Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

A Software Architecture for Graph-based Metadata Management for CAE Data

Tobias Boceck

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Supervisor:	Julian Ziegler, M.Sc.
Commenced:	March 31, 2020
Completed:	November 25, 2020

Abstract

This thesis provides an overview of different software architectural styles and how they can be used to create a software architecture for a graph-based metadata management for CAE data. Use cases are created to extract software architecture requirements which are the basis for the discussion of the architectural concepts. A concept architecture is presented for each of the software architectural styles. Those are used to discuss the advantages and disadvantages of the architectural styles in the context of the defined software architecture requirements. The REST architectural style and Microservices architectural style both provide the best trade-offs in this context. A final software architecture is introduced that uses both of the architectural styles. This software architecture has been implemented as a prototype which is validated by creating a validation plan for each use case where each step of these plans is mapped to requests of the prototype.

Contents

1 Introduction **15**

2 Fundamentals **17**
2.1 Data and Metadata . . . 17
2.2 Computer Aided Engineering . . . 18
2.3 Data Lake . . . 18
2.4 Graph Database . . . 19
2.5 Ontology . . . 20

3 Data Management for CAE Data **21**

4 Software Architecture Requirements and Use Cases **25**
4.1 Use Cases . . . 25
4.2 Requirements . . . 29
4.3 Challenges . . . 31

5 Software Architecture **33**
5.1 Client-Server Architecture . . . 35
5.2 Service Oriented Architecture . . . 45
5.3 Representational State Transfer . . . 50
5.4 Microservice Architecture . . . 54
5.5 Evaluation of Architectural Styles and Final Architecture . . . 63

6 Architecture Implementation **67**
6.1 Components . . . 67
6.2 Hypermedia-Driven Prototype . . . 79
6.3 Build Automation . . . 80
6.4 Testing . . . 81
6.5 Documentation . . . 82
6.6 Continuous Integration . . . 82

7 Validation **85**
7.1 Environment . . . 85
7.2 Preparation . . . 85
7.3 Execution and Measurement . . . 87

8 Conclusion and Outlook **91**

A Appendix **93**

Bibliography **113**

List of Figures

3.1 CAE Data and Metadata 21
3.2 Ontology and Instance as a Graph 23
3.3 Graph-based Metadata Management for CAE Data 24

4.1 Use Case: Data Upload and Metadata Annotation 26
4.2 Use Case: Visualization Client 27
4.3 Use Case: Data Transformation for CAE Data Analysis 28

5.1 2-Tier Architecture 37
5.2 3-Tier Architecture 40
5.3 N-Tier Architecture 43
5.4 Service-Oriented Architecture with SOAP and WSDL 47
5.5 RESTful Architecture 52
5.6 Microservice Architecture with MOM 57
5.7 Microservice Architecture with REST 60
5.8 Restful Architecture with Microservices Mindset 65

6.1 Project Modules of the Prototype Implementation 67
6.2 Class Diagram of the User Module 69
6.3 Class Diagram of the Auth Module 70
6.4 Class Diagram of the Data Module 72
6.5 Class Diagram of the Metadata Module 74
6.6 Class Diagram of the Ontology Management Classes of the Ontology Modul . . 76
6.7 Class Diagram of the Ontology Metadata Classes of the Ontology Modul 77
6.8 Class Diagram of the Plugin Module 78
6.9 State Diagram of the Prototype 80

List of Tables

6.1 Security Policy of the REST API Endpoints 71

A.1 Authentication RESTful API 93
A.2 Metadata RESTful API 94
A.3 Ontology RESTful API 96
A.4 Plugin RESTful API 97
A.5 Execution Plan for Use Case 1 107
A.6 Execution Plan for Use Case 2 109
A.7 Execution Plan for Use Case 3 112

List of Listings

6.1 Metadata Node Model as JSON 73
6.2 Metadata Relationship Model as JSON 75
A.1 Sample Java Plugin 98
A.2 Sample Ontology 99

List of Abbreviations

ACID Atomicity, Consitency, Isolation, Durability. 20

AD Architectural Decision. 82

ADR Architectural Decision Record. 82

AI Artificial Intelligence. 20

AMQP Advanced Message Queuing Protocol. 56

API Application Programming Interface. 29

BASE Basic Availability, Soft-State, Eventual Consistency. 20

CAD Computer Aided Design. 15

CAE Computer Aided Engineering. 15

CAT Computer Aided Testing. 15

CD Continuous Delivery. 55

CI Continuous Integration. 55

CRUD Create, Read, Update, Delete. 19

CSV Comma-Separated Values. 28

Data Transfer Object DTO. 68

DC Dublin Core. 18

ETL Extract, Transform, Load. 15

HAL Hypertext Application Language. 79

HATEOAS Hypermedia as the Engine of Application State. 51

HDFS Hadoop Distributed File System. 71

HTML Hypertext Markup Language. 55

HTTP Hypertext Transfer Protocol. 32

HTTPS Hyper Text Transfer Protocol Secure. 32

IasC Infrastructure-as-Code. 63

ID Identifier. 53

JSON Javascript Object Notation. 35

JWT JSON Web Token. 70

LOC Lines of Code. 34

MADR Markdown Architectural Decision Records. 82

MCF Meta Content Framework. 18

MOM Message Oriented Middleware. 56

MVC Model-View-Controller. 34

NISO National Information Standards Organization. 17

OLAP Online Analytical Processing. 19

OLTP Online Transaction Processin. 19

ORM Object-Relational Mapping. 69

OS Operating System. 17

PID Process Identifier. 78

RDF Resource Description Framework. 18

REST Representational State Transfer. 34

RPC Remote Prodecure Call. 44

SOA Service Oriented Architecture. 34

SOAP Simple Object Access Protocol. 46

SQL Structured Query Language. 34

UDDI Universal Description, Discovery, and Integration. 46

UDP User Datagram Protocol. 47

UI User Interface. 36

URI Uniform Resource Identifier. 22

URL Uniform Resource Locator. 53

UUID Universally Unique Identifier. 73

VM Virtual Machine. 85

WSDL Web Service Description Language. 46

XML Extensible Markup Language. 35

XSD XML Schema Definition. 46

1 Introduction

Modern product development uses Computer Aided Engineering (CAE) to help reduce the time and effort to develop physical prototypes. Engineers often use a dedicated computer application for a specific process such as Computer Aided Design (CAD) or Computer Aided Testing (CAT) during product development [PD10]. While a single computer application usually offers analysis functionalities on its own, it is also common to use external CAE computer applications for data analysis [PD10]. However, in contrast to previous decades, the data science analysis methods and resources have changed and huge data sets from all kinds of applications are often used for analysis [GLN+05]. Therefore, many user groups might require access to or produce data that is used during data analysis. The management of these amounts of data requires additional annotation of the data in form of metadata which allows to use the data independently and enables broad data access [GLN+05]. Using external data analysis computer applications might also require multiple Extract, Transform, Load (ETL) steps. This makes it difficult to keep track of the origin of data used in the analysis. Tracing data back to its origin and documenting the generating processes is part of data provenance and a common problem in the scientific field [BKT00].

Ziegler et al. (2020) proposed to store data and metadata in a data lake that consists of a data store for heterogeneous files and a graph database for metadata. The metadata model consists of process, data and metadata nodes which have typed interconnections indicating their relationship. This allows engineers to search for data based on a combination of processes, metadata, and their connections. It also allows to visualize the data flow between processes and helps to trace data back to its origin which is commonly known as data lineage. The concept also includes a class hierarchy with specialization nodes that extend the process and data nodes. This allows data scientists to use inference based on the classes to gain new insights about a specialized set of process or data nodes instead of a single one. [ZRKM20]

However, the graph-based metadata concept also introduces a couple of new challenges due to the inclusion of different distributed data storage systems. Engineers need to use a new storage system for their CAE files and a different store system to keep the files' metadata up to date. Additionally, they need to maintain the graph model by creating nodes in the graph database which correlate to the actual data assets and processes which are stored in the data lake. A product development process also includes mostly non-tech affine user groups who might need access to the data and its metadata but do not know how to send queries to the specific data stores.

Therefore, a unified access and management interface must be created that overcomes these challenges and helps each user group to focus on developing products. The goal of this thesis is to find the best-suitable software architecture to create an interface for the graph-based metadata management by discussing different architectural styles based on use cases and requirements.

This thesis is structured as follows. Chapter 2 introduces the basic notions that are needed to understand the thesis and its context. Chapter 3 explains why a unified interface for the graph-based metadata management for CAE data is needed and what capabilities it needs to provide. In Chapter 4,

use cases for a graph-based metadata management are presented and used to extract software architecture requirements. In Chapter 5, different architectural styles are described and for each style, a concept architecture for the graph-based metadata management is presented. Chapter 6 presents the implementation of a prototype for a selected architecture. Chapter 7 describes how the prototype implementation can be used to fulfill the use cases. Chapter 8 gives an overview of the thesis and concludes the presented concepts.

2 Fundamentals

In this chapter the basic notions will be introduced that are used throughout the thesis. Apart from these notions a basic understanding of software engineering and its terminology is required to understand the concepts of this thesis.

2.1 Data and Metadata

Data in computer science refers to any set of symbols which is stored in a series of bits in a computer storage. It is unorganized and does not provide any information by itself. Data can be created by the developers or the users of a computer based program. Developers declare variables with types such as integer and boolean to allocate memory for a certain type of data. The users of a computer based program can often use such a program to create files which in turn store data. Commonly, user files include data of pictures, sound, video or text. While data itself is unstructured, files can also be used to organize and structure the data. Once organized and structured, data is put into context and information can be obtained.

Metadata is a specialized form of data that is used to describe and manage information represented as data [DHSW02]. According to the National Information Standards Organization (NISO), metadata can be categorized into descriptive, structural and administrative metadata. Descriptive metadata is used to describe data for the purpose of identification and discovery [Ril17]. An example is the filename which Operating Systems (OSs) use to allow the users to find their data. Structural metadata indicates how individual data can be put together to form something new. As an example, large files are often split into smaller junks but to allow putting them back together, the order of the junks and the merge algorithm need to be specified as structural metadata. Administrative metadata is used to organize and manage data [Ril17]. For example the creation date, size and type of file are all administrative metadata. Using metadata to enrich otherwise unorganized data helps to organize, manage and process the data [Ril17]. Search, filter, and sorting functionalities which are frequently used when dealing with data, only work because of the information metadata provides.

Rich metadata is extending the concept of metadata by providing even more details for attributes and their relationships [DRC+14]. It can be used to gain even more insights about its related data. A common example for rich metadata is provenance where a data history as well as its generating processes are stored [DRC+14]. This allows to examine the data output as a result of the previous processes which helps to identify problems with the generated data.

Metadata as well as rich metadata has to be stored and made accessible to be useful. Unfortunately, there is no all-in-one solution and metadata can be stored in files or in a dedicated database. However, it is important to use a file format or database (schema) which allows to model the metadata for a specific use case. There are already numerous standards available for metadata documentation

such as Dublin Core (DC), Resource Description Framework (RDF) and Meta Content Framework (MCF) [NJ04]. However, when storing metadata in databases there are no standards and developers need to either rely on existing scientific approaches or create a custom solution.

2.2 Computer Aided Engineering

CAE is essential in modern product development and describes the use of computer systems during product development. The product development workflow starts once a product specification has been created and is an iterative process. A specified product will be planned and a virtual prototype will be created and simulated using CAE systems. Based on the simulation results a physical prototype can be planned and constructed. The physical prototype will be tested using CAT systems and the test results will be analyzed against the simulation results. The results of the analysis will be used to start the planning phase of a new iteration to improve the prototype. In these processes, CAE systems will be used to design, simulate, analyze and validate virtual or physical product prototypes. During any of these processes, domain specific data is either used as input or produced as output by an engineer using a supporting computer system. [ZRKM20]

These data files are of business value since they either represent a physical product prototype or expose their properties which means they become assets. E.g. CAE assets are 3D-models of physical prototypes, simulation results, test results or properties described as metadata. These assets can be produced and used by different user groups such as engineers and data scientists. The latter are called citizen data scientists if their primary work is not related to statistics or data analysis. Based on their data analysis a product can be optimized and a new iteration of a prototype can be created.

2.3 Data Lake

A data lake is a conceptual construct to store arbitrary data by using low cost technologies typically used within enterprises [Fan15]. It might consist of multiple storage systems such as different database systems but appears as a single storage system to its clients. It is characterized by the ability to store data of any kind and size in its original data format using any import method [Mat17]. The general idea of using a data lake is to have a central data store which can be used for any kind of future use cases [Sul+19]. These are most commonly analytical use cases where more insights about the data and its producer are gained which created the role of the data scientist [Mat17]. Usually, the analysis is done by using the mass storage of data to train statistical models which can be implemented in applications or used in optimization scenarios [Mat17]. To make use of the raw data files, sometimes additional ETL steps might be required but these are not part of the concept of a data lake [Sul+19]. While the role of the data scientist was introduced by the creation of the data lake, it is by no means the only user of the system [Mat17]. Depending on the context of where a data lake is used it can have administrators, developers, data scientists or other users which might need access to the stored data [Mat17].

In this sense it is similar to a data warehouse which is also often found in enterprises but they have a few key differences. Enterprise data warehouses are designed to handle large amounts of data and store them in a homogeneous format. Therefore, additional ETL steps are required to transform

the data into a predefined format defined by a database schema. A data lake on the other hand is designed to handle massive amounts of data by using low cost technologies. The storage of raw data files requires no schema and brings the flexibility for users to decide how to process the data. [Fan15]

As Giebler et al. (2019) noted there are numerous — sometimes contradictory — definitions of the users of a data lake and additional analysis and management capabilities [GGH+19]. However, in this thesis a data lake is a logical construct to store and access any kind of CAE data in its raw format which includes 3D models of physical product prototypes, spreadsheet files that represent simulation and test results.

2.4 Graph Database

A graph consists of nodes that are connected by edges which indicate their relationship and can have a reading direction [RWE15]. Therefore, data entities can be represented as nodes and the way they relate to each other can be represented by relationships. To model data entities as graph representation in a database, a graph model is needed. There are different models for related data such as property graphs, hypergraphs or triples where the former is the most common one [RWE15]. Property graphs use nodes with one or more labels and a set of key-values, called properties to model data entities [RWE15]. The relationships on a property graph are named and can also contain properties but must define exactly one start node and one end node giving them a direction [RWE15]. Hypergraphs are a variation of property graphs with no relationship constraint which means any number of start and end nodes are allowed [RWE15]. Triples originate from the semantic web which describes linked data in a subject-predicate-object structure and is typically described in a RDF format [BHL01].

A graph database allows querying data based on a graph model and provides typical Create, Read, Update, Delete (CRUD) operations to manage its internal data. There are native and non-native graph databases which differentiate in the way they store the exposed graph structure. While native graph databases use a storage system optimized for graph structures, non-native graph databases might use a relational or document-based database. Another difference is in the performance of the underlying processing engine which is significantly better for native graph databases. This is achieved by physically creating pointers between stored nodes which are connected by a relationship in the graph model. Non-native graph databases do not store relationships as data itself and therefore, have to compute them. [RWE15]

The exposed graph structure allows for data exploration and visualization by iterating over the nodes and following its relationships. However, by using a graph model to represent data, algorithms from the graph theory can directly be applied on said data. Therefore, queries for similarity search, path finding or link prediction are sometimes directly provided by native graph databases [NH19]. However, the performance might heavily vary depending on the underlying storage system.

While non-native graph databases might be implemented using a relational database, native graph databases use NoSQL stores and focus on Online Analytical Processing (OLAP) as compared to Online Transaction Processin (OLTP) [NH19]. The difference is that the former describes processing of longer lasting operations where the read operations outnumber the write operations by far, whereas the latter describes processing of large numbers of short living operations with

a focus on data integrity [NH19]. Therefore, native graph databases usually use a more relaxed consistency model such as Basic Availability, Soft-State, Eventual Consistency (BASE) whereas OLTP systems usually have a stricter consistency model such as Atomicity, Consitency, Isolation, Durability (ACID) [RWE15].

2.5 Ontology

The word ontology has been around for as long as the seventeenth century and used to be a branch of philosophy to classify entities in any aspect of life [Smi12]. With the advancement of data intensive research fields such as software engineering, databases and Artificial Intelligence (AI) it also gained traction in the fields of computer science to enable sharing and reusing of knowledge extracted from data [Wel03]. As stated by Christopher Welty (2003) numerous definitions of the word ontology exist in the context of computer science. In this thesis an ontology is seen as something that enables sharing and reusing knowledge by enriching data, metadata, and relationships with semantics. By adding semantics to these entities, the data will become knowledge and additional implicit information can be inferred.

Christopher Welty (2003) states that Ontologies are a common approach when dealing with the meaning of data which is most prominent in databases and AI. While metadata already enriches data, an ontology can be used to further enrich the data as well as the metadata. By adding semantic meaning to data and metadata, new information can be inferred which can help with the management of such data.

As an example, in the CAE context, there might be thousands of files of different types where some files contain simulation results while the rest contain test results. All files are either the result of a simulation process or a test process. In this constellation there is no knowledge about whether the simulation process is semantically equivalent to the test process or if their data files have an equivalent meaning. An ontology can be created to enrich the processes with the missing semantics. The ontology could be defined by creating a new entity which is called process and connecting it with a typed relationship to the simulation process as well as the test process. This allows to infer additional information in a way that every file is part of a common process entity.

3 Data Management for CAE Data

Whenever software programs are involved in development processes, they will create data. Depending on the domain this can include test files, statistical models, 3D models of real-world objects, media files and many more. Usually, files are created to serve a purpose which means the files will be required by the same or different programs and by the same user or other user groups. While a single user already needs to keep track of different versions and properties of his files it becomes even more important and increasingly complex when files are shared among different user groups and originate from different sources. In a corporate environment, data management is an important topic because lots of heterogeneous data can originate from multiple different sources and might need to be shared with many other user groups. Therefore, data management can be seen as the tasks of collecting, storing, providing access and using data. As data by itself is meaningless, annotations in form of metadata can be created to give data a meaning. Therefore, data becomes independent of the producer and can be understood by other consumers. However, sometimes additional metadata information is required by consumers to make use of the data. Therefore, the metadata is an essential part of data and needs to be considered in the data management.

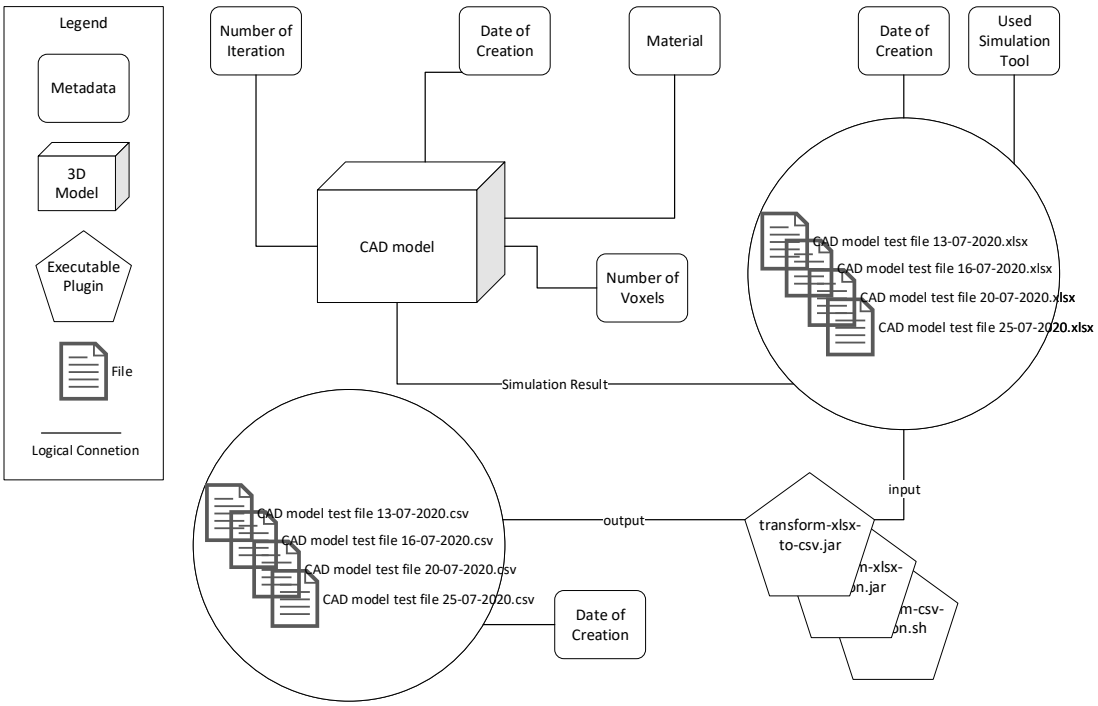


Figure 3.1: An overview about different kinds of data assets and their metadata that also shows their relationships.

