Czech Technical University in Prague

Faculty of Nuclear Sciences and Physical Engineering

Department of Software Engineering
Course: Applied Software Engineering

COMPASS experiment data servers API

API datových serverů experimentu COMPASS

Master Thesis

Author: Bc. Matouš Jandek
Supervisor: Ing. Martin Bodlák
Year: 2018

České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská

Katedra softwarového inženýrství Akademický rok 2017/2018

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student:	**Bc. Matouš Jandek**
Studijní program:	**Aplikace přírodních věd**
Obor:	**Aplikace softwarového inženýrství**
Název práce česky:	**API datových serverů experimentu COMPASS**
Název práce anglicky:	**COMPASS Experiment Data Servers API**

Pokyny pro vypracování

1. Seznamte se s experimentem COMPASS a jeho systémem pro sběr dat.
2. Seznamte se s technologiemi umožňujícími implementaci rozhraní pro přístup k datům uloženým na datových serverech experimentu COMPASS.
3. Navrhněte API, které by zjednodušilo některé vybrané operace běžně prováděné na experimentu COMPASS.
4. Navržené řešení implementujte a otestujte.

Doporučená literatura:

[1] GAMMA, E., HELM, R., JOHNSON, R. a VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, Mass.: Addison-Wesley, c1995. ISBN 978-0-201-63361-0.

[2] STEVENS, R. W., FENNER, B. a RUDOLFF, A. M. *Unix Network Programming: The Sockets Networking API.* Třetí vydání. Boston: Addison-Wesley, 2004. ISBN 978-0-13-141155-5.

[3] WILLIAMS, A. *C++ Concurrency in Action: Practical Multithreading.* Shelter Island, NY: Manning, c2012. ISBN 978-1-933988-77-1.

[4] BELL, C. A. *Expert MySQL.* Druhé vydání. New York: Springer, c2012. ISBN 978-1-4302-4659-6.

[5] DAIGNEAU, R. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services.* Upper Saddle River, NJ: Addison-Wesley, c2012. ISBN 978-0-321-54420-9.

[6] BLANCHETTE, J. a SUMMERFIELD, M. *C++ GUI Programming with Qt 4.* Druhé vydání. Upper Saddle River, NJ: Prentice Hall, c2008. ISBN 978-0-13-235416-5.

Jméno a pracoviště vedoucího práce:

Ing. Martin Bodlák
Matematicko-fyzikální fakulta Univerzity Karlovy

Bodlák
.....
vedoucí práce

Datum zadání diplomové práce: 20. 10. 2017

Termín odevzdání diplomové práce: 7. 5. 2018

Doba platnosti zadání je dva roky od data zadání.

.....
vedoucí katedry

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta jaderná a fyzikálně inženýrská
-1-

.....
děkan

V Praze dne 20. 10. 2017

Statutory declaration

I hereby declare that I have elaborated this master thesis independently and used no other aids than those cited. In each individual case, I have clearly identified the source of the passages that are taken word by word or paraphrased from other works.

In Prague ....

Bc. Matouš Jandek

Acknowledgment

I would like to thank my family for supporting me in my effort. I would also like to thank my supervisor, Ing. Martin Bodlák, for guiding and assisting me during the work on this master thesis.

Bc. Matouš Jandek

Název práce:
API datových serverů experimentu COMPASS

Autor: Bc. Matouš Jandek

Obor: Aplikace softwarového inženýrství

Druh práce: Diplomová práce

Vedoucí práce: Ing. Martin Bodlák
Matematicko-fyzikální fakulta Univerzity Karlovy

Abstrakt: Během provozu experimentu COMPASS v CERN je potřeba zpracovávat velké množství digitálních dat. Počítačové systémy používané na tomto pracovišti, které se tímto úkolem zabývají, potřebují přistupovat k různým druhům digitálních úložišť, například k databázovým serverům nebo složkám sdíleným přes síť v CERN. Neexistuje ovšem žádné jednotné rozhraní, které by umožnilo s těmito zdroji, což ve svém důsledku způsobuje, že procedury pro přístup k datům musejí být implementovány několikrát v jednotlivých počítačových programech. Tato situace ústí nejen v delší vývojový čas nového softwarového vybavení a náročnější údržbu stávajících programů, ale i ve zvýšené riziko vzniku chyby v programech a tedy i riziko narušení intergrity dat. COMPASS API je navrhovaný software, který má za cíl napravit tuto situaci implementováním rozhraní s jasně definovanou strukturou, které by bylo využitelné z jiných aplikací, a které by umožnilo vykonávat manipulaci s dat sjednoceným způsobem, a tím snížit riziko poškození dat.

Klíčová slova: CERN, COMPASS, software, server, databáze, síť, složka, C++, Qt, API, RESTful

Title:
COMPASS experiment data servers API

Author: Bc. Matouš Jandek

Abstract: During operation of the COMPASS experiment at CERN, it is required to process large amounts of digital data. Computer systems used at the site that deal with this task need to access various types of digital storage, for example database servers or folders shared over the CERN network. However, no common interface that would allow to interact with these resources exists, and as a result, data-access procedures often have to be implemented multiple times in different computer programs. This situation not only results in longer development times of new sotware equipment and more demanding maintenance of existing programs, but also increases the possibility of errors and thus higher risk of compromising data integrity. COMPASS API is a proposed piece of software, which aims to correct these issues by implementing an interface with clearly defined structure which would be usable by other applications, and which would allow to perform data manipulation in a centralized and unified manner, thus reducing the risk of compromising data.

Keywords: CERN, COMPASS, software, server, database, network, folder, C++, Qt, API, RESTful

Contents

Introduction

The Common Muon and Proton Apparatus for Structure and Spectroscopy, abbreviated as COMPASS, is a high energy physics experiment with a fixed target, which is conducted at the CERN laboratory in Geneva. Since the COMPASS experiment uses a large number of complex measuring devices to achieve its specified goals, numerous computer systems are deployed to support the experimental setup.

Such systems include most prominently the Data Acquisition System (DAQ), which is used to collect data from the experiment's detectors and monitor the overall status of the equipment and its functionality. Other computer systems include for example the electronic logbook or various data analysis tools.

Most of these computer systems process identical data during their normal operation. To facilitate this, several methods of storage are utilized on the COMPASS experiment. However, none of them provide any interface to access the stored data. The absence of such unified procedure for data access has several drawbacks, which the COMPASS API intends to correct.

Chapter 1 introduces the COMPASS experiment. The goals of this high-energy physics experiment are outlined, and are subsequently followed by a basic description of the experimental setup.

Chapter 2 briefly describes the data acquisition system (DAQ) of the COMPASS experiment. The DAQ is one of the principal pieces of equipment on the COMPASS experiment, and therefore must be taken into consideration when deploying any additional program to COMPASS computer systems.

Chapter 3 presents the current state of several specific parts of COMPASS computer equipment, and summarizes the motives for development of the COMPASS API software. Furthermore, the requirements for the COMPASS API software are analyzed and listed.

Chapter 4 focuses on technologies that were used to develop the COMPASS API software. It lists the main features of utilized programming languages, libraries and frameworks, but also describes other aspects such as data formats or design concepts. Furthermore, reasons for the selection of listed technologies are elaborated.

Chapter 5 contains the description of the implementation of the COMPASS API. The software is documented using standard UML diagrams. Descriptions of key parts of the code are provided as well.

Chapter 6 deals with testing of the COMPASS API software. It describes the methodology used in this process and presents the results of performed tests.

Chapter 7 serves as a guideline for developers who aim to use the COMPASS API in their applications, or intend to expand the set of available functions implemented in COMPASS API.

Chapter 1

COMPASS experiment

COMPASS, or Common Muon and Proton Apparatus for Structure and Spectroscopy, is a high energy physics experiment which is located at the CERN laboratory in Geneva, Switzerland.

COMPASS is a fixed-target experiment, and is located in CERN laboratory's North Area. The particle beam is obtained from the Super Proton Synchrotron (SPS) accelerator, from where it is guided through an underground beamline to the COMPASS experiment's site. The location of the experiment within CERN's Geneva facilities is displayed in figure 1.1.

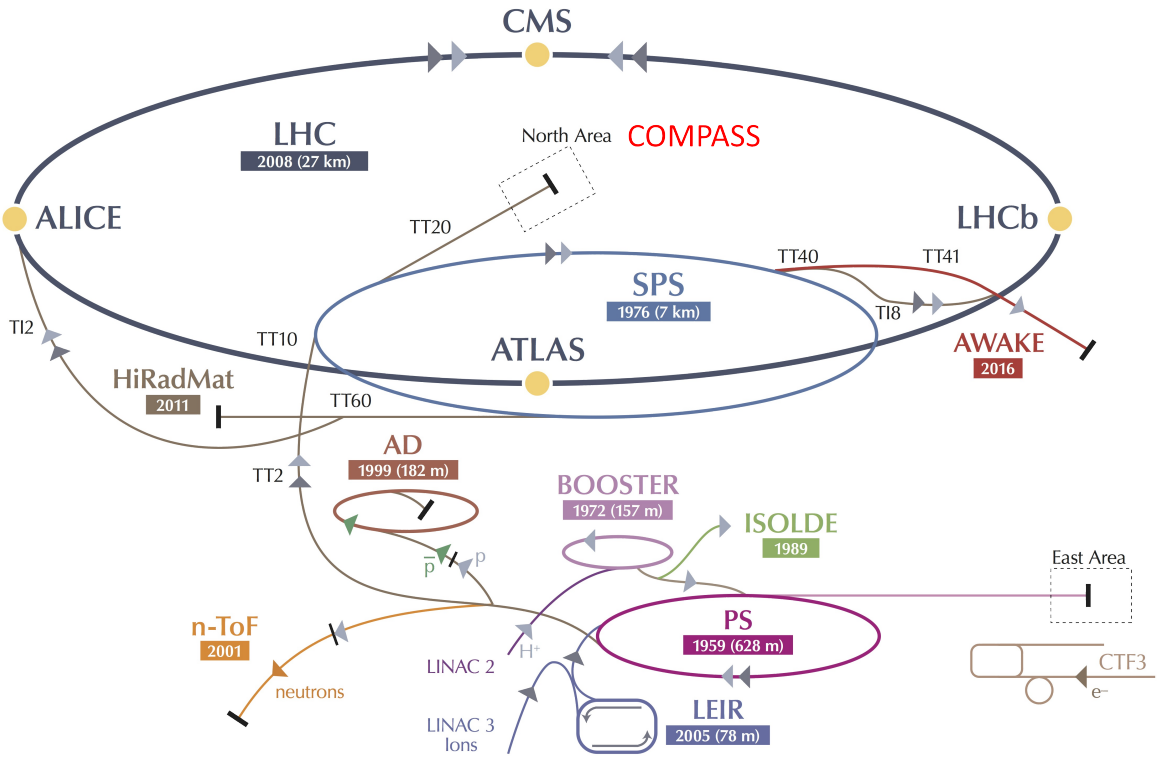


Figure 1.1: CERN laboratory accelerators layout [1]

The experiment is dedicated to examination of the structure of hadron particles and hadron spectroscopy. For this purpose, a high intensity beam composed of muons or hadrons is utilized [2]. More recently, measurement methods which utilize various phenomena, such as deeply-virtual Compton scattering (DVCS), hard exclusive meson production (HEMP), semi-inclusive deep-inelastic scattering (SIDIS), polarized Drell-Yan process or Primakoff reaction are being used.

COMPASSf was established in 1997 through receiving an approval from CERN. The installation of the equipment was conducted between years 1999 and 2000, and the initial technical run was performed in 2001 [2].

The subsequent operation of the experiment may be divided into two periods, called COMPASS I and COMPASS II. The first period lasted from 2002 to 2011, while COMPASS II is operational from 2012 to this day. COMPASS II is expected to end in the year 2018, however, during the following LS2 period, the facilities are expected to be subjected to a lifetime extension maintenance, and the experimental apparatus will be operated under another name.

The measuring apparatus of COMPASS experiment consists of two main stages, where one focuses on particles with wider scattering angle after collision with the experiment's target, while the other concentrates on particles with smaller angles. Each of these stages consists of various types of detectors, such as electro-magnetic calorimeters (ECAL), hadronic calorimeters (HCAL), ring-imaging Cherenkov detector (RICH), micro mesh gas detectors (MicroMegas), gaseous electron multipliers (GEM), multi-wire proportional chambers (MWPC), scintillating fiber detectors (SciFi) or straw trackers. An illustration of the experimental setup is displayed in figure 1.2. The approximate length of the depicted apparatus is 60 meters.

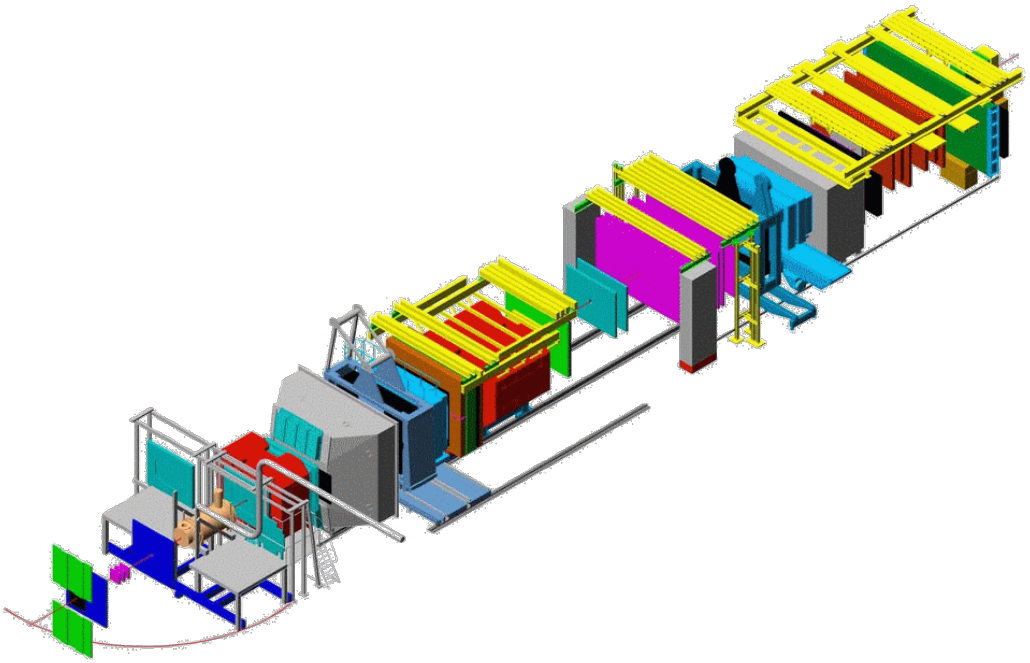


Figure 1.2: COMPASS experimental setup [2]

Chapter 2

COMPASS DAQ

The COMPASS DAQ is a system responsible for reading signals from individual detectors included in the experimental setup and to transform these signals into a form suitable for storage on hard disk drives or other magnetic storage devices.

The current COMPASS DAQ system has been in use since year 2014 and has replaced a DAQ system which has been employed since the beginning of measurements on the COMPASS experiment.

In high energy physics experiments, data taking is usually separated into measurements of individual physics events. These events may represent a collision of a subatomic particle with another particle, or, as is the case on COMPASS, with a fixed target. The measurements of these events focus on the characteristic of such collisions and their resulting effects.

Such events however usually occur at very high rates, and many of them are caused by undesired external influences, such as cosmic radiation. As a result it is technologically challenging to provide hardware capable of reading, and especially storing such large amounts of digital data. This aspect of data taking is usually circumvented by so called trigger systems. Trigger system evaluates each measured event based on a set of predefined criteria, and asserts whether the given event demonstrates interesting properties. Subsequently, only events accepted by the trigger system are stored for further processing. COMPASS experiment's DAQ follows this design pattern, and employs several triggers to reduce the amount of events.

The first step in processing of the signals from detectors is the conversion of these analog data into digital form. This task is performed in the frontend cards. On COMPASS there are approximately 300 000 output channels leading from the detectors to the DAQ.

Next, the digitalized channels are routed through three layers of multiplexers. The first layer consists of HGeSiCa, Catch and Gandalf modules, the second consists of Slink multiplexers and TIGER VXS data concentrators, and the third of FPGA-based multiplexers [3].

After passing through the multiplexer layers, the data reaches an FPGA based switch, which performs the reconstruction of events by combining the data which originated from different channels.

Finally, the data are read by readout computers from the FPGA switch and stored on hard

drives. The hard drives however acts only as a buffer, and every few hours, the data are moved from the disks to CASTOR, CERN central storage solution, which is described in [4].

The whole structure of COMPASS DAQ is depicted in figure 2.1.

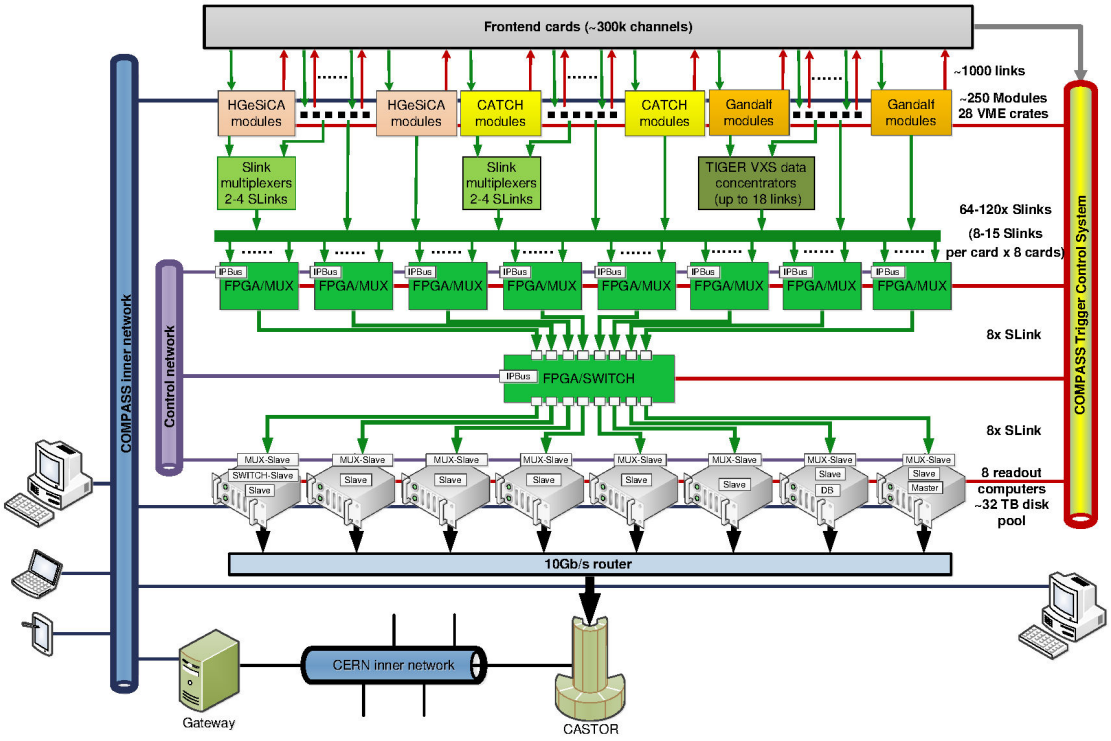


Figure 2.1: COMPASS DAQ structure [3]

Chapter 3

Incentive for Development

This chapter focuses on the underlying reasons that motivated the development of the software described in chapter 5. The contents of this chapter are largely based on research and experience gained during work on [5] and [6]. Systems, that have direct or indirect influence on the design of COMPASS API are examined in order to identify requirements placed on the newly developed software. These requirements are then summarized in the final part of this chapter.

3.1 Background Information

COMPASS experiment's data are primarily stored using two distinct solutions—*MySQL* DBMS and network-accessible directories. The COMPASS API software is intended to improve and simplify the workflow when accessing the data stored in these types of data storage. Therefore, it is necessary to outline key aspects of these computer systems before attempting to design the COMPASS API.

3.1.1 MySQL Database

The databases which are used on COMPASS experiment run on the *MySQL* database management system, and are deployed on servers operated by COMPASS. The servers are organized in a layout with one master server and two slave servers that serve as a backup, as depicted in figure 3.1. More detailed description of the structure of the database servers is provided in [6].

The database servers store several databases. These may vary in implementation details, including the storage engines that are used to store individual tables. Description of the architecture of the databases is part of [7]. Recent analysis of the logbook database, as well as proposition of changes to its structure, which are based on a study of typical use cases of this part of COMPASS's database storage, are detailed in [6].

The basic implementation details of the database servers, such as servers' hardware and software equipment, and even low-level implementation details of the databases like storage

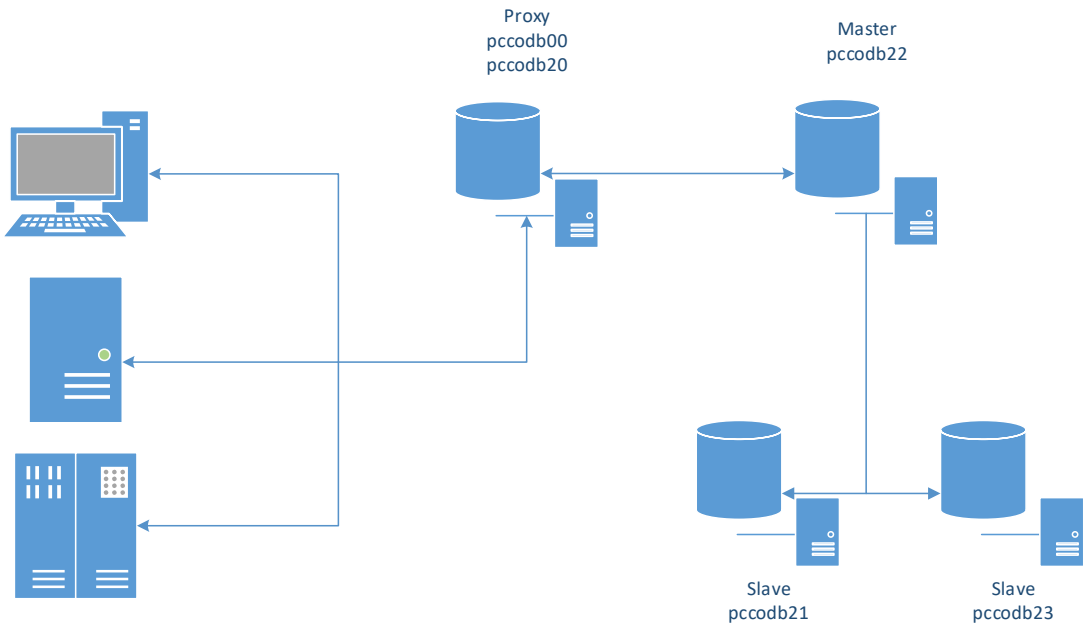


Figure 3.1: COMPASS database servers layout [6]

engines that were used, are well hidden from applications that intend to use the database. Nevertheless, applications still need to deal with the relational model of the implementation. Structures of individual tables and connections between them have to be considered when developing an application that uses the data stored in the database, and database access routines have to be tailored to this specific structure.

3.1.2 Network-Accessible Directories

Network-accessible directories are used on the COMPASS experiment primarily to store binary data, or data which are otherwise unsuitable to be stored in a database management system.

Network accessible directories may be either stored on servers operated by compass, or on servers maintained by CERN and provided to individual experiments.

Individual files are usually recorded in one of the databases described in section 3.1.1. Usually, data necessary to locate the file, such as hostname, path in the file host's file system and file name are stored.

Unfortunately, it was uncovered during analysis performed in [6], that in many use cases no synchronization between the contents of the database and the contents of described directories is performed. The records in the database therefore may not reflect the actual state of the directories.

3.1.3 Summary

Based on properties indicated in sections 3.1.1 and 3.1.2, it is possible to summarize the principal problems tied to manipulation with stored data in the COMPASS experiment's computer environment.

This summary may be divided into three main categories, which are listed below.

1. Manual construction of *SQL* queries is required when working with database, which has following consequences:
   (a) Increased development time of applications,
   (b) More demanding maintenance of existing applications,
   (c) Applications directly depend on structure of the COMPASS databases, therefore any change must be reflected in both the source codes of relevant applications and the database.
2. Storage of files is not performed by a centralized logic, which has similar implications as specified in item 1.
3. In general, no synchronization is performed between file-related metadata that are stored in the COMPASS database and the actual files, which results in the risk of referential integrity corruption.

3.2 Solution Outline

Based on characteristics described in section 3.1.3, it is possible to outline basic properties of the proposed COMPASS API, which aims to correct these issues.

