ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kováč** Jméno: **Jakub** Osobní číslo: **474548**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Software**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Návrh a prototypová implementace logovacího frameworku pro nástroj Manta

Název bakalářské práce anglicky:

Design and prototype implementation of a logging framework for Manta Flow

Pokyny pro vypracování:

Cílem práce je analyzovat, navrhnout a implementovat ve formě funkčního prototypu rozšiřitelný logovací framework pro nástroj Manta Flow.
Postupujte v následujících krocích:
1. Analyzujte aktuální stav logování v nástroji Manta Flow a na základě interview se zadavatelem specifikujte požadavky na nové řešení.
2. Vypracujte rešerši běžných logovacích frameworků v jazyce Java (Log4j, Jakarta, JavaUtilLogging, Slf4j).
3. Navrhněte strukturu logování - tj. kategorie běžných chyb, logovací hlášení, ...
4. Diskutujte a navrhněte softwarovou architekturu logování a zobrazování logů.
5. Návrh implementujte formou prototypu.

Seznam doporučené literatury:

CHUVAKIN, Anton, Kevin J. SCHMIDT, Chris PHILLIPS a Patricia MOULDER. Logging and log management: the authoritative guide to understanding the concepts surrounding logging and log management. Amsterdam: Elsevier/Syngress, [2013]. ISBN 1597496359.
NARKHEDE, Neha, Gwen SHAPIRA a Todd PALINO. Kafka: the definitive guide : real-time data and stream processing at scale. Sebastopol, CA: O'Reilly Media, 2017. ISBN 1491936169.
RICHARDS, Mark, Richard MONSON-HAEFEL a David A CHAPPELL. Java Message Service: Creating Distributed Enterprise Applications, CA: O'Reilly Media, [2009]. ISBN 9780596522049.
GUPTA, Ravi Kumar, Yuvraj GUPTA. Mastering Elastic Stack, Packt Publishing, [2017]. ISBN 9781786460011.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Michal Valenta, Ph.D., katedra softwarového inženýrství FIT

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **03.02.2020** Termín odevzdání bakalářské práce: **22.05.2020**

Platnost zadání bakalářské práce: **30.09.2021**

Ing. Michal Valenta, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

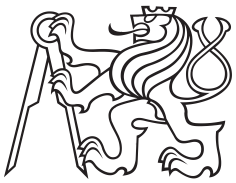
III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. _____ _____

Datum převzetí zadání Podpis studenta

Bachelor Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Design and prototype implementation of a logging framework for Manta Flow

Jakub Kováč

Supervisor: Ing. Michal Valenta, Ph.D.
May 2020

Acknowledgements

Rád bych poděkoval své rodině, která při mě během celého studia stála a jejíž podpora, trpělivost a motivace byly během studia klíčové.

Dále bych chtěl poděkovat Ing. Michalu Valentovi Ph.D. za vedení práce a rady při její tvorbě a všem z týmu Manty za podporu při tvorbě závěrečné práce a za to, že jsem se od nich mnohému naučil.

Declaration

I hereby declare that I have authored this thesis independently, and that all sources used are declared in accordance with the “Metodický pokyn o etické přípravě vysokoškolských závěrečných prací".

I acknowledge that my thesis (work) is subject to the rights and obligations arising from Act No. 121/2000 Coll., on Copyright and Rights Related to Copyright and on Amendments to Certain Laws (the Copyright Act), as amended, (hereinafter as the “Copyright Act"), in particular § 35, and § 60 of the Copyright Act governing the school work.

With respect to the computer programs that are part of my thesis (work) and with respect to all documentation related to the computer programs (“software"), in accordance with Article 2373 of the Act No. 89/2012 Coll., the Civil Code, I hereby grant a nonexclusive and irrevocable authorisation (license) to use this software, to any and all persons that wish to use the software. Such persons are entitled to use the software in any way without any limitations (including use for-profit purposes). This license is not limited in terms of time, location and quantity, is granted free of charge, and also covers the right to alter or modify the software, combine it with another work, and/or include the software in a collective work.

Prague, May 20, 2020

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., o právu au-

torském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů (dále jen „autorský zákon“), především § 35 a § 60 autorského zákona upravující školní dílo.

V případě počítačových programů, jež jsou součástí mojí práce či její přílohou, a veškeré související dokumentace k počítačovým programům (dále jen „software“), uděluji v souladu s ust. § 2373 zákona 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, nevýhradní a neodvolatelné oprávnění (licenci) k užití software, a to všem osobám, které si přejí software užít. Tyto osoby jsou oprávněny software užít jakýmkoli způsobem a za jakýmkoli účelem v neomezeném rozsahu (včetně užití k výdělečným účelům), vč. možnosti software upravit či měnit, spojit jej s jiným dílem a/nebo zařadit jej do díla souborného. Toto oprávnění je časově, teritoriálně i množstevně neomezené a uděluji jej bezúplatně.

V Praze, 20. května 2020

Abstract

This thesis describes design and prototypical implementation of a log management solution for the application ecosystem of a Czech company Manta. Manta is specialized in developing software that analyzes and produces data lineage of data warehouses and other data stores. The process of building data lineage for large enough data stores is a long process that naturally produces a large amount of logs. Due to this the log analysis becomes a very difficult process for people because the logs are stored in text files which can grow to a significant size for a human reader. This thesis therefore analyzes the issues with the previous solution and the requirements stemming from it. In the analytical part of the thesis are analyzed existing solutions for different aspects of log management - logging itself, transport to a centralized location and persisting the transported data in a way that simplifies the log analysis. The output of this thesis are prototypes of a logging API that enables developers to enhance logs with additional data that are stated in the requirements and a web application that facilitates access and other operations with logs for the end user.

Keywords: logging, log analysis, Manta, data lineage, Java, Spring, ActiveMQ, Log4j 2, source code generation

Supervisor: Ing. Michal Valenta, Ph.D.
Fakulta informačních technologií ČVUT,
Thákurova 2077/7,
160 00 Praha 6

Abstrakt

Tato bakalářská práce se zabývá návrhem a prototypovou implementací systému pro zpracovávání logů, které jsou vygenerovány aplikacemi české firmy Manta. Manta vyvýjí software pro analýzu a tvorbu data lineage datových skladů a jiných úložišť dat. Tvorba data lineage pro dostatečně velká úložiště dat je dlouhý proces, během kterého je vygenerováno velké množství logů. Logy jsou uloženy v textových souborech, které mohou narůst do velikostí pro člověka nezpracovatelných. Analýza logů a řešení problémů, které popisují, se proto stává více a více náročná. Tato práce analyzuje problémy s aktuálním řešením logování v Mantě a požadavky na nový systém z toho pramenící. V analytické části práce je rešerše existujících řešení všech aspektů log managementu - samotné logování, přenos logů do centralizovaného úložiště a ukládaní přenesených logů takovým způsobem, aby zefektivnil jejich analýzu. Výstupem této práce jsou prototypové implementace logovacího API, které umožní vývojářům jednoduše obohatit logy o další důležité informace, a webové aplikace umožňující vygenerované logy prohlížet, filtrovat a dále s nimi manipulovat.

Klíčová slova: logování, analýza logů, Manta, data lineage, Java, Spring, ActiveMQ, Log4j 2, generování zdrojového kódu

Překlad názvu: Návrh a prototypová implementace logovacího frameworku pro nástroj Manta

Contents

Figures

Tables

Chapter 1

Introduction

A significant portion of each software engineer's work is developing, monitoring, debugging and fixing software. In order to be efficient at these activities, it is necessary to have information about the inner workings of the software in question, which is provided by logging.

Log is a message emitted by an application or an operating system describing its inner state, what it does or its communication with other entities. They are an essential feature of most of the applications or operating systems and are greatly utilized for dealing with issues that may arise. That is why most of the companies developing software pay great deal of attention towards providing relevant information in logs.

Though after time, once the volume of logs becomes too large, a new issue arises in regards to the log analysis. As the amount of logs increases, the analysis becomes more difficult and thus detecting and dealing with application issues becomes more demanding. That is why log analysis tools or specialized log storages are becoming more and more popular. Companies also invest more into these tools in order to detect new issues early and effectively.

The main focus of this thesis is to analyze this subject area and design a new logging solution for the ecosystem of Manta, which is a Czech company producing software for data lineage analysis. Data lineage diagrams show where a piece of data origins, where it is moved, how it is transferred and where it is outputted. Companies use data lineage to track and find errors in their data governance or for optimizing their existing systems. The process of building a data lineage for large data stores naturally produces a large amount of logs. The logging solution is therefore specifically tailored to the requirements of Manta.

The new logging solution deals with all aspects of modern log analysis problematic - log generation, transport, storing and retrieval of logs. This thesis will therefore focus on these areas. At the beginning of the thesis, business and system requirements from the contractee - Manta - are stated and analyzed. Following are chapters focusing on researching already existing solutions and discussing their compliance with the requirements, representing the analytical part of the thesis. Last is the design and prototypical implementation of the new logging solution as well as its testing.

1.1 Goal of the thesis

The main goal of this thesis is to design and implement a new logging solution for Manta ecosystem encompassing all aspects of logging - from writing a single logging command in a Java application to presenting that log in a user-friendly fashion in a web application to a consumer.

The implementation is therefore divided into two parts - logging API used by the programmers and a backend of a web application, referred to as *Log Viewer*, providing these logs to the end user.

1.2 Chapters

The thesis is divided into eight chapters:

- **Introduction** which describes the issue and motivations,
- **Business analysis** which analyzes requirements from the contractee and the issues with the current implementation,
- **Logging layer** which analyzes existing logging frameworks of Java programming language and their features,
- **Transport layer** which analyzes possible existing solutions for log data transport and discusses them in regards to the requirements,
- **Persistence layer** which analyzes possible solutions for log storage in accordance to the specified requirements,
- **Realization** which describes the design and implementation of the API as well as the design of software architecture and implementation of the prototype of the Log Viewer application,
- **Testing** which describes the approaches towards the testing of the prototype,
- **Conclusion** ends the thesis by discussing the fulfillment of requirements and possible other approaches that may be taken in this subject area,

Chapter 2

Business analysis

2.1 Manta

Manta is a Czech startup company specialized in developing software for analyzing and visualizing data lineage. Its customers are mainly large corporations from abroad, mostly from the United States. Software produced by this company is greatly utilized by these companies to automate data warehouse optimization or regulatory and other processes that would normally take weeks or months, when done by people, to be done within hours. Manta analyzes programming languages like SQL or Java and from the created analysis builds the appropriate data lineage.

2.1.1 System architecture

Manta application is composed of the following components:

- Manta Flow CLI
- Manta Flow Server
- Admin UI

Manta Flow CLI

Manta Flow CLI is a Java console application. In the process of analyzing and creating data lineage, this application serves multiple purposes - it extracts scripts and other data from source databases or data warehouses, analyses them and sends the retrieved metadata to Manta Flow Server. Furthermore, the application can process and send the metadata to third-party applications, such as IBM InfoSphere Information Governance Catalog (IBM IGC) or Informatica EDC.

Manta CLI is run in scenarios. Each scenario may have a master scenario and one master scenario may have multiple child scenarios. Both master and normal scenarios are implemented as Java beans. One master scenario usually represents one step of the data lineage creation phase for one supported technology, for example OracleExtractionMasterScenario. During one

application run, Manta CLI iteratively launches chosen master scenarios and its child scenarios.

Manta Flow Server

Manta Flow Server is a Java web application built on Spring MVC framework. The application saves metadata processed in Manta Flow CLI into a graph database, transforms it into the correct format for exporting into the previously mentioned third-party technologies or actually exports itself into other third-party technologies, such as Collibra. Furthermore, Manta Flow Server creates the visualisation of data lineage, which is later visualised in a browser or third-party applications via API.

Admin UI

Admin UI is a Java web application using Spring MVC framework. Admin UI works as an administration console for controlling Manta Flow CLI. Using it, the user may launch the CLI, change its configuration or perform updates of the system. The prototype implemented in this thesis is targeted to be a component of this application.

2.2 Logging in Manta, its issues and resulting requirements

2.2.1 Current state

Logging in Manta Flow CLI

Currently, Manta Flow CLI is configured to log into rolling files. It uses a Rolling File Appender from the logging framework Log4j 2. Rollover of the file is triggered on startup, i.e. every time the application is started, old log files are archived and logs are written into a new file. All log files from all application runs are saved into one folder. Log files that were archived are not automatically removed.

As stated in the previous section, Manta Flow CLI is run in scenarios. Each run of the scenario generates one log file. Both the master scenario and the normal scenario have their own log file. Currently, Manta has more than 150 scenarios. Therefore, after each full run of the application, hundreds of log files may be generated. If the data source is large enough, the size of one log file may be in hundreds of megabytes (hundreds of thousands to couple of millions of logs).

Logging in Manta Flow Server

In Manta Flow Server, logging is configured so that logs are written into rolling log files using Rolling File Appender from Log4j 2. It uses a Time-

Based Triggering Policy (rollover is triggered after a set time period) and SizeBased Triggering Policy (rollover is triggered after the file reaches a certain size). After rollover, only 10 latest log files are archived, the older ones are automatically deleted.

Logging in Admin UI

Admin UI has the same logging configuration as Manta Flow Server.

Current logging architecture

In the Figure 2.1, current architecture of Manta Flow and its logging is modeled. This diagram is one of the possible ways of using Manta. In the real world, it is possible to either deploy Manta to one machine, as it is pictured here, or decompose it to multiple physical servers.

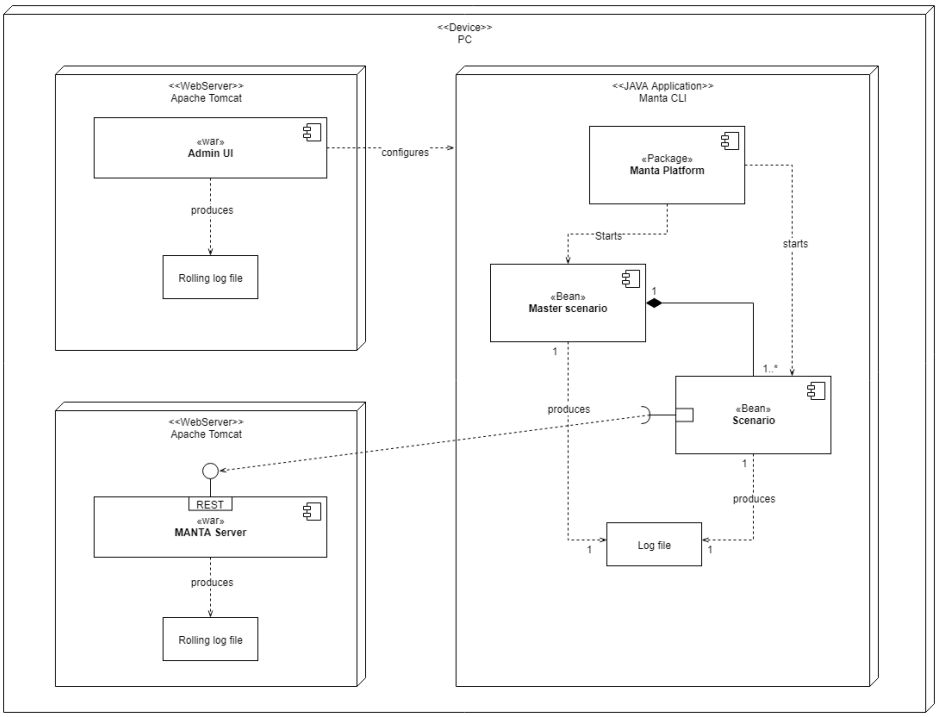


Figure 2.1: Current architecture of Manta and its logging

2.2.2 Issues with the current logging configuration

From the descriptions of the current state of logging in the previous section it is apparent that this configuration is not simple to maintain, read nor analyze. The reasons are namely - large amount of log files and their potential size, due to which searching for information contained in them becomes highly ineffective, or a high degree of decentralization (each component logs locally and with custom settings).

Large amount of log files

It is very complicated for a customer to get accustomed to a large amount of raw information stored in log files and in case of any issue with the application, it is very difficult, without an advanced knowledge of the application, to find errors causing it. As a result, customers usually contact customer support with even relatively trivial issues that can be easily fixed. It not only overburdens the customer support, but also causes delays on the side of the customer. The fact that old log files are not automatically deleted, or archived elsewhere, only increases the problem.

Size of the log files

Parsing information from a text file with size ranging from tens of thousands to millions of lines is very tedious for a human. To make this process faster, it is necessary to be able to filter the unwanted information from the file. Currently, the customer support uses console tools *grep* and *sed* which, using regular expressions, are able to partially filter the logs. However, this method is not completely reliable and flawless, since, for example, two developers may log one error with two different messages and thus make the filtration even more difficult. Furthermore, the scalability is very ineffective, since in a continual development, more and more new errors are created and it is difficult to incorporate them into this method.

