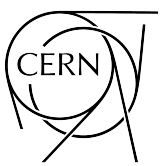
The Compact Muon Solenoid Experiment
TriDAS Trigger and Data Aquisition

Online Database Access Application for the CMS DAQ System in the High Luminosity LHC

Christine Zeh
TU Wien, Vienna, Austria

Document: Summer Student Report
Supervisor: Andrea Petrucci (UCSD)
Group: EP-CMD
E-mail: christine.zeh@cern.ch

September 27, 2024



European Organization for Nuclear Research
Organisation Européenne pour la recherché nucléaire
Espl. des Particules 1/1211, 23 Genève, Switzerland

Contents

1 Introduction

In large-scale experiments involving thousands of components, such as the Compact Muon Solenoid (CMS) at CERN, the dynamic and automated configuration of the system, coupled with continuous monitoring, is critical for the successful execution of data collection. The primary goal in such experiments is to gather meaningful data for Physic analysis, which requires robust infrastructure to ensure reliability and efficiency.

To address these needs, the XDAQ framework was introduced in 2005 as a software platform designed for distributed data acquisition systems [1]. Originally developed for the CMS experiment, XDAQ provides middleware functionality that simplifies the design, implementation, and management of the applications required for data acquisition (DAQ). Although XDAQ was tailored for CMS, its general architecture can be adapted for use in other data acquisition systems, and several Physic experiments in High Energy Physics (HEP) are using it.

Within CMS, XDAQ acts as the backbone of the data acquisition system, managing the distributed data collection, processing, and monitoring of data from the DAQ system. Applications in XDAQ are modular components that perform specific tasks, such as reading, filtering, and processing data. Services and libraries provide shared utilities, including network communication and memory and database access. Control communication between these distributed components is handled via SOAP, ensuring reliable and structured messaging across the system. A binary protocol is used for data transfer between processes. This modular and scalable architecture allows XDAQ to integrate seamlessly with other CMS subsystems, such as the Event Builder and Trigger System, making it a vital part of the overall data acquisition infrastructure.

Building on this distributed system, a dedicated monitoring pipeline was implemented in 2010 to track system performance and identify anomalies [2]. As an enhancement to this monitoring infrastructure, a fault detection and alerting architecture was incorporated, enabling rapid identification and resolution of issues as they arise [3]. In addition, the monitoring and exception data are stored in a database to support future analysis and prevention of similar issues. This task is managed by the TStore service, which provides a SOAP interface to facilitate communication between the DAQ applications and monitoring system to the database.

2 Data Storing Application in LHC Run 3

In the XDAQ framework used for the DAQ of the CMS experiment, the service TStore provides functionality for storing information to Oracle databases. It acts as a proxy that facilitates client application interaction with a database while also providing some advantages in comparison to direct loading of dependencies and connection to the database. The advantages offered by this system are:

- **Security:** Database credentials are stored on the server, eliminating the need for clients to handle or transmit credentials.
- **Connection Management:** Connections are managed by TStore, ensuring that only the necessary number of connections are open at any time. Open connections are automatically closed in the event of application crashes.
- **Fault Tolerance:** TStore decouples the database from the application, allowing fault tolerance measures to be implemented. The deployment of updates and bug fixes can be simplified and accelerated, as the OCCI library is updated using the TStore service.

2.1 Architecture

The TStore application runs as a XDAQ service, which provides the primary interface to communicate with the database. Multiple libraries implement metadata and database interaction and client helper

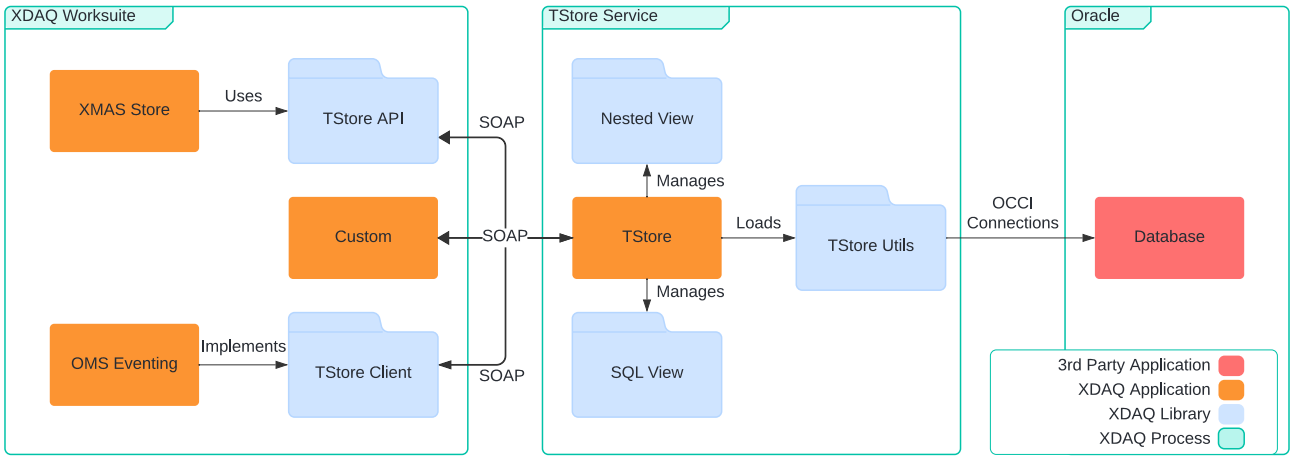


Figure 1. The architecture is based on a three-tier model: Client, TStore service, and Oracle database. Two clients that use the TStore service are shown as examples in the client layer.