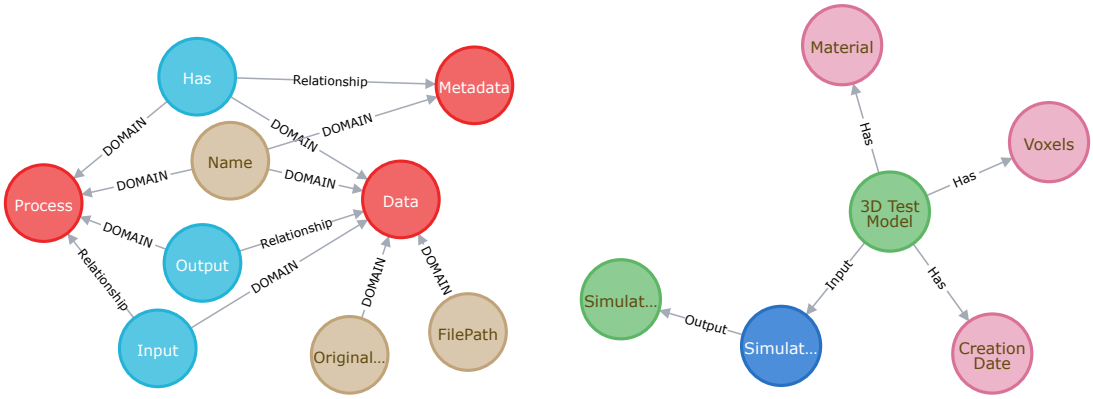
Throughout the product development process, numerous heterogeneous data assets are created. For instance, a product might get designed using a CAD system which outputs a file representing a 3D model of a virtual prototype. The model will be used in CAE systems for simulations which output files containing the simulation results. CAT systems will be used to test a physical prototype which output files containing the test results. The available data will be used to analyze the prototype to optimize it. This often involves transforming existing data into different formats which is automated by using custom executable scripts. However, due to the massive amount of simulation and test data these scripts can run for a long time. Since product development is an iterative process, these systems will run many times using different parameters and input files and producing many output files. Additionally, metadata might be created to enrich the data assets with additional meaning. An example of such metadata is the number of voxels and the material that were used during the modeling of a virtual prototype. Figure 3.1 shows typical CAE data assets, their metadata and transformation scripts involved in an example product development process. It also shows the relationships between the data, metadata, and transformation scripts.

Saving the metadata alongside the prototypes eases the management of the models and allows the citizen data scientist to draw conclusions about the effect of said parameters on the simulation and test results. These can be used to run optimization algorithms to further tune the virtual prototype's creation parameters. Typically, multiple user groups are involved in the data management of CAE data. However, users in each group store data assets in their own storage system which creates separate data silos for each group. While the data can still be shared within development teams when needed, a holistic view and uniform access to the data and its metadata is not available. Therefore, use cases like tracing data back to its originating processes, visualizing the data flow, or running optimization and data analysis tasks using all available data and metadata, is not possible so far.

Ziegler et. al. (2019) notes that several proprietary CAE data management software tools exist but they are mostly part of an environment for specific simulation software and therefore, tightly coupled to their analysis tools. They also mention academic concepts to tackle the CAE data management tasks but all having some sort of restrictions. Instead, he proposes a concept to store CAE data in a data lake while having a separate graph database to store the CAE metadata. The stored data in the data lake is tightly connected with the metadata in the graph database. Each data asset is represented by a virtual graph node in the graph database. This allows to connect the node representation of a data asset to other nodes which can contain metadata information. [ZRKM20]

While a graph structure allows to model relationships between the heterogeneous data and metadata it doesn't provide any meaning without semantically enriching the graph elements. A graph database allows to additionally define typed relationships which bring additional meaning on how the CAE data and metadata relate to each other. Ziegler et. al. (2019) additionally defines that nodes shall be one of the following three types 1. *Data Node*, which. is a virtual representation of a CAE data item holding a link in form of an access Uniform Resource Identifier (URI) pointing to the data lake 2. *Process Node*, which receives data as input, produces data as output or both. 3. *Metadata Node*, which holds information in the node itself via properties and is attached to data nodes, process nodes or both. Knowledge can be obtained from the information by enriching the graph nodes with a definition of possible node types and their interconnections in form of a semantic model. [ZRKM20]

The semantic model can be described by an ontology and represented in different ways. One approach is to store the definition in a single file as shown in Listing A.2, whereas another approach is to store the model in a predefined database schema. When using a graph database, the ontology



(a) A graph representation of an ontology where red nodes define *classes*, teal nodes define *relationships*, and brown nodes define class *properties*. *DOMAIN* relationships denote the originator of a relationship instance while *Relationship* relationships denote the target.

(b) A graph representation of sample instances of the ontology where the color green denotes instances of the *Data* class, blue denotes instances of the *Process* class, and pink denotes instances of the *Metadata* class. Properties have been omitted for readability reasons.

Figure 3.2: A graph representation of an ontology and a sample instantiation.

can be modeled as a graph of nodes where each node either defines a class of nodes, a node's properties, or a specific type of relationship between nodes as shown in Figure 3.2. A *name* property on the nodes indicates the defined class name, property label, or relationship type, respectively. The relationships between the ontology nodes need to include predefined types that indicate on which side of a defined relationship, a specific class node can occur. By using such an ontology to semantically enrich graph elements, the generic graph-based metadata management can be tuned to a specific domain as shown in Figure 3.2. Currently, the semantic enrichment has to be handled externally which is why an interface is required that unifies the graph-based metadata management with its semantic enrichment.

Processes in this concept refer to processing steps of the product development processes. However, supporting processes such as data transformation steps could also be modeled. As already mentioned, small executable scripts are often used to automate these repetitive transformation steps. Since this is common practice during product development, a graph-based metadata management system for CAE data should also support the execution and management of these scripts. These executable scripts will be called plugins for the remainder of this thesis and can be written in any programming language. They will use the virtual graph nodes to retrieve the data files in the data lake. Therefore, they will reuse the same graph database and data lake that the graph-based metadata management concept describes. Furthermore, the plugins themselves can be modeled as *Process Nodes* in the graph database and therefore, are an active part in the graph-based metadata management concept. However, since the plugins need to be executed on demand it is not advisable to store them in the data lake themselves. Plugins should rather be stored on the filesystem of a server where they can be executed on demand.

The graph-based metadata management concept involves multiple data stores which requires users to connect and interact with different technologies. Furthermore, to execute the plugins, citizen data scientists need to set up specific environments and run plugin specific execution commands based on the language they have been written in. The resulting data would have to be manually imported

into the data lake and manually annotated with metadata stored in the graph database. Since there are quite some manual processing steps invloved and all the subsystems are part of the graph-based metadata concept, a common and unified interface is needed to address all parts and subsystems of the concept without much difference in usage. Having a single interface for the management of all CAE data and metadata for all user groups is called democratization of data access. This enables the users to access and use all data and metadata in any process of the product development.

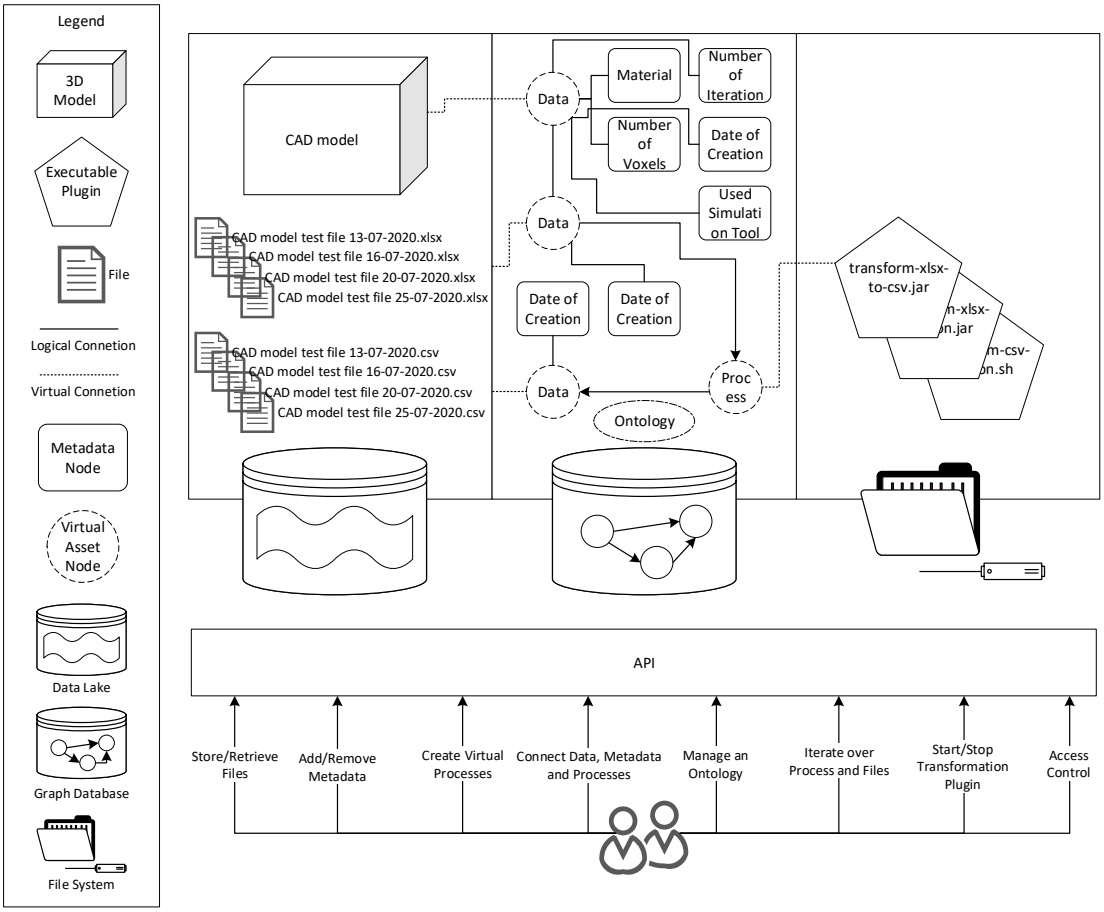


Figure 3.3: An overview of the graph-based metadata management concept that Ziegler et. al. (2019) proposed. It shows how data and metadata is related to each other as well as what kind of functionalities a unified management interface must provide.

Figure 3.3 shows a high level overview of the graph-based metadata management, as well as what capabilities a single interface needs to provide. It is important to provide the default CRUD operations on data items as well as on metadata nodes. While the conceptual data model is generic, support for a domain specific metadata model represented in form of an ontology should also be included. Plugins are an integral part to support the analysis process of citizen data scientists and should be managed by the same interface. The interface needs to provide the capability to start, stop and to query their progress. Finally, the interface needs to implement an authentication mechanism to prevent unauthorized data access.

4 Software Architecture Requirements and Use Cases

Software architecture is a set of architectural elements, their relationships and properties which are needed to reason about a system [BCK12]. Depending on which aspect of a system should be focused, the architectural elements can be defined to represent modules, components with connectors or allocation structures [BCK12]. Therefore, architecture is a part of every software application which is either explicitly defined or exists implicitly. When a new software shall be developed its functional and non-functional requirements must be defined, so an architecture can be created that supports these requirements. It is important to think about how to achieve this before the actual coding process starts.

In Section 4.1, three use cases are presented which show how the user will interact with the software interface. Section 4.2 shows the software requirements that are extracted from the use cases. In Section 4.3, a set of challenges is introduced that will also have an impact on the design of the software architecture.

4.1 Use Cases

As already mentioned in the previous chapters a software application interface needs to be created which helps engineers, data scientists and other user groups involved in engineering projects to manage CAE data and metadata as well as support additional ETL steps on the data. However, the users of the software application do not necessarily have to be specific users of the mentioned user groups themselves but can also be other software clients. These clients can be software tools used by the user groups in their specific domain to directly extract or insert CAE data and metadata instead of using the user as a middleman to manually transfer data between software applications. An example of such a client can be a visualization application that automatically fetches the data and presents it in a visually appealing way or software scripts used by a data scientist to automatically fetch the newest CAE data and metadata for his analysis. This thesis is about how an architecture for the concept of an interface of a graph-based metadata management looks like and what capabilities it must provide. This means that the use cases are not directly showing how to use the interface but rather how future clients might look like that will use this common interface.

Use cases are used to extract software requirements and architectural constraints because it is important to know who will be using the software and what do they want to achieve with it. Use cases are not only important to visualize the interaction of users with the system to be developed but it also allows to estimate the development effort before starting the actual development process [MAC05]. In the following sections, three use cases will be introduced which give a more detailed overview about what metadata management interface has to offer in terms of functionality.

4.1.1 Linking CAE Data with Graph-based Metadata

This use case covers the basic operations that an interface for a graph-based metadata management system must provide. As explained in Chapter 3, CAE data needs to be stored and managed with their metadata. Data management includes but is not limited to storing, accessing, updating, and deleting data items which is commonly referred as CRUD operations. The client, users employ to upload and query data in the use case is not further specified and different clients should be supported as well.

An engineer uploads a 3D model of a product prototype to a web service using the graph-based metadata management interface. Additionally, the engineer adds metadata to the model such as the creation date, the number of voxels and the material he used when he created the prototype. Another engineer uploads simulation results in form of text files. The engineer adds a virtual representation of a simulation process and links the uploaded 3D model via an input relationship to the process representation. He connects the process representation with the simulation result via an output relationship. This way, anyone is able to understand the origin of the files and how they are already used. To verify the CAE files are successfully stored in the graph-based metadata management system, he uses an ontology to query all nodes of the *Data* class. Figure 4.1 shows a visual representation of this use case and how the actions are related.

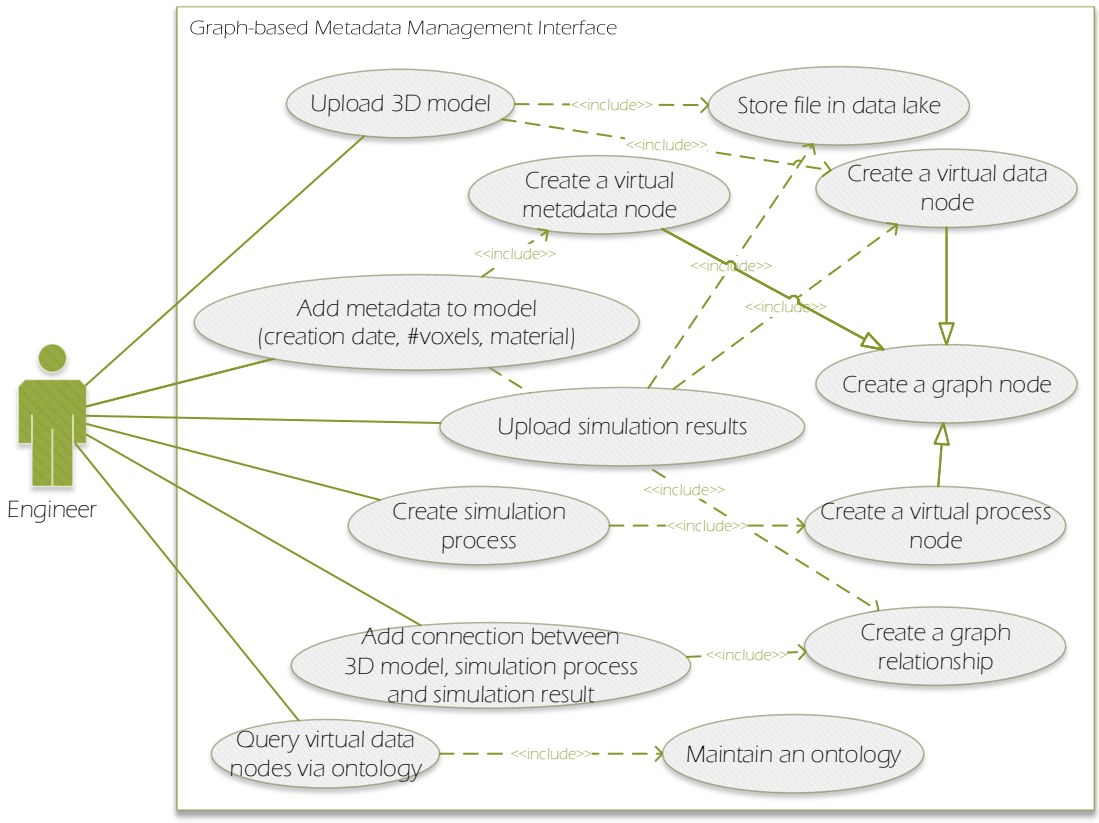


Figure 4.1: Use case diagram showing the interaction of a user and the graph-based metadata management API to upload data files and add metadata.

4.1.2 Visualizing Graph-based Metadata

The CRUD operations are an essential part of any data management but sometimes it includes additional tasks such as analyses. In the context of CAE data, this means gaining new insights from available data. To do so, it helps to understand where the data originated from which is known as data lineage. This can be done by visualizing the data flow and then following the connections between data items. It is important to know how such a client might be used to understand how the interface needs to expose certain functionality so, that such a client is also supported in the future.

In this use case, a visualization client is shown which could be one of the future clients. A citizen data scientist will use the visualization client to understand the flow of each data item. To present a graph to the citizen data scientist, the visualization client has to query all graph nodes and relationships from the graph-based metadata management interface. When the citizen data scientist selects a graph node, additional information about the node shall be presented. Therefore, the visualization client needs to query additional information about the selected node. The visualization client presents a detailed view about the selected graph node and shows incoming and outgoing relationships. The citizen data scientist follows an outgoing relationship and the traget node will be selected. This triggers the same procedures like on the previously selected graph node. Figure 4.2 shows a use case diagram of the interaction between a visualization client and the graph-based metadata management interface.

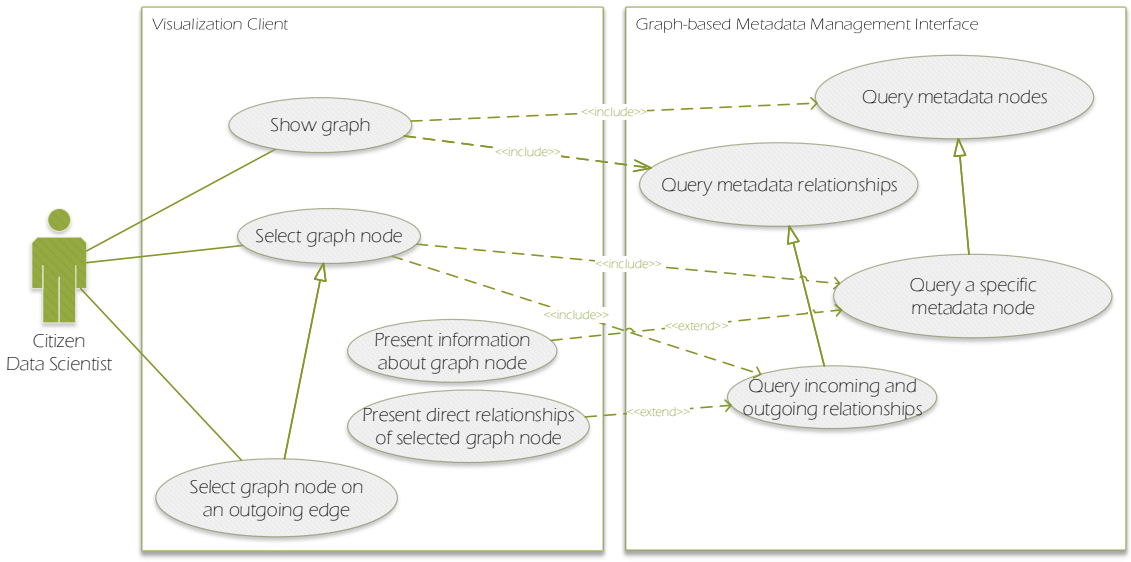


Figure 4.2: Use case diagram showing the interaction of a visualization client and the graph-based metadata management API to visually present the metadata to a user.

4.1.3 Transforming CAE Data

The third use case is about additional support for data analyses. Citizen data scientists often use machine learning tools in their analysis where machine learning models are trained on available data sets. These tools usually need a specific input format which might not be the same format or layout in which simulation results or test results are stored. Therefore, a citizen data scientist often uses small scripts, that can be executed whenever needed to transform new data to be used by machine learning tools. However, these scripts can be useful to other citizen data scientists as well. Therefore, they are created as self-contained executable files which are referred to as plugins throughout the remainder of this thesis. The execution of these plugins might take a long time depending on the amount of data items and the processing steps required. The citizen data scientist should be supported by running transformation processes on a server on demand or in a certain interval. It is important to look at such a use case to understand how the graph-based metadata management interface needs to expose certain functionality to support such a client in the future.

In this use case the citizen data scientist creates a plugin to transform simulation results from text files to Comma-Separated Values (CSV) files. The plugin is stored on the webserver that exposes the graph-based metadata management interface. He verifies the plugin is recognized by the web service by requesting a list of all available plugins. He starts the plugin with the ids of the data items that he wants to transform. Later, he checks the status of the plugin process which gives him information such as the exit code of the plugin process. Additionally, he verifies that no errors occurred during its execution by requesting the process output and error log of the plugin process. Figure 4.3 shows a visual representation of the use case and how the actions are related.