Decentralized logging

As was mentioned in subsection 2.1.1, Manta works as a cluster of three closely cooperating components. These three components each log locally, which may cause potential issues with log analysis. During CLI scenario execution, CLI continuously sends or receives data from Manta Flow Server, which also has functionality of its own. In case of certain issues, it is sometimes necessary to analyze multiple logs from the same time written in multiple files. Ideally, the new solution would centralize logging and thus enable a user to unite the logs from multiple sources.

Inappropriate log messages

Currently, a great amount of errors are logged in such a way that is not understandable to a normal user. The actual users of Manta increasingly do not have an IT background and someone, who is not familiar with the source code of the application, will deduce very little from the log in Figure 2.2 and it is highly unlikely that the user will successfully solve the issue. Alternatively, if an error is not detrimental to a desired output but is logged in such a way that it generates an impression in the user that it is, the user may think that the output is not correct. Manta customer support is regularly contacted with issues that are not detrimental to the output of the program only because users think that there might be an issue due to an inappropriate log message.

The solution to this issue may be to log errors dually - for a user and for a developer. The error log message intended for the end user would be user friendly, while the developer log would incorporate technical details which are necessary during the development or fixing bugs.

```
2019-07-15 17:12:57.074 4 ERROR eu.profinit.manta.dataflow.generator.modelutils.GraphScenario Task "PostgresqlDataflowTask" failed while processing
"\postgres\public\Functions\foo.4.sql". The processing will continue with next step, the result may be inaccurate however, or next steps may fail.
java.lang.NullPointerException: null
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.process(FlowVisitor.java:1121) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.process(FlowVisitor.java:108) ~[?:?]
    at eu.profinit.manta.connector.postgresql.resolver.ast.impl.AstMasterSelect.accept(AstMasterSelect.java:195) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.processAstNode(FlowVisitor.java:316) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.process(FlowVisitor.java:1087) ~[?:?]
    at eu.profinit.manta.connector.postgresql.resolver.ast.impl.AstStandaloneSelect.accept(AstStandaloneSelect.java:32) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.processAstNode(FlowVisitor.java:316) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.processChildren(FlowVisitor.java:298) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.process(FlowVisitor.java:176) ~[?:?]
    at eu.profinit.manta.connector.postgresql.resolver.ast.PostgresqlAstNode.accept(PostgresqlAstNode.java:59) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.processAstNode(FlowVisitor.java:316) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.processChildren(FlowVisitor.java:298) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.process(FlowVisitor.java:176) ~[?:?]
    at eu.profinit.manta.connector.postgresql.resolver.ast.PostgresqlAstNode.accept(PostgresqlAstNode.java:59) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.processAstNode(FlowVisitor.java:316) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.processChildren(FlowVisitor.java:298) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.process(FlowVisitor.java:1897) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.process(FlowVisitor.java:108) ~[?:?]
    at eu.profinit.manta.connector.postgresql.resolver.ast.impl.AstBlock.accept(AstBlock.java:55) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.processAstNode(FlowVisitor.java:316) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.processChildren(FlowVisitor.java:298) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.process(FlowVisitor.java:176) ~[?:?]
    at eu.profinit.manta.connector.postgresql.resolver.ast.PostgresqlAstNode.accept(PostgresqlAstNode.java:59) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.processAstNode(FlowVisitor.java:316) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.processChildren(FlowVisitor.java:298) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.process(FlowVisitor.java:176) ~[?:?]
    at eu.profinit.manta.connector.postgresql.resolver.ast.PostgresqlAstNode.accept(PostgresqlAstNode.java:59) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.processAstNode(FlowVisitor.java:316) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.processRoutineDefinition(FlowVisitor.java:2004) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.process(FlowVisitor.java:1916) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.process(FlowVisitor.java:108) ~[?:?]
    at eu.profinit.manta.connector.postgresql.resolver.ast.impl.AstFunctionDefinition.accept(AstFunctionDefinition.java:49) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.processAstNode(FlowVisitor.java:316) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.processChildren(FlowVisitor.java:298) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.process(FlowVisitor.java:1904) ~[?:?]
    at eu.profinit.manta.connector.postgresql.resolver.ast.impl.AstCreateFunction.accept(AstCreateFunction.java:42) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.processAstNode(FlowVisitor.java:316) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.processChildren(FlowVisitor.java:298) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.process(FlowVisitor.java:176) ~[?:?]
    at eu.profinit.manta.connector.postgresql.resolver.ast.PostgresqlAstNode.accept(PostgresqlAstNode.java:59) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.processAstNode(FlowVisitor.java:316) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.processChildren(FlowVisitor.java:298) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.FlowVisitor.process(FlowVisitor.java:176) ~[?:?]
    at eu.profinit.manta.connector.postgresql.resolver.ast.PostgresqlAstNode.accept(PostgresqlAstNode.java:59) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.scenario.AbstractVisitorGraphTask.doExecute(AbstractVisitorGraphTask.java:47) ~[?:?]
    at eu.profinit.manta.dataflow.generator.postgresql.scenario.AbstractVisitorGraphTask.doExecute(AbstractVisitorGraphTask.java:19) ~[?:?]
    at eu.profinit.manta.platform.automation.AbstractTask.execute(AbstractTask.java:54) ~[manta-platform-automation-1.25-SNAPSHOT.jar:?]
    at eu.profinit.manta.platform.automation.AbstractScenario.executeTask(AbstractScenario.java:129) [manta-platform-automation-1.25-SNAPSHOT.jar:?]
    at eu.profinit.manta.platform.automation.AbstractScenario.executeTasks(AbstractScenario.java:116) [manta-platform-automation-1.25-SNAPSHOT.jar:?]
    at eu.profinit.manta.dataflow.generator.modelutils.GraphScenario.doExecute(GraphScenario.java:115)
    [manta-dataflow-generator-model-utils-1.25-SNAPSHOT.jar:?]
    at eu.profinit.manta.platform.automation.AbstractScenario.execute(AbstractScenario.java:88) [manta-platform-automation-1.25-SNAPSHOT.jar:?]
    at eu.profinit.manta.platform.parallel.ParallelScenario$ScenarioLauncher.run(ParallelScenario.java:169)
    [manta-platform-parallel-1.25-20190611.223931-105.jar:?]
    at java.util.concurrent.ThreadPoolExecutor.runWorker(Unknown Source) [?:1.8.0_201]
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(Unknown Source) [?:1.8.0_201]
    at java.lang.Thread.run(Unknown Source) [?:1.8.0_201]
```

Figure 2.2: Inappropriate error messages

2.2.3 Requirements

In this section, requirements for the new log management solution, stemming from the issues with the current solution, will be discussed. The main requirement is to design a log management system that allows to view, filter and search logs from all components in one central component. Requirements are described in detail in the following subsections.

Log Viewer

The main requirement is to design and implement backend of a Log Viewer component for the Admin UI web application. In the context of this thesis the implementation will be prototypical and will be a standalone web application. The log viewer must enable users to filter, group, search or analyze in other ways logs generated by all of the components of Manta.

Persistence layer

A central log viewer requires a central storage of all logs for effective analysis. The new solution must handle large amount of data without affecting the performance of the application. The data storage must be queryable.

Transport layer

Since Manta works as a cluster of three components, logs need to be transported to a central data storage using a transport layer that may be used by any other application with a simple configuration.

Filtration and querying

The individual logs can be currently filtered by application run, scenario, component or log severity. These parameters are not sufficient for effective filtration, since it is possible and not rare for one scenario to log thousands of errors. More attributes should be added to allow more effective querying and filtration.

Error categorization

Design and implement an extendable error categorization API. This API should be simple to use and it should not be possible to knowingly or unknowingly log errors in any other way.

Message duality

Each categorized error should have a user friendly message for normal users and a detailed message for developers.

Logging API

Implement a logging API incorporating additional log parameters, error categorization and message duality functionality in such a way that it is simple for a developer to use so that developers are not discouraged from logging.

Support packages

Usually, when customers have any issues with an application, they send logs from an application run. Due to a large amount of log files, the customer often does not know which ones to send, so they send all of them. The new solution should enable user to easily select relevant logs, package them and send them to the customer support. It should be possible to open the package again in the Log Viewer.

Software architecture

The log viewer component should not introduce any architectural changes to the application. That means, any third party technologies that are used by the logging solution must be embedded into the existing application and must not run separately to the application (e.g. no separate database server, it must be embedded into the application).

Encryption

In order to secure the communication between components, all remote communication must be encrypted using Secure Sockets Layer (or TLS) protocol, which Manta currently also uses to secure communication. Even though Manta is usually used on one machine where all connectivity is to localhost, where this would not be required as much, it is possible to run it on multiple machines where this would be required. Logs may contain sensitive user data (mostly database metadata) and additional security is therefore useful.

2.3 Functional requirements

In the Table 2.1 are laid out all of the functional requirements on the new solution, most of which were described more in detail in the previous section. Later in the text, these requirements will be referenced by codes that were assigned in this table.

Code	Requirement
F1	Design software architecture for Log Viewer component
F2	Design and implement a central data storage of all logs from all components
F3	Design and implement a transport layer between components
F4	Create a logging API that fulfills the requirements below
F4.1	Implement an extendable error categorization as a part of the logging API
F4.2	Implement dual messages for error logs as a part of logging API
F4.3	Logged errors should have possible solutions to resolve it
F5	Design and implement a functionality of adding context to logs, such as assigning data source objects to logs
F6	Implement a service for support package creation
F7	Implement a service for support package upload
F8	Implement automatized pruning of old logs
F9	Implement full-text searching over error logs

Table 2.1: Functional requirements

2.4 Non-functional requirements

In the Table 2.2 are laid out all of the non-functional requirements on the new solution. Later in the text, these requirements will referenced by codes that were assigned in this table.

Code	Requirement
NF1	Querying of log data storage should be fast enough to be realistically usable
NF2	Logging API should be simple to use and the only possible way to log
NF3	Logging API should not negatively impact the application performance
NF4	All tools required by the new solution should be installable using the Manta installer
NF5	Installation should be possible without using the installer
NF6	All third party tools should be free to use, have open-source license with no GPL-like behavior
NF7	Application architecture must not be changed
NF8	Remote communication must be encrypted using SSL

Table 2.2: Non-functional requirements

2.5 Use cases of the new solution

In this section are described use cases of the Log Viewer component and logging API that powers it, as well as actors using certain aspects of the new solution.

2.5.1 Actors

Actors in Unified Modeling Language use case diagrams are entities that interact with the system that is being modeled. There are four main actors participating in the usage of the new logging solution:

- **Software engineer**, who develops and maintains Manta software components, will use the new solution mainly in the lower (API) levels,
- **Customer**, who is the end user of the application, interacting with the application purely from the front-end,
- **Customer support**, who uses the application largely the same as a customer,
- **Time**, since there will be an automatized, time-triggered internal functionality

2.5.2 Use case diagram

In the Figure 2.3 is an UML diagram describing use cases of the new solution and assigning them to appropriate actors. It is apparent, that the *customer support* actor shares most of the use cases with a *customer*. It is intended to be so that the Log Viewer application is used to solve issues that arise in the application as much as possible, even internally in the company.

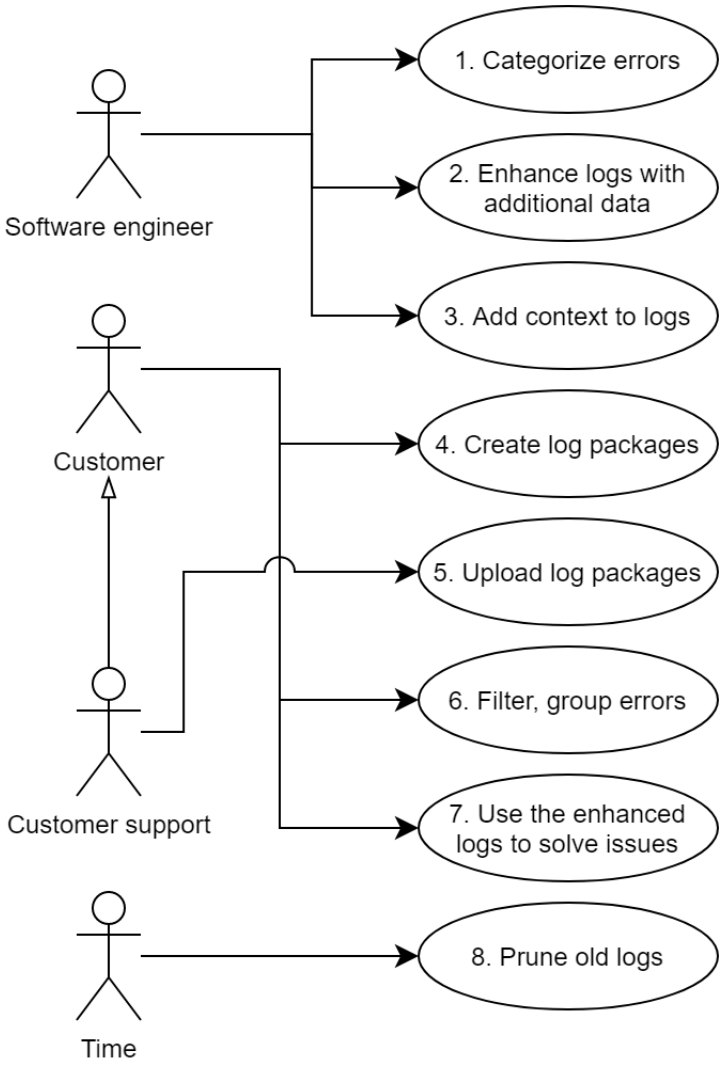


Figure 2.3: Use case diagram

2.5.3 Aggregated use cases

In the table below is a list of use cases declared in a use case diagram in the previous section. Use cases are named, described and mapped to corresponding requirements.

Name	Use case	Requirement
Error categorization	Using an API for creating and categorizing logged errors	F4.1
Error attributes	Using an API for adding further information to error logs	F4.2, F4.3
Logging context	Using an API for adding context to following logged errors	F5
Package creation	Create a package containing logs from an application/scenario execution	F6
Package upload	Upload a package containing logs from an application/scenario execution and show them in log viewer	F7
Query error logs	Use log viewer to query persisted errors using its attributes or full text search	F7
Solve errors	Solve errors listed in the log viewer using provided solution	F4.3
Prune old logs	Automatically prune expired logs from old runs	F8

Table 2.3: Use cases of the new solution

Chapter 3

Logging layer

The following chapter presents existing logging solutions for the Java programming language, in which is Manta developed. The logging solutions will be compared with each other in order to find the most fitting for the requirements stated in the previous section. At the end of the chapter will be stated the chosen technology discussed reasons for doing so.

When developing in Java, a developer has a great array of choices for logging. They may use a logging facade which gives developers flexibility in using multiple implementations of logging frameworks, or they may use one logging framework directly.

3.1 Logging facades

Logging facade is an implementation of the facade design pattern. "Facade defines a higher-level interface that makes the subsystem easier to use." [1] In other words, logging facade provides an abstraction over an existing logging framework. This allows developers to change the underlying logging framework without having to refactor logging in the whole application. Manta currently uses SLF4J facade.

3.1.1 SLF4J

"The Simple Logging Facade for Java (SLF4J) serves as a simple facade or abstraction for various logging frameworks (e.g. java.util.logging, logback, log4j) allowing the end user to plug in the desired logging framework at deployment time."[2] "The way SLF4J works is that at runtime SLF4J scans the class path and picks the first .jar that implements the SLF4j API."[3] SLF4J is distributed with bindings for the following logging frameworks:

- Log4j
- Java Logging API
- Apache Commons Logging
- Logback

- Simple (logging to System.out or Sytem.err)
- Nop (discarting all logging)

Furthermore, SLF4J provides ability to reroute logging from other implementations into SLF4J. "SLF4J ships with several bridging modules which redirect calls made to log4j, JCL and java.util.logging APIs to behave as if they were made to the SLF4J API instead."[4] A great amount of legacy systems still uses JCL, which is discussed in the following section. Famous example is the IoC framework Spring, which still uses JCL. Though in its documentation it is recommended to turn off the JCL logging and use the SLF4J bridge. "If we could turn back the clock and start Spring now as a new project it would use a different logging dependency."[5]

3.1.2 Apache Commons Logging

Apache Commons Logging, also known as Jakarta Commons Logging, is a logging facade developed by Apache Software Foundation. Commons Logging is a very thin facade and does not provide abstractions of many advanced logging features like mapped diagnostic context, markers and others, which SLF4J does. JCL is also infamous for its issues with the class loader. "The class loader based automatic 'discovery process' is the principal weakness of the commons-logging API because it results in a substantial jump in complexity." [6] Furthermore, the discovery algorithm also causes unexpected behavior and even memory leaks. SLF4J deals with the classloader issues by using static binding, i.e. the classloader always looks for an implementation of a class *org.slf4j.impl.StaticLoggerBinder*, which contains binding to the desired logging framework.

3.2 Logging frameworks

3.2.1 Java Logging API

"The Java™ Logging APIs, introduced in package java.util.logging, facilitate software servicing and maintenance at customer sites by producing log reports suitable for analysis by end users, system administrators, field service engineers, and software development teams."[7] It is the official logging implementation of Java distribution. It has been a part of the Java distribution since version 1.4.

