Unifying access to data from heterogeneous sources through a RESTful API using an efficient and dynamic SQL-query builder

Bachelor Thesis of

Christian Wernet

at the Faculty of Computer Science and Business Information Systems
Subject Area Distributed Systems (VSY)

Reviewer: Prof. Dr. rer. nat. Christian Zirpins
Second reviewer: Prof. Dr. rer. nat. Oliver Waldhorst
Advisor: M. Sc. Andrea Petrucci

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Geneva, 18th November 2017

(Christian Wernet)
Abstract

The Web Based Monitoring (WBM) system of the Compact Muon Solenoid experiment at the CERN Large Hadron Collider was introduced in 2006. Over the last decade, this monitoring system has been extended and continuously updated with different technologies and approaches and has therefore nowadays become hard to maintain. To resolve these problems, the WBM system will be redesigned using state-of-the-art technologies to solve these issues and to prepare the system for the years of operation to come. This thesis describes the development of the first prototype as a RESTful API of the aggregation layer of this upgrade. The aggregation layer will be used to enable access to the stored data for the presentation layer of the upgrade.

Before describing the work on the prototype, an overview over the topics to form the basis is given. Subsequently, the task, the required outcome and prerequisites as well as the existing system were being analyzed to formulate adequate requirements for the prototype. In addition to this analysis, an evaluation of possible technologies was performed including a proof of concept implementation for each technology. After deciding on a technology, the API and the connection to the database was designed to fit the requirements. The design is then implemented with a special focus on two services. Firstly a developed converter service which is needed to be able to set the data from the different data sources in relation to each other. Secondly a custom query builder used to dynamically create database queries based on the request made to the API. To prove the capabilities of the developed prototype, a performance test was carried out as well as a functional evaluation to show that the requirements are fulfilled. The conclusion then sums up the work done with an emphasis on occurred problems like the non-heterogeneous data sources and the challenging database layout. In addition to this, the further development of the project is addressed by making suggestions on what could be considered interesting for future work.
Zusammenfassung


Acknowledgements

Firstly, I would like to thank Prof. Dr. Zirpins who supervised and reviewed my bachelor thesis for the helpful suggestions and constructive criticism during the preparation of this work.

I would like to thank Andrea Petrucci for his support and guidance during the work on this thesis. Without your extensive knowledge of the existing WBM, I would have been lost in its code more than one time. I highly appreciate all the little remarks and tips you share while visiting our office for a coffee.

A big thank you to Michael Lettrich and Nicholas Skram for being the best office mates I could ever hope for.

Furthermore, I would like to thank my Norwegian lunch group for all the great lunchtimes and other activities during my stay at CERN. Takk for alt!

Thanks to all my old friends in or outside university for the good times spent together.

A special thanks to my family for their love and support which allowed me to get to this point.
Contents

Abstract i

Zusammenfassung iii

List of Figures ix

List of Tables xi

List of Listings xiii

Acronyms xv

Glossary xvii

1. Introduction 1
   1.1. Motivation .................................................. 1
   1.2. Place of work .............................................. 2
   1.3. Structure of this thesis .................................. 3

2. Introduction to Web Services and data fetching 5
   2.1. Web Services ................................................ 5
       2.1.1. Remote Procedure Calls ............................... 6
       2.1.2. SOAP .................................................. 7
       2.1.3. REST .................................................. 10
   2.2. Dynamic Database Queries ............................... 13

3. Analysis 15
   3.1. Essential Services ......................................... 16
       3.1.1. Fill Summary ......................................... 17
       3.1.2. Fill Report ........................................... 17
       3.1.3. Run Summary ......................................... 17
       3.1.4. Run Report ........................................... 18
       3.1.5. L1 Report ............................................. 18
       3.1.6. L1 Algo Trigger Report ............................... 19
   3.2. Computer science context ............................... 19
   3.3. Technology evaluation .................................... 21
       3.3.1. GraphQL ............................................... 21
       3.3.2. Spring ............................................... 23
       3.3.3. Katharsis and Dropwizard ........................... 26
# List of Figures

2.1. Flow of a Remote Procedure Call (RPC) call ................................. 7  
2.2. Flow of a Simple Object Access Protocol (SOAP) request .................. 8  
2.3. Structure of a SOAP-Message with Envelope, Header and Body ............ 9  
2.4. The steps towards Representational State Transfer (REST) .................. 12  
3.1. The Fill Summary service with control and content portlet .................. 17  
3.2. The Run Summary service with control and content portlet ................. 18  
3.3. Architecture of a data warehouse ............................................. 20  
3.4. Histogram of the measured latency values .................................... 27  
3.5. Comparison of the filesizes containing the response of the different technolo-gies. ................................................................. 28  
4.1. The response object of the API ............................................... 35  
4.2. The metadata object of the response ....................................... 37  
5.1. Three-tier architecture ......................................................... 39  
5.2. All used technologies/libraries ............................................... 40  
5.3. UML diagram of the classes needed for the metadata ....................... 44  
6.1. The static plots plotted with data from the InstLumi and VacuumSummary resources ............................................................ 51  
6.2. The setup for the performance test in Point 5 ................................ 51  
6.3. Results of the performance test ............................................. 52
List of Tables

2. Time units ........................................... xviii

2.1. The most commonly used Hypertext Transfer Protocol (HTTP) verbs and their CRUD counterpart ........................................... 11

A.1. DAQ-Readout resource ........................................... 60
A.2. Metadata that must be included with each Run resource ........... 60
A.3. Metadata that must be included with each Fill resource ........... 60
A.4. Run resource ........................................... 61
A.5. Fill resource ........................................... 62
A.6. Lumisection resource ........................................... 63
A.7. Downtime resource ........................................... 63
A.8. L1ConfigurationKey resource ........................................... 64
A.9. L1AlgorithmTriggers resource ........................................... 64
A.10. L1TriggerRates resource ........................................... 65
A.11. L1OverallDeadTime resource ........................................... 66
A.12. L1BeamActiveDeadTime resource ........................................... 67
A.13. L1AlgorithmDetails resource ........................................... 68
A.14. InstLumi resource ........................................... 68
A.15. Era resource ........................................... 69
A.16. Sequence resource ........................................... 69
A.17. VacuumSummary resource ........................................... 69
## List of Listings

2.1. XML-RPC call ................................................................. 8  
2.2. XML-RPC response ......................................................... 8  
2.3. Example of a SOAP request sent over HTTP. ......................... 9  
2.4. Example of a SOAP response sent over HTTP. ...................... 9  
3.1. GraphQL schema .............................................................. 21  
3.2. GraphQL query to get some Eras ....................................... 22  
3.3. JSON response to the request in Listing 3.2 .......................... 23  
3.4. Class body of a Spring REST controller. ............................ 24  
3.5. Plain Old Java Object (POJO) of a Spring resource. ............... 24  
3.6. JSON response of Spring ................................................ 25  
3.7. JSON response of Katharsis ............................................ 26  
4.1. Data part of a response .................................................. 31  
5.1. POJO representing the basic Run resource ........................... 40  
5.2. The custom RunRepository interface ................................. 41  
5.3. Class body of the implementation of the RunRepository interface . 42  
5.4. POJO member to include metadata ..................................... 43  
5.5. POJO member to include a relation to the Fill resource .......... 45  
5.6. RelationshipRepository for the relation from Run to Fill ........ 45  
5.7. Inner SQL query for the fill_type_party2 attribute of the Run resource. 46  
A.1. Metadata part of a response ............................................ 59  
A.2. Queries to retrieve attributes of the Run resource as used in the WBM Part 1. 70  
A.3. Queries to retrieve attributes of the Run resource as used in the WBM Part 2. 71  
A.4. Queries to retrieve attributes of the Run resource as used in the WBM Part 3. 72  
A.5. Functions to retrieve attributes of the Run resource as used in the WBM Part 1. 72  
A.6. Functions to retrieve attributes of the Run resource as used in the WBM Part 2. 73  
A.7. Functions to retrieve attributes of the Run resource as used in the WBM Part 3. 74  
A.8. Functions to retrieve attributes of the Run resource as used in the WBM Part 4. 75
## Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALICE</td>
<td>A Large Ion Collider Experiment</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ATLAS</td>
<td>A Torodial LHC ApparatuS</td>
</tr>
<tr>
<td>BRIL</td>
<td>Beam Radiation Instrumentation and Luminosity</td>
</tr>
<tr>
<td>CERN</td>
<td>European Organization for Nuclear Research</td>
</tr>
<tr>
<td>CMS</td>
<td>Compact Muon Solenoid</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>CRUD</td>
<td>Create, Read, Update, Delete</td>
</tr>
<tr>
<td>DAQ</td>
<td>Data Acquisition</td>
</tr>
<tr>
<td>DCOM</td>
<td>Distributed Component Object Model</td>
</tr>
<tr>
<td>DIP</td>
<td>Data Interchange Protocol</td>
</tr>
<tr>
<td>EP</td>
<td>Experimental Physics</td>
</tr>
<tr>
<td>ETL</td>
<td>Extract, Transform, Load</td>
</tr>
<tr>
<td>FED</td>
<td>Front-End Driver</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HATEOAS</td>
<td>Hypermedia as the Engine of Application State</td>
</tr>
<tr>
<td>HEP</td>
<td>High-energy physics</td>
</tr>
<tr>
<td>HLT</td>
<td>High-Level Trigger</td>
</tr>
<tr>
<td>HTML</td>
<td>HyperText Markup Language</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>IDL</td>
<td>Interface Definition Language</td>
</tr>
<tr>
<td>JDBC</td>
<td>Java Database Connectivity</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>LHC</td>
<td>Large Hadron Collider</td>
</tr>
<tr>
<td>LHCb</td>
<td>Large Hadron Collider beauty</td>
</tr>
<tr>
<td>OLAP</td>
<td>On-Line Analytical Processing</td>
</tr>
<tr>
<td>Acronyms</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>ORM</td>
<td>Object-Relation Mapping</td>
</tr>
<tr>
<td>POJO</td>
<td>Plain Old Java Object</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation</td>
</tr>
<tr>
<td>ROLAP</td>
<td>Relational On-Line Analytical Processing (OLAP)</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>UCP</td>
<td>Universal Connection Pool</td>
</tr>
<tr>
<td>UDDI</td>
<td>Universal Description Discovery and Integration</td>
</tr>
<tr>
<td>URI</td>
<td>Uniform Resource Identifier</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>WBM</td>
<td>Web Based Monitoring</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Services Description Language</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
</tbody>
</table>
**Glossary**

**API**  An Application Programming Interface (API) is a set of subroutine definitions, protocols, and tools for building application software. In general terms, it is a set of clearly defined methods of communication between various software components.

**BRIL**  The Beam Radiation Instrumentation and Luminosity (BRIL) group “is responsible for measuring [the] luminosity, machine induced background (MIB) and beam timing. [They] also provide active protection in case of intense beam loss events and give various inputs to the CMS trigger system.” [Gut16]

**DIP**  Data Interchange Protocol (DIP) is a system which allows relatively small amounts of real-time data to be exchanged between very loosely coupled heterogeneous systems. These systems do not need very low latency. The data is assumed to be mostly summarised data rather than low-level parameters from the individual systems, i.e. cooling plant status rather than the opening level of a particular valve. At CERN, DIP is used to exchange data between the LHC machine and the different experiments.

**Downtime**  A downtime is a period of time where a HEP detector does not take data.

**Era**  An era is a time period introduced by the CMS experiment to split the period of data taking by year, LHC type of run, etc.

**Event**  One collision of protons inside the CMS experiment is called event.

**FED**  Analogue or digital data from CMS sub-systems are transmitted to Front-End Driver (FED) modules over optical data links. The FED modules provide the formatting and local buffering of data fragments and handle the interface protocols for the control and readout buses. [LHC94]

**Fill**  A time period defined by the LHC. A Fill starts usually with the injection of the two beams into the LHC and ends with the declaration of the next Fill after losing or dumping of beams. This means, that there are three stages in every Fill:

- Injection and acceleration of the beams
- Collisions of the beams
- Dumping of the beams and more or less time until the next injection

---

1[https://readthedocs.web.cern.ch/display/ICKB/DIP+and+DIM]
HATEOAS  Hypermedia as the Engine of Application State (HATEOAS) is a constraint of the REST application architecture. A hypermedia-driven site provides information to navigate the REST interface dynamically by including hypermedia links with the responses\cite{Piv17b}.

JDBC  Java Database Connectivity (JDBC) is an API for the programming language Java, which defines how a client may access a database.

JSON  JavaScript Object Notation (JSON) is a lightweight data-interchange format. It is human read- and writeable. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language. [ECM13]

L1 Trigger  Level 1 of the trigger is an extremely fast and wholly automatic process that looks for simple signs of interesting physics, e.g. particles with a large amount of energy or in unusual combinations. This way, the trigger selects the best 100,000 events or “issues” each second from the billion available. See also chapter 8 of [Cha+08]

Luminosity  Luminosity is the number of particles per unit area per unit time times the opacity of the target, usually expressed in cm$^{-2}$s$^{-1}$ or b$^{-1}$s$^{-1}$. The integrated luminosity is the integral of the luminosity with respect to time. The luminosity is an important value to characterize the performance of an accelerator.\cite{Dyk06}

Lumisection  A lumisection is a sub-section of a Run at the CMS experiment. The duration of a lumisection is $\approx 23.3$ seconds (see table 2)

<table>
<thead>
<tr>
<th>Unit</th>
<th>Value</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 orbit</td>
<td>11245 Hz</td>
<td>1 / 11245 s</td>
</tr>
<tr>
<td>1 nibble</td>
<td>4096 orbits</td>
<td>$\approx 0.36425077812361$ s</td>
</tr>
<tr>
<td>1 lumisection</td>
<td>64 nibbles</td>
<td>$\approx 23.312049799911072$ s</td>
</tr>
</tbody>
</table>

Table 2.: Time units\cite{Tha16}

Non-physics data  Non-physics data is information, that “describes” the conditions under which an event took place. This includes e.g. data about the magnetic field, the beam conditions, the Luminosity, etc.

Resource  A resource is “[t]he key abstraction of information in REST”\cite{Fie00}. It is a unit of a REST API that one can call. A resource comprises an HTTP verb and a URI path that is subordinate to the context root of the API. By configuring the resource, you define how the API is exposed to your developers.

Run  A period of data taking at the CMS experiment.
1. Introduction

This first chapter is divided in three parts: The motivation describes the background of the project of this thesis and sets it into a bigger environment. The second part introduces the workplace where this thesis was written. To conclude the introduction, the third part outlines the structure of this thesis.

1.1. Motivation

Monitoring or Condition Monitoring is defined as “the process of monitoring one or more parameters of machinery for a significant change in the machine parameter(s) that is indicative of a current or developing condition (e.g. failure, fault, [proper working condition] etc.)” by Glomann et al. in [GYN09].

In order to recognize such a change, the monitored parameters have to be displayed in some way to the operator(s) of the machine or analysed by an intelligent system, e.g. a neural network, that will raise an alarm if there is an anomaly in the system. If said machine is producing data as a main result of its operation, it is possible to correlate the output data with the monitoring data taken at the same moment.

In case of the Compact Muon Solenoid (CMS) experiment at the Large Hadron Collider (LHC) at the European Organization for Nuclear Research (CERN) the parameters to be monitored are not only coming from the experiment itself but also from different other parts of the LHC. These include for example the beam condition which is coming from the Run Operation of the LHC. All these parameters are then stored in relational databases from where they can be accessed.

One software to access this recorded data is the current Web Based Monitoring (WBM)\(^1\) system. It was first introduced in 2006 to help with the monitoring of the CMS experiment in its commissioning phase and later during the run of the LHC. Since then the WBM has been extended and updated many times in the last decade. Nowadays it has become hard to maintain due to the amount of different technologies (languages, frameworks and libraries) used as well as the fact that the majority of the developers are not at CERN any more. This is aggravated by the mostly incomplete or outdated documentation of the different parts of the WBM.

In 2016 it was therefore decided to develop a new system that should serve as upgrade to the current WBM. It should be built with state-of-the-art technologies, libraries and techniques to last again for at least another decade.

Being part of this upgrade project, the aim of this thesis is to design and develop a prototype of a RESTful Application Programming Interface (API) which will be used by a web monitoring front-end and other clients that require access to the monitored data from

\(^1\)For more information on the WBM see the paper [Bad+14] by Badgett et al.
the database. The main goal is to provide a simple interface to the different subsystems and researchers of CMS. Other goals are to find a way to generate queries to the database based on the request to the API. Furthermore, we need to find a way to correlate data from different sources. An approach to do this would be creating a converter service to convert the different appearing time units.

### 1.2. Place of work

This thesis was written during a Technical Student internship at CERN on the Franco-Swiss border in Meyrin, Switzerland near Geneva. CERN was founded in 1954 as recommended before by the Conseil Européen pour la Recherche Nucléaire "following the example of international organizations"[CER17a]. The goal was to "unite European scientists"[CER17a] and also — from a more economic point of view — "to share the increasing costs of nuclear physics facilities"[CER17a]. Today, CERN houses the world's largest particle accelerator — the LHC — and around 16,868 scientists, employees and students are working on the various projects and experiments, either being on site in Switzerland or at collaborating universities, laboratories, institutes, etc. all around the world. The LHC accelerates two beams of proton clouds to 99.9999991 % of the speed of light and loads them with an energy of up to 13 TeV. Finally, the beams are slightly bend to collide inside the four big experiments along the LHC:

- the two general purpose detectors A Torodial LHC ApparatuS (ATLAS) and CMS which are used to investigate a large range of physics,
- A Large Ion Collider Experiment (ALICE), which is designed to detect so called quark-gluon plasma and
- the Large Hadron Collider beauty (LHCb) which is specialized on the investigation of the minimal differences between matter and antimatter by studying the so called b(eauty) quark [CER17b].