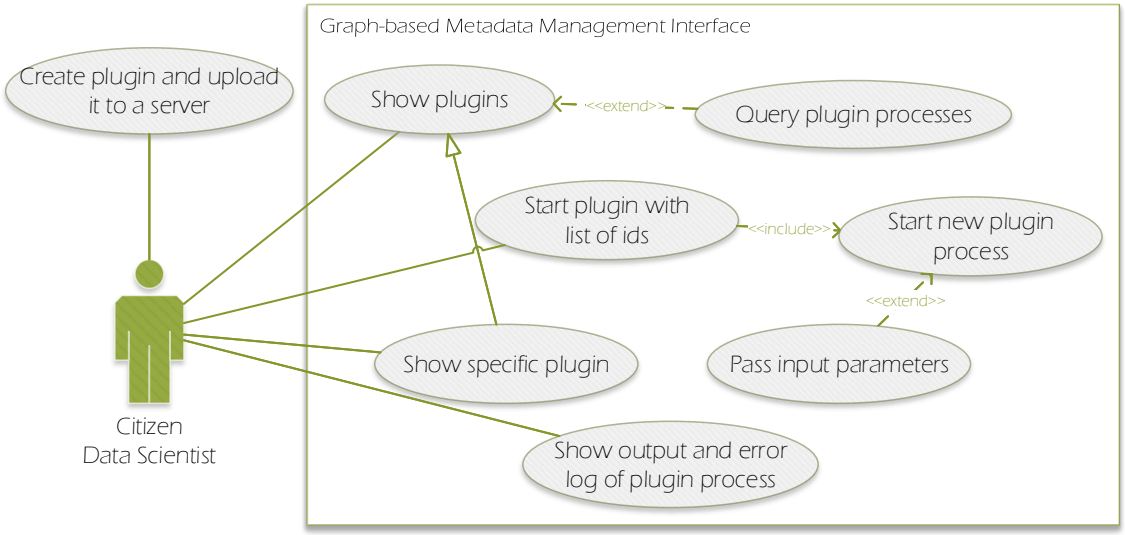


Figure 4.3: Use case diagram showing the interaction of a user and the graph-based metadata management API to transform data for an analysis step.

4.2 Requirements

Three use cases were introduced in Section 4.1 which give an idea of how different users and clients might use the interface of a graph-based metadata management. They show, the most common way to interact with the interface is by calling scripts or using custom clients. This means that the interface of the graph-based metadata management needs to be an Application Programming Interface (API) and the exposed data or functionality needs to be in a format that can easily be understood by other applications. The API should provide an abstraction layer of the underlying graph-based metadata management system which should not be exposed. This will ensure that the underlying concept can be extended or exchanged. An example of such an extension would be the introduction of another database management system that is used to store specific CAE data differently.

The API prevents the user from observing where the data and metadata is stored and does not need to change its access functionality which is important when multiple clients might use an already existing interface. Therefore, a uniform interface is required that covers data, metadata, and plugins. Data and metadata need to be linked with relationships to expose the data connections which are required by some clients, such as the visualization client. It also needs to be possible to iterate over the data and metadata items to follow paths on the graph. The API needs to be extensible in a way that changes to the underlying graph-based metadata management concept do not invalidate the architecture of the API. Citizen data scientists might also use different languages to create their plugins which means a wide range of different plugin types should be supported and a concept needs to be created to allow additional plugin types in the future without major modification of the software's architecture. The use cases show, it is planned to have many different user clients as well as user groups using the API. This means it is necessary to create a security concept that allows different clients as well as different users to use and access different parts of the API. Allowing different user groups access to the data of different data stores is also called democratization of the data access.

The use cases show that the CRUD operations need to be created for data items, metadata and partially for the plugins. In case of data items, this means a file upload and download, while the metadata needs to be uploaded in a format that can easily be created by the clients that use the API. However, it is important to note that CRUD operations of data items should seamlessly integrate into the CRUD operations of metadata since it belongs together in the interface's abstraction. This is an important point why the interface is necessary since otherwise multiple APIs could be created each dedicated to their respective domain (data, metadata, plugins). Another argument against individual APIs is data integrity because the responsibility of linking data with metadata is transferred to the client. As an example, a user could delete a data item but accidentally keep its metadata which introduces inconsistency in the graph-based metadata management. Therefore, a single interface is required to unify the data and metadata management. Plugins should only be accessed and started by the users because they are domain experts but not programmers. Instead, the upload, update, or deletion of plugins should be handled by an administrator of the server in consultation with a programming expert. Otherwise, it might be possible to upload plugins with a malicious intent making the web server unresponsive or stealing sensitive data. Based on the use cases, the following list of software requirements, which is relevant to the architecture of the application, has been created.

R1 **Uniform Web Interface in Form of an API**

Data, metadata, and plugins should be accessed via a common and uniform interface to create a cohesion between them. The graph-based metadata management concept includes external storage systems on distributed servers. This makes it difficult for engineers to maintain and access data and its metadata. Therefore, data democratization needs to happen by providing a central location to access all CAE related data, metadata, and plugins. The interface needs to be implemented as an API to automate data access by a variety of clients. This way, future clients can also easily integrate with the existing interface. Therefore, it is important to provide a uniform web interface as central access point for the graph-based metadata management system.

R2 **Unified Management Operations for Data and Metadata**

CAE files are often in a proprietary format and can be seen as unstructured while metadata can be expressed in a structured format and should be handled as such. The interface needs to provide management operations to allow the user to manage data and metadata. However, the management of data and metadata shall be unified, so that one cohesive set of management operations exists. For example, a file upload operation should support to store related metadata in the same operation.

R3 **Link Data to Graph-based Metadata**

CAE data files are stored in a data lake while metadata is stored in a graph database. To correlate metadata to a specific data file, a virtual graph node needs to be created for the data. The virtual graph node needs to linked to the data node by storing a reference to the data in its properties. This way, data can be connected to its metadata by connecting the virtual data node with the metadata nodes in the graph database.

R4 **Management Operations for Graph Relationships**

Data and process graph nodes need to be connected via relationships. Therefore, the interface needs to provide management operations for relationships between graph nodes. The use case *Visualizing Graph-based Metadata* shows that a visualization client might be used to visualize the data flow between processes. To be able to follow the connections between data items, the connections for a specific graph node needs to be retrievable. Therefore, the interface also needs to provide a functionality to retrieve all incoming and outgoing connections of a specific node.

R5 **Plugin Management**

In this context, the plugins refer to small scripts or programs which are used to automate data transformation steps required by citizen data scientists. However, the plugins are written by different users or might have been developed externally. The API needs to expose functionalities to manage plugins which means arbitrary code might be executed on the server. In addition to the security concerns, the API needs to provide life cycle management capabilities for the plugins which means plugins need to be started, observed, and stopped at any time. The observation must include the standard output of the plugin as well as any error logging. The intention of using plugins is to transform some data in the data lake which means users need to be able to pass input parameters to the plugins which can define the data to be transformed.

R6 **Access Control**

The API shall expose data assets and functionality to execute processes on the web server which means a security mechanism must be implemented that prevents unauthorized data access and code execution. Since the users of the API are using custom clients to access the API it is necessary to use an authentication method that can be easily used by a client. Additionally, the API needs a means to distinguish between users groups and allow access to different parts of the data or functionality. For example, the functionality to execute plugins is not needed by some user groups while others might not need access to the stored data.

R7 **Automated Test Environment**

Testing is a fundamental part in software engineering to verify a system is working correctly. Common types of tests include unit, integration, and system tests. A unit test verifies procedures and the interconnection between them. An integration test verifies the integration of a single component in a system. A system test includes all components of a system and verifies they work together as intended. The test environment should also be as close to the production system as possible. The effort to set up and run tests is increasingly difficult from the former to the latter test types because it includes an increasing number of components. The graph-based metadata management concept includes a distributed data lake, a graph data database and plugins stored on a server. Additionally, the web service providing the interface to the graph-based metadata management system might be split into distributed components as well. Therefore, it is necessary to automate the test environment to quickly set up and test all components of the system.

4.3 Challenges

In the previous section a list of software requirements has been presented that need to be considered for the architecture of a graph-based metadata management system. It includes requirements that are very common to many other software applications as well. However, in the context of product development with CAE the requirements impose some domain specific challenges which will be presented in the following list.

C1 **Transfer of Large Heterogeneous Files**

File transfer is required to provide management operations for data files. However, in the context of CAE data these files can be extremely large and might very well be over hundreds of gigabytes or even a terabyte in size. Additionally, the CAE files are heterogeneous and do not have a common format. This raises the challenge how these huge heterogeneous files can be transferred to an external data lake via a uniform interface.

C2 **Create a Catalog for CAE Data**

A metadata management system can be used to create a look up of data and metadata that can be used to select, filter and sort the results. However, the graph-based metadata management additionally provides predefined classes of graph nodes and typed relationships between them. Therefore, domain experts have a detailed overview about their data and processes. However,

providing an interface with simple CRUD operations for the individual resources would on only allow tech affine user groups to use the graph-based metadata management system. Therefore, the uniform interface also needs to act as a catalog for CAE data for all user groups.

C3 **Extensible Plugin Support**

The API shall provide a functionality to support citizen data scientists in preparing CAE data and metadata for data analyses. Therefore, the interface needs to provide capabilities to start, stop, and monitor available plugins. Additionally, different programmers might use different languages to create their plugins or scripts which means the plugin functionality should be able to support new plugin types in the future.

C4 **Semantic Attribution of Metadata via Ontology**

Metadata helps to manage and analyze data if there is a meaningful connection between them. A graph-based metadata management already adds meaningful connections between data and metadata by using typed relationships. However, an ontology can be used to gain even more insight about the data by adding semantic meaning to the data and metadata. By using an ontology, the citizen data scientists can use inference to gain more insights about the data and metadata. Therefore, semantic attribution should be possible

C5 **Integration in Enterprise Environment**

The graph-based metadata management system shall be used in the product development life cycle which means it has to work in an enterprise environment. This means it is very likely that the metadata management system will operate behind a company firewall to ensure additional security of the company's internal network. Therefore, the graph-based metadata management should preferably work with Hypertext Transfer Protocol (HTTP) or Hyper Text Transfer Protocol Secure (HTTPS) which are opened by default in most company firewalls. If ports need to be opened, their number shall be reduced as much as possible and the default ports of common technologies shall be used instead of custom port ranges. Additionally, enterprise environments vary a lot depending on the size of the enterprise. Small enterprises might initially have fewer integration requirements but once they evolve into bigger enterprises their software systems need to adapt in terms of performance, workload, and additional functionality. To cater the needs of smaller as well as bigger enterprises the software architecture shall be developed with extensibility and evolvability in mind. This will support small companies when transitioning to large enterprises without changing the underlying software architecture.

C6 **Manage Resources and Functions**

The graph-based metadata management interface shall support the management of resources and the execution of processing functions. The management of resources includes CRUD operations which are typically used in synchronous scenarios such as the ones described in the first two use cases. However, the execution of processing functions as described in the third use case describes an asynchronous scenario because a client should not be blocked during the execution of long-running plugins. Therefore, it is important to find an architecture which supports both types of scenarios.

5 Software Architecture

Software architecture is a broad topic in software engineering and an integral part of every software even if it is not explicitly defined [BCK12]. It is comprised of architectural elements and the connections amongst them and is formed by a series of architectural decisions [BCK12]. The architectural elements represent logical and physical components of software and hardware. Architectural views represent a part of a software in the eyes of a specific group of stakeholders of the software. They consist of a set of coherent architectural elements which are of importance by some stakeholders. The different views can be categorized into the following three major categories: 1. Modules 2. Components and Connectors 3. Allocation [BCK12]. The first category provides architectural elements to model sets of functionalities or smaller units as a single module. This can be useful to break down a large software application into smaller modules with each having their own responsibility. Considering the fact that a software application typically has more than a single responsibility in terms of functionality, it is preferred to split it in multiple modules which can be independently developed. An architecture diagram allows to give an overview of all modules and how they make up the complete software system. The second category, components and connectors, can be used to model smaller components which provide only a single or a small set of functionalities. It also allows a more fine-grained modeling of their relationships which can be used to show the components' data flow and types. This can include classes, data types, methods and other programming related constructs. The last category, allocation, is used to create an allocation model that shows how the software application is allocated to (virtual) hardware resources. It can also give a more high level overview of how the software application is allocated next to other software applications in a bigger system. The architectural views presented here include high-level as well as low-level details of a system, despite some controversy that speaking of low-level details is often referred to as *software design* [Mar18]. While it is important to create a detailed software architecture, there should be no implementation details included [BCK12]. This means the developers can still evaluate different programming languages and frameworks to choose the best tool to implement a specified architecture.

Software architecture can also be considered similar to real architecture in the sense that it gives a blueprint of the system in form of diagrams [Sch13]. The blueprint includes all the details which are needed during the construction for all the involved user groups. Similarly, the software architecture has to provide all the details for all involved user groups. But software architecture does not only help to explain the developers how they should build a software, it is also important to meet non-functional requirements specified by stakeholders. While functional requirements specify what kind of functionalities a software product has to offer, non-functional requirements define the quality attribute of theses functionalities. Similarly to how a real architecture defines the maximum weight to be put on a bridge, software architecture has to define the maximum number of users of the application and the average response time for a web functionality. Creating and evaluating a software architecture before the coding process starts helps to meet the non-functional requirements without having to recode an already existing code-base. Some non-functional requirements, such as

scalability, availability, reliability, recoverability and performance, are crucial to the functionality of an application in an end users point of view. However, it is important to note that there are other non-functional requirements, such as serviceability and maintainability, that are equally important from a developers' point of view.

Case studies revealed, that with every new release cycle of the life-cycle of a software, the cost per Lines of Code (LOC) goes significantly up while the productivity per release drops in the same velocity if there is no focus on software quality [Mar18]. This means initially saving costs by skipping on defining a good software architecture will just lead to more costs when the product life-cycle progresses [Mar18].

Unfortunately, while there is no construction site starting to work on a building without having an architectural blue print, there are software projects that are being developed without having architectural diagrams. In the end software architecture should always be defined before any coding starts which helps to meet functional as well as non-functional requirements. A good software architecture is a key component to create software applications with long life-cycles as it helps to greatly reduce the developing costs in future release stages. Since the software architecture that has been developed in this thesis is part of a concept that might be extended in the future it is even more useful to thoroughly think about its architecture and how to leave room for future extensions. Software architecture of applications often have to provide solutions to common and reoccurring types of problems.

This is where the notion of architectural patterns and architectural is introduced. While patterns have a long standing history and originate from real architecture, they have also become an important tool in the context of software engineering [Mal18]. Patterns provide proven and well documented solutions to reoccurring problems and are an important tool for software architects as well as software developers [Mal18]. While patterns are used to solve problems, architectural styles merely provide a language to categorize reoccurring types of software architectures. Popular architectural patterns include *Request-Reply*, *Publish-Subscribe*, and *Model-View-Controller (MVC)* whereas popular architectural styles include *Service Oriented Architecture (SOA)*, *Representational State Transfer (REST)*, *Layered Architecture*, and *Pipes-and-Filter*. An architecture that is built around a specific set of architectural patterns is sometimes also referred to by the patterns' name.

An important step in creating a software architecture is to look at reference architectures of similar projects. Miloš Simić (2016) proposed *Clover*, a graph-based metadata management on general data files and folders, stored on a file based storage infrastructure with metadata stored in a graph database [Sim16]. While this system is not specifically tuned for handling CAE data, its general idea of storing and managing metadata is close to the one that we are creating an architecture for. The system is composed of a *Clover* service which handles the data operations by using a storage infrastructure and the metadata operations by communicating with a metadata service. The metadata service is implemented by using a graph-database for storage and is independent of the storage infrastructure. The *Clover* service is made publicly available to clients via HTTP as the single interface of the whole system. Unfortunately, there are no further details as to how the *Clover* service communicates with either the storage infrastructure or the metadata service.

Beheshti et al. (2018) proposed *CoreKG* — a knowledge lake service which provides graph-based metadata management and extraction capabilities on a data lake — which can handle structured and unstructured data in different Structured Query Language (SQL) and NoSQL databases [BBNT18]. The architecture is composed of a knowledge lake which is composed of a knowledge graph for

metadata storage and a data lake for data storage. The data lake itself is composed of different data islands which reflect the type of their databases and are provided as database-as-a-service offerings. The metadata is automatically extracted from the data lake by curation services which are communicating with the database-as-a-service offerings and the knowledge graph. Query and search operations on both data and metadata is done by using built-in SQL queries and dedicated search tools. *CoreKG* provides a single REST API to serve any client with CRUD and query capabilities on the data and metadata. The REST API also implements a token based authentication for access control.

While these reference architectures are a good starting point neither of them specifically focus on the management of CAE data and its unique requirements. Specifically, the management and execution of plugins for CAE data and their semantic enrichment is missing. By using architectural styles and patterns for distributed systems, different software architectures for the graph-based metadata management of CAE data have been developed. In the following sections, these architectures are presented and discussed based on the use cases, requirements, and challenges defined in Section 4.1, Section 4.2, and Section 4.3, respectively. In Section 5.1, the client-server architecture is described and an architecture for the graph-based metadata management system is presented in form of a 2-Tier, 3-Tier, and N-Tier architecture in Section 5.1.1, Section 5.1.2, and Section 5.1.3, respectively. Section 5.2 and Section 5.3 describe SOA and REST, respectively. A software architecture for each architectural style is also presented and discussed in these sections. In Section 5.4, the microservice architectural style is described. A message oriented and a hypermedia-driven microservice architecture are presented and discussed in Section 5.4.1 and Section 5.4.2, respectively. Section 5.5 provides a recap about the presented architectures and presents a REST-based microservice architecture which offers the best trade-offs for the defined use cases and requirements.

5.1 Client-Server Architecture

The client-server architecture is a well known and used architectural style (or sometimes called architectural pattern [BCK12]) for creating a distributed software architecture [SKA15]. The concept includes clients which initiate interactions in form of requests and servers responding to said requests [BCK12]. Clients and servers are connected through a transportation medium and a servers' address needs to be known by the clients but not vice versa [BCK12]. Similarly, a common data format needs to be defined that is used for the communication. The common data format should be structured to prevent additional parsing steps, which is why often the well known data formats Extensible Markup Language (XML) or Javascript Object Notation (JSON) are used. The communication is done in a *request-reply* style and therefore, synchronous, e.g. the client waits for an answer of the server after sending a request [Olu14]. Therefore, client and server are tightly coupled in this architecture in terms of 1. Reference, since the server address needs to be known by the client 2. Time, because of synchronous communication 3. Format, because the format in which they are communicating needs to be defined before communication 4. Platform, because the communicated information needs to be understood on both sides. The most prominent examples of such an architecture are web-based applications which serve websites on a known address when requested by a web browser [BCK12].

There are different types of client-server architectures which differentiate by how responsibilities are split to clients and servers. The functionality of a client-server architecture is often split into different layers to achieve separation of concerns. An interface is used to abstract the implementation details which can enable a layered system to be distributed to multiple servers. In a layered system, each layer is only allowed to communicate with components of the same or the underlying layer. Commonly, an application can be separated into at least the following three layers: 1. Presentation layer which exposes a User Interface (UI) a user can interact with. 2. Business layer which handles the application logic and the business processes of the application. 3. Infrastructure layer which handles any data and infrastructure related tasks. [SKA15] When talking about a client-server architecture, the layers which are implemented on the client side are referred to as the frontend and the layers that reside on the servers which are used by the clients, are called backend.

The most common distributed client-server architectural styles are called 2-Tier, 3-Tier, and N-Tier architectural styles [Olu14]. In the following sections an architecture for the graph-based metadata management will be described and discussed for each of these client-server architectural styles.

5.1.1 2-Tier Architecture

In a 2-Tier architecture there are five ways to distribute the three layers onto the client and the server. 1. Presentation layer on the client, presentation, business and infrastructure layer on the server 2. Presentation layer on the client, business and infrastructure layer on the server 3. Presentation and business layer on the client, business and infrastructure layer on the server 4. Presentation and business layer on the client, infrastructure layer on the server 5. Presentation, business and infrastructure layer on the client, infrastructure layer on the server . As the use case *Visualizing Graph-based Metadata* shows, a visualization client shall be one of the clients of the graph-based metadata system which will provide a presentation layer. As the metadata-management backend shall serve a number of different clients, it is advisable to separate the presentation layer from the business and infrastructure layer. While the visualization client might also include some business logic, the use case *Linking CAE Data with Graph-based Metadata* should also be performed by a lightweight client.

Figure 5.1 shows a concept of a 2-Tier Architecture where the client only implements the presentation layer while a single server is used to provide the business and infrastructure layer of the graph-based metadata management. The client will communicate with the server by using a single interface in form of an HTTP API as defined by the uniform interface requirement in Section 4.2. This makes the server application versatile in a way that any client can easily send business requests which can be read and understood by applications. This means any future client defined in the use cases in Section 4.1 will be served by the same backend. The client will communicate with the server by using HTTP as transport protocol and invoking HTTP endpoints where required data can be uploaded. This way the graph-based metadata management could be made publicly available even when located within a corporate environment since firewalls, by default, allow the HTTP port (80) to pass through.