The COMPASS API shall have a form of an interface, which would be usable from any point in the COMPASS network, and preferably would also offer a procedure to utilize it from outside of COMPASS network.

Since the types of data-manipulating operations and their respective implementations are likely to change during the product lifetime of the COMPASS API, the interface shall provide a method of defining request types for accessing data without the need to make changes to the core implementation of the interface. The interface's design shall hide as much logic as possible from these additional modules, so that during their implementation it is not required to deal with unnecessary details, such as network communication. Instead, the implementation of plugins shall have to deal only with processing of the requests and with construction of a suitable response.

3.3 Analysis of Requirements

The general design characteristics of the proposed COMPASS API that were described in section 3.2 are transformed in this section into a more thorough set of requirements, which

the COMPASS API shall meet. All requirements are based either on these characteristics, or were formulated to improve the usability and functionality of an application, which would be implemented on the basis of these requirements.

The requirements are split into functional requirements, which are discussed in section 3.3.1, and non-functional requirements, which are discussed in section 3.3.2.

Items in the requirement list may be either labeled as REQ-##, which denote an actual requirement that is expected to be fulfilled by the final version of the COMPASS API, or labeled as INFO-##, which may contain definitions of terms used in other items, accompanying notes or other information, which is not directly a requirement on its own.

3.3.1 Functional Requirements

These requirements specify what functions shall the COMPASS API offer.

INFO-01 Client application is such computer application that uses COMPASS API and utilizes the functions it provides.

INFO-02 COMPASS network is an ethernet-based network, which connects computers used on and maintained by the COMPASS experiment.

REQ-03 COMPASS API shall allow to be utilized by client applications, which are described in requirement information INFO-01, and are being executed on a computer that is located within the COMPASS network, which is described in requirement information INFO-02.

REQ-04 (optional) COMPASS API shall provide a method that allows to utilize it by client applications, which are described in requirement information INFO-01, and are being executed on a computer that is located outside of the COMPASS network, which is described in requirement information INFO-02, by using a connection over a network that is based on the *TCP/IP* protocol stack.

INFO-05 Accessible database is an *SQL*-based database management server which is accessible from the computer running the current instance of COMPASS API over a network that is based on the *TCP/IP* protocol stack.

REQ-06 COMPASS API shall have a method to open connection to accessible databases, which are described in requirement information INFO-05.

INFO-07 COMPASS database is a database stored in *MySQL* DBMS that is described in [6].

INFO-08 COMPASS database, which is described in requirement information INFO-07, is considered as an accessible database, as described in requirement information INFO-05, if the current instance of COMPASS API runs on a computer located in the COMPASS network, which is described in requirement information INFO-02.

REQ-09 COMPASS API shall obtain the hostname of the computer running the DBMS containing the COMPASS database, which is described in requirement information INFO-07, by loading it from environment variable `DB_SERVER` that shall be available in the operating system in which is the current instance of COMPASS API application running.

REQ-10 COMPASS API shall obtain the username that shall be used for authentication during connection to the DBMS containing the COMPASS database, which is described in requirement information INFO-07, by loading it from environment variable `DB_USER` that shall be available in the operating system in which is the current instance of COMPASS API application running.

REQ-11 COMPASS API shall obtain the password that shall be used for authentication during connection to the DBMS containing the COMPASS database, which is described in requirement information INFO-07, by loading it from environment variable `DB_PASSWD` that shall be available in the operating system in which is the current instance of COMPASS API application running.

INFO-12 COMPASS network folders consist of file system folders whose descriptions are listed in the COMPASS database, which is described in requirement information INFO-07. The folders are described in table `'runlb'.'tb_directories'`. This table shall list all necessary information required to identify and access these folders, including the hostname of the computer hosting a particular folder, and its path within the computer's file system.

REQ-13 COMPASS API shall be able to access and manipulate the COMPASS network folders, which are described in requirement information INFO-12, and the files stored within them, by using *SSH* and *SCP* protocols. Allowed manipulations consist of creating, copying, moving and deleting files stored inside these folders.

INFO-14 Supported data storages are storage solutions described in requirement information INFO-05 and requirement information INFO-12.

REQ-15 COMPASS API shall allow client applications, which are described in requirement information INFO-01, to request creation of data in supported data storages, which are described in requirement information INFO-14.

REQ-16 COMPASS API shall allow client applications, which are described in requirement information INFO-01, to request reading of data stored on supported data storages, which are described in requirement information INFO-14.

REQ-17 COMPASS API shall allow client applications, which are described in requirement information INFO-01, to request update of data stored on supported data storages, which are described in requirement information INFO-14.

REQ-18 COMPASS API shall allow client applications, which are described in requirement information INFO-01, to request deletion of data stored on supported data storages, which are described in requirement information INFO-14.

REQ-19 COMPASS API shall respond to all requests, which are described in requirements REQ-15 to REQ-18, with a response that contains information about the result of processing of the request.

REQ-20 COMPASS API shall respond to successfully executed read requests described in requirement REQ-16 with a response, which contains the requested data in addition to information described in requirement REQ-19.

INFO-21 Any request made to COMPASS API, which belongs into one of the categories described in requirements REQ-15 to REQ-18, is referred to as accepted request under the condition that COMPASS API application has a definition of such request at it's disposal and will attempt to process it as a result.

REQ-22 COMPASS API shall have modular structure that allows to define types of accepted requests, which are described in requirement information INFO-21, as well as the processing logic applied to such requests, by providing supplementary program modules.

REQ-23 All definitions of accepted requests, which are described in requirement information INFO-21, as well as the processing logic applied to such requests, shall be defined exclusively in supplementary program modules described in requirement REQ-22.

REQ-24 COMPASS API shall load supplementary program modules described in requirement REQ-22 in a manner that does not require to recompile or restart the application.

REQ-25 COMPASS API shall provide an internal interface that shall allow to execute database queries in all databases to which COMPASS API currently has an open connection.

REQ-26 The interface that is described in requirement REQ-25 shall be exposed to modules that are described in requirement REQ-22.

REQ-27 COMPASS API shall provide an internal interface that shall allow to execute file operations inside COMPASS network folders, which are described in requirement information INFO-12, to an extent described in requirement REQ-13.

REQ-28 The interface that is described in requirement REQ-27 shall be exposed to modules that are described in requirement REQ-22.

REQ-29 COMPASS API shall provide an interface that shall allow to control selected functionality of COMPASS API at runtime.

REQ-30 COMPASS API's control interface described in requirement REQ-29 shall allow to trigger a restart of the application.

REQ-31 COMPASS API's control interface described in requirement REQ-29 shall allow to load additional program modules, which are described in requirement REQ-22.

REQ-32 COMPASS API's control interface described in requirement REQ-29 shall allow to obtain a list of available types of accepted requests described in requirement information INFO-21.

3.3.2 Non-Functional Requirements

REQ-33 COMPASS API shall run on CERN CentOS 7 operating system.

REQ-34 (optional) COMPASS API shall be implemented with the use of technologies already deployed on the COMPASS experiment.

REQ-35 COMPASS API shall run as a background process.

REQ-36 COMPASS API shall use a communications protocol that is possible to implement and use in *C++* programming language.

REQ-37 COMPASS API shall use a communications protocol that is possible to implement and use in *PHP* programming language.

REQ-38 COMPASS API shall use a communications protocol that is possible to implement and use in *Python* programming language.

REQ-39 COMPASS API shall use a communications protocol that is possible to implement and use in *JavaScript* programming language.

REQ-40 COMPASS API shall use a communications protocol that is possible to implement and use in *Bash* programming language.

REQ-41 COMPASS API shall use *SSH* and *SCP* protocols, whose use cases are described in requirement REQ-13, under the assumption that the public key of the target machine is available to the implementation of *SCP* protocol on the computer running the current instance of COMPASS API application.

3.4 Preliminary Design

With the software requirements formulated, it is necessary to devise a preliminary design of the COMPASS API in order to select suitable technologies for implementation. Key technologies that were used during implementation will only be named in this section, comprehensive description is provided in chapter 4.

Given requirements REQ-03 and REQ-04, it is clear that COMPASS API must be a software accessible by network connection. COMPASS API shall therefore be implemented in a form of a network server. This server would accept network requests, process them, and return response messages to the client application that sent the request.

Additionally, COMPASS API is required to be always accessible from computers located in the COMPASS network, as is stated in requirement REQ-03. Since many computers in the COMPASS network are not connected to the internet, the simplest procedure to fulfill this requirement is to operate the COMPASS API from one of the computers located in the COMPASS network.

COMPASS API must also define the communications protocol that will be used for communication between client applications and the server. Given it's availability and universality,

the *TCP/IP* protocol stack is the obvious choice. This protocol fulfills requirements REQ-36 to REQ-40, as it is available in all these programming languages, and many other, not explicitly listed languages are able to use this protocol as well.

The communication also requires unified format of the messages used for communications. If we consider requirements REQ-15 to REQ-18, we may identify that the system actually supports all of CRUD (create, read, update, delete) operations. It is therefore possible to map them to operations in an existing protocol, that also supports all of CRUD operations. If using an existing protocol, COMPASS API may take advantage of already developed libraries that implement the logic necessary to work with the technology, which will reduce required development time when compared to developing a proprietary data format. The *HTTP* protocol, used according to the *REST* architectural style, was selected for this purpose. *REST* architectural style is described in section 4.3.

When selecting the technology to implement the COMPASS API server itself, requirement REQ-34 shall be considered. This optional requirement was placed into the requirement list in an effort to reduce the number of technologies used for implementation of the software used on the COMPASS experiment. The aim of this endeavor is the simplification of software maintenance, as it results in easier integration of individual programs. Given this constraint, the selection is reduced primarily to *C++* language with the possibility to utilize the *Qt* framework, a combination that is used to implement most prominently the COMPASS DAQ software [8], but also several other tools, described for example in [9], [10] or [11]. Other choices consist of one of the languages used for implementation of various web applications used on compass—*PHP*, *Python* and *JavaScript*. Such applications are described for example in [12], or [13]. The *C++* programming language, which is described in section 4.1, was selected, especially due to performance reasons. Since the already mentioned *Qt* framework, which is described in section 4.2, provides software libraries that directly cover several of the requirements, this technology was selected to be used to implement the server as well.

Neither *C++* and its standard library, nor the *Qt* framework provide an implementation of an *HTTP* server. *Qt* contains a *TCP* server class in its *Qt Network* module, using it to implement a custom *HTTP* server would however demand a development of the entire *HTTP* layer of network communication. Several additional libraries that would supply this functionality were therefore examined. The selected library, as well as the alternatives that were considered, are described in section 4.4.

The resulting implementation is described in chapter 5.

Chapter 4

Used Technologies

This chapter describes principal technologies that were used to develop the COMPASS API. The technologies described consist of programming languages, programming libraries, third-party software, design paradigms and other tools used during development.

Each of the sections focuses on particular technology, and outlines its basic specification and characteristics, as well as describes selected features of the technology. Additionally, an analysis dealing with reasons for using the technology is included. Furthermore, the name and version of used implementation of given technology is stated, where applicable.

4.1 C++

C++ is one of the most widespread programming languages in the world. At the time of writing it is the third most used language according to the TIOBE programming language popularity index [14]. The language is defined by an International Organization for Standardization (ISO) standard ISO/IEC 14882 [15]. The last revision of the standard was released in December 2017.

The language was originally conceived as an extension of the C programming language. As such, it shares most of the C language's syntax, and most C language constructs may be used as a valid code in programs written in *C++*. *C++* however offers many enhancements of functionality over standard C.

C++ implementations that conform to the ISO standard use compiler to produce a machine code from the *C++* source code. However, non-standard implementations of the programming language that use interpreter to process the source code at runtime exist.

4.1.1 Type System

C++ aims to limit the programmer in as few ways as possible, even at the expense of allowing potentially dangerous operations. This approach is reflected in the languages type system as well.

C++ may be considered to be primarily manifest-typed, where all variables are required to have their types explicitly declared, but since the later standard versions, a support for inferred typing is present as well.

Similarly, *C++* may be considered statically typed, but a limited number of features for dynamic type checking is available.

Unlike the C language, *C++* allows no implicit violation of its type system. However, explicit violations requested by the programmer are possible.

As a result of these properties, *C++* is usually characterized as a "strongly typed, weakly checked language" [16].

4.1.2 Programming Paradigms Support

C++ is designed as a multi-paradigm language, and this principle remains as one of the basic rules of the standardization committee when proposing any changes to the language standard. The intent behind this design principle is to not limit the programmer, and to allow them to choose a programming paradigm depending on a specific task, and even combine them. Programming techniques, that are most commonly associated with *C++* are procedural programming, object-oriented programming and generic programming [16].

Object-Oriented Programming

One of the major advantages of *C++* over standard C, and also one of the underlying reasons to implement an upgraded version of the C language, is the support for object-oriented programming paradigm. The language provides all common features of object-oriented programming, such as encapsulation, inheritance and polymorphism.

C++ object's members always have an associated access specifier, which governs the access rules for usage of the given member. Whenever a given member is used in the program, the context is checked against these access rules, and if a violation of such rules is detected, the program won't compile. The three access levels in *C++* are `public`, `protected` and `private`. `public` access level grants access from any part of program. `protected` access level allows access only from the members of the class, and from members of any derived class. `private` is the most restrictive access level, and grants access only to members of the same class.

There are three basic types of objects available in *C++*. The first two types are the `class` object and the `struct` object. These two differ only in the default value of the access specifier, where members of `class` object has default access specifier value `private`, and `struct` on the other hand defaults to `public`. The main reason why `struct` is provided is backwards compatibility with the C language. The third type of object that is available in *C++* is the `union` object. Union's data members share the same memory, and `union` is therefore only as large as its biggest data member. As a result, only the most recently set data member of a union is to be considered valid.

Inheritance mechanism is available in *C++* as well. Apart from regular inheritance, *C++* also supports multiple inheritance, i.e. inheritance of multiple base classes at once. Solution

to the so-called diamond problem[1] is also available through a facility called virtual inheritance. Base classes may also be abstract, which means that some of the methods do not have an implementation provided, and it is up to the inheriting class to supply the implementation. Abstract base classes may be utilized to create interfaces, i.e. classes where none of its members provide an implementation.

Generic Programming

Generic programming may be characterized as programming style, where the code is written for generalized input data types. These generalized inputs may be viewed as parameters of the code, which are specified during distinct use case.

The support for the generic programming paradigm in *C++* is implemented primarily through the `template` construct. Templates are parametrized entities, which correspond to other, non-templated *C++* language constructs, for example functions or classes. However, by using the templated version of such constructs, the programmer may define them with a signature that is dependent on the template's arguments. Templates which have the arguments provided to them are called template specializations. The actual code for each specialization used in the program is then automatically generated by the compiler. This approach has the main advantage of allowing to write effective and faster code with fewer lines, which stems from the very fact that each specialization is generated at compile time. This however also results in the biggest disadvantage. Since a similar machine code is generated for each used template specialization, the resulting binary may increase in size considerably.

4.1.3 Libraries

The core functionality of *C++* is extensible through additional libraries, which may be included into any *C++* project. Inclusion of libraries is handled by the *C++* language's preprocessor. Any library must provide it's header file so that it's interface is known to the compiler (and the programmer), but it's implementation itself may be distributed already compiled, in a form of a shared library.

The most prominent *C++* library is the standard library, whose properties are defined as part of the ISO/IEC 14882 standard. The standard describes not only the API of the library, but also defines performance constraints that any standard-compliant implementation must uphold. The standard library consists of many functions and classes, that implement common programming constructs and algorithms, or provide platform-independent abstraction of system resources.

C++ contains implementation of the standard library of the C language in the C99 standard in two versions—one legacy version, and another version where all parts of the library are placed in the standard namespace [15].

Next, the standard library provides numerous utility constructs. These consist for example of definition of `string` data type, regular expressions, data types like `pair` or `tuple`, smart

[1] Diamond problem is a situation, where a class inherits from two base classes, which both in turn inherit from the same base class themselves, thus creating diamond-shaped class hierarchy

pointers and numerous utility functions.

Abstraction of system resources is as previously stated provided as well. Programmers may utilize input and output streams, access to system clock, or concurrency library, which implements substantial number of common parallel programming techniques.

A central part of the standard library is however the standard template library [17]. It primarily holds definition of generic data containers for storing data collections. Several data containers with differing internal structure and data organization are provided. Consequently, each data container has various advantages and drawbacks, which primarily influence computational performance. Additionally, the standard template library includes various other constructs for accessing the data in the containers, and for processing the data with predefined or user-defined algorithms.

Apart from the standard library, vast number of third-party libraries are available for *C++*. These libraries focus on many different areas of programming, ranging from generic libraries, through networking, numerical calculations and creation of graphical user interfaces to machine learning. An example of an established *C++* library is the Boost library. This library aims to complement the standard library and extend it in a compatible way, and many libraries which originally were part of boost collection were eventually standardized in *C++*11, *C++*14 and *C++*17 standards.

4.1.4 Usage of the Technology in Master Thesis

The selection process of a programming language used for implementation of COMPASS API was directed mainly by requirement REQ-34. This requirement is motivated by an effort to minimize the number of technologies used during the implementation of software equipment of the COMPASS experiment. As a result there were two options available for the primary language which would be used to develop the main part of the COMPASS API—*PHP* and *C++*.

PHP is used on COMPASS as a primary language for implementation of server-side logic of websites. It could be used to create a *REST* server, which would run on an *Apache* Web server.

Since *C++* in its standard-compliant form is a compiled programming language, it is available on every COMPASS computer. *C++* applications may implement network communication, and the technology is therefore also suitable for implementation of *REST* server.

While *PHP* offers several advantages, the *C++* language was selected. This was done primarily because it offers potential performance benefits over the *PHP* language. These benefits stem from the fact that *PHP* is an interpreted language, and *C++* in its recent standards also offers more powerful facilities for parallel programming.

Furthermore *C++* language abilities may be easily enhanced by using the *Qt* framework, which also meets the requirement REQ-34. *Qt* framework is described in section 4.2.

Finally the author of this master thesis is proficient with both *C++* and *Qt*. While this is a subjective reason, using familiar technologies increase the quality of the resulting code and decreases the development time.

4.1.5 Used Implementation

COMPASS API is intended to run on an x86 64-bit Linux distribution. The target ABI is therefore x86 64-bit generic Linux extensible and linkable format. During development, the GCC compiler version 7.2.1 was used.

COMPASS experiment computers are at present usually running on Scientific Linux CERN 6, which is equipped with GCC version 4.4.7. However, it is expected that by the time of the deployment process of COMPASS API, COMPASS computers will be upgraded to CERN CentOS 7, as this upgrade is planned for CERN LS2. The CERN CentOS operating systems comes with GCC 4.8.5, which still does not support all necessary *C++* language features, however, GCC 7.2.1 is available on CERN CentOS 7 through package *devtoolset-7*.

4.2 Qt

Qt is a cross-platform framework that is primarily oriented at the development of computer applications. *Qt* is primarily aimed at development using *C++* programming language, and comes with numerous *C++* libraries. However, *Qt* also provides additional tools to support application development. Especially tools focusing on development of graphical user interfaces are noteworthy, and *Qt* is able to either automatically generate *C++* GUIs from XML files, or to display GUIs that are interpreted at runtime from sources written in proprietary language called *QML*.

4.2.1 Build System

Code written in *Qt* is usually not compliant with standard *C++*, and sources of *Qt* projects require to be processed by several preprocessors. These tools are collected in a tool called *qmake*. This tool deals with all non-*C++* and non-standard *C++* files, generates *C++* code adhering to *C++* standard from these files where necessary, and generates a makefile for the *make* build automation tool. Project processed in this manner may be subsequently compiled by any supported *C++* compiler. Most of the common *C++* compilers in reasonably recent versions are usable for compilation of *Qt* projects processed by *qmake*.

Recently, a new build system called *Qbs* (pronounced "Cubes") was released by *Qt*'s development team. It is planned for *Qbs* to eventually replace *qmake*, whose entire concept has several drawbacks. Unlike *qmake*, or *qmake*'s alternatives like *CMake* or *meson*, *Qbs* is not merely a makefile generator, whose output is then processed by another program, but it directly calls compilers and other necessary tools.

4.2.2 Modules

Qt framework is divided into multiple blocks called modules, each of which focuses on specific area of programming. These modules may be loaded individually, and therefore only required parts of the framework may be used during the build process.

Modules present in the basic, open-source version of *Qt* are divided into two categories. The first category is called *Qt Essentials*, and consists of modules that are available on all platforms supported by *Qt*. Second category is called *Qt Add-Ons* and contains more specialized modules, often intended for more distinctive use-cases or implementing platform-specific functions. Additionally, commercial versions of *Qt* may provide further modules, which for example focus on specific type of industry [18].

This section describes the modules that are used by COMPASS API. All of the described modules belong to the *Qt Essentials* category.

Qt Core

Qt Core is the basic module of the framework, and all other modules depend on it to some degree. It contains the implementation of the core mechanisms of the framework, as well as many generic classes.

The cornerstone of *Qt*'s functionality is the built-in meta-object system [18]. This apparatus provides several language features currently not present in the *C++* standard, namely more advanced type introspection and reflection. This is achieved by using a preprocessor called meta-object compiler (*moc*), which works together with several macros and classes built into the framework.

Framework features that were made possible by the meta-object system are primarily represented by *Qt*'s signals and slots facility. Signals and slots is a mechanism for event-based communication between objects, where signals represent an event, and slots implement action to be taken upon that event. The programmer may connect arbitrary number of slots to any signal in the code, and whenever that particular signal is emitted, all of the registered slots are executed. Many signals for various events are predefined in the built-in *Qt* classes, and the programmers may define their own signals as well.

Owing to the meta-object system, *Qt* also presents the possibility to utilize the internationalization and localization functionality, which enables to quickly adapt an application written using *Qt* to different regional requirements and quickly swap language versions of the application.

The meta-object system additionally provides advanced type introspection, which would be otherwise unavailable by exclusively using *C++* language features.

Moreover, many features not directly related to meta-object system are available in *Qt Core* module. Classes for multi threading support, Container system similar to standard template library or state machine framework using XML files to automatically generate *C++* state machines are provided. As part of this module, *Qt* also includes facilities for data abstraction in a form of a model/view framework, which separates the data structure from the actual presentation layer, thus allowing to easily represent data in different ways.

One of the features in *Qt Core* module, that should also be mentioned is the *Qt plugins* component described in [19], which is crucial for implementation of some of the required features of COMPASS API. *Qt plugins* allow to compile *C++* dynamic library, that contains a class that is then used to enhance the functionality of the main program during runtime. The

class must implement an interface, that is known to both the dynamic library (the plugin) and the program loading the library. For description of how *Qt plugins* are used in COMPASS API, see section 4.4.2.

Qt Network

Qt Network is module that offers provisions for network communication. Classes for low-level communication and high-level communication are given at programmers disposal. Furthermore, an API that enables control over available network interfaces is available.

Basic network communication is possible by utilizing either *TCP* protocol or *UDP* protocol. *Qt Network* implements classes representing *TCP* sockets and *TCP* servers for use of the *TCP* protocol, and a class that represents *UDP* socket for utilizing *UDP* protocol. Additionally, *SSL* sockets are provided for encrypted communication.

High-level network operations may be achieved by using provided API designed around a request/response principle. Through this API, applications may either synchronously or asynchronously send network requests and process subsequent responses. Currently, *HTTP*, *FTP* and local file *URL*s are supported.

Qt SQL

Qt SQL is a *Qt* module enabling communication and interaction with common *SQL*-based database systems.

Members of this module may be divided into three abstraction layers—driver layer, low-level API layer and high-level API layer.

The driver layer contains driver plugins for individual database systems. Drivers for common DBMS are already included with the framework. If required, the programmer may also implement additional driver plugins. This mechanism uses the *Qt plugins* component implemented in the *Qt Core* module, which is also used during implementation of COMPASS API. It requires the programmer developing new driver for *Qt SQL* module to implement certain interface at compile the resulting class as a dynamic library.

The low-level API layer provides access to the database. The programmer may open database connections, execute *SQL* queries or perform other database tasks supported by the driver.

The high-level API uses the model/view facilities from the *Qt Core* module, and implements data abstraction layer above the database. By using the classes from the high-level API, it is possible to read and manipulate the data in the database without the need to execute individual *SQL* queries.

Qt Test

Qt Test is a unit test framework integrated into the *Qt* environment. It aims to allow programmers to quickly write unit tests for their classes written using *Qt* framework, and also

to write benchmarks for their code.

Apart from common facilities aimed at unit testing of developed *C++* classes, *Qt Test* also provides advanced tools for testing of specific *Qt* features. Mechanisms for GUI testing are available, and allow to simulate GUI events normally performed by the user, thus automating testing of user interfaces. In addition, it is possible to use signal introspection to monitor emitting of *Qt* framework's signals.