functionality. Communication between TStore and XDAQ applications is handled using SOAP[1] (Simple Object Access Protocol) messages. Figure 1 illustrates the architecture and interaction of the components within the TStore system.

Two libraries simplify the process of communicating with the service. "TStore Client" is a low-level library used for attaching data to SOAP messages and simplifying access. In this case, the application is responsible for constructing, sending, and handling errors in SOAP messages. This library is used in systems such as "OMS Eventing", which is responsible for storing downtime and lumisection data [4]. The "TStore API" library is a higher-level library that automatically constructs certain types of SOAP messages. Message construction is simplified, but the application remains responsible for sending the messages and managing errors. The "XMAS Store", which is used to store flashlists in the database, utilizes this library. Alternatively, developers have the option to bypass these libraries and implement direct communication with the TStore, provided that the application is written in C++ due to the binary attachment requirements.

Once a request is sent to TStore, the service handles it by forwarding it to the database using the "TStore Utils" library. It leverages the C++ library OCCI[2] (Oracle C++ Call Interface) to manage communication with an Oracle database. Upon receiving a SOAP message, the request is checked against the metadata and is then converted into an OCCI call, which is passed to the Oracle database for execution.

2.2 Metadata Information

In order to control access to specific tables and databases, TStore uses metadata files, known as views. There are two types of views: "SQL View" and "Nested View", which define the types of requests allowed on different tables.

- **SQL View:** Designed for data manipulation and querying on an individual table. While it can encompass multiple tables, it does not support relationships between them. This type is typically used by client applications to access table data for configuration or monitoring purposes.
- **Nested View:** Capable of performing more complex operations involving multiple tables and their relations. The configuration includes a parent table and specifies the maximum depth of relational

[1] www.w3.org/TR/soap12
[2] www.oracle.com/database/technologies/appdev/oci.html

tables to consider. Unlike the SQL View, the Nested View has no restrictions on table usage and allows full access to all tables of the database.

This description shows that Nested Views provide broader access privileges and fewer restrictions compared to SQL Views. In contrast, SQL Views specify permitted calls and restrict access to certain tables. A table can be connected to a view by adding it as an XML element, as demonstrated in listing 1. This example shows a database table with all its columns. It is allowed, but not required, to include column types and primary keys.

```
<tstore:table name="CMS_OMS_AGG.DOWNTIMES_EVENTING">
   <tstore:column name="ID"/>
   <tstore:column name="START_TIME"/>
   <tstore:column name="START_FILL"/>
   <tstore:column name="START_RUN_NUMBER"/>
   <tstore:column name="START_LS"/>
   <tstore:column name="START_NIBBLE"/>
   <tstore:column name="STOP_TIME"/>
   <tstore:column name="STOP_FILL"/>
   <tstore:column name="STOP_RUN_NUMBER"/>
   <tstore:column name="STOP_LS"/>
   <tstore:column name="STOP_NIBBLE"/>
</tstore:table>
```

Listing 1. A table definition in a SQL view used by the OMS Eventing application of CMS allowing to query this table using TStore.

To support database DML (Data Manipulation Language) operations, such as Insert, Update, and Delete, one or more of the elements presented in listing 2 can be incorporated into the view.

```
<sql:insert name="insertNewDowntime">
 <sql:table name="CMS_OMS_AGG.DOWNTIMES_EVENTING"/>
</sql:insert>

<sql:update name="updateDowntime">
 <sql:table name="CMS_OMS_AGG.DOWNTIMES_EVENTING"/>
</sql:update>

<sql:delete name="deleteDowntime">
 <sql:table name="CMS_OMS_AGG.DOWNTIMES_EVENTING">
    <tstore:column name="ID"/>
 </sql:table>
</sql:delete>
```

Listing 2. DML calls stored in a SQL view used by OMS Eventing in the CMS experiment, which allow to execute insert, update and delete calls on the table DOWNTIMES_EVENTING.

