv, rm.

E. Utilities

The utilities microservice provides synchronous execution of the following linux commands. As calls to the utilities microservice are blocking operations, these have a timeout and are not recursive.

- GET /Utilities/ls
- GET /Utilities/file
- POST /Utilities/mkdir
- POST /Utilities/rename
- POST /Utilities/chmod
- POST /Utilities/chown

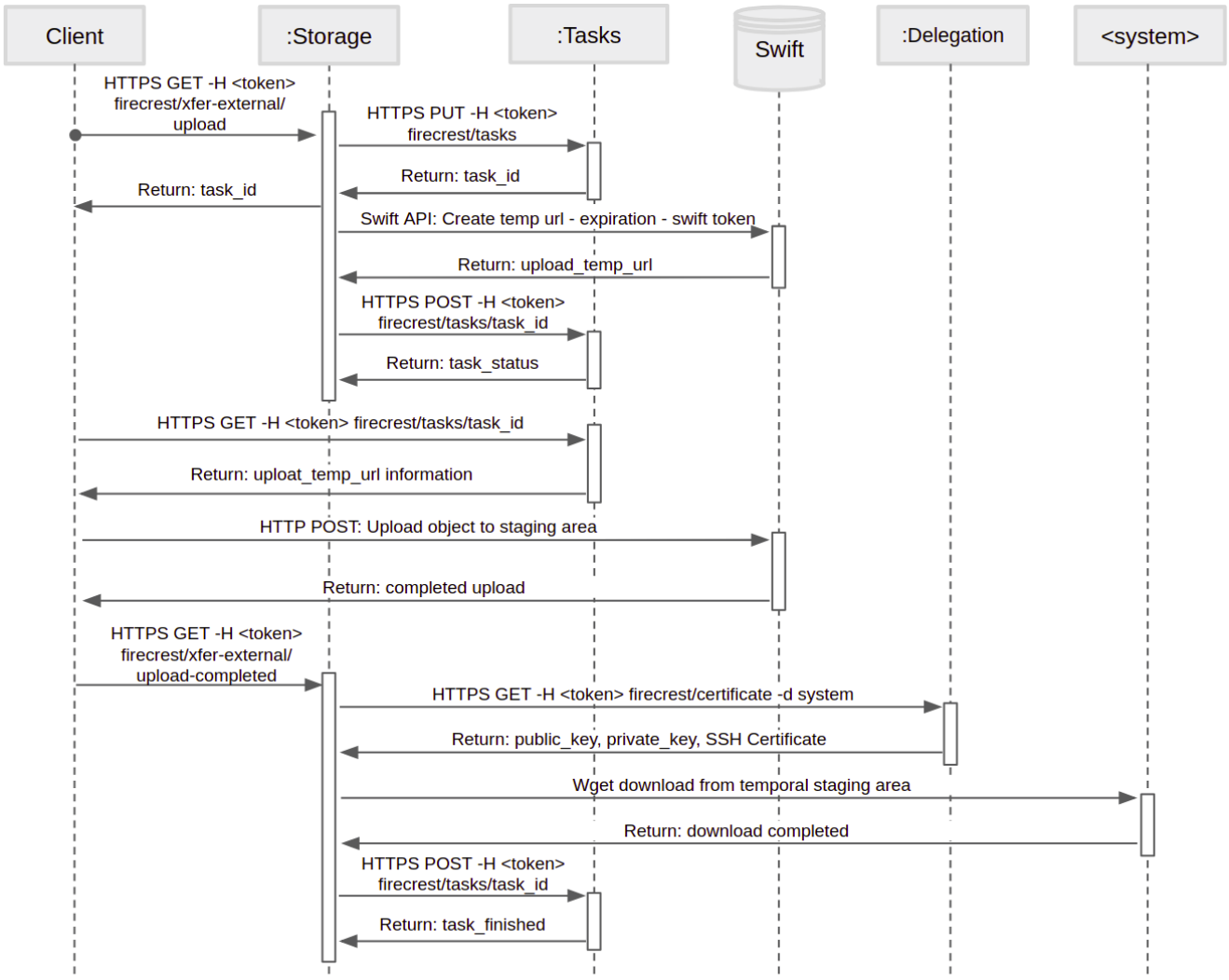

Figure 4. Sequence diagram of the data upload workflow that describes an asynchronous upload of a local file into CSCS infrastructure. The client first calls storage microservice giving parameters such as system, filesystem path and local file path. The storage microservice then uses Python's swiftclient with a FirecREST SWIFT service account in order to generate an HTTP form for temporary file upload. The client can then use the temporary upload URL to upload the file into an staging area. This circumvent the need for FirecREST to implement a high-performance fileserver, and leverages on existing solutions. Once the client upload to the staging area is completed, the user calls notifies the storage microservice by callong the upload-complete endpoint, this also provides the required user credentials that allow the storage microservice to start downloading object from SWIFT server into CSCS filesystem on behalf of the user.

- POST /Utilities/symlink

The utilities microservice also provides two convenient endpoint for uploading and downloading smalls files, these transfers are limited to files under a few megabytes and are intended for setting up experiments and other limited filesystem updates.

1) Endpoint: /Utilities/upload
   a) Operation: POST
      - Parameter: user, machine, path, file.
      - Description: Blocking call that uploads a file to the specified path on the machine filesystem.
2) Endpoint: /Utilities/download
   a) Operation: GET
      - Parameter: user, machine, path.
      - Description: Blocking call that returns the file from the specified path on the machine filesystem.

F. Delegation

The delegation microservice is a FirecREST internal service that is not exposed to the user. This service takes a valid JWT access token as input and creates a short-lived SSH certificate to be used to user authentication.

OpenSSH user-certificates are formed by: a public key; user identity information; and a set of constraints that limits