Handlers

"A Handler object takes log messages from a Logger and exports them. It might for example, write them to a console or write them to a file, or send them to a network logging service, or forward them to an OS log, or whatever."[9] Developers may also implement their own handlers or extend the existing ones. Java Logging API is distributed with following implemented handlers:

Handler	Description
ConsoleHandler	A simple handler for writing formatted records to System.err
FileHandler	A handler that writes formatted log records either to a single file, or to a set of rotating log files
StreamHandler	A simple handler for writing formatted records to an *OutputStream*
SocketHandler	A handler that writes formatted log records to remote TCP ports
MemoryHandler	A handler that buffers log records in memory

Table 3.1: Handlers of Java Logging API [10]

Levels

Every logged message has its severity as an attribute. Different implementations of logging use different names for these attributes, but the principle is always the same. Correct usage of severities improves the effectiveness of log analysis. Java Logging API has the following predefined levels:

Level	Description
SEVERE	Message level indicating a serious failure.
WARNING	Message level indicating a potential problem.
INFO	Message level for informational messages.
CONFIG	Message level for static configuration messages.
FINE	Message levels for debug information by degree of importance.
FINER	
FINEST	

Table 3.2: Levels of Java Logging API [11]

3.2.2 Log4j 2

Log4j 2 is a logging framework developed by the Apache Software Foundation. It is the continuation of Log4j framework whose development started in 1999. Log4j was the first Java logging framework. Over the years, due to compatibility issues with very old versions of Java, its development slowed down, until it was stopped in 2015. Framework was then refactored and released as Log4j 2.[12]

Appenders

Appenders, which are equivalent to handlers in Java Logging API, are responsible for writing logs to the set destination. Log4j 2 is distributed with a great amount of appenders [13]:

- **AsyncAppender** is an asynchronous appender that wraps other appenders and writes data to them in another thread.
- **CassandraAppender** writes to NoSQL database Apache Cassandra.
- **ConsoleAppender** writes logs to System.out and System.err.
- **FailoverAppender** groups multiple appenders, sorts them by importance and when an appender with higher importance fails during export, it uses an appender with lesser importance.
- **FileAppender** writes logs to a text file.
- **FlumeAppender** writes logs to Apache Flume, which is a "distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data."[14]
- **JDBCAppender** writes logs to a relational database using a standard JDBC API.
- **JMSAppender** writes logs to Java Messaging Service, which is "a Java API that allows applications to create, send, receive, and read messages."[15]
- **JPAAppender** writes logs to a relational database using Java Persistence API[13].
- **HttpAppender** sends logs to a remote location using HTTP protocol.
- **KafkaAppender** writes logs to Apache Kafka, which is a "distributed streaming platform"[16].
- **MemoryMappedFileAppender** "maps a part of the specified file into memory and writes log events to this memory, relying on the operating system's virtual memory manager to synchronize the changes to the storage device." [13]
- **NoSQLAppender** "writes log events to a NoSQL database." [13] Log4j 2 provides further specified appenders for MongoDB NoSQL dataabse.
- **OutputStreamAppender** "provides the base for many of the other Appenders such as the File and Socket appenders that write the event to an Output Stream. It cannot be directly configured."[13]
- **RandomAccessFileAppender** writes to a file using RandomAccessFile Java class.
- **RewriteAppender** "allows the LogEvent to manipulated before it is processed by another Appender. This can be used to mask sensitive information such as passwords or to inject information into each event." [13]

- **RollingFileAppender** writes logs to a text file, which is rolled according to a set TriggeringPolicy and RolloverPolicy. "The triggering policy determines if a rollover should be performed while the RolloverStrategy defines how the rollover should be done."[13]
- **RollingRandomAccessFileAppender** combines the functionality of RandomAccessFileAppender and RollingFileAppender.
- **RoutingAppender** reroutes logs to correct appender according to set conditions.
- **SMTPAppender** "sends an e-mail when a specific logging event occurs, typically on errors or fatal errors."[13]
- **SocketAppender** "is an OutputStreamAppender that writes its output to a remote destination specified by a host and port."[13]
- **SyslogAppender** "is a SocketAppender that writes its output to a remote destination specified by a host and port in a format that conforms with either the BSD Syslog format or the RFC 5424 format."[13]
- **ZeroMQ/JeroMQ Appender** writes to ZeroMQ, which is a "high-performance asynchronous messaging library, aimed at use in distributed or concurrent applications."[17]

Levels

Log4j 2 uses different log levels than Java Logging API:

Level	Description
OFF	No events will be logged.
FATAL	A severe error that will prevent the application from continuing.
ERROR	An error in the application, possibly recoverable.
WARN	An event that might possible lead to an error.
INFO	An event for informational purposes.
DEBUG	A general debugging event.
TRACE	A fine-grained debug message, typically capturing the flow through the application.
ALL	All events should be logged.

Table 3.3: Levels of Log4j [18]

Parametrized logging

Parametrized logging allows developers effectively adding outside parameters to log messages. Possible ways of inserting parameters to log messages may be:

```
logger.info(person.getName() + " is " + person.getAge() + "
    years old and has " + person.getNumOfChildren() + "
    children.");

logger.info("{} is {} years old and has {} children.",
    person.getName(), person.getAge(),
    person.getNumOfChildren());
```

According to the SLF4J documentation, the latter way of inserting parameters delays the message construction until after the framework checks, whether the logging level is enabled. The first way of logging constructs the message regardless. The latter "will outperform the first form by a factor of at least 30, in case of a *disabled* logging statement."[2] Furthermore, it has an added benefit of better readability.

Thread Context

"Stamping log events with a common tag or set of data elements allows the complete flow of a transaction or a request to be tracked. We call this *Fish Tagging*. Log4j provides two mechanisms for performing Fish Tagging; the Thread Context Map and the Thread Context Stack. The Thread Context Map allows any number of items to be added and be identified using key/value pairs. The Thread Context Stack allows one or more items to be pushed on the Stack and then be identified by their order in the Stack or by the data itself."[19] Logging facade SLF4J provides a facade of this feature.

Asynchronous logging

"Asynchronous logging can improve your application's performance by executing the I/O operations in a separate thread."[20] This feature can be implemented by wrapping the desired output appender using AsyncAppender or using AsyncLogger, which requires external dependency on LMAX Disruptor. "Asynchronous Loggers internally use the Disruptor, a lock-free inter-thread communication library, instead of queues, resulting in higher throughput and lower latency."[20]

3.2.3 Logback

Logback is a Java logging framework "intended as a successor to the popular log4j project"[21]. It is closely coupled with SLF4J library, which it natively implements. Logback built upon the classic Log4j framework and brought many of the advanced features that are now common in logging frameworks. New version of Log4j - Log4j 2 - included those features and brought many others as well. Furthremore, Log4j 2 implemented asynchronous logging, which is much faster than async appender of Logback[1].

[1] Benchmarks comparing throughput of different async logging implementations, available from: https://logging.apache.org/log4j/2.x/manual/async.html#Performance

3.3 Technology choice

Manta currently uses the combination of the logging facade SLF4J and logging framework Log4j 2. The main reason, why SLF4J is used in different projects, is due to flexibility it provides in changing logging frameworks. This new solution will serve a very similar function - provide a logging facade with an added functionality. Therefore, there is no reason to keep SLF4J anymore, since it does not provide any further functionality above Log4j 2 and will thus become redundant. If the developers desire to change the logging framework some time in the future, only refactoring the base logging API classes will be necessary. Regarding the logging framework, Log4j 2, there is no need to change it, since it offers the most advanced functions and has the best performance among the logging frameworks.

Chapter 4

Transport layer

In this chapter will be discussed existing solutions for communication within distributed enterprise systems. In the scope of this work, these technologies offer solutions for transportation of logs to the centralized data storage. At the beginning of the chapter will be described common messaging models. After that will be described implementations of these models and their compliance with the requirements. At the end of the chapter will be discussed the technology choice and reasons for doing so.

4.1 Message-oriented middleware

Message-oriented middleware is an infrastructure facilitating communication between systems based on messages. "A client of a MOM[1] system can send messages to, and receive messages from, other clients of the messaging system. Each client connects to one or more servers that act as an intermediary in the sending and receiving of messages. MOM uses a model with a peer-to-peer relationship between individual clients; in this model, each peer can send and receive messages to and from other client peers."[22] In contrast with an older interprocess communication technique, remote procedure calls, message-oriented middleware allows distributed systems to be more loosely coupled.[15]

Messaging infrastructure is usually implemented using message brokers. "A message broker is software that enables applications, systems, and services to communicate with each other and exchange information. The message broker does this by translating messages between formal messaging protocols."[25] "They serve as intermediaries between other applications, allowing senders to issue messages without knowing where the receivers are, whether or not they are active, or how many of them there are."[25] Most commonly known message brokers are ActiveMQ, RabbitMQ or Apache Kafka which will be discussed in the following sections.

[1] Message-oriented middleware

4.1.1 Messaging models

Message-oriented middleware is based on two main messaging models - Point-to-Point and Publish/Subscribe.

Point-to-Point model utilizes First-In-First-Out (FIFO) queues. Producers (in the context of this model also called *senders*) push messages to the end of the queue, in which they are sorted in the order a messaging broker receives them. Consequently, consumers (also *receivers*) consume the messages by polling them from the queue. Each message is consumed by one receiver. Messages are delivered and persisted in the messaging broker until a receiver consumes them.[2] In the Figure 4.1 is a diagram representing a Point-to-Point messaging model where each message is consumed only once by a different receiver. "The point-to-point model supports load balancing, which allows multiple receivers to listen on the same queue, therefore distributing the load."[27]

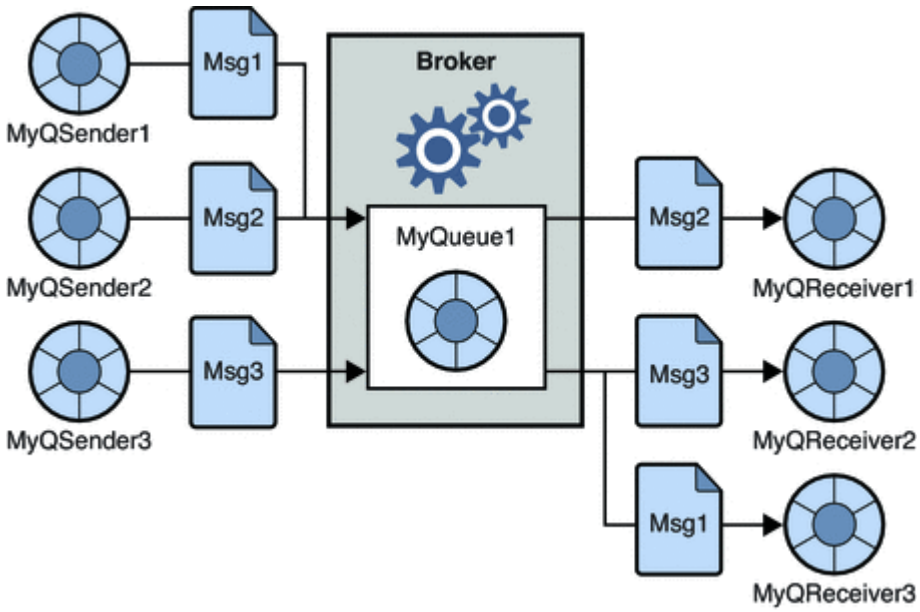


Figure 4.1: Complex Point-to-Point Messaging, diagram from [23]

Publish/subscribe model differs from Point-to-Point model in several aspects. Producers (in the context of this model also *publishers*) publish messages to defined topics from which the messaging broker reroutes the messages to consumers (also *subscribers*) who have subscribed to the defined topic. A subscriber therefore does not poll a message from message queue itself but subscribes to a topic and waits until broker delivers published messages to it. This model furthermore allows one producer to send messages to an arbitrary number of consumers, whereas Point-to-Point model is used to deliver messages to one consumer. In the Figure 4.2 is a diagram representing a simple Publish/subscribe messaging model which one message is consumed

[2]Paragraph sourced from [27]

by two subscribers.[3]

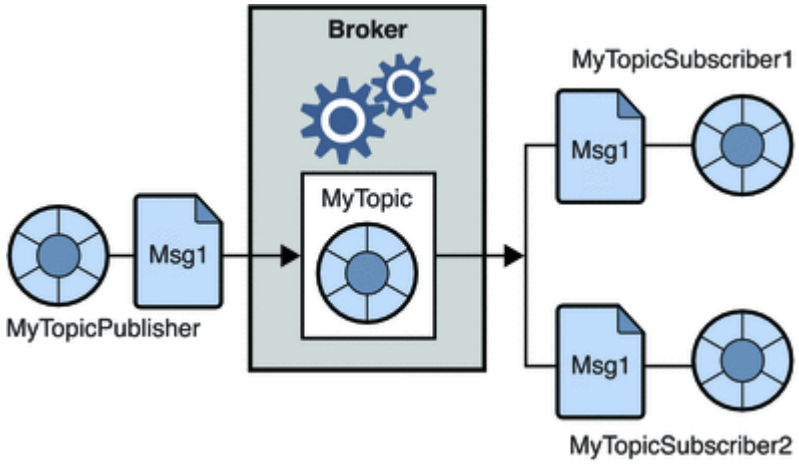


Figure 4.2: Simple Publish/Subscribe Messaging, diagram from [24]

4.2 Java Message Service

Java Message Service (JMS) is an abstract API providing interface for message-oriented middleware implementations. It is a part of Java Enterprise Edition. "JMS defines a set of interfaces and semantics that allow Java applications to communicate with other messaging implementations. A JMS implementation is known as a JMS provider. JMS makes the learning curve easy by minimizing the set of concepts a Java developer must learn to use enterprise messaging products, and at the same time it maximizes the portability of messaging applications."[26] It supports both messaging models discussed in the previous section.

Java applications utilizing JMS providers by sending or receiving messages are called JMS clients[4].

4.2.1 JMS providers

There are many industry used implementations of JMS interface. It is possible to choose from a wide range of open-source solutions like ActiveMQ, ActiveMQ Artemis or RabbitMQ to properietary solutions like Amazon SQS or IBM MQ. Due to the non-functional requirement NF6, only open-source solutions will be considered in the following subsections.

ActiveMQ

Apache ActiveMQ is a message broker implemented in Java. According to the ActiveMQ documentation, it supports integration not just with Java Message Service, but also with numerous other platforms based on C++,

[3] Paragraph sourced from [22]
[4] Sourced from [26]

C# or Ruby. Its usage can range from running a large cluster of servers to running embedded within an application's Java Virtual Machine. "With such a deployment, communication between the client and broker (server) take place within the same JVM, avoiding the extra hop required to go from producer to broker to consumer."[22] Therefore, ActiveMQ fulfills the architectural requirement NF7. Furthermore, ActiveMQ supports encrypting communication using SSL, thus complying with the NF8 requirement.

ActiveMQ Artemis

ActiveMQ Artemis is an open-source message broker implemented in Java and is supposed to be the successor to the classic ActiveMQ, once they share enough key features, according to the ActiveMQ documentation. It is based on the HornetMQ codebase, developed by JBoss, that was donated to the Apache Foundation, which re-released it under the name "ActiveMQ Artemis". It officially supports embedded execution and SSL encryption, just like the classic ActiveMQ.

RabbitMQ

RabbitMQ is a message broker, developed by Pivotal Software, implemented in Erlang. Although it is one of the most popular message brokers currently used in the industry, it does not meet the necessary requirements for this work, since it cannot be officially executed embedded within an application's JVM and has to run as a separate server application.

4.3 Apache Kafka

Apache Kafka is an open-source distributed data streaming platform developed by Apache Foundation. It has a much wider scope of use cases than traditional message brokers like ActiveMQ or RabbitMQ, ranging from traditional messaging to website activity tracking, collecting metrics data, log aggregation or stream processing. It is meant to be highly horizontally-scalable and fault tolerant.

Originally, Kafka's main use case was website activity tracking. "A website's users interact with frontend applications, which generate messages regarding actions the user is taking. This can be passive information, such as page views and click tracking, or it can be more complex actions, such as information that a user adds to their profile."[28] This data can be further fed to applications that process them, like monitoring applications, machine learning systems or, more specifically, recommendation making systems on e-commerce sites.

More in the scope of this work, Kafka is also often used for log aggregation from different systems. "Log messages can be published in the same way, and can be routed to dedicated log search systems like Elastisearch or security analysis applications."[28] Alternatively, it can automatically reroute physical

log files from client applications to centralized data storages like file servers or Hadoop.[5]

Kafka has been enjoying a majorly increased popularity among large corporations in recent years. "Kafka is being used by tens of thousands of organizations, including over a third of the Fortune 500 companies."[28] More recently, Confluent, a company founded by engineers who started development on Kafka and dealing with data stream analytics, has reported that over 60 percent of Fortune 100 companies rely on Apache Kafka[6].