By specifying the name of the SQL call, the correct request is executed on the database. It is also possible to apply column filters by including only the relevant columns in the table definition, as demonstrated in the delete example of listing 2, line 9. Similarly to the DML calls, query data can be requested from a predefined table. More complex queries can be built by filling SQL queries with dynamic parameters during the SOAP request, such as in the example shown in listing 3 that is used in the CMS experiment.

```
<sql:query name="selectRowByPK">
   <sql:parameter name="fullTableName" ><![CDATA[]]> </sql:parameter>
   <sql:parameter name="PKColumnName" ><![CDATA[]]> </sql:parameter>
   <sql:parameter name="PKValue" bind="yes"> <![CDATA[]]> </sql:parameter>
   <![CDATA[SELECT *
              FROM $fullTableName
             WHERE $PKColumnName=$PKValue
   ]]>
</sql:query>
```

Listing 3. Dynamic query builder used by the level 1 trigger of the CMS experiment to load configurations.

2.3 SOAP API Definition

In the XDAQ 1st generation, the communication between application and services was implemented using SOAP messages. It is a lightweight protocol that can be carried over HTTP (Hypertext Transfer Protocol). These messages have an XML format, with the root element being an envelope. This envelope contains all the information about the message and defines the XML as a SOAP message. In the envelope, the header element can be found, which can include additional information like authentication credentials. In the body element the actual message is stored.

When working with SOAP, functionality of a server can be exposed through services which handle the XML messages and attachment provided. The SOAP API of TStore provides three groups of requests, which are discussed separately.

- **Connection to database:** Connect, Disconnect, Renew
- **SQL:** Query, Definition, Insert, Update, Delete and Truncate
- **Admin:** Get Views, Get Config, Set Config, Update DB, Add Table, Remove Table, Destroy, Add View, Delete View

The **Connection** group is responsible for handling the most basic requests from TStore clients, including establishing and terminating connections to the Oracle database. The third request is necessary due to the manner in which TStore manages connections. When establishing a connection to the database, the client is required to specify a timeout period. Following the expiration of this timeout, the connection will be automatically terminated to prevent orphaned connections in the event of an application failure. The "Renew" request must be sent at regular intervals to avoid the timeout of a still active connection.

The group of **SQL** requests includes typical database operations such as querying, inserting, updating, and deleting. Table data exchanged in these requests is stored in a C++ table object and attached to the HTTP message. TStore also supports requesting a table definition, which returns a C++ object of the table without any data. This can streamline the data entry process as it eliminates the need to manually built the table object. By using the “Truncate” request, the client is able to remove all existing data from the table and reset it to its original state.

The **Administrative** requests can be classified into two categories, those requiring database access and those requiring metadata access. Requests that access only metadata files permit the client to add, delete, or modify views or change their configuration information. The information required for these requests is included in the XML body of the SOAP message, eliminating the need for an attachment. It is possible to add new tables or delete existing ones by accessing the database. Altering a table directly via the SOAP interface is not supported. The communication of request and response for the creation of multiple tables can be found in listings 4 and 5. The full table information is appended to the request as a binary object, simplifying the request body.

```
<env:Envelope env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
       xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
   <env:Header/>
   <env:Body>
      <tstoresoap:addTable tstoresoap:connectionID="0x7fd4bc1a6db0"
            xmlns:tstoresoap="http://xdaq.web.cern.ch/xdaq/xsd/2006/tstore-10.xsd"
            xmlns:viewtype="urn:tstore-view-SQL">
         <tstoresoap:table key="column5primary" name="test_table2"/>
         <tstoresoap:table key="column2primary, column7primary" name="test_table1"/>
         <tstoresoap:table key="column1primary" name="test_table0"/>
      </tstoresoap:addTable>
   </env:Body>
</env:Envelope>
```

Listing 4. Example SOAP request creating three tables in the database. It includes a binary attachment that is not shown here.

The response contains two pieces of information for every table. Firstly, that the table was added successfully to the database and secondly, that it was included to the view configuration. For the view configuration, the table definition is translated to XML and added for verification.

```
<env:Envelope env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
   <env:Header/>
   <env:Body>
      <tstoresoap:addTableResponse xmlns:tstore="urn:xdaq-tstore:1.0"
            xmlns:tstoresoap="http://xdaq.web.cern.ch/xdaq/xsd/2006/tstore-10.xsd">

         <tstoresoap:addTableToDatabase key="column5primary" table="test_table0"/>
         <tstoresoap:addTableToConfig table="test_table0">
            <tstore:table key="column5primary" name="test_table0">
               <tstore:column name="column0" type="time"/>
               <tstore:column name="column1" type="time"/>
               <tstore:column name="column2" type="string"/>
               <tstore:column name="column3" type="string"/>
               <tstore:column name="column4" type="double"/>
               <tstore:column name="column5primary" type="unsigned int"/>
            </tstore:table>
         </tstoresoap:addTableToConfig>

         <tstoresoap:addTableToDatabase key="column0primary" table="test_table1"/>
         <tstoresoap:addTableToConfig table="test_table1">
            <tstore:table key="column0primary" name="test_table1">
               <tstore:column name="column0primary" type="string"/>
            </tstore:table>
         </tstoresoap:addTableToConfig>

         <tstoresoap:addTableToDatabase
            key="column1primary,column0primary"
            table="test_table2"/>
         <tstoresoap:addTableToConfig table="test_table2">
            <tstore:table key="column1primary,column0primary" name="test_table2">
               <tstore:column name="column0primary" type="unsigned short"/>
               <tstore:column name="column1primary" type="int"/>
            </tstore:table>
         </tstoresoap:addTableToConfig>

      </tstoresoap:addTableResponse>
   </env:Body>
</env:Envelope>
```

Listing 5. Example SOAP response after creating three tables in the database and storing them to the metadata configuration.