The backend functionality itself is split into different components which can still have dependencies on each other and their functionality can be invoked by API endpoints. The system consists of components that provide the business functions and all the data stores that are needed by the graph-based metadata management. These include a filesystem, a data lake, a graph-database and a

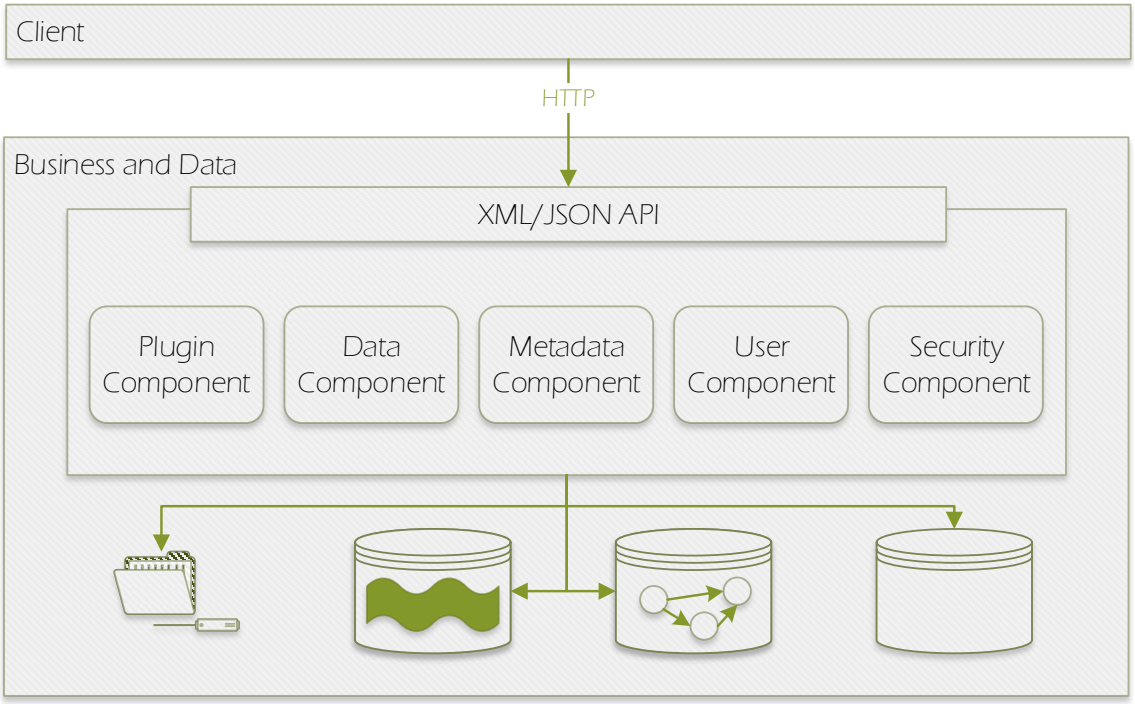


Figure 5.1: A client-server 2-Tier architecture for a graph-based metadata management of CAE. The icons represent a filesystem, data lake, graph database and a SQL database, respectively when read from left to right.

SQL database which are represented by the four icons in Figure 5.1. The icons will be used with the same meaning throughout the remainder of this thesis. While the business functions are split into independent components the API gives the impression of a unified system. In the following list each component is described in more detail.

- Plugin Component

  The plugin component handles the life-cycle of the CAE data ETL plugins. The API exposes endpoints to start, stop, and to show the current status of the plugins and its processes. Input parameters can be passed by using a structured format like XML or JSON which can be easily parsed by the software component. The plugins are located on the filesystem of the server and will be started by the plugin component. The plugin component will maintain a registry of supported plugin types by scanning for plugin types defined by a common interface which allows to add more plugin types in the future. The plugin component can maintain an in-memory registry or a database connected via an embedded driver to store all running plugin processes. This will speed up requests that ask for a list of all running plugin processes since this does not require additional communication with the OS whenever such a request is made.

- Data Component

  The data component will handle the upload of CAE data files and can also be called via the HTTP API since HTTP does not restrict the upload size of files. The data component will be responsible to pass the data through to the data lake and is connected via an embedded driver to the data lake which resides on the same system. In order to prevent out-of-memory errors, the data has to be directly streamed to the data lake.

- Metadata Component

  The metadata component will expose the CRUD functionalities of the graph-based metadata and is connected to a graph-based database on the same system via an embedded driver. Additionally, this is where the semantic attribution of the metadata will happen. To do so, the metadata component needs to expose an interface to upload an ontology file which needs to be transformed so that the represented ontology can be stored in a way that the metadata component can make use of it. This can either be done in an in-memory representation, a SQL database or a NoSQL database such as the graph-database where the metadata itself is stored.

- User Component

  The user component will create an administrator account and provides user management functionalities which can be accessed by the administrator. The user component uses a SQL database that is directly connected via an embedded driver and resides on the same system. An SQL database is used because it is made for storing structured data and provides ACID properties. Exposing the user management functionalities on the API allows the system administrator to use scripts which help to manage the users of the graph-based metadata management. For example, the administrator could create a script to transfer existing user accounts of the enterprise system into the graph-based metadata management system.

- Security Component

  The security component will be used to authenticate and authorize requests. By using HTTP as transport protocol we can make use of proven HTTP security mechanisms. One way to provide access to the functionality of the backend is to provide a login endpoint via HTTPS where a client authenticates by sending its username and password as form parameters in the request. Once a client is logged in as a user he can access any endpoint that the user has permission to use. This can be realized by adding permission roles to users and to check if the logged in user has the permission to call the endpoint.

The presented 2-Tier Architecture is quite simple and exposes a common interface which unifies the functionality of the graph-based metadata management on a single remote server. The architecture splits the system into layers where the presentation layers are on the client side, while the business and data layers reside on the server. The distributed nature of the architecture allows users to store and share CAE data and metadata on a central server which can be made public either to an enterprises' internal network or to the world wide web. The simplicity of the architecture helps to cope with the challenges defined in Section 4.3. The testing of a single server side backend is much easier than having a more distributed system with lots of applications which need to be coordinated. The deployment of such a system is also quite easy as everything can be built in one project and deployed on the same server which means there is less configuration to be done. It is also great in terms of networking since a direct communication between the client and all its components and

data stores is used. This means CAE data files can be directly uploaded to the target server where they are stored in the data lake. When talking about large files, this can have a huge impact when there is no need to forward data between multiple servers.

Unfortunately, the simplicity comes with the drawbacks of less extensibility and scalability. While the API provides a great way to abstract the underlying components, any modification has to be made on the whole backend. While layers in the components provide some separation of concern which prevents that a change in some line of code has an impact in different areas of the overall code, the application is still deployed as a whole. This means whenever a modification is done the whole backend and its API will go down until the deployment has finished. Since the infrastructure layer is deployed with the whole application it can not be easily scaled horizontally since the applications would need to provide a synchronization mechanism to share their session state. The security mechanism which is based on a login mechanism does also not scale horizontally and is not commonly used when securing APIs. However, since this architecture can already not be scaled horizontally it makes sense to provide a login mechanism since this reflects how the user is mostly represented on the client side. Once logged in, a user can use all functionality of the UI on the client side.

Another drawback of the architecture is the CAE data file upload which does not provide any retry functionality. This means, any client which needs to upload CAE data files will have to implement a lightweight retry logic in case of upload failure. Since the data is very large, it would be great to include a mechanism to upload smaller junks of a large file and to send only the remaining junks once an upload failure occurs. A variation of the presented architecture could include a client that includes a business layer which handles the slicing of large files and implements an upload retry logic for the remaining parts. This would change the data component to include a functionality to either merge the junks of the large file or to directly save the junks in the data lake. The former method means that the server needs to persist the junks on the filesystem before merging them. This means the server might quickly run out of memory when the remaining parts are not removed from the system, especially when the junks are not complete, although they are not used anymore.

The biggest problems of this architecture is extensibility, evolvability and horizontal scaling which makes it unsuitable for large enterprises. At first sight this might not be a problem for smaller companies, but as these grow in size the architecture will not adjust to their needs and can not be used anymore.

5.1.2 3-Tier Architecture

The 3-Tier architecture is an extension of the 2-Tier architecture and extends the architectural pattern with another server between the client and the data server that is also known as middleware or middleware-tier [Olu14]. In this concept the client will implement the presentation layer which communicates to the middleware that implements the business layer and is implemented on its own remote server [Olu14]. The middleware will communicate with another remote server that provides the infrastructure layer making the middleware act as both, client and server. However, to the initial client any requests and responses will look like they are processes by or originate from the server that acts as middleware-tier. The same applies to the server that implements the infrastructure layer. Requests received by this machine look like they originated from the client that directly communicates with it.

Splitting the business logic and the infrastructure logic prevents synchronization problems when horizontally scaling the application. Since this is a big problem in the 2-Tier architecture that has been presented in Section 5.1.1, we will present a 3-Tier architecture for the graph-based metadata management which will not have this shortcoming. Figure 5.2 shows the concept of a 3-Tier Architecture where the client only implements the presentation layer while the business and infrastructure layer is split onto two different servers.

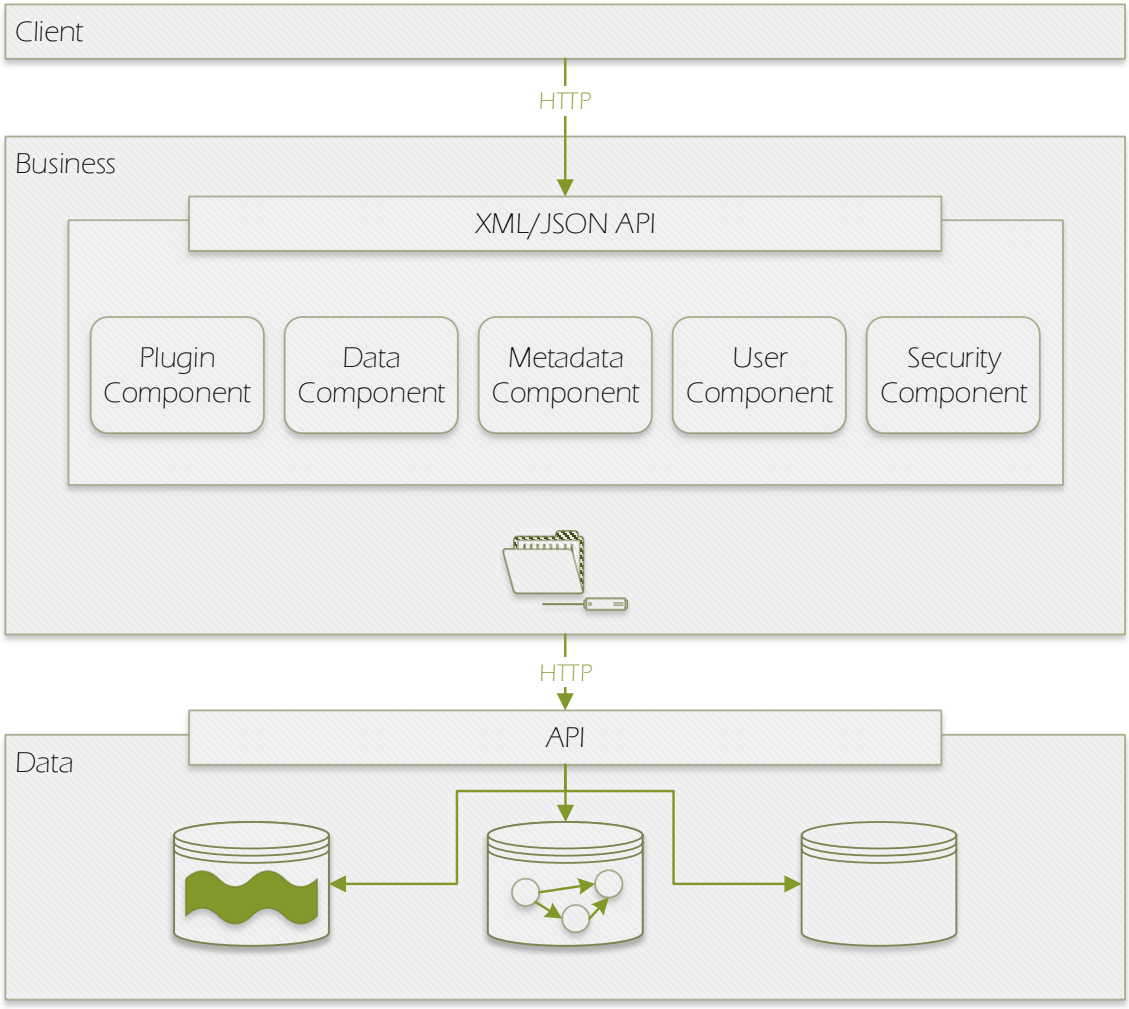


Figure 5.2: A client-server 3-Tier architecture for a graph-based metadata management of CAE data.

The client will communicate with the a middleware-tier by using a single interface in form of an HTTP API as defined by the uniform interface requirement in Section 4.2. To the client, this will look just like the interface of the presented 2-Tier architecture. This means the middleware-tier becomes a versatile server that any future client should connect to and send business request which can be read and understood by applications. The business layer communicates with the

infrastructure layer to store and receive data instead of having them on its own server. The same requirements as in the concept of the presented 2-Tier architecture are fulfilled. All clients will also communicate with the server using HTTP as transport protocol.

The backend functionality is implemented in the business layer on an application server while the data for the business requests is stored in an infrastructure layer on a separate data-tier. The business layer itself is split into different components whose functionality can be invoked by the API endpoints. Any required data will be included in the HTTP request in an easy to parse structured format like XML or JSON. While the components might still have dependencies between each other a data abstraction layer can be used to split the components. Similar to the presented 2-Tier architecture the business layer is implemented with the following components.

- Plugin Component

  This is the same component as the plugin component of the 2-Tier architecture and only differs in the way it stores running processes. It will still store and run plugins on the same server but it will store information about the running processes in an external SQL database. This decouples the platform where the plugins are executed on from the platform which stores the plugin process information.

- Data Component

  This is the same component as the data component of the 2-Tier architecture and only differs in the way it handles the data storage. Instead of directly using an embedded driver to communicate with the data lake, the data component in this architecture uses an API to communicate with the data lake that is stored on the data-tier. It is also important to keep in mind that the data needs to be directly streamed to the data-tier without caching them on the middleware-tier to prevent high storage usage on the middleware-tier and duplicated data files on middleware and data-tier.

- Metadata Component

  This is the same component as the metadata component of the 2-Tier architecture and only differs in the way it handles the data storage. Instead of using an embedded driver to connect to the graph-database an API of the data tier will be used to store and retrieve metadata nodes and relationships. In this architecture the semantic attribution in form of an ontology has to be represented on the data layer as well. Otherwise, horizontal scaling can not be done since multiple instances might have a different internal semantic attribution of the metadata and would produce different results. Additionally, a caching mechanism can be implemented to further boost the performance.

- User Component

  This is the same component as the user component of the 2-Tier architecture and only differs in the way where it stores the data. While the user data will still be stored in a SQL database it is not on the same server but has to be sent to an API endpoint of the data tier.

- Security Component

  The security component has the same responsibility as in the 2-Tier architecture but has to use a different authentication mechanism to prevent session state on the business layer as this would prevent the application from scaling horizontally. In this architecture the security

component will implement basic authentication which is specified by the HTTP protocol and will be included in every request. This separates the authentication of the business layer from using the same credentials like the infrastructure layer which adds additional security and a more fine-grained access control on the business functions instead of on the data itself.

The presented 3-Tier architecture extends the previously presented 2-Tier architecture by decoupling the business and infrastructure layer into separate tiers. This way the middleware-tier and infrastructure-tier can be scaled independently which is a big concern of the 2-Tier architecture. The decoupling between business functions and the data means that either layer can be swapped and deployed independently. For instance one implementation of a graph-database could be exchanged for a different one and the client couldn't tell a difference nor has to be notified.

The separation and deployment of business and infrastructure layer on two different servers means that more inter server communication is needed. This means a single request-reply interaction of the client with the API of the business layer might very likely result in additional requests and replies between the business and infrastructure layer. While the decoupling is good in terms of independent deployment of the business and data applications, the additional round trips of requests and replies also impose additional network problems. A 2-Tier application depends on the network connection between the clients and the backend server who hosts the business and infrastructure layers. When the client uploads a file it will directly be on the target server where it will be stored in the data lake. If a connection problem happens between the client and the server, it is the clients responsibility to try reuploading the file if necessary. The same applies in the presented 3-Tier architecture where the middleware becomes a client to the data server, which means the middleware should also be responsible to provide a retry mechanism in case the connection to the data server fails. One solution could be to cache the uploaded file on the middleware-tier and implement a retry mechanism on a fixed schedule which tries to reupload the cached file to the target data server without revealing the problem to the original client. However, caching is not an acceptable solution when dealing with large files as specified in the challenges in Section 4.3. In this architecture, the best solution is to consider minimizing connection failures and possible bandwidth problems. This can be done by deploying both tiers on the same physical server by using virtual machines or containers, or by deploying the tiers on physical servers which are physically wired to the same network. This way the connection between the user client and the middleware-tier will be the more dominant factor for determining the upload speed and stability.

It is important to note that this architecture exposes an additional API for the data layer which can be found by any clients. Therefore, it is important that this API is secured similarly as the API of the business layer. Although, it is wise to use different users and groups than on the business layer to not automatically allow direct data access to any client of the business layer. The additional API means that the architecture can evolve in the future to some degree. Clients could however directly communicate with the data layer if necessary or a new business layer can be operated while still keeping the current business layer for legacy support. The API could also evolve in the future so that clients could use the API to access the CAE data directly without having to use metadata endpoints.

The architecture provides a separation of concern regarding the presentation, business and infrastructure layer on the different tiers. However, the layers provide a single API to the whole functionality of the layers on their respective tier. This means a redeployment of the API will make the whole system temporarily unavailable. A horizontal scaling of the middleware is also very limited because multiple instances of specific components can only be deployed on the same physical server. In

addition to the problem of limited scaling, the architecture is still limited in its extensibility and evolvability. Any change in the interface of a component of the middleware as well as the addition of any new component will inevitably lead to redeploying the whole business layer which means additional downtime for the system. While the layers in this architecture are more decoupled than in the 2-Tier architecture, the same problems still persist although, to a lesser extent.

5.1.3 N-Tier Architecture

The N-Tier architecture extends the concept of the 2-Tier architecture similarly as the 3-Tier architecture but is not limited by the number of servers [Olu14]. This makes a N-Tier architecture more distributed than the previous client-server architecture types and is a common occurrence in enterprise environment because of numerous data management and application systems which reside on different systems instead of a single server. It also allows for a better access control on data and business functions from within or from outside the enterprises' network. To still give the clients a coherent view of the overall system an integration layer is needed which in turn might communicate with additional integration layers before the target server is contacted. The integration layers allow developing, building, deploying and scaling components independently of each other.

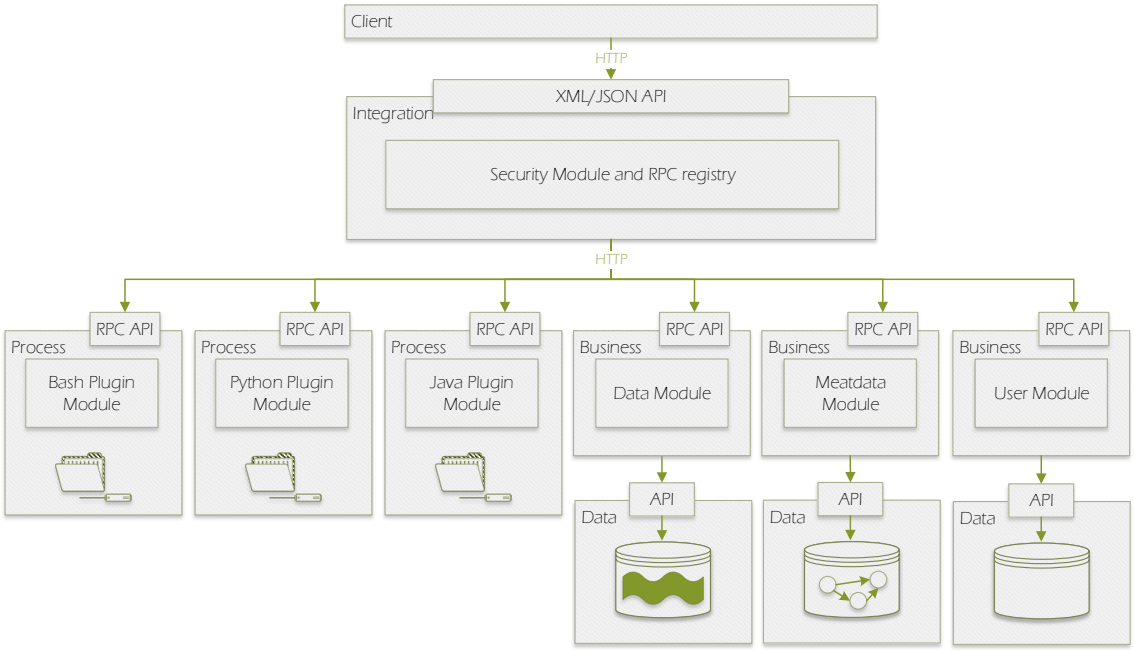


Figure 5.3: A client-server N-Tier architecture for a graph-based metadata management of CAE data.