4.2.3 Supported Integrated Development Environments

Qt framework consists not only of a *C++* library, but also from a collection of various tools aiming to support development process, which must be executed at specific time. As a result, using this technology may become overly complicated. Development process may however be simplified by using an integrated development environment, which handles the underlying tools and provides additional utilities to simplify the process of creating computer programs that use *Qt*.

Qt framework comes with the *Qt Creator* Integrated Development Environment. This IDE was developed by the same organization which is also responsible for the development of the framework, and therefore it seamlessly incorporates itself with other parts of the framework. It is also immediately updated to use any new components of the *Qt* framework when necessary.

Because it was intended for development of *Qt* applications, the IDE primarily offers support for *C++* and *QML* languages. In addition it provides editor for WYSIWYG creation of graphical user interfaces, either based on *QML* or on *Qt Widgets*, and for creation of scxml-based state machines.

Moreover, the ability to integrate with numerous external tools is provided. Examples of such tools may be various version control systems, such as *Git* or *SVN*, static code analyzers, tools for automatic code formatting, but also numerous other tools.

In addition to *Qt Creator*, *Qt Project* organization also offers a plugin for Microsoft Visual Studio IDE. Furthermore, several build systems offer the possibility to build *Qt*-based projects without official support from the *Qt Project*, such as the Meson build system.

4.2.4 Usage of the Technology in Master Thesis

Since the *C++* programming language was selected for implementation of COMPASS API, as is described in section 4.1.4, it became possible to utilize some of the libraries and frameworks available for this programming language.

To abide by requirement REQ-34, *Qt* framework was one of the candidates. This *C++* framework is already employed by COMPASS DAQ, and number of other computer programs on COMPASS as well. *Qt* framework is viewed primarily as a tool for creating applications with GUI, but it has equally evolved facilities for creating modern, event-driven command line applications.

Qt also implements several mechanism that directly deal with problems outlined by requirements stated in section 3.3.1 and section 3.3.2.

Requirement REQ-22 may be easily fulfilled through the *Qt plugins* component and its `QPluginLoader` class, which are described in section 4.2.2. This class allows to load shared libraries during runtime, and is thus well suited for the task drafted by requirement REQ-22.

Qt also comes with the *Qt SQL* module, which provides an API to access *SQL*-based database servers, including *MySQL*, and therefore covers the problematic implied by requirement REQ-06.

4.2.5 Used Implementation

COMPASS API was developed using *Qt* version 5.6.2. COMPASS API requires *Qt Core*, *Qt Network* and *Qt SQL* modules, and for testing purposes, the *Qt Test* module was also utilized.

Similarly to the situation described in section 4.1.5, COMPASS computers usually have *Qt* version 5.5.1 installed on them. However, with an OS upgrade planned for LS2, the required version of *Qt* or newer is expected to be available in the targeted environment.

The project was managed using the *qmake* build system, and the build process is defined in a *qmake*'s *.pro* file as a result.

The development of the source code was conducted in the *Qt Creator* IDE, which was in version 4.1.0.

4.2.6 License Conditions

The *Qt* library is available under several different licenses, from which is the user required to choose. Since the available license choices contain the GPL and LGPL licenses among others, for academic use and other use cases, where the created software is not distributed to third parties, the *Qt* libraries are provided free of charge or any other obligations.

4.3 REST

Representational state transfer, abbreviated as *REST*, is an architectural style for designing software systems. The architectural style was conceived for designing web based applications, or, in more generic terms, distributed hypermedia systems [20].

Application programming interfaces based on the *REST* principle are usually closely tied to the *HTTP* protocol. However, there is no limitation that would hinder the possibility of implementing a *RESTful* API using different transfer protocol. Nevertheless, the design of *HTTP* protocol itself follows the principles of *REST*[2], and as such is an obvious candidate

[2]The main reason for these shared design characteristics is the fact that the author of the *REST* architectural principle, Roy Fielding, is also one of the principal authors of the *HTTP* specification.

for implementing a *RESTful* API.

4.3.1 Constraints of RESTful API

REST itself may be defined by a set of constraints, which any *REST* compliant API should respect in its design. These design constraints shall be perceived as a gradually evolving system, where later constraints usually rely on already defined constraints [20]. This way the architectural style is gradually built from the ground up.

Client-Server Architecture

Splitting a software system into multiple layers offers several advantages. Individual domains of the applications functionality may be effectively separated, which in general simplifies software's maintainability and scalability, as any modification will be isolated to a single software layer (provided the layers were well-designed in the first place).

The client-server based architecture is separated into two basic layers, where the client sends requests to the server side when needed, and is in turn served with responses containing the result of the requested operation. The server on the other hand contains the backend part of the application which handles the manipulation of any data sources, and the processing logic. The communication is directed by the client, even though most of the programming logic unusually resides within the server's domain.

In a context of *RESTful* API-based applications, or in more general terms network-based applications, such separation allows to improve and modify the data loading logic, which is contained within the server part of a system, independently of the user applications that act as the clients. On the other hand, applications may take advantage of the same backend logic and concentrate on the presentation layer.

Statelessness

This constrain implicates the fact, that the server cannot store any information related to a particular client (a state), especially any recordings of previous interactions. In other words, every request from a client contains all required information to perform the given operation, and always yields the same result, provided that conditions external to the server, such as data available to the system, remain identical.

Caching

Caching constraint in the *REST* design may significantly improve the performance and lower the amount of network traffic in systems employing *REST* architecture. This effect is achieved by storing the responses for requests and then reusing them when the same request is sent again instead of conducting its processing once more.

Caching may be done at the servers end, in which case all requests may use the cached response. This method results in lowered amount of requests that need to be processed by the actual server's logic, and therefore less computer resources that need to be allocated at servers disposal.

Different method is to maintain a cache at the client's side of application. When using this strategy, only one particular client may benefit from the cached data. On the other hand, the request does not have to be transmitted by the selected communications protocol to the server, which lowers the amount of used bandwidth in the network.

The trade-off is that the client may not be served with the most up-to-date data. Thus it may not always be viable to implement caching for every request in a *RESTful* system. The general rule is that every response should be either implicitly or explicitly tagged with an information, whether it is allowed to use caching in the particular case [20].

Uniform Interface

Uniformity of interface may be considered as the most distinctive architectural constraint of *REST*, when compared to other network-oriented architectural styles [20].

The constraint implies, that services' interfaces need to follow a generalized principle when defining their access points. Interfaces following a generalized rules of design may be inefficient to certain degree, because interface designed to suit a specific use case will be thinner, and less obstructed by features which are unnecessary for given situation. Generalized interfaces however result in more user-friendly interfaces, since every service is accessed in similar manner.

Uniform interface constraint itself follows four sub-constraints [20], which are explained bellow:

1. **Identification of resources:** A resource is a crucial element of *REST* architectural principle. It represents one piece of information, that is managed by the server side of *RESTful* API, and is available to clients connecting to that server.

   Resource itself is however a function that relates to a set of data, not the data itself. The function's input variable is time, and based on the time parameter, resource returns the most recent data [20].

   Any resource must be posses a unique identifier within the system, so that other components in the system may access it.

2. **Manipulation of Resources Through Representations:** The concept of resources in *REST* principle also serves to decouple the representation of the data returned by the resource from its actual representation that is used to store the set of data tied to a resource. Resource may map to a completely different format during the processing of a request made to it, which in turn allows to completely change the internal storage format without affecting the clients that depend on the data.

   Additionally, when any part of the system has a representation of a particular resource at its disposal, it must posses necessary amount of information to manipulate the resource. In other words, it must be able to interface with the server handling

the resource and trigger a modifying operation on that resource (provided such modification is allowed by access rules). This may equal for example to an id of a row in *SQL* table.

3. **Self-descriptive Messages:** Every resource representation consists of actual data, and accompanying meta-data. The meta-data should most prominently contain information on how to interpret the data. The constraint of "Self-descriptive Messages" may be reinterpreted as a statement saying that all resource representations, including any interpretation of it, must be self-contained, and shall not require additional information. To adhere to this principle, a system component that possesses a resource representation must know how to process it. This information shall thus always be included in the provided meta-data.

4. **Hypermedia as the engine of application state:** Hypermedia as the engine of application state, often abbreviated as HATEOAS, is a constraint stating that client applications that enter a *RESTful* system require only the knowledge of the identifier of the resource required for entry into the system. All other resource identifiers required to communicate with the server further will be included in one of previously obtained resource representations.

   Therefore, if a client information obtains a resource representation, and that resource representation requires other resource representations to work correctly, the information on how to access these additional resources must be contained in the first resource representation.

Layered System

In *REST* server implementation, components which provide individual services may be separated into layers. The key property that must be however preserved is the concealment of this fact to connected clients. To any client application communicating with the server, the method to access arbitrary service must remain the same.

Layered server systems offer several advantages. Additional layer may implement various security measures, such as authentication of users accessing the server, or by acting as a proxy server that hides the actual API server from other parts of a network. Another possible role of extra layers is to perform load balancing between several physical servers. Groups of logically bound services, for example legacy components, may also be moved to a separate layer, in order to prevent unnecessary influence on implementation of current components.

However, systems with several layers usually have higher processing times due to added latencies and more logic needed to be executed in individual layers before routing the request to the correct processing unit. Nevertheless, this effect may be mitigated by caching [20].

Code-on-Demand

RESTful API servers may also serve the clients with actual code which may then be executed on the client's side of a system. This may lower the implementation requirements on client

applications, as parts of the required code may be pre-implemented and then delivered by the server for immediate execution.

This constraint is labeled as optional in the design of *REST* architectural style [20]. The main reason for this optionality is that otherwise, all client applications would be obliged to be able to execute the delivered logic, for example by a common interpreter, thus placing certain technology requirements on all clients. Since the constraint is optional, only certain parts of the entire system may employ this mechanism.

4.3.2 Usage of the Technology in Master Thesis

COMPASS API aims to provide a centralized software system that may be used by other applications to access and modify remote resources. This shall be done in a manner agnostic to the technology used to implement the client application that accesses the resources.

Given these requirements, *REST* design principle offers simple and proven, yet efficient solution to the problematics of how to design an interface usable in such way. *REST* is independent of any programming language, and only requires a common communication protocol, which enables to use applications developed using different technologies with the same API.

4.3.3 Used Implementation

As described at the beginning of section 4.3, *REST* is a software system design principle, and is not tied to any specific technology. However, majority of APIs designed according to *REST* principles use very similar model, where the *HTTP* protocol is used as the transport protocol, and *URI* used for identification of specific resource

In this implementation, The resource that is being accessed is identified by an "endpoint", which is a synonymous term for the path specification in the *URI* used to access the resource. Additionally, *URI*'s query may be used to filter the data returned by the resource.

The type of resource manipulation is determined by the *HTTP* protocol method that was used for the request. The mapping of *HTTP* methods to specific operations is detailed in table table 4.1.

HTTP method	mapped operation
POST	create data
GET	read data
PUT	update data
DELETE	delete data

Table 4.1: Mapping of *HTTP* methods to *REST*-compliant resource manipulations

The *REST* server developed as part of the COMPASS API follows this de facto standard, and uses the described *HTTP*/*URI* combination for resource management.

4.4 QHTTPEngine

The *QHTTPEngine* library provides necessary provisions for running an *HTTP* server in *C++* applications using *Qt*.

The library itself was developed using the *Qt* framework. It is not supplied by the *Qt Project*, the governing body behind *Qt*'s development, but rather by a third party. It is however based on the *Qt Network* module, and fully integrates with official library facilities.

The library is provided as an RPM package in update repositories of the CERN CentOS 7 operating system. However, only an outdated version is provided through this distribution channel. It is nevertheless possible to download the sources from the official GitHub repository and build the binary locally. The library sources are provided as a *CMake* project, which may be opened and built in the *Qt Creator* IDE, which is described in section 4.2.3.

4.4.1 Library Structure

The library is mainly built by deriving from classes contained in *Qt Network* module, which is described in section 4.2.2, and follows the same structure as the *Qt Network* components. The central classes of the library are therefore the `QHttpEngine::Server` class, which is derived from `QTcpServer`, and `QHttpEngine::Socket` class, which is derived from `QIODevice` and owns an instance of `QTcpSocket`. However, both provide additional functions related to the *HTTP* protocol when compared to standard *TCP* communication facilities provided by the *Qt* libraries.

The `QHttpEngine::Server` class contains methods for *SSL* configuration setup, and most importantly methods for linking a root handler. Handlers in *QHTTPEngine* are classes that are used to implement the basic logic of *HTTP* request handling. Several preprogrammed handler classes are provided out of the box, all of which are derived from `QHttpEngine::Handler` class. Additional handlers may be provided by the user of the library by implementing a new class that is derived from `QHttpEngine::Handler`.

`QHttpEngine::Socket` on the other hand may be viewed as an instance of specific connection established between the server and a client. Its enhancements over standard `QTcpSocket` class consist of methods for request parsing, such as parsing of header information, query string parsing or parsing of message contents which are in the *JSON* format, but also provides methods for constructing the responses sent back to the client application.

4.4.2 Usage of the Technology in Master Thesis

One of the key parts of the entire COMPASS API system is the *REST* server, as discussed in section 4.3, which will be used to process the requests for data manipulation.

Several possibilities of how to implement the required functionality were considered:

- **Custom implementation:** *Qt* framework provides an implementation of *TCP* server in the *Qt Network* module. It is therefore possible to build custom implementation of *HTTP* server on top of the `QTcpServer` class. However, this would require additional

work, since detailed study of requirements placed on such server would have to be conducted, and using existing solution has the additional benefit of using technology that has already been proven at least to some degree.

- **GNU libmicrohttpd:** *libmicrohttpd* is a pure C library provided in the GNU collection. As such, it has a trustworthy guarantee of quality. However, since it is a pure C library, integration into any object-oriented *C++* application may be less than ideal from coding style point of view [21].

- **libhttpserver:** The shortcoming of *libmicrohttpd* library being a pure C library could be circumvented by using a library wrapper *libhttpserver*. This software acts as a *C++* programming interface for the *libmicrohttpd* library. However, ideal solution would also provide the possibility to integrate with the application by means that are standard in the *Qt* framework, namely by connecting with *Qt*'s signal/slot facilities. *libhttpserver* is however a standard compliant *C++* library, and such requirement would result in the need to write additional *Qt* wrapper layer [22].

- **libqmhd:** *libqmhd* is an alternative wrapper for *libmicrohttpd* library, which provides an interface that uses *Qt* framework's facilities. However, the library has not been developed for several years, and any bugs or lack of functionality is therefore unlikely to be fixed by its authors [23].

- **QttpServer:** *QttpServer* is a third-party library for the *Qt* framework. It specifically targets the development of *C++*-based *REST* API servers. It promises very fast request processing, which would also perfectly suit the requirements placed on COMPASS API. Unfortunately, it lacks support for *SSL* certificates integration and user authentication [24].

- **QHTTPEngine:** *QHTTPEngine* library is described in detail in section 4.4.1.

After weighing respective advantages and disadvantages of the options listed in this section, the *QHTTPEngine* library was selected. It allows for the most seamless integration into the COMPASS API application, and offers all required features.

4.4.3 Used Implementation

COMPASS API uses the *QHTTPEngine* library in version 1.0.1. However, the library is provided along with the rest of the sources of the COMPASS API, and its setup for local use is integrated into the build process of COMPASS API. For testing purposes, it is therefore not required to have the library installed on local computer.

4.4.4 License Conditions

QHTTPEngine library is available on GitHub on the following URL: `https://github.com/nitroshare/qhttpengine`. The software is distributed under MIT license [25], which permits to use the library free of charge or any other obligations [26].

4.5 JSON

JSON, or *JavaScript Object Notation*, is a text format for storing digitalized data. It was originally developed from *JavaScript* programming language, but it is not directly dependent on it, and is presently used for data exchange in applications which use different programming languages.

One of *JSON*'s main advantages is the fact that it is human-readable, but at the same time it may be easily parsed by computer programs.

JSON format is defined in two contesting standards, which are ECMA-404 [27] and RFC 8259 [28].

4.5.1 Structure of the Language

JSON is primarily composed of two types of structures [29]. The first is the *object* structure, which is an unordered container for *key-value* pairs. The structure of *object* is depicted in figure 4.1.

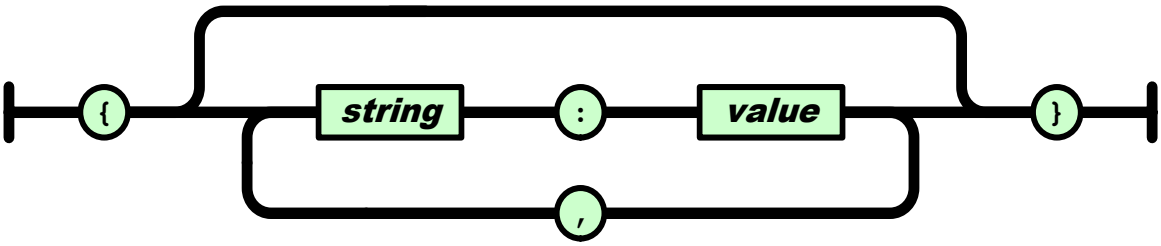


Figure 4.1: Structure of *JSON object* [29]

The second is the *array* structure, which acts as an ordered collection of *values*. The structure of *array* is illustrated in figure 4.2.

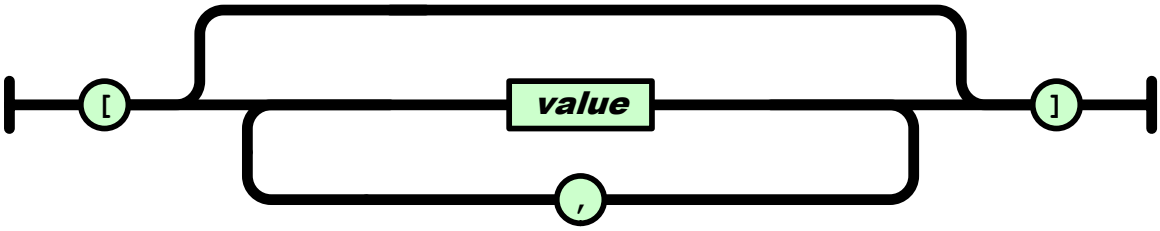


Figure 4.2: Structure of *JSON array* [29]

In the context of *JSON* language, *value* may be one of several entities. It may be either a *string*, a *number*, another *object* or *array*, or one of three possible constants, which are *true*, *false* and *null* [29]. Visual description of *value* is included in figure 4.3.

A *string* in *JSON* is a sequence of zero or more characters which use the Unicode encoding, and are encompassed in double quotes. Special characters are escaped with backslash [29].

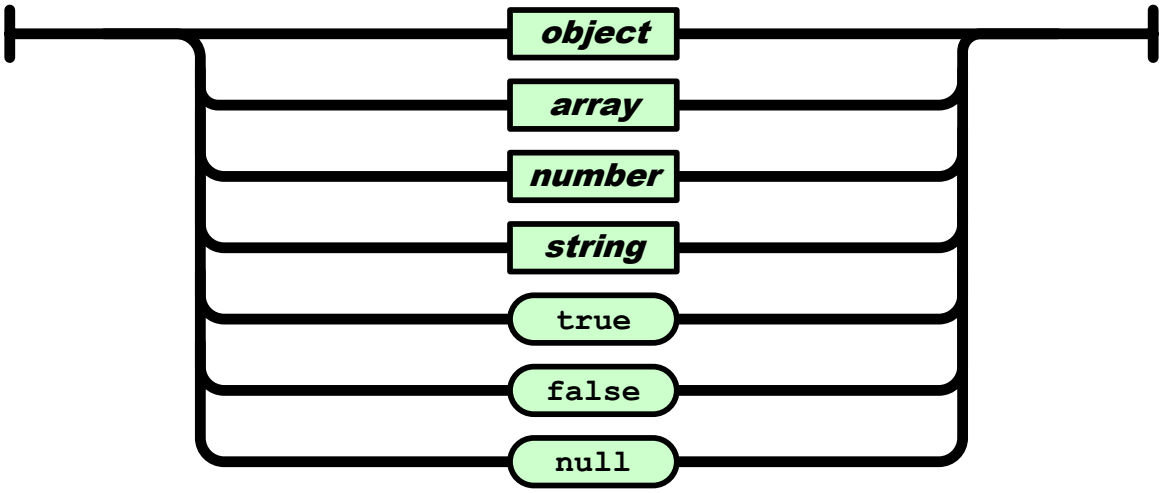


Figure 4.3: Structure of *JSON value* [29]

JSON treats all numbers as equal and the standard does not differentiate between integer numbers and floating point numbers. It supports the scientific notation, but does not allow for octal and hexadecimal representations and non-numbers such as NaN or Inf [29].

4.5.2 Usage of the Technology in Master Thesis

In the COMPASS API, *JSON* is the recommended format for data interchange between the *REST* server and its clients.

While the format of responses is entirely dependent on the choice made by developers of the program modules, which are described in requirement REQ-22, it is recommended to use the *JSON* format for it's properties and to maintain a unified interface. Additionally, COMPASS API *REST* server simplifies reading of *JSON* documents from the network communication socket, as well as subsequently writing them into it. This reduces the effort required to use this format.

Chapter 5

Implementation

This chapter describes the implementation of the COMPASS API. The system is composed of a total of four components, where two of these components are the main concern of this master thesis.

First, the entire system is focused on as a whole in section 5.1, and subsequently the individual components are described separately in more detail in sections 5.2 to 5.5.

The two components, which are of crucial importance to this thesis, are described in sections 5.2 and 5.3. For these two components, their structure is described both verbally and by using UML class diagrams. Additionally, tasks conducted by these component, which are of key importance for its correct operation, or are otherwise noteworthy, are focused on, and the individual steps of the operation are pictured in UML activity diagrams.

5.1 COMPASS API

COMPASS API is designed according to the *REST* architectural style that is described in section 4.3. Therefore, the system must be composed of at least two basic components—a server, to which multiple client applications are connected.

Additionally, other components that are specific to COMPASS API must be considered. Requirements REQ-22 and REQ-23 state that definitions and processing logic implementation for each request must be implemented in independent program modules, which shall be loaded at runtime. The *Qt plugins* mechanism, which is described in section 4.2.2, was used to implement this feature. *Qt plugins* component requires an interface to be defined, which is subsequently available to both the *REST* server and each of the plugins. This interface is as a result an additional component. Moreover, plugin is in the system's design considered as an additional component as well.

COMPASS API therefore consists of four main components, as was previously stated. These are the *REST server*, *plugin interfaces*, *plugin* components and the *client* applications.

The structure of the system is depicted in a component model represented by UML class diagram in figure 5.1. The class diagram shows the four main components of the COMPASS

API system and the relations between these components.

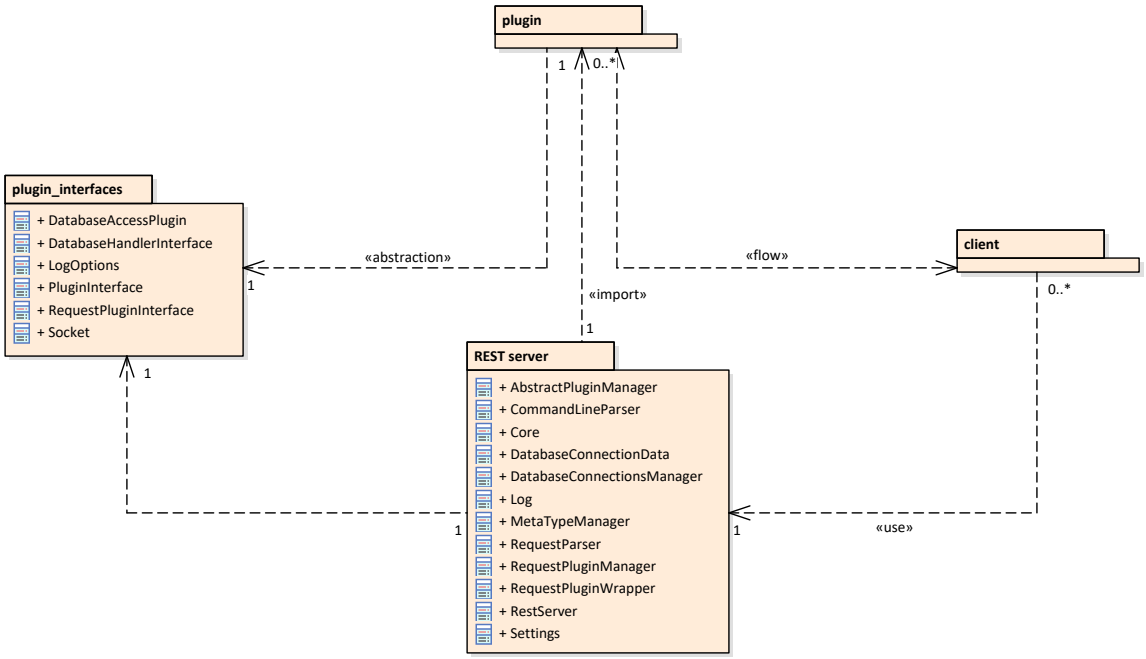


Figure 5.1: UML component diagram of COMPASS API

REST server imports any number of *plugins*. Since these *plugins* must implement certain, predefined interface, which is represented by component *plugin interfaces*, the *REST server* depends on this component. The *REST server* also acts as an intermediary in communication between *client* applications, and individual *plugin* component.