The final SOAP request in the administration group that also requires a database connection is the "Update DB" request. This is necessary if the view configuration has been changed, either locally or through the "Set Config" request, or if the tables used in the database have been modified. This can result in the generation of inconsistent information. To resolve this, the "Update DB" request can be employed with the option of updating the configuration, the database, or both.

2.4 Usage in the CMS Experiment

When upgrading a service used by multiple applications, it is essential to analyse the service's users and identify the functionalities that are most important to them. To obtain data on the usage of TStore, the XaaS configurations of the subsystems of the run control of the CMS experiment and their instances operating at the experiment site are analysed [3]. Table 1 presents a comprehensive list of all subsystems, with the primary applications highlighted in bold. The first subtable depicts the existence of a XaaS configuration and its utilization in the production environment for running the subsystem. The second subtable illustrates the usage of TStore by the subsystem, either directly through the creation of a view and communication with TStore, or indirectly by loading the XMAS Store. The star in the Nested View column indicates the indirect usage of TStore.

	XaaS		TStore		
Sub-System	**GitLab**	**XaaS Used**	**SQL View**	**Nested View**	**XMAS Store**
PIXEL	✓	✓	-	★	✓
TRACKER	✓	✓	-	-	-
ES	✓	✓	✓	-	-
ECAL	✓	✓	✓	-	-
HCAL	✓	✓	-	-	-
DT	✓	✓	-	-	-
GEM	-	-	-	-	-
CSC	✓	✓	✓	-	-
RPC	✓	✓	✓	-	-
CTTPS TOT	-	-	-	-	-
CTTPS	✓	✓	-	-	-
TCDS	✓	✓	-	★	✓
L1 SCOUT	-	-	-	-	-
L1 TRIGGER	✓	✓	✓	★	✓
DAQ & OMS	✓	✓	✓	✓	✓
DQM	✓	-	-	-	-
DCS	✓	-	-	-	-
BRIL	✓	✓	-	★	✓

Table 1. The usage of TStore in the subsystems of the run control of the CMS experiment, as gathered from XaaS configurations and running instances at the experiment site.

From this table it can be extracted, that 9 out of the 13 primary systems to operate the CMS experiment are using TStore either directly or indirectly. The analysis of the view configurations also revealed that the RPC and L1 trigger subsystems use dynamic query building. Therefore, it is important to ensure that no functionality is lost when porting to the REST API.

2.5 Updating of OCCI Library

TStore was not moved to the last version of XDAQ because it was not compatible with the upgrades of this version. With this release the framework was migrated from CentOS 7[3] to AlmaLinux 9[4] and the OCCI (Oracle C++ Call Interface) library was initially incompatible with C++17 and therefore AlmaLinux 9. This issue led to TStore being excluded from the new release, and it remained unmaintained for several years. Recently, a new version of the OCCI library was released, finally making it compatible with the new environment. The task of upgrading TStore to this new version and ensuring the functionality of TStore and its associated test applications is discussed in this section.

When upgrading a critical database application, it is important to ensure that all components continue to work properly with the new infrastructure. Oracle OCCI serves as a critical API that allows C++ applications to interact with Oracle databases using an object-oriented approach, making it easier and more efficient to manage complex database operations. The transition from Oracle OCCI version 19.8 to version 23.4 includes a significant update to both the underlying database features and the capabilities of the API.