By further splitting the business layer and the infrastructure layer and deploying them independently on different servers, the shortcomings of the previously presented architectures can be addressed. Figure 5.3 shows the concept of a N-Tier architecture for the graph-based metadata management which splits the system into an integration layer that integrates the different business and infrastructure layers which are distributed to multiple tiers. The integration layer provides a single HTTP API

for the clients with access to all the systems functionalities and acts as a request distributor for the underlying distributed system. The underlying system is composed of six tiers each of which exposes an Remote Prodecure Call (RPC) API. The integration layer maintains a registry with all available RPC calls by providing the underlying components a registration mechanism to its registry. Any request that needs to be distributed to a different component must be transformed on the integration layer. Additionally, the integration layer implements a security module which is used to handle user authentication. This way, all the servers which implement the business layer can be put behind a firewall which only allows requests of the address of the integration layer to pass through. However, the security module uses the business functionality of a user module to check credentials and does not have to maintain a separate data store. In order to boost request performance additional caching can be implemented on the integration layer so a transformation of requests into RPC calls only has to be done on demand.

The integration layer communicates with the plugin, metadata, and user modules via RPC APIs. These modules will be doing the heavy processing tasks on their respective server. For each supported plugin type a separate module will be used which stores all plugins of the mentioned type. The plugins themselves must implement a common plugin RPC interface in the language of the respective plugin type and will be directly called by the integration layer. This means an additional process manager is not needed to manage the life-cycle of all the plugins. On the other side, the plugins have to do heavy lifting as they have to implement some of the integration logic themselves such as the registration process. The synchronous communication of RPC enables the client to get immediate feedback as to when the process has finished or advances. However, long-running processes on the server will block the requesting thread on the integration layer which might become a bottleneck.

The integration layer exposes an endpoint to allow clients to simultaneously upload data files as well as metadata in a structured format such as XML and JSON. The uploaded file will be streamed into smaller junks which will be used in a RPC call that transfers the file to the data module. This module will verify the junks, merges them and uses a streaming API of a corresponding driver to upload the reconstructed file in an external data lake. Once the upload is done the integration layer will make a second RPC call to transfer the metadata storage request to the metadata module which will create new metadata nodes and relationships and store them in the external graph-database. Due to the synchronous communication the process for the upload endpoint might be blocked for some time since the uploaded files can become very large as specified in the challenges in Section 4.3.

The user module, the last module that makes up the business layer of the architecture, uses an external SQL database to store user credentials. Any user request on the integration layer will also be forwarded as RPC calls. However, since requests on the user module will be very lightweight, they will not impact the performance of the integration layer.

This architecture provides a more fine-grained deployment of the modules than in the previously introduced 3-Tier architecture. Similarly to the 3-Tier architecture, any storage system can be exchanged without redeploying any modules in the business layer. The architecture also allows exchanging and somewhat scaling the data, metadata and user modules independently of the integration layer and hidden from the clients. However, the integration layer becomes a very heavyweight component which can serve only a few users. Long-running plugins or data transfer calls will block the respective threads and it might become a bottleneck in the architecture. Distributing the individual modules and using RPC interfaces also increases the development complexity of the individual modules. It also causes more communication as the integration layer

sits in between the client and the business layer. This is especially important when talking about file up- and downloads since the data needs to be routed between multiple servers. A way to deal with the additional routing overhead is to make the connection between the additional layers more reliable by deploying the modules on the same physical infrastructure or on physically close systems. Additionally, the RPC calls can implement a mechanism to transfer smaller junks that get merged on the other server and implement a retry mechanism in case of communication failure.

Another drawback of the architecture is the testability and administration of the overall system. The system is composed of modules which can be tested separately although it becomes more complex to implement integration tests to see if the system as a whole behaves as expected which is not automatically the case even if each module itself passes all tests. Since the modules are deployed independently of each other, the deployment as well as other administration tasks become more complex. Additionally, this architecture requires an active firewall in between the integration layer and the underlying business layers since the authentication is only happening on the integration layer. However, this is a trade-off which means more administration tasks result in less implementation overhead for the business and data layers because they do not require additional authentication mechanisms. Although, this comes at the cost of not being able to access the data layer or business modules directly without having to use the API of the integration layer.

The presented N-Tier architecture excels at performance due to the direct RPC communication but suffers from the bottleneck of the integration layer, additional administration tasks as well as availability and communication overhead. The architecture does not allow to dynamically add additional business functionalities without adding new endpoints to the HTTP API of the integration layer which limits the extensibility and evolvability that were defined as requirements in Section 4.2.

5.2 Service Oriented Architecture

SOA is a design paradigm that resolves around the idea of separation of concerns to bundle capabilities that belong to the same business domain into a reusable component known as service [Erl16]. The service is the central building block of SOA and exposes a set of capabilities in form of an API which is specified by a service contract [Erl16]. The service contract is an interface definition which specifies the interaction with it and includes the service interface, interface documents, service policies, quality of service, and performance [RLSB12]. Clients, that are using the API to invoke these capabilities, are called service consumers but can also be services themselves [Erl16]. The service is either a self-contained business function or a composite service which uses other services to fulfill its service contract [Bar13]. SOA follows the contract first philosophy which means services are built around a standardized service contract and not vice versa [Erl16]. This makes SOA a good choice when developing business functions as a software which should be used enterprise wide by a variety of different clients in a standardized way.

A service is built around the following eight core principles

1. Standardized Service Contract, which defines the publicly available capabilities of a service
2. Loose Coupling, which means reducing the assumptions two parties need to make when communicating with one another
3. Service Abstraction, which means hiding implementation details behind a service contract
4. Service Reusability, which means the set of capabilities can be reused in the enterprise environment
5. Service Autonomy, which means the service shall have control over its platform and environment
6. Service Statelessness, which means to externalize any state information
7. Service Discoverability, a service needs to publish meta information to be discoverable
8. Service Composability, which means services shall be highly composed, so they can solve more complex business functions when used as a compound service. [Erl16]

To realize a SOA architecture, a service specification is needed which is used to define the service contracts. Historically Web Service Description Language (WSDL), Simple Object Access Protocol (SOAP) and Universal Description, Discovery, and Integration (UDDI) was used as specification for the interface contract although there are numerous specifications available including WSDL/SOAP/no-UDDI, REST, XML and JSON [Bar13]. However, SOAP, WSDL and WS-Discovery provide many industry standards and are widely used in enterprise SOA. Therefore, this chapter will focus on SOA based on SOAP, WSDL and WS-Discovery.

SOAP is a lightweight data exchange protocol to transfer application semantics in the XML format. It consists of a SOAP envelope, SOAP encoding rules, and a SOAP RPC representation [BEK+00]. The SOAP envelope consists of an optional SOAP header and a mandatory SOAP body which contains the message data. Header and body are further subdivided into blocks and the header is used to specify a receiver. The SOAP encoding rules define how application specific data types have to be encoded in order to be sent via the SOAP envelope. The SOAP RPC representation defines how RPC calls are represented in a SOAP envelope. Simple and complex data types can be defined by using a XML Schema Definition (XSD) file which can be used by communication partners to understand the data even though they might have a different internal representation of said data.

While SOAP can be used as a standardized exchange protocol for messages, WSDL can be used to define a service and how to communicate with it. WSDL defines a service as a set of related endpoints which are also referred to as WSDL ports. A WSDL port combines a network address in form of an URI with a specific protocol and data exchange format for each supported operation. In WSDL, the set of all supported operations on a port is called port type and the protocol and data format that the port type uses is referred to as binding. A WSDL service description can also bind to SOAP which means service communication can be described as combination of defined SOAP header and SOAP bodies wrapped in SOAP envelopes. WSDL service descriptions are written in XML and therefore, they are platform and language independent. This allows to generate service client stubs on any platform and in any programming language that supports SOAP and WSDL. [CCMW+01]

In a SOA architecture clients need a means to discover the service providers in order to use their services. Different approaches exist to either use a registry of services or to use a dynamic discovery method to dynamically add and remove services from the SOA system. WS-Discovery can be used with either way, use a dedicated discovery service or dynamically discover target services. The essential building block is that service discovery works by using a multicast protocol to either distribute messages about available services or to distribute messages about service requests in an ad-hoc network. In the former method, target services send messages with a reference to their service description via a multicast protocol. The discovery service receives these messages and maintains a registry of available service descriptions. Clients which are looking for a specific service can use the discovery service to look up service descriptions. Without the discovery service, the clients use a mulitcast protocol to distribute messages that describe the required service. The target services will receive these messages and respond only if their interface definition matches the required service description. [NRMK09]

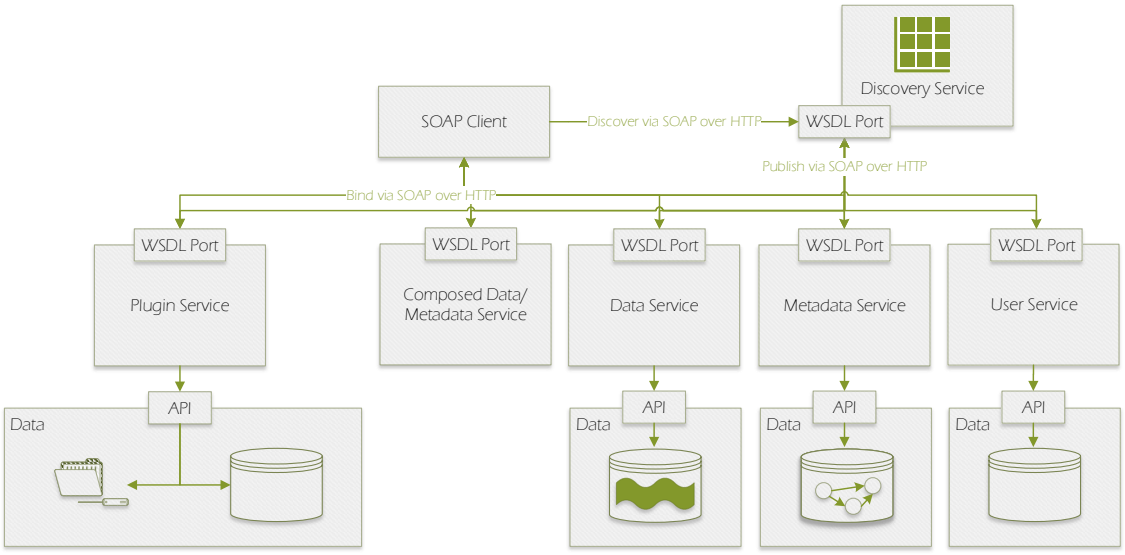


Figure 5.4: A service-oriented architecture for a graph-based metadata management of CAE data.

Figure 5.4 shows a service-oriented architecture which uses SOAP as message exchange protocol, WSDL to describe services and WS-Discovery as discovery mechanism. The architecture consists of SOAP clients, a service registry and the specific service providers for the graph-based metadata management denoted as plugin service, data service, metadata service and user service. SOAP is used as message exchange format throughout the whole system. Service providers define ports described by WSDL definitions and broadcast service registration messages via User Datagram Protocol (UDP) in the network. The discovery service will receive these messages and process them by registering the service and its port definition. A SOAP client will send a look up request to the discovery service requesting a service with a specific set of capabilities. The discovery service looks up matching port types and returns the port which uses this port type. The client will use the port to directly communicate with the target service in an RPC device by supplying necessary parameters and directly invoking defined operations.

A user service is used to create and manage user entities in an external database as well as to provide and authenticate access tokens to clients. The tokens can be included in the header of a SOAP envelope when binding to a port by using the WS-Security standard. Each service can use the user service to verify the validity of the sent access token. User data as well as access tokens will be stored in an external SQL database to conform to the statelessness property of SOA.

A metadata service provides CRUD functionality for metadata which will be stored in an external graph-database. The WSDL port definition provides information about all supported operations which includes a *createOntologyFromFile* operation to create an ontology based on a provided ontology file from the client. Additional operations which depend on the ontology can also be created in the future by implementing them in the service and adding its port type to the service definition. The metadata service publishes a registration message which will lead to the registration of the new port type. Once this process has been done, new clients will be able to use the new port type making system extensible and evolvable.

A data service publishes an interface to upload and download CAE data that will be stored in an external data lake. This can be done by sending the CAE data file as SOAP attachment on a SOAP message. The data service can implement a junked upload by expecting a unique file group name as well as a file part number. The SOAP client will have to split the file in smaller junks send a new SOAP message with parts of the file which will be merged on the service making it stateful component. Although, this goes against the statelessness property of SOA and limits its scalability, it will improve the data upload for SOAP clients as they only need to implement a retry mechanism that only needs to consider the remaining junks instead of the whole file in case of connection failures.

Additionally, a composed data and metadata service is available which is used to perform multiple operations of the dedicated data and dedicated metadata service in a single request. This allows to upload a single file and to create metadata for this file at the same time by reusing the underlying data metadata service. The intention is that CAE data shall not be uploaded without additionally creating a corresponding metadata node in the graph database. By using an endpoint where data as well as metadata is required, this constraint can be enforced. To further prohibit normal SOAP clients from using the original data and metadata service ports a role-based authentication mechanism is introduced. Access tokens will be issued with a user role while the composed service uses an access token with a more privileged role. This way the architecture is open for extension to support future SOAP clients that might need to directly use the data or metadata service by issuing them access tokens with higher privileges.

The plugin service provides operations to query available plugins and their information as well as to start and stop plugin processes. The plugins will reside on an external filesystem and will have to be downloaded to the filesystem of the service when they need to be executed. Additionally, any information about running plugin processes as well as their output will be stored in an external SQL database to be stateless. The plugin service needs to provide a platform for any supported plugin type that shall be supported. Alternatively, plugins can be defined directly as services themselves and publish their own interface definition. This will make them even easier to discover and exactly define the input parameters they require. Unfortunately, this would also mean that the creator of the plugins needs to understand and implement SOAP and WSDL as well as any necessary WS-* technologies. Either way, the power of SOAP comes into play when using asynchronous communication to run the pugins so that they are not blocked. This is made possible by using separate HTTP connection as a callback to receive the answer once the process has finished. However, this also means that

there will be no intermediate status updates of the running processes since the callback will only be executed once the asynchronous communication has been finished. Given the fact that plugins might need to run ETL steps of thousands of files, it might take a long time for a plugin to finish its execution. This means a SOAP client requesting the operation is not blocked during the execution of the plugin and is receiving a callback once the process has finished.

The combination of standards allows to create an architecture of independent services that communicate via standardized interfaces. The resulting system is a RPC style SOA architecture which has clearly defined interfaces to describe a document or procedure-oriented information exchange. By using a discovery service, the discovery functionality is extracted to a separate dedicated service which creates a catalog of active services to be used enterprise wide. By using an architecture which follows industry standards additional enterprise services can also be integrated or the graph-based metadata management system itself can be integrated into a bigger enterprise wide SOA landscape. Additionally, the standardized interfaces allow to operate the graph-based metadata management across company boarders and the services can be made discoverable to clients of external contractors. The XML based definitions and messages abstract the underlying platform and language which provides developers of SOAP clients as well as developers of SOAP services the freedom to use their programming language and platform of choice. Unfortunately, this freedom comes at the price of performance, as the XML format has to be parsed and generated which will result in slower communication as opposed to a pure RPC API as presented in the N-Tier architecture.

A big difference to the previously presented client-server architectures is how services are discovered. Instead of having a RPC API at a specific network address the service itself publishes its ports to a discovery service which can be used by the clients to look up the address of a service which publishes an interface that matches the client's requesting message. The additional flexibility in discovering services requires additional communication with the discovery service prior to the actual message exchange which adds a small delay to service calls. However, the dynamic registration and discovery of services makes the architecture extensible to future services and clients and allows the architecture to evolve. The statelessness of the components allows to deploy multiple instances which means the services scale horizontally and therefore, even support large user groups which is a common occurrence in enterprises. By using composed services, certain operations of different services can be reused to create new functionality without much effort. However, each non composed service can be deployed and maintained individually which means the availability of the overall system is not taking a hit if a single service is unavailable. The additional extensibility of SOAP also allows to use the big WS-* technology stack which includes standards for many quality of service attributes that can be used with SOAP.

However, the same technology stack can make using SOAP very difficult as the communication will become quite complex rather quickly. This is further conditioned by the lengthy nature of the well-defined XML syntax which uses opening as well as closing tags which can be arbitrarily defined. The use of standardized communication via SOAP also means that clients are less flexible in how they can communicate, and they need to use a rather complex and bloated technology stack which makes developing additional clients a very complex task. The distributed nature of the system as well as the dynamic registration of the services also decrease the testability of the overall system.

5.3 Representational State Transfer

REST is an architectural style introduced in the dissertation of Fielding in 2000 for creating distributed hypermedia systems [FT00]. While the architectural styles of client-server and SOA focus on providing business functions to clients, REST is focusing on managing resources [PWA13]. It is a combination of different network-based architectural styles, including the already presented client-server style, but adds additional architectural constraints to define a uniform interface [FT00].

The core idea of REST is to provide constraints to create scalable architectures with a uniform interface to manage resources on a remote server. Resources are any kind of information that can be mapped to a name and are uniquely identifiable by a URI. They are represented by a data format which can be structured (e.g. XML or JSON) or unstructured (e.g. binary) and act upon by sending actions as control data to the URI of the respective resource. Additionally, the request also includes data to specify the representation of the provided or requested resource which is known as media type. The request must also enclose any data that the server requires to process the request such as authentication information. Requests from RESTful servers need to provide additional URIs which indicate further actions on resources that are supported by the server after processing the information of the last request. [FT00]

An implementation of the REST architectural style is called *RESTful* and needs to satisfy the following six architectural constraints.

1. Client-Server

   The client-server constraint enables a separation of concern which decouples the user interface from the data storage. This means client and server can scale and evolve independently. [FT00]

2. Stateless

   This constraint says that the communication in a RESTful architecture shall be stateless which means, each request must provide any information about the context that the server requires to process the request. Therefore, state information need to be maintained on the client side. This enables horizontal scalability because additional servers can be deployed and directly used by the clients as every request is self contained. [FT00]

3. Cache

   The cache constraint defines that a RESTful architecture shall allow responses to be marked as cacheable. This allows the client to maintain a response cache which decreases network communication with the server. This has the benefits of better scalability and performance of the overall system. [FT00]

4. Uniform Interface

   REST has the constraint of a general uniform interface to improve visibility and to decouple the implementation from the provided service. While this comes at the cost of data transformation from translating to and from a common data format it improves the evolvability of the system by being self descriptive and exploratory. The interface itself is defined by the following four constraints a) Resource identification, which lets a client uniquely identify a resource by a URI b) Resource manipulation, by using a predefined set of actions to indicate the state change of a resource. c) Self-descriptive messages, which means any information that the

server needs to process a request is included in the request itself. d) Hypermedia as the Engine of Application State (HATEOAS), which means hypermedia links shall be enclosed in the response of a RESTful server to indicate the client any actions that are supported in the systems current state. [FT00]

5. Layered System

   This constraint means that a RESTful architecture is implemented as a layered system which means clients and servers can only communicate with the layer that is directly above or below them. Therefore, clients will not be able to distinguish if a response has been produced by the server it communicates to or if it has been produced by a different server. Same goes for the servers in terms of the requests. While this enables additional scaling capabilities for the system, the additional routing of requests means increased latency for requests and responses. [FT00]

6. Code-On-Demand (optional)

   This constraint allows the server to dynamically extend the functionality of the client by sending code to be interpreted and executed on the client side. However, it is an optional requirement because it decreases visibility of the data to be downloaded. [FT00]

Compared to the previously presented architectures, business functions need to be addressed as resources when creating a RESTful architecture. The business functions of the graph-based metadata can be extracted as 1. CRUD functionality for CAE data 2. CRUD functionality for metadata nodes and relationships 3. CRUD functionality for ontology classes, properties and relationships 4. CRUD functionality for users 5. Management functionality for plugins and plugin processes, specifically starting and stopping plugins as well as querying the status of running plugin processes. Resources are any components that need to be identifiable and linked to which means, a resource can be created for each noun of the previous list. Therefore, the resources are *Data Resource*, *Metadata Node Resource*, *Metadata Relationships Resource*, *Ontology Node Resource*, *Ontology Property Resource*, *Ontology Relationship Resource*, *User Resource*, *Plugin Resource*, and *Process Resource*.