In letting the proton clouds collide, the LHC creates conditions such as those that existed shortly after the Big Bang in order to gain an insight into the origin of our universe. The data taken of the collisions by the four experiments is then used to do research in fundamental particle physics.

The internship was hosted at the CMS Data Acquisition (DAQ) & Trigger Group of the Experimental Physics (EP) department. The group is responsible for the data acquisition at the CMS experiment. This includes the so called Trigger system which is used to select around 1,000 events of the 40 million that are taking place inside the CMS experiment every second when it is running at full capacity. The selection is done by two systems: the Level 1 (hardware) and the High Level Triggers (software).

---

2The numbers are from 2016; see CERN's Personal Statistics of 2016 under [Dep17] for a more detailed view.

3The electronvolt (symbol eV) is a non-SI unit of energy equal to $\approx 1.6 \times 10^{-19}$ joules [MW08].
Furthermore, the DAQ Group is also in charge of collecting the non-physics data which is then stored in a database and made accessible through the current WBM as described in chapter 1.1.

As a third responsibility, the DAQ Group operates and maintains the server farm at Point 5 in Cessy (France), the home of the CMS experiment.

1.3. Structure of this thesis

After describing the motivation behind this thesis and giving a quick overview over the place of work where it was written, we want to give an overview of the remaining thesis. Chapter 2 introduces topics that form the basis of the work. Chapter 3 gathers the requirements for the work, sets the whole task in a computer science context and evaluates different technologies for their usability in the given task. In chapter 4 the API and the software in general are designed, before chapter 5 describes its implementation. Afterwards, chapter 6 evaluates the implemented software functional and performs a performance test. Chapter 7 then gives a final overview over the work, discusses general conclusion and gives an outlook on possible further developments of the project.

\footnote{For a more detailed description of Triggering and Data Acquisition see \url{https://cms.web.cern.ch/news/triggering-and-data-acquisition}.}
2. Introduction to Web Services and data fetching

“A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.”[Web04]

This is the definition of a web service used by the W3C Web Services Architecture Working Group in 2004. In this chapter, we want to explain what a web service is, how it has evolved over the years from Remote Procedure Calls (RPCs) over the Simple Object Access Protocol (SOAP) protocol to the Representational State Transfer (REST) architecture. Additionally, we want to describe how dynamic database queries can be introduced to an application.

2.1. Web Services

A web service is a software application that accepts requests, processes them, and returns a response. The evolved over time from the inter-process communication first in only one machine (using shared memory and pipes) and then between multiple machines over the network (RPC). In contrast to being inaccessible and technology-bound, the key features of a web service are the following:

- they are platform independent — web services use several open standards like Hypertext Transfer Protocol (HTTP), eXtensible Markup Language (XML) and JSON in order to be accessible from clients using different programming languages on different platforms;
- they are self-contained — web services are not depending on additional software, one only needs a programming language with support for XML/JSON and an HTTP client to start using them;
- they are self-describing;
- they are modular — web services can be aggregated/“stacked” in order to form a more complex web service.
In general, web services have two use-cases:

- Reuse application contents in different applications without using libraries. So called micro services can for example deliver a weather forecast. This functionality can then be used by other applications which want to include this functionality without having to implement it themselves or using a library;

- Distributed Computing.

Nowadays, the main goal of a web service is to provide a way of interaction between a web application and human clients, Graphical User Interfaces (GUIs), like portlets etc. The following subsections aim to describe the history and therefore evolution of web services.

### 2.1.1. Remote Procedure Calls

A Remote Procedure Call (RPC) is a request-response protocol. These protocols appeared first in the days of early distributed computing in the end of the 1960s. After the first ideas in the 1970s, the first implementations of RPCs were coming up in the early 1980s. The idea behind a RPC is to “make the process of executing code on a remote machine as simple and straightforward as calling a local function” [AA15]. In order to be able to use RPCs, one first has to define the set of functions that will be able to be called remotely. This is done by defining an interface and therefore using the Interface Definition Language (IDL). This interface is then processed by an RPC compiler/stub generator which generates a server skeleton and a client stub. Both of them contain all of the functions specified in the interface. The software on the server side has then to implement the functions specified in this interface so that the skeleton is able to call them. In order to use these functions, a client program would call simply call them on the client stub. Inside the client stub, all the functions contain everything needed to make the RPC work (see also fig. 2.1):

- marshal the arguments (= serialization of the message);
- send the message to the server;
- wait for the reply;
- unmarshal the response (= deserialization);
- return to the caller.

On the server side, the generated server stub does similar things:

- unmarshal the message;
- call the actual function;
- marshal the result;
- send the reply.
In the late 1990s, several technologies extended the concept of RPCs in allowing them between different systems; e.g.:

- the Common Object Request Broker Architecture (CORBA)\(^1\), which introduced a language and platform neutral specification for RPCs;
- the Remote Method Invocation (RMI)\(^2\), which offered something similar but was restricted to Java;
- the Distributed Component Object Model (DCOM)\(^3\), which was only usable between native Windows programs.

In 1998, the XML-RPC protocol was created by Dave Winer of UserLandPrograms in collaboration with Microsoft [Win99] [Box01]. The new and interesting part of it was that it uses the XML format to encode the calls and relies on HTTP to transport them. These points resolved the biggest issues RPC had so far: the XML provided for the first time language neutrality and the use of HTTP instead of proprietary protocols made it easier to open ports in firewalls to communicate with remote servers. Additionally, the protocol itself was very simple since only a few data types and commands were defined and became therefore quite popular. [T07]

An example of response and answer in XML-RPC is shown in listing 2.1 and listing 2.2.

### 2.1.2. SOAP

The SOAP protocol is the evolution of the XML-RPC specification. In comparison to XML-RPC, SOAP offers more structure and support for data types and operations semantic. It has three major characteristics:

1. see [http://www.corba.org/](http://www.corba.org/)
2. For more information see for example [https://docs.oracle.com/javase/tutorial/rmi/index.html](https://docs.oracle.com/javase/tutorial/rmi/index.html)
3. For more information see for example this explanation by Microsoft: [https://technet.microsoft.com/en-us/library/cc958799.aspx](https://technet.microsoft.com/en-us/library/cc958799.aspx)
2. Introduction to Web Services and data fetching

1. extensibility — it allows to be extended by additional parts, e.g. for security;
2. neutrality — it can operate over any protocol, e.g. HTTP, TCP, etc.;
3. independence — it allows for any programming model.

Shortly after it’s introduction, SOAP became the base layer of a more complex set of web services which arose due to the need for more automation in the field. This included for example message parsing, code generation or the discovery of web services on a network. Two additional languages were developed to service these needs:

- Web Services Description Language (WSDL) to describe a web services interface;
- Universal Description Discovery and Integration (UDDI) to register and find web services on the Web.

This stack of SOAP, WSDL and UDDI was commonly referred to as web service during the 2000 years.

The communication with a SOAP-based web service works in 4 steps (see figure fig. 2.2):

1. The service provider publishes interface definition via WSDL to the service registry (UDDI);
2.1. Web Services

Figure 2.3.: Structure of a SOAP-Message with Envelope, Header and Body; taken from https://commons.wikimedia.org/wiki/File:SOAP.svg in November 2017

2. The consumer looks up the service and downloads the WSDL description of it;

3. The consumer sends a SOAP request (a XML document consisting of a SOAP envelope containing an optional SOAP header and a required SOAP body which could additionally contain a so called “Fault” element to indicate error messages) to the service provider (see fig. 2.3);

4. After processing the request, the service provider sends a SOAP response to the consumer.

Listing 2.3 and listing 2.4 show examples of a SOAP communication\(^4\). Due to amongst others the verbosity of the protocol, in certain cases, SOAP messages were found to be 2-4 times lager than other protocols (see [KS11]), and the slow parsing speed of XML, the domination in the wide field of web services led to other technologies which use less verbose protocols, for example REST.

```
POST /Quotation HTTP/1.0
Host: www.xyz.org
Content-Type: text/xml; charset = utf-8
Content-Length: nnn
<?xml version = "1.0"?><SOAP-ENV:Envelope
xmlns:SOAP-ENV = "http://www.w3.org/2001/12/soap-envelope"
xmlns:encodingStyle = "http://www.w3.org/2001/12/soap-encoding">
    <SOAP-ENV:Body xmlns:m = "http://www.xyz.org/quotations">
        <m:GetQuotation>
            <m:QuotationsName>
                MiscroSoft
            </m:QuotationsName>
        </m:GetQuotation>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 2.3.: Example of a SOAP request sent over HTTP.

```
HTTP/1.0 200 OK
Content-Type: text/xml; charset = utf-8
Content-Length: nnn
<?xml version = "1.0"?><SOAP-ENV:Envelope
xmlns:SOAP-ENV = "http://www.w3.org/2001/12/soap-envelope"
xmlns:encodingStyle = "http://www.w3.org/2001/12/soap-encoding">
    <SOAP-ENV:Body xmlns:m = "http://www.xyz.org/quotation">
        <m:GetQuotationResponse>
            <m:Quotation>
                Here is the quotation
            </m:Quotation>
        </m:GetQuotationResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 2.4.: Example of a SOAP response sent over HTTP.

\(^4\)Examples taken from https://www.tutorialspoint.com/soap/soap_examples.htm
2. Introduction to Web Services and data fetching

2.1.3. REST

REST (or RESTful) is an architectural style first describe by Fielding in his PhD dissertation [Fie00] in 2000. It is – like the web itself – centered around resources which are identified by global identifiers (Uniform Resource Identifier (URI)). As Halili et al. wrote in their paper [HHN14], the “main idea lies on the data, not on actions. It is an architectural style that treats the Web as a resource-centric application”.

A resource can be any kind of information like a document, a service (e.g. a weather forecast), a collection of resources etc. or, as Fielding wrote: “any concept that might be the target of an author’s hypertext reference”. Each resource can be located via its URI and is represented via different media types once requested.

According to Fielding in his dissertation [Fie00], the architectural style is demanding for six characteristics a REST service has to have:

Client-server In general, all requirements of the client-server architecture have to be fulfilled. A server provides a service that can be requested by a client.

Statelessness The communication between server and client has to be stateless, meaning that every request from client to server has to contain all necessary information to be understood. The state of the current session is therefore kept completely on the client. This characteristic supports the scalability of a web service since it allows an uncomplicated load balancing.

Cacheable Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests [Fie00].

Uniform interface This constraint is fundamental to the design of a REST service. It simplifies and decouples the architecture, which enables each part to evolve independently. To obtain such an uniform interface, it has to fulfill the following four constraints:

Identification of resources Individual resources are identified in requests using a base Uniform Resource Locator (URL) for the REST-conform service which is extended to an URI for each available resource.

Manipulation of resources through representations The resources available at an URL can have different representations. Based on a request, a server may deliver different representations of a single resource, for example different formats like HyperText Markup Language (HTML), JavaScript Object Notation (JSON) or XML, or just a description or documentation of the service. Based on this representation and accompanying metadata, the client is then holding all necessary information needed to modify or delete the resource.

Self-descriptive messages Each message includes enough information to describe how to process the message (e.g. which parser to invoke).
2.1. Web Services

**Hypermedia as the Engine of Application State (HATEOAS)** After accessing the initial URI/URL of a REST-application, the server includes additional information (for example via links) in its served answers so that the client is able to discover all actions and resources available. This goes on for each “step” the client makes. Therefore there is no need for the client to know how the service is structured, which means that the client does not contain any more information on that matter except for the base URL as the entry point of the service.

**Layered system** A layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting[^Fie00]. This allows to hide further layers and to encapsulate e.g. legacy services from new clients and the other way around. It also sets a bound to the complexity of the overall system.

**Code on demand (optional)** REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts, e.g. the transmission of JavaScript Code while serving an HTML-page[^Fie00]. Any web service API that implements these architectural constraints is called a RESTful API. These APIs, sometimes also called “Web API” are therefore defined with the following three points:

- They have a base URL, e.g. `http://api.example.com/resources`;
- They use media types as a representation of the resources, e.g. JSON or XML;
- They use the standard HTTP methods/verbs (see table 2.1).

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
<th>Comment</th>
<th>CRUD counterpart</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Requests a resource.</td>
<td>Has no side effects and is therefore considered safe. Nullipotent.</td>
<td>READ</td>
</tr>
<tr>
<td>POST</td>
<td>Adds a new (sub)resource below the described resource.</td>
<td>Not idempotent. Each call creates a new object instead of returning the existing one.</td>
<td>CREATE</td>
</tr>
<tr>
<td>PUT</td>
<td>Creates the described resource. If it already exists, the resource will be changed.</td>
<td>Idempotent. The first call has side effects but not the succeeding ones.</td>
<td>CREATE/UPDATE</td>
</tr>
<tr>
<td>PATCH</td>
<td>A part of the described resource will be changed. Side effects are allowed.</td>
<td>Optional. Not needed to map CRUD operations.</td>
<td>UPDATE</td>
</tr>
<tr>
<td>DELETE</td>
<td>Deletes the described resource.</td>
<td>Idempotent. The first call has side effects but not the succeeding ones.</td>
<td>DELETE</td>
</tr>
<tr>
<td>HEAD</td>
<td>Requests metadata for a resource.</td>
<td>Nullipotent and optional. Not needed to map CRUD operations.</td>
<td></td>
</tr>
<tr>
<td>OPTIONS</td>
<td>Checks which methods are available for a resource.</td>
<td>Nullipotent and optional. Not needed to map CRUD operations.</td>
<td></td>
</tr>
</tbody>
</table>

[^Fie00]: File Reference

Table 2.1.: The most commonly used HTTP verbs and their Create, Read, Update, Delete (CRUD) counterpart.
Since (as mentioned before) REST is an architectural style and not a protocol like SOAP there is no official standard which developers have to implement. Because of that, some describe their API as RESTful even if it does not fulfill all of the constraints mentioned above. In this case, they are not RESTful but something similar. To categorize these kind of APIs, one can use the Richardson Maturity Model.

### 2.1.3.1. Richardson Maturity Model

The Richardson Maturity model was developed by Richardson and first explained in his talk at QCon in 2008 [Ric08]. He defined four “classes” of web services, depending on how much they stick to the RESTful constraints (see fig. 2.4).

![Glory of REST](image)

Figure 2.4.: The steps towards REST; adapted from [Fow10].

- **Level 0** is basically a SOAP service. All of its functions are accessed by posting to one URI.
- **Level 1** introduces resources. This means that the POST messages are now sent to different URIs.
- **Level 2** then makes use of the HTTP verbs. Instead of sending something like `POST <availableBooksRequest` the client actually performs a GET request to the resource URI, e.g. `GET /books?status=available`.
- **Level 3** finally introduces HATEOAS. Each response from the service now includes a link element telling the client how it can proceed its navigation through the service⁵.

⁵A more detailed description of the different levels can be found at Fowler [Fow10].
2.2. Dynamic Database Queries

Most web services or web APIs are exposing data which is stored in an underlying database. For developers writing these APIs there are now basically two options to load this data from within the application:

- Write the queries as explicit static String into the code. This is a valid option, but has some downsides, for example the application is forced to load everything stated in the query even if the API request is limited via parameters which results in a poorer performance. To avoid this, the second option can help.

- Build the queries dynamically based on what the API needs in that moment. To do this, a developer can use for example an Object-Relation Mapping (ORM) software to help him.

ORM software helps to map data represented in a tabular format to objects inside an object-oriented languages such as for example Java. By mapping object attributes to database columns, ORM software is able to build the queries so that only the columns needed to fill the desired attributes are retrieved. This results in a better performance of the application. The downside of these software is however an application can only take full advantage of this mapping if the tables in the database are representing one object. As soon as the database layout is requiring columns from two or more tables to fill the attributes of an object, the automatically created queries get a lot more complex and take more time to be executed on the database. [Hib]
3. Analysis

In December 2015, a review of the current WBM marked the begin of the upgrade project. In it’s final review document, the committee remarked among other things the following points:

- The whole WBM project is extremely wide;
- The code was written without a coherent design in mind;
- Different services use different technologies and have a different look-and-feel.

It was recommended to “reorganize [the project] with a clear scope, well defined boundaries, and factorized responsibilities in order to assure the continuity of the essential services.”[Mom+15] Another recommendation was the split of the handling of the data into two parts/layers: the presentation and the aggregation.

While the presentation layer “includes the development and maintenance of the web user interface”[Mom+15], the aggregation layer is responsible to collect and expose “[non-event] data from heterogeneous sources, [with] different data formats and changing contents”[Mom+15]. Between the two layers, “a clear interface should be defined and documented”[Mom+15]. These two last citations are the main features of the aggregation layer.

The clear interface to be defined and documented has additional requirements:

- It should follow the REST architectural style. REST was already chosen as the way to go by the people responsible for the upgrade project. The advantages of the style in comparison with SOAP have already been mentioned in the previous chapter. Using REST also makes sure that other clients than the parallel developed presentation layer can use it without much problems.

- The response format should implement the JSON:API specification¹. The presentation team suggested to use this specification which handles concerns like the following without the need to think about a custom definition:

  - Each response has a member describing the data type. This means, every client can recognize the data easily without necessarily knowing which REST endpoint was requested.
  - JSON:API has a specific format for errors which makes them easy to process automatically.