Architecturally, Kafka is meant to be run in a distributed system on a cluster of servers. It does not officially support execution where it is embedded within an application's JVM, since such limited setup defeats most of the advantages that Kafka provides - that is horizontal scalability and fault-tolerance that stems from it. Therefore, Kafka is not compliant with requirement NF7.

4.4 Other industry used messaging tools

With the increased focus on Big Data in the software industry, there has been a corresponding major increase in industry focus on technologies providing data transportation. Other major players in this sector are for example Apache Flume or Logstash, which is a part of Elastic stack, which will also be discussed in the next chapter.

Apache Flume is a "distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data."[14]. According to its documentation, its main use case is in aggregating log data within a distributed system and pushing them forward to a data storage or other processor, mainly Hadoop file system. Although there are official data sinks for technologies like Elasticsearch, HBase or even Kafka. Its usage is rather similar to Kafka in that it is meant to run in a distributed system on a cluster of servers and does not support embedded execution.

Lastly, there are also specific networking libraries dealing with interprocess communication. Libraries like ZeroMQ or Netty are the lowest-level implementations of data transportation methods described in this chapter. Netty, for example, is even used internally in ActiveMQ Artemis for networking [37]. Generally, libraries like these provide a great performance potential but on the other hand, they are often more difficult to implement than simply configuring a message broker and may have missing some key features - ZeroMQ does not for example support encrypting communication using SSL out of the box. Nevertheless, these libraries are naturally embedded in the application and fulfill other necessary requirements on the new solution.

[5]Previous paragraphs sourced from [28]
[6]Sourced from [36]

4.5 Chosen technology

Transport layer is an abstraction of the F3 requirement, which says that logs from all applications should be transported to a centralized location. In previous sections were described possible ways of implementing log aggregation in a distributed system. The chosen solution has to meet the necessary requirements, specifically requirements NF4, NF5, NF6, NF7 and NF8. In summary, these requirements mean that chosen solution must be embedded in the Admin UI web application and must be secure.

In the mentioned chapter were presented solutions that greatly differ in size and use cases. At one end, there is a system like Apache Kafka, which is very popular in the industry for the required use case. At the other end, there are network libraries like Netty or ZeroMQ and lastly, middle-of-the-road solutions like ActiveMQ. The solution that was eventually chosen was ActiveMQ Artemis.

Firstly, it meets all of the necessary requirements - it is possible to embed it in any Java application, which means that it does not change the installation process of Manta in any way, and provides simple interface for securing all communication using Secure Sockets Layer. Secondly, it already provides all of the necessary functionality out of the box and is thus simple to configure and implement.

The other possible solution would be to use a library like Netty for interprocess communication. Netty is even used internally in the ActiveMQ Artemis for facilitating communication. Artemis was therefore chosen because it is already tested solution in the industry and prevents reinventing the wheel, which would occur when using more low-level libraries.

In comparison to Kafka, ActiveMQ is a much more lightweight solution. Kafka, despite its popularity, is not a suitable solution, given the current architectural limitations. It cannot be embedded in an application, requires another process to run on top of Kafka itself (Apache Zookeeper) and is a very heavyweight solution. For example, the binary distribution of Kafka is 4 times bigger than Admin UI web application itself. This is a similar case with many other log aggregation tools like Flume or Logstash.

Lastly, Artemis is a preferable solution to the classic ActiveMQ because the classic version will soon be deprecated and it is generally recommended to use Artemis when developing new applications.

Chapter 5

Persistence layer

This chapter will present possible industry solutions for storing logs, such as files or databases. Choice of the data governance strategy varies greatly depending on the organization's needs, technologies discussed in this chapter will therefore be examined based on the requirements stated in section 2.3 and section 2.4.

Choice of the repository is limited by non-functional requirements NF1, NF5, NF6 and NF7. In other words, the chosen technology must be an open source project that does not change the software architecture of the product and must support effective querying, including full text searching over log messages.

According to *Logging and log management*, logs can be stored in any medium capable of storing data - options span from disks, DVDs, cloud storages, RDBMS and others. The choice depends on the individual use case where variables influencing the decision are mostly "price, capacity, and speed of access and—what is VERY important is the ability to get to the right log records in a reasonable period of time."[30]

5.1 Text-based files

The most common approach for log data persistence is to store it in text-based files. The reason for that is the low computing cost of creating them and appending to them and the simplicity of the implementation of this solution in any programming language. Logs can be stored both in flat-text and binary files.

Flat-text files are "a flat schema-less file that may follow a common pattern or be free form."[30] These files are human readable and thus log analysis can be performed by simply reading the files, presuming they are of reasonable size. Furthermore, it is possible to use tools like *grep* and *sed* which help the log analysis by filtering the logs and performing other text-based operations over them.

Logs in the text files can follow a common pattern. Most of the logging frameworks, like Log4j or Logback, enable users to choose a pattern themselves. Alternatively, logs can use a syslog pattern, which is standardized in an operating system, although implementations may vary across operat-

ing systems. Manta currently uses text-based files for logging in all of its components, as described in the previous chapters.

Once flat text files become too large, aforementioned tools like *grep* or *sed* start to become inefficient and human-based log analysis increasingly more difficult. To counter this, it is possible to index the text files for more effective querying. One of the tools commonly used for indexing text files is Java-based full-text search library Apache Lucene. Lucene is commonly used by many tools providing full-text search functionality, including many databases or search engines like Elasticsearch or Apache Solr.[1]

5.2 Binary files

Alternative to the human-readable flat text files are binary files. "Binary log files as the name indicates are machine-readable log files that applications generate that require special utilities or tools to read and process them."[30] Binary files are for example used for internal logging in databases like MySQL[2] or MaxDB[3], an enterprise relational database system from SAP, or other enterprise applications like Informatica Service Manager[4], which is a administration console for tools developed by data governance company Informatica. Binary files further require an encoding and decoding logic in order to make the logs human readable, editable and searchable, which makes them more difficult to implement and thus less popular than basic text files.

5.3 Database

A commonly used alternative for storing log data is a database. "Many organizations find that writing log information to a database is useful in opening access to stakeholders that need log information in a format that can be quickly searched and queried, and to facilitate the setup and use of front-end tools in the log review process."[30] Database systems can range from simple file based databases like H2 or SQLite to client-server database management systems like PostgreSQL or MySQL to highly horizontally scalable systems like Apache Cassandra. Database systems will be more specifically discussed in subsection 5.3.1.

According to *Logging and Log Management*, the main advantage of using a database system as a repository for logs is the speed and ease of querying specific log records, required by a user. All database systems provide programmatic interface for easy development of front-end tools manipulating stored data. Furthermore, many database systems already provide additional

[1] Previous paragraphs inspired from [30]

[2] https://dev.mysql.com/doc/internals/en/binary-log-overview.html

[3] https://help.sap.com/viewer/2c2effc99b6746019aeb1af52ad59f5d/1909.001/en-US/44d7cb202e6338d3e10000000a1553f7.html

[4] https://docs.informatica.com/data-integration/data-services/10-2/administrator-guide/log-management/log-management-overview.html

functionality, including indexing for full-text search engines (either using a library like aforementioned Apache Lucene or a native implementation).

On the other hand, the author states that using a database as a log repository may bring potential risks and disadvantages in comparison to a text-file system. Firstly, its write performance is worse than when using files - "writing data to the database will be significantly slower than writing to a local on disk text file due to network latency, database SQL parsing, index updates, and committing the information to disk."[30] Secondly, database storage has a higher disk space requirements in contrast with simple text files due to database indices.

Although in the context of this thesis and its requirements, the disadvantages of using a database system discussed in the previous paragraph are minimized due to the architecture and the behavior of the application. Mainly, since data would be written to the database asynchronously in a separate web application, possibly slower insertion speed is not an issue because it does not slow down the client application. And secondly, even if disk space requirements are higher for a database, the log data would generally have at most hundreds of megabytes to couple of gigabytes, which would not be an issue.

Log data retrieval speed can also be optimized using well-designed indices. As stated in *Logging and Log Management*: "A critical item will be to define the columns in the database that will be used for daily review or part of common queries for reporting and alerting."[30] Author further recommends to create indices on fields *severity*, *date and time*, *generating host* and *message*.

5.3.1 Database technologies

In this subsection will be discussed relational and non-relational database technologies in accordance to the requirements defined earlier. Specifically, the attributes that the database must have to comply with the requirements are open-source license, embeddability and a full-text search implementation.

Traditional relational database systems have been and still are the most popular data stores in the industry. According to DB-Engines Ranking[5], seven out of ten most popular database systems are relational databases[6]. Though many of these systems do not comply with the requirements as they are proprietary. In the Table 5.1 are listed some of the more popular relational database systems and their compliance with the stated requirements. Only databases that fulfil the open-source license requirement are listed.

Databases can be generally divided into two groups - database systems running as separate server applications using the clien/server model for access, and serverless databases using on-disk or in-memory storage. Examples of databases divided like this can be seen in Table 5.1, where pure database server have a *No* in the *Serverless* column and the latter have a *Yes*. Nevertheless, there are database systems that enable database access using both methods,

[5] https://db-engines.com/en/ranking
[6] As of April, 2020

like Firebird.

From the Table 5.1 it is apparent, that commonly used databases that fit the requirements are SQLite, H2, Derby and HyperSQL. In the context of this thesis, there are no requirements on more advanced features of these databases, therefore comparing them further based on their features is not necessary.

Database system	Serverless	Full-text search
MySQL	No	Yes
PostgreSQL	No	Yes
SQLite	Yes	Yes
MariaDB	No	Yes
Firebird	Yes	No
H2	Yes	Yes, a native implementation and a Lucene implementation
Apache Derby	Yes	Using a Lucene plugin
HyperSQL	Yes	Yes

Table 5.1: Relational databases and their features

NoSQL database systems

Databases that do not use tabular relations or schemas, unlike traditional relational databases are called NoSQL ("not only SQL") databases, "While NoSQL databases have existed for many years, NoSQL databases have only recently become more popular in the era of cloud, big data and high-volume web and mobile applications. They are chosen today for their attributes around scale, performance and ease of use. The most common types of NoSQL databases are key-value, document, column and graph databases."[32]

Key-value NoSQL databases are modeled based on hash table data structures. Each record has a unique key, using which the record can be retrieved. The values do not have to follow any pre-defined schema and they can contain any type of data. Both key and its value are treated as simple byte arrays. According to DB-Engines, the most used key-value stores are Redis and Amazon DynamoDB, which has a proprietary license. Nevertheless, key-value stores are not very suitable for storing application logs due to its data structure. Logs would have to have a generated unique key and operations like sorting or querying logs based on their attributes require more implementation overhead, in comparison to traditional relational database systems, which have these functions already implemented.

Document-oriented NoSQL database systems use unstructured documents as a model for storing inserted data. They can be treated as a special type of key-value stores because they maintain the key-value structure internally in the documents (usually they use JSON format, but XML is possible as well). "Document databases are designed for flexibility. They aren't typically forced to have a schema and are therefore easy to modify. If an application requires the ability to store varying attributes along with large amounts of

data, document databases are a good option."[32] The most widely used document-oriented database is MongoDB, which is also one of the most used database systems in general. Other commonly used systems are for example Couchbase or CouchDB.

Document-oriented systems, thanks to their structure and more advanced querying options, are much more suitable as a log repository than simple key-value stores. Logs can be stored as separate documents and have a variable structure, which is an advantage over relational database systems, which have a rigid structure. Furthermore, all document-oriented systems provide an API for querying data, which usually corresponds to classic SQL[7]. Though for compliance with the stated requirements, such a system would have to be executable in an embedded mode. Document-oriented database systems, able to be embedded in a Java applications are for example Couchbase Lite, OrientDB (also a graph database) or Nitrite.

(Wide) column "models enable very quick data access using a row key, column name, and cell timestamp. The flexible schema of these types of databases means that the columns don't have to be consistent across records, and you can add a column to specific rows without having to add them to every single record."[32] Wide columns can therefore be described as two dimensional key-value stores. Although their structure is suitable for storing log data, there are no database systems complying with requirements in this context due to the fact that is no serverless wide column store. Among the most widely used are Cassandra or HBase, which are also discussed more in the next section.

Last basic type of NoSQL database systems is a graph-based system. "In graph theory, structures are composed of vertices and edges (data and connections), or what would later be called 'data relationships.' Graphs behave similarly to how people think—in specific relationships between discrete units of data. This database type is particularly useful for visualizing, analyzing, or helping you find connections between different pieces of data. As a result, businesses leverage graph technologies for recommendation engines, fraud analytics, and network analysis."[32] This type of database may be suitable for example when logging interconnected events, for which mutual relationships are the most important attribute, thanks to its performance when traversing a well-connected graph. Examples of such database systems are Neo4j or JanusGraph.[8]

5.4 Cloud and other distributed solutions

Once databases store too much data, they become slower and require more disk space and computational power. To solve issue with storing too much data on one machine, many companies use distributed data stores or file

[7] An example of such API for MongoDB can be seen here: https://docs.mongodb.com/manual/tutorial/query-documents/

[8] Paragraphs describing different types NoSQL databases inspired from [32]

systems. Distributing the computational load to more nodes in a network increases the query speed when handling large amounts of data.

The most popular distributed file system is a part of Apache Hadoop, which "is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models."[31] According to its documentation, it uses MapReduce programming model for aggregating data from large horizontally scaled systems. "Hadoop shares many of the advantages of a traditional database system. Hadoop allows for the quick retrieval and searching of log data rather than using platform-specific query tools on each system. Hadoop scales well as data size grows by distributing search requests to cluster nodes to quickly find, process, and retrieve results."[30] Furthermore, Hadoop is fault-tolerant thanks to the data replication across multiple nodes, so that even if one node fails, data is still available from other nodes.

On top of HDFS, there is a vast array of data stores that structurize the usually unstructured data stored in Hadoop. Among the most popular are distributed data stores like Cassandra, HBase or Hive. Although these databases are usually schema-less (i.e. NoSQL), in the context of log persistence they share most of their advantages and characteristics with traditional database systems. The major difference between these technologies is the volume of data they are able to handle.

Lastly, in recent years, an increasingly popular alternative to using a local instance of Hadoop in form of cloud data stores from companies like Amazon, Google or Microsoft, has emerged. While using a cloud data store greatly simplifies the implementation and reduces the infrastructural requirements of having a local instance of Hadoop, they have big disadvantages: "Most notable the time to write data to an in the cloud solution will be significantly higher as the data is no longer in close proximity to many of the hosts generating logs. Also, organizations with long log retention periods may find that the costs are greater due to the growing storage size and long-term storage period."[30]

5.5 Elastic Stack

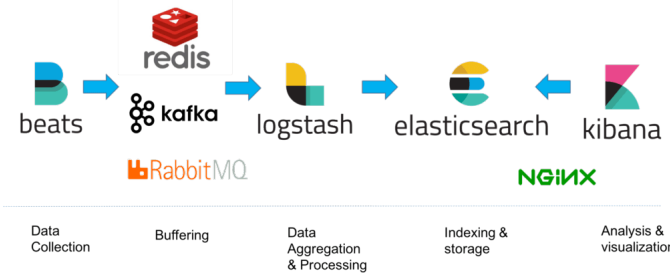


Figure 5.1: An example of Elastic Stack data pipeline [33]

Elastic Stack is an industry standard for centralized logging, providing everything from collecting raw data, normalizing it, indexing it, storing it and

visualizing it using Kibana. In the Figure 5.1 is an example of a data pipeline, used by many organizations not only to collect, store and visualize logs, but many other kinds of data as well. Although it does not fit the use case and requirements in this thesis, it is worth mentioning because it is an already existing solution implementing most of the required functionality in this thesis.

5.6 Choice of technology

In previous sections were presented all possible ways of storing logs, ranging from simple flat text files to large cloud repositories. Due to limitations set by the non-functional requirement NF7, there should not be a change to architecture of Manta, i.e. no new external application dependencies like database servers. This means that the log repository will have to be embedded in the Admin UI web application.

In regards to the stated requirements, an H2 database was chosen as the log repository for following reasons.

Firstly, an H2 database is already used in Manta for other use cases and therefore using H2 will not introduce any new dependencies.

Secondly, it fulfills the requirement NF1 requiring a fast querying of the log repository. Storing data in a database is much more efficient in achieving this than using files and does not require implementing logic for writing and reading from binary files. Although database systems have a drawback in comparison to files in having slower write performance, in this case it is not a large issue, as was discussed previously.

And lastly, this database is along with SQLite an industry standard in embedded, file-based database systems and has an active support.

Generally, a database choice is often a matter of company preference, when more database systems fit their requirements. In this case, H2 would not be the only system achieving that (in chapter 5 were mentioned other systems that fit the requirements).

Chapter 6

Realization

This chapter describes the implementation of the prototype and reasons for the choices that were made in regards to the implementation. At the beginning will be described the development process which was used for designing and implementing this prototype. Then will be described tools that were used during the implementation and design of the new software architecture. After that will be described implementations of each of the logical layers.