However, since this is also a layered system the resources do not necessarily have to reflect the underlying composition of the system in said components. Figure 5.5 shows a concept of how the components can be distributed in the overall system and how they are connected. The system is a client server architecture that consists of multiple components communicating via HTTP. This way the system can be deployed as 2-Tier, 3-Tier or N-Tier system and the HTTP verbs POST, GET, PUT and DELETE can be mapped to the CRUD functionalities. Additionally, POST and DELETE can be mapped to start and stop a plugin process. Each component in the business and process layer is stateless by storing their application state in an external storage system in the data layer. The user module is responsible for the management of users which represent the clients of the system. It exposes CRUD functionalities via a RESTful API over HTTP to easily register new users and to modify or delete existing ones. The managed users are stored and retrieved from an external SQL database, to achieve the stateless property of REST. Caching can be applied when storing entities but the cache needs to be invalidated whenever a user is modified or deleted. The authentication module exposes an authentication endpoint to generate and validate API tokens for users. API tokens are used in this system because the components shall be stateless and therefore, require any information necessary to process incoming requests. This mean users can not just log in once and a session is created but rather have to send their API token whenever they make a request. The same external SQL database is used to store the API tokens where the users are

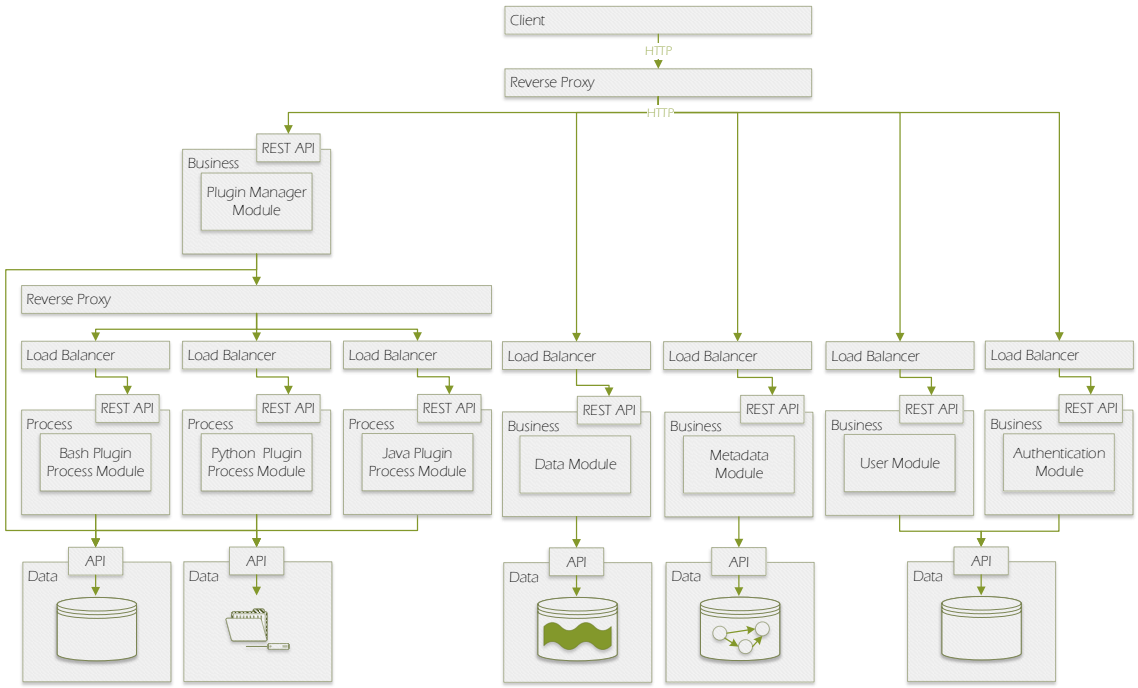


Figure 5.5: A RESTful architecture for a graph-based metadata management of CAE data.

already stored by the user module. The metadata component exposes functionality to manage graph-based metadata in form of nodes and relationships as well as ontologies via a RESTful API. Ontologies will be stored in the same graph-structure as the actual metadata and is stored as class, property and relationship nodes which represent a class hierarchy of graph-based metadata nodes, their properties and which relationships they have. The graph database is a separate component to fulfill the stateless property on the metadata module. The data module is used expose a RESTful API to upload and download CAE data which will be passed to an external data lake to prevent session state on the data module. Since this is the main functionality of the graph-based metadata management system, a load balancer is used in front of the data module and the metadata module which evenly distributes requests. This way multiple upload and download requests as well as metadata requests can be handled simultaneously by the deployed instances.

The plugin functionality is divided in a plugin manager module which communicates with plugin process modules. The plugins themselves are stored on an external filesystem which could also be part of the data lake but as the modules are not directly CAE data items to be managed, they are on a separate filesystem in this architecture. The plugin manager uses an API of the filesystem to be able to list plugins and to expose functionality to start, stop and query the status of them in form of a RESTful API. The plugin process modules are used to execute a specific plugin type such as Bash, Python or Java plugins. They expose functionality to start and stop a plugin process of a specific plugin type as well as to query the output or any errors that occurred in form of a RESTful API. To be stateless, they download the plugin to be executed from the external filesystem and write any process information to a SQL database. They also write any plugin output into files which are stored in the same directory of the external filesystem where the plugin is stored. A link to the files will be stored in the SQL database to correlate process information to output files. By including the

plugin type in the URI of a plugin process resource, they dynamically register as plugin process executor to the plugin manager module. To do so, the plugin manager receives a request to execute a specific plugin where the plugin manager module will determine the type of the plugin to be executed. It then forwards the request to a Uniform Resource Locator (URL) which includes a URI with the plugin type and includes the name of the plugin to be executed in the request. If a plugin process resource is available on the specified URI it will download the requested plugin and start it locally and return a response that includes a process Identifier (ID) for the plugin process he started. If no plugin process resource is available on the specified URI a response will be sent to the client describing that this plugin type is not yet supported. To start multiple plugins of the same type simultaneously a reverse proxy and a load balancer will be in between the plugin manager and the specific plugin process modules. The reverse proxy is used to forward requests of specific plugin types to their respective load balance which evenly distributes the request on plugin process instances of the specific plugin type.

Each module is using a token based authentication mechanism and uses the authentication module to verify the validity of the token. Caching can be implemented on each module except for the data module because large files can be uploaded to the data module which in turn might run out of memory. To create the illusion of a single server with a uniform interface a reverse proxy is used between the graph-based metadata components and the client. This allows the client to have a single server address where all its requests will be sent to while having a highly distributed and scalable system. Any client requests can be distributed to dedicated servers that implement a load balancer which in turn further forwards the request to the target sever. This makes the components decoupled from each other which means each component can be built, developed, tested, deployed and scaled individually. Plugin process executors for specific plugin types make themselves dynamically available to a plugin manager which means new plugin types can be supported in the future by developing and deploying a new plugin type process executor without further registration and management process. Likewise, new components can be dynamically made available for the user to explore by simply deploying a new component and registering it on the reverse proxy.

Therefore, the architecture is great for extensibility, evolvability, scalability and availability. However, the biggest drawbacks are the additional administration and management tasks that arise with the individual deployment, configuration and maintenance tasks of each service. Similarly to the N-Tier architecture, there is also a lot of communication overhead since requests need to be routed between different modules quite often. This can be solved by deploying the individual modules on the same or physically close servers to reduce the latency of the additional communication. Compared to the N-Tier architecture, this architecture will also perform slower since many transformation tasks have to be done by parsing the request formats to achieve the loose coupling between the individual modules. The testability of the system is also a trade-off between simple testing of each individual modules as they are isolated but a complex and cumbersome task to test the integration of the modules on the overall system.

5.4 Microservice Architecture

The microservice architectural style is an evolution of SOA that focuses on developing independent and small services where each service is running in its own process and communicates via a lightweight communication protocol [FL14]. While there is no specific definition of the microservice architectural style, Fowler and Lewis (2014) have extracted the following key characteristics of the microservice architectural style.

1. Services as Components of Larger Systems

   Traditionally, large software systems are developed by splitting functionality of the software into smaller components. Components could be developed independently which prevents small changes in one component to interfere with the development of other components. Microservices take this approach a step further by extracting each business functionality into a separate service which can not only be developed independently but also deployed and scaled on its own. [FL14]

2. Organized around Business Capabilities

   While traditional application development is based on the organizational structure of the enterprise, microservices focus on organizing a cross-functional developing team around the business capabilities. This leads to microservices that do not follow the organizational structure and therefore, business functionality is split instead of creating a siloed application architecture. [FL14]

3. Products instead of Projects

   Traditional approaches of developing software are managed as projects which are abandoned once a *definition of done* is completed. However, the microservice approach makes a developing team responsible for the product they create, for its whole lifetime. This leads to a longer maintenance, faster roll-outs and better support of the product. [FL14]

4. Smart Endpoints and Dumb Pipes

   While the previously presented SOA style requires a lot of effort to be put into the communication between services, the microservice approach tries to be as decoupled and as flexible as possible. This is done by supporting as many formats as possible and doing any transformation logic on the endpoint rather than along the way of the communication. [FL14]

5. Decentralized Governance

   Microservices aim to provide flexibility to the developers by not using a single standardized service contract but rather provide patterns to combat reoccurring problems which are shared publicly. This provides other developers with a large framework and toolset to combat reoccurring problems by using different technologies. [FL14]

6. Decentralized Data Management

   Data can be organized into different domains which have bounded-context. In a traditional system with a single data store this can be achieved by creating separate managed entities while the microservice approach often uses a different data store to store each domain specific data. This makes the data context more explicit but imposes some challenges regarding

consistency as different services might not have a consistent data view. Strict consistency is sacrificed for eventual consistency and improved availability in the decentralized data management of the microservice approach. [FL14]

7. Infrastructure Automation

   Architectures based on microservices tend to use Continuous Integration (CI) and Continuous Delivery (CD) to automate the deployment of the additional modules. By using these techniques the additional development and deployment overhead introduced when adding additional microservices, is kept to a minimum. [FL14]

8. Design for Failure

   Microservices provide business capabilities to clients but they might fail at any time due to their environment. Network based microservices might become unavailable due to connection issues and local microservices might fail due to an failure of the underlying OS. Compared to the more traditional approach where components are developed in a more monolithic architecture, microservices need to be built to be resilient to service failures. Microservice architectures are typically using advanced logging and monitoring tools to quickly find and compensate failing microservices. [FL14]

9. Evolutionary Design

   Splitting components into separate decoupled services allows an architecture to evolve. Additional functionality can be included by developing additional microservices which can be developed, tested and deployed independently of the overall system. [FL14]

In the microservice architectural style, clients communicate via an API with microservices which is also used by the microservices themselves. There are two main API designs to look at when speaking about microservices which are message-oriented and hypermedia-driven APIs [NMMA16]. The former describes that microservices pass messages to exchange information which is inherited from an object-oriented design approach [NMMA16]. Messages can be described in a structured format like XML or JSON similarly to the already presented SOA architecture. However, the Hypermedia-driven approach adds additional information to the data contained in the message-oriented approach by specifying a format that includes hypermedia links to allow a client to explore additional actions upon receiving a hypermedia-based reply [NMMA16]. This is essentially how the web is working as hypermedia links are already embedded in Hypertext Markup Language (HTML) documents which allows the user to explore additional actions by following the links [NMMA16]. One specification of a hypermedia-driven API design is REST which is also a common way to build APIs for microservices. In the following two sections a microservice-based architecture will be presented for either approach.

5.4.1 Message Oriented Microservices using a Message Oriented Middleware

Microservices are built around the idea of loose coupling which is hard to achieve by using an information exchange protocol that is build on top of HTTP. The synchronous nature of HTTP only allows a request-reply communication. Therefore, a client will block and wait for a reply when communicating with a server which means, they are tightly coupled. The REST architectural style can be used to decouple the availability of sender and receiver by creating resources for long-running tasks. It is a client's responsibility to query the created resource to receive any updates about its status.

In this approach, the client is responsible to request updates about long-running tasks on the server. A different approach is to use a subscription service on the server where clients can subscribe to receive updates about a long-running task which are sent by the server once an update occurs. To implement this approach a Message Oriented Middleware (MOM) is commonly used.

A MOM supports distributed communication via asynchronous message passing by using message queues. Instead of a direct communication between client and server, messages are sent to a message queue of the MOM where they might be consumed or passed between many participants which could inspect, modify or delete the message. A MOM supports reliable communication in the sense that messages will not get lost during communication by using a store-and-forward approach but it is neither guaranteed that a message has a receiver nor when the message will be delivered. Messages are stored in queues which are referred to as message channels that can either be used as a point-to-point (many-to-one) or publish-subscribe (many-to-many) communication model. In a point-to-point channel many clients can send messages and each message gets consumed only once. However, many consumers can connect to the message channel at once allowing to horizontally scale the communication. A publish-subscribe channel can be used by many producers to publish messages on specific topics. Multiple consumers can subscribe to topics they want to receive messages from and each gets a copy of the published message upon arrival. [Cur04]

By using the store-and-forward approach of the MOM, producer and consumer don't have to be available at the same time since messages can still be delivered once a consumer becomes available even if the producer has become unavailable after sending a message. This highly improves the overall availability of the system compared to direct communication of the traditional client-server model. However, asynchronous communication might be slower and therefore, messages will not be delivered immediately.

Figure 5.6 shows a concept of a message-based microservice architecture for the graph-based metadata management which uses a MOM to distribute its messages. In this architecture communication is decoupled between senders and receivers by using messages as communication medium and queues as communication channels provided by a message broker. The message broker includes tools for managing users and adds user authentication to the queues. The communication protocol is defined by a common queuing protocol such as Advanced Message Queuing Protocol (AMQP). The messages need to indicate their purpose by specifying an action flag that represents a CRUD action. A correlation identifier can be used if return message is expected which models a request-response communication pattern. Clients that want to use the graph-based metadata management system will have to send messages to queues in order to perform actions or query data. They also need to specify a flag to indicate the responsible subsystem which needs to be one of data, metadata, or plugin. All messages can have a data payload and depending on the target subsystem they might require additional fields.

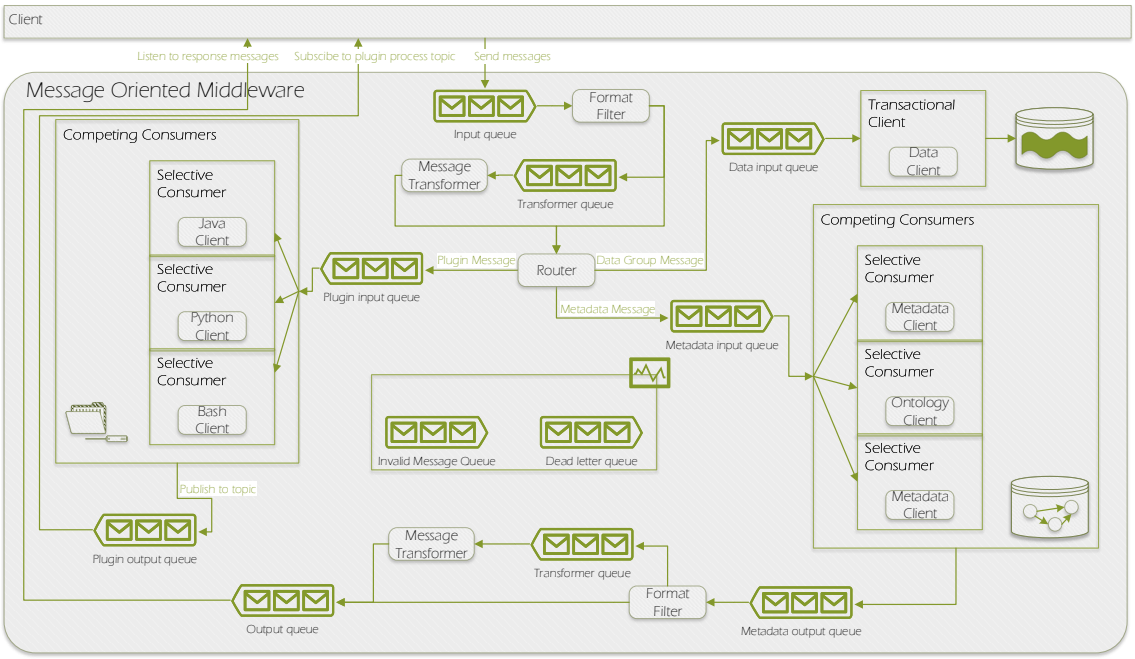


Figure 5.6: A messaging-based microservice architecture for a graph-based metadata management of CAE data.

To make the system as easy to use for clients as possible, any incoming communication from the client will be sent to a single entry in form of an input queue. These include command messages to either change or query the state of the system. A format filter service is connected to the input queue to consume the messages and to check the message format of any data items. Messages that include data in a format other than JSON will be sent to a transformer queue which is connected to a data transformer. The data transformer will transform any known message format into a JSON format and adds a flag for the original format. Formats that are unknown to this service will be disregarded into an invalid message queue which needs to be monitored. Successfully transformed messages will be sent to a message router. Messages that are in the input queue and their representation of data matches the JSON format will be sent to a routing service which determines the queue of the subsystem, the message has to be sent to. Each of the subsystems input queues are point-to-point channels which means each message will only be received once.

Messages intended for the plugin subsystem will include the CRUD action flag that is mapped to the start plugin, query plugin, restart and stop plugin actions. The message data payload will include the name of the plugin to start and any input required by the plugin. The plugin subsystem consist of a plugin service for each supported plugin type such as Bash, Python or Java and are connected to the plugin input queue. There can be multiple instances of each individual service which are set up in a competing consumer scenario [HW04] which means they continuously poll the plugin input queue and the first service to poll after a message arrives, consumes said message. However, each service can only process messages of a specific plugin type which is why they need to filter messages out which they can not process. This is done by using a selective consumer pattern [HW04] where a consumer can peek a message without consuming it from the queue. Based on the peek, the selective consumer can consume and therefore, remove the message from the

channel or leave it as is [HW04]. This way a plugin service will only execute a plugin if it polls the queue which is only when it is not already executing another plugin. This ensures the plugin service works on its own pace and can not get overloaded. The plugins themselves are stored on an external filesystem and will have to be downloaded by the plugin service before it can be executed. The plugin services are also connected as publishers to a publish-subscribe queue which uses the plugin type, combined with the name of the plugin as topic to publish messages to. The client can subscribe to the plugins he is interested to and will get notified as soon as an update to the plugin execution occurs. Information about previously executed plugin processes will not be stored.

Messages intended for the data subsystem will include a group identifier as well as a position number. The intention is that the client will split CAE data files that shall be uploaded to the data lake into smaller file junks and sends them as data payload in separate messages. The messages will be routed to the data input queue where a transactional client [HW04] is used to make sure that either all file junks are committed to the data lake or none. This ensures the integrity of the data lake and allows the client to pause the upload of large files by delaying the sending of data messages. However, this makes the data service a stateful component which can not easily be scaled. Additionally, the file junks of the same message group need to be stored on the server until all messages of said group have arrived which can fill up the filesystem pretty fast. In order to disregard invalid message groups, they will only be kept for a set period of time otherwise the messages will be sent to the dead letter queue.

Messages intended for the metadata subsystem will be routed to the metadata input queue. Metadata as well as ontology metadata clients will be using a selective consumer pattern to filter which client can be used to handle the metadata message. These need to either include an action flag or a metadata query as data payload. Metadata services will consume messages with action flags that represent CRUD actions and process the data payload by communicating with the graph database. Ontology services will only consume messages without action flag and execute the data payload as query by using the information of an existing ontology which can be created by using the metadata services. By communicating with the external graph-database, the services are stateless which means they can run as competing consumers and can be scaled independently. By communicating with the metadata subsystem a request-reply messaging pattern [HW04] can be used to retrieve metadata from the graph-based metadata management system after requesting it via a message. This is achieved by using a correlation ID on the messages produced by the metadata and ontology services which references the message ID of the request. The messages will be sent to a metadata output queue where a format filter is connected to. The format filter consumes the messages and checks whether the message format of the requesting message matches the one of the response message. In case it does not match, the message will be routed to a message transformer which transforms the data payload of the message. In either case, the message will be sent to an output queue which the clients can connect to as a selective filter and only consume the messages with the correct correlation ID.

While the decoupling means greater resilience of the overall system, there are many opportunities for communication issues which is why the messaging system needs to monitor messages. This is done by sending any malformed messages into an invalid message queue and undeliverable messages, e.g. due to a timeout to a dead letter queue. Both queues need to be monitored and any messages inside need to be regularly checked.

The presented architecture decouples the data format of the producer from the data format of the consumer by providing additional dedicated message transformer services. Any incoming message will be transformed into the internal representation and any outgoing message will be transformed into the requested output format. Additionally, services can be scaled independently as they run as competing consumers which means they are polling for messages rather than getting messages pushed to them. This greatly improves the availability of the whole system because individual components can not be overloaded. While the system is highly distributed, any outside client will not be aware of the complex internal structure of the system because they can only address three queues. Each queue requires a client to be authenticated in order to send messages. This means the authentication level is more coarse-grained since any input of outside client will be sent to a single queue. It is the clients responsibility to provide a more fine-grained access control to individual parts of its application.