¹http://jsonapi.org/
3. Analysis

- The data is decoupled from auxiliary data like relationships, links and meta data. This allows clients to ignore the auxiliary data without having to parse the whole response. It also allows to in- or exclude this auxiliary data based on performance decisions.

- The specification already provides a definition on how request parameters for projection, pagination, filtering and sorting have to be named and used.

Chapter 4 shows the definition of our API based on the JSON:API specification.

By examining the current WBM, we can find the following points regarding the data sources. To collect the data to be exposed, the aggregation layer has to retrieve the data from an Oracle database which stores the data coming from the heterogeneous sources. The mentioned heterogeneous sources are the different subsystems of the CMS experiment and the LHC. CMS subsystems are storing their data mainly in the database but in some other cases the data is provided via files, Data Interchange Protocol (DIP), DAQ-applications, etc. In the second case the data is then being extracted, transformed and loaded into the database. Based on the model of this database, the data of each source is loaded into a different table. Therefore, the aggregation layer has to collect data from multiple tables and/or views (depending on the available access rights) spread across the database like the current WBM does.

Having a look at the persistence layer of the current WBM, all of the queries to access the database are written in the source code as they were provided by the responsible persons without performing further refactoring or improvements. Therefore the software is performing several queries to the database in order to collect the data needed for one page resulting in a lower performance than possible. This means, that an additional requirement is to rework these queries in order to reduce the amount of queries to be made. Additionally, the aggregation layer has to enable correlations by time to be made between data points. These correlations are required for the user interface so that the users can later search for data occurring in a specific Fill, Run, Lumisection or in a period of time. To be able to make these connections, the aggregation layer has to provide an internal conversion service to convert the different time units. This is due to the fact that the data coming from LHC uses the standard Unix time, Fill and orbits while data coming from CMS is mostly based on Runs or Lumisections as time units.

3.1. Essential Services

The essential services were defined for the prototype as the following six: Fill Summary, Fill Report, Run Summary, Run Report, L1 Report and L1 Algo Trigger Report.

A “Summary” in this case is an operational list of data objects that can be filtered by a date range, flags, etc. while a “Report” is detailed information on a single data item. This section aims to describe these services in respect to what type of information a user of the new WBM can get of them.
3.1. Essential Services

Figure 3.1.: The Fill Summary service with control (on top) and content portlet (below)

3.1.1. Fill Summary

The Fill Summary service provides a control portlet ("controller") and a content portlet (see fig. 3.1). The content portlet contains a list of Fills that match the criteria set in the control portlet. These criteria can be a range of Fill numbers, a date range or a specific Era. Additionally, the user can add options to display only Fills with a stable beam, protons, ions or proton-ions. These four options can be selected in which combination the user likes.

3.1.2. Fill Report

The Fill Report service provides one controller and various content portlets. The controller can be used to select a specific Fill using a Fill number. There are portlets showing Fill details, a Run Summary (which means again a list of Runs occurring during the Fill), a dynamic chart about the integrated Luminosity, Downtimes during the Fill and various other portlets containing each a static plot.

3.1.3. Run Summary

Similar to the Fill Summary, the Run Summary service provides a controller and a content portlet (see fig. 3.2). The content portlet contains a list of Runs that match the criteria set in the controller. Again, these criteria can be a range of Run numbers, a date range or a
3. Analysis

Figure 3.2.: The Run Summary service with control (on top) and content portlet (below)

specific Fill. The user can as well add different options to display only Runs that match these options. Options are a certain sequence and/or a combination of components which have to be online during a Run.

3.1.4. Run Report

Similar to the Fill Report, the Run Report service provides one controller and four content portlets. The controller enables the user to select a specific Run via the Run number. The content portlets show Run details, Downtimes, Lumisections and information about the partitions and FEDs during the selected Run.

3.1.5. L1 Report

The L1 Report service also contains a controller and several content portlets. Similar to the controllers in the other Reports, the user can select a specific Run via the Run number. The portlets of the L1 Report are:

- L1 Details;
- L1 Overall Deadtimes;
- L1 Beam Active Deadtimes;
- L1 Trigger Rates;
- L1 Algorithm Triggers.
3.2. Computer science context

3.1.6. L1 Algo Trigger Report

This service contains a controller and three content portlets. With the controller, the user can select a specific Run via the Run number and an algorithm via its bit number. The portlets then show details about this L1 Algorithm Trigger, a chart about the trigger rates during the different Lumissections and a third portlet containing the numbers shown in the chart.

3.2. Computer science context

After analyzing the work to do and gathering all requirements, we would like to set the task into an computer science context. This helps to understand the different disciplines which need to work together in order to complete the task successfully.

The function of the aggregation layer can be seen in two different ways. One way would be looking at it as a whole. Like this the aggregation layer would fit into the field of data warehousing. It would represent the data logic layer of a data warehouse architecture and use the concept of On-Line Analytical Processing (OLAP) or more specific Relational OLAP (ROLAP) to serve the data from the underlying data storage to the data presentation layer. Figure 3.3 shows such a data warehouse architecture. To bring it into the context of the WBM system, one can make the following connections:

- The data source layer corresponds to the different subsystems of the CMS experiment.

- Data extraction layer, staging area and Extract, Transform, Load (ETL) layer correspond to the various procedures used to get the data from the subsystems into the database.

- The data storage layer corresponds to the Oracle database which stores all the available data.

Another way to look at the aggregation layer would be like “drilling down” and look at the internal layers separately. In this way, the aggregation layer would fit into different computer science fields: the API would fit into the field of interface design or rather web service development/software engineering whereas retrieving data from the database and the needed query optimization would fit more into the field of databases.
3. Analysis

Figure 3.3.: Architecture of a data warehouse; adapted from https://www.1keydata.com/datawarehousing/data-warehouse-architecture.html
3.3. Technology evaluation

This section introduces three possible tools that allow to develop an API fitting the requirements. With all three of them, a minimal working example with the designed of a simple resource was set-up to get an understanding of the capabilities and flaws of each possibility. They were compared in different aspects so that a decision could be made on solid arguments.

3.3.1. GraphQL

GraphQL\(^2\) has been developed by Facebook Inc. since 2012. It was first used internally before releasing it as OpenSource Software in 2015. On the official website, GraphQL is described as “query language for APIs and a runtime for fulfilling those queries with your existing data”\(^{[Fac17]}\) providing a standard of this query language.

Today, more and more companies are developing/migrating their APIs to GraphQL or use GraphQL to power their mobile apps and websites, e.g. GitHub, Pinterest, Sky, The New York Times, Twitter, Yelp, etc. \(^{[Fac17]}\)

To provide a GraphQL service, one has to specify a schema which serves as “a complete and understandable description of the [provided] data”\(^{[Fac17]}\). This allows the clients to ask for exactly what they want to have and nothing more.

In such schema, one defines types (comparable to objects in JSON) and also the fields of these types (like the attributes of a JSON object). One also has to provide functions which have to resolve the fields of each type meaning that these functions have to care about where to get the data from. If these resolvers are implemented in the right way, no unnecessary data is collected during the building of the answer.

```
schema {
    query: Query
}

type Query {
    allEras(filter: EraFilter, skip: Int = 0, first: Int = 0): [Era]
}

type Era {
    name: String!
    period: Int!
    start_time: Float!
    end_time: Float!
    start_run: Int!
    end_run: Int!
    start_fill: Int!
    end_fill: Int!
}

type EraFilter {
    name_contains: String
}
```

Listing 3.1.: GraphQL schema

\(^2\)https://graphql.org/
3. Analysis

In the case of this technology evaluation, the Era resources was chosen due to the fact that all of its attributes are stored in one table in the database. Its schema is shown in listing 3.1.

The root object of the schema is a Query (lines 1-3) which can be used to ask for a list of Era types (line 6). The query accepts a filter (EraFilter) and also allows pagination with the help of the skip and first parameters.

As its name indicates, it will return a list of all available Era resources if no filter or parameter is used. The EraFilter takes as input a String and will apply it, so that only Eras containing that String in the name field will be returned. This is done with a WHERE-clause inside the SQL-query like this one:

```sql
WHERE ERA LIKE '%STRING%'
```

The Era type itself consists of several fields as defined in chapter 4.

GraphQL services listen to a specific URL where clients can send their queries via a request. These queries will then be validated and executed.

```
query {
  allEras (filter: {name_contains: "E"}) {
    name
    period
    start_time
    end_time
    start_run
    end_run
    start_fill
    end_fill
  }
}
```

Listing 3.2.: GraphQL query to get all available Eras containing an “E” in their name field with all fields

Listing 3.2 shows an example query which can be used with the schema shown in listing 3.1. The client would send it to the server via the following request

```
http://localhost:8080/graphql?query={allEras(filter:{name_contains:"E"})
  {name%20period%20start_time%20end_time%20start_run%20end_run%20start_fill%20end_fill}}
```

and the server would in this case return a list with JSON-objects looking like the one in listing 3.3.

Apart from a reference implementation in JavaScript for both server and client, there are several other libraries available for either server or client or both, for example for C#, Go, Java, PHP, Python, etc. For the evaluation, the graphql-java library was used for the implementation. The server was provided by Jetty. [Fac16] [Fac17] [Már15]
3.3. Technology evaluation

```json
{
  "data": {
    "allEras": [
      {
        "name": "2016E",
        "period": 1,
        "start_time": 1468583936,
        "end_time": 1469447936,
        "start_run": 276831,
        "end_run": 277420,
        "start_fill": 5096,
        "end_fill": 5117
      },
      {
        "name": "2017E",
        "period": 1,
        "start_time": 1505938304,
        "end_time": 1507792640,
        "start_run": 303435,
        "end_run": 304826,
        "start_fill": 6222,
        "end_fill": 6293
      }
    ]
  }
}
```

Listing 3.3.: JSON response to the request in Listing 3.2

3.3.2. Spring

Spring\(^3\) is a collection of different projects that can be used to set up web services quickly and without too much configuration depending on the need of the programmer. The first version of the original Spring Framework was developed by Pivotal Software Inc. in 2002 as a “response to the complexity of the early J2EE specifications”\([17]\). Over the years, other projects were added, like Spring Data, Spring Boot, Spring HATEOAS, Spring LDAP, etc. All the projects are as of June 2003 open source under the Apache License 2.0 and "modular by design"\([Piv17a]\) so that they can be used in whatever combination is needed.

For the technology evaluation, only two projects were used: Spring Boot and Spring HATEOAS.

Spring Boot helps to set up a running service as quick as possible without the need for extensive configuration. A plus point is that Spring Boot applications can run within the embedded Jetty Server so that there is no need to package and deploy WAR-files.

Spring HATEOAS was used to extend the REST representations from Spring Boot so that they follow the HATEOAS principle. These links allow the client to browse the API interactively. In order to provide a Service, one only needs a class annotated with `@RestController` to tell Spring that it contains methods used as endpoints (see listing 3.4) and another class representing the desired resource as a Plain Old Java Object (POJO), see listing 3.5. The controller contains a method to get the data from e.g. a database which is annotated with `@RequestMapping()`, containing the endpoint path as parameter, e.g. "/eras".

\(^3\)https://spring.io/
3. Analysis

Listing 3.4.: Class body of a Spring REST controller.

```java
package ch.cern.cms.daq.oms.aggregation.spring.controller;

@RestController
public class EraController {
    //...
    public EraController() {
        //...
    }
    @RequestMapping("/eras")
    public HttpEntity<List<Era>> eras(
        @RequestParam(value = "name", required = false, defaultValue = ") String name,
        @RequestParam(value = "skip", required = false, defaultValue = ") String skip,
        @RequestParam(value = "first", required = false, defaultValue = ") String first
    ) {
        //...
    }
}
```

Listing 3.5.: POJO of a Spring resource.

```java
package ch.cern.cms.daq.oms.aggregation.spring.model;

public class Era extends ResourceSupport{
    // fields representing the attributes of the resource
    @JsonCreator
    public Era(
        @JsonProperty("name") String name, @JsonProperty("period") Integer period,
        @JsonProperty("start\_time") Long start\_time, @JsonProperty("end\_time") Long end\_time,
        @JsonProperty("start\_run") Integer start\_run, @JsonProperty("end\_run") Integer end\_run,
        @JsonProperty("start\_fill") Integer start\_fill, @JsonProperty("end\_fill") Integer end\_fill
    ) {
        super();
        // set parameter to fields
    }
}
```

3.3. Technology evaluation

Since this approach is also using the HATEOAS project, the controller has to have a return object of the generic type `HttpEntity<T>`. This `HttpEntity` contains header and body of the Http-request or -response and was defined by the Spring Framework. In order to add links to the resource, the representing POJO has to extend the `ResourceSupport` class provided by the HATEOAS project. This allows the developer to easily add automatically generated links to methods of controller classes.

In order to be able to process query parameters, the method which is annotated with `@RequestMapping` can use parameters that are annotated with `@RequestParam`. These annotated parameters have to specify a value (which indicates the name of the request parameter used in the URL), whether they are required and a default value which will be used if the parameter is not used in a request. For example, the following line

```java
@RequestParam(value = "name", required = false, defaultValue = "") String name
```

specifies a request parameter called "name" which is not required and has as default value the empty `String`. Inside the mapping method, the parameter is represented and can be used as a `String` object.

If we now perform the following request `http://localhost:8080/eras?name=E` on the service, the answer we get is showed in listing 3.6.

```
{
    "name":"2016E",
    "period":1,
    "start_time":1468584000,
    "end_time":1469448000,
    "start_run":276831,
    "end_run":277420,
    "start_fill":5096,
    "end_fill":5117,
    "links": [
      {
        "rel":"self",
        "href":"http://188.185.100.206:8080/eras?name=E&skip=&first="
      }
    ],
    {
      "name":"2017E",
      "period":1,
      "start_time":1505938352,
      "end_time":1507792681,
      "start_run":303435,
      "end_run":304826,
      "start_fill":6222,
      "end_fill":6293,
      "links": [
        {
          "rel":"self",
          "href":"http://188.185.100.206:8080/eras?name=E&skip=&first="
        }
      ]
    }
}
```

Listing 3.6.: JSON response of Spring
3. Analysis

3.3.3. Katharsis and Dropwizard

Katharsis\(^4\) is a Java library that is added as an additional layer to a RESTful web service. In doing so, it implements the JSON:API specification and supports HATEOAS. Katharsis provides an out-of-the-box support for query parameters like projection, filtering, ordering and pagination. It is open-source, available under the Apache License 2.0 and was first released in June 2015.

Since Katharsis does not provide a stand-alone web service, there is the need to do this with another framework/library. We chose Dropwizard\(^5\) because it is a standalone tool with good performance, comprehensive configuration possibilities and out-of-the-box support to measure metrics of the application. Dropwizard describes itself as a “Java framework for developing [] high-performance [and] RESTful web services” which was initially released in December 2011. It is also open-source software and available under the same license as Katharsis.

To get a first endpoint with a desired resource running, one needs to create some classes:

- a class to map the required configuration file;
- a domain object to represent the resource;
- a repository class to provide data for the created domain object (in the case of this thesis with a connection to the used database);
- an application class which tells Katharsis where the library should look for the classes representing the desired resources.

Once this has been done, the server will listen to a specified URL where it is now able to receive HTTP requests. A request to get all available Era resource returns a list of JSON objects which look like the example resource in listing 3.7.

```json
{
  "data": [
  {
    "attributes": {
      "name": "2011A",
      "start_fill": 1613,
      "end_fill": 2040,
      "start_time": 1299974400,
      "start_run": 160404,
      "period": 1,
      "end_time": 1313971200,
      "end_run": 173692
    },
    "type": "eras",
    "id": "2011A_1",
    "links": {
      "self": "http://localhost:8080/api/v1/eras/2011A_1"
    }
  }
  ]
}
```

Listing 3.7.: JSON response of Katharsis

\(^4\)https://katharsis.io/
\(^5\)http://www.dropwizard.io/1.2.0/docs/index.html
3.3. Technology evaluation

3.3.4. Comparison

To get to a decision on which technology stack to use we did a performance evaluation and a comparison of the features of each solution.

3.3.4.1. Performance evaluation

To test the performance of the used technologies, all of them were deployed on the same (without running parallel) instance of CERNs OpenStack infrastructure. The instance has the following specifications: two VCPUs with a base frequency of 2.2 GHz and the Skylake instruction set, 3.7GB of RAM and a disk volume of 20GB. To automate the evaluation as much as possible, a Python script was written to run the tests. Additionally, all caching mechanisms of the technologies were turned off/removed to make sure that with every request the application has to fetch the data from the database. The testing procedure itself followed these three steps:

- Start the web service;
- Run 100 requests for all available Era resources without taking timing measurements to allow the service to initialize possible threads;
- Run 1000 requests for all available Era resources and take measure the latency (meaning the time it takes for the service to answer the request).

To be absolutely sure that no background processes interfere with the time taking, another OpenStack instance of the same flavour was set up to run the measurement script. After removing the five lowest as well as the five highest values for each technology and rounding the taken data to three decimals, a histogram as in fig. 3.4 can be plotted.

![Histogram of measured latency values](image)

Figure 3.4.: Histogram of the measured latency values
3. Analysis

The chart shows on the x-axis the time in seconds with a range from five milliseconds to 30 milliseconds. The frequencies of individual latencies are plotted on the y-axis with logarithmic scale.

**GraphQL** has the most values (≈ 490) at around seven milliseconds latency and declines gradually until the value of 16 milliseconds. It then has another spike with five measurements for 18 milliseconds and the slowest measurement with 19 milliseconds.

The graph of **Katharsis** looks similar, also with the most values (≈ 530) at around seven milliseconds latency. It then declines steeper and has another local maximum with eight datapoints around twelve milliseconds. The third spike is then located with three measurements at 17 milliseconds and the slowest measurement at again 19 milliseconds.