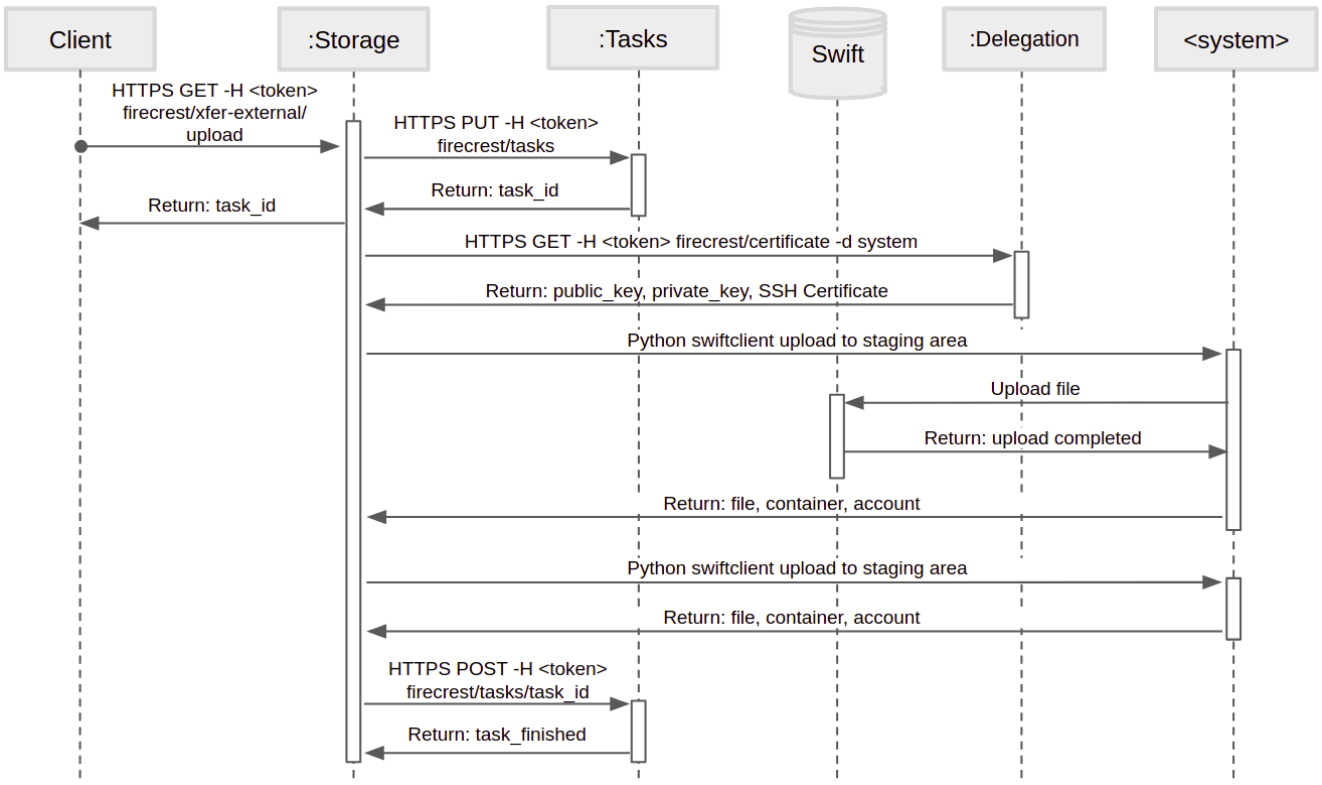

Figure 5. Sequence diagram of the data download workflow. The workflow for external data download is initiated by the client requesting a file from one of the HPC filesystem. An SSH user-certificate is created by the delegation microservice. The SSH certificate allows the storage microservice to upload the file into the FirecREST service account in SWIFT that is used as an staging area. Upon completion of the upload into the staging area, the storage microservice uses SWIFT API to create a temporary download URL. The temporal download URL is created by SWIFT with an unique hash containing that expiration time, object name and a secret, thus circumventing the need for the client to authenticate in order to start the remote download from SWIFT. Finally, temporary URL is returned to client.

the certificate validity. SSH certificates are signed using the `ssh-keygen` tool of `OpenSSH` with the `Certificate authority key` of the delegation microservice, also a standard SSH key. In order to enable SSH servers to accept certificates for user authentication, the sshd server must also be configured to trust the microservice CA public key.

Once a valid certificate is generated by the delegation microservice, other FirecREST microservices can use the certificate to perform remote command execution on behalf of the user. As such, this microservice enables FirecREST to perform delegation by means of secure system access using SSH certificates.

G. Status

This microservice provides information on the availability and state of services and relevant infrastructure that is accessible through FirecREST, such as the Cray XC system at CSCS.

We now provide an overview of all the status microservice functionality by API endpoint, operation, and parameters:

1) Endpoint: /status/systems
   a) Operation: GET
      - Parameter: user.
      - Description: Returns a list containing all available systems and response status.