[3] www.centos.org
[4] www.almalinux.org

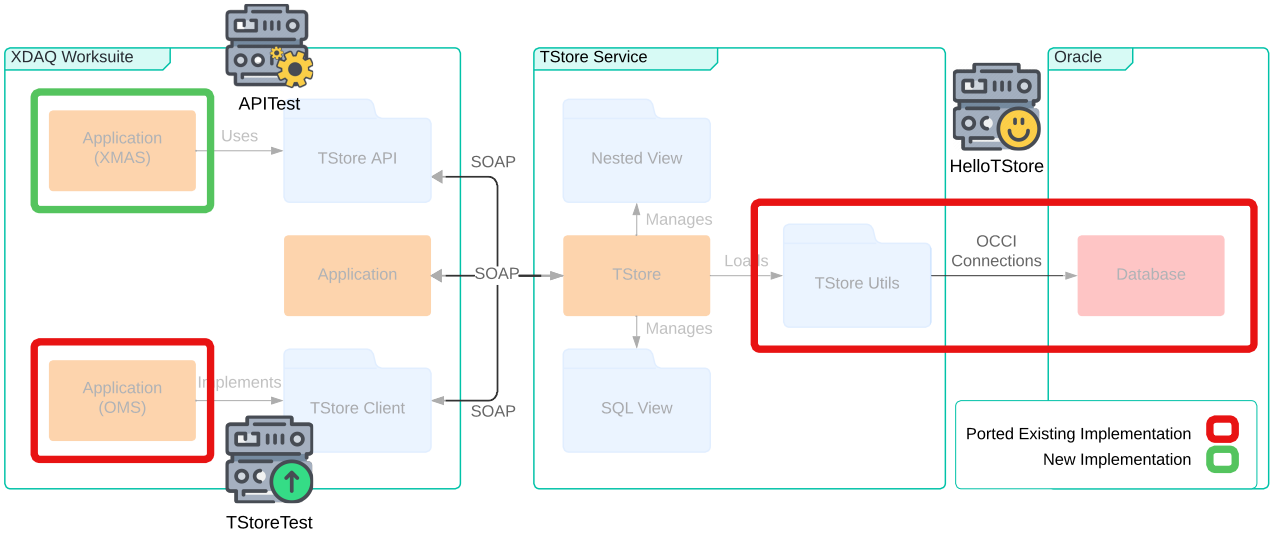


Figure 2. The TStore provides three test applications which are running with the current version. The TStoreTest and HelloTStore were already included in earlier version of the TStore, the APITest is a new implementation.

The upgrade of this library had to be done for the TStore, which is used by many XDAQ applications in the CMS experiment, but also in a service called SpotlightOCCI. This is a service that stores exception and alert information in the database, but does not use the TStore. Therefore, the same steps had to be performed and tested separately for this service.

2.6 Unit Testing

Two test applications were implemented in the past to facilitate thorough testing. A minimal configuration example called HelloTStore and a web client, TStoreTest, designed to evaluate all TStore functionalities. The existing documentation required some functionality to be inferred from the code. To enhance clarity and ease of understanding, application-specific documentation was added to the code. Until now, no test application was provided for the TStore API. This had to be developed based on the implementation of the TStoreTest. The areas to be tested by each application can be seen in figure 2.

The HelloTStore provides a simple client, which is able to request a SQL query from a TStore connected to a database in the same XDAQ process. For this, a SQLView is loaded by the TStore service. This view doesn't contain any table, but a single query element, which requests the string 'Hello World' from a dummy table. By successfully executing this communication, the authentication of the server client connection and the connection to the database can be ensured.

The TStoreTest application is a client designed to rigorously evaluate the functionality of the TStore service, especially for applications implementing the TStoreClient. This application plays a crucial role in testing and validating the TStore in managing various data operations. To imitate a realistic environment, the TStore and the test application are run in separate processes. TStoreTest allows the interaction with the TStore service in a controlled setting, facilitating a comprehensive evaluation of the service's capabilities.

Similarly, the APITest implements all the SOAP message building functionality to be tested with the TStore using the TStoreAPI library. In this application, the focus is less on the TStore and more on the correct implementation of the message building functionality to ensure that all requests can be sent successfully.

After the initial validation of database connectivity and TStoreClient and TStoreAPI functionality, the evaluation was extended to include the XMAS Store and the SpotlightOCCI applications. A standalone XDAQ as a Service (XaaS) environment was deployed on two virtual machines running on AlmaLinux 9 to simulate operational conditions. In addition to the standard services, XaaS included the XMAS Store, the job control, and the SpotlightOCCI applications, where the transmission and storage of messages was tested. Specifically, it was validated that the data is accurately stored and retrieved from one machine to another using the XMAS Store and TStoreAPI. The integrity of these operations was verified by checking flash list statuses in the XMAS Store interface and ensuring that data was correctly inserted into the database tables. This thorough testing confirmed that the new Oracle OCCI release maintains compatibility and functionality across all integrated systems, supporting a seamless transition.

3 Designing the REST API for the High Luminosity LHC

REST (Representational State Transfer) provides an API to access data. In general, REST is not a protocol but defines an architecture style [5]. In this context, for the TStore service, the messages are converted from SOAP to an HTTP and JSON-based API architecture.

3.1 Specification

The goal is to standardize and simplify development requirements by using a specification that is already widely used. To stay consistent with other REST API implementations already in use in the CMS experiment, the JSON API specification was chosen [5].

The JSON API specification is a standardization for efficient data exchange between client and server that facilitates the REST API architecture and the JSON file type. It defines how requests, responses, and errors are structured and ensures consistent handling of resources and metadata. It also allows the construction of dynamic requests by filtering returned resources or their parameters. The custom implementation required for the TStore server and client communication can be minimized by adopting this specification.