**Spring** is in general a bit slower than the other two technologies. The most measurements were taken for a latency of around eight milliseconds (≈ 340). The graph then also declines until one measurement for each 19 and 21 milliseconds. It then has another two local maxima at 22 and 26 milliseconds leading to the longest latency at 29 milliseconds.

In general, all three technologies are having the majority of their latency measurements below 15 milliseconds.

With respect to the basic advice regarding response times given by Nielsen in chapter 5.5 subsection “Response Time” of [Nie93], in which he states that “0.1 second[s] is about the limit for having the user feel that the system is reacting instantaneously”, all the values are very good, given the fact that the data has to be queried from a database before serving the response. The measured values of Katharsis look even better when taking into account, that the total size of the response object is nearly twice the size of the GraphQL answer (see figure 3.5).

![Comparison of JSON file sizes](image)

**Figure 3.5.** Comparison of the file sizes containing the response of the different technologies.
3.3. Technology evaluation

3.3.4.2. Features of the technologies

The query-based requests of GraphQL allows the clients to ask for what they need and to get only back for what they asked. Since a query can also follow references between different resources, they can define an exact query to only ask the API once instead of using multiple requests to different URLs.

For developers, it is easier to evolve a GraphQL API since they can just add new fields to the schema or mark outdated fields as deprecated. In this way, they do not have to support resource versioning and also the clients do not have to update the code to a new version of a resource. Also, the query style of the request makes it easier readable and also to understand, what the response object looks like.

The advantages of Spring and in this case Spring Boot and Spring HATEOAS are, that it is quiet easy to set the API up. The developer does not need to clutter around with an extensive configuration and Spring automates as much as possible.

Also, Spring embeds a web server to run the API directly, so that there is no need to deploy any WAR or JAR files on a server. Spring HATEOAS automates the link creation for the HATEOAS principle to automatically point to the respective controller methods.

In case the requirements for the project would change in the future and it would be necessary to include another Spring project, this include would be as easy as possible since Spring projects are designed to work seamlessly together.

As the only technology in our evaluation, Katharsis comes with an out-of-the-box support of standard request parameters for HTTP-requests like pagination, projection, etc. Also, it is the only used technology to implement the JSON:API standard which defines a REST resource including metadata and relation links from the HATEOAS principle like "self" without having to add a lot of logic manually.

For all the technologies studied, they make it easy to connect to different databases. Each of them is consisting in how to define a (REST) resource and a corresponding repository to get the data.

3.3.4.3. Decision

With respect to the requirements of the aggregation layer, one possible solution would be to use the GraphQL query language since it makes querying for different resources easy (like the Run Report which is also using fields from the Fill Report) and runs fast and stable as seen in the Performance evaluation.

As the aggregation layer is only the back-end of the project, the group that develops the front-end has also to be taken into account for the requirements. After speaking with them and presenting them the possibilities of each of the used technologies, they liked the ideas behind GraphQL. But as already mentioned in chapter 3, they decided to require a JSON:API compliance of the resources which means for the aggregation layer that Katharsis is the choice to continue the implementation.
This chapter describes how the API was designed and therefore also already shapes the design of the implementation.

As earlier described in section 3, the API should follow the REST architectural style and implement the JSON:API specification. As mentioned in section 2.1.3 a RESTful architecture is based on Resources which are accessible through a resource identifier like an URL. Each resource has its representation, in this case JSON.

A JSON response consists of two parts: the data (see listing 4.1) and the metadata (see listing A.1).

```
{
  "data": [
    {
      "id": "2011A_1",
      "type": "eras",
      "attributes": {
        "start_time": 1299974400,
        "name": "2011A",
        "end_time": 1313971200
      },
      "links": {
        "self": "http://localhost:8080/api/v1/eras/2011A_1"
      },
      "meta": {
      }
    }
  ]
}
```

Listing 4.1.: Data part of a response

The data part can be either one resource or a list of resources. Each resource is represented as an object. The metadata is appearing in two different types: The first type is the metadata of the response as a whole which is used to describe it. This includes e.g. the number of resources in the response, the resource type in the response, the version or the fields of the resource, links to the next page (if pagination was used), etc. The second type is the metadata of one resource. This includes e.g. the relationships to other resources (including links to them), a link to the resource itself, descriptions of a field of the resource which is not the same for every resource of that type, etc [Fie00].

After clarifying these terms, let’s start to define the resources. In order to minimize the number of queries the front-end has to use to get the data necessary to fill the pages, portlets and controllers for the first prototype, a resource is defined as the data needed in one of these units. The pages, portlets and controllers used for the prototype are described
In the course of their work on the prototype, the front-end team has created a document in which they gathered the necessary requirements the API has to deliver in order to fill the different portlets and controllers required to reach the set goal of the prototype (see document “API requirements for portlet and controllers” on ¹ by Mantas Stankevicius). The resources of the API were designed based on this document.

### 4.1. Resource definitions

The following subsection defines and describes all resources needed to provide data for the prototype services described in section 3.1. A more detailed view on the fields of each resource can be found in appendix A.2. The first idea was to just take the tables in the mentioned document and define them as resources. With a closer look to the attributes used, it becomes apparent that some of them appear in different tables. This means, that these tables can be aggregated in a bigger table. On the other hand this meant also that there will be some requests only asking for some attributes of the aggregated tables which would result in a lot of unnecessary transmitted and processed data which slows the aggregation layer down without being used. The consideration in this case was then for the smaller aggregated tables to not really care due to having only a few attributes and – with a look to the database queries used in the current WBM – using none or only one or two joins. For the bigger tables (the Run and Fill resource) it was decided to implement a helper class – the query builder – which should build the query based on the requested attributes so that joins only have to be resolved if the corresponding attribute is requested.

#### 4.1.1. Run

The Run resource describes the details of a CMS Run. Table A.4 lists the definition of this resource. To identify a specific resource, the field run_number will be used.

#### 4.1.2. DAQ-Readout

The DAQ-Readout resource describes the DAQ-details of a Run. It holds information about the FED-IDs which were included and excluded in the current Run. Table A.1 lists the definition of this resource. Since the information in this resource belong to a specific Run, the run_number is used as identifier.

#### 4.1.3. Fill

The Fill resource describes the details of a LHC Fill. The definition of this resource can be found in table A.5 To identify a specific resource, the field fill_number will be used.

¹https://cwernet.de/ba-thesis
4.1. Resource definitions

4.1.4. Lumisection

The Lumisection resource describes the details of a Lumisection. As a Lumisection is a sub-section of a Run, one Run contains many Lumisections, but a Lumisection is always assigned to only one Run.
The Lumisections are only consecutively numbered inside a Run which means that with a new Run, the numbering starts from one again.
To identify a specific resource, we need two fields then: the lumisection_number as well as the run_number.
The whole definition of this resource can be found in table A.6.

4.1.5. Downtime

The Downtime resource describes the details of a Downtime. This includes details about the cause of the downtime, its length, in which run and fill it occurred and comments on it.
All details can be found in table A.7.

4.1.6. L1ConfigurationKey

The L1ConfigurationKey resource holds detailed information about the configuration of the L1 Trigger during a specific run. Therefore, the run_number is used as identifier of a specific resource. The whole definition of this resource can be found in table A.8.

4.1.7. L1TriggerRates

The L1TriggerRates resource describes in detail at which rates data was taken by the L1 Trigger during a specific Run. Therefore, the run_number is used as identifier of a specific resource. Table A.10 lists the definition of this resource.

4.1.8. L1OverallDeadTime

The L1OverallDeadTime resource holds information about the overall deadtime of the L1 Trigger during a specific run. Deadtime is here used as the time in which no data was taken by the experiment. It lists the start and end time, the duration and the different reasons of a deadtime. The reasons are listed in percentages and in discrete numbers. Since the resource gives detailed information to a Run, the run_number is used as identifier. More details of the fields of this resource can be found in table A.11.

4.1.9. L1BeamActiveDeadTime

The L1OverallDeadTime resource holds information about the deadtime of the L1 Trigger while the beam was active during a specific run. The definition of this resource can be found in table A.12. This resource holds detailed information for a specific Run, so the run_number will be used as identifier.
4. API design

4.1.10. L1AlgorithmTriggers

Table A.9 lists the definition of this resource. This resource contains information about all algorithms that are used to trigger the L1 Trigger and the High-Level Trigger (HLT). The information stored is the count and rate for each algorithm at different stages of the L1 triggering process:

- *Pre-Deadtime (DT) before prescale* means at the very beginning of the process and therefore the highest number;
- *Pre-DT after prescale* means that a first prescaling has been done to sort out some of the events;
- *Post-DT from HLT* means the number of triggered events that reach the HLT.

These numbers each belong to a specific Run and a specific algorithm, so the run_number as well as the internally used bit algorithm number will be used as identifiers.

4.1.11. L1AlgorithmDetails

This resource holds basically the same information as the previous section. The difference is that the numbers are now belonging to only one Run, one algorithm and are given for each Lumisection. Therefore Run, so the run_number, the lumisection_number as well as the internally used bit algorithm number will be used as identifiers. All field definitions of this resource can be found in table A.13.

4.1.12. InstLumi

The InstLumi resource holds information about the instantaneous Luminosity. The Luminosity is measured with different so called Luminometers and logged every four nibbles by the Beam Radiation Instrumentation and Luminosity (BRIL) subsystem. The InstLumi is mainly used to generate some of the static plots which are used in the current WBM and will also be used in the front-end of the new monitoring system. The whole definition of this resource can be found in table A.14.

4.1.13. Era

The Era resource gives information about an Era. As identifier, the name of the Era concatenated with its period field is used. The resource holds information about the start and end time of the Era as well as the numbers of the first and last Run and the numbers of the first and last Fill inside it. The Era is also a property of a Run or a Fill and it can have multiple running periods. Table A.15 lists the definition of this resource.

4.1.14. Sequence

The Sequence resource holds a list of names a Run can have to identify what has been going on in CMS during that run. For example, GLOBAL-RUN is used to indicate that CMS
ran in “production” whereas all other Sequence-names indicate that there has been a maintenance or calibration of a subsystem. The definition of this resource can be found in table A.16. The resource holds only one field (the name of the sequence) and is used to fill a drop-down list in the Run Summary portlet. Apart from being a resource on its own, the sequence is also a field of the Run resource.

### 4.1.15. VacuumSummary

The VacuumSummary resource holds information about the vacuum in the LHC around Point 5 (the location of the CMS experiment). It is mainly used to retrieve the information for the corresponding static plot. The definition of all field of this resource can be found in table A.17.

### 4.2. Designing the response object

The design of the response object follows the JSON:API specification. Figure 4.1 shows a schematic view of it. Each response must either contain an errors, meta or data member.

![Diagram of the response object](image)

*Figure 4.1.: The response object of the API.*

Errors and data objects may never appear together in one response. The optional “links” member of the response object contains links used to navigate through the resources, e.g. when pagination is used, the member contains links to the next and
4. API design

previous page. The specification also mentions two additional optional members called “jsonapi” and “included”. Both will not be used in the aggregation layer.

4.2.1. Data

The data member contains either one resource object or a list of resource objects. One resource object must contain at least the members “id” and “type”. The “type” describes, which resource the object represents. Additionally, there are some optional members:

- **attributes**: contains the attributes of a resource (as defined in the previous section)
- **relationships**: contains links to related resources
- **links**: contains e.g. a link to the resource itself
- **meta**: can contain additional metadata describing one or more attributes in more detail, e.g. the unit of an attribute

4.2.2. Metadata

The “meta” member is shown in fig. 4.2. It always contains the “totalResourceCount” member which shows, how many resource objects are contained inside the “data” member. If the API request contains the `include=meta` parameter, the optional members are included. Otherwise, only the counter is contained. The optional members are the following:

- **resource**: The name of the resources inside the “data” member
- **version**: The version number of the resource
- **fields_order**: An array containing the order of the attributes of a resource as given in the projection. If no projection is specified, it is the default order.
- **fields**: This member contains all the metadata objects about the fields targeted by a request. The metadata objects for fields not targeted are null. One field contains the following members:
  - **title**: title of the field used by the Presentation Layer
  - **sourceType**: type for the field in the Data Source
  - **apiType**: type for the field in the Aggregation API
  - **units**: units for the field used by the Presentation Layer
  - **description**: description of the field used by the Presentation Layer
  - **comments**: comments regarding the field
4.3. Basic API structure

In addition to defining resources and the response object, the basic structure of the API was described using the OpenAPI specification\(^2\). Each resource can be accessed using the base URL `/api/<version>/<resourceName>`. These URLs return a list of available resources. To access one specific resource, one can use either an additional path parameter to specify the ID of a resource, for example `/api/v1/eras/<era_id>` or use a query parameter to filter for a specific value on the ID field, for example `/api/v1/eras?filter[id]=<era_id>`. On all endpoints, the following common API options are available as query parameters:

**projection** Sometimes it is not required to retrieve all the fields defined in a resource but a subset of it. For these cases the API allows to use the projection feature where it is possible to define which fields are included in the response.

**filtering** Filtering is used to filter and narrow down the result set by specifying some criteria. The WBM Aggregation API uses the “filter” parameter to define

\(^2\)https://swagger.io/specification/
4. API design

the filtering criteria. The property name in the filtering expression needs to match the field name of the requested resource.

ordering  Ordering is the way that the items are ordered in a payload and returned to the client.

pagination  The paged pagination is used to reduce the items within a dataset. This option is controlled by two parameters; the “limit” controls the size of one page and the “offset” (which has to be 0 or a multiple of the limit) controls which page is requested.

Since a full description in the OpenAPI specification would be rather comprehensive and time intensive to write, only the description of the Era\(^3\) was written down.

4.4. Designing the database connection

The connection to the database has to be reliable and stable. To ensure a stable connection the implementation has to be resilient towards connection losses or errors. This means that in such a case, the application should not crash but rather try to reconnect automatically as quickly as possible. By using a connection pool, we are able to do this without having to implement a solution on our own. A connection pool is a cache of database connection objects. They offer some benefits like the following (taken from [Ora09]):

- Reducing the number of times new connection objects are created;
- Promoting connection object reuse;
- Accelerating the process of getting a connection;
- Reducing the amount of effort required to manually manage connection objects;
- Minimizing the number of stale connections;
- Controlling the amount of resources spent on maintaining connections.

Overall, they help database intensive applications to increase their performance and reliability.

\(^3\)see https://cwernet.de/ba-thesis/
5. Implementation

The following chapter describes the implementation of the aggregation layer. In the classical three-tier architecture of a web service with client applications and a data store, it represents the application tier in the middle of the other two (see fig. 5.1). Inside the application, we have two parts:

- The HTTP server communicates with the client applications by accepting HTTP requests and creating as well as sending the corresponding HTTP responses in the JSON format.

- The business logic processes the requests accepted by the HTTP server, queries the needed data from the data store tier (in this case an Oracle relational database) and delivers this data back to the server so that it can parse it into the desired response format.

![Three-tier architecture](image)

**Figure 5.1.: Three-tier architecture**

Figure 5.2 shows an overview of all used technologies in the prototype, categorized into the data part (on the right) and the RESTful/HTTP server part (on the left). The whole API is implemented in Java.

Since the database is from Oracle, we use JDBI as well as the Oracle implementation of the JDBC driver to connect to it. To manage the connections to the database (e.g. for connection pooling and timings) the Oracle Universal Connection Pool (UCP) is used. On the server side we use Dropwizard with the integrated Jetty server and Jersey framework to provide a RESTful web service. On top of Dropwizard and the Jersey framework we use the Katharsis library to provide support for the JSON:API specification.
5. Implementation

This section describes the implementation of a resource in Katharsis using the Run resource as an example. Since all resources are implemented in the same way, the example can be transferred.

First, we need a class that represents the resource according to its definition in section 4.1.1:

```java
package ch.cern.cms.daq.oms.api.aggregation.model.run;
import io.katharsis.resource.annotations.*;

@JsonApiResource(type = RunMeta.RESOURCE_NAME + "s")
public class Run {

    @JsonApiId
    private Integer id;

    private Integer run_number;
    private Integer fill_number;

    // resource fields as private class attributes
    public Run() {
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    // public Getter and Setter to access the attributes
}
```

Listing 5.1.: POJO representing the basic Run resource

The `@JsonApiResource` annotation is the annotation which defines a resource. It requires the "type" parameter to be defined, in this case with the plural of the resource name. This parameter is then used to form a URL and also the type field of the JSON response (as
5.1. Implement a resource

seen in listing 3.7). @JsonApiId tells Katharsis, to use the annotated field of the resource as identifier of it. The annotation can only be present on fields which are either a simple type or implement the Serializable interface. To fill the resource with actual values, we need to implement a repository for it.