6.1 Development process

Work on this thesis was done using the waterfall methodology of software development, which is an official development methodology in the company. Professional Java explains this methodology thus: "The Waterfall methodology consists of a series of activities separated by control gates. These control gates determine whether a given activity has been completed and would move across to the next activity. The requirements phase handles determining all of the software requirements. The design phase, as the name implies, determines the design of the entire system. Next, the code is written in the code phase. The code is then tested. Finally, the product is delivered."[29]

6.2 Other technologies used in the realization

Spring Framework

Spring Framework is a Java framework implementing the inversion of control principle. It not only provides modules for dependency injection, but also for developing web applications (Spring MVC), transaction management, authentication and testing. In this implementation will be used core Spring for dependency injection, Spring MVC for web application backend, Spring JMS for implementing message consumer, Spring Transaction for transaction management, Spring JDBC and Spring Test for testing.

MyBatis

"MyBatis is a first class persistence framework with support for custom SQL, stored procedures and advanced mappings. MyBatis eliminates almost all of the JDBC code and manual setting of parameters and retrieval of results. MyBatis can use simple XML or Annotations for configuration and map primitives, Map interfaces and Java POJOs (Plain Old Java Objects) to database records."[34] It is a framework already used in Manta for mapping Java methods to stored procedures.

Liquibase

Liquibase is a library for database version control. It is already used in Manta for this purpose.

Javapoet

JavaPoet[1] is a library providing API to generate Java source code files. It is used for code generation in the logging layer.

6.3 Software architecture

In this chapter, the software architecture of the new logging solution will be discussed, in accordance to the requirements discussed in subsection 2.2.3, section 2.3 and section 2.4. Specifically, requirements dealing with the architecture of the new solution are F1, F2 and F3.

In the Figure 6.1 (appendix Figure C.2 in higher resolution) is the deployment diagram of the proposed solution.

At the beginning, logs are generated in the applications using the Logging API, which is a part of the new logging framework. Logging API is a wrapper to the Log4j 2 logging framework, similarly to logging facades like SLF4J. Logging API then handles the logs, based on their type, injects placeholders and sends them to appenders. Logs will still appended to text files in the same way as the previous solution. Text files provide certainty that no logs will be lost, in case of lost connection to the Admin UI web application. Manta CLI application should not depend purely on Admin UI with its logging because firstly, not all customers will use Admin UI and secondly, viewing logs in Log Viewer is a less important use case of Manta and its possible failure should not greatly affect the main use cases (having no logs at all would be an issue).

Logs from all Manta applications will be sent via SSL-secured TCP connection to embedded ActiveMQ Artemis messaging broker in JSON format. SSL in Manta uses an internal logic for accessing keystore and truststore files and their passwords, in the prototype there will be a prototypical keystore and truststore on the classpath.

[1] https://github.com/square/javapoet

Admin UI will have an embedded ActiveMQ Artemis instance collecting messages from all connected clients. The amount of possible clients sending their logs is not limited. Message consumer will listen on the logging message queue and retrieve the logs. Logs will be then parsed from JSON and sent to services handling business logic of log persistence. After that will be logs persisted in an embedded H2 database using DAO layer.

The front-end of the Log Viewer (design and implementation of which is not a part of this thesis) component will be implemented as a part of the Admin UI front-end. A user will work with the application using a web browser, from which will be sent requests using REST API. Using requests it will be possible to retrieve logs from the repository, filter them, group them using dynamic filters from the front-end or do a full text search over messages. Furthermore, it will be possible to request packages containing logs from the repository.

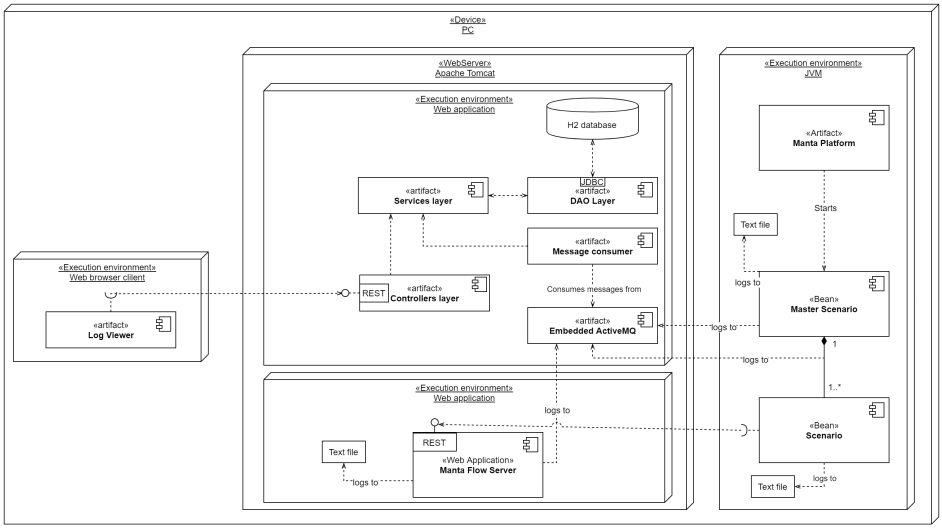


Figure 6.1: Deployment diagram of the new logging solution

6.4 Logging layer

This section describes realization of functional requirements F4, F4.1, F4.2 and F4.3 while keeping in mind non-functional requirements NF2 and NF3.

6.4.1 Categorization

In order to enable effective querying, filtration and grouping of errors in log viewer component, an extendable categorization of errors needs to be designed and implemented. Errors are grouped into categories with a common theme. Each error has a set of optional and non-optional attributes in accordance to F4.2. The common terms in categorization context will be thus defined as:

- **Error type** is a declaration of one error that can occur during an application run.
- **Category** is a group of error types that share similar characteristics.

6.4.2 Error type attributes

Each error type has a set of optional and non-optional attributes. Requirement F4.2 states, that errors should be logged dually, i.e. it should have a message for a normal user and for a developer or a more advanced user. Furthermore, according to F4.3, each error should have a possible solution that can resolve it. Each error type has these attributes:

- **Name of the error** sets the name of the error that will appear in the log viewer. It is an optional parameter and normally will be generated by the logging framework.
- **User message** is a user friendly message describing the error without technical details that have no use for non-advanced user.
- **Technical message** is a detail-heavy, clear message intended for customer support, developers or testers describing exactly what happened with necessary details.
- **Solution** is a possible way of resolving this error.
- **Lineage impact** is an optional parameter describing how much the error affects the resulting lineage. Allowed values are constants from enumeration (Script, Statement, etc.), default value is UNDEFINED (it cannot be said with absolute certainty that an error has no impact on the lineage).
- **Error severity** describes the severity of the error - WARN, ERROR or FATAL. This attribute is used further in the logging API to make logging methods *error()*, *warn()* and *fatal()* redundant.

6.4.3 Logging API

The above functionality is incorporated into the existing API as a wrapper of the Log4j 2 framework. New functionality will be added to logs with severity WARN and above, while the implementation of logs with severity INFO and below will remain the same. It is due to the fact that these logs provide context to the errors and it is not necessary to categorize them.

Non-functional requirement NF2 states that the API should be simple to use and the only possible way to log errors. In order to make logging errors as simple as possible, it is necessary to minimize the amount of files that needs to be amended and minimize the amount of text that the developer needs to write. Making it too difficult would have negative consequences of a lesser amount of logged errors which would make supporting the product even harder.

Error type declaration in source code

```
@Error(
    userMessage = "Data source file was not found.",
    technicalMessage = "Data source file was not found
        while accessing a metadata repository."
    solution = "Check db connection string.",
    severity = ErrorSeverity.ERROR
)
public void fileNotFound(){}
```

Listing 6.1: Example of an error declaration

Error type is defined as an annotated Java *method*, where the annotation parameters are the attributes of the error type. The implementation is making use of the builder design pattern where the logging event is built using fluent API. Error type methods are declared and implemented in classes, which represent *Categories*.

Parametrized logging in error types

The error type declared in the Listing 6.1 loses the option to insert parameters into logged messages. Parametrized logging is however something that is used in a vast majority of logs, it is therefore necessary to have this functionality available. The most user-friendly way of implementing the insertion of parameters is by adding further methods in the logging fluent API. It is possible to generate these methods using annotation processors. The annotation processor parses the user message, technical message and solution and generates a new builder class, reserved for that particular error type. The error method only needs to return the builder and pass the category itself to the builder. Error type declaration is then redefined to:

```
@Error(
    userMessage = "Data source file %{filename} was not
        found.",
    technicalMessage = "Data source file %{filename}was not
        found while %{action}."
    solution = "Check db connection string in
        %{configurationfile}.",
    severity = ErrorSeverity.ERROR
)
public FileNotFoundBuilder fileNotFound(){ return new
   FileNotFoundBuilder(this); }
```

Listing 6.2: Error declaration with parametrized logging

Annotation processor will then, during compilation, generate the following builder:

```
public class FileNotFoundBuilder extends GenericBuilder {
    private ErrorTypeData data;
```

```
    public FileNotFoundBuilder filename(Object arg){
        data.putArg("filename", arg.toString());
        return this;
    }

    public FileNotFoundBuilder action(Object arg){
        data.putArg("action", arg.toString());
        return this;
    }

    public FileNotFoundBuilder configurationfile(Object
       arg){
        data.putArg("configurationfile", arg.toString());
        return this;
    }
}
```

Listing 6.3: Generated builder from Listing 6.2

6.4.4 Builder generation

Error builders are generated by an annotation processor during compile time. Annotation processors can be described as plugins of Java compilers. "The compiler locates the annotations of the source files. Each annotation processor is executed in turn and given the annotations in which it expressed an interest. If an annotation processor creates a new source file, the process is repeated. Once a processing round yields no further source files, all source files are compiled."[35] In the Listing 6.4 is a snippet of the annotation processor implementation generating error builders. In this case, annotation processor looks for methods annotated with *@Error* annotation. Then, it parses placeholders of the messages, which are parameters of the annotations, using regular expressions. Processor will then create a new error builder class, which will extend a generic builder holding functionality common to all builders, and implement methods for all placeholders. All code generation logic is implemented using the JavaPoet library.

```
@SupportedAnnotationTypes("")
public class BuilderProcessor extends AbstractProcessor {
    ...
    public boolean process(Set<? extends TypeElement>
        annotations, RoundEnvironment roundEnv){
        for (Element annotatedElement :
            roundEnv.getAnnotation(Eroor.class)){
            Error error =
                annotatedElement.getAnnotation(Error.class);

            // get package name, class name, initialize
                class builder etc.

            List<String> placeholders =
                MessageParser.parsePlaceholders(error);
```

```
            placeholders.forEach(placeholder -> {
                MethodSpec placeholderMethod =
                    MethodSpec.methodBuilder(placeholder)
                     .addModifiers(Modifier.PUBLIC)
                     .addParameter(Object.class, "value")
                     .returns(returnType) // this
                     .addStatement("putArg(\"$L\", value)",
                         placeholder)
                     .addStatement("return $L", "this")
                     .build();
                generatedClassBuilder
                     .addMethod(placeholderMethod);
            });

            ...
            // add constructor
            ...
            // create new source file
        }
        return true;
    }
    ...
}
```

Listing 6.4: Code generation

Usage

In the Listing 6.5 is a code snippet showing usage of the logging API described in the previous paragraphs. Firstly, logger is initialized as any other Java object, then a normal error is logged, then an error emitted from an exception and lastly a basic log with level INFO, implementation of which stays the same.

```
//Initialization of the logger
Logger logger = new Logger(getClass());

//Logging an error
logger.log(Categories.IOErrors().fileNotFound()
    .filename(myFileName)
    .action("accessing metadata")
    .configurationfile("settings.xml"));

//Logging an exception
try{
    // do something that may throw an exception
} catch (Exception e){
    logger.log(Categories.IOErrors().fileNotFound()
    .filename(myFileName)
    .action("accessing metadata")
    .configurationfile("settings.xml")
```

```
    .catching(e));
}

//Logging INFO
logger.info("INFO and below stay the same.");
```

Listing 6.5: Usage of the logging API

Declaring a new error type requires only implementing a method and annotating it in the appropriate category class.

Holder of categories

In the Listing 6.5 is further called class *Categories* that holds methods returning the concrete categories classes. It is a user-defined class that may extend some globally defined categories and their errors. For example, in a larger project, there may be a module, that would just hold all of the errors that may occurr anywhere in the project. In a concrete module, a developer would then extend this class and add categories specific to that particular module. All common development environments would then offer the developer all categories as a hint, after calling the holder class and the developer would not have to remember or check the names of the error categories.

6.4.5 Logging API UML model

In the appendix Figure C.3 there is an UML representation of the logging API and the underlying framework described in the previous section. In the *Common artifact* frame is the functionality expected to be implemented from a user - that is *Categories* class and *ConcreteCategory* class. *ConcreteCategory* is any kind of error category class. The annotation processor generates the *ConcreteErrorTypeBuilder*, which resides in the same package as the category that declares that error type.

The *Logging framework* frame describes the underlying logging framework and all of the functionality that happens in the background. It includes all of the abstract classes, *Logger* which is a wrapper to the Log4j 2, annotation processor and other helper classes.

6.4.6 Logging context

The functional requirement F5 states, that a part of the logging API should be an interface providing functionality of assigning additional context to logs. Logging context will enable more filtering parameters of logs from the repository. Since the parameters have to be set beforehand, to enable filtering from the front-end they are set to:

- **workflow_execution_id**, which is a unique identifier of an application run (application run is defined as a set of executed Manta scenarios). The ID is provided by a different Manta component.

- **scenario_execution_id**, which is a unique identifier of a Manta scenario execution. The ID is as well provided by a different Manta component.
- **technology** is an attribute of a scenario with the name of the technology that is being scanned, having its data lineage created or being exported to (for example 'Oracle' or 'IGC').
- **connection_id** is the name of the connection to a database or other technology that is used during a scenario execution.
- **phase** represents the phase of data lineage creation that the scenario is a part of (extraction, analysis or export to a third-party technology).
- **scenario** represents a normalized name of a scenario (e.g. 'Oracle DDL Analysis').
- **context** is a dynamic context, which differentiates the logs on the lowest level. For example, based on SQL scripts.

The above attributes are stored in the database as columns and enable more specific filtering. An example use case may be: A user runs extraction and analysis on their Oracle database. To see if any errors occurred, they set the *technology* parameter to *Oracle* and see all logs emitted from *Oracle* scenarios in one screen.

The logging context API is implemented using the ThreadContext feature of Log4j 2, which servers exactly for this purpose of adding additional context to logs. The API for developers is shown in Listing 6.6. Context is set on a per-thread basis, where child threads inherit the context of the parent thread.

```
LoggingContext.setWorkflowExecutionId(workflowExecutionId);
LoggingContext.setScenarioExecutionId(scenarioExecutionId);
LoggingContext.setTechnology(technology);
LoggingContext.setConnectionId(connectionId);
LoggingContext.setPhase(phase);
LoggingContext.setScenario(scenario);

Logger logger = new Logger(getClass());

// logged without context
logger.log(Categories.testErrors().testError());

// all logs from here will belong to 'someFile.sql'
LoggingContext.setContext("someFile.sql")

// logged with context
logger.log(Categories.testErrors().testError());

// logged without context by using flag for overriding the
    context
logger.log(Categories.testErrors().testError(), true);
```

```
// all logs after this will be again without context
LoggingContext.clearContext();
```

Listing 6.6: Logging context API

6.5 Transport layer

As was discussed in section 4.5, the chosen transport layer technology is Apache ActiveMQ Artemis. It will be implemented as an embedded messaging broker in Admin UI. This section will describe that implementation, as well as the necessary Log4j appender that will append the logs to the remote broker. It is the implementation of the functional requirement F3.

6.5.1 Log4j 2 appender

In chapter 3 were described all existing Log4j 2 appenders. For communication with ActiveMQ Artemis, the ideal and inteded appender to use is *JMSAppender*. Using a logging configuration with such an appender requires only one XML file to set up, as can be seen in Listing 6.7. The JMS appender has the following compulsory parameters:

- **name** is the name of the appender, by which it can be referred to
- **destinationBindingName** sets the name of the destination queue (*dynamicQueues* prefix allows JMS to connect to queues programatically without having a JNDI configuration file, where JMS looks up these properties by default)
- **factoryBindingName** is the name of the JMS connection factory
- **providerURL** sets the URL of the remote broker, along with parameters for secure connection using SSL
- **factoryName** is a class that provides JMS initial context

In the child tag of *JMS*, is set the layout in which logs will be sent. Log4j 2 provides JSON layout using which logs will be logged in JSON format. The JSON log by default includes all of the internal log attributes, like for example thread ID, log message or the timestamp in epoch seconds. It is possible to further include the whole ThreadContext map (by using *properties="true"*, as is in the listing below) as well as add custom JSON key-value pairs, using *KeyValuePair* tag. In the implementation, there are two custom fields - one is for determining the type of log (ERROR, which are logs with log level WARN, ERROR or FATAL, or DEBUG, which have log level TRACE, DEBUG or INFO) used for object mapping the JSON log after it is received in the consumer, and the application name. Application name is hardcoded in the configuration, because each application has its own Log4j 2 configuration. Excluded from this listing is the File appender because the transport layer is not dependent on it.

```
<Configuration>
  <Appenders>
    <JMS
      name="jms"
      destinationBindingName="dynamicQueues/loggingQueue"
      factoryBindingName="ConnectionFactory"
      providerURL="tcp://localhost:61616?sslEnabled=true
                   &trustStorePath=truststore.ts
                   &trustStorePassword=password"
      factoryName="org.apache.activemq.artemis
                   .jndi.ActiveMQInitialContextFactory">
      <JsonLayout properties="true" stacktraceAsString="true">
        <KeyValuePair key="type" value="$${marker:}"/>
        <KeyValuePair key="application" value="Manta CLI"/>
      </JsonLayout>
    </JMS>
  </Appenders>
  <Loggers>
    <AsyncLogger name="eu.profinit.manta" level="trace">
        <AppenderRef ref="jms"/>
    </AsyncLogger>
  </Loggers>
</Configuration>
```

Listing 6.7: Log4j 2 configuration with JMS appender

6.5.2 Implementation

The transport layer abstracts the functionality of consuming logs, parsing them and passing them to DAO layer, that is a part of the persistence layer. The following section will describe the implementation up to that point.