3.2 General Structure

As a REST API is working in a resource based fashion, the functionality based requests of SOAP have to be restructured. To access the resource path, it is necessary to determine whether it is a request for data access or administration. When considering the admin request, it's also reasonable to split database and metadata access. This results in the following list of TStore operations allowed in the new REST API:

- **Data Access**: Query, Insert, Update, Delete, Truncate
- **Meta Access:** Get Views, Add View, Delete View, Get Config, Set Config, Get Tables, Table Definition
- **Table Control:** Add Table, Drop Table, Drop View, Update DB, Update Config

The connection to the database is orthogonal to the previous actions, and the following operations are available:

- **Connection:** Connect, Renew, Disconnect

Now that the TStore operations have been separated, the next step is to define the REST resources. Four resources are required to represent all levels of TStore operations: views, tables, rows, and connections. The view and table resources are used for creating and modifying the metadata as well

[5] www.jsonapi.org

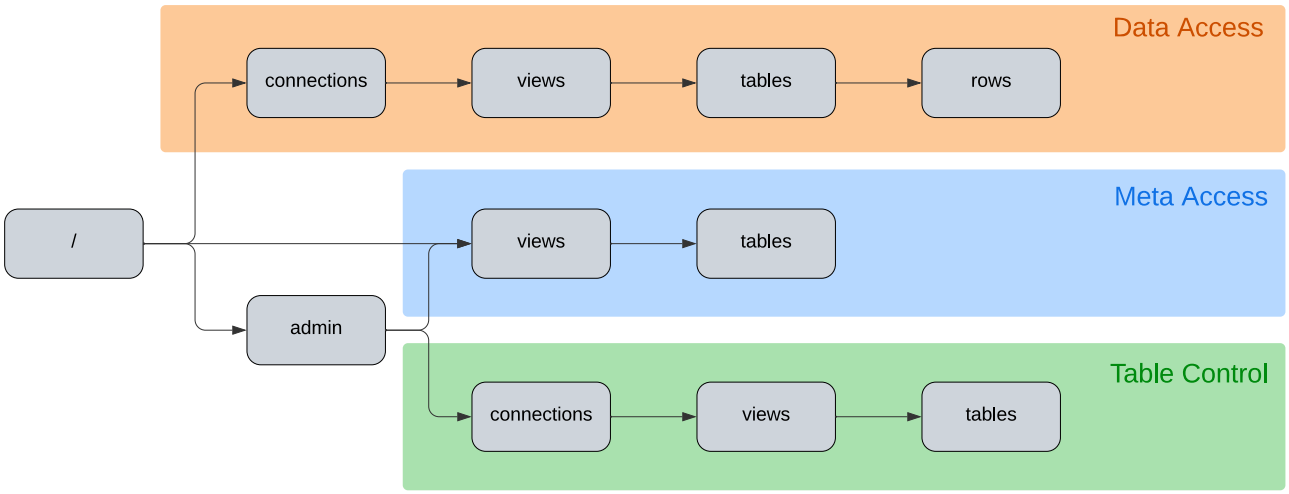


Figure 3. The hierarchical structure of the REST API for TStore, where the TStore operations are grouped by the three main types of data access and administration actions.

as the table definitions. The row resource is necessary for making changes to individual rows of a table, including inserting, updating, or deleting data. The connections resource provides access to information about active connections.

Using these resources, all requests can now be represented by a hierarchical tree structure, as shown in the figure 3. For Data Access and Table Control, the connections resource list is at the top, indicating that an open connection must be provided for these requests. Having the keyword 'admin' in the resource path allows access to the admin subtree and prevents accidental modification of tables or metadata.

3.3 Requests

In order to create all types of TStore operations using resources, it is essential to leverage HTTP methods that define specific actions on those resources. The core methods include:

- **GET** for retrieving information,
- **POST** for adding new resources,
- **PATCH** for updating existing resources,
- **DELETE** for removing resources.

These methods provide a flexible framework to cover the full spectrum of TStore operations. The structure of these HTTP requests is shown in figure 4, demonstrating how it relies on the hierarchical structure of resources to create simple yet functional URL paths. This approach serves as the foundation for redefining and explaining TStore operations, which are outlined in detail in the subsequent sections.

The connection action must be performed for data access and table control requests when communicating with the TStore service. Its HTTP requests are quite simple, as can be seen in the table 2. It is necessary to create, renew and delete a connection, but it should not be allowed to return a resource list, as this could be a security vulnerability. Creating a new connection is done by sending a request directly to the resource list. To renew or remove a connection, the connection ID must be added to the URL path of the HTTP request. The timeout parameter is added to connection and renewal requests, to allow the TStore to automatically close the connection if it is not renewed.