In order to do this, we first extend the ResourceRepositoryV2<Run, Integer> interface provided by Katharsis to define a custom ResourceList called RunList which will be needed when it comes to add metadata (see section 5.2) and links (see line 10ff of listing 5.2).

```
package ch.cern.cms.daq.oms.api.aggregation.model.run;

//...
@Path(RunMeta.RESOURCE_NAME + "s")
public interface RunRepository extends ResourceRepositoryV2<Run, Integer> {

//...
@Override
public Run findOne(Integer id, QuerySpec querySpec);

@Override
public RunList findAll(QuerySpec querySpec);

class RunList extends ResourceListBase<Run, RunMeta, DefaultPagedLinksInformation> {

//...}
}
```

Listing 5.2.: The custom RunRepository interface

The @Path annotation above the interface declaration identifies the URI path template to which the resource responds. In this case, the run resource can be retrieved with e.g. "GET http://<URL base>/runs". Listing 5.3 shows the class body of the implementation of the custom RunRepository interface. The class also extends the provided abstract ResourceRepositoryBase<T, I extends Serializable> class. This abstract class provides per default all necessary methods for a repository. To be able to make the API read-only working, one only needs to implement the findAll method since the findOne method is per default only calling the findAll method internally. We needed to override it non the less in order to be able to make the metadata (see section 5.2) work as intended. The findOne method is called, when a resource is accessed using its identifier, for example GET /<URL-base/runs/<run number>. The method takes two parameters: an Integer and a QuerySpec object. The Integer holds the requested Run number and the QuerySpec object all possible query options like filtering, projection, ordering or pagination. In order to avoid duplicate code, the findOne method is now adding the Run number as another filter to the QuerySpec object and passes this “extended” object on to the findAll method. The findAll method now calls the query builder to get the SQL-query fitting the API request. This query is then run on the database and the result is mapped to the POJO representing the resource using a custom mapper class to create a list of Run objects. This list is then used to apply all QuerySpecs in order to “sort out” all objects which do not fit these query parameters. Finally, the necessary steps to add the result metadata are performed (more on that in section 5.2) and the list is returned, converted to JSON and returned to the client. In order to deliver proper error messages, we also implemented the methods normally used to create, save or delete a resource to raise a UnsupportedActionException with the hint that the repository is read only.
5. Implementation

Listing 5.3.: Class body of the implementation of the RunRepository interface

5.2. Implement metadata

This section describes how to implement the two types of metadata in Katharsis. As mentioned in chapter 4, the first type describes the response as a whole. It holds information about how many resources were returned, the version of the resource, it indicates the order of the attributes inside the resource(s) in case of a projection and describes each field of the resource. The second type is a part of one resource and describes one or more attributes in detail, for example the unit of an attribute. While the Run resource uses both types of metadata this does not apply to all the resources (see appendix A.2 for more information on that).

In order to be able to add metadata to the resource, we first have to add the following field inclusive getters and setters to the POJO representing the resource:

The annotation tells Katharsis that this member carries an implementation of the MetaInformation interface. Since both the classes for the response metadata as well as
5.2. Implement metadata

Listing 5.4.: POJO member to include metadata.

```java
@JsonApiMetaInformation
private FindOneMeta meta;
```

The resource metadata implement the FindOneMeta interface which extends the MetaInformation interface, the member carries all the needed metadata. Figure 5.3 shows a class diagram of all classes that are used to create the metadata. The orange colored interfaces are provided by Katharsis; they indicate on the one hand that classes are used to store metadata and on the other hand help Katharsis to compute pagination links by providing the total number of returned resources (PagedMetaInformation). The green colored classes are generic classes and interfaces of the aggregation layer; they are used by all resources. The blue colored classes contain resource specific implementations of the metadata. The RunMeta class stores all attributes names and their specific descriptions as seen in table A.4. AggregationResourceMeta is in this case the class representing one of these field descriptions with title, sourceType, apiType, units, description and comments. RowMeta provides general information on the response, like type of resource, version, number of resources in the response, order of the fields and the fields contained in a resource (to indicate filtering and/or projection). The violet colored class is the POJO representing the resource itself.

Now that the response specific metadata is working, we only need to implement the resource specific metadata that is so far used in both the Run and the Fill resources because the units for example the Luminosity could be differently scaled for two different Fills or Runs. To do that, the RowMeta class is used to represent the metadata object itself which contains a Map of String and UnitsMeta pairs where the String indicates the field and the UnitsMeta the corresponding unit. During the mapping of the database results to the POJO, this kind of metadata is added to the resource using a custom ConverterService (see section 5.4) to map the Run to the corresponding scaling factor which is then applied to the value of the database.

As mentioned before, in order to add links to the resources, we have defined a custom ResourceList. It extends the base implementation provided by Katharsis and specifies which classes are used to provide the metadata and the links information. Once this is defined, Katharsis is filling in these information automatically.

While the resource metadata and the totalResourceCount is always included, the rest of the response metadata is only delivered when adding the parameter include=meta into the request, for example GET api/v1/runs/69120?include=meta.
Figure 5.3.: UML diagram of the classes needed for the metadata.
5.3. Implement Relationships

In order to have relationship links included we have to first add a new field for each relation to the POJO class:

```java
@ToJsonApiRelation(lookUp=LookupIncludeBehavior.NONE)
private Fill fills;
```

Listing 5.5.: POJO member to include a relation to the Fill resource

The annotation indicates that the resource has “an association to either a single value or collection of resources”\(^1\) while the type of the field also has to be a resource. The lookUp=LookupIncludeBehavior.NONE parameter of the annotation tells Katharsis, that a corresponding implementation of the RelationshipRepositoryV2 interface is responsible for returning the related resources. This implementation looks like listing 5.6.

```java
package ch.cern.cms.daq.oms.api.aggregation.model.run;
//...
public class RunToFillRepository implements RelationshipRepositoryV2<Run, Integer, Fill, Integer> {
  private RunRepositoryImpl runRepository;
  private FillRepositoryImpl fillRepository;
  public RunToFillRepository() {
    }
  @Inject
  public RunToFillRepository (RunRepositoryImpl runRep, FillRepositoryImpl fillRepository) {
    this.runRepository = runRep;
    this.fillRepository = fillRepository;
  }
  @Override
  public Class<Run> getSourceResourceClass() {
    return Run.class;
  }
  @Override
  public Class<Fill> getTargetResourceClass() {
    return Fill.class;
  }
  // Override methods to set, add or remove a relation in order to raise a proper error since the repository is read-only
  @Override
  public Fill findOneTarget(Integer sourceId, String fieldName, QuerySpec querySpec) {
    Run run = runRepository.findOne(sourceId, querySpec);
    return fillRepository.findOne(run.getFill_number(), querySpec);
  }
  @Override
  public ResourceList<Fill> findManyTargets(Integer sourceId, String fieldName, QuerySpec querySpec) {
    return null;
  }
}
```

Listing 5.6.: RelationshipRepository for the relation from Run to Fill

As you can see, to find the corresponding Fill resource we only take the fill_number from the Run and look it up in the Fill repository.

---

\(^1\)https://katharsis-jsonapi.readthedocs.io/en/latest/user-docs.html#relationships
5. Implementation

5.4. The Converter Service

Due to the database layout and the operation method of the different subsystems, some of the values stored in database are raw values, for example the different Luminosity values or the energy value. Depending on the Run they were taken in, we have to apply different scaling factors before delivering them to the presentation layer. The scaling factors are also stored in a table in the database. The service works in the following way:

The ConverterService is implemented as a singleton. During the creation of the class, it queries the database to get all available Run periods and the corresponding scale factors in order to not query the database each time it is used. Each ResourceRepository that needs the service holds the instance of this singleton. When a RowMapper implementation is called to map the database result on to the POJO, the Mapper asks the ConverterService for the associated RunPeriodScaling object based on the run_number. This RunPeriodScaling holds all scale factors used during the particular period. With it, the Mapper then asks the ConverterService to scale the value of the database according to the factor. Additionally, the RunPeriodScaling provides the RowMetaData to display the unit of the scaled value in the response object.

5.5. The Query Builder

The query builder was a challenge. To implement it for the Run resource, we started by looking at the Structured Query Language (SQL) queries and functions used in the current WBM to retrieve the different fields of one Run (see the queries and functions in Appendix A.3). Several queries and functions are used to gather the attributes from different tables. That meant that we first had to combine them all into one big query. This big query was then rewritten to move necessary joins with tables other than the CMS_WBM.RUNSUMMARY table to the column selects where they were needed. This means, that all columns which are not in the runsummary table or are queried with a function are now queried with a subquery inside the big SELECT statement, for example the column holds the data for the fill_type_party2 attribute of the Run resource (see listing 5.7).

```sql
SELECT * FROM (SELECT
    R.RUNNUMBER,
    (Select PARTY2 from CMS_RUNTIME_LOGGER.RUNTIME_SUMMARY where LHCFILL=R.LHCFILL),
    CMS_RUNTIME_LOGGER.RUNTIME_GET_RUN_DELIVLUMI(R.RUNNUMBER),
FROM CMS_WBM.RUNSUMMARY R
WHERE R.RUNNUMBER is not null;
```

Listing 5.7.: Inner SQL query for the fill_type_party2 attribute of the Run resource.

This allows us to remove columns which are maybe not requested through the API. To create the query for each request, we first iterate over the QuerySpec object of the request to read out all IncludeFieldSpec objects representing the requested attributes
to add them to the `SELECT` statement. Then we iterate over the contained `FilterSpec` objects to build and add necessary `WHERE` clauses. As a third step, we read out the sorting parameters of the request in order to add fitting `ORDER BY` clauses.
6. Evaluation

After implementing the first prototype of the aggregation layer of the WBM upgrade, this chapter looks back to evaluate the work done and compare it to the initial requirements. As a first evaluation, the function of the prototype is examined. As a second evaluation, the performance is being measured as one of the crucial characteristics of a web service.

6.1. Functional evaluation

First of all it is to say that the prototype is up and running and serves the data to the portlets of the presentation layer team. It consist in total of 135 classes in 39 packages. By looking at the identified requirements in chapter 3, all of them have been fulfilled:

- It follows the RESTful architectural style by having the five necessary characteristics REST asks for and therefore also uses HATEOAS as well as it provides metadata information:
  - The prototype is based on a **client-server architecture** with the aggregation layer as server and the presentation layer as client.
  - It is **stateless**: no state of the sessions is kept in any way on the side of the aggregation layer.
  - All responses by the aggregation layer can be cached by the client although they are not explicitly labeled as **cacheable**. Since the whole API is read-only, this is not necessary.
  - All resources are identified by a URI under which they can be requested via a **HTTP GET** and are represented by the JSON format. Each response of the server includes on demand metadata to describe the members of the resource as well as hypermedia information on how to navigate to the resource itself as well as to related resources or in case of a paginated response to the next and/or previous page. These characteristics fulfill the point of the **uniform interface**.
  - The aggregation layer itself is built as a **layered system**. These layers are the HTTP server, the business logic and the database connection layer.
- The definition of the API as well as the design of the response objects are based on the JSON:API specification.
- The data needed to provide the essential services as described in section 3.1 is completely accessible through the aggregation layer.
6. Evaluation

Additionally, we implemented two resources used to draw a so called static plot. These static plots are periodically generated, stored on disk and shown within the portlets. They are not interactively accessible because there are too many data points in one plot to render them in a reasonable amount of time. On request of the presentation layer team, we also included special request parameters apart from the common options projection, filtering, ordering and pagination so that they were able to get the data faster where needed. These special request parameters are the following two:

- The “turbo button” (include=turbo). This parameter is named after the identically named button on PCs of the 8086s- to Pentium-1 era. This option is available for the Run, Fill, InstLumi and VacuumSummary resources and works along with the page[limit] pagination parameter. Normally, the aggregation layer runs two queries. The first one retrieves the requested items (the number specified in the per default available page[limit] parameter) and the second one calculates the number of possible items in the database. The turbo parameter tells the aggregation layer to only run the database query to get the requested items. This means, that the total number of resources in the response is equal to the number of requested items and there is no indication that there might be more items in the database.

- The dataonly option (include=dataonly). This parameter is available for the InstLumi and VacuumSummary resources. Since they are used to plot the static plots, they do not need any accompanying metadata. For this reason, the dataonly option tells the aggregation layer to only have the data in the response. No metadata, no links and no information on version or type of resource.

It is to say that due to time restrictions a query builder was only implemented for the Run and the Fill resource being the biggest resources. Also, it makes only limited sense to implement one for the remaining resources since they are mostly fully requested or have too few attributes to make a difference.
6.2. Performance

To measure the performance of the aggregation layer and to tune it a bit some test were performed. The test used the InstLumi and VacuumSummary resource (normally used for the static plots, see fig. 6.1) which hold for the longest Fill so far (Fill number 5045 with a length of 37 hours) around 91 thousand rows in the database.

During the performance test, these rows were split into ten time intervals holding first ten thousand then twenty thousand, thirty thousand, ... rows to avoid caching in the database. The amount of requested rows is therefore increasing with each interval. To collect enough measurements, each interval was requested a hundred times. A setup in the CMS server farm at Point 5 was used to perform the test (see fig. 6.2).

Figure 6.1.: The static plots plotted with data from the InstLumi and VacuumSummary resources.

Figure 6.2.: The setup for the performance test in Point 5.
6. Evaluation

The test shows how the aggregation layer performs when having to deal with several clients requesting data at the same time. The measurements were performed with the VacuumSummary resource and the requests included the parameter to include only the data.

Figure 6.3.: Results of the performance test.

Figure 6.3 shows the result of these measurements. On the x-axis you can see the number of records queried from the database and on the y-axis you can see the time in seconds of the request latency. While the performance with only 10,000 records requested is nearly the same regardless of the number of clients, you can see that this changes as the number of queried records increases. The slowest performance is with six clients around 12 seconds to retrieve the nearly 91,000 records while the slowest performance with one client is around nine seconds. Considering that one record has a size of 391 Bytes, the aggregation layer is receiving, processing and sending a total amount of \( \approx 34 \text{ Megabytes} \) in these times. Calculated down to one record this is about 1.31 respective 0.98 milliseconds. These values are very good considering that the records have to be queried from the database and parsed into the JSON format before being sent to the client.
7. Conclusion

The aim of this thesis was to design and implement a first prototype for the aggregation layer of the WBM upgrade project. Through designing the API to be used by the presentation layer, selecting a fitting technology and implementing the API in this technology, this has been achieved. Problems like the challenging database layout in terms of timing or setting the data in relation could be solved by designing the query builder and implementing the converter service.

In retrospective, the aggregation layer for the first prototype of the WBM upgrade is a solid piece of software. It works well and serves the responses in a reasonable amount of time so that the presentation layer does perform without any lagging.

General conclusions could be that it is challenging to design an API that fits an already existing database layout where the resources to be used as data representation are different from the tables of the database so that using available ORM software is no good idea. While working with the Katharsis library it became apparent that the library is great to set up a JSON:API compliant REST API using java as programming language. In combination with Dropwizard it forms a robust server which can be tweaked to make the best out of the available hardware.

For further works during the project it makes sense to implement a query builder for each resource in order to provide a homogeneous codebase for each resource. In this respect, it might also be a thought to consider implementing a second service providing the SQL snippets necessary to build the query in the query builder so that they can be stored in for example a database and edited via a GUI. This would allow editing from people not familiar with the code in case the database layout is changing in the future which could at least in parts prevent one of the problems that led to this upgrade project. The same idea goes for the metadata of each resource which is currently also hard coded. One could combine these two ideas and provide a metadata catalog like the metadata layer of the data warehouse architecture.
Bibliography


A. Appendix

A.1. JSON Example

```json
{
  "meta": {
    "totalResourceCount": 1,
    "resource": "era",
    "version": "1.0.0",
    "fields_order": [
      "name",
      "start_time",
      "end_time"
    ],
  },
  "fields": {
    "start_time": {
      "title": "Start time first run",
      "sourceType": "timestamp(9)",
      "apiType": "integer",
      "units": "sec",
      "description": "Time when the first run of this era period was started",
      "comments": ""
    },
    "name": {
      "title": "Name",
      "sourceType": "varchar2(40)",
      "apiType": "string",
      "units": "",
      "description": "Era name",
      "comments": ""
    },
    "end_time": {
      "title": "Stop time last run",
      "sourceType": "timestamp(9)",
      "apiType": "integer",
      "units": "sec",
      "description": "Time when the last run of this era period was stopped",
      "comments": ""
    }
  }
}
```

Listing A.1.: Metadata part of a response

A.2. API Resources
### Table A.1.: DAQ-Readout resource

<table>
<thead>
<tr>
<th>Name</th>
<th>Units (example)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>init_lumi</td>
<td>$10^{29}$ cm$^{-2}$s$^{-1}$</td>
<td></td>
</tr>
<tr>
<td>end_lumi</td>
<td>$10^{29}$ cm$^{-2}$s$^{-1}$</td>
<td></td>
</tr>
<tr>
<td>delivered_lumi</td>
<td>$\mu$b$^{-1}$</td>
<td></td>
</tr>
<tr>
<td>recorded_lumi</td>
<td>$\mu$b$^{-1}$</td>
<td></td>
</tr>
<tr>
<td>energy</td>
<td>GeV</td>
<td></td>
</tr>
<tr>
<td>b_field</td>
<td>T</td>
<td></td>
</tr>
</tbody>
</table>

### Table A.2.: Metadata that must be included with each Run resource

<table>
<thead>
<tr>
<th>Name</th>
<th>Units (example)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>init_lumi</td>
<td>$10^{29}$ cm$^{-2}$s$^{-1}$</td>
<td></td>
</tr>
<tr>
<td>peak_lumi</td>
<td>$10^{29}$ cm$^{-2}$s$^{-1}$</td>
<td></td>
</tr>
<tr>
<td>peak_specific_lumi</td>
<td>$10^{21}$ cm$^{-2}$s$^{-1}$</td>
<td></td>
</tr>
<tr>
<td>delivered_lumi</td>
<td>$\mu$b$^{-1}$</td>
<td></td>
</tr>
<tr>
<td>recorded_lumi</td>
<td>$\mu$b$^{-1}$</td>
<td></td>
</tr>
<tr>
<td>intensity_beam1</td>
<td>$10^{11}$</td>
<td></td>
</tr>
<tr>
<td>intensity_beam2</td>
<td>$10^{11}$</td>
<td></td>
</tr>
<tr>
<td>crossing_angle</td>
<td>$\mu$rad</td>
<td></td>
</tr>
<tr>
<td>beta_star</td>
<td>cm</td>
<td></td>
</tr>
</tbody>
</table>