2) Endpoint: /status/services
   a) Operation: POST
      - Parameter: user.
      - Description: Returns a list containing all available microservices with a name, description, status, and endpoint.

H. Tasks

The task microservice responds to the need of managing the state of request that are being resolved asynchronously. One clear example for the need of this microservice can be observed in the data transfer operations handled by the storage microservice (see section III-D), as otherwise some of those workflows would not be possible. As such, firecrest microservices during an asynchronous request can rapidly create and respond with a new task resource. The operational result of the request is then tracked as the originating microservice continuously updates the task as progress is being made. Thus, task resources allow a client to perform

other activities while a FirecREST asynchronous tasks are completed.

We now provide an overview of all the Tasks microservice functionality by API endpoint, operation, and parameters:

1) Endpoint: /Tasks/
   a) Operation: GET
      - Parameter: user.
      - Description: List all of the user's recorded tasks and their status.
2) Endpoint: /Tasks/
   a) Operation: POST
      - Parameter: user.
      - Description: Create a new task entry to keep track and link to resources. Exclusively used by FirecREST microservices. Not exposed to user.
3) Endpoint: /Tasks/id
   a) Operation: GET
      - Parameter: user, id.
      - Description: Retrieves information of a task. Exposed to user.
   b) Operation: PUT
      - Parameter: user, id.
      - Description: Updates a tasks identified by its id. Exclusively used by FirecREST microservices. Not exposed to user.
   c) Operation: DELETE
      - Parameter: user, id.
      - Description: Deletes a tasks identified by its id. Exclusively used by FirecREST microservices. Not exposed to user.