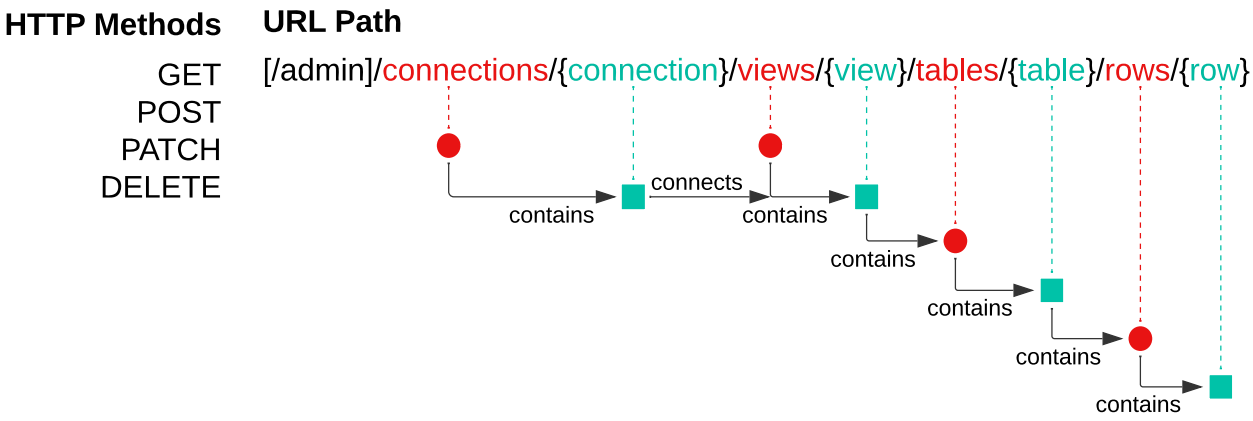


Figure 4. Combining the hierarchical resource structure of the REST API for TStore with HTTP methods allows representing all TStore operations in a simple, meaningful way.

Operation	HTTP Method	URL Path
Connect	POST + JSON	/connections
Renew	PATCH + JSON	/connections/:connectionID
Disconnect	DELETE	/connections/:connectionID

Table 2. The HTTP requests to open, close and renew a connection to the Oracle database while using the TStore service.

Operation	HTTP Method	URL Path
Insert	POST + JSON	/connections/:connectionID/views/:viewID/tables/:tableID/rows
Multiple	POST + JSON	/connections/:connectionID/views/:viewID/tables/:tableID
Update	PATCH + JSON	/connections/:connectionID/views/:viewID/tables/:tableID/rows/:rowID
Multiple	PATCH + JSON	/connections/:connectionID/views/:viewID/tables/:tableID
Delete	DELETE	/connections/:connectionID/views/:viewID/tables/:tableID/rows/:rowID
Multiple	DELETE	/connections/:connectionID/views/:viewID/tables/:tableID
Truncate	DELETE	/connections/:connectionID/views/:viewID/tables/:tableID
Query	GET	/connections/:connectionID/views/:viewID/tables/:tableID?...

Table 3. The HTTP requests to add, alter and delete rows of database tables that are connected to a specific view.

3.4 Data Access

The basic unit of the Data Access group is a row in the database tables. The TStore service identifies the correct database by adding the connection ID to the URL path. The view must also be specified, as multiple views may connect using the same schema and database. The next parameter to provide is the table name the request will interact with. It is possible to add or modify either a single row or multiple rows at once. The associated requests can be seen in the table 3.

The REST API allows data filtering for the query request by adding multiple parameters, some of which are listed in the table 4. The fields parameter defines the columns to be fetched from the database. The filter parameter can build complex concatenated conditions to restrict the rows returned to satisfy those conditions. The filter parameter design is based on the JSON API specification and

REST Parameter	SQL	Description
fields[tables]=column0,column1	SELECT	Filtering the columns added to the result table
filter[column1][eq]=5	WHERE	Filtering which rows to return from the table data
sort=-column1,column2	ORDER BY	Order the table data

Table 4. Multiple parameters can be added to a query request to dynamically adapt the SQL request that is extracted from it.

is also implemented in the OMS Aggregation API [6]. To sort the resulting data based on column values, the sort parameter can be used.

Which calls are allowed to be sent to a table is defined in the view configuration. During the porting of the API from SOAP to REST, the format of views will be changed from XML to YAML and table call information will be moved inside the table definition. Therefore, for each table the allowed calls can be set. It is also supported to include predefined SQL calls or dynamic queries to have backwards compatibility. An excerpt of a refactored example view can be seen in the listing 6.

```yaml
allowed_calls:
   query:
      all: true # a full table query is allowed
      specialQuery2:
         query: select column0, column1 from sql_table where column0 < 2000 order by column5
      specialQuery3:
         query: select $columns from sql_table where column0 < $val order by column5
         parameter:
            columns:
               default_value: column0, column1
            val:
               default_value: 2000
   insert: true
   update: false
   definition: true
   delete: false
   truncate: false
```

Listing 6. A section of the refactored view configuration defines which Data Access calls are allowed for a specific table in the view.

3.5 Meta Access

The meta access does not require a connection to the database, but since it contains some admin requests, the keyword "admin" must be added to the root of the URL path for these requests. It provides functionality to add, alter and delete view configurations, or to get a list of all associated tables of a view or its definition. Since the configuration is only at the view level, all configuration changes can be made by adding the view ID to the URL path of the request. To get all views loaded by the TStore, the resource list "views" can be requested with the GET HTTP method. Similarly, a list of all tables belonging to a view can be requested by sending a GET request to the tables resource list of a specific view. For receiving the table definition from the view configuration, a GET request on the table ID is used. All defined requests can be viewed in table 5.