In the center of the implementation is an embedded instance of Apache AciveMQ Artemis server, which is implemented as a Spring bean. In the prototype, the server listens on port *61616* for trusted connections that use its SSL certificate. Producer (the logging API) sends logs to the server using a JMS appender from Log4j 2 logging framework. At the other end *LogListener* listens on the designated logging message queue, retrieves the incoming logs, parses them in from JSON using *LogResolver* and passes them to the handlers which are specific based on the log type. Handlers then pass the logs to the persistence layer from which they are written to the database.

In the Figure 6.2 are modeled components of the transport layer and their dependencies. All of the following components are Spring-managed beans. The components are namely:

- **Embedded ActiveMQ Artemis** is the embedded instance of the messaging broker, implemented as a Spring bean, communicating with the logging API using the TCP protocol
- **LogListener** is the message consumer that listens on the designated logging queue, consumes the logs and passes them to *handlers*

- **LogResolver** maps the logs in JSON format from String data type to POJO using Jackson object mapping library
- **MessageHandler** is a class handling logs depending on their *type*, implementing the visitor design pattern
- **LogHandler** is the concrete visitor for ERROR and DEBUG logs, whose function is to pass the logs to the persistence layer

Message handlers implement the visitor design pattern to provide simple scalability for other possible types of logs that may be wanted in the future. More specific descriptions of these components can be read in the attached JavaDoc.

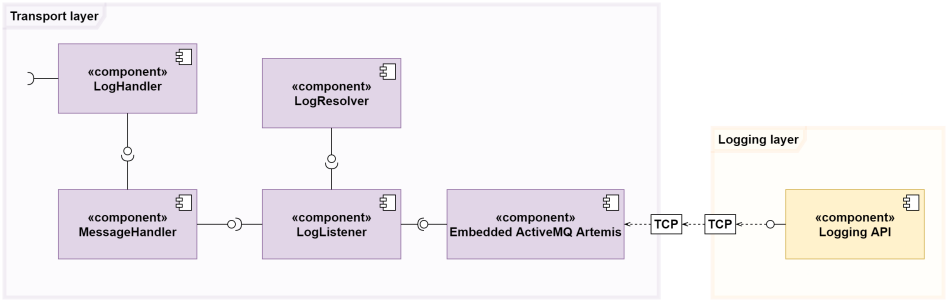


Figure 6.2: Component diagram of the transport layer

6.6 Persistence layer

The following section will describe the implementation of the persistence layer, i.e. the functional requirements F2, F6, F7, F8 and F9. As was stated in the section 5.6, the chosen repository is an H2 file database.

6.6.1 Database schema

In the appendix Figure C.4 is the database schema of the log repository. Database is designed in accordance to the queries that are going to be ran against it. In the center of the design is a *logs* table that stores all logs that were sent to the Log Viewer. It stores messages for DEBUG logs and metadata for all logs. Error messages are stored in *error_definitions* table in order to keep the size of the database smaller. Most of the main queries regarding errors that were emitted by any application are against the *error_metadata* table, which stores the aggregate information about errors that occurred in a particular context. For example, error that occurred during a DDL Analysis of Oracle database "OracleTestDB" has its own row, as well as an error that occurred during a DDL Analysis of Oracle database "OracleProductionDB". This table has a realistic upper limit of *(number of scenarios * amount of all possible errors * number of connections)* which is not a large number for database queries. This will therefore enable very fast queries regarding any

possible errors that occurred during an application run or were emitted from any application.

All of the database tables are namely:

- **workflow_execution** table stores all of the application runs that currently have logs stored in the database. It is defined by a unique workflow execution ID that is passed in the logging context. Each workflow execution has at least one scenario execution.
- **scenario_execution** table stores all of the scenario executions that have logs stored in the database. It always belongs to a particular workflow execution and is defined by a scenario execution ID that is passed in the logging context.
- **connections** table stores all connections that were used during a particular scenario execution. Since *connection_id* is not a unique identifier (e.g. 'OracleTestDB' can be used in multiple scenario executions), it has a database generated ID.
- **uploaded_packages** table stores metadata regarding all uploaded packages to the database
- **logs** table stores all logs that were sent to the repository and their metadata
- **error_metadata** table stores distinct errors that occurred in a particular context (combination of *workflow_execution_id*, *scenario_execution_id*, *conn_db_id*, *package_name* and *error_definition_id* is always unique)
- **error_definitions** table stores all error messages and other error metadata that is provided from the Logging API

6.6.2 Indices

In section 5.3 was recommended that if logs are stored in a relational database, then it is suitable to have indices on fields *severity* (*type*), *date and time*, *generating host* (*application*) and *message*. Even though in this case are these columns called differently, the recommendation can still be followed, since these fields are some of the most used in queries. For example, date and time is stored in two columns - *epoch_second* and *nano_second*, therefore they will be stored in a composite index. The order is specified for queries providing log context, i.e. logs that happened immediately before and after the requested log. Other indices are based on the general queries that will be run against the database, mostly for filtering errors from separate executions. Lastly, full text search index using Apache Lucene engine is set on fields *user_message* and *technical_message*. In the Listing 6.8 is the SQL script creating Lucene indices for full text search.

```
-- lucene full text search index
CREATE ALIAS IF NOT EXISTS FTL_INIT FOR
    "org.h2.fulltext.FullTextLucene.init";
CALL FTL_INIT();

-- create index on error messages
CALL FTL_CREATE_INDEX('PUBLIC', 'ERROR_DEFINITIONS',
    'USER_MESSAGE,TECHNICAL_MESSAGE');
```

Listing 6.8: Creation of index for full text searching

6.6.3 Components

In the appendix diagram Figure C.5 are displayed components of the persistence layer. The implementation has a multi-tier architecture with data layer containing DAO interfaces and MyBatis mappers, service layer containing business logic and presentation layer containing controllers (not pictured here because it is discussed in the following section).

DAO components are designed on a per-table basis with additional DAO components serving a single specific use case. Data access objects provide an abstraction over underlying database mechanism, making it loosely coupled with other layers, allowing developers to change the data store without greatly affecting the business and upper layers.

Database queries are executed using persistence framework MyBatis. MyBatis provides greater flexibility in building and generating prepared SQL statements. In the prototype, queries are stored in the SQL files and mapped to interfaces, which are then used by the DAO components. DAO components are namely:

- **WorkflowExecutionDAO** provides basic DAO interface for queries over *workflow_executions* table,
- **PackageDAO** provides interface for queries over *uploaded_packages* table,
- **ScenarioExecutionDAO** provides interface for queries over *scenario_executions* table,
- **LogDAO** provides interface for queries over *logs* table,
- **ConnectionsDAO** provides interface for queries over *connecitons* table,
- **ErrorDefinitionDAO** provides interface for queries over *error_definitions* table,
- **ErrorMetadataDAO** provides interface for queries over *error_metadata* table,
- **RepositoryMetadataDAO** provides interface for helper queries regarding the the general information about the repository, such as which distinct applications have logs stored in the repository,

- **PackageUploadDAO** provides interface for queries and H2 methods facilitating uploading support packages to the repository, database dumps are uploaded using H2 *CSVREAD* method,
- **PackageCreateDAO** provides interface for queries and H2 methods facilitating creation of support packages, database dumps are created using H2 *CSVWRITE* method,
- **PruneCheckingDAO** provides interface for helper queries regarding consequences of changing pruning settings (e.g. if a user wants logs to be deleted after one day and there are logs in the repository that are older than one day, then these methods will provide information regarding the amount of logs deleted)

Above the DAO layer is a serivce layer handling business logic of all operations of Log Viewer. It creates and uploads packages, builds log context or checks constraints of incoming logs. Services are namely:

- **LogPersistenceService** implements *write* logic for logs. After it receives a new log to write, it checks whether the logs is a part of a new workflow, scenario execution or connection and if it is, insert the new executions to appropriate tables. After that will be the log inserted.
- **CreatePackageService** creates a zip file containing a dump of logs from the database for a single *connection* used in a given *scenario_execution* or *application*, a JSON file containing metadata of the requested scenario and a dump of relevant *error_definitions* that are used in the log package.
- **UploadPackageService** inserts the logs from the given database dump to the database along with its *error_definitions*.
- **PruneService** handles all logic regarding scheduling automatized pruning of all logs. For that it uses the *TaskScheduler* from Spring framework. The scheduled time is calculated by adding the user-set prune expiration time (2 days by default) to *execution_timestamp* of a workflow execution. Furthermore, it checks once an hour for expired logs that are not a part of any workflow execution.
- **ForcePruneService** deletes logs from the repository as a result of a request from a user.
- **UpdateSettingsService** handles all changes to settings that a user may make from the front-end and their consequences. For example, if a user changes prune expiratation time, it changes the internal property representing the expiration time and reschedules scheduled prunings.
- **MetadataProviderService** handles all requests regarding repository metadata.

- **FullTextSearchService** handles full text search over *user_message* and *technical_message* of error logs.
- **ExpandLogsService** provides concrete error logs for a given *error type* to a user who requested them.
- **LogQueryService** provides error metadata filtered according to parameters passed by a user.
- **LogContextService** builds context of a requested log. Context of a log are logs that are emitted immediately before and after the requested log in the same thread.

6.7 Rest API endpoints and controllers

Communication with the front-end is facilitated using REST API, which is documented in Swagger in the attached CD. The API serves only for internal purposes and is not intended to be for integrations with other third-party solutions, therefore it is modeled mainly according to the needs of the Log Viewer application. API is mainly structured around the main execution types - *workflow_executions*, *scenario_executinos*, *connecions* or separate *uploaded packages* - which are represented as path variables. Filtering options are passed as request parameters. It is possible to filter logs by *application* (user chooses applications from which the logs should be returned), *error categories* and *error types*. In the diagram Figure C.6 is a component diagram of the entire Log Viewer architecture, including the presentation layer containing controllers providing the REST endpoints.

6.8 Prototype implementation

The prototype is implemented as a separate web application distributed as a *war* archive. It can be run in any Java servlet container like Apache Tomcat. The prototype also provides a simple web page front-end used for demonstration purposes of the back-end functionality. The web page is implemented using HTML, CSS, JavaScript and the jQuery library.

Chapter 7

Testing

All implemented functionality of the prototype, except the front-end, which only serves for demonstration purposes, is covered in tests.

Logging API is tested for correct behavior of helper classes, among which are for example *MessageParser*, which reads provided messages from annotation parameters and passes them to the *BuilderProcessor*, or *MessageFormatter*, which inserts the provided placeholders into the messages using regular expressions. Generated builders are tested for the correct amount of methods and whether they have correct names. Example of such test is in the Listing 7.1. Reflection is used for retrieving the methods of the generated builder. Using *getDeclaredMethods* method enables to check only the methods that were generated with the builder without inherited methods.

```
@Test
public void testTwoParams(){
    TwoParamsBuilder twoParamsBuilder = twoParams();
    Method[] methods =
        twoParamsBuilder.getClass().getDeclaredMethods();
    assertEquals(3, methods.length);

    List<String> methodNames = Arrays.stream(methods)
    .map(Method::getName)
    .collect(Collectors.toList());

    assertTrue(methodNames.contains("catching"));
    assertTrue(methodNames.contains("param1"));
    assertTrue(methodNames.contains("param2"));
}

@Error(
    userMessage = "%{param1}",
    technicalMessage = "%{param2}",
    solution = "",
    severity = Severity.ERROR
)
public TwoParamsBuilder twoParams(){
    return new TwoParamsBuilder(this);
}
```

Listing 7.1: Test of a generated builder

Each layer of the Log Viewer application is tested separately, except the presentation layer containing Log Viewer controllers which is tested in integration tests. A test instance of H2 database with test data is used for testing as well as other test features provided from the *Spring framework*.

Transport layer is tested for correct behavior when receiving different types of logs, correctness of object mapping from JSON format to POJO. In these tests, *LogPersistenceService* is mocked using Mockito test framework.

In DAO layer, CRUD methods of each component are covered by unit tests. *UploadPackageDAO* uses resource files that are in a correct format in order to test correct behavior of reading the packages. All tests are *@Transactional*, which means that each transaction from one test is reverted after the test in order to keep the test database in the same state as before the tests.

Service layer components are tested separately with required dependencies from the DAO layer being mocked. This approach allows to test the behavior of the service component separately without possible issues originating in another layer affecting the test. All service components that implement its own logic are tested. Service components that only call the DAO layer are tested in integration tests.

Lastly, integration tests are used to verify the correctness of behavior of the application as a whole. These tests send mock requests to controllers using *MockMvc* from *Spring framework* and verify whether the response is correct. No components are mocked in these tests, in contrast to the tests of the service layer, as they mainly test the communication between components. Example of an integration test verifying a correct response to a query requesting error metadata from all workflow executions is in the Listing 7.2. All possible incorrect inputs are verified to return an error response.

```
@Test
public void integrationTest() throws Exception {
    // insert a log to the test db
    ErrorLog log =  getErrorLog("WE1", "SE1",
        "OracleTestDB");
    logPersistenceService.persistErrorLog(log);

    MvcResult mvcResult = this.mockMvc
            .perform(get(baseUrl)
            .param("offset", "0"))
            .andExpect(status().isOk())
            .andReturn();

    List<ErrorDTO> response = objectMapper.readValue(
        mvcResult
        .getResponse()
        .getContentAsString(),
        new TypeReference<List<ErrorDTO>>(){}
    );

    assertEquals(1, response.size());

    ErrorDTO errorDTO = response.get(0);
```

```
        assertEquals(log.getCategory(),
            errorDTO.getErrorCategory());
        assertEquals(log.getErrorType(),
            errorDTO.getErrorType());
        assertEquals(log.getUserMessage(),
            errorDTO.getUserMessage());
        assertEquals(log.getTechnicalMessage(),
            errorDTO.getTechnicalMessage());
        assertEquals(log.getSolution(), errorDTO.getSolution());
        assertEquals(log.getLineageImpact(),
            errorDTO.getImpact());
        assertEquals(log.getLevel(), errorDTO.getLogLevel());
        assertEquals(log.getApplication(),
            errorDTO.getApplication());
    }
```

Listing 7.2: Integration test for log queries

All tests of the Log Viewer web application prototype are in the *manta-bachelor-thesis-log-viewer-test* module, tests for the Logging API are stored in the same module as the API. Even though the implementation is a functional prototype, its testing is thorough, containing in total 150 test cases with additional tests of the logging API.

Chapter 8

Conclusion

The output of this thesis is a design of a logging API providing flexibility in declaring errors and injecting parameters to error logs that are emitted from any Manta application and a design and implementation of the back-end of Log Viewer web application. Both of these designs were implemented as prototypes. These prototypes were then tested using unit and integration tests. The prototypes can be used for logging in any Java program and presenting those logs in the Log Viewer application with all of its required functionality. The prototype fulfills the requirements of the contractee in full.

At the beginning of the thesis were analyzed business and system requirements of the logging API and Log Viewer. The main aim of the solution was to create a framework for logging, which would be simple to use for developers yet would provide a greater amount of detail and options for filtering for the end user, which was the main issue with the previous solution. Since the volume of logs produced from Manta is very large and difficult for people to analyze, especially for people unfamiliar with the system, troubleshooting and dealing with issues was very difficult for users and they often needed professional help.

After an analysis of issues regarding logging in Manta and requirements on the new solution stemming from them is a research of already existing solutions and discussion regarding their possible usage and their compliance with the set requirements. The analytical part of the thesis is divided into three separate logical layers, each dealing with one part of the issue. First was the *logging layer*, in which were analyzed logging frameworks of Java programming language and their features. Second was the *transport layer* analyzing technologies facilitating a transport of logs to a centralized storage from multiple sources. And the last chapter of the analytical part of the thesis was the *persistence layer* analyzing existing solution for log storage and their compliance with the requirements. The new logging solution is built on Log4j 2 logging framework, Apache ActiveMQ Artemis messaging broker and H2 database.