Component *plugin interfaces* is an abstraction of each *plugin*, that is supposed to be loaded by the *REST server*. The *REST server* itself, as was already stated, depends on *plugin interfaces* for this exact purpose.

Every *plugin* component must implement one or more classes belonging to the *plugin interfaces* component that acts as *plugin* abstraction as a result. *Plugins* are primarily responsible for processing requests made by *client* components and constructing the responses for such requests. There is therefore a bidirectional information flow between these two components. The *plugins* themselves are managed by *REST server*, which also manages the communication between them and *clients*.

Any *client* component communicates with one or more *plugin* components to obtain information. This dataflow itself is directed by the *REST server*.

The subject of this master thesis is the implementation of the *REST server* and *plugin interfaces*. This step will allow to implement *plugin* and *client* components in the future for specific purposes and thus brings the COMPASS API to fully functional state. However, implementation of *client* applications and any *plugins* that they require for correct operation are beyond the scope of the thesis, as the possibility to easily expand functionality of other software by using the COMPASS API should already be considered as a large benefit that

stems from the conducted work. Nevertheless, for demonstrative purposes a modified version of an application which is currently in use on the COMPASS experiment is provided along with necessary *plugins*, which uses the COMPASS API instead of direct database access to work with data sources. See appendix E for installation instructions for this application.

5.2 Plugin Interfaces

plugin interfaces component serves as a common interface between individual *plugins* loaded by the *REST server*, and *REST server* itself. Since these components are designed to be built independently, common interface for correct communication is required. This mechanism is implemented using *Qt plugins* component, which is described in section 4.2.2.

The structure of the component is described in section 5.2.1. Individual classes and header files that compose this component are then described in sections 5.2.2 to 5.2.8.

5.2.1 Component's Structure

The central class of this component is the `PluginInterface` class, which serves as base class for other provided *plugin interfaces*, and defines functionality that is common to all *plugins*. Classes that fulfill the role of interfaces for *plugins* consist of class `DatabaseHandlerInterface` for database *plugins*, and class `RequestPluginInterface` for request *plugins*. Additionally, `DatabaseAccessPlugin` class is intended to be inherited by request *plugins* that require to access a database. However, this class does not inherit from `PluginInterface`.

Furthermore, this component contains `LogOptions` and `Socket` classes, which mediate transfer of data between *plugins* and *REST server*, and header file section 5.2.6, which provides functions for working with *Qt*'s Meta-Object system.

The classes that form this component are pictured in UML class diagram in figure 5.2. They are also described in sections 5.2.2 to 5.2.8.

5.2.2 DatabaseAccessPlugin

`DatabaseAccessPlugin` is an abstract class, which is supposed to be inherited by request *plugins* that require an access to a *MySQL* database. Such *plugins* shall use multiple inheritance and inherit two classes. The `RequestPluginInterface` class must be inherited to allow the *REST server* to identify the dynamic library as a request *plugin*. In addition, the *plugin* shall inherit this class to gain the database access functionality.

The database that is meant to be accessed is identified by `QString` identifier, which shall in the case of intended manner of usage have the same value as the `QString` returned by the `PluginInterface::identifier`, method of the `DatabaseHandlerInterface`-derived *plugin* that holds the connection information for requested database.

SQL queries are executed by using `DatabaseAccessPlugin::databaseQuery` method, which accepts the above mentioned identifier as an argument, and returns an object of type

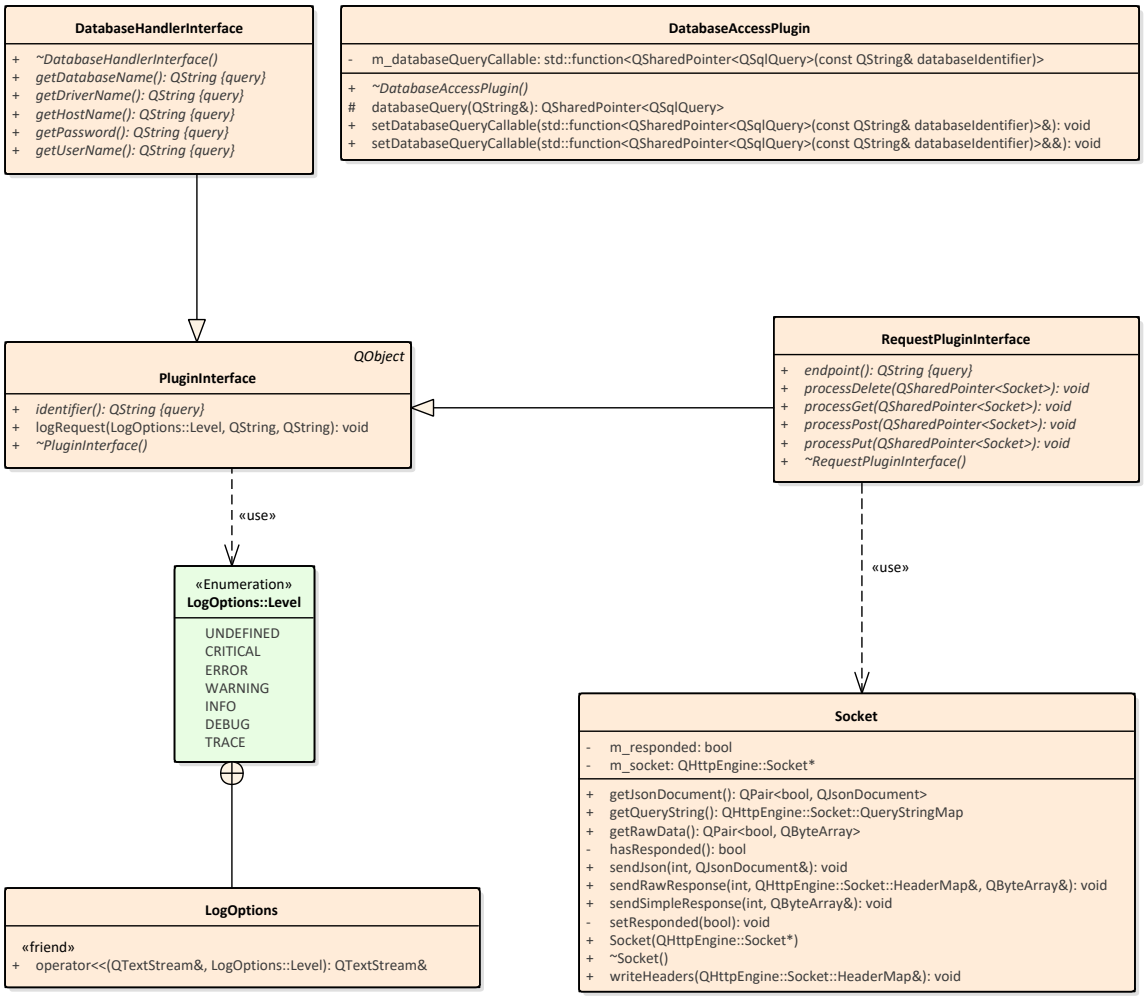


Figure 5.2: UML class diagram of *plugin interfaces*

`QSharedPoiter<QSqlQuery>`. The method does so by calling a stored callable object, which must be first registered by the application that loads the *plugin* which implements this interface, otherwise an exception is thrown when the derived class calls this method.

5.2.3 DatabaseHandlerInterface

`DatabaseHandlerInterface` abstract class is derived from `PluginInterface` class, and is itself intended to be derived by *plugin* classes which contain database connection details.

The *plugin* thus contains methods that return the name of the required database driver, hostname of the computer which contains the database server and port number which is supposed to be used, username and password for logging into the database server, and the name of the database to be used.

`DatabaseHandlerInterface` however does not contain the logic to open and keep the actual

connection to a database. The database opening logic instead resides in *REST server* in class `DatabaseConnectionsManager`, which is described in section 5.3.6.

5.2.4 LogOptions

`LogOptions` is a helper class for communication between the `PluginInterface` abstract class and a logging system which is implemented in the application that loads the *plugin* that is derived from `PluginInterface`. This communication is described in section 5.2.5.

The class encapsulates nested `enum class` type called `Level`, whose enumerators represent the severity of a logging message. This parameter is sent by the `PluginInterface::logRequest` signal in order to indicate the importance of the message. For description of how log message severity is processed, see section 5.3.7.

Additionally, the class also includes an overload of the stream operator for class `QTextStream`, which prints the name of an enumerator as text. This stream operator overload makes use of header file *qtmetasystemfunctions*, which in turn uses compile time type introspection of enumeration types, and allows to obtain the name of the enumerator.

However, technical limitations of this feature in the used *Qt* version are the reason why the `Level enum class` had to be placed in the `LogOptions` class. This class was artificially introduced only for this purposes, and would not otherwise be required.

5.2.5 PluginInterface

Class `PluginInterface` is an abstract class that is designed to act as a base class for interface classes for *plugins* of single type. Such types of *plugins* are described in sections 5.2.3 and 5.2.7.

This class defines pure virtual method `PluginInterface::identifier`, which shall return a unique identifier of the *plugin* that is indirectly derived from this class.

Additionally, this class defines *Qt* signal `PluginInterface::logRequest`. This signal may be used to request an output to the logging system of the application, which loads the *plugin* that implements this interface. Since the *plugin* is in its essence a dynamic library and has no direct access to other components of the parent application, this is the most straightforward method of using the applications logging system. The application may connect this signal to a slot, which mediates the corresponding procedure that performs the log message output, and thus allow the *plugin* to call this slot.

5.2.6 qtmetasystemfunctions

qtmetasystemfunctions is not a class, but a header file that contains several helper functions which simplify work with *Qt*'s Meta-Object system, specifically with it's features for enumeration types type introspection. It is thus not depicted in the UML class diagram in figure 5.2, but is listed here for completeness.

The header file contains two functions. Function `qEnum2CString` is a template function that returns a string literal representation of an enumerator passed to it as an argument. The second function, `qString2QEnum`, works in the opposite direction, and transforms a `QString` representation of an enumerator to the actual enumerator.

Note that for both functions, the enumeration type must be registered in the *Qt*'s Meta-Object system by using the macro `Q_ENUM`, otherwise both these functions will fail at compile time.

5.2.7 RequestPluginInterface

`RequestPluginInterface` is an abstract class derived from the `PluginInterface`. This class serves as the base class for all *plugins* that define additional request types for the *REST server*. This design aspect of the system is described in section 5.1.

It adds additional pure virtual methods in addition to those declared in `PluginInterface`. The implementations of these methods in derived classes provide the processing of individual types of requests. There are four types of these requests, in accordance with the CRUD principle, and therefore four methods must be implemented in derived classes. These methods consist of `RequestPluginInterface::processPost` for processing of requests for creating data, `RequestPluginInterface::processGet` for processing of requests for reading data, `RequestPluginInterface::processPut` for processing of requests for updating data, and finally `RequestPluginInterface::processDelete` for processing of requests for deleting data.

Additionally, the *plugins* which are derived from this class must implement pure virtual method `RequestPluginInterface::endpoint` which shall return the path part of the *URI* that must be used when accessing the service.

5.2.8 Socket

The purpose of the `Socket` class is to serve as a wrapper around the `QHttpEngine::Socket` class from the *QHTTPEngine* library, which is described in section 4.4. Since this class represents an active connection to a *client*, it is crucial for correct operation of COMPASS API. The `Socket` class from *plugin interfaces* component is passed to request processing methods which are implemented in *plugins* that implement the `RequestPluginInterface` interface.

The wrapper was developed with two goals in mind. The first is to restrict access to some of the methods from `QHttpEngine::Socket`, so as to protect the *REST server* from incorrectly implemented *plugins*. Additionally the wrapper is tasked with catching any exceptions that may be thrown by the *plugin*'s methods, thus further securing the *REST server* from misbehaving *plugins*.

The main reason for implementation of a wrapper class is however to provide additional methods, which allow to read and write data in certain format to the socket.

Methods that facilitate data reading from the socket consist of methods for obtaining the *URI* query string of the request, and for reading any available message body content stored either as raw data, or in the *JSON* format, which is described in section 4.5. This is usually relevant

to *POST* and *PUT* requests processing, since these request types must contain the data that is supposed to be written or updated. These methods are the following:

- `Socket::queryString` parses the request's query string, which is then provided as an instance of `QHttpEngine::QueryStringMap`, which contains key-value pairs that contain the data from the query string parameters.
- `Socket::readRawData` serves to read the message body as raw data. If any are available, they will returned as an instance of `QByteArray`.
- `Socket::readJson` serves to read a *JSON* document sent by the *client*, if one is available. The document is returned in the form of a `QJsonDocument` class.

The interfaces for construction of responses consists of multiple methods as well. Firstly, it allows to send simple responses which contain only the response code. Secondly, it allows to send a message containing data in the *JSON* format, which is described in section 4.5. The corresponding method accepts an instance of `QJsonDocument` class, serializes it and writes it to the *TCP* socket used for communication between the *REST server* and *client*. Finally, it provides a generalized interface for writing entirely user-defined response with custom headers and body content.

- `Socket::sendSimpleResponse` sends a response that consists of an *HTTP* status code and optionally a short string describing the reason for the status.
- `Socket::sendJson` allows to send a *JSON* document to the *client*, which is passed to the function in the form of a `QJsonDocument` class. Optionally, the user may also select the status code of the *HTTP* response.
- `Socket::sendRawResponse` serves to send a custom response, with predefined headers and message body.

5.3 REST Server

REST server is central component of COMPASS API, and its development is the main focus of this thesis. It's functionality may be divided into two tasks.

Firstly, *REST server* is responsible for management of all *plugins*, which are described in section 5.4. The management consists of loading the *plugins* at runtime from shared libraries, and from providing the *plugins* with access to necessary resources, which are then used to conduct the specified tasks.

Secondly, this component acts as the middle man in communication between *client* components and *plugin* components by relaying requests made by *clients* to *plugins* that implement the requested service. *REST server* therefore contains all programming logic that deals with network communication, and thus separates the *plugins* from this concern and simplifies their development.

The structure of the component is described in section 5.3.1. Individual classes that compose this component are then described in sections 5.3.2 to 5.3.13.

5.3.1 Component's Structure

The program follows common model of *Qt* applications, where the `main` function serves to setup the `QCoreApplication` class, instantiates the central class that holds application-specific logic and starts an event loop. The application is from that point driven entirely by events, which trigger a predefined process. Such event in the case of this application is accepting a new *TCP* connection from a *client* application, for which a response is then composed.

The central class of the program, and the entry point into the structure of *REST server* is the `Core` class. It holds instances of other classes that are required to provide necessary functionality of the *REST server*, and performs other necessary setup tasks. Classes held by `Core` class consist of `CommandLineParser`, `RestServer`, and `RequestPluginManager`.

`CommandLineParser` class uses *Qt*'s `QCommandLineParser` class to process program's command line arguments. Class `RestServer` handles the *TCP* network communication with *clients*, and acts as the abstraction layer for dealing with *HTTP* protocol. It's main purpose is to parse requests and relay them to other classes for processing, and subsequently to send responses defined by these classes back to the *client*. As stated in requirements REQ-22 and REQ-23, processing of the requests is performed by *plugins* loaded by the *REST server*. Responsibilities of `RequestPluginManager` consist of management of request *plugins*. This class both loads the *plugins* from shared library files and holds the resulting instances. It also contains an instance of `DatabaseConnectionsManager` class that manages *REST server*'s database connections.

The structure of COMPASS API *REST server* is depicted in UML class diagram presented in figures 5.3 to 5.8. For presentation purposes, the class diagram in question was split into several parts. Each of these parts deals with certain subtask performed by the *REST server*, and contains a necessary subset of classes that are needed to describe it. As a result, classes may be present in more than one of the figures. However, every class is present on at least one of the diagrams. Furthermore, if a class is only minorly significant in the context of particular figure, it may be displayed without a description of its members. Full specification of each class is nevertheless depicted on at least one of the figures. It should also be noted that classes which have a blue background belong to the *plugin interfaces* component, and their description is present in section 5.2.

In addition, the classes depicted in diagrams in figures 5.3 to 5.8 are further described in sections 5.3.2 to 5.3.13.

5.3.2 AbstractPluginManager

`AbstractPluginManager` is an abstract template class, that is meant to be inherited by classes that manage *Qt plugins* loaded from dynamic libraries. To simplify tasks associated with this function, `AbstractPluginManager` provides generalized facilities that allow the derived classes to load *plugins* without having to implement additional procedures.

The class has two template parameters. The first determines the type of the *plugin* that will be managed by the subclass. `AbstractPluginManager` automatically checks during com-