### Table A.3.: Metadata that must be included with each Fill resource

<table>
<thead>
<tr>
<th>Name</th>
<th>Units (example)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>energy</td>
<td>GeV</td>
<td></td>
</tr>
<tr>
<td>b_field</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>init_lumi</td>
<td>$10^{29}$ cm$^{-2}$s$^{-1}$</td>
<td></td>
</tr>
</tbody>
</table>
### API Resources

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Source Type</th>
<th>API Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>run_number</td>
<td>Run</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td></td>
<td>Run number</td>
</tr>
<tr>
<td>fill_number</td>
<td>Fill</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td></td>
<td>Fill number</td>
</tr>
<tr>
<td>sequence</td>
<td>Sequence</td>
<td>VARCHAR2(4000)</td>
<td>string</td>
<td></td>
<td>Run type</td>
</tr>
<tr>
<td>fill_type_runtime</td>
<td>Fill Type</td>
<td>NUMBER</td>
<td>string</td>
<td></td>
<td>Fill type</td>
</tr>
<tr>
<td>fill_type_party1</td>
<td>Fill Type Party 1</td>
<td>NUMBER</td>
<td>string</td>
<td></td>
<td>Fill Type Party 1</td>
</tr>
<tr>
<td>fill_type_party2</td>
<td>Fill Type Party 2</td>
<td>NUMBER</td>
<td>string</td>
<td></td>
<td>Fill Type Party 2</td>
</tr>
<tr>
<td>stable_beam</td>
<td>StableBeam</td>
<td>N/A</td>
<td>boolean</td>
<td></td>
<td>Stable beam declared</td>
</tr>
<tr>
<td>delivered_lumi</td>
<td>Delivered Lumi</td>
<td>FLOAT(63)</td>
<td>fraction</td>
<td>6</td>
<td>Integrated stable luminosity delivered by LHC</td>
</tr>
<tr>
<td>recorded_lumi</td>
<td>Recorded Lumi</td>
<td>FLOAT(63)</td>
<td>fraction</td>
<td>6</td>
<td>Integrated stable luminosity delivered by CMS</td>
</tr>
<tr>
<td>start_time</td>
<td>StartTime</td>
<td>TIMESTAMP(6)</td>
<td>integer</td>
<td>sec</td>
<td>Time when the run was started</td>
</tr>
<tr>
<td>end_time</td>
<td>StopTime</td>
<td>TIMESTAMP(6)</td>
<td>integer</td>
<td>sec</td>
<td>Time when the run was stopped</td>
</tr>
<tr>
<td>trigger_mode</td>
<td>TriggerMode</td>
<td>VARCHAR2(256)</td>
<td>string</td>
<td></td>
<td>Running mode of CMS</td>
</tr>
<tr>
<td>l1_key</td>
<td>L1 Key</td>
<td>VARCHAR2(256)</td>
<td>string</td>
<td></td>
<td>L1 trigger configuration key</td>
</tr>
<tr>
<td>hlt_key</td>
<td>HLT Key</td>
<td>VARCHAR2(256)</td>
<td>string</td>
<td></td>
<td>HLT configuration key</td>
</tr>
<tr>
<td>b_field</td>
<td>B Field</td>
<td>FLOAT(63)</td>
<td>fraction</td>
<td>3</td>
<td>Magnetic field</td>
</tr>
<tr>
<td>tier0_transfer</td>
<td>Tier0 Transfer</td>
<td>NUMBER(1)</td>
<td>boolean</td>
<td></td>
<td>Transfer data to tier0</td>
</tr>
<tr>
<td>components</td>
<td>Components</td>
<td>VARCHAR2(1000)</td>
<td>array of strings</td>
<td></td>
<td>List of subsystems included into the Run</td>
</tr>
<tr>
<td>initial_prescale_index</td>
<td>Initial Prescale Index</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td></td>
<td>Initial prescale index</td>
</tr>
<tr>
<td>cmsww_version</td>
<td>Online Version</td>
<td>VARCHAR2(4000)</td>
<td>string</td>
<td></td>
<td>CMSWW version</td>
</tr>
<tr>
<td>l1_rate</td>
<td>L1 Rate</td>
<td>NUMBER(14.4)</td>
<td>fraction</td>
<td>3</td>
<td>L1 rate</td>
</tr>
<tr>
<td>energy</td>
<td>LHC Energy</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td></td>
<td>LHC energy</td>
</tr>
<tr>
<td>clock_type</td>
<td>Clock type</td>
<td>VARCHAR2(4000)</td>
<td>string</td>
<td></td>
<td>Clock type</td>
</tr>
<tr>
<td>init_lumi</td>
<td>Initial Lumi</td>
<td>FLOAT(63)</td>
<td>fraction</td>
<td>6</td>
<td>Luminosity at the beginning of the run</td>
</tr>
<tr>
<td>end_lumi</td>
<td>Ending Lumi</td>
<td>FLOAT(63)</td>
<td>fraction</td>
<td>6</td>
<td>Luminosity at the end of the run</td>
</tr>
<tr>
<td>l1_triggers_counter</td>
<td>L1 Triggers</td>
<td>NUMBER(20)</td>
<td>integer</td>
<td></td>
<td>Number of L1 triggers</td>
</tr>
<tr>
<td>hlt_physics_rate</td>
<td>HLT Rate</td>
<td>NUMBER(20)</td>
<td>fraction</td>
<td>3</td>
<td>HLT rate for Physics streams</td>
</tr>
<tr>
<td>hlt_physics_counter</td>
<td>HLT Triggers</td>
<td>NUMBER(20)</td>
<td>integer</td>
<td></td>
<td>HLT triggers for Physics streams</td>
</tr>
<tr>
<td>hlt_physics_size</td>
<td>HLT size</td>
<td>NUMBER(20)</td>
<td>fraction</td>
<td>3</td>
<td>HLT size for Physics streams</td>
</tr>
<tr>
<td>hlt_physics_throughput</td>
<td>HLT Data Rate</td>
<td>NUMBER(20)</td>
<td>fraction</td>
<td>3</td>
<td>HLT data rate for Physics streams</td>
</tr>
</tbody>
</table>

Table A.4.: Run resource
### A. Appendix

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Source Type</th>
<th>API Type</th>
<th>Units</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill_number</td>
<td>Fill</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td></td>
<td>Fill number</td>
<td></td>
</tr>
<tr>
<td>fill_type_runtime</td>
<td>Fill Type</td>
<td>NUMBER</td>
<td>string</td>
<td></td>
<td>Fill type</td>
<td>Possible values: PROTONS = 1, PB = 2, PROTONS_PB = 21</td>
</tr>
<tr>
<td>fill_type_party1</td>
<td>Fill Type Party 1</td>
<td>NUMBER(1)</td>
<td>string</td>
<td></td>
<td>Fill Type Party 1</td>
<td>Possible values: UNKNOWN = 0, PROTON = 1, PB82 = 2</td>
</tr>
<tr>
<td>fill_type_party2</td>
<td>Fill Type Party 2</td>
<td>NUMBER(1)</td>
<td>string</td>
<td></td>
<td>Fill Type Party 2</td>
<td>Possible values: UNKNOWN = 0, PROTON = 1, PB82 = 2</td>
</tr>
<tr>
<td>era</td>
<td>Era</td>
<td>VARCHAR2(40)</td>
<td>string</td>
<td></td>
<td>Era Name</td>
<td>To be used for filtering by era. Use same names as in /eras</td>
</tr>
<tr>
<td>start_time</td>
<td>Create Time (declared)</td>
<td>TIMESTAMP(9)</td>
<td>integer</td>
<td>sec</td>
<td>Time when the fill was created</td>
<td></td>
</tr>
<tr>
<td>start_stable_beam</td>
<td>Begin Time (stable)</td>
<td>TIMESTAMP(9)</td>
<td>integer</td>
<td>sec</td>
<td>Time when the stable beam was declared</td>
<td>Null if no stable beam</td>
</tr>
<tr>
<td>end_stable_beam</td>
<td>End Time (stable beam)</td>
<td>TIMESTAMP(9)</td>
<td>integer</td>
<td>sec</td>
<td>Time when the stable beam was stopped (Warning from LHC)</td>
<td>Null if no stable beam or run is ongoing</td>
</tr>
<tr>
<td>end_time</td>
<td>End Time (dumped)</td>
<td>TIMESTAMP(9)</td>
<td>integer</td>
<td>sec</td>
<td>Time when the beams were dumped or start of new fill</td>
<td>Null if run is ongoing</td>
</tr>
<tr>
<td>energy</td>
<td>Energy</td>
<td>FLOAT(32)</td>
<td>fraction</td>
<td>(3)</td>
<td>LHC target energy</td>
<td></td>
</tr>
<tr>
<td>b_field</td>
<td>BField</td>
<td>FLOAT(63)</td>
<td>fraction</td>
<td>(3)</td>
<td>Magnetic field</td>
<td></td>
</tr>
<tr>
<td>init_lumi</td>
<td>Initial.Lumi</td>
<td>FLOAT(63)</td>
<td>fraction</td>
<td>(6)</td>
<td>Luminosity at the beginning of the fill</td>
<td></td>
</tr>
<tr>
<td>peak_lumi</td>
<td>PeakLumi</td>
<td>FLOAT(63)</td>
<td>fraction</td>
<td>(6)</td>
<td>Peak of instantaneous luminosity during the fill</td>
<td></td>
</tr>
<tr>
<td>peak_pileup</td>
<td>PeakPileup (interactions/BX)</td>
<td>FLOAT(63)</td>
<td>fraction</td>
<td>(6)</td>
<td>Peak pileup</td>
<td>Peak Pileup (interactions/BX)</td>
</tr>
<tr>
<td>peak_specific_lumi</td>
<td>PeakSpecificLumi</td>
<td>FLOAT(32)</td>
<td>fraction</td>
<td>(6)</td>
<td>Peak value of average specific luminosity</td>
<td></td>
</tr>
<tr>
<td>delivered_lumi</td>
<td>Delivered Lumi</td>
<td>FLOAT(63)</td>
<td>fraction</td>
<td>(6)</td>
<td>Integrated stable luminosity delivered by LHC</td>
<td></td>
</tr>
<tr>
<td>recorded_lumi</td>
<td>Recorded Lumi</td>
<td>FLOAT(63)</td>
<td>fraction</td>
<td>(6)</td>
<td>Integrated stable luminosity delivered by CMS</td>
<td></td>
</tr>
<tr>
<td>injection_scheme</td>
<td>Injection scheme</td>
<td>VARCHAR2(100)</td>
<td>string</td>
<td></td>
<td>Injection scheme</td>
<td></td>
</tr>
<tr>
<td>intensity_beam1</td>
<td>IntensityBeam1</td>
<td>FLOAT(32)</td>
<td>fraction</td>
<td>(6)</td>
<td>Beam 1 peak intensity</td>
<td></td>
</tr>
<tr>
<td>intensity_beam2</td>
<td>IntensityBeam2</td>
<td>FLOAT(32)</td>
<td>fraction</td>
<td>(6)</td>
<td>Beam 2 peak intensity</td>
<td></td>
</tr>
<tr>
<td>bunches_beam1</td>
<td>nBunchesBeam1</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td></td>
<td>Number of bunches beam 1</td>
<td></td>
</tr>
<tr>
<td>bunches_beam2</td>
<td>nBunchesBeam2</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td></td>
<td>Number of bunches beam 2</td>
<td></td>
</tr>
<tr>
<td>bunches_colliding</td>
<td>nCollidingBunches</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td></td>
<td>Number of colliding bunches measured by CMS</td>
<td></td>
</tr>
<tr>
<td>bunches_target</td>
<td>nTargetBunches</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td></td>
<td>Target number of colliding bunches by LHC configuration</td>
<td></td>
</tr>
<tr>
<td>crossing_angle</td>
<td>Crossing Angle</td>
<td>FLOAT(32)</td>
<td>fraction</td>
<td>(1)</td>
<td>Crossing angle by LHC configuration</td>
<td></td>
</tr>
<tr>
<td>beta_star</td>
<td>β*</td>
<td>FLOAT(32)</td>
<td>fraction</td>
<td>(1)</td>
<td>cm β*</td>
<td></td>
</tr>
<tr>
<td>to_ready</td>
<td>toReady (to HV on)</td>
<td>TIMESTAMP(9)</td>
<td>fraction</td>
<td>(3)</td>
<td>minutes Time from stable beam declared to all tracker HV on</td>
<td></td>
</tr>
<tr>
<td>to_dump_ready</td>
<td>toDumpReady</td>
<td>TIMESTAMP(9)</td>
<td>fraction</td>
<td>(3)</td>
<td>minutes Time from LHC message regarding the planned dump of beams to all tracker HV off</td>
<td></td>
</tr>
<tr>
<td>dump_ready_to_dump</td>
<td>dumpReadyToDump</td>
<td>TIMESTAMP(9)</td>
<td>fraction</td>
<td>(3)</td>
<td>minutes Time from all tracker HV off to beam dumped</td>
<td></td>
</tr>
</tbody>
</table>

Table A.5.: Fill resource
### A.2. API Resources

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Source Type</th>
<th>API Type</th>
<th>Units</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill_number</td>
<td>Fill</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td></td>
<td>Fill number</td>
<td></td>
</tr>
<tr>
<td>run_number</td>
<td>Run</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td></td>
<td>Run number</td>
<td></td>
</tr>
<tr>
<td>lumisection_number</td>
<td>Lumi Section</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td></td>
<td>Lumisection number</td>
<td></td>
</tr>
<tr>
<td>start_time</td>
<td>StartTime</td>
<td>TIMESTAMP(9)</td>
<td>integer</td>
<td>sec</td>
<td>Time when the lumisection was</td>
<td></td>
</tr>
<tr>
<td>end_time</td>
<td>StopTime</td>
<td>TIMESTAMP(9)</td>
<td>integer</td>
<td>sec</td>
<td>Time when the lumisection was</td>
<td>Null if lumisection is</td>
</tr>
<tr>
<td>init_lumi</td>
<td>InitialLumi</td>
<td>FLOAT(63)</td>
<td>fraction</td>
<td></td>
<td>Luminosity at the beginning</td>
<td></td>
</tr>
<tr>
<td>end_lumi</td>
<td>EndingLumi</td>
<td>FLOAT(63)</td>
<td>fraction</td>
<td></td>
<td>Luminosity at the end of the lumisection</td>
<td></td>
</tr>
<tr>
<td>delivered_lumi</td>
<td>Delivered Lumi</td>
<td>FLOAT(63)</td>
<td>fraction</td>
<td></td>
<td>Integrated stable luminosity</td>
<td></td>
</tr>
<tr>
<td>recorded_lumi</td>
<td>Recorded Lumi</td>
<td>FLOAT(63)</td>
<td>fraction</td>
<td></td>
<td>Integrated stable luminosity</td>
<td></td>
</tr>
</tbody>
</table>

**Table A.6.: Lumisection resource**

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Source Type</th>
<th>API Type</th>
<th>Units</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill_number</td>
<td>Fill</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td></td>
<td>Fill number</td>
<td></td>
</tr>
<tr>
<td>run_number</td>
<td>Run</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td></td>
<td>Run number</td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>Identifier</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td></td>
<td>Downtime Identifier</td>
<td></td>
</tr>
<tr>
<td>start_time</td>
<td>StartTime</td>
<td>TIMESTAMP(9)</td>
<td>integer</td>
<td>sec</td>
<td>Time when the downtime was started</td>
<td></td>
</tr>
<tr>
<td>end_time</td>
<td>StopTime</td>
<td>TIMESTAMP(9)</td>
<td>integer</td>
<td>sec</td>
<td>Time when the downtime was finished</td>
<td>Null if downtime is ongoing</td>
</tr>
<tr>
<td>lost_lumi</td>
<td>Lost Lumi</td>
<td>FLOAT(63)</td>
<td>fraction</td>
<td></td>
<td>Integrated luminosity lost during the downtime</td>
<td></td>
</tr>
<tr>
<td>group</td>
<td>Group</td>
<td>VARCHAR2(40)</td>
<td>string</td>
<td></td>
<td>Group category of the downtime</td>
<td>For e.g. DAQ, TRIGGER, GENERAL</td>
</tr>
<tr>
<td>category</td>
<td>Category</td>
<td>VARCHAR2(40)</td>
<td>string</td>
<td></td>
<td>Category of the downtime</td>
<td>For e.g. CSC_DAQ, CASTOR_DAQ</td>
</tr>
<tr>
<td>details</td>
<td>Details</td>
<td>VARCHAR2(1000)</td>
<td>string</td>
<td></td>
<td>Comments of the downtime</td>
<td></td>
</tr>
</tbody>
</table>