IV. FirecREST API specification

The FirecREST API has been described using `OpenAPI`[13] in `YAML` format. `OpenAPI` is a language-agnostic standard for describing all aspects of a REST API: endpoints, endpoint operations, operation input parameters, operation output, and authentication methods. Moreover, the FirecREST project can leverage on open-source tools [14] built to support OpenAPI that simplify reading and writing the API, API documentation, and automatic library generation.

V. Related work

The European project UNICORE [8] [9] aims to develop a general-purpose federation software suite by following standard Grid and Web services. Its core framework also named UNICORE for Uniform Interface to Computing Resources can federate in a single view different systems ranging from high-end HPC systems to single Linux servers. UNICORE development follows a project-based funding which constraint the project to a discontinuous-pace development focusing on adding new features. For instance, key features such as using modern protocols for authentication and authorization or using a API specification like OpenAPI[15] are not yet integrated. Our work offers a simpler approach by using a standard API which reduces its development cost compared to UNICORE.

NEWT [10] [11] aims to make HPC resources easily accessible to scientist by using Web applications. NEWT provides a RESTful API and we investigated its feature set prior to start this work. We found that some key aspects of our requirements were not integrated inside NEWT. For instance, NEWT has a monolithic architecture with one point of failure, authentication and authorization should be ported to modern protocols to integrate with third party client and delegation needs to be implemented. We concluded that NEWT has been developed and tailored with NERSC ecosystem requirements, and, porting it to CSCS environment requires an equivalent effort in terms of development.

VI. Conclusion and future work

In this work we present FirecREST, a RESTful Web API infrastructure that scientific gateways utilize to integrate with the High-Performance resources and services available from the Cray XC systems at CSCS. We intend to use FirecREST with the use cases presented in this paper.

As new use cases will emerge new requirements will be requested for FirecREST. As a concrete example, CSCS and the Paul Scherrer Institute (PSI) are collaborating to couple PSI scientific devices with CSCS compute capability. FirecREST is a key component to enable this connection, and, it will be extended to interface with a reservation service of compute nodes and a configurable data transformation service.

[13] https://github.com/OAI/OpenAPI-Specification/

[14] https://swagger.io/docs/open-source-tools/swagger-editor/

[15] https://www.openapis.org/

References

[1] M. Masse, *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O'Reilly Media, Inc., 2011.

[2] L. Richardson and S. Ruby, *RESTful web services*. " O'Reilly Media, Inc.", 2008.

[3] K. A. Lawrence, M. Zentner, N. Wilkins-Diehr, J. A. Wernert, M. Pierce, S. Marru, and S. Michael, "Science gateways today and tomorrow: positive perspectives of nearly 5000 members of the research community," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 16, pp. 4252–4268, 2015. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3526

[4] Swiss Data Science Center, "RENKU: Multidisciplinary data science collaborations made trustful and easy," https://datascience.ch/solutions/.

[5] G. Pizzi, A. Cepellotti, R. Sabatini, N. Marzari, and B. Kozinsky, "AiiDA: automated interactive infrastructure and database for computational science," *Computational Materials Science*, vol. 111, pp. 218 – 230, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0927025615005820

[6] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, and C. Willing, "Jupyter Notebooks – a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt, Eds. IOS Press, 2016, pp. 87 – 90.

[7] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60.

[8] D. W. Erwin and D. F. Snelling, "UNICORE: A grid computing environment," in *Euro-Par 2001 Parallel Processing*, R. Sakellariou, J. Gurd, L. Freeman, and J. Keane, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 825–834.

[9] M. Romberg, "UNICORE: Beyond web-based job-submission," in *Proceedings of the 42nd Cray User Group Conference*, 2000, pp. 22–26.

[10] S. Cholia, D. Skinner, and J. Boverhof, "NEWT: A RESTful service for building high performance computing web applications," in *2010 Gateway Computing Environments Workshop (GCE)*. IEEE, 2010, pp. 1–11.

[11] S. Cholia and T. Sun, "The NEWT platform: an extensible plugin framework for creating ReSTful HPC APIs," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 16, pp. 4304–4317, 2015.