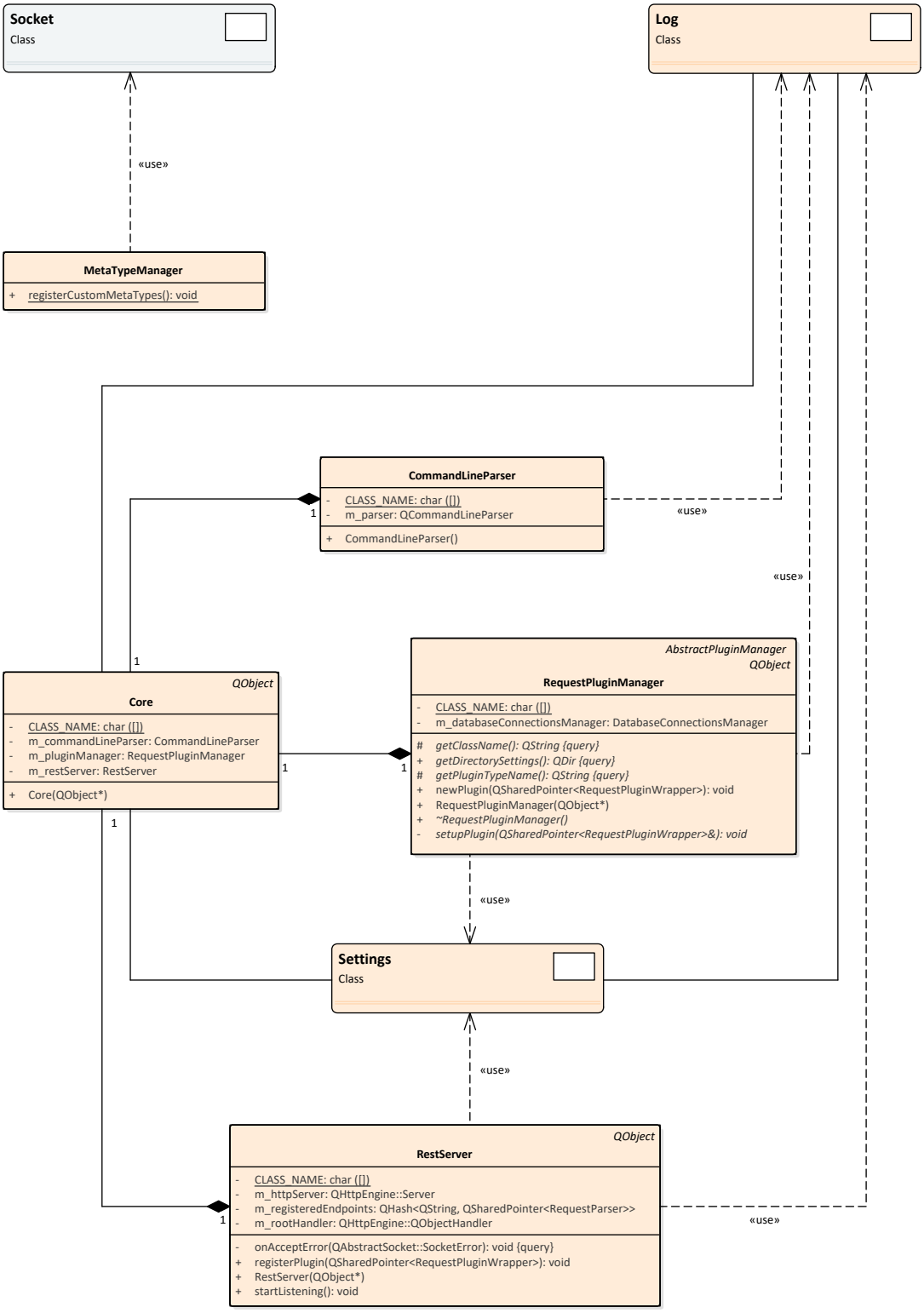


Figure 5.3: UML class diagram of *REST server*: core structure

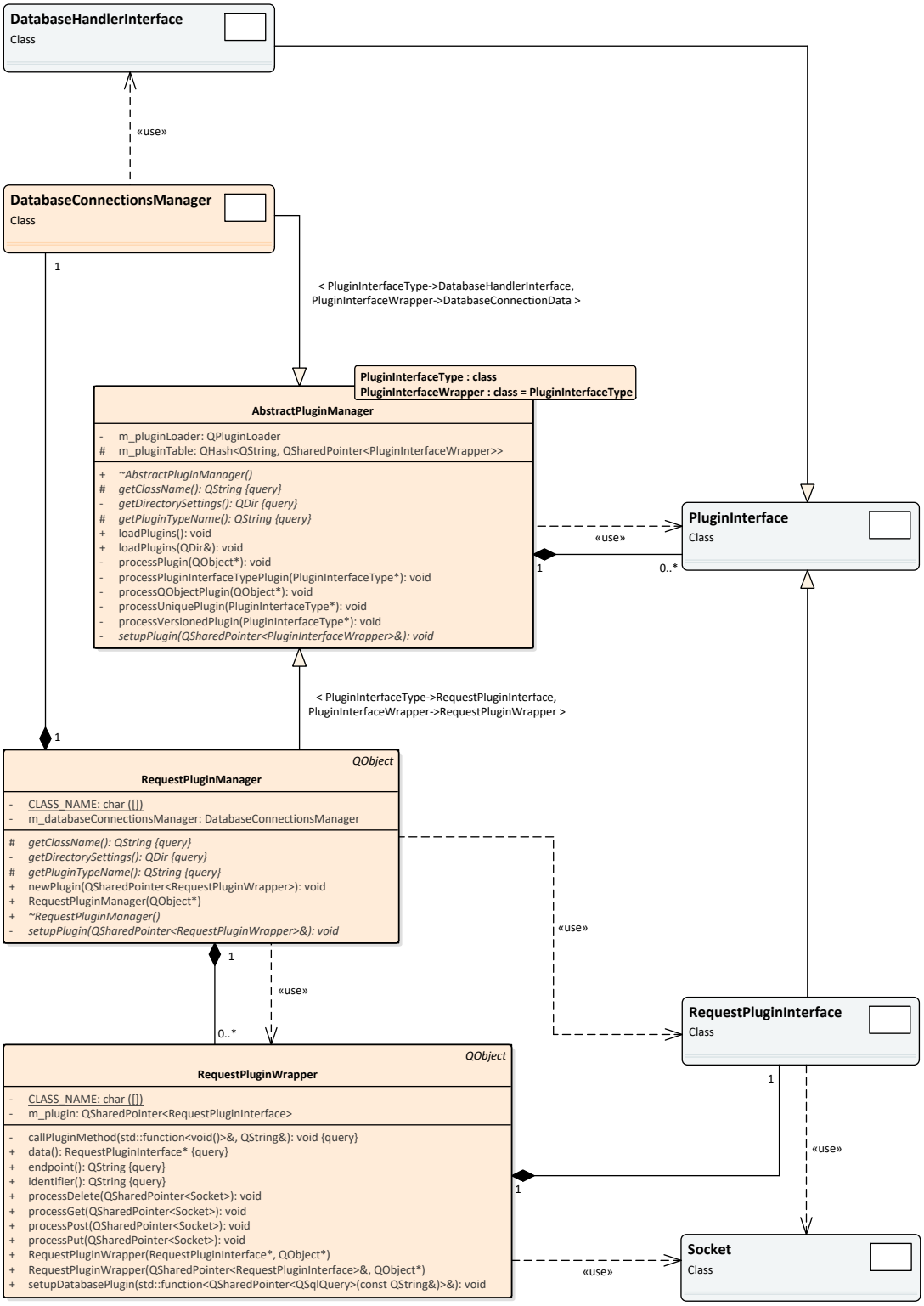


Figure 5.4: UML class diagram of *REST server*: request *plugins* handling

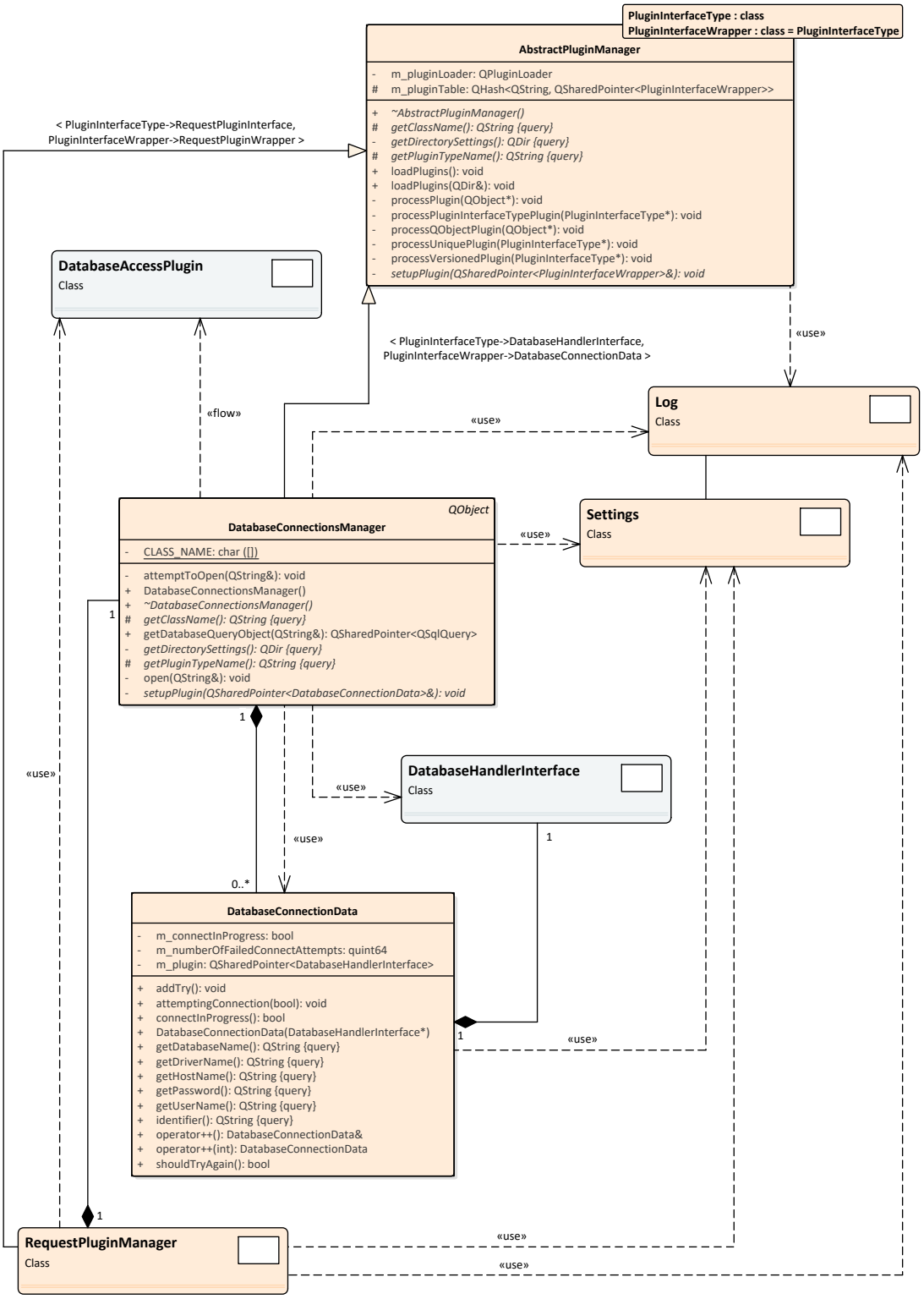


Figure 5.5: UML class diagram of *REST server*: database *plugins* handling

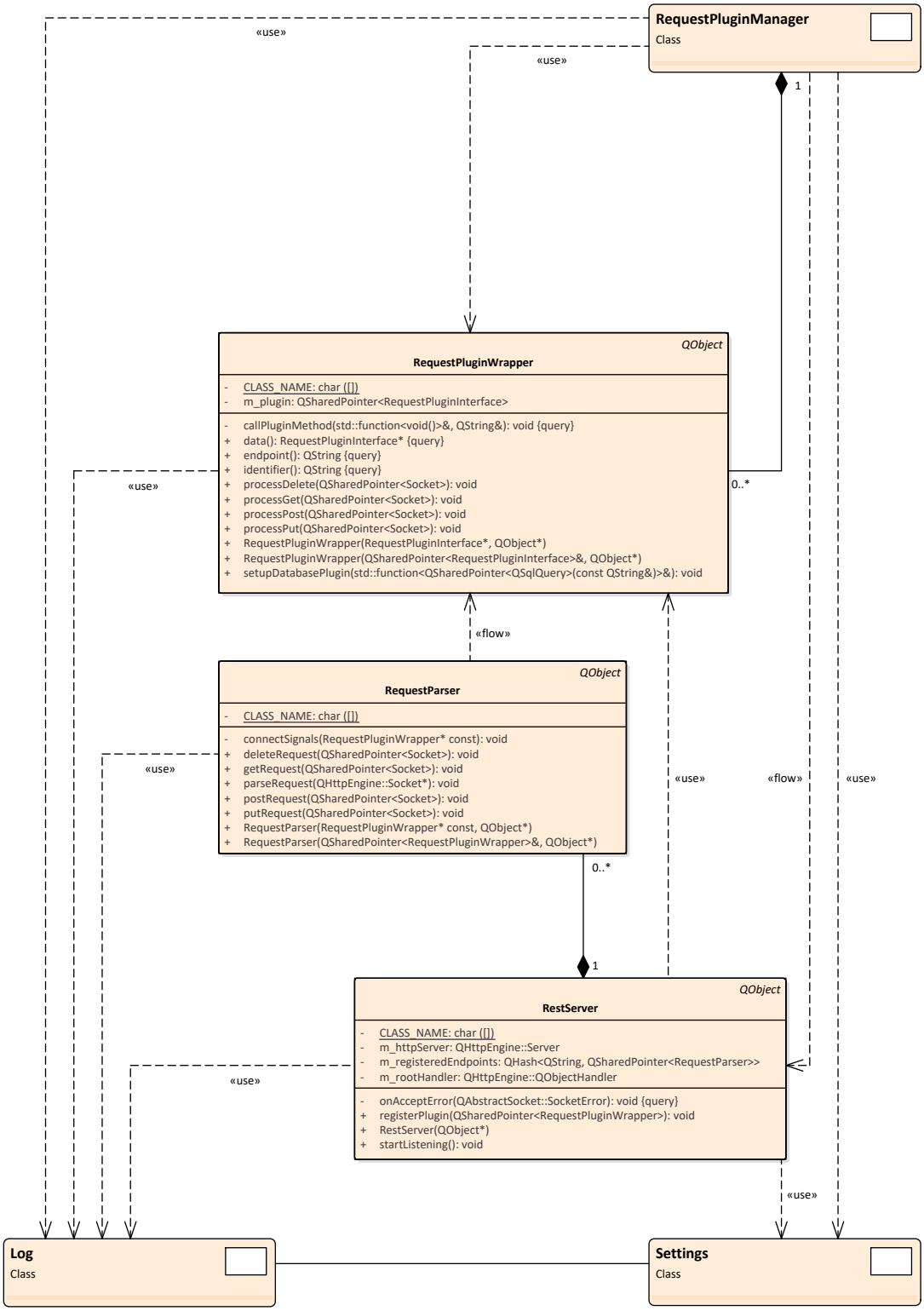


Figure 5.6: UML class diagram of *REST server*: *REST* server

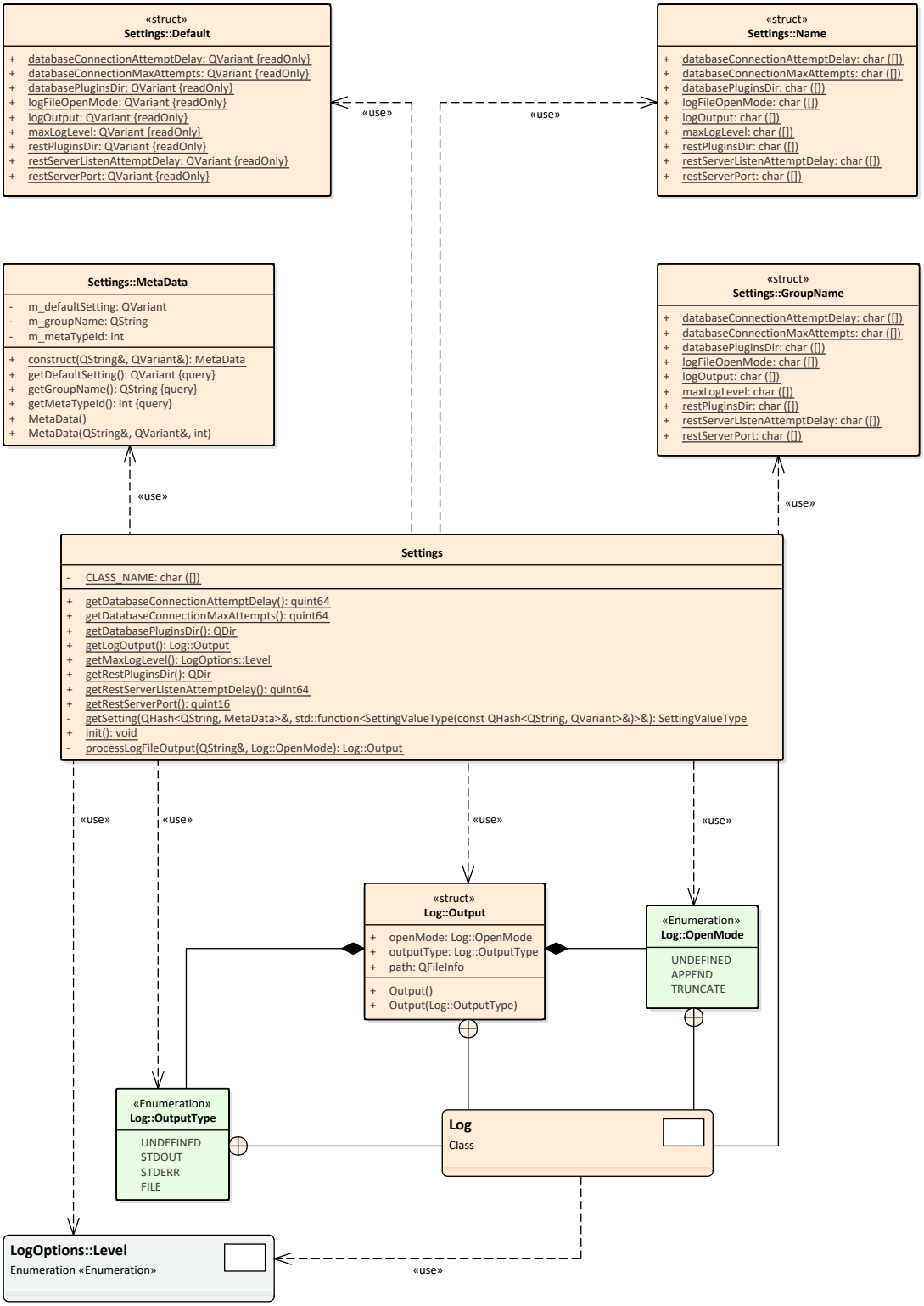


Figure 5.7: UML class diagram of *REST server*: settings handling

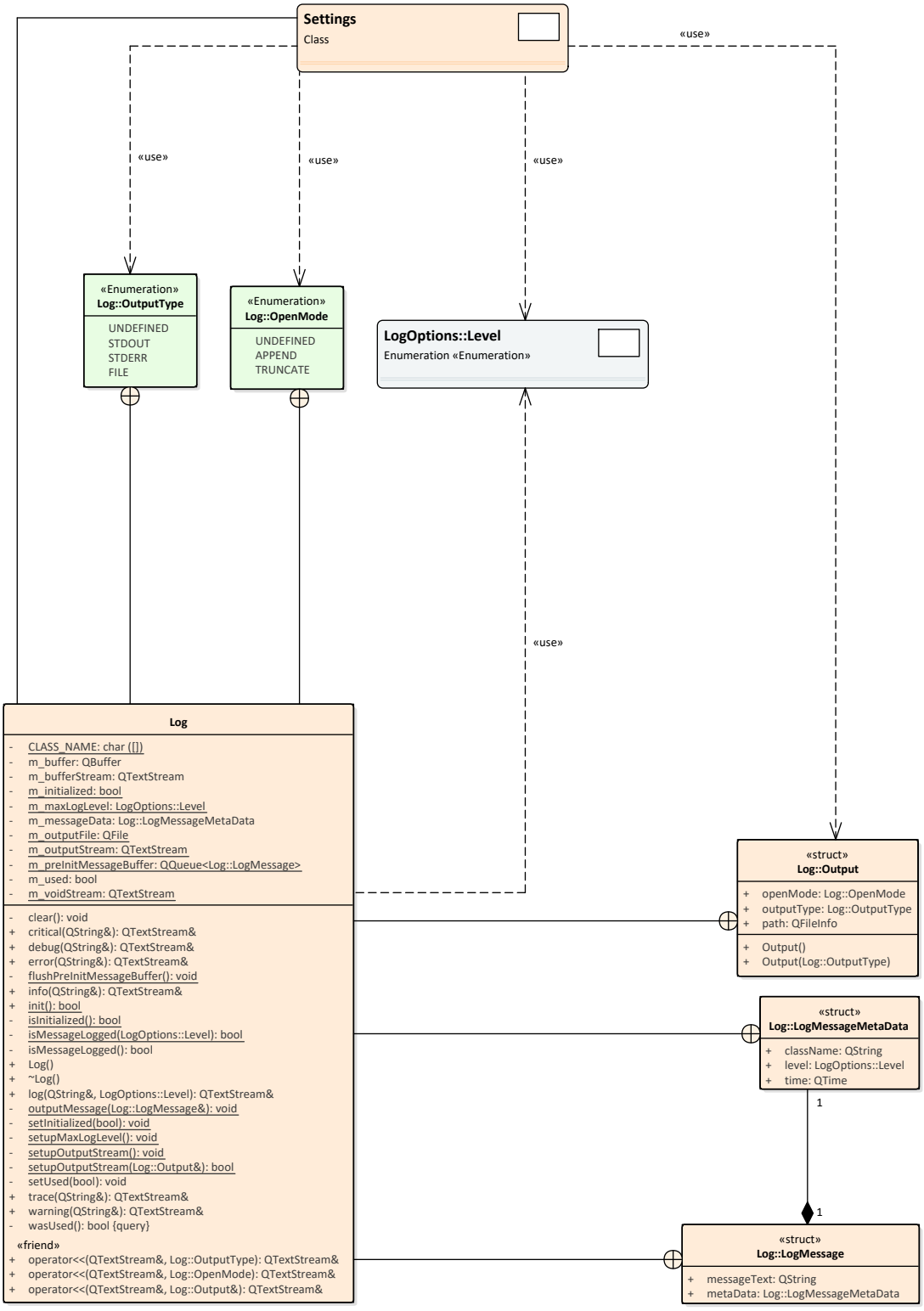


Figure 5.8: UML class diagram of *REST server*: log output

pilation that this argument is a class derived from `PluginInterace` class that is described in section 5.2.5.

The second parameter is by default equal to the first parameter and decides what type will be internally used to store the *plugin*. In all cases, the type used to store the *plugin* is actually a `QSharedPointer` smart pointer class, which manages a pointer to the type defined by the second template argument. Because all COMPASS API *plugins* are indirectly derived from `QObject` class to support *Qt*'s signals and slots mechanism, it is impossible to copy-construct them, and therefore it is impossible to store them inside container classes, since these usually require copy constructor. One possible solution to overcome this limitation is to store a pointer to the *plugins*. Smart pointers have the additional ability of almost entirely removing the difficulties associated with management of lifetime of raw pointers.

Classes which intend to inherit from `AbstractPluginManager` must implement a total of four pure virtual methods. All of these methods are called internally from other, already implemented methods of `AbstractPluginManager`. Their implementation serves to define behavior specific to *plugin* type that is defined by the first template argument of this class. The most remarkable of these methods is `AbstractPluginManager::setupPlugin`, which defines the actions that need to be taken to successfully register *plugin*, once it has been verified to be valid. `AbstractPluginManager::getDirectorySettings` should return the default directory for *plugin* search. The remaining two methods that are called `AbstractPluginManager::getClassName` and `AbstractPluginManager::getPluginTypeName` and should return a `QString` that contains the name of the class and the name of the *plugin* type respectively. Both these methods are used by `AbstractPluginManager` for logging purposes.

The entire public functionality of `AbstractPluginManager` class is available through method `AbstractPluginManager::loadPlugins`, which is present in two overload variants. The first accepts a directory as an argument of type `QDir` and attempts to load all files inside the directory as *plugins*. The second overload accepts no arguments, and calls the first overload using the value returned by `AbstractPluginManager::getDirectorySettings` method.

Plugin loading process mainly consists of various checks that assert whether the *plugin* meets certain criteria. The performed checks consist of following operations, which are done in the same order as listed:

1. Assessment of whether the file is a *Qt plugin*.
2. Evaluation of what interface does the *Qt plugin* implement.
3. Comparison of the *plugin*'s *plugin interfaces* version and the *plugin interfaces* version used by the *REST server*. The versions must match.
4. Iteration over existing *plugins* to avoid any potential conflict of *plugin* identifiers, which must be unique.

The criteria that any *plugin* must meet are detailed in section 7.1.

In case that one of the checks fails, the file is ignored. However, if these conditions are satisfied, *plugin*'s `PluginInterace::logRequest` signal is connected to application's logging system, and the *plugin* is passed to the `AbstractPluginManager::setupPlugin` method for *plugin* type specific processing. This method may also perform additional integrity checks that may result

in the *plugin* being rejected by the class derived from `AbstractPluginManager`. The process is depicted in UML activity diagram in figure 5.9.

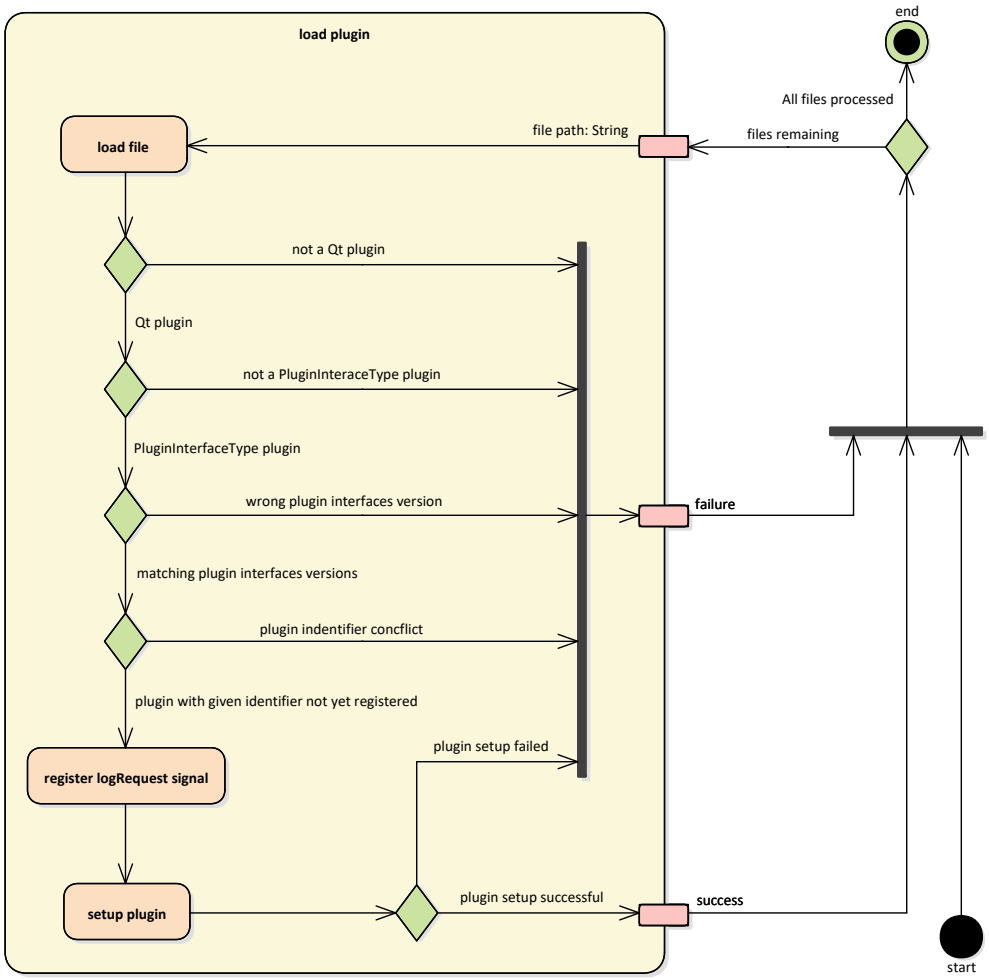


Figure 5.9: UML activity diagram of *plugin* loading process

5.3.3 CommandLineParser

The class `CommandLineParser` is responsible for parsing command line arguments passed to the application. It is built around *Qt*'s `QCommandLineParser` class to achieve this goal.

The application currently supports two command line options, which are listed in table 5.1 along with their descriptions.

The current capabilities of this class are somewhat limited, however, due to easy extensibility of the `QCommandLineParser` class, and in turn also `CommandLineParser`, this class provides the option of easily extending the application with further command line arguments, should the need arise.

option	description
`-h`, `--help`	Displays description of the application and lists all available command line arguments with description.
`-v`, `--version`	Displays version information.

Table 5.1: List of command line arguments available in *REST server*

5.3.4 Core

Class `Core` is the primary class of the *REST server* component. It directly or indirectly holds instances of every other class that is required to exist throughout the entire lifetime of the program. It is also responsible for all setup tasks and for connecting *Qt* signals and slots of the member classes.

This class holds instances of `CommandLineParser`, `RestServer`, and `RequestPluginManager` that are described in sections 5.3.3, 5.3.10 and 5.3.12 respectively.

Additionally, the class is also responsible for calling the initialization methods of classes `Log` and `Settings`, which are described in sections 5.3.7 and 5.3.13. These methods need to called in order to ensure correct functioning of these classes.

It also initiates the *plugin* loading process for class `RequestPluginManager`, and connects its signal `RequestPluginManager::newPlugin` to appropriate slot in class `RestServer`, so that this components gets notified of new *plugins*, and may update its request routing logic to reflect this circumstance.

5.3.5 DatabaseConnectionData

`DatabaseConnectionData` class serves as a wrapper class around *plugins* that are derived from `DatabaseHandlerInterface` abstract class, which is described in section 5.2.3.

The class stores a single *plugin* pointer inside a `QSharedPointer` smart pointer, and provides access to all methods implemented in the `DatabaseHandlerInterface`-derived *plugin* through methods with the same name as those present in the *plugins*.

Additionally, it contains mechanisms required to control the database connection process. It allows to check, if the `DatabaseConnectionsManager` class, which is described in section 5.3.6, is currently in the process of attempting to establish a connection with the database that is described by the stored *plugin*. Moreover, `DatabaseConnectionsManager` may be configured to execute only a limited amount of connection attempts before giving up. The counter which represents the number of attempts is stored inside this class as well. Finally, several methods which enable to read and manipulate these variables are provided.

This class is used in class `DatabaseConnectionsManager` as the second template argument during inheritance of `AbstractPluginManager`, which is described in section 5.3.2. It is therefore used to store database *plugins* in the *REST server*.

5.3.6 DatabaseConnectionsManager

This class manages all network connections that are opened by the *REST server* to external database servers. These connections are then available to all request *plugins*, which implement the `DatabaseAccessPlugin` that is described in section 5.2.2, and any other components that require database access.

`DatabaseConnectionsManager` employs the *Qt SQL* module to manage the connections. Connections to the database are opened by using the `QSqlDatabase` class. Individual *SQL* queries are executed using the `QSqlQuery` class.