The asynchronous nature of messaging is well suited for dealing with the execution of the plugins on different services and running them on their own dedicated platform. New plugin types can be supported by adding a new plugin service and simply make it poll the plugin input queue which is great for extensibility and evolvability which was specified as requirement in Section 4.2. Command messages can be used to start, stop and restart plugins while any updates of the plugin processes will be published on a publish-subscribe queue which means subscribers will be immediately notified of updates. However, previous executions of plugins will not be available to clients that subscribe after the plugin has already been executed. The highly distributed system becomes quite complex to monitor as many messages are sent in many queues at the same time. This also means additional administration tasks as the messages in the invalid message queue and the dead letter queue needs to be checked frequently. The many individual microservices provide an additional challenge as they might fail at any time and the operator of the system needs to make sure these services will be restarted.

While the time autonomy of the producer and consumer is great for the plugin subsystem, the request-reply pattern needs to be used to communicate in a synchronous way with the metadata subsystem. This provides an immediate response to an output channel and therefore, a visualization client will not have to wait for the requested data. Unfortunately, any client that communicates with the graph-based metadata management system needs to be messaging aware and has to communicate via a messaging protocol instead of the more standard HTTP.

Overall, decoupling the components as seen in the presented architecture improves the availability and evolvability of the overall system by improving the extensibility and scalability of the individual services which is in line with the requirements and challenges defined in Section 4.2 and Section 4.3 respectively. The asynchronous nature of the system helps with the plugin execution but the use case *Visualizing Graph-based Metadata* requires a synchronous request-reply with immediate data feedback. By using this architecture, clients will also become very complex applications and the system can not easily be accessed by non-technical employees without a dedicated messaging client. Additionally, a special client to upload CAE data files is needed which supports the same algorithm as the data service for splitting and merging the data files. The individual services can be individually developed and tested but integration testing becomes a very complex task and additional monitoring of queues is needed throughout the whole operation of the system. This means a lot of complex administration and monitoring tasks are required when using this architecture.

5.4.2 Hypermedia-Driven Microservices using REST

The hypermedia-driven approach of microservices is an extension of the message-oriented approach where the messages contain not only data [NMMA16]. Instead, messages are enriched with possible follow-up actions in form of hypermedia links which is essentially what the web is built around [NMMA16]. This way the data as well as the actions are be loosely coupled which embraces evolvability [NMMA16]. Using REST as architectural style to create loosely coupled APIs for distributed microservices focuses on the overall evolvability of each API as well as of the extensibility of the system as a whole.

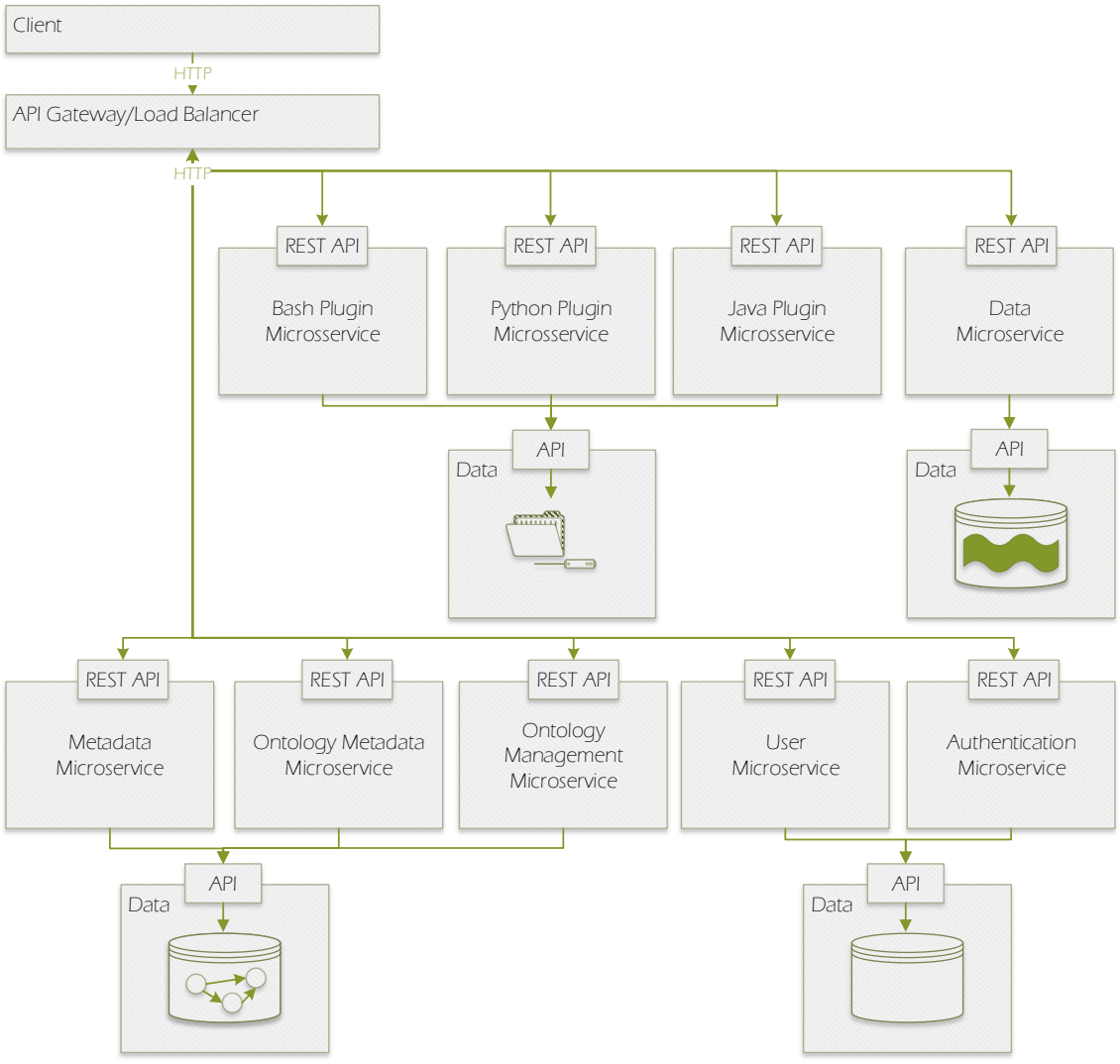


Figure 5.7: A REST based microservice architecture for a graph-based metadata management of CAE data.

Figure 5.7 shows a hypermedia-driven microservice architecture using RESTful APIs as communication interface of the microservices. The architecture is built similarly as the SOA architecture but uses a less strict model to describe its interfaces and uses a more fine-grained interpretation of how to separate business functions in individual services. The RESTful APIs are built around the same concept as in the previously presented REST architecture of Section 5.3 which uses resources to address data. The main difference is the more loosely-coupled nature of microservices and the idea that each microservice can be used independently.

The architecture uses a client-server and layered system style where clients will connect to an API gateway which acts as a reverse proxy and load balancer for all available microservices. This makes the architecture extensible by simply including routing information to additional microservices in the future. Each microservice exposes a RESTful API and follows the rule of *smart endpoints and dumb pipes*. This means that the API shall support content negotiation where a client can send a request in a number of different formats and the microservice will do the transformations. Additionally, each microservice is responsible to provide an authentication mechanism by itself.

In this architecture, the authentication microservice provides endpoints to create and invalidate API tokens by using basic authentication. This means HTTPS should be used to communicate with the authentication microservice as basic authentication involves sending a base64 encoded username and password as plain text to the server. An issued API token will have a limited validity and will be stored along user credentials in an external SQL database. The token will be included as authentication header in every request by the clients and the microservices use the user microservice to check whether the token is valid for an existing user. The user microservice provides endpoints for CRUD operations on users. A user can use these endpoints to read and update its status by providing a valid API token. This will be used by the other microservices to check whether a provided API token is valid. The user microservice's create and delete operations shall only be available to administrators to prevent misuse.

A data microservice provides endpoints to allow the upload and download of data files which are getting streamed into a data lake. Therefore, the files do not have to be cached by the data microservice which is helpful to prevent out-of-memory errors when dealing with large files as it is the case with CAE files. However, no caching means that the uploaded files will need to be re-uploaded when a connection issue appears during the upload. This means the client is responsible to notice connection failures and to implement a retry mechanism to ensure the file is completely uploaded.

A metadata microservice is providing CRUD endpoints for the graph-based interaction with the metadata where the nodes and edges can be seen as the resources. The nodes as well as the relationships resource provide a model for data upload that include a named node or relationship type and key-value properties. Additionally, a start and an end node need to be specified when creating a relationship. However, the metadata functionality is completely decoupled and independent of the data functionality which means its the clients responsibility to create a metadata node whenever a data file has been uploaded. The client will also have to manage how to link the data files with the graph nodes that represent the metadata.

Additionally, the system includes an ontology management microservice which provides the CRUD functionality for ontologies. One way of doing this is to use expose CRUD endpoints for ontology nodes and relationships that work similarly as the metadata CRUD endpoints. Alternatively, files could be uploaded that represent the ontology in a common ontology format. This way a single API

call could be used to create the whole ontology on the server instead of multiple consecutive calls with the CRUD endpoints. Either way the ontology will be created in the same format and in the same external graph database as the metadata which makes it consistent in the way it can be used and managed.

A separate ontology metadata microservice provides additional endpoints for the functionality that comes with the use of an ontology. The classes defined in the ontology will be used as resources to quickly retrieve all nodes that are either the class itself or a subclass of it. Any additional functionality that comes from the use of an ontology can be modeled similarly and can be included in this microservice. It is also a stateless component as the information it needs from the ontology as well as the metadata are stored in the same external graph database.

The plugin functionality is separated into individual plugin microservices where each of them is responsible to support a single plugin type. This way the architecture supports additional plugin types by adding new microservices for specific plugin types in the future. The initially supported plugin types are Bash, Python and Java and each microservice provides a platform where a plugin of this type can be executed. This means the microservice itself could be written in the same programming language as the plugin type it supports since its runtime environment is already present. The plugins themselves are stored on an external filesystem and will be downloaded by the plugin microservice whenever a plugin should be started. The API of a plugin microservice of a specific type will use the plugins of said type as its resource. E.g. the Java plugin microservice will use *java-plugins* as collection resource. Whenever the resource is called via a HTTP GET request, it will use an API of the remote filesystem to retrieve a list of all available plugins that this specific plugin type supports. A plugin can be started by using a HTTP POST request where the plugins name will be passed as parameter. This will create a new resource on the server which represents the plugin process which means each plugin can have exactly one process running at any given time. Using GET or DELETE requests on the newly created resource will retrieve information about the running process or stop the execution of the running process respectively. The information of the running process can be retrieved on the microservice by using OS and platform dependent tooling.

This architecture is quite similar to the previously presented RESTful architecture in the way the components interact with each other and by using RESTful interfaces. However, the microservice architecture focuses more on the separation of concerns and decoupling the functionality of the components. Each microservice implements a separate business functionality and its the clients responsibility to correctly call the microservices in an order that supports the overall goal. E.g. it is the clients choice to either upload a CAE file to the data microservice and afterwards create a virtual metadata node by calling the metadata microservice or to do it vice versa. This is where the evolutionary design philosophy of microservices comes into play which requires a certain decoupling of the individual microservices to allow the system to evolve.

Unfortunately, this transfers a lot of responsibility to the clients which have to make sure to use the microservices in the right way to achieve their use case with the graph-based metadata management system. E.g. it is the clients responsibility to make sure metadata is only created once a CAE data file is successfully uploaded or to implement a retry mechanism for the data upload if the metadata is created beforehand to ensure the integrity of the CAE data and their corresponding metadata nodes and relationships.

An automated deployment mechanism is a key component used in microservice architectures to ease the deployment and management of the individual microservices. By using common software engineering tools, the microservices deployment configuration can be managed as Infrastructure-as-Code (IasC) and helps to quickly set up different environments of the whole system. By using the microservice mindset, integration testing can be done by running a test environment of the whole system to test in a near production environment without any additional management overhead. Additionally, the decoupling of the microservices makes them easy to test individually by using unit tests specific to its business functionality.

Overall the architecture focuses on the separation of concerns by splitting business functionality into decoupled microservices. An API gateway is used to provide a single server as entry point to show a unified system to clients. This makes the system extensible and evolvable by providing horizontal scaling on individual microservices. The communication between clients and target server is always a 2-step connection due to the gateway. By additionally using RESTful APIs on the microservices, a uniform interface has been created for the graph-based metadata management system. However, future functionality could also be included by using other interfaces or even other communication technologies such as a MOM and integrating them with the API gateway. However, the decoupling of the microservices comes with the flexibility of the clients to use the functionality of the microservices independently and therefore, transfers a lot of responsibility to the clients to use the graph-based metadata management system as intended. E.g. the requirement of linking files to virtual nodes defined in Section 4.2 is supported by the architecture but its the clients responsibility to call the required microservice in sequence and handle any errors that might occur between them.

5.5 Evaluation of Architectural Styles and Final Architecture

The previous sections presented architectural concepts to create a software architecture for the graph-based metadata management by using different architectural styles. Each architecture has its advantages and disadvantages which need to be considered when choosing a final architecture.

The client-server architecture uses a direct connection between client and server and follows the request-reply pattern making it a good choice when dealing with synchronous scenarios like querying and iterating over metadata as described in the use cases *Linking CAE Data with Graph-based Metadata* and *Visualizing Graph-based Metadata*. A visualization client could query information about a particular metadata graph node and display it upon arrival of a server's response. Whenever a user selects another node it requests a different node and therefore, allows to iterate over the metadata graph nodes giving immediate feedback to the user. However, the presented 2-Tier and 3-Tier architectures of Section 5.1.1 and Section 5.1.2 respectively, lack extensibility, evolvability and scalability. The presented N-Tier architecture of Section 5.1.3 helps in these aspects, however users have to manually choose a different server instance on their own in case their first chosen instance is overloaded. The XML/JSON API that the clients use provides a single interface to the graph-based metadata system but does not enforce uniformity. Additionally, these client-server architectures are not suitable for asynchronous scenarios. Therefore, a request for a plugin execution might block the client for a long period of time.

The SOA architecture presented in Section 5.2 introduced the notion of services that expose an API with a well defined contract. Separating the business logic into individual services allows a more decoupled distributed system. This improves the scalability of individual businesss functions. The extensibility and evolvability are also improved by using a discovery service to dynamically register and de-register services. Using SOAP and WSDL as tool set provides a standardized way of defining service requests and endpoints. SOAP can also be used in synchronous as well as asynchronous use cases making it a good choice for all defined use cases of Section 4.1 However, by using the WS-* technology stack, the development of the services and their communication pipes can become very complex. Additionally, clients need to support the SOAP protocol which only supports the XML format and require a custom client to be created for each use case.

The REST architecture presented in Section 5.3 makes the shift to *smart endpoints and dumb pipes* which makes it usable by existing HTTP clients such as simple web browsers. The architecture is extensible, evolvable, scalable and provides a uniform interface in form of RESTful APIs. However, the individual components do not necessarily need to separate any business functionality into services but are rather grouped around their managed resources. Similarly to the SOA architecture, both architectures come with a big administrative overhead to manage, test, deploy, configure and maintain all components and their connections.

The microservice architectures presented in Section 5.4.1 and Section 5.4.2 respectively show that services shall have a very limited scope that is organized around business capabilities. The development of small services also leads to a microservice mindset which requires lots of automation in order to cope with the many individual services. This is where CI/CD comes into play which deals with the problem of administration overhead of the SOA and REST architectures. While the message-oriented microservice architecture excels especially at asynchronous scenarios, most of the defined use cases of Section 4.1 are rather synchronous scenarios. It also requires custom clients and has limited security options and while it has a single interface in form of a message queue, it is not uniform. The hypermedia-based microservices approach does provide the uniform interface but uses an API gateway to make the microservices communicate with one another and to give a single interface for clients to the graph-based metadata system. This brings the downside of having each microservice exposing its functionality via a public API.

The REST architectural style and the hypermedia-based microservice approach have the best trade-offs to support the use cases and their requirements defined in Section 4.1 and Section 4.2 respectively. With this in mind, a final architecture has been created that makes use of the layered client-server communication of REST and uses the microservices approach to keep the scope of each service limited to its business functionality. Using the microservices approach also implies to use CI/CD to automate the integration and deployment process of new software changes.

Figure 5.8 shows an overview of the final architecture which is in many ways similar to the RESTful architecture presented in Section 5.3. The plugin functionality is split into the already introduced plugin manager and specific plugin type executors which are addressed by a reverse proxy that points to load balancers which enables scaling of each executor of a specific plugin type. As shown in the lower left part of the figure, the plugins reside on a remote filesystem and any output of plugin processes will be written to an external SQL database. This way, each of the plugin related microservices can be developed as stateless components. The plugin manager component manages the *plugin* resource and the abstract *process* resource for all processes of any plugin type. The *process* resources for a specific plugin type are handled by the specific plugin type modules. A HTTP POST method is used to start a plugin by giving the plugin name as path parameter such as

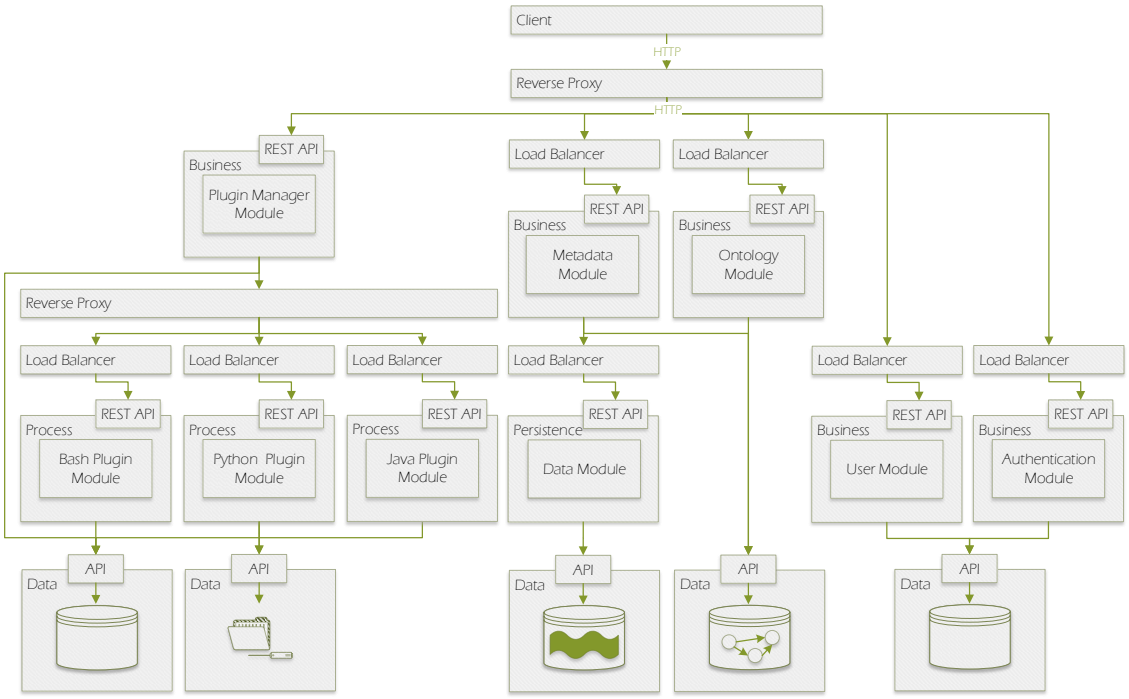


Figure 5.8: A REST based architecture for a graph-based metadata management of CAE data with the advantages of microservices.

/plugins/transform.jar. The plugin module extracts the plugin type and requests the URI resource for it which could be named */jars/transform/processes*. HTTP GET requests retrieve information about the plugins and their running processes and HTTP DELETE requests will stop the execution of a running process. This architecture uses a pull model which means that clients are responsible to inform themselves about the state of resources by using polling techniques.

The user module, on the right side of Figure 5.8, manages the *user* resource and maps the CRUD functionality to the HTTP methods POST, GET, PUT and DELETE respectively. However, a role-based authorization needs to be implemented to prevent any user to create new users or to even delete arbitrary users. The authentication service is used to provide an endpoint to generate API access tokens which have to be included as bearer tokens for every request that clients make to the graph-based metadata management system. Tokens will be seen as a resource and can be created by providing username and password in a HTTP POST request. To invalidate a token, the HTTP DELETE method is used. Both services communicate with an external SQL database to store user credentials and tokens.

The data module, in the middle of Figure 5.8, is a self contained reusable service in this architecture and manages the *file* resource by using an external data lake. The CRUD functionalities for this resource are also mapped to the HTTP methods POST, GET, PUT and DELETE respectively. However, compared to the hypermedia-based microservice architectures, the REST API of the service is not publicly available. The data microservice is used by a dedicated metadata and a dedicated ontology microservice to provide a metadata based API which also can make use of data access capabilities. The metadata module manages the metadata graph *nodes* and *relationships*

as resources and has the CRUD capabilities of the resources mapped to HTTP POST, GET, PUT and DELETE methods respectively on its RESTful API. However when using these capabilities to create a new metadata node, a CAE data file can be uploaded which will be passed to the data service. Once the file has been successfully transfered to the data component its file location will be returned to the metadata service which automatically creates a new metadata node with the file location as property. Therefore, the data upload is part of the metadata management concept and the integrity of the data files and their correlating metadata nodes is guaranteed. Likewise, deleting a metadata node will remove its CAE data file from the data lake by using the data module. The ontology module is used to provide CRUD functionalities on ontology related resource which are *classes*, *properties* and *relationship*. Similarly to the previous API endpoints the functionality is mapped to the HTTP POST, GET, PUT and DELETE methods respectively. Additionally, it bridges the gap between metadata nodes and its defined classes by also managing *class instances* as resource. For example, the endpoint to show all instances of a defined metadata class or any of its subclasses can be implemented by the URI */classes/{classname}/instances*. By using hypermedia links, class instances can be referenced by ontology class resources and vice versa.