In the implementation section of the thesis was described the implementation of the prototypes of the logging API and Log Viewer back-end. This section also follows the same logical structure as the analytical section of the work, dividing the implementation into three logical layers. The logging API

implementation is largely based on the Java source code generation using annotation processors, which greatly simplifies the work of developers when defining new errors and thus motivating them to be more thorough at it, which then naturally enhances the quality of the output for the end user. Log Viewer prototype is a web application built on Spring framework.

As was stated at the beginning of the thesis, this is a solution specifically tailored to the needs and requirements of Manta. Implementation choices are greatly limited which disallowed the use of many of the commonly used log analysis systems, such as Elastic stack. The resulting technology and design choices are therefore not universal for all Java applications, except the logging API, which, although also implemented according to Manta specifications, provides a possible universal approach for logging errors in greater detail.

The chosen technologies for the Log Viewer would vary a lot depending on the architecture of the required system. For example, an ideal solution for a large enterprise application running in one environment would be to use industry established choices of Kafka and Elastic stack.

Bibliography

[1] GAMMA, Erich. *Design patterns: elements of reusable object-oriented software*. Reading, Mass.: Addison-Wesley, c1995, pp.185. ISBN 0201633612.

[2] *SLF4J* [online]. [cit. 2020-01-11]. Available from: http://www.slf4j.org/

[3] JUNEAU, Josh. *Java 9 Recipes: A Problem-Solution Approach*. Third edition. New York, NY: Springer Science+Business Media, 2017, pp.232. ISBN 978-1484219751.

[4] *Log4j Bridge* [online]. [cit. 2020-01-11]. Available from: http://www.slf4j.org/legacy.html

[5] *Logging Dependencies in Spring* [online]. [cit. 2020-01-11]. Available from: https://spring.io/blog/2009/12/04/logging-dependencies-in-spring/

[6] GÜLCÜ, Ceki. Think again before adopting the commons-logging API [online]. [cit. 2020-01-11]. Available from: http://articles.qos.ch/thinkAgain.html

[7] *Java Logging Technology* [online]. [cit. 2020-01-11]. Available from: https://docs.oracle.com/javase/6/docs/technotes/guides/logging/

[8] *Java.util.logging* [online]. [cit. 2020-01-11]. Available from: https://docs.oracle.com/javase/8/docs/api/java/util/logging/package-summary.html

[9] *Handler (Java Platform SE 8)* [online]. [cit. 2020-01-11]. Available from: https://docs.oracle.com/javase/8/docs/api/java/util/logging/Handler.html

[10] *Java TM Logging Overview* [online]. [cit. 2020-01-11]. Available from: https://docs.oracle.com/javase/6/docs/technotes/guides/logging/overview.html

[11] *Uses of Class java.util.logging.Level (Java Platform SE 8)* [online]. [cit. 2020-01-11]. Available from: https://docs.oracle.com/javase/8/docs/api/java/util/logging/class-use/Level.html

[12] *Log4j – Overview* [online]. [cit. 2020-01-12]. Available from: https://logging.apache.org/log4j/2.x/manual/index.html

[13] *Log4j - Log4j 2 Appenders* [online]. [cit. 2020-01-11]. Available from: https://logging.apache.org/log4j/2.x/manual/appenders.html

[14] *Welcome to Apache Flume — Apache Flume* [online]. [cit. 2020-01-11]. Available from: https://flume.apache.org/

[15] *Overview of the JMS API - The Java EE 6 Tutorial* [online]. [cit. 2020-01-11]. Available from: https://docs.oracle.com/javaee/6/tutorial/doc/bncdr.html

[16] *Apache Kafka* [online]. [cit. 2020-01-11]. Available from: https://kafka.apache.org/intro

[17] *ZeroMQ | Get started* [online]. [cit. 2020-01-11]. Available from: https://zeromq.org/get-started/

[18] *Level (Apache Log4j API 2.13.0 API)* [online]. [cit. 2020-01-11]. Available from: https://logging.apache.org/log4j/2.x/log4j-api/apidocs/index.html

[19] *Log4j – Log4j 2 Thread Context* [online]. [cit. 2020-01-11]. Available from: https://logging.apache.org/log4j/2.x/manual/thread-context.html

[20] *Log4j – Log4j 2 Lock-free Asynchronous Loggers for Low-Latency Logging* [online]. [cit. 2020-01-11]. Available from: https://logging.apache.org/log4j/2.x/manual/async.html

[21] *Logback Home* [online]. [cit. 2020-01-11]. Available from: http://logback.qos.ch/

[22] CURRY, Edward. *A Coordination Approach for Self-Managed Middleware*. Galway, 2006. Dissertation thesis. National University of Ireland, National University of Ireland, Galway, Faculty of Science, Department of Information Technology.

[23] *Point-To-Point Messaging (Sun Java System Message Queue 4.3 Technical Overview)* [online]. [cit. 2020-04-04]. Available from: https://docs.oracle.com/cd/E19316-01/820-6424/aerbj/index.html

[24] *Publish/Subscribe Messaging (Sun Java System Message Queue 4.3 Technical Overview)* [online]. [cit. 2020-04-04]. Available from: https://docs.oracle.com/cd/E19316-01/820-6424/aerbk/index.html

[25] *What are Message Brokers? | IBM* [online]. [cit. 2020-04-04]. Available from: https://www.ibm.com/cloud/learn/message-brokers

[26] *Getting Started with Java Message Service (JMS)* [online]. [cit. 2020-04-04]. Available from: https://www.oracle.com/technical-resources/articles/java/intro-java-message-service.html

[27] RICHARDS, Mark, Richard MONSON-HAEFEL, David A. CHAPPELL and Richard MONSON-HAEFEL. Java message service. 2nd ed. Sebastopol, CA: O'Reilly, c2009, pp.10. ISBN 978-0596522049.

[28] NARKHEDE, Neha, Gwen SHAPIRA and Todd PALINO. Kafka: the definitive guide : real-time data and stream processing at scale. Sebastopol, CA: O'Reilly Media, 2017, pp.XIII, 12-13. ISBN 978-1491936160.

[29] RICHARDSON, W. Clay. Professional Java JDK 6 edition. Indianapolis, IN: Wiley Technology Pub., c2007, pp.82. ISBN 978-0-471-77710-6.

[30] CHUVAKIN, Anton, Kevin J. SCHMIDT, Chris PHILLIPS and Patricia MOULDER. Logging and log management: the authoritative guide to understanding the concepts surrounding logging and log management. Amsterdam: Elsevier/Syngress, 2013, pp.71-84. ISBN 978-1597496353.

[31] *Apache Hadoop* [online]. [cit. 2020-04-12]. Available from: https://hadoop.apache.org/.

[32] *NoSQL Databases Explained | IBM* [online]. [cit. 2020-04-12]. Available from: https://www.ibm.com/cloud/learn/nosql-databases.

[33] *The Complete Guide to the ELK Stack | Logz.io* [online]. [cit. 2020-04-12]. Available from: https://logz.io/learn/complete-guide-elk-stack/.

[34] *mybatis - MyBatis 3 | Introduction* [online]. [cit. 2020-04-13]. Available from: https://mybatis.org/mybatis-3/.

[35] HORSTMANN, Cay S. Core Java: Volume II–Advanced Features. 10th edition. Boston, MA: Prentice Hall, 2016, pp.476. ISBN 978-0134177298.

[36] Confluent Propels Data Architecture into Event Streaming Era with $125 Million Series D | Business Wire [online]. [cit. 2020-04-10]. Available from: https://www.businesswire.com/news/home/20190123005240/en/Confluent-Propels-Data-Architecture-Event-Streaming-Era

[37] Configuring Transports · ActiveMQ Artemis Documentation [online]. [cit. 2020-04-10]. Available from: https://activemq.apache.org/components/artemis/documentation/latest/configuring-transports.html

Appendices

Appendix A

Abbreviations

API - application programming interface

SQL - structured query language

CLI - command-line interface

UI - user interface

MVC - model-view-controller

IT - information technology

SSL - secure sockets layer

TLS - transport layer security

UML - unified modeling language

IoC - inversion of control

NoSQL - not only SQL

HTTP - hypertext transfer protocol

JDBC - Java database connectivity

JMS - Java Messaging Service

JPA - Java Persistence API

SMTP - simple mail transfer protocol

BSD - Berkeley Software Distribution

RFC - Request for Comments

MOM - message oriented middleware

FIFO - first in, first out

JVM - Java Virtual Machine

DVD - digital versatile disc

RDBMS - relational database management system

JSON - JavaScript Object Notation

XML - extensible markup language

HDFS - Hadoop Distributed File System

POJO - plain old Java object

DAO - data access object

REST - representational state transfer

URL - uniform resource locator

TCP - transmission control protocol

DDL - data definition (or description) language

CRUD - create, read, update and delete

Appendix B

CD content

- `Bachelor thesis` — root folder of the CD
  - `Documentation` — documentation directory
    - `Javadoc` — aggregate Javadoc for all modules
    - `Swagger` — Swagger docs for the REST API
    - `Dependencies` — generated list of libraries and their licenses
  - `Source code` — folder containing source code
    - `manta-bachelor-thesis-logging-api`
    - `manta-bachelor-thesis-log-viewer`
    - `manta-bachelor-thesis-log-viewer-test`
    - `manta-bachelor-thesis-proof-of-concept`
    - `pom.xml` — parent pom
  - `Thesis` — folder containing all thesis related documents
    - `Diagrams` — folder with all diagrams
    - `thesis.pdf` — the thesis itself