To gain the information required to open the actual connections, the class uses *plugins* derived from `DatabaseHandlerInterface` class, which is described in section 5.2.3. These *plugins* contain information necessary to open the database connection, including the name of the *Qt SQL* driver plugin that shall be used. As a result from this desire to use a plugin system, this class inherits from `AsbtractPluginManager` class, which is described in section 5.3.2. During the inheritance, classes `DatabaseHandlerInterface` and `DatabaseConnectionData`, which are described in sections 5.2.3 and 5.3.5 respectively, were used as `AsbtractPluginManager`'s template arguments.

When attempting to connect to a database, `DatabaseConnectionsManager` periodically attempts to open the connection, until it is successfully established. It is also possible to limit the number of connection attempts through application's settings, which are described in section 5.3.13. If the maximum number of unsuccessful connection tries was reached, `DatabaseConnectionsManager` leave the connection unopened, but if another component makes a request to use the connection, new attempts will be made. The process is depicted in UML activity diagram in figure 5.10.

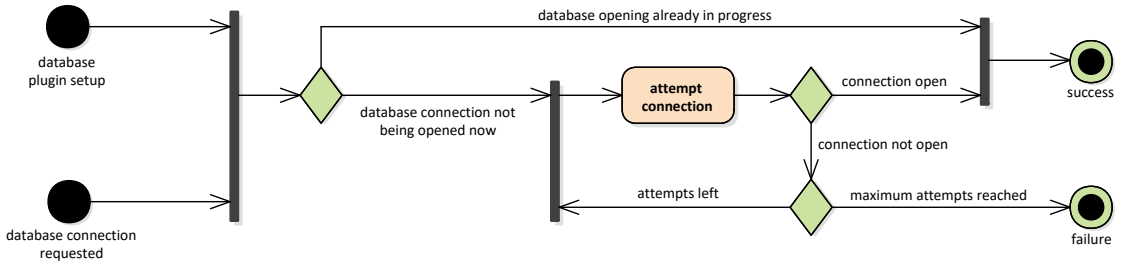


Figure 5.10: UML activity diagram of connection opening process

The database connections are available to `DatabaseAccessPlugin`-derived *plugins* through method `DatabaseConnectionsManager::getDatabaseQueryObject`. It accepts a `QString`, whose purpose is to identify the database *plugin* that contains the connection specification, and returns a `QSharedPointer<QSqlQuery>` class, which may be used to execute *SQL* queries. If there is no connection available under the given identifier, or the database connection is not available, the `QSharedPointer` will be empty. The process that is used by the actual *plugins* to access this method requires a correct setup of such *plugin*. This process is described in section 5.3.11.

5.3.7 Log

The `Log` class aims to facilitate a logging system for use by other classes composing the *REST server*.

Depending on application's configuration, which is read using the `Settings` class, `Log` class may output the log to either standard output, standard error output, or a file.

The logging system employs a message importance system, where each message is assigned a severity level by the user. Based on application's settings, which are described in section 5.3.13, the `Log` class decides whether the message will be written to the actual output. This way, the user may limit the amount and detail of log messages using only the configuration file, without requiring to change the corresponding parts of the source code. There are six severity levels available, which are listed in table table 5.2. The order in the table lists the levels from most important to least important.

message severity	intended use
`CRITICAL`	Critical errors that result in termination of the application
`ERROR`	Errors that result in significant loss of functionality, but other functions and services of the application remain unaffected
`WARNING`	Errors that affect the applications functionality only to a lesser degree, or can be corrected by the applications logic without intervention
`INFO`	Messages informing about significant events, which are part of standard execution of the application
`DEBUG`	Debugging messages, which inform about the application's standard execution in more detail
`TRACE`	Messages containing the most details about the application's standard operation, usually used to provide information to the developer about specific call order of functions and methods

Table 5.2: COMPASS API log message severity levels

The relevant settings in application's configuration file determines the lowest allowed message severity level. For example, when the configuration is set to `INFO`, only messages with severity levels `CRITICAL`, `ERROR`, `WARNING` and `INFO` will be written to log output.

It should be noted that the definition of message severity levels is actually contained inside the *plugin interfaces* component inside the `LogOptions` class, which is described in section 5.2.4. This layout was chosen to allow logging from *plugins*, since every possible implementation of such feature requires to have the knowledge of the logging levels at its disposal. The implemented *plugin* logging feature is described in section 5.2.5.

The setup of all of the settings described in this Section is automatically performed upon calling static method `Log::init`.

Each logging message is represented by a single instance of `Log`, which may be used to specify the contents of the message, which is then processed by the logging system upon the instance's destruction. The message contents are defined by writing to a stream, which is provided

by methods `Log::critical`, `Log::error`, `Log::warning`, `Log::info`, `Log::debug`, `Log::trace` or `Log::log`. These methods allow to define messages that are logged using corresponding severity level, with the exception of `log`, which accepts additional argument that defines the desired logging level. All of these methods however require the name of the component using it. This name is used for logging purposes and is displayed as part of the logging message.

Example of using the `Log` class, which is also the recommended approach, is provided in listing 5.1. A possible output resulting from the displayed code may be the message "`12:34:56 INFO [MyClass] never odd or even`" (without the quotation marks).

```
1  Log().info("MyClass") << "never odd or even";
```

Listing 5.1: Example usage of `Log` class

Internally, `Log` contains a `QBuffer` instance, which is written to by the stream writing operation. Upon the destruction of the instance, a decision is made based on the allowed message severity levels, whether the `QBuffer` will be written to log output or discarded. It should be noted that given instance of log class will overwrite it's internal buffer if one of the methods, which provide the access to the stream, is called again on the same instance. As a result only the last message will be processed upon the destruction of the current instance. The whole message output process is depicted in UML activity diagram in figure 5.11.

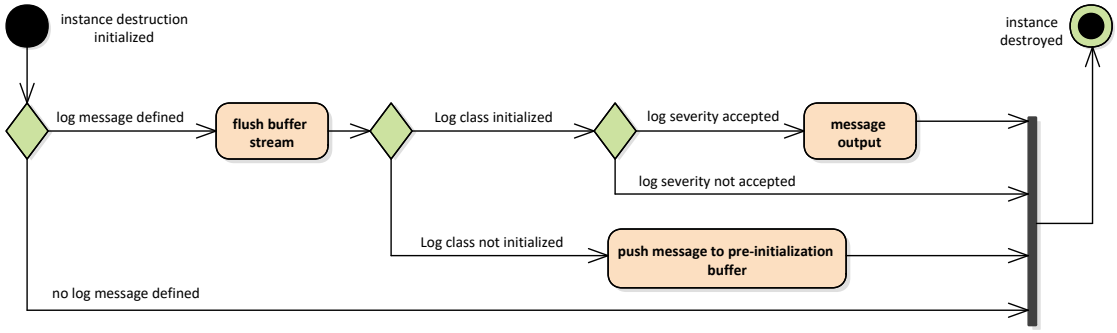


Figure 5.11: UML activity diagram of log message output

Finally, `Log` class also provides a pre-initialization buffer. This buffer is used to store all messages before the class is initialized using the `Log::init` method. Upon successful initialization, the buffer is flushed to output, and logging will be from that point onward performed in a standard manner.

5.3.8 MetaTypeManager

`MetaTypeManager` is responsible for registering custom types to *Qt*'s Meta-Object system. In it's current version, `Socket` class from *plugin interfaces* component, which is described in section 5.2.8, is the only class that requires to undertake this procedure, as it is passed as an argument in a signal/slot communication.

`MetaTypeManager` is the only class, that is not either directly or indirectly used from the `Core` class. All of the functionality that is required from this class is implemented inside a static method `MetaTypeManager::registerCustomMetaTypes`, which is called from `main` function.

5.3.9 RequestParser

`RequestParser` serves as an intermediary between the `RestServer` class, and individual request *plugins*, which are described in section 5.4. `RestServer` is responsible for routing the requests made at a given endpoint to corresponding `RequestParser`, and the `RequestParser` instance is in turn responsible for calling the appropriate *plugin* method, depending on used *HTTP* method.

As a result, there are as many instances of `RequestParser` present in *REST server*, as there are loaded *plugins*. These are stored inside the `RestServer` class. Each `RequestParser` holds a reference to corresponding `RequestPluginWrapper`. This reference is held as a smart pointer of type `QSharedPointer`, which shares its reference counter with a `QSharedPointer` of the same type that is stored in `RequestPluginManager` class.

`RequestParser` instance is tasked with calling the appropriate *plugin* method through its `RequestParser::parseRequest` slot, which accepts a pointer to `QHttpEngine::Socket` class from the *QHTTPEngine* library, which is described in section 4.4. `RequestParser` employs the `QHttpEngine::Socket` to determine the method that was used to place the request. If it is one of the supported *HTTP* methods (*POST*, *GET*, *PUT*, *DELETE*), `RequestParser` uses the `QHttpEngine::Socket` instance to create an instance of `QSharedPointer<Socket>`, where the `Socket` in this case is an instance of `Socket` class from the *plugin interfaces* component that is described in section 5.2.8. The *plugin interfaces* `Socket` is then used as an argument during a call to appropriate *plugin* method.

5.3.10 RequestPluginManager

Class `RequestPluginManager` is the primary class that implements the *REST server*'s ability to use external definitions of allowed request types and their processing, which are in the form of *Qt plugins*.

The request *plugins* shall implement the `RequestPluginInterface`, which is described in section 5.2.7. This interfaces thus serves as the communication interfaces between the *plugins* and the *REST server*.

`RequestPluginManager` is tasked with loading the *plugins* and storing their instances. As a result, it is derived from the class `AsbtractPluginManager`, which supplies most of the necessary logic to achieve this. During the inheritance of this abstract base class, `RequestPluginManager` uses the `RequestPluginInterface` interface as the first template argument. To store the *plugins*, the class `RequestPluginWrapper` is used, and is therefore used as the second argument.

During the setup of the *plugin*, after all of the *plugin* loading steps that are described in section 5.3.2 were taken, the setup of `DatabaseAccessPlugin`-derived *plugins* is performed. The setup routines for these *plugins* are however implemented in `RequestPluginWrapper`,

and `RequestPluginManager` only calls the corresponding method. This setup method requires a callable that returns a `QSqlQuery` object for given database as an argument. This signature is matched by method `DatabaseConnectionsManager::getDatabaseQueryObject`

Finally, and the end of the setup process, `RequestPluginWrapper` class emits its *Qt* signal `RequestPluginWrapper::newPlugin` to notify any listening components that new *plugin* was loaded.

5.3.11 RequestPluginWrapper

Class `RequestPluginWrapper` acts as a wrapping class for request *plugins*, which are derived from `RequestPluginInterface` class described in section 5.2.7. In the *REST server* application, it is used for storage of loaded *plugin* instances.

The instance of the *plugin* is stored inside a `QSharedPointer`, and this wrapper provides access to all of the *plugin*'s methods through methods with the same name. The most important of these members are the slots that call the *plugin* methods which process *REST* requests. Not only are these slots frequently used during standard operation of the COMPASS API, they are also significant since they are tasked with catching all exceptions that may be thrown by the *plugins*, and thus protect the server from crashing due to uncaught exceptions.

`RequestPluginWrapper` additionally contains the implementation of additional setup steps, if the plugin is also derived from `DatabaseAccessPlugin` class, which is described in section 5.2.2. This class is inherited by *plugins* that require to access the database. The setup is done by calling the method `RequestPluginWrapper::setupDatabasePlugin`. This method checks, if the *plugin* implements the `DatabaseAccessPlugin` using `dynamic_cast`. If the *plugin* passes the check, *plugin*'s function `DatabaseAccessPlugin::setDatabaseQueryCallable` with a callable that may be used to access the database.

Due to this requirement, `RequestPluginManager` holds an instance of database management class `DatabaseConnectionsManager`, which manages database connections for the entire application. Within the context of the *REST server* application, the callable that is used for this purpose is the method `DatabaseConnectionsManager::getDatabaseQueryObject`, which is described in section 5.3.6.

5.3.12 RestServer

`RestServer` is a class responsible for network communication between the server application, and *client* applications that want to use the COMPASS API.

This class is mostly based on the `QHttpEngine::Server` class from the *QHTTPEngine* library, which is described in section 4.4. Therefore, the implementation of *TCP* layer of network communication is supplied by the *Qt* framework, and the *HTTP* layer is implemented in the `QHttpEngine::Server` class.

Besides the network communication, `RestServer` class implements the logic to route the requests to the *plugin* which implements the service. First, any new *plugin* must be registered. This functionality is achieved by the slot `RestServer::registerPlugin`, which accepts

an instance of `QSharedPointer<RequestPluginWrapper>`. This class is used to obtain the name of the service through the `RequestPluginWrapper::endpoint` method, which is subsequently registered in the instance of `QHttpEngine::Server`.

Registration of request processing procedures is done in the *QHTTPEngine* library through classes derived from `QHttpEngine::Handler`. `RestServer` class uses an already implemented `QHttpEngine::QObjectHandler`, which calls a user defined method upon each new request. In this particular case, requests trigger a call of `RequestParser::parseRequest` method.

5.3.13 Settings

Class `Settings` implements the ability to read and parse configuration files for the purpose of modifying the application's functionality. It's public interface primarily consists of static methods, which may be used from other classes to access various application settings.

`Settings` uses the `QSettings` class to parse configuration files in the *ini* format. `Settings` however provides additional functionality when parsing such settings by providing the key names under which are individual settings stored, and also handles default values if the settings are unavailable or unreadable.

Some of the public methods may also read multiple *ini* keys and compose the setting returned to caller from them. The process of reading a setting is depicted in UML activity diagram in figure 5.12.

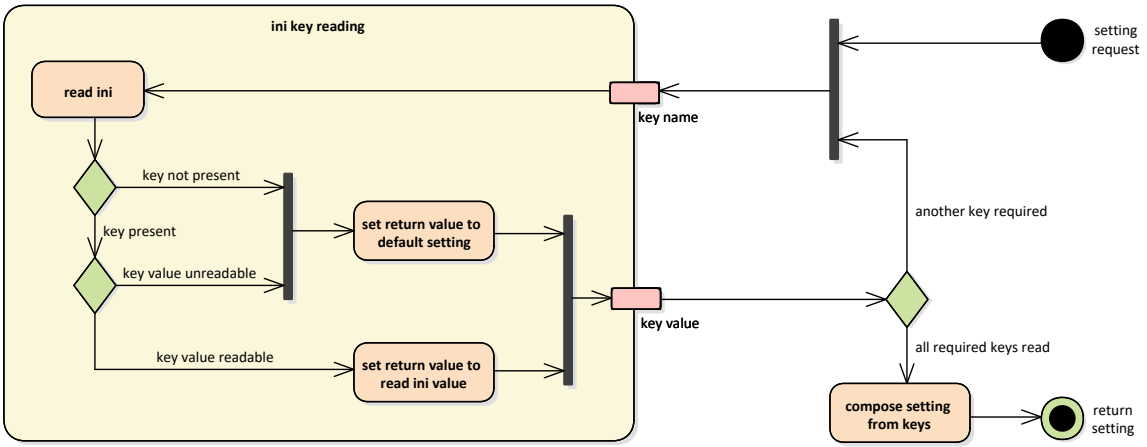


Figure 5.12: UML activity diagram of settings reading

Public methods provided by this class are listed in table 5.3 along with their respective return types.

The purpose of these methods is:

- **Settings::getRestPluginsDir** returns the directory that contains the request *plugin* shared libraries.

method	return value type
`Settings::getRestPluginsDir`	`QDir`
`Settings::getDatabasePluginsDir`	`QDir`
`Settings::getLogOutput`	`Log::Output`
`Settings::getMaxLogLevel`	`LogOptions::Level`
`Settings::getRestServerPort`	`quint16`
`Settings::getRestServerListenAttemptDelay`	`quint64`
`Settings::getDatabaseConnectionAttemptDelay`	`quint64`
`Settings::getDatabaseConnectionMaxAttempts`	`quint64`

Table 5.3: Software packages required for compilation of COMPASS API

- **Settings::getDatabasePluginsDir** returns the directory that contains the database *plugin* shared libraries.
- **Settings::getLogOutput** returns information about logging system. This information consists of the target of the logging system, which may be wither standard output, standard error output or file. Secondly, it contains a setting whether log files should be overwritten when new instance of *REST server* is started, or the log shall be appended to the previous log.
- **Settings::getMaxLogLevel** returns the setting containing information about allowed log message severity levels.
- **Settings::getRestServerPort** returns the number of the port which the *REST server* shall use to listen for incoming connections.
- **Settings::getRestServerListenAttemptDelay** returns the delay between application's attempts to start listening on given port if the previous attempt was unsuccessful.
- **Settings::getDatabaseConnectionAttemptDelay** returns the delay between application's attempts to connect to a database if the previous connection attempt was unsuccessful.
- **Settings::getDatabaseConnectionMaxAttempts** returns the maximum allowed number of database connection attempts before the *REST server* stops trying to open the connection.

The configuration file that will be used is determined by the `QSettings` class, and is described in [30]. Recognized configuration file keys are described in appendix C.

5.4 Plugins

Plugins are responsible for one of the principal properties of the *REST server*, as they enable it to load new definitions of how to process requests at runtime. *Plugins* make use of the *Qt plugins* component of the *Qt* framework, which is described in section 4.2.2. This library component provides an easy to use framework that allows an application to load

C++ dynamic libraries at any point during the program's execution. These dynamic libraries always contain one main class that implements an interface (or in strict *C++* terms abstract class), whose implementation is also known to the program which loads the libraries. Request definition *plugins* may therefore be developed and recompiled without the need to modify the *REST server* in any way.

The interface that all request *plugins* must implement is defined in the *plugin interfaces* component in the `RequestPluginInterface` class, and is described in section 5.2.7. In addition to this required interface, request *plugins* which are intended to work with data stored in a database may also optionally implement the `DatabaseAccessPlugin` that is described in section 5.2.2. This interface allows the *plugin* to use the database connections maintained by the *REST server*.

Moreover, another type of *plugins* is used by the COMPASS API. These *plugins* implement the `DatabaseHandlerInterface` class as its interface, and contain data necessary to connect to a database server. These *plugins* are loaded by the *REST server*'s instance of class `DatabaseConnectionsManager`, which is responsible for managing database connections used by the *REST server* and loaded request *plugins*. The class `DatabaseConnectionsManager` is described in section 5.3.6.

Development of *plugins* is outside the scope of this master thesis. Section 7.1 however provides guidelines on how to develop dynamic libraries that aim to be used as *plugins* in the *REST server*.

Sample *plugins* for testing purposes are supplied on the enclosed CD. They are described in appendix E.

5.5 Client Applications

Client application is any computer application that connects to the COMPASS API, and uses it to obtain or manipulate data through operations defined by individual *plugins*, and thus acts as a *client* component within the COMPASS API system.

Development of *client* applications is outside of the scope of this master thesis. Section 7.2 however provides guidelines on how to develop or modify applications with the intent to utilize COMPASS API during their operation.

Sample *client* application for testing purposes is supplied on the enclosed CD. It is described in appendix E.

Chapter 6

Software Testing

This chapter deals with testing of the COMPASS API *REST server* component. The testing is separated into several areas. First, the coverage of requirements that were formulated in section 3.3 is discussed. Second, unit tests that were performed to verify the functionality of individual classes are described. Next, the testing of integration of developed components is described. Finally, used static and dynamic analysis tools are outlined.

6.1 Coverage of Requirements

This section discusses every individual requirement that was stated in section 3.3, and based on functionality of COMPASS API *REST server*, it is asserted whether this requirement was met.

List of requirements is present in the table below. For each requirement, this table lists the class, and if available the method as well, which is responsible for satisfying the requirement, or marks that the requirement was not satisfied. Several of the requirements are also discussed individually further in this section.

requirement	coverage
REQ-03	`RestServer`
REQ-04	`RestServer`
REQ-06	`DatabaseConnectionsManager::open`
REQ-09	`DatabasePlugin_CompassLogbook`
REQ-10	`DatabasePlugin_CompassLogbook`
REQ-11	`DatabasePlugin_CompassLogbook`
REQ-13	`QProcess::startDetached`
REQ-15	`RequestPluginWrapper::processPost`
REQ-16	`RequestPluginWrapper::processGet`
REQ-17	`RequestPluginWrapper::processPut`
REQ-18	`RequestPluginWrapper::processDelete`
REQ-19	`Socket::~Socket`

requirement	coverage
REQ-20	Responsibility of individual plugins
REQ-22	`RequestPluginManager::loadPlugins`
REQ-23	`RestServer`
REQ-24	`QPluginLoader`
REQ-25	`DatabaseConnectionsManager::getDatabaseQueryObject`
REQ-26	`DatabaseAccessInterface`
REQ-27	not satisified
REQ-28	not satisified
REQ-29	not satisified
REQ-30	not satisified
REQ-31	not satisified
REQ-32	not satisified
REQ-33	satisified
REQ-34	satisified
REQ-35	satisified
REQ-36	satisified
REQ-37	satisified
REQ-38	satisified
REQ-39	satisified
REQ-40	satisified
REQ-41	N/A

- **REQ-04:** `RestServer` class is capable of accepting requests from *clients* that are located outside of the COMPASS network. This requirement is satisfied, provided that the network configuration allows outside computers to access the COMPASS network.
- **REQ-13:** Plugins may use *Qt* framework's class `QProcess` and it's static method `QProcess::startDetached` to use the `scp` command.
- **REQ-15 to REQ-18:** `RequestPluginWrapper` class allows to process each type of *HTTP* method by calling an appropriate method from given request *plugin*.
- **REQ-19:** The destructor of the `Socket` class checks whether the *plugin* responsible for processing the request sent a response. If that is not the case, the destructor itself sends a generic response.
- **REQ-23:** `RestServer` class checks upon arrival of new request if it has a plugin for given endpoint at its disposal. If not, it automatically responds with an *HTTP* 404 error.
- **REQ-27 and REQ-28:** In its current version, the *REST server* does not provide a mediating class for access to files. Individual plugins are however free to execute file operations in local or remote file systems on their own. This functionality is planned to be added later.
- **REQ-29 to REQ-32:** The control interface and related functions were deliberately omitted for the provided release of COMPASS API *REST server*. The reason for this decision is that it was assessed that the best course of action is to utilize one of the existing control interfaces already used on COMPASS. The control of COMPASS API

may thus be integrated for example with COMPASS CLI, which is described in [10]. The development of such feature however necessitates an involvement of other members of COMPASS personnel.

6.2 Unit Testing

COMPASS API *REST server* was subjected to several unit tests. These tests were designed to verify the functionality of individual classes.

The description of unit tests that were conducted is provided in the list below.

- **tested method:** `RestServer::registerPlugin`

input	result	passed
`RequestPluginWrapper` containing `RequestPlugin_Runs`	`m_registeredEntpoints` container size increase by 1	yes
`RequestPluginWrapper` containing `RequestPlugin_RunTypes`	`m_registeredEntpoints` container size increase by 1	yes
`RequestPluginWrapper` containing `RequestPlugin_RunProblems`	`m_registeredEntpoints` container size increase by 1	yes

notes: The plugins used during this unit test are the plugins developed for demonstration of modified *run_ manager*, which is described in appendix E.

- **tested method:** `RequestPluginManager::loadPlugins`

input	result	passed
placing 0 files in plugin directory	`m_pluginTable` container size increase by 0	yes
placing 1 valid plugin in plugin directory	`m_pluginTable` container size increase by 1	yes
placing 1 invalid plugin in plugin directory	`m_pluginTable` container size increase by 0	yes
placing 1 valid plugin and 1 invalid plugin in plugin directory	`m_pluginTable` container size increase by 1	yes
placing 2 valid plugins in plugin directory	`m_pluginTable` container size increase by 2	yes

notes: The plugins used during this unit test are the plugins developed for demonstration of modified *run_ manager*, which is described in appendix E.

Invalid plugin in this context may mean both plugins which implement a wrong interface, or a completely random file.

- **tested method:** `DatabaseConnectionsManager::loadPlugins`

input	result	passed
placing 0 files in plugin directory	`m_pluginTable` container size not changed	yes
placing 1 valid plugin in plugin directory and database accessible	`m_pluginTable` container size increase by 1 and database connection opened	yes
placing 1 valid plugin in plugin directory and database not accessible	`m_pluginTable` container size increase by 1 and database connection not opened	yes
placing 1 invalid plugin in plugin directory	`m_pluginTable` container size not changed	yes

notes: The plugins used during this unit test are the plugins developed for demonstration of modified *run_ manager*, which is described in appendix E.

Invalid plugin in this context may mean both plugins which implement a wrong interface, or a completely random file.