**Table A.7.: Downtime resource**
### A. Appendix

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Source Type</th>
<th>API Type</th>
<th>Units</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>run_number</td>
<td>Run Number</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td></td>
<td>Run Number</td>
<td></td>
</tr>
<tr>
<td>l1_key</td>
<td>L1 Summary Key</td>
<td>VARCHAR2(256)</td>
<td>string</td>
<td></td>
<td>L1 trigger configuration key</td>
<td></td>
</tr>
<tr>
<td>gt_key</td>
<td>GT Key</td>
<td>VARCHAR2(256)</td>
<td>string</td>
<td></td>
<td>GT configuration key</td>
<td>For RUN 1 GTKey instead for RUN 2 uGTKey</td>
</tr>
<tr>
<td>run_settings_key</td>
<td>RunSettingsKey</td>
<td>VARCHAR2(256)</td>
<td>string</td>
<td></td>
<td>Run settings key</td>
<td>For RUN 1 GTRunSettingsKey instead for RUN 2 runSettingsKey</td>
</tr>
<tr>
<td>l1_menu</td>
<td>L1 Menu</td>
<td>VARCHAR2(256)</td>
<td>string</td>
<td></td>
<td>L1 menu</td>
<td>Not yet available for RUN 2</td>
</tr>
<tr>
<td>gt_source</td>
<td>GT Source</td>
<td>Multiple sources</td>
<td>Object</td>
<td></td>
<td>GT source</td>
<td></td>
</tr>
<tr>
<td>muon_trigger_keys</td>
<td>MuonTriggersKeys</td>
<td>dictionary</td>
<td>array</td>
<td></td>
<td>Muon trigger keys</td>
<td></td>
</tr>
<tr>
<td>calo_trigger_keys</td>
<td>CalorimeterTriggerKeys</td>
<td>dictionary</td>
<td>array</td>
<td></td>
<td>Calorimeter trigger keys</td>
<td></td>
</tr>
</tbody>
</table>

**Table A.8.: L1ConfigurationKey resource**

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Source Type</th>
<th>API Type</th>
<th>Units</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>run_number</td>
<td>Run Number</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td></td>
<td>Run Number</td>
<td></td>
</tr>
<tr>
<td>bit</td>
<td>Bit</td>
<td>NUMBER(4)</td>
<td>integer</td>
<td></td>
<td>L1 algorithm trigger bit number</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>Name</td>
<td>VARCHAR2(1024)</td>
<td>string</td>
<td></td>
<td>L1 algorithm trigger name</td>
<td></td>
</tr>
<tr>
<td>mask</td>
<td>Mask</td>
<td>NUMBER(1)</td>
<td>boolean</td>
<td></td>
<td>True if the algorithm trigger bit is masked</td>
<td></td>
</tr>
<tr>
<td>pre_dt_before_prescale_counter</td>
<td>Trigger counts before prescaling</td>
<td>NUMBER(20)</td>
<td>integer</td>
<td></td>
<td>Number of triggers for this algorithm pre-deadtime and before prescale</td>
<td></td>
</tr>
<tr>
<td>pre_dt_before_prescale_rate</td>
<td>Trigger rate before prescaling</td>
<td>NUMBER(14,4)</td>
<td>fraction(3)</td>
<td>Hz</td>
<td>Rate of triggers for this algorithm pre-deadtime and before prescale</td>
<td></td>
</tr>
<tr>
<td>pre_dt_counter</td>
<td>Trigger counts pre-deadtime</td>
<td>NUMBER(20)</td>
<td>integer</td>
<td></td>
<td>Number of triggers for this algorithm pre-deadtime and after prescale</td>
<td></td>
</tr>
<tr>
<td>pre_dt_rate</td>
<td>Trigger rate pre-deadtime</td>
<td>NUMBER(14,4)</td>
<td>fraction(3)</td>
<td>Hz</td>
<td>Rate of triggers for this algorithm pre-deadtime and after prescale</td>
<td></td>
</tr>
<tr>
<td>post_dt_hlt_counter</td>
<td>Trigger counts post-deadtime (HLT)</td>
<td>NUMBER(20)</td>
<td>integer</td>
<td></td>
<td>Number of triggers for this algorithm post-deadtime from HLT</td>
<td></td>
</tr>
<tr>
<td>post_dt_hlt_rate</td>
<td>Trigger rate post-deadtime (HLT)</td>
<td>NUMBER(14,4)</td>
<td>fraction(3)</td>
<td>Hz</td>
<td>Rate of triggers for this algorithm post-deadtime from HLT</td>
<td></td>
</tr>
<tr>
<td>final_prescale</td>
<td>Final Prescale</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td></td>
<td>Final prescale</td>
<td></td>
</tr>
</tbody>
</table>

**Table A.9.: L1AlgorithmTriggers resource**
## A.2. API Resources

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Source Type</th>
<th>API Type</th>
<th>Units</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>run_number</td>
<td>Run</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td>Run number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>l1a_physics_counter</td>
<td>L1A Physics</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td>Number of L1A triggers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>l1a_physics_rate</td>
<td>L1A Physics Rate</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>Hz</td>
<td>Rate of L1A triggers</td>
<td></td>
</tr>
<tr>
<td>l1a_calibration_counter</td>
<td>L1A Calibration</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td>Number of L1A</td>
<td>calibration triggers</td>
<td></td>
</tr>
<tr>
<td>l1a_calibration_rate</td>
<td>L1A Calibration Rate</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>Hz</td>
<td>Rate of L1A calibration triggers</td>
<td></td>
</tr>
<tr>
<td>l1a_random_counter</td>
<td>L1A Random</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td>Number of L1A</td>
<td>random triggers</td>
<td></td>
</tr>
<tr>
<td>l1a_random_rate</td>
<td>L1A Random Rate</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>Hz</td>
<td>Rate of L1A random triggers</td>
<td></td>
</tr>
<tr>
<td>physics_generated_fdl_gt_counter</td>
<td>PhysicsGeneratedFDL_GT</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td>Number physics</td>
<td>triggers generated by FDL (GT)</td>
<td>GT till run 265395</td>
</tr>
<tr>
<td>physics_generated_fdl_gt_rate</td>
<td>PhysicsGeneratedFDL_GT Rate</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>Hz</td>
<td>Rate physics triggers generated by FDL (GT)</td>
<td>GT till run 265395</td>
</tr>
<tr>
<td>physics_generated_fdl_tcds_counter</td>
<td>PhysicsGeneratedFDL TCDS</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td>Number physics</td>
<td>triggers generated by FDL (TCDS)</td>
<td>TCDS from run 243052</td>
</tr>
<tr>
<td>physics_generated_fdl_tcds_rate</td>
<td>PhysicsGeneratedFDL TCDS Rate</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>Hz</td>
<td>Rate physics triggers generated by FDL (TCDS)</td>
<td>TCDS from run 243052</td>
</tr>
<tr>
<td>trigger_physics_lost_counter</td>
<td>TriggersPhysicsLost</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td>Number physics</td>
<td>lost triggers</td>
<td></td>
</tr>
<tr>
<td>trigger_physics_lost_rate</td>
<td>TriggersPhysicsLost</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>Hz</td>
<td>Rate physics lost triggers</td>
<td></td>
</tr>
<tr>
<td>trigger_physics_lost_beam_active_counter</td>
<td>TriggersPhysicsLost BeamActive</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td>Number physics lost triggers during stable beam</td>
<td></td>
<td></td>
</tr>
<tr>
<td>trigger_physics_lost_beam_active_rate</td>
<td>TriggersPhysicsLost BeamActive Rate</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>Hz</td>
<td>Rate physics lost triggers during stable beam</td>
<td></td>
</tr>
<tr>
<td>trigger_physics_lost_beam_inactive_counter</td>
<td>TriggersPhysicsLost BeamInactive</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td>Number physics lost triggers during not stable beam</td>
<td></td>
<td></td>
</tr>
<tr>
<td>trigger_physics_lost_beam_inactive_rate</td>
<td>TriggersPhysicsLost BeamInactive Rate</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>Hz</td>
<td>Rate physics lost triggers during not stable beam</td>
<td></td>
</tr>
</tbody>
</table>

Table A.10.: L1TriggerRates resource
<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Source Type</th>
<th>API Type</th>
<th>Units</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>run_number</td>
<td>Run</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td></td>
<td>Run number</td>
<td></td>
</tr>
<tr>
<td>total_deadtime_counter</td>
<td>Total</td>
<td>N/A</td>
<td>integer</td>
<td></td>
<td>Number DeadTime triggers</td>
<td>Generate from percent</td>
</tr>
<tr>
<td>total_deadtime_percent</td>
<td>Total Percent</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>%</td>
<td>Rate DeadTime triggers</td>
<td></td>
</tr>
<tr>
<td>tts_counter</td>
<td>TTS</td>
<td>N/A</td>
<td>integer</td>
<td></td>
<td>Number DeadTime triggers due to TTS</td>
<td>Generate from percent</td>
</tr>
<tr>
<td>tts_percent</td>
<td>TTS Percent</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>%</td>
<td>Rate DeadTime triggers due to TTS</td>
<td></td>
</tr>
<tr>
<td>trigger_rules_counter</td>
<td>Trigger Rules</td>
<td>N/A</td>
<td>integer</td>
<td></td>
<td>Number DeadTime triggers due to trigger rules</td>
<td></td>
</tr>
<tr>
<td>trigger_rules_percent</td>
<td>Trigger Rules Percent</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>%</td>
<td>Rate DeadTime triggers due to trigger rules</td>
<td></td>
</tr>
<tr>
<td>bunch_mask_counter</td>
<td>Bunch Mask</td>
<td>N/A</td>
<td>integer</td>
<td></td>
<td>Number DeadTime triggers due to bunch mask</td>
<td>Generate from percent</td>
</tr>
<tr>
<td>bunch_mask_percent</td>
<td>Bunch Mask Percent</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>%</td>
<td>Rate DeadTime triggers due to bunch mask</td>
<td></td>
</tr>
<tr>
<td>re_tri_counter</td>
<td>ReTri</td>
<td>N/A</td>
<td>integer</td>
<td></td>
<td>Number DeadTime triggers due to reTri</td>
<td>Generate from percent</td>
</tr>
<tr>
<td>re_tri_percent</td>
<td>ReTri Percent</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>%</td>
<td>Rate DeadTime triggers due to reTri</td>
<td></td>
</tr>
<tr>
<td>apve_counter</td>
<td>APVE</td>
<td>N/A</td>
<td>integer</td>
<td></td>
<td>Number DeadTime triggers due to APVE</td>
<td>Generate from percent</td>
</tr>
<tr>
<td>apve_percent</td>
<td>APVE Percent</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>%</td>
<td>Rate DeadTime triggers due to APVE</td>
<td></td>
</tr>
<tr>
<td>daq_backpressure_counter</td>
<td>DAQ Backpressure</td>
<td>N/A</td>
<td>integer</td>
<td></td>
<td>Number DeadTime triggers due to DAQ backpressure</td>
<td>Generate from percent</td>
</tr>
<tr>
<td>daq_backpressure_percent</td>
<td>DAQ Backpressure Percent</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>%</td>
<td>Rate DeadTime triggers due to DAQ backpressure</td>
<td></td>
</tr>
<tr>
<td>calibration_counter</td>
<td>Calibration</td>
<td>N/A</td>
<td>integer</td>
<td></td>
<td>Number DeadTime triggers due to calibration</td>
<td>Generate from percent</td>
</tr>
<tr>
<td>calibration_percent</td>
<td>Calibration Percent</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>%</td>
<td>Rate DeadTime triggers due to calibration</td>
<td></td>
</tr>
<tr>
<td>firmware_pause_counter</td>
<td>Firmware Pause</td>
<td>N/A</td>
<td>integer</td>
<td></td>
<td>Number DeadTime triggers due to firmware pause</td>
<td>Generate from percent</td>
</tr>
<tr>
<td>firmware_pause_percent</td>
<td>Firmware Pause Percent</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>%</td>
<td>Rate DeadTime triggers due to firmware pause</td>
<td></td>
</tr>
<tr>
<td>software_pause_counter</td>
<td>Software Pause</td>
<td>N/A</td>
<td>integer</td>
<td></td>
<td>Number DeadTime triggers due to software pause</td>
<td>Generate from percent</td>
</tr>
<tr>
<td>software_pause_percent</td>
<td>Software Pause Percent</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>%</td>
<td>Rate DeadTime triggers due to software pause</td>
<td></td>
</tr>
</tbody>
</table>

Table A.11.: L1OverallDeadTime resource
## A.2. API Resources

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Source Type</th>
<th>API Type</th>
<th>Units</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>run_number</td>
<td>Run</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td></td>
<td>Run number</td>
<td></td>
</tr>
<tr>
<td>total_deadtime_counter</td>
<td>Total</td>
<td>N/A</td>
<td>integer</td>
<td></td>
<td>Number DeadTime triggers</td>
<td>Generate from percent</td>
</tr>
<tr>
<td>total_deadtime_percent</td>
<td>Total Percent</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>%</td>
<td>Rate DeadTime triggers</td>
<td></td>
</tr>
<tr>
<td>tts_counter</td>
<td>TTS</td>
<td>N/A</td>
<td>integer</td>
<td></td>
<td>Number DeadTime triggers due to TTS</td>
<td>Generate from percent</td>
</tr>
<tr>
<td>tts_percent</td>
<td>TTS Percent</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>%</td>
<td>Rate DeadTime triggers due to TTS</td>
<td></td>
</tr>
<tr>
<td>trigger_rules_counter</td>
<td>Trigger Rules</td>
<td>N/A</td>
<td>integer</td>
<td></td>
<td>Number DeadTime triggers due to trigger rules</td>
<td></td>
</tr>
<tr>
<td>trigger_rules_percent</td>
<td>Trigger Rules Percent</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>%</td>
<td>Rate DeadTime triggers due to trigger rules</td>
<td></td>
</tr>
<tr>
<td>bunch_mask_counter</td>
<td>Bunch Mask</td>
<td>N/A</td>
<td>integer</td>
<td></td>
<td>Number DeadTime triggers due to bunch mask</td>
<td>Generate from percent</td>
</tr>
<tr>
<td>bunch_mask_percent</td>
<td>Bunch Mask Percent</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>%</td>
<td>Rate DeadTime triggers due to bunch mask</td>
<td></td>
</tr>
<tr>
<td>re_tri_counter</td>
<td>ReTri</td>
<td>N/A</td>
<td>integer</td>
<td></td>
<td>Number DeadTime triggers due to reTri</td>
<td>Generate from percent</td>
</tr>
<tr>
<td>re_tri_percent</td>
<td>ReTri Percent</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>%</td>
<td>Rate DeadTime triggers due to reTri</td>
<td></td>
</tr>
<tr>
<td>apve_counter</td>
<td>APVE</td>
<td>N/A</td>
<td>integer</td>
<td></td>
<td>Number DeadTime triggers due to APVE</td>
<td>Generate from percent</td>
</tr>
<tr>
<td>apve_percent</td>
<td>APVE Percent</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>%</td>
<td>Rate DeadTime triggers due to APVE</td>
<td></td>
</tr>
<tr>
<td>daq_backpressure_counter</td>
<td>DAQ Backpressure</td>
<td>N/A</td>
<td>integer</td>
<td></td>
<td>Number DeadTime triggers due to DAQ backpressure</td>
<td>Generate from percent</td>
</tr>
<tr>
<td>daq_backpressure_percent</td>
<td>DAQ Backpressure Percent</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>%</td>
<td>Rate DeadTime triggers due to DAQ backpressure</td>
<td></td>
</tr>
<tr>
<td>calibration_counter</td>
<td>Calibration</td>
<td>N/A</td>
<td>integer</td>
<td></td>
<td>Number DeadTime triggers due to calibration</td>
<td></td>
</tr>
<tr>
<td>calibration_percent</td>
<td>Calibration Percent</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>%</td>
<td>Rate DeadTime triggers due to calibration</td>
<td></td>
</tr>
<tr>
<td>firmware_pause_counter</td>
<td>Firmware Pause</td>
<td>N/A</td>
<td>integer</td>
<td></td>
<td>Number DeadTime triggers due to firmware pause</td>
<td>Generate from percent</td>
</tr>
<tr>
<td>firmware_pause_percent</td>
<td>Firmware Pause Percent</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>%</td>
<td>Rate DeadTime triggers due to firmware pause</td>
<td></td>
</tr>
<tr>
<td>software_pause_counter</td>
<td>Software Pause</td>
<td>N/A</td>
<td>integer</td>
<td></td>
<td>Number DeadTime triggers due to software pause</td>
<td>Generate from percent</td>
</tr>
<tr>
<td>software_pause_percent</td>
<td>Software Pause Percent</td>
<td>BINARY_FLOAT</td>
<td>fraction (2)</td>
<td>%</td>
<td>Rate DeadTime triggers due to software pause</td>
<td></td>
</tr>
</tbody>
</table>

Table A.12.: L1BeamActiveDeadTime resource
### Table A.13.: L1AlgorithmDetails resource