Appendix C

Diagrams

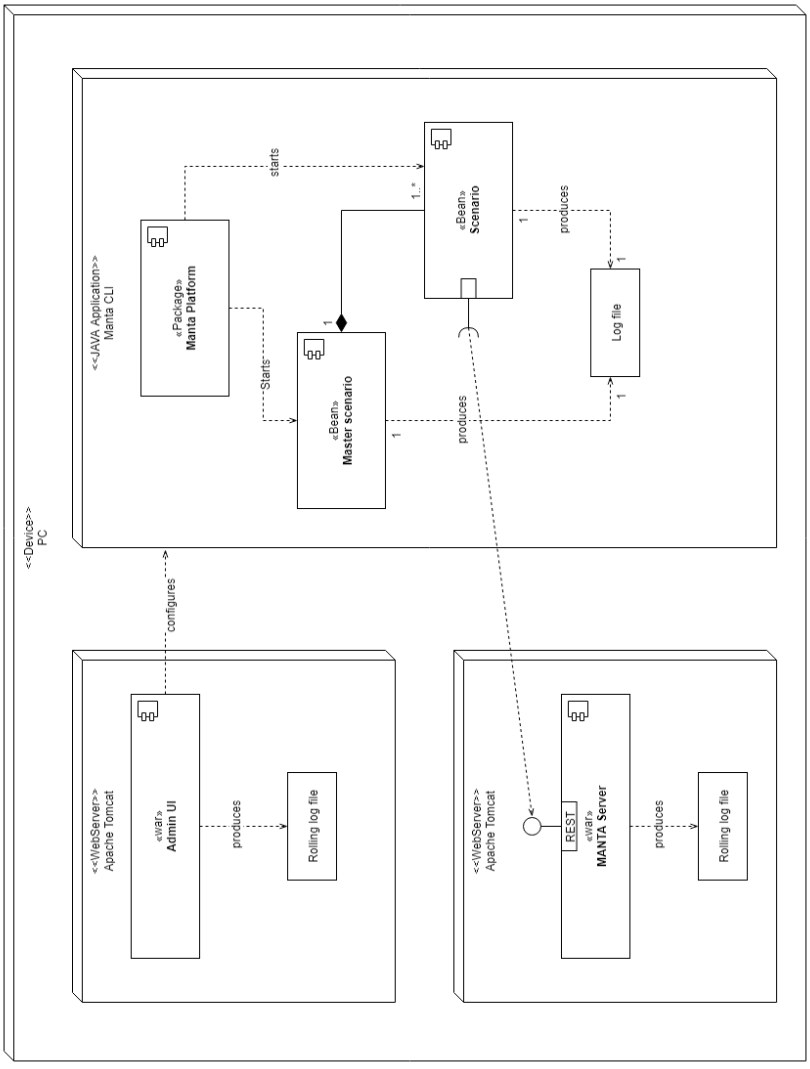


Figure C.1: Architecture of the new logging solution

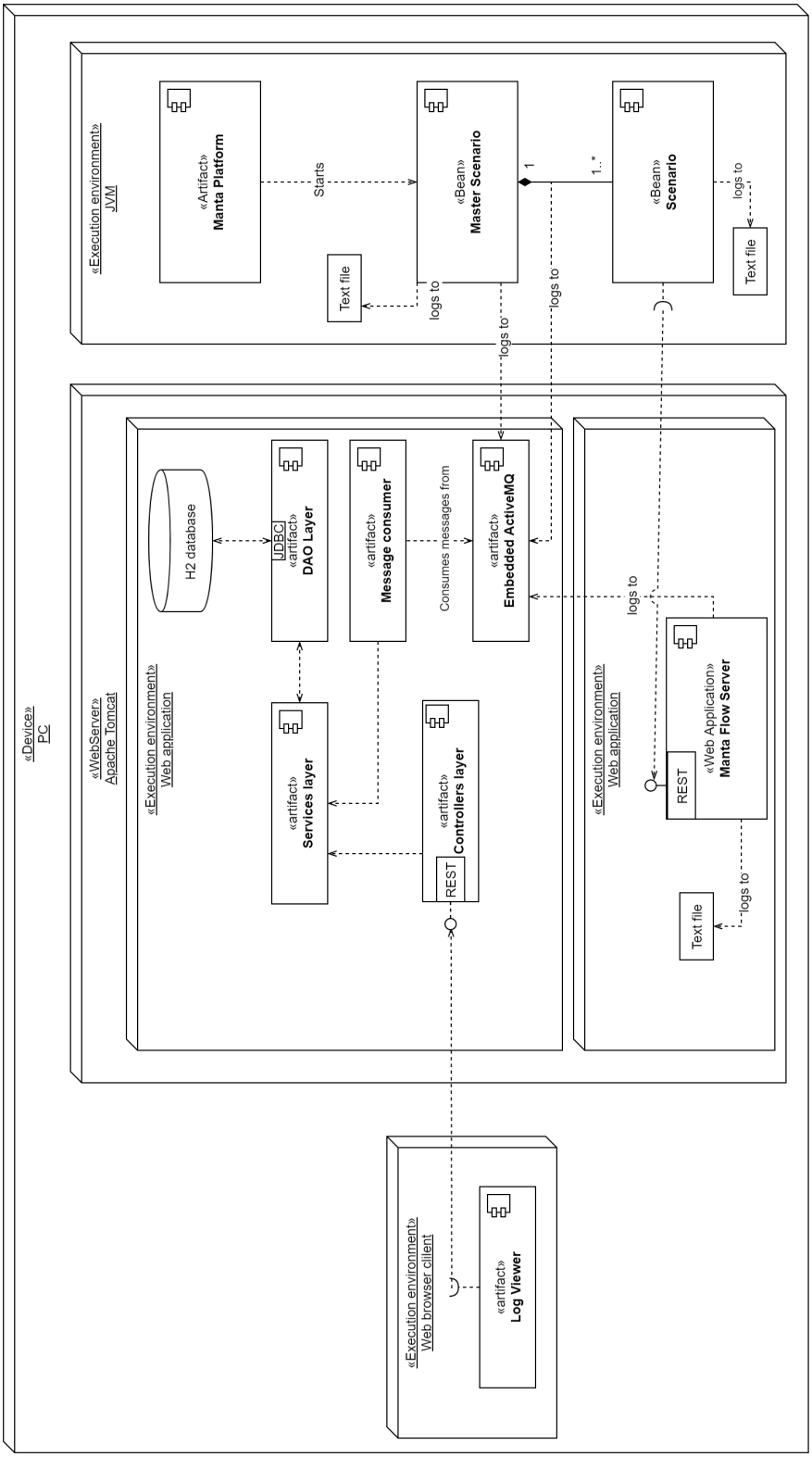


Figure C.2: Architecture of the new logging solution

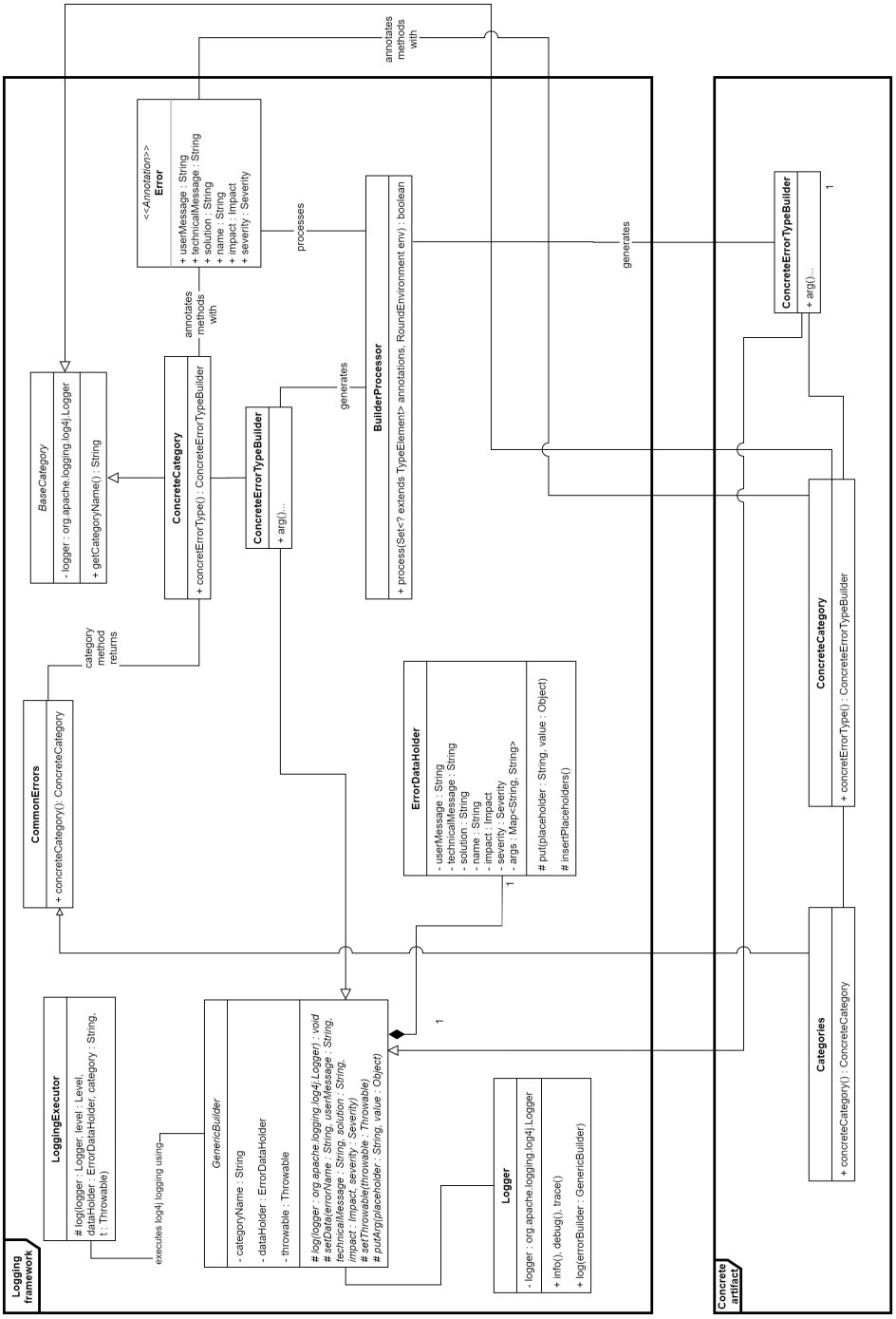


Figure C.3: UML of the designed logging API and the underlying framework

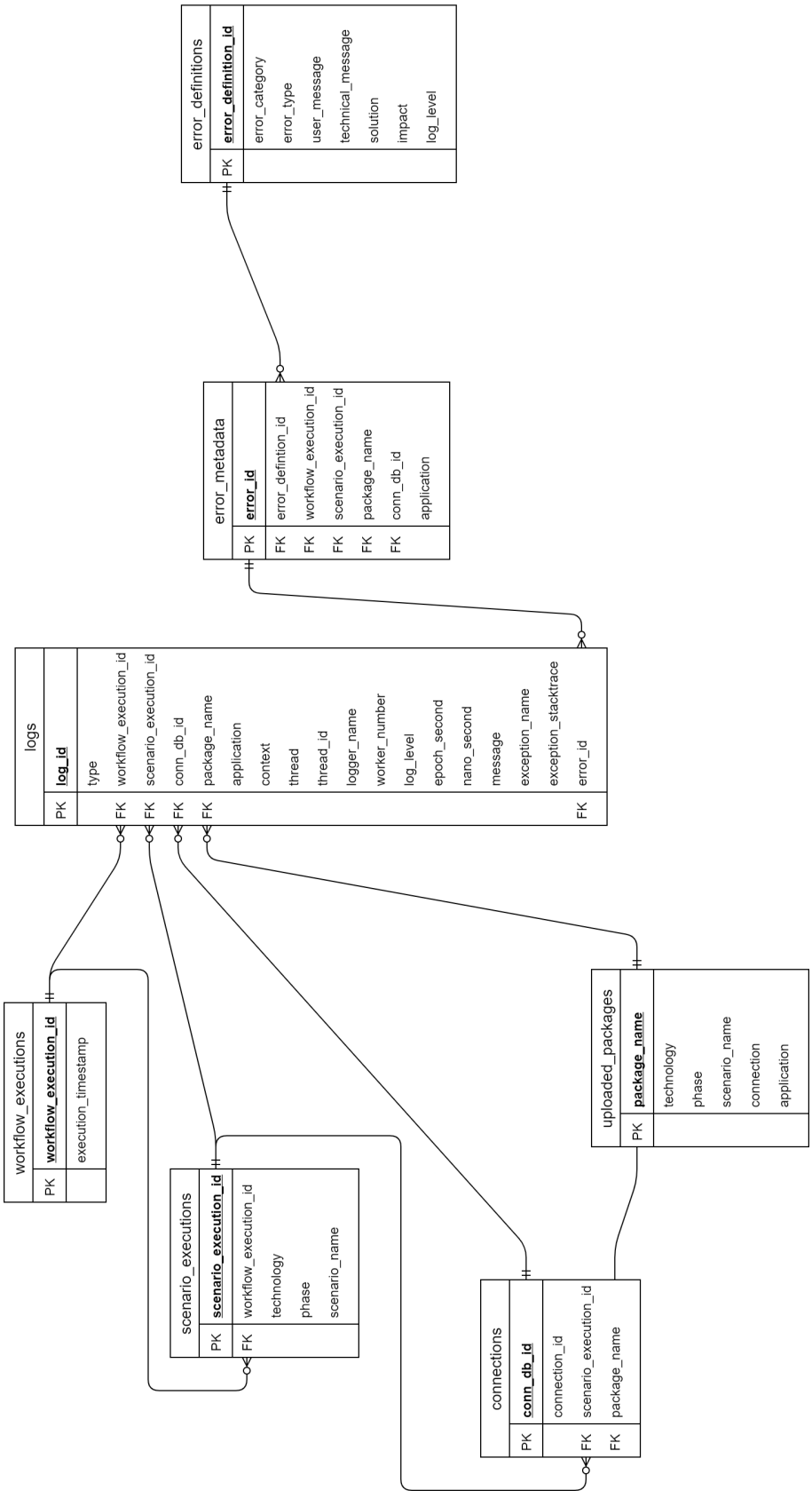


Figure C.4: Database schema of the log repository

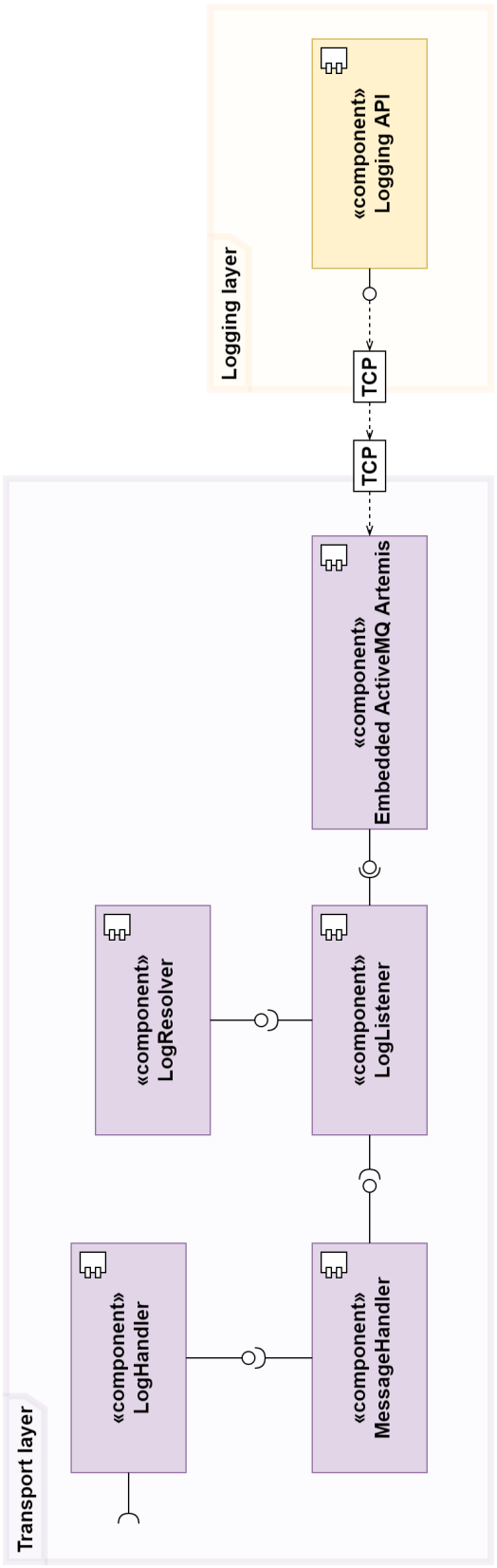


Figure C.5: Components of the transport layer and logging layer

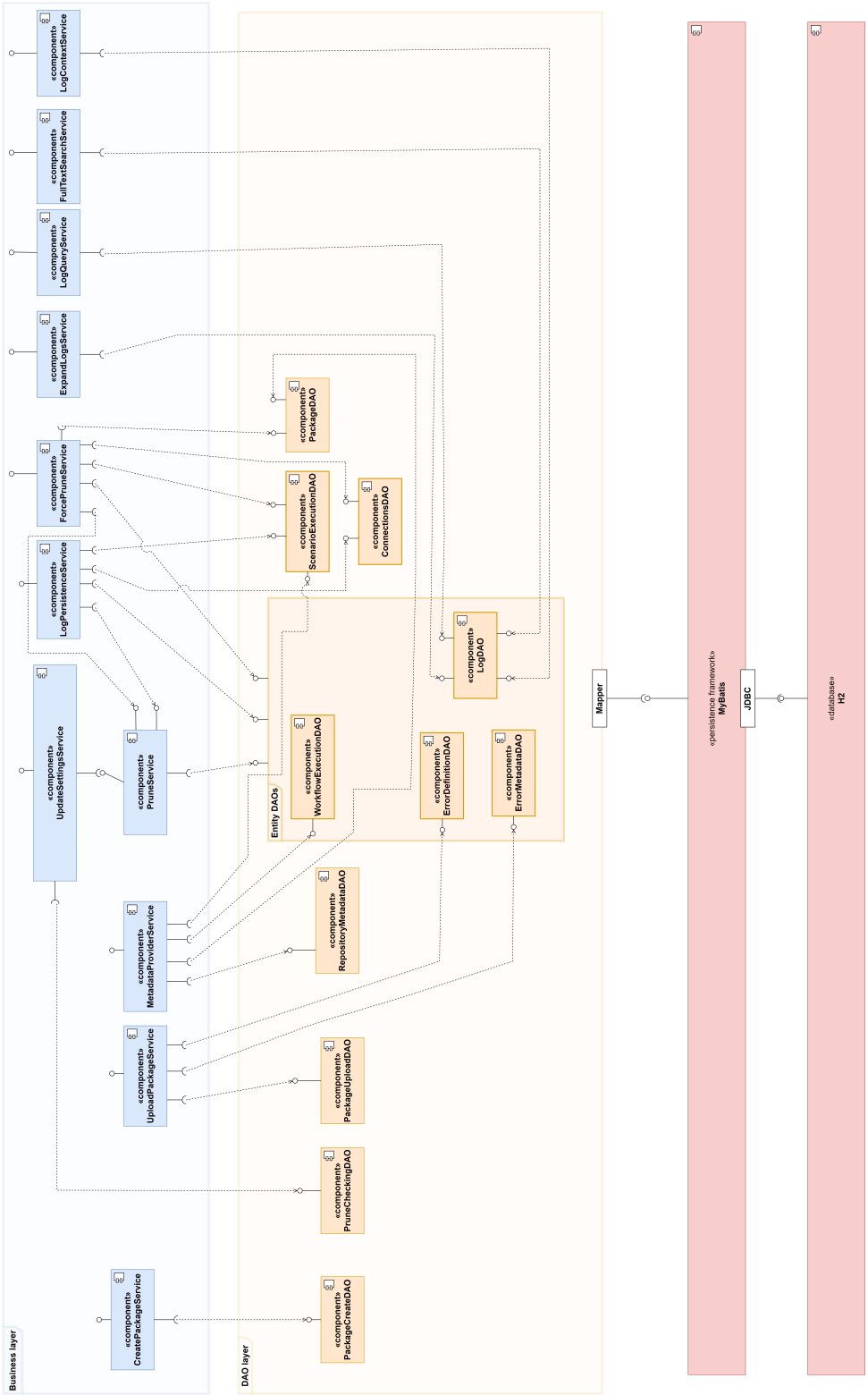


Figure C.6: Components of the persistence layer

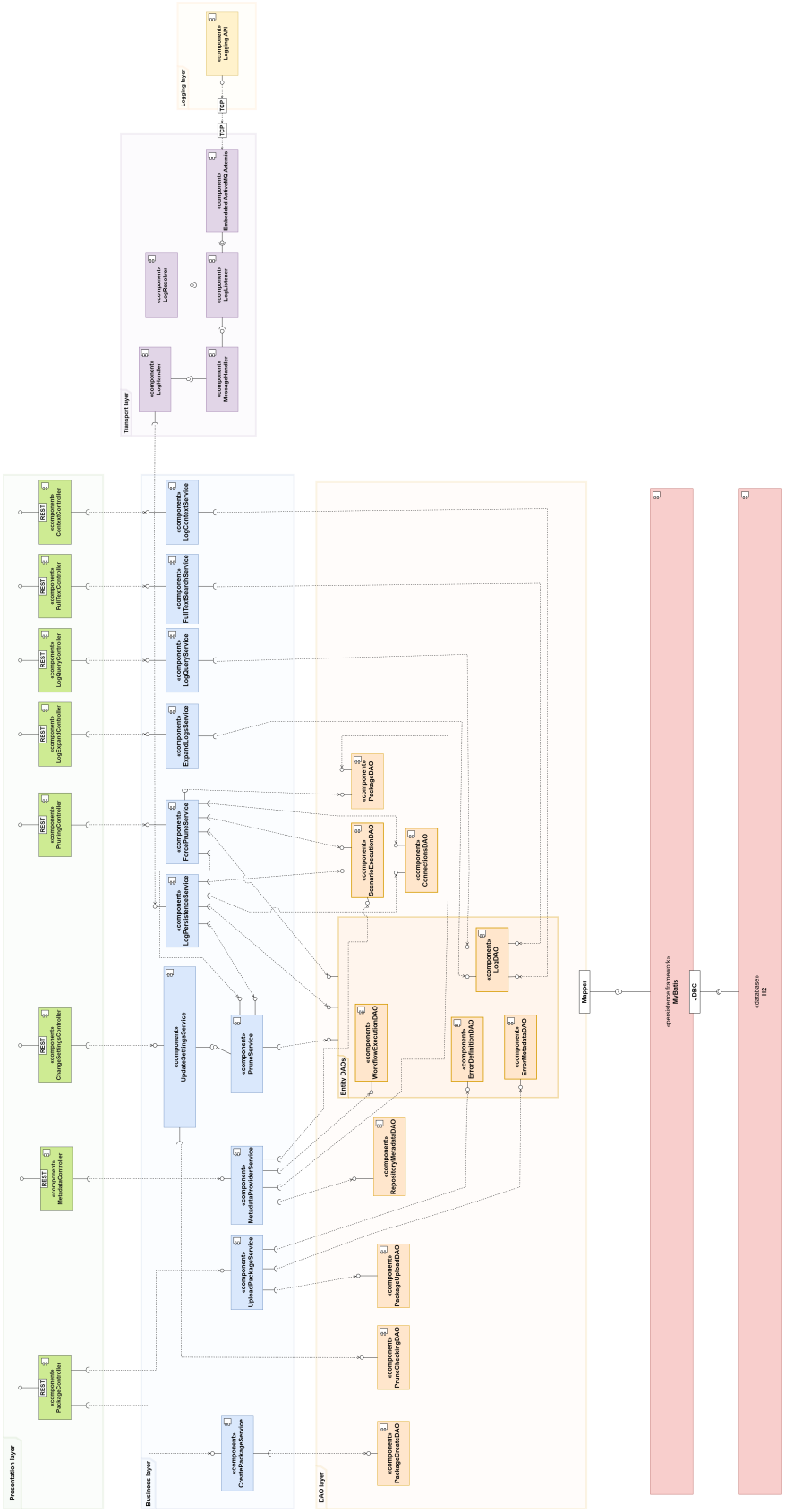


Figure C.7: Component diagram of the whole Log Viewer backend