- **tested method:** `Socket::~Socket`

input	result	passed
setting no response	socket sent response with code 500 automatically	yes
setting a simple response	custom response sent	yes
setting a JSON response	response containing a JSON file sent	yes
setting a response with custom body	response containing the custom data sent	yes

notes: The purpose of this test was to determine, if the `Socket` class automatically sends a response back to the *client*, if one was not set by the *plugin*.

6.3 Integration Testing

Integration testing of the COMPASS API *REST server* must necessarily include at least several additional request and database *plugins*. Therefore it depends at least to some degree on these external components.

Integration tests of correct interoperability of individual components were performed using the modified version of *run_ manager* application, which is described in appendix E. Various scenarios, such as attempts to load different runs from the database, and updates to the structure of loaded runs were attempted.

During this process, no errors were found in the *REST server application*. It correctly routed accepted requests to appropriate plugin methods, and handled any potential errors.

6.4 Static and Dynamic Analysis

For the purpose of testing the application, several third-party tools that perform an analysis of the code used.

Static analysis of COMPASS API consists of using the Clang Static Analyzer. This tool reads the source code of the application directly, and based on it, it tries to finds potential mistakes. It generally attempts to detect more complex errors than traditional compilers, or errors that do not directly violate the *C++* language standard, and are thus ignored by compilers. COMPASS API *REST server* source code passed this test without any errors or warnings.

During dynamic analysis of the code, the *valgrind* memory usage analyzer was used to determine, if any memory leaks occur during the operation of the *REST server*. The program was analyzed while loading several plugins, and subsequently processing a number of *HTTP* requests. Such operational conditions did not result in detection of any memory management problems.

Chapter 7

COMPASS API User's Guide

This section serves as a guide for developers who intend to develop *plugins* which will be used to extend the functionality of COMPASS API server, or develop applications that use the COMPASS API to work with data sources available at the COMPASS experiment.

Guidelines for developing COMPASS API *plugins* are presented in section 7.1, and guidelines for developing *client* applications are presented in section 7.2.

7.1 Plugin Developer's Guide

COMPASS API uses two types of *plugins*. The first type implements the logic required for processing of certain kind of requests sent to COMPASS API server, and the second type defines necessary parameters of database connections. The basic principles of implementing either of these types of *plugins* are the same, and are described in section 7.1.1. Aspects specific to implementation of request *plugins* and database *plugins* are then described in section 7.1.2 and section 7.1.3 respectively.

7.1.1 General Implementation Procedure of COMPASS API Plugins

Plugins for the COMPASS API server are developed as *Qt plugins*, which are described in section 4.2.2. *Plugins* for COMPASS API must therefore be implemented as *C++* dynamic libraries. Additionally, since the libraries will not be loaded by the application at linktime, but at runtime, they must be configured in a specific way.

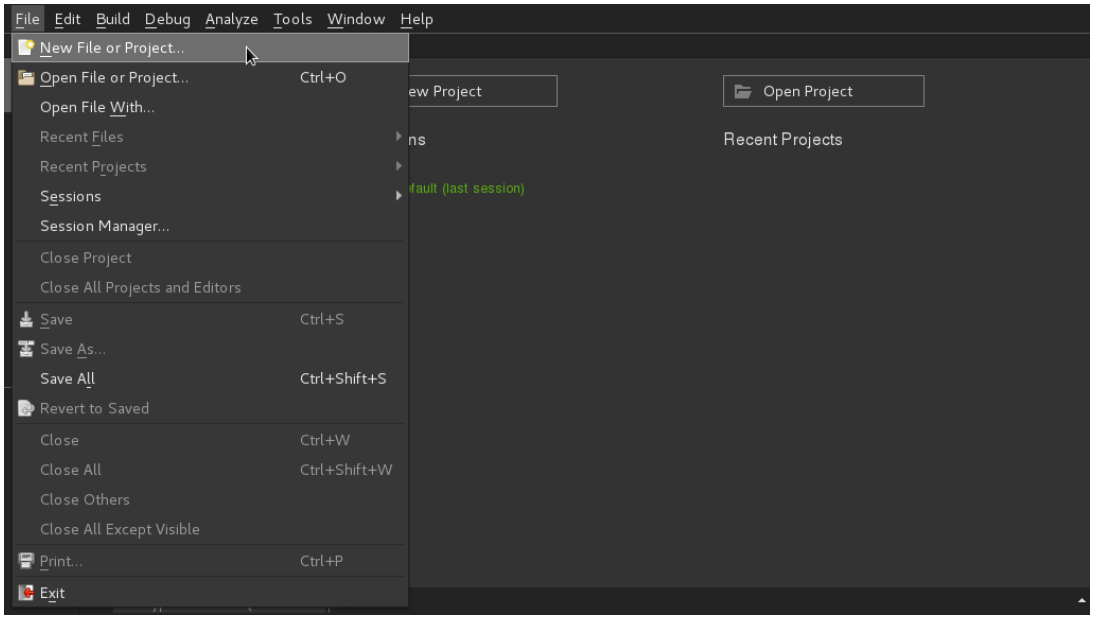
Since the configuration of the library is handled by the *Qt* framework, standard way of creating *Qt plugins* is by setting up a new *qmake*-based project for *Qt* with appropriate settings. It is possible to write the project file manually, or to use the project creation wizard in the *Qt Creator* IDE, which is described in section 4.2.3. This wizard will create a project with all required settings already in place. Both methods are described in this section

Additionally, this section contains general plugin implementation guidelines that are applicable to all COMPASS API *plugins*.

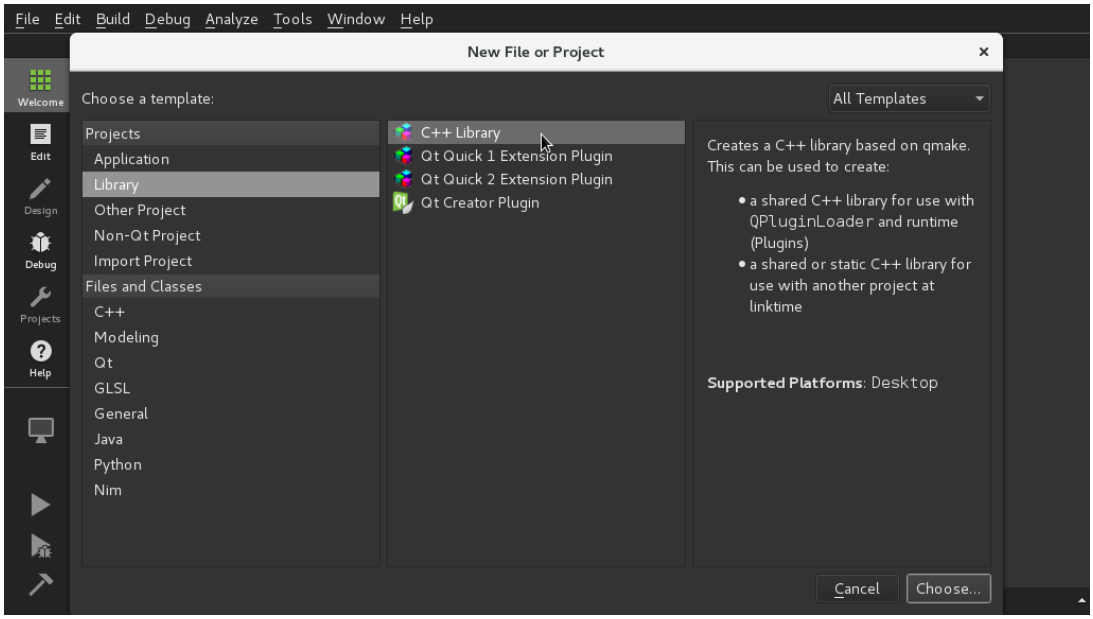
Using Qt Creator to Setup Plugin Project

If the developer chooses to use the project creation wizard in *Qt Creator* IDE, the following steps should be taken:

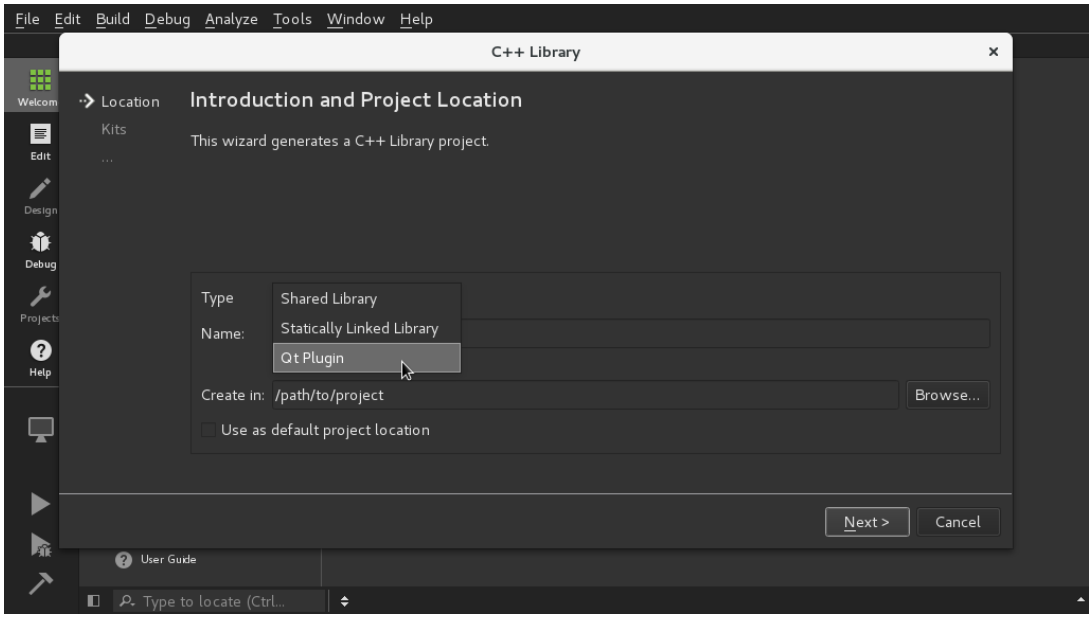
1. Open a new project in the File menu of *Qt Creator* IDE.



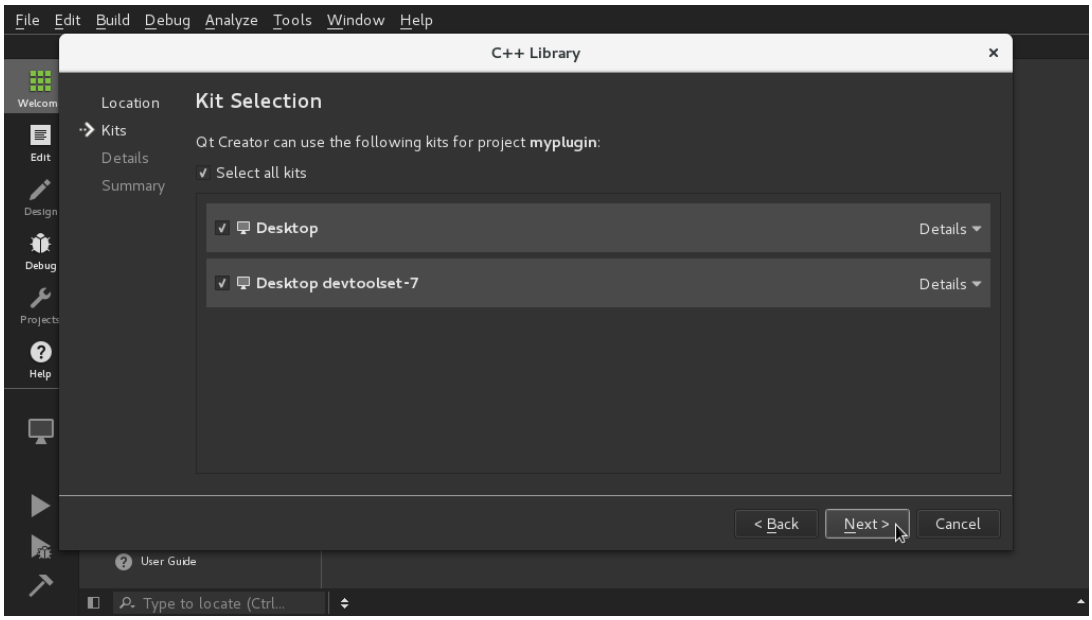
2. Select the project template Library → *C++* Library.



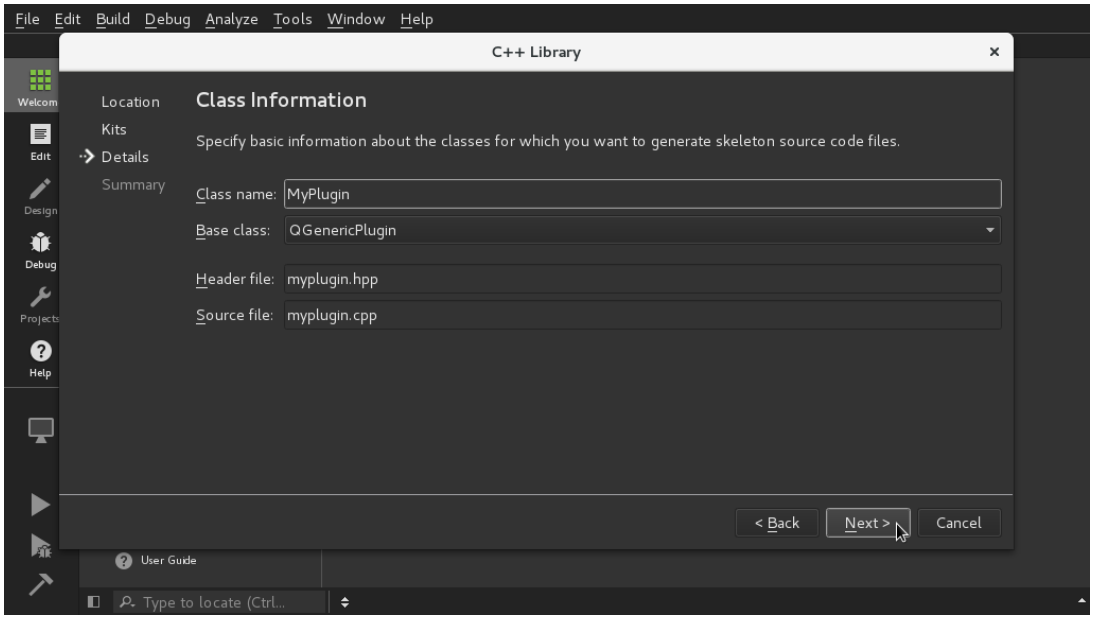
3. In the next step of the wizard, select “Qt Plugin” as the type of the library, and select a name and location of the project.



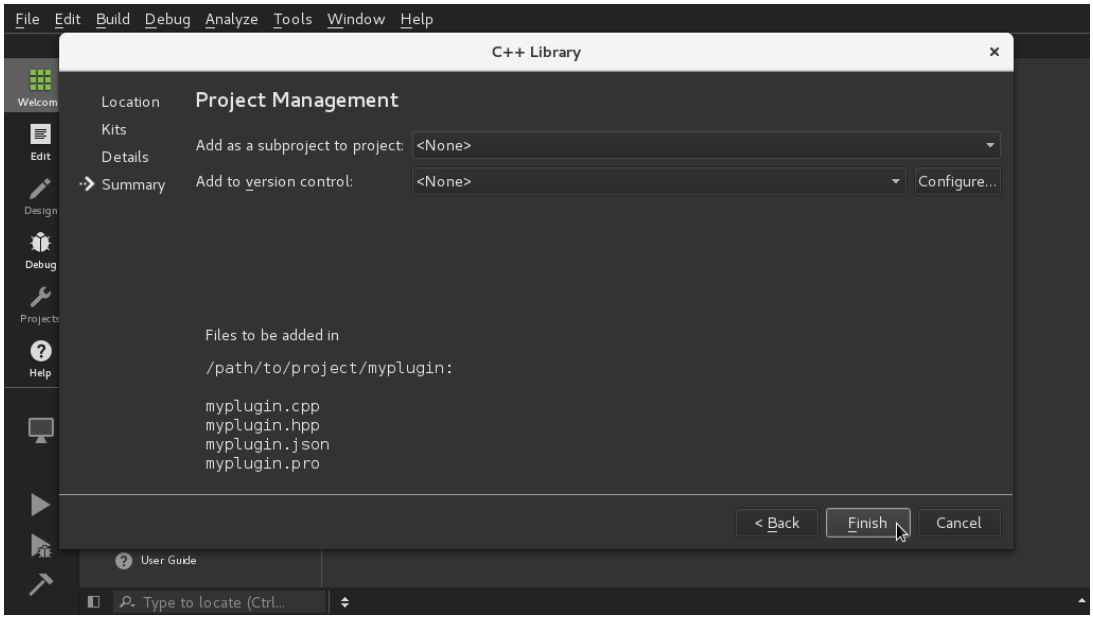
4. Select one of your predefined build kits (the selection available in this window will vary depending on user’s configuration of build kits).



5. Select the name of the primary class in your plugin. In addition to that, a base class for the *plugin* must be selected. However, since COMPASS API *plugins* use custom base class that is not available for selection, arbitrary base class must be selected and then deleted from the project.



6. Optionally configure a VCS to use with the project.



7. Remove the abstract base class that was selected in step 5 by removing it as the base class of your *plugin*, and removing the `#include` directive for the base class. Optionally, it is also possible to remove all preprocessor conditionals that check what version of *Qt* is used for compilation. COMPASS API is designed to compile with *Qt* versions stated in section 4.2.5, therefore these conditionals are not necessary, and *plugins* may safely assume that *Qt* version greater than 5.0 will be used.

8. *Plugin* is now ready to be implemented. Description of the implementation process for specific COMPASS API *plugins* is contained in sections 7.1.2 and 7.1.3.

This process applies to the version of *Qt Creator* that is described in section 4.2.5.

Using the *Qt Creator* IDE for creating the *plugin* project should be considered as the preferred method, since creating the project manually offers no benefits but is more time consuming.

Setting Up Plugin Project Manually

To setup the project file for building a *Qt plugin* manually, a new *qmake* project file must be created. Two configuration options in the project file must be properly set:

1. The project must use the `lib` template. Therefore, the line `TEMPLATE = lib` must be present in the project file, and the value of the project's `TEMPLATE` must not be changed afterwards.

2. The `CONFIG` variable of the project must contain the value `plugin`. This may be achieved using the line `CONFIG += plugin` inside the project file.

Other parts of the project must be properly configured to handle its specific structure. Detailed description of writing *qmake* project files is present in [31].

General Plugin Implementation Steps

After correctly setting up the project for building *Qt plugins*, it is possible to implement the *plugin*'s logic. Developing the source code of the *plugin* consists of four main steps [19]:

1. Implementing a class which inherits from the one of the plugin interfaces from component *plugin interfaces*, which are described in section 5.2.

2. Using the `Q_INTERFACES` macro to notify the *Qt* meta-object system about what interfaces are used by the *plugin*.

3. Exporting the meta data of the *plugin* to *Qt* meta-object system by using the macro `Q_PLUGIN_METADATA`. Note that in COMPASS API, the metadata must contain a key `interfaces_version`, whose value must be set to a string containing the *Git* version of the *plugin_ interfaces Git* repository that is described in appendix D. The version of the *Git* repository may be obtained using the command `git describe --always`. *Plugins*, which don't have this key in their metadata, or use a different version of *plugin interfaces* will be rejected. This restriction was imposed as a security measure, since

using a different version of *plugin interfaces* might result in a crash of the COMPASS API server.

4. Building the *plugin* using appropriate project file.

In ideal case, all *plugins* will be built using the same *Qt* framework version as the one that was used to build the *REST server*. If that is not possible however, it should be noted that *Qt* framework itself places limitations on used *Qt* version when building *plugins*, which are outlined in [19]. Therefore, the used *Qt* version must meet the following conditions:

1. *plugin* must not be built using *Qt* with higher version number than the one used for building the *REST server*.

2. *plugin* must not be built using *Qt* with lower major version number than the one used for building the *REST server*.

Information about *Qt* framework version that was used to build the *REST server* is included in section 4.2.5.

Furthermore, in order to avoid conflicts stemming from ABI incompatibility, the *plugins* are required to be built using the same compiler in the same version as the one that was used for building the *REST server*. Information about the *C++* compiler that was used to build the *REST server* is included in section 4.1.5.

7.1.2 Request Plugins Implementation

In addition to the general guidelines for *plugin* implementation, which were outlined in section 7.1.1, request *plugins* need to implement the `RequestPluginInterface` that is described in section 5.2.7.

The main class in every request *plugin* shall therefore inherit from `RequestPluginInterface` class, and implement all of the interface's pure virtual methods. These methods are as follows.

- `PluginInterface::identifier` shall return the *plugin*'s identifier in a form of a `QString`. This identifier is used to identify the *plugin* in several classes of the *REST server* component, and must therefore be unique.

- `RequestPluginInterface::endpoint` is used by the *REST server* component to determine what endpoint should be used to access the service provided by the *plugin*. This method shall return the name of the endpoint as a `QString`.

- `RequestPluginInterface::processPost` is called by the *REST server* whenever a valid *HTTP POST* request is received by that class at the endpoint specified by method `RequestPluginInterface::endpoint`. This method shall therefore implement the functionality for processing these types of requests.

- `RequestPluginInterface::processGet` is called by the *REST server* whenever a valid *HTTP GET* request is received by that class at the endpoint specified by method `RequestPluginInterface::endpoint`. This method shall therefore implement the functionality for processing these types of requests.

- `RequestPluginInterface::processPut` is called by the *REST server* whenever a valid *HTTP PUT* request is received by that class at the endpoint specified by method `RequestPluginInterface::endpoint`. This method shall therefore implement the functionality for processing these types of requests.

- `RequestPluginInterface::processDelete` is called by the *REST server* whenever a valid *HTTP DELETE* request is received by that class at the endpoint specified by method `RequestPluginInterface::endpoint`. This method shall therefore implement the functionality for processing these types of requests.

The core part of the logic that a `RequestPluginInterface`-derived *plugin* implements consists of the functionality in methods `processPost`, `processGet`, `processPut` and `processDelete`. These four methods correspond to the *REST server* architectural style that is described in section 4.3. While there is no strict requirement that would enforce any conditions on implementation of these methods, it is strongly advised that they follow the *REST* principles outlined in that section, and especially the *HTTP* method mapping displayed in table 4.1.

If the request *plugin* intends to access a *MySQL* database, which most prominently includes the COMPASS database that is described in requirement information INFO-07, additional steps must be taken during implementation. In addition to `RequestPluginInterface`, the developed *plugin* needs to inherit from the `DatabaseAccessPlugin` class, which is described in section 5.2.2. This class provides a protected method `DatabaseAccessPlugin::databaseQuery`, which accepts a `QString` identifier of a database *plugin*, and returns an object of type `QSharedPointer<QSqlQuery>` for the database defined by the given database *plugin*. The implemented request *plugin* may use this method to execute *SQL* queries in the database server. Description of how is the `QSharedPointer<QSqlQuery>` object obtained may be found in section 5.3.10. Note that this method may throw exception of type `std::domain_error`, if a database *plugin* with given identifier was not found. Additionally, exception of type `std::bad_function_call` may be thrown if the plugin was not set up correctly. This would however imply a logic error in the *REST server*, and is thus highly unlikely if the application was well tested.

7.1.3 Database Plugins Implementation

Database *plugins* shall follow all guidelines described in section 7.1.1. Additionally they need to implement the `DatabaseHandlerInterface` that is described in section 5.2.3.

Every *plugin*'s primary class shall therefore inherit from `DatabaseHandlerInterface` class, and implement all of its pure virtual methods. These methods are as follows.

- `PluginInterface::identifier` shall return the *plugin*'s identifier in a form of a `QString`. This identifier is used to identify the *plugin* in several classes of the *REST server* component, and must therefore be unique.

- `DatabaseHandlerInterface::getDatabaseName` shall return the name of the database stored inside the database server targeted by the *plugin*. The database name shall be returned in a form of a `QString`.

- `DatabaseHandlerInterface::getDriverName` shall return a `QString` containing the name

of the *Qt SQL* database driver plugin that will be used to connect to the database server targeted by the *plugin*. *Qt SQL* component is described in section 4.2.2.

- `DatabaseHandlerInterface::getHostName` shall return the hostname of the computer that hosts the database server targeted by the *plugin* as a `QString`.
- `DatabaseHandlerInterface::getPassword` shall return a `QString` that represents the password that will be used to authenticate on the database server targeted by the *plugin*.
- `DatabaseHandlerInterface::getUserName` shall return the username that will be used to authenticate on the database server targeted by the *plugin*. The username is returned as a `QString` class.