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Source Type</th>
<th>API Type</th>
<th>Units</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>run_number</td>
<td>Run number</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td></td>
<td>Run number</td>
<td></td>
</tr>
<tr>
<td>lumisection_number</td>
<td>LS number</td>
<td>NUMBER(5)</td>
<td>integer</td>
<td></td>
<td>Lumisection number</td>
<td></td>
</tr>
<tr>
<td>bit</td>
<td>Trigger bit</td>
<td>NUMBER(4)</td>
<td>integer</td>
<td></td>
<td>L1 algorithm trigger bit number</td>
<td></td>
</tr>
<tr>
<td>start_time</td>
<td>LS start time</td>
<td>long</td>
<td>integer</td>
<td>sec</td>
<td>Time when the lumisection has started</td>
<td></td>
</tr>
<tr>
<td>pre_dt_before_prescale_counter</td>
<td>Trigger counts before prescaling</td>
<td>NUMBER(20)</td>
<td>integer</td>
<td></td>
<td>Number of triggers for this algorithm pre-deadtime and before prescale</td>
<td></td>
</tr>
<tr>
<td>pre_dt_before_prescale_rate</td>
<td>Trigger rate</td>
<td>NUMBER(14,4)</td>
<td>fraction (3)</td>
<td>Hz</td>
<td>Rate of triggers for this algorithm pre-deadtime and before prescale</td>
<td></td>
</tr>
<tr>
<td>pre_dt_counter</td>
<td>Trigger counts</td>
<td>NUMBER(20)</td>
<td>integer</td>
<td></td>
<td>Number of triggers for this algorithm pre-deadtime and after prescale</td>
<td></td>
</tr>
<tr>
<td>pre_dt_rate</td>
<td>Trigger rate</td>
<td>NUMBER(14,4)</td>
<td>fraction (3)</td>
<td>Hz</td>
<td>Rate of triggers for this algorithm pre-deadtime and after prescale</td>
<td></td>
</tr>
<tr>
<td>post_dt_hlt_counter</td>
<td>Trigger counts</td>
<td>NUMBER(20)</td>
<td>integer</td>
<td></td>
<td>Number of triggers for this algorithm post-deadtime from HLT</td>
<td></td>
</tr>
<tr>
<td>post_dt_hlt_rate</td>
<td>Trigger rate</td>
<td>NUMBER(14,4)</td>
<td>fraction (3)</td>
<td>Hz</td>
<td>Rate of triggers for this algorithm post-deadtime from HLT</td>
<td></td>
</tr>
<tr>
<td>post_dt_ugt_counter</td>
<td>Trigger counts</td>
<td>NUMBER(20)</td>
<td>integer</td>
<td></td>
<td>Number of triggers for this algorithm post-deadtime from uGT</td>
<td></td>
</tr>
<tr>
<td>post_dt_ugt_rate</td>
<td>Trigger rate</td>
<td>NUMBER(14,4)</td>
<td>fraction (3)</td>
<td>Hz</td>
<td>Rate of triggers for this algorithm post-deadtime from uGT</td>
<td></td>
</tr>
</tbody>
</table>

### Table A.14.: InstLumi resource

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Source Type</th>
<th>API Type</th>
<th>Units</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>dip_time</td>
<td>Timestamp</td>
<td>TIMESTAMP(9)</td>
<td>integer</td>
<td>sec</td>
<td>Timestamp when the instantaneous luminosity was logged in DIP</td>
<td></td>
</tr>
<tr>
<td>inst_lumi</td>
<td>Best instantaneous luminosity</td>
<td>FLOAT(32)</td>
<td>fraction (2)</td>
<td></td>
<td>The instantaneous luminosity of the best luminometer</td>
<td></td>
</tr>
<tr>
<td>plt_inst_lumi</td>
<td>FLOAT(32)</td>
<td>fraction (2)</td>
<td></td>
<td></td>
<td>The instantaneous luminosity of the PLT luminometer</td>
<td></td>
</tr>
<tr>
<td>pltzero_inst_lumi</td>
<td>FLOAT(32)</td>
<td>fraction (2)</td>
<td></td>
<td></td>
<td>The instantaneous luminosity of the PLTzero luminometer</td>
<td></td>
</tr>
<tr>
<td>bcmf_inst_lumi</td>
<td>FLOAT(32)</td>
<td>fraction (2)</td>
<td></td>
<td></td>
<td>The instantaneous luminosity of the BCMF luminometer</td>
<td></td>
</tr>
<tr>
<td>hf_inst_lumi</td>
<td>FLOAT(32)</td>
<td>fraction (2)</td>
<td></td>
<td></td>
<td>The instantaneous luminosity of the HF luminometer</td>
<td></td>
</tr>
</tbody>
</table>
### A.2. API Resources

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Source Type</th>
<th>API Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name</td>
<td>VARCHAR2(40)</td>
<td>string</td>
<td></td>
<td>Era name</td>
</tr>
<tr>
<td>period</td>
<td>Period</td>
<td>int</td>
<td>integer</td>
<td></td>
<td>Index for era</td>
</tr>
<tr>
<td>start_time</td>
<td>Start time first run</td>
<td>TIMESTAMP(9)</td>
<td>integer</td>
<td></td>
<td>Time when the first run of this era period was started</td>
</tr>
<tr>
<td>end_time</td>
<td>Stop time last run</td>
<td>TIMESTAMP(9)</td>
<td>integer</td>
<td></td>
<td>Time when the last run of this era period was stopped</td>
</tr>
<tr>
<td>start_run</td>
<td>First run</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td></td>
<td>First run number of this era period</td>
</tr>
<tr>
<td>end_run</td>
<td>Last run</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td></td>
<td>Last run number of this era period</td>
</tr>
<tr>
<td>start_fill</td>
<td>Fill of first run</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td></td>
<td>First fill number of this era period</td>
</tr>
<tr>
<td>end_fill</td>
<td>Fill of last run</td>
<td>NUMBER(10)</td>
<td>integer</td>
<td></td>
<td>Last fill number of this era period</td>
</tr>
</tbody>
</table>

Table A.15.: Era resource

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Source Type</th>
<th>API Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Sequence Name</td>
<td>VARCHAR2(4000)</td>
<td>string</td>
<td></td>
<td>Sequence name</td>
</tr>
</tbody>
</table>

Table A.16.: Sequence resource

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Source Type</th>
<th>API Type</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dip_time</td>
<td>Timestamp</td>
<td>TIMESTAMP(9)</td>
<td>integer</td>
<td>sec</td>
<td>Time when the vacuum values were logged in DIP</td>
</tr>
<tr>
<td>vgi_183_1r5</td>
<td>Vacuum pressure at VGI_183_1R5</td>
<td>FLOAT(32)</td>
<td>fraction (12)</td>
<td>Torr</td>
<td>The vacuum pressure at VGI_183_1r5</td>
</tr>
<tr>
<td>vgi_183_1l5</td>
<td>Vacuum pressure at VGI_183_1L5</td>
<td>FLOAT(32)</td>
<td>fraction (12)</td>
<td>Torr</td>
<td>The vacuum pressure at VGI_183_1l5</td>
</tr>
<tr>
<td>vgi_220_1r5</td>
<td>Vacuum pressure at VGI_220_1R5</td>
<td>FLOAT(32)</td>
<td>fraction (12)</td>
<td>Torr</td>
<td>The vacuum pressure at VGI_220_1r5</td>
</tr>
<tr>
<td>vgi_220_1l5</td>
<td>Vacuum pressure at VGI_220_1L5</td>
<td>FLOAT(32)</td>
<td>fraction (12)</td>
<td>Torr</td>
<td>The vacuum pressure at VGI_220_1l5</td>
</tr>
<tr>
<td>vgp_7_4r5</td>
<td>Vacuum pressure at VGPB_7_4R5</td>
<td>FLOAT(32)</td>
<td>fraction (12)</td>
<td>Torr</td>
<td>The vacuum pressure at VGPB_7_4r5</td>
</tr>
<tr>
<td>vgp_7_4l5</td>
<td>Vacuum pressure at VGPB_7_4L5</td>
<td>FLOAT(32)</td>
<td>fraction (12)</td>
<td>Torr</td>
<td>The vacuum pressure at VGPB_7_4l5</td>
</tr>
<tr>
<td>vgp_147_1r5</td>
<td>Vacuum pressure at VGPB_147_1R5</td>
<td>FLOAT(32)</td>
<td>fraction (12)</td>
<td>Torr</td>
<td>The vacuum pressure at VGPB_147_1r5</td>
</tr>
<tr>
<td>vgp_147_1l5</td>
<td>Vacuum pressure at VGPB_147_1L5</td>
<td>FLOAT(32)</td>
<td>fraction (12)</td>
<td>Torr</td>
<td>The vacuum pressure at VGPB_147_1l5</td>
</tr>
<tr>
<td>vgp_148_1r5</td>
<td>Vacuum pressure at VGPB_148_1R5</td>
<td>FLOAT(32)</td>
<td>fraction (12)</td>
<td>Torr</td>
<td>The vacuum pressure at VGPB_148_1r5</td>
</tr>
<tr>
<td>vgp_148_1l5</td>
<td>Vacuum pressure at VGPB_148_1L5</td>
<td>FLOAT(32)</td>
<td>fraction (12)</td>
<td>Torr</td>
<td>The vacuum pressure at VGPB_148_1l5</td>
</tr>
<tr>
<td>vgp_183_1r5</td>
<td>Vacuum pressure at VGPB_183_1R5</td>
<td>FLOAT(32)</td>
<td>fraction (12)</td>
<td>Torr</td>
<td>The vacuum pressure at VGPB_183_1r5</td>
</tr>
<tr>
<td>vgp_183_1l5</td>
<td>Vacuum pressure at VGPB_183_1L5</td>
<td>FLOAT(32)</td>
<td>fraction (12)</td>
<td>Torr</td>
<td>The vacuum pressure at VGPB_183_1l5</td>
</tr>
<tr>
<td>vgp_220_1r5</td>
<td>Vacuum pressure at VGPB_220_1R5</td>
<td>FLOAT(32)</td>
<td>Double</td>
<td>Torr</td>
<td>The vacuum pressure at VGPB_220_1r5</td>
</tr>
<tr>
<td>vgp_220_1l5</td>
<td>Vacuum pressure at VGPB_220_1L5</td>
<td>FLOAT(32)</td>
<td>Double</td>
<td>Torr</td>
<td>The vacuum pressure at VGPB_220_1l5</td>
</tr>
</tbody>
</table>

Table A.17.: VacuumSummary resource
A. Appendix

A.3. SQL statements

```sql
-- (01) : big query to retrieve data for RUN resource
SELECT R.LHCFILL,
       R.RUNNUMBER,
       R.SEQUENCENAME,
       CMS_RUNTIME_LOGGER.RUNTIME.GET_RUN_INITLUMI(RUNNUMBER) INITLUMI,
       TO_CHAR(CMS_RUNTIME_LOGGER.RUNTIME.GET_RUN_LIVELUMI(RUNNUMBER), '9,999,990.000000') RUNLIVELUMI,
       CMS_RUNTIME_LOGGER.RUNTIME.GET_RUN_DELIVLUMI(RUNNUMBER) DELIVLUMI,
       R.STARTTIME,
       R.STOPTIME,
       R.TRIGGERMODE,
       R.BFIELD,
       CMS_WBM.GET_COMPONENTS(RUNNUMBER) COMPONENTS,
       R.TIER0_TRANSFER,
       R.LHCENERGY,
       TO_NUMBER(STRING_VALUE) FROM CMS_RUNINFO.RUNSESSION_PARAMETER G
WHERE G.RUNNUMBER=R.RUNNUMBER
AND G.NAME='CMS.SCAL:lumiFillBegin' LUMIFILLBEGIN,
(SELECT MAX(TO_NUMBER(STRING_VALUE)) FROM CMS_RUNINFO.RUNSESSION_PARAMETER G
WHERE G.RUNNUMBER=R.RUNNUMBER
AND G.NAME='CMS.SCAL:lumiFillEnd') LUMIFILLEND,
(SELECT MIN(TO_NUMBER(STRING_VALUE)) FROM CMS_RUNINFO.RUNSESSION_PARAMETER G
WHERE G.RUNNUMBER=R.RUNNUMBER
AND G.NAME='CMS.SCAL:liveLumiFillBegin') LIVELUMIFILLBEGIN,
(SELECT MAX(TO_NUMBER(STRING_VALUE)) FROM CMS_RUNINFO.RUNSESSION_PARAMETER G
WHERE G.RUNNUMBER=R.RUNNUMBER
AND G.NAME='CMS.SCAL:liveLumiFillEnd') LIVELUMIFILLEND,
R.TRIGGERS
FROM CMS_WBM.RUNSUMMARY R,
CMS_HLT_GDR.U_CONFVERSIONS C
WHERE R.runnumber=286516
and C.CONFIGID = R.HLTKEY;

-- (02) : stable beam
SELECT R.runnumber
FROM CMS_WBM.RUNSUMMARY R,
CMS_RUNTIME_LOGGER.RUNTIME_SUMMARY A
WHERE R.runnumber=286516
and R.LhcFill=5575
and (R.STARTTIME > A.BEGINTIME
or (R.stoptime > A.BEGINTIME or R.stoptime is null))
and LhcFill=A.LHCFILL
and TRG_PRESENT=1
and R.TRIGGERS>0
order by R.runnumber;
```

Listing A.2.: Queries to retrieve attributes of the Run resource as used in the WBM Part 1.
--- (03) : fill_type_runtime, fill_type_party1 and fill_type_pary2 attributes

```sql
select RUNTIME_TYPE_ID,
PARTY1,
PARTY2
from CMS_RUNTIME_LOGGER.RUNTIME_SUMMARY
where LHCFILL=5575;
```

--- (04) : initial_prescale_index

```sql
SELECT INITIALPRESCALEINDEX
FROM CMS_RUNTIME_LOGGER.INITIAL_PRESCALE_INDEX
WHERE RUNNUMBER=286516;
```

--- (05) : cmssw_version

```sql
SELECT STRING_VALUE
FROM CMS_RUNINFO.RUNSESSION_PARAMETER
WHERE NAME='CMS.DAQ:CMSSW_VERSION'
AND RUNNUMBER =286516;
```

--- (06) : L1 Rate Run before 243051

```sql
select max(trigger_rate_avg_hz) KEEP (DENSE_RANK LAST ORDER BY sample_time)
from CMS_GT_MON.SCALERS_TCS_GENERAL
where run_number = 180289;
```

--- (07) : L1 Rate run after 243051

```sql
select avg(TRG_RATE_TOTAL)
from CMS_TCDS_MONITORING.TCDS_CPM_RATES_V
where RUN_NUMBER=286516
AND TRG_RATE_TOTAL > 0;
```

--- (08) : hlt_triggers_physics (run < 243051)

```sql
SELECT SUM(S_NEVENTS)
FROM CMS_STOMGR.SM_SUMMARY
WHERE STREAM='A'
and RUNNUMBER=180289;
```

--- (09) : hlt_triggers_physics (run > 243051)

```sql
SELECT SUM(I.NEVENTS)
FROM CMS_STOMGR.FILES_CREATED H,
CMS_STOMGR.FILES_INJECTED I
WHERE H.runnumber=286516
AND H.FILENAME=I.FILENAME
AND (H.STREAM='A' OR H.STREAM LIKE 'Physics%');
```

--- (10) : hlt_size_physics_gb (run < 243051)

```sql
SELECT SUM(S_FILESIZE2T0)
FROM CMS_STOMGR.SM_SUMMARY K
WHERE STREAM='A'
and RUNNUMBER=180289;
```

--- (11) : hlt_size_physics_gb (run > 243051)

```sql
SELECT SUM(S_FILESIZE2T0)
FROM CMS_STOMGR.SM_SUMMARY K
WHERE STREAM='A'
and RUNNUMBER=180289;
```

--- (12) : hlt_triggers_physics (run < 243051)

```sql
SELECT SUM(S_NEVENTS)/(NVL(CMS_GT.FUNC_GET_LS_SEC_BY_RUNNR(180289),23.31040958)*
CMS_GT_MON.GET_MAX_LUMI_SECTION(180289))
from CMS_STOMGR.SM_SUMMARY
WHERE STREAM='A'
and RUNNUMBER=180289;
```

--- (13) : hlt_triggers_physics (run > 243051)

```sql
SELECT SUM(I.NEVENTS)/(NVL(CMS_GT.FUNC_GET_LS_SEC_BY_RUNNR(286516),23.31040958)*
CMS_GT_MON.GET_MAX_LUMI_SECTION(286516))
from CMS_STOMGR.SM_SUMMARY
WHERE STREAM='A'
and RUNNUMBER=180289;
```

--- (14) : hlt_size_physics_gb (run > 243051)

```sql
SELECT SUM(S_FILESIZE2T0)
FROM CMS_STOMGR.SM_SUMMARY K
WHERE STREAM='A'
and RUNNUMBER=180289;
```

Listing A.3.: Queries to retrieve attributes of the Run resource as used in the WBM Part 2.
A. Appendix

Listing A.4.: Queries to retrieve attributes of the Run resource as used in the WBM Part 3.

Listing A.5.: Functions to retrieve attributes of the Run resource as used in the WBM Part 1.
Listing A.6.: Functions to retrieve attributes of the Run resource as used in the WBM Part 2.
Listing A.7.: Functions to retrieve attributes of the Run resource as used in the WBM Part 3.
A.3. SQL statements

```sql
-- (09) : Function to retrieve endlumi
FUNCTION GET_RUN_ENDLUMI(RUN IN NUMBER) RETURN NUMBER IS
    
    REPLY LUMI_SECTIONS.INSTLUMI%TYPE;
    MAX_LS LUMI_SECTIONS.LUMISECTION%TYPE;
    
    BEGIN
        REPLY := 0;
        
        SELECT MAX(LUMISECTION) INTO MAX_LS
        
        FROM CMS_RUNTIME_LOGGER.LUMI_SECTIONS
        WHERE RUNNUMBER=RUN
            AND INSTLUMI>0.0000001
            AND BEAM1.STABLE=1
            AND BEAM2.STABLE=1;

        IF ( ( MAX_LS IS NOT NULL ) AND ( MAX_LS >= 1 ) ) THEN
            SELECT INSTLUMI INTO REPLY
            FROM LUMI_SECTIONS
            WHERE RUNNUMBER=RUN
                AND LUMISECTION=MAX_LS;
        
        IF ( REPLY IS NULL ) THEN
            END IF;
        ELSE
            REPLY := 0;
        END IF;
        
        EXCEPTION
            WHEN OTHERS THEN
                REPLY := 0;
        
        END;
        
    RETURN REPLY;

END GET_RUN_ENDLUMI;
```

Listing A.8.: Functions to retrieve attributes of the Run resource as used in the WBM Part 4.