3.6 Table Control

The Admin Table Control group allows the client applications to create tables and add them to an existing view, as well as drop tables from the view and database. This requires an open connection and the keyword “admin” at the root of the URL path. The “Update DB” and “Update Config” requests allow the client to synchronize the database and view configuration if they have become out of sync after a change to the configuration or some changes to the database tables. The two different

Operation	HTTP Method	URL Path
Get Views	GET	/admin/views
Add View	POST + JSON	/admin/views
Delete View	DELETE	/admin/views/:viewID
Get Config	GET	/admin/views/:viewID
Update Config	PATCH + JSON	/admin/views/:viewID
List of Tables	GET	/views/:viewID/tables
Table Definition	GET	/views/:viewID/tables/:tableID

Table 5. The HTTP requests to add, alter and delete view configurations, request a list of all associated tables of a view or a table definition.

Operation	HTTP Method	URL Path
Add Table	POST + JSON	/admin/connections/:connectionID/views/:viewID/tables
Drop Table	DELETE	/admin/connections/:connectionID/views/:viewID/tables/:table_name
Update DB	PATCH	/admin/connections/:connectionID/views/:viewID?mode=db
Update Config	PATCH	/admin/connections/:connectionID/views/:viewID

Table 6. The HTTP requests to add and remove tables from a view configuration and database, and to synchronize a view configuration with the database.

requests specify if the sync should be done in the direction of the database, "Update DB", or the view configuration, "Update Config". Even though no new information is added to the JSON body, the HTTP method PATH must be used because the view configuration may be changed. The detailed request definition can be found in table 6.

4 Results and Outlook

The porting of the current TStore implementation to the new OCCI version 23.4, with support for C++17, was successful and has been included in the latest baseline release of XDAQ. It also ensured that the test applications worked as expected and that the code changes did not alter their behaviour. The development of a new test application for the API library provided a comprehensive environment for testing the functionality of the full TStore system. By also providing a standalone XaaS, it simplifies local testing of the full production system.

The proposed REST API for TStore communication within the 2nd generation of XDAQ supports all SOAP TStore operations. The transition from SOAP-based communication to a RESTful architecture using the JSON API specification was outlined. This new API structure simplifies data management operations, standardizes request handling, and improves usability. Using the JSON format for data transport decouples the API from the C++ language, enabling clients to use different programming languages.

The next step in porting the TStore to a REST API is a feasibility study to assess the resource and time requirements for a full implementation. By implementing the widely used JSON API specification, the development effort can be reduced if suitable libraries or similar projects exist in C++. In addition, adopting the REST API implementation in the 2nd generation of XDAQ can further reduce the implementation effort. After this assessment, a prototype can be implemented to identify potential challenges and risks.

Acknowledgements

My sincere appreciation goes to my supervisor Andrea Petrucci for his guidance, mentorship, and support throughout the internship. His expertise and insights have been invaluable in shaping my understanding of the data acquisition process at CMS and the design of large scale software projects. I would also like to thank the Summer Student Team for planning such an enriching program. Attending the lectures and workshops helped me get insights of the experiments at CERN, as well as understand the importance of all the subsystems needed for a successful experiment.

References

[1] J. Gutleber, S. Murray, and L. Orsini, "Towards a homogeneous architecture for high-energy physics data acquisition systems", *Computer Physics Communications*, vol. 153, no. 2, pp. 155–163, 2003, ISSN: 0010-4655. DOI: https://doi.org/10.1016/S0010-4655(03)00161-9. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010465503001619.

[2] G. Bauer, U. Behrens, K. Biery, *et al.*, "Monitoring the CMS Data Acquisition System", CERN, Geneva, Tech. Rep., 2010. DOI: 10.1088/1742-6596/219/2/022042.

[3] G. Bauer, U. Behrens, M. Bowen, *et al.*, "Distributed error and alarm processing in the CMS data acquisition system", CERN, Geneva, Tech. Rep., 2012.

[4] V. Amoiridis *et al.*, "First year of experience with the new operational monitoring tool for data taking in CMS during Run 3", *EPJ Web Conf.*, vol. 295, p. 02 013, 2024. DOI: 10.1051/epjconf/202429502013.

[5] R. T. Fielding and R. N. Taylor, "Architectural styles and the design of network-based software architectures", AAI9980887, Ph.D. dissertation, University of California, Irvine, 2000, ISBN: 0599871180.

[6] C. Wernet, "Unifying access to data from heterogeneous sources through a RESTful API using an efficient and dynamic SQL-query builder", Presented 15 Dec 2017, Ph.D. dissertation, Hochschule, Eng. Econ., Karlsruhe, 2017.