The implementation of all pure virtual methods in a `DatabaseHandlerInterface`-derived *plugin* is under normal conditions trivial. Apart from `PluginInterface::identifier`, all of the methods return a `QString` that contains a part of the information necessary to connect to a database. These data are subsequently used by `DatabaseConnectionsManager` class, which is described in section 5.3.6, to open the actual database connection.

7.2 Client Applications Developer's Guide

Client applications are required to communicate with the COMPASS API by using the *TCP* protocol. Generally, it is necessary to open a *TCP* connection to the computer hosting the COMPASS API using an *URI*, that should contain at least:

- used protocol, which in case of COMPASS API will always be *HTTP* or *HTTPS*
- hostname or IP address of the computer hosting COMPASS API server
- port on which the COMPASS API server listens (this parameter is decided by COMPASS API's configuration)
- path, which shall be equal to the requested service's endpoint

and optionally also

- optional query list of key-value pairs, if any are required

Example of a possible *URI* may be the *URI* `http://localhost:12345/test`. Using this *URI*, the *client* application would access a COMPASS server running on the local operating system on port 12345, and access the *plugin* that provides a service called `test`.

After the *TCP* connection is established, the application must send an *HTTP* request to the server, to which the server will always send a response. The response will be an *HTTP* response, that primarily contains the *HTTP* status code, and if required, service specific data in the response's body. The developer's of the *plugins* are encouraged to use the *JSON* format for transmitting the data, however, the specific format of the response's body is *plugin*-dependent.

Since there is no requirement placed on the programming languages that shall be used to implement the *client* applications, it is not possible to define a universal procedure which would

interface with the services provided by the server that was developed as part of this thesis. However, since it is the predominant programming language currently used on the COMPASS experiment, an example is provided in listing 7.1 that shows the construction of a simple request in a *C++* application that uses the *Qt* framework.

```
1  QNetworkAccessManager manager;
2
3  QNetworkRequest request;
4  request.setUrl(QUrl("http://pccompassapi:12345/comment?id=42"));
5
6  QNetworkReply* reply = manager->get(request);
7  connect(reply, &QNetworkReply::readyRead, this, &MyClass::slotReadyRead);
8  connect(reply, &QNetworkReply::error, this, &MyClass::slotError);
```

Listing 7.1: sending *GET* request using *Qt*

The piece of code in listing 7.1 will send a *GET* request to *URI* `http://pccompassapi:12345/comment?id=42`. The stream associated with the underlying *TCP* socket will be processed in the (not shown) *Qt* slot `slotReadyRead`. Error handling will be done in the (also not shown) *Qt* slot `slotError`.

When analyzing the code line-by-line, we shall first focus on the `QNetworkAccessManager` class. This class, which is part of *Qt Network* module that is described in section 4.2.2, serves to send network requests and process the replies to these requests. The request in the code is represented by `QNetworkRequest` class. To send a simple request, it is necessary to set at least the *URI*, which is represented by the `QUrl` class. The `QNetworkAccessManager` may then be instructed to use the instance of the `QNetworkRequest` class to send a *GET* request by calling the method `QNetworkAccessManager::get`. This method returns an instance of `QNetworkReply` class, which may then be used to access the data through *Qt*'s signals/slots mechanism.

Depending on the implementation of endpoint "comment" in corresponding COMPASS API, the code in listing 7.1 may for example obtain comment with `id` equal to 42 (see [5] for description of comments system that is used on the COMPASS experiment). Such code is significantly shorter than source code with similar function that uses direct access to database, for instance the database access routines used in applications described in [5].

Conclusion

This master thesis was focused on designing an application programming interfaces which would provide unified access to data storage systems that are used on the COMPASS experiment at the CERN laboratory.

The first part of the task consisted of conducting an analysis of the problematics. It focused mainly on structure and characteristics of the storage solutions, specifically the database and file folders shared over a network. Based on the findings, a set of requirements, which were placed on the proposed COMPASS API was outlined. These were subsequently transformed into a preliminary design of an application that would fulfill these requirements.

The design process continued with a selection of suitable technologies. Since the COMPASS API needs to be intuitive and easy to use, this step played principal role in the development process, because technologies which allow to introduce an appropriate form of abstraction will add to the usability of the API considerably.

The development process was subsequently conducted, and resulted in an introduction of a COMPASS API software, which comes in a form of a *REST* server, and accomplish the tasks set forth by the requirements.

The developed COMPASS API possesses all of the major properties, which are primarily extensibility and indifference to the programming languages used to implement its client applications. Nevertheless, not all of the requirements first outlined were satisfied. Nevertheless, these requirements describe functionality, which can be provided without much effort at a later time given the applications structure, and core functionality of the program is already prepared.

The next step for the COMPASS API in its lifecycle is the deployment into production environment on the COMPASS experiment. However, due to the impact that the COMPASS API may potentially have on the whole computer system, this task is best deferred until there is no active data taking performed for significant period on COMPASS. This would allow to properly test and verify that no key computer system on COMPASS is affected in a negative manner. A suitable opportunity to complete this procedure may be offered by the planned CERN LS2 phase.

Subsequently, the functionality of the COMPASS API may be extended with the plugins that were described several times throughout this thesis.

Bibliography

[1] *CERN*. URL: https://home.cern/ (visited on July 24, 2016).

[2] *COMPASS*. URL: https://wwwcompass.cern.ch/ (visited on July 24, 2016).

[3] J. Nový. "Processing of large quantity of data from the COMPASS experiment". written dissertation preparation. Czech Technical University in Prague, Faculty of Nuclear Sciences and Physical Engineering, 2016.

[4] *CERN. CASTOR*. URL: http://castor.web.cern.ch/ (visited on May 6, 2018).

[5] M. Jandek. "User interface for control of logbook of COMPASS experiment at CERN". Bachelor's Degree Project. Czech Technical University in Prague, Faculty of Nuclear Sciences and Physical Engineering, 2016.

[6] M. Jandek. "COMPASS electronic logbook data flow and storage optimization". Research Project. Czech Technical University in Prague, Faculty of Nuclear Sciences and Physical Engineering, 2017.

[7] V. Jarý. "Analysis and proposal of the new architecture of the selected parts of the software support of the COMPASS experiment". Doctoral dissertation. Czech Technical University in Prague, Faculty of Nuclear Sciences and Physical Engineering, 2012.

[8] M. Bodlak et al. "Development of new data acquisition system for COMPASS experiment". In: *Nucl. Part. Phys. Proc.* 273-275 (2016), pp. 976–981.

[9] J. Hrušovský. "Deployment application for the data acquisition system of the COMPASS experiment at CERN". Bachelor's Degree Project. Czech Technical University in Prague, Faculty of Nuclear Sciences and Physical Engineering, 2017.

[10] A. Květoň. "Remote user interface for the control system of the COMPASS experiment at CERN". Master Thesis. Czech Technical University in Prague, Faculty of Electrical Engineering, 2017.

[11] M. Zemko. "Data flow analysis for the data acquisition system of the COMPASS experiment at CERN". Bachelor's Degree Project. Czech Technical University in Prague, Faculty of Nuclear Sciences and Physical Engineering, 2017.

[12] L. Bátrla. "COMPASS experiment shift reservation system". Bachelor's Degree Project. Czech Technical University in Prague, Faculty of Nuclear Sciences and Physical Engineering, 2017.

[13] T. Brabec. "Electronic checklist for COMPASS experiment at CERN". Bachelor's Degree Project. Czech Technical University in Prague, Faculty of Nuclear Sciences and Physical Engineering, 2017.

[14] *TIOBE*. URL: https://www.tiobe.com/ (visited on February 16, 2018).

[15] *Standard C++*. URL: https://isocpp.org/ (visited on February 16, 2018).

[16] B. Stroustrup. *The C++ Programming Language*. 4th edition. Boston, MA, USA: Addison-Wesley, 2013. ISBN: 978-0-321-56384-2.

[17] N. M. Josuttis. *The C++ Standard Library. A Tutorial and Reference*. 2nd edition. Boston, MA, USA: Addison-Wesley, 2012. ISBN: 978-0-321-62321-8.

[18] *Qt Documentation*. URL: http://doc.qt.io/ (visited on February 24, 2018).

[19] *Qt Documentation. How to Create Qt Plugins*. URL: http://doc.qt.io/qt-5/plugins-howto.html (visited on April 14, 2018).

[20] R. T. Fielding. "REST: Architectural Styles and the Design of Network-based Software Architectures". Doctoral dissertation. University of California, Irvine, 2000.

[21] *GNU Operating System. GNU Libmicrohttpd*. URL: https://www.gnu.org/software/libmicrohttpd/ (visited on March 30, 2018).

[22] *GitHub. libhttpserver*. URL: https://github.com/etr/libhttpserve/r (visited on March 30, 2018).

[23] *GitHub. Libqmhd*. URL: https://github.com/francoiscolas/libqmhd/ (visited on March 30, 2018).

[24] *GitHub. QttpServer*. URL: https://github.com/supamii/QttpServer/ (visited on March 30, 2018).

[25] *GitHub. QHTTPEngine*. URL: https://github.com/nitroshare/qhttpengine/ (visited on March 29, 2018).

[26] *Open Source Initiative. The MIT License*. URL: https://opensource.org/licenses/MIT/ (visited on March 29, 2018).

[27] *Ecma International. The JSON Data Interchange Syntax*. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf (visited on May 5, 2018).

[28] *Internet Engineering Task Force. The JavaScript Object Notation (JSON) Data Interchange Format*. URL: https://tools.ietf.org/html/rfc8259/ (visited on May 5, 2018).

[29] *JSON. Introducing JSON*. URL: https://www.json.org/ (visited on May 4, 2018).

[30] *Qt Documentation. QSettings Class*. URL: http://doc.qt.io/qt-5/qsettings.html (visited on May 3, 2018).

[31] *Qt Documentation. qmake Manual*. URL: http://doc.qt.io/qt-5/qmake-manual.html (visited on April 25, 2018).

[32] *CERN. Linux@CERN*. April 4, 2017. URL: https://linux.web.cern.ch/linux/centos7/ (visited on April 9, 2018).

Appendix A

Contents of the Enclosed CD

This appendix lists all documents that are stored on the CD included with this master thesis.

- The electronic version of this document.
- Source codes for the COMPASS API *REST server* and *plugin interfaces* components.
- Source code of a modified version of the *run_ manager* applications, which was developed as part of [5]. The included version was adapted to use the COMPASS API to obtain all required data from the database.
- Subset of logbook database in *SQL* format. This testing database may be used for testing of COMPASS API and the included *run_ manager* application in local environment.

Appendix B

Installation Instructions for COMPASS API

This appendix contains the instructions for compiling and installing the COMPASS API server.

B.1 Operating System Note

COMPASS API was designed, in accordance with requirement REQ-33 that was stated in section 3.3.2, to run on the CERN CentOS 7 operating system. Only the most recent version of the operating system shall be used, with all available updates for installed packages applied as well.

Installation files for CERN CentOS 7 operating system may be obtained from [32]. The linked webpage also contains instructions on how to install the operating system. Note that the installation process requires an internet connection, as the most of the components of the OS are downloaded from remote repository during the installation.

Other operating systems, including different Linux distributions, may be able to compile and run COMPASS API, however, no guarantee that such attempt will succeed is provided.

All further instructions will be written under the assumption that CERN CentOS 7 system, which was installed in the "Software development workstation" configuration and with the option "'additional development headers", is being used to compile COMPASS API and run the resulting binary.

B.2 Installation of Required Packages

Several packages are required to be installed in the operating system for successful compilation of COMPASS API. Table B.1 lists all packages that are required to be installed to compile COMPASS API, and provides a brief description of why are they needed.

package name	requried for
cmake3	compilation of *QHTTPEngine* library
devtoolset-7	compilation of the entire COMPASS API project
qt-creator	opening and compilation of COMPASS API *Qt* project
qt5-qtbase-devel	compilation of COMPASS API and *QHTTPEngine* library
qt5-qtbase-mysql	compilation of COMPASS API

Table B.1: Software packages required for compilation of COMPASS API

CERN CentOS 7 operating system uses the *yum* package manager, which may be used to install all packages mentioned in table B.1. It will also install all packages, which the listed packages require for correct functioning. Installation of the packages may be triggered by the following command:

```
yum install centos-release-scl
yum install cmake3
yum install devtoolset-7* --skip-broken
yum install qt-creator
yum install qt5-qtbase-devel
yum install qt5-qtbase-mysql
```

B.3 Configuration of Qt Creator

In order to compile the provided sources of COMPASS API, it is necessary to configure a new build kit in the *Qt Creator*.

The new kit shall posses the settings as the default kit, however, it is necessary to set the compiler to the gcc compiler which is provided by the *devtoolset-7* package. This compiler is under normal conditions located in path `/opt/rh/devtoolset-7/root/usr/bin/g++`.

Additionally, it is necessary to set the sysroot path in the newly configured kit to the path `/opt/rh/devtoolset-7/root`.

B.4 Compilation

It is now possible to open the project that manages the build of the application. The project's path relative to the COMPASS API's sources is `project/codiac.oro`.

Before building the project, it is also necessary to configure a local variable in the build environment in *Qt Creator*. The variable shall be named `CMAKE` and its value equal to the path of *cmake3* binary.

After this configuration step, it is possible to build the application.

B.5 Running

COMPASS API *REST server* may be run directly from the IDE. If the user wants to start the binary from standard environment, it is necessary to make the libraries *libqhttpengine.so.1* and *libcodiacplugins.so.MT_ VERSION* available to it, for example through environment variable `LD_LIBRARY_PATH`.

Additionally, it is necessary to place the configuration file, which may be found relative to the sources root in `rest_server/build_codiac_debug/codiac.conf`, into user's configuration file path in `~/.config/compass`. Subsequently, options in the config file may be changed. Configuration file options are described in appendix C.

Moreover, if it is required to use the test database, which is included on the enclosed CD, for example to test the *run_ manager* application, one must set the database according to parameters defined in database plugin `database_plugin_test`. These parameters are outlined in the plugins `.cpp file`. Next, the database must be filled using the *SQL* script provided on the CD.

Appendix C

REST Server Application Recognized Configuration File Keys

The functionality of COMPASS API *REST server*'s binary is modifiable to certain extent through a configuration file in an *INI* format.

This appendix describes the options available through this configuration file. The options are listed in table C.1, and then described in detail.

group name	option name	allowed values
plugins	rest_plugin_dir	directory path
plugins	database_plugin_dir	directory path
logging	file	file, STDOUT, STDERR
logging	file_open_mode	APPEND, TRUNCATE
logging	max_log_level	see table 5.2
rest_server	port	port number
rest_server	listen_attempt_delay	number of milliseconds
database_connections	connection_attempt_delay	number of milliseconds
database_connections	max_connection_attempts	number

Table C.1: Recognized configuration keys

- **plugins.rest_plugin_dir**: Contains path to the directory that holds compiled request plugins, which are described in section 7.1.2.
- **plugins.database_plugin_dir**: Contains path to the directory that holds compiled database plugins, which are described in section 7.1.3.
- **logging.file**: Determines the target of the application's logging system. This value may either contain the string `STDOUT` for output to application's standard output, `STDERR` for output to application's standard error output, otherwise the string will be evaluated as a path to a log file. This path may be either relative or absolute.
- **logging.file_open_mode**: This options holds a string, which may equal to either `TRUNCATE` or `APPEND`. If the option *logging.file* is evaluated as a file path, this option

decides, if the file contents shall be truncated, or appended with the log for current application session.

- **logging.max_log_level**: Determines the lowest allowed severity level of log messages that will be written to log target. Messages with severity level lower than this setting will be ignored.
- **rest_server.port**: Contains a number of the port that will be used by the *REST server* to listen for incoming TCP connections. Value of 0 results in the port being chosen automatically.
- **rest_server.listen_attempt_delay**: Decides the delay between *REST server*'s unsuccessful attempt to start listening, and the next attempt that will be made. This value is evaluated as milliseconds.
- **database_connections.connection_attempt_delay**: Decides the delay between *REST server*'s unsuccessful attempt to open a database connection, and the next attempt that will be made. This value is evaluated as milliseconds.
- **database_connections.max_connection_attempts**: Controls the maximum allowed number of attempts to open a database connection, before the *REST server* gives up. Value of 0 means that there is no limit.

The application utilizes the *Qt* framework's `QSettings` class, and the location where is the configuration file loaded from is therefore guided by the rules of this class, which are described in [30]. The standard file path of the configuration file on the CERN CentOS 7 operating system is nevertheless `/.config/compass/codiac.conf`.

Note that if the application fails to load a configuration file, it holds default values for every setting. These default values are listed in table C.2.

option	default value
plugins.rest_plugin_dir	current directory
plugins.database_plugin_dir	current directory
logging.file	STDOUT
logging.file_open_mode	APPEND
logging.max_log_level	INFO
rest_server.port	0
rest_server.listen_attempt_delay	1000
database_connections.connection_attempt_delay	5000
database_connections.max_connection_attempts	0

Table C.2: Configuration keys default values

Appendix D

Git Repository Structure

This appendix contains a description of the *Git* repository that was used during the development of *REST server* and *plugin interfaces* COMPASS API components, which are described in sections 5.2 and 5.3 respectively.

The *Git* repository makes use of the submodule feature, which allows *Git* repositories to store a reference to a specific commit that is present in another *Git* repository. This results in an effective dependency management system, since it is possible to reuse the contents of a single repository in multiple projects, and even to allow these projects to use different versions of the contents of the shared *Git* repository. This in effect results in a tree-like structure of the repositories.

The repository is located on CERN's GitLab server, and is thus publicly unavailable. However, a snapshot of the repository is provided on the enclosed CD.

The structure of the *Git* repositories is illustrated using a UML component diagram on figure D.1.

All of COMPASS API *Git* repositories are encapsulated within the repository *codiac*[1]. This main repository holds the *rest_ server* repository, and the *database_ plugins* and *rest_ plugins* repositories.

The *rest_ server* repository holds the source code for the *REST server* component of the system. Since the *REST server* depends on the *plugin interfaces* component, this repository contains the *plugin_ interfaces* repository as a submodule. *REST server* additionally depends on the *QHTTPEngine* library. The GitHub repository for this library is however referenced by the *plugin_ interfaces* repository, and the sources are reused.

The *rest_ plugins* and *database_ plugins* repositories are intended to hold all repositories that contain *plugin* implementations as submodules. Currently, only the database plugin that provides access to COMPASS logbook database is officially provided.

plugin_ interfaces repository provides the interface classes that are described in section 5.2, which have to be implement by all *plugins*. As such it serves as a submodule in the *rest_ server*

[1] *codiac* is the working name of the COMPASS API. It is an acronym which stands for COmpass Data Interaction and Access Control

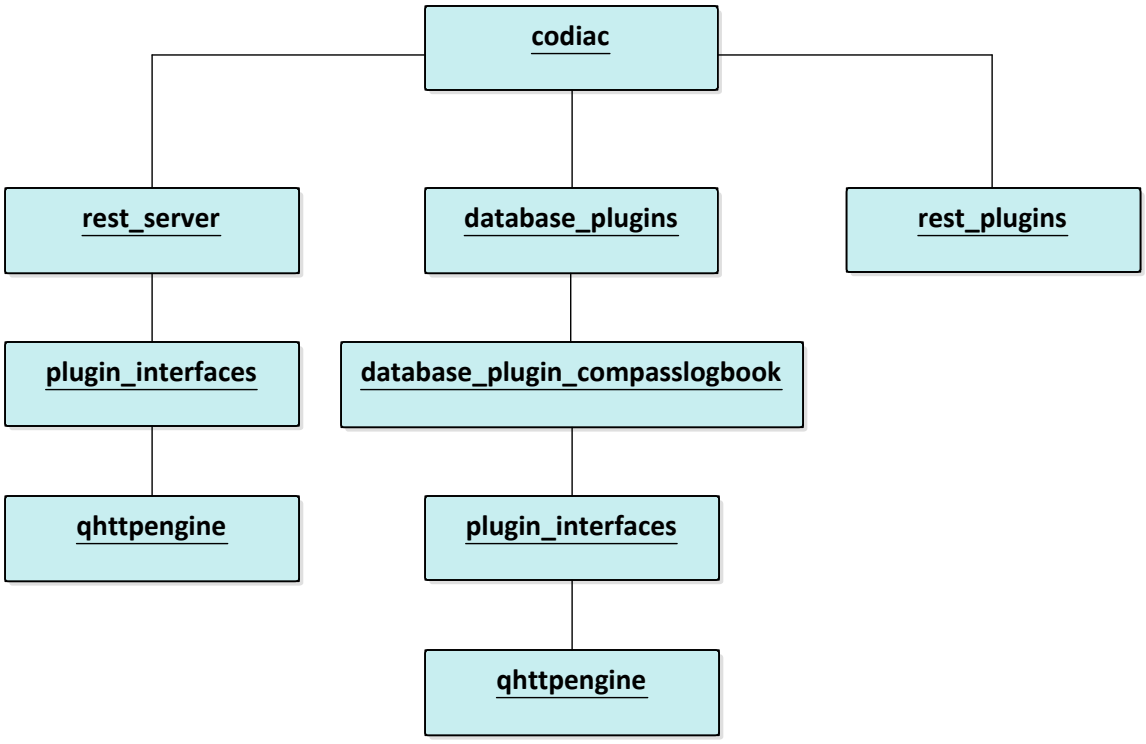


Figure D.1: UML component diagram representing *Git* repository structure

repository, and should be included as a submodule in all repositories which accommodate sources codes for *plugins* as well, because plugins depend on the *plugin interfaces* component.

This structure was introduced to maintain an invariant that for every repository, all of its dependencies are either included directly in the repository, or are contained as a submodule. For the future, it is planned that while the described structure will be maintained, however, during a complete build of the entire COMPASS API system one of the *plugin_ interfaces* repositories will be selected as primary by the build script and will be the only one to be compiled. This will prevent compilation of the same source code multiple times, which is currently the case.

Appendix E

Installation Instructions for Modified run_manager

This appendix contains the instructions for compiling, installing and testing the modified *run_ manager* application that is described in appendix A.

E.1 Installation of COMPASS API

Install the COMPASS API according to instructions described in appendix B.

E.2 Compilation

The compilation is expected to be conducted in the same environment as was described in appendix B.

To compile *run_ manager*, open its project file in *Qt Creator* IDE. The project file that is required to be opened is the file `run_manager.pro` which is located relative to *run_ manager*'s sources root directory in path `compass-rccars-logbookgui-run-manager`. It is possible to compiler it using the default build kit provided by the system, or the build kit that was configured in appendix B.3.

E.3 Running

The application may be ran either directly from the IDE or from normal environment. It is however required that an instance of COMPASS API *REST server* is already running, and that it is listening on port 41596.

Appendix F

Used Terminology and Abbreviations

This appendix contains a list of technical terms and abbreviations that were used throughout this master thesis.

abbreviation	meaning
ABI	application binary interface
API	application programming interface
Bash	Bourne again shell (linux command interpreter and programming language)
COMPASS	Common Muon and Proton Apparatus for Structure and Spectroscopy
CERN	European Organization for Nuclear Research
CRUD	create, read, update, delete (basic operations of persistent storage)
DAQ	data acquisition
DBMS	database management system
ECAL	electro-magnetic calorimeter
endpoint	Synonym for path specification in a URI, when used as a resource identifier in a RESTful API
GCC	GNU Compiler Collection
GEM	gaseous electron multiplier
GNU	GNU's Not Unix! (operating system and software tool collection)
HCAL	hadronic calorimeter
HTTP	hypertext transfer protocol
IDE	integrated development environment
IP	internet protocol
ISO	International Organization for Standardization
JSON	JavaScript Object Notation (data interchange format)
LS2	long shutdown 2 (CERN laboratory machine development period scheduled for years 2019-2020)
MicroMegas	micro mesh gas detector
moc	meta-object compiler (preprocessor from the Qt framework)

abbreviation	meaning
MWPC	multi-wire proportional chamber
OS	operating system
PHP	PHP: Hypertext Preprocessor (programming language)
Qbs	Qt Build Suite (Qt framework's build tool)
QML	Qt Modeling Language
Qt	Qt framework (C++ framework)
REST	representational state transfer (system architecture style)
RESTful API	API adhering to REST design principles
RICH	ring-imaging Cherenkov detector
SciFi	scintillating fiber detector
SCP	secure copy protocol(network file transfer protocol)
SPS	Super Proton Synchrotron (CERN laboratory accelerator)
SSL	secure sockets layer
TCP	transmission control protocol
TCP/IP	combination of TCP and IP protocols, usually referred to as internet protocol suite
UDP	user datagram protocol
UML	unified modeling language
URI	uniform resource identifier
URL	uniform resource locator
VCS	version control system
XML	extensible markup language