Each microservice can be scaled independently by using load balancers which is why a reverse proxy is used to make sure a single server can be used as interface to the graph-based metadata management system. Implementing a RESTful API for each microservice has the advantage that each microservice is decoupled from its clients and could be reused in a different location. This allows the architecture to evolve over time and to easily integrate new microservices which reuse existing services.

6 Architecture Implementation

Different architectural concepts using multiple architectural styles were introduced in Chapter 5, each having their advantages and disadvantages. The architectural styles were discussed and a concept of an architecture was presented that is most suitable for our use cases and supports our requirements defined in Section 4.2. In this chapter, a prototypical implementation of this architecture is described as a functional proof of concept that supports the graph-based metadata management of CAE data. Section 6.1 gives a detailed insight about the seven components that make up the implemented prototype. In Section 6.2, the use of hypermedia links in the implementation is explained in more detail. Section 6.3 describes the automated build process that was created. In Section 6.4, the implemented tests and deployment techniques are described. Section 6.5 shows which types of documentation are available for the prototype. Section 6.6 describes the overall development workflow and the implemented continuous integration process.

6.1 Components

The microservice-like architecture makes the implementation of the individual microservices platform and language independent. This means any programming language supporting web technologies as well as any platform supporting such programming languages can be used to implement the prototype. Furthermore, each microservice can be developed in a different language and platform.

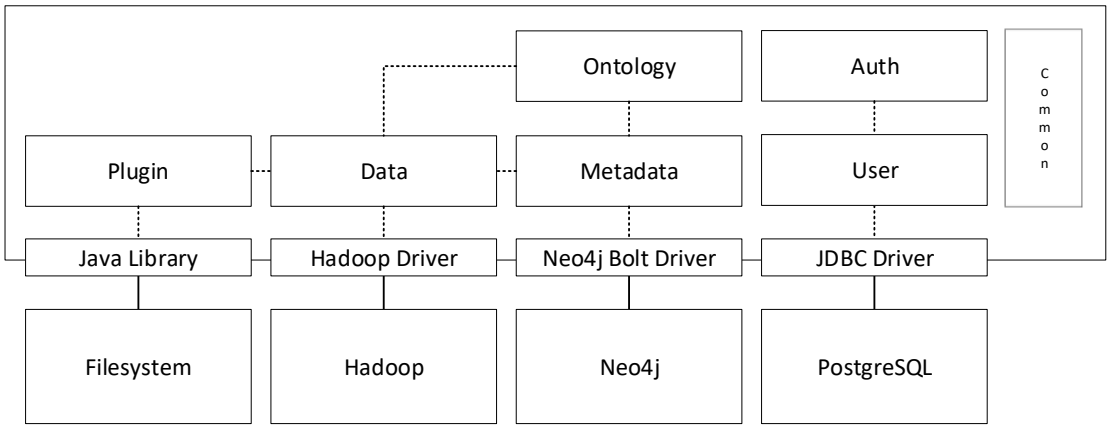


Figure 6.1: Overview of the project modules used to make up the prototypical implementation of the graph-based metadata management and their relation to the respective data storage systems.

Due to the strong web and data support as well as the huge range of high-quality development tools, Java was chosen as the main programming language for all components. Due to the simplicity of the prototype and the single developer, all components were developed as part of a single project. The project was created as a multi-module Gradle [1] project where each module can represent an individual application. To accelerate the development process, the Spring Framework [2] was chosen to be used in all the components. This means each component uses a layered architecture which is roughly split into controller, service, component, and repository classes. Methods in the controller classes become available via the web and expose the web interface by using Spring annotations. The controller classes use component and service classes to call business functions which in turn can use repository classes that provide an interface to access managed data. While the module can be deployed as a service itself making the functionalities publicly available, the service classes implement the actual business functionality and provides a Java interface, so they can be included into other modules. All exposed endpoints include a version number embedded in the URI to make sure an extension of the functionality can be used the same way as previous versions while still providing access to previous versions by simply using a different version number in the URI. To have a uniform URI scheme, all RESTful API endpoints that make up the graph-based metadata management use a prefix in the form of */v1/metadata*. The authentication endpoint is available on the URI */v1/tokens* and is also a RESTful web API. Figure 6.1 shows how the project is split into the different modules and how they are interconnected. The following sections describes the modules in detail and how the layered application architecture is implemented in each module.

Common Module

The common module is used as a shared module that provides a means to share common data models and provides interfaces to components that shall be available to all other modules. These include interfaces for a *Converter* class that needs to be implemented by each module to convert data models to DTOs (Data Transfer Objects). This is a commonly used practice when having a different internal data model than the data structure which is exposed on the API endpoints [Fow02]. Additionally, a *HATEOASHelper* component is defined which needs to be implemented by each module and can be used to add hypermedia links to the data models in order to be RESTful. While the specific implementations of these classes differ, their interface will be the same. Therefore, they are omitted from any class diagrams, so the focus is on the new classes of each module. Other classes in this module include a *RoutingUtil* and *OntologyNamespace* class which define global routing rules and the names for ontology expressions respectively. The ontology expressions represent identifiers of the graph database to identify relationships, classes, properties and hierarchies and can be adjusted in the external *ontology.properties* file. However, the files in the common module do not have dependencies between each other and neither is the module an application on its own nor does it provide a client interface in form of an external API.

[1] https://gradle.org/

[2] https://spring.io/projects/spring-framework

User Module

The user module bundles the functionality of a simple user management which is needed to store user credentials for the authentication mechanism. It provides the capability to simply create an administrator user in an external SQL database. It is implemented by using PostgreSQL [3] with a schema that consists of the three tables *user*, *role*, and *user_role*. The *user* table has the columns *id*, *username* and *password* while the *role* table has the columns *id* and *name*. The *user_role* table has the columns *user_id* and *role_id* which is used to create a one-to-many relation between the *user* and *role* entities. This is a very simplified database schema and only considers storing the most basic information to implement a role-based user authentication. However, the schema is open for extension as SQL entities can easily be extended with new columns, tables, and relations. For example, the *user* table could include additional columns for name, given name, birthday, and a relation to a record in a new *address* table. Hibernate [4] is used as an Object-Relational Mapping (ORM) mapper framework to model the SQL schema as *UserEntity* class as shown in Figure 6.2. The *UserRepository* class is responsible to connect to the external database and map the queries into the *UserEntity* class. A *UserService* interface defines the business functionality of the user module but they are not exposed in form of an API and limited to simply saving and querying a single user object via a direct method call. The business capability of the user module is defined by the *CreateAdminCommand* class which runs at startup of the module and simply creates an administrator user with the username *admin*. The password is either randomly generated or set as a command line argument and appears in the server logs during system startup. The *admin* user has the roles *ADMIN*, *ANALYST* and *USER*.

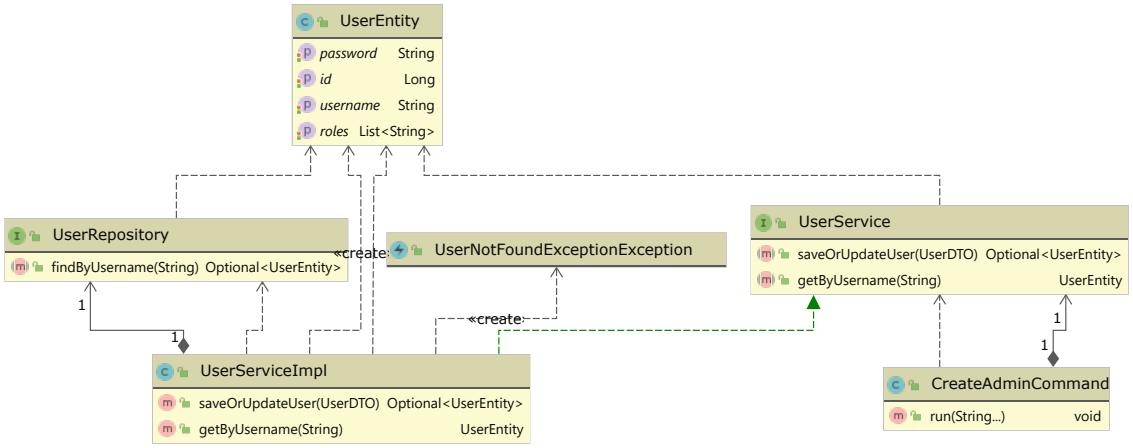


Figure 6.2: Class diagram that shows the connections between the logical components of the user module.

The user module in its current implementation is only used as proof-of-concept to implement a role-based authentication and therefore, does not provide an additional REST API to access CRUD functionalities of the user entities. To support a complete user management in the future,

[3] https://www.postgresql.org
[4] https://hibernate.org/

the *UserRepsitory* and the *UserService* must be extended with methods that provide CRUD functionalities on individual *UserEntities* and an additional REST controller class is required to make the functionality publicly available. The REST controller must also implement a filter to only allow authenticated clients with *admin* role to add new users. Instead of having to deal with the role-based access in each service module the functionality can be integrated in a dedicated authentication/authorization module that is already implemented and is introduced next.

Auth Module

The auth module is responsible for providing authentication and authorization tools to secure the API by using a token-based authentication mechanism. Therefore, we can identify tokens as managed REST resource. The module is used to provide a central place to apply all security policies to the endpoints of each service. The auth module includes a *TokenUtil* class which uses a *TokenConfig* class to provide the functionality to generate tokens and to check their validity as shown in Figure 6.3. The tokens are implemented as JSON Web Token (JWT) and generated by using a random secret string and a default validity of 30 days to ensure compromised tokens cannot be used indefinitely.

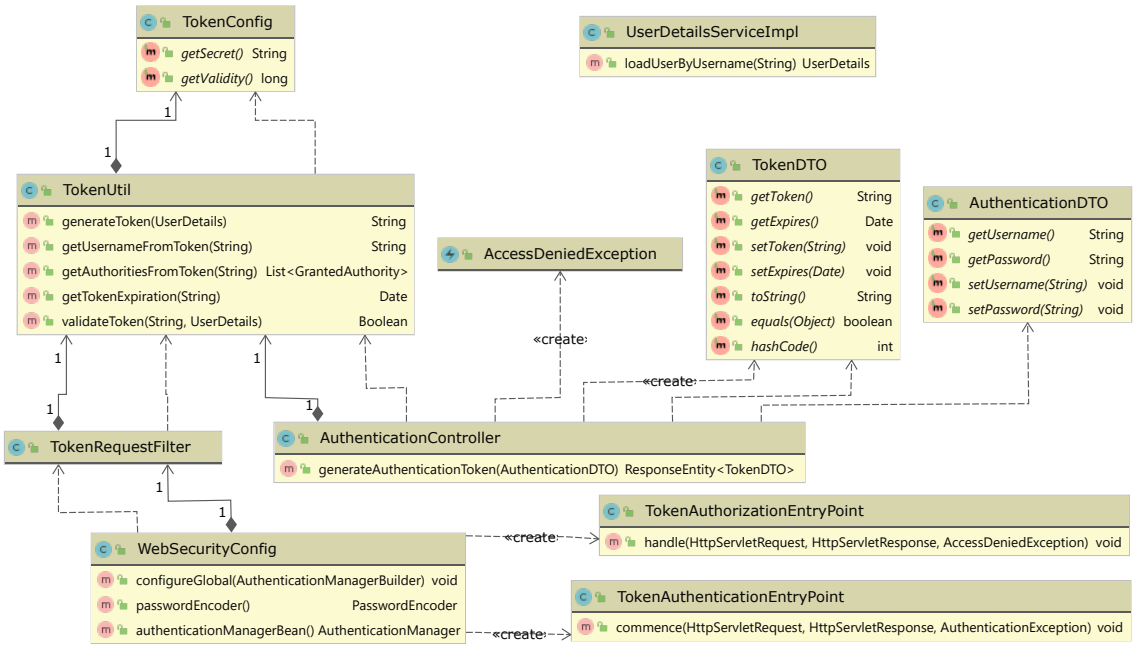


Figure 6.3: Class diagram that shows the connections between the logical components of the auth module.

To apply authentication and authorization mechanisms to any public API endpoint, a global security policy is defined in the *WebSecurityConfig* class. The security configuration is enforced by adding an authentication filter in form of the *TokenRequestFilter* class to the web security configuration. Endpoints are matched via regular expressions and an authentication mechanism is applied for specific HTTP methods that is allowed for certain roles. In general, all requests that modify resources

Matching Endpoints	HTTP Method	Authentication Type	Authorized Roles
/v1/tokens	Any	None	Any
/v1/metadata/plugins/**	Any	Bearer Token	Admin, Analyst
/v1/metadata/**	POST	Bearer Token	Admin, Analyst, User
/v1/metadata/**	PUT	Bearer Token	Admin, Analyst, User
/v1/metadata/**	DELETE	Bearer Token	Admin, Analyst, User
/v1/metadata/**	GET	None	Any
**	Any	Bearer Token	None

Table 6.1: An overview of the authentication mechanism for each endpoint and which user roles are permitted authorized to access them.

on the server require authentication. Additionally, the plugin endpoints require authentication with a user that has the *ADMIN* or *ANALYST* role. The higher privileges of the plugin endpoints is caused due to the higher security risk of the misuse of the functionality since it allows to run executable files on a remote server as well as retrieving OS sensitive information. A complete overview of the security configuration for each endpoint is shown in Table 6.1.

The *TokenAutheniticationEntryPoint* and *TokenAuthorizationEntryPoint* classes define the actions to be taken in case of unauthenticated and unauthorized requests respectively. In this case the requests are rejected, and an appropriate error message is sent as response. The *AuthenticationController* class is used to provide a RESTful API endpoint to allow clients to generate API tokens by sending their username and password in a request body. The full URI as well as the HTTP method of the authentication endpoint is shown in Table A.1. The tokens are validated against the credentials stored and managed by the user module. This is done via the *WebSecurityConfiguration* which provides an *authenticationManagerBean* class that uses the *UserDetailsServiceImpl* class to check whether the given username and password match a record in the user database. Once validated, a response is generated that includes the API as well its expiration date. Therefore, the auth module provides a central way to manage authentication mechanisms and user permissions for all endpoints of the graph-based metadata management API and exposes the capability to generate API tokens in form of a public web API.

Data Module

The data module is used to provide data access functionalities such as saving and retrieving files on local and remote filesystems. This functionality can be used to read plugin files from the local filesystem, to store any uploaded files locally as well as storing and retrieving CAE data files in a remote fileysystem. The remote filesystem is implemented using the Hadoop Distributed File System (HDFS) [5] and makes up the concept of the data lake in the graph-based metadata management concept. Since the concept states that metadata and data shall be managed together, the module does not offer a public web API so clients have no direct access to the data access capabilities. Instead, the module is used by the metadata, ontology, and plugin module to separate the data

[5] https://hadoop.apache.org/

access layer from their other functionalities. The idea to separate all data access functionality into a separate module also allows to extend its functionality independently in the future and to deploy it individually.

Figure 6.4 shows, the data module uses a single interface to define common filesystem functionalities which is extended by more specific filesystem interfaces that make use of the local and remote filesystem properties respectively. The actual implementation of the *HDFSService* interface uses the class *HDFSConfiguration* to configure the remote filesystem properties and custom directories. The values are read from an external *hadoop.properties* file which defines the root directory and file storage directory to be used when interacting with the HDFS. The communication is done by via the HDFS communication protocol which specifically supports the streaming of large datasets as stated on their website.

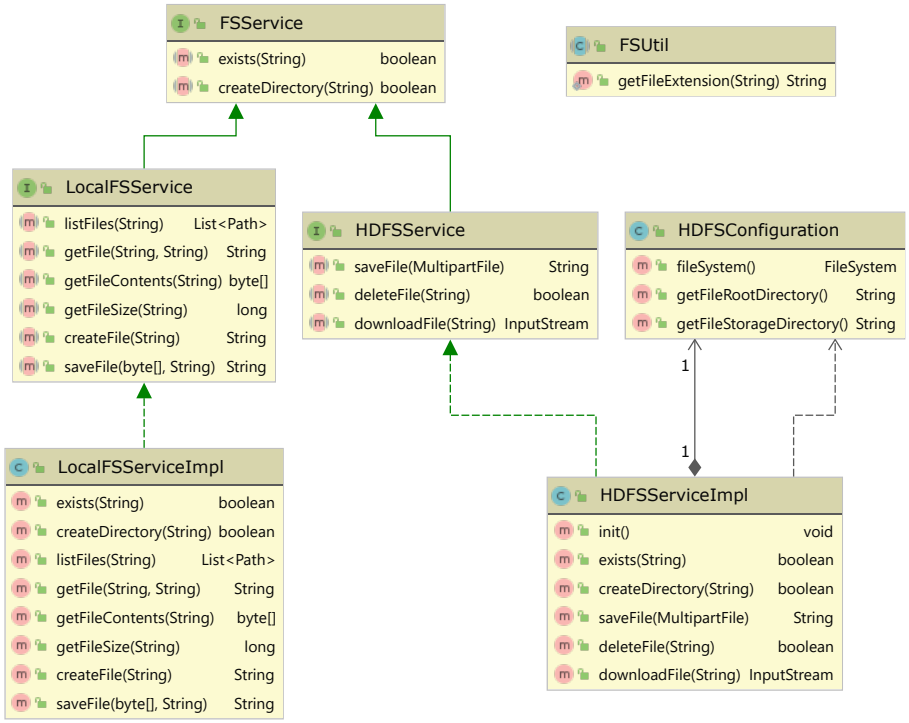


Figure 6.4: Class diagram that shows the connections between the logical components of the data module.

Metadata Module

The metadata module provides the functionalities to access and modify the graph resources. Three managed resources are implemented, namely *nodes*, *relationships* and *queries*. The *node* and *relationship* resources allow CRUD operations and directly refer to graph nodes and relationships of the graph database, respectively. The *query* resource is exposing the functionality of the graph database by giving access to its native query functionality. The resources were chosen to offer

Listing 6.1 A metadata node model as JSON representation.

```
{
    "id" : 1,
    "uuid" : null,
    "labels" : [ "Data" ],
    "properties" : {
        "fileName" : "TestPlan.xls"
    }
}
```

the clients the comfort of simple metadata CRUD operations without having knowledge of the underlying graph structure as well as allowing more advanced queries leveraging the full power of the graph database.

The methods for the functionality are defined in the *NodeService*, *RelationshipService* and *QueryService* interfaces respectively which are shown in Figure 6.5. Their implementation is done in separate classes which are injected via dependency injection by the framework but they are omitted in the class diagram to improve its readability. The services will be used by the *NodeController*, *RelationshipController* and *QueryController* which expose a RESTful API to provide web access to the functionality of the services. The CRUD functionality of the *NodeService* and *RelationshipService* are mapped to the HTTP POST, GET, PUT and DELETE methods via the controller classes. The *QueryController* only supports HTTP POST requests and allows the client to pass a Cypher [6] query as request model. A complete overview of the prototype's metadata endpoints is shown in Table A.2. The communication flow follows the layered structure where the controller classes only communicate with service classes which provide the functionality. The services are set up to communicate with an external Neo4j [7] graph database. The *GraphSession* class holds a session object which is used by the service implementations to run queries against the graph database.

Due to the implementation of Neo4j as graph database, IDs are provided which can be used to reference graph nodes and relationships in queries. However, as stated on their website [8], it is considered bad practice as the given IDs can be reassigned and it is advised to use a self-managed Universally Unique Identifier (UUID). Therefore, a *GraphUUID* class was created which is inherited by the *NodeDTO* and *RelationshipDTO* classes. A representation of a graph node consists of a *UUID*, *ID*, *labels* and *properties* while the representation of a graph relationship consists of a *UUID*, *ID*, type *startNodeId* and *EndNodeId*. These define the communication model for the services and controllers. *UUID* and *ID* are automatically assigned when a new node is created.

A client can supply a *NodeDTO* model to create a new metadata graph node which can have multiple types in form of labels and key-value pairs as properties. A JSON representation of a model to create a sample metadata node for a CAE test file is shown in Listing 6.1. However, this does not make sure a CAE data file is present in the data lake whenever a metadata node is created which references this file. To ensure this integrity, a file can be uploaded alongside the model representation of the metadata node. The *NodeController* makes use of the already presented data

[6] https://neo4j.com/developer/cypher/
[7] https://neo4j.com/
[8] https://neo4j.com/blog/dark-side-neo4j-worst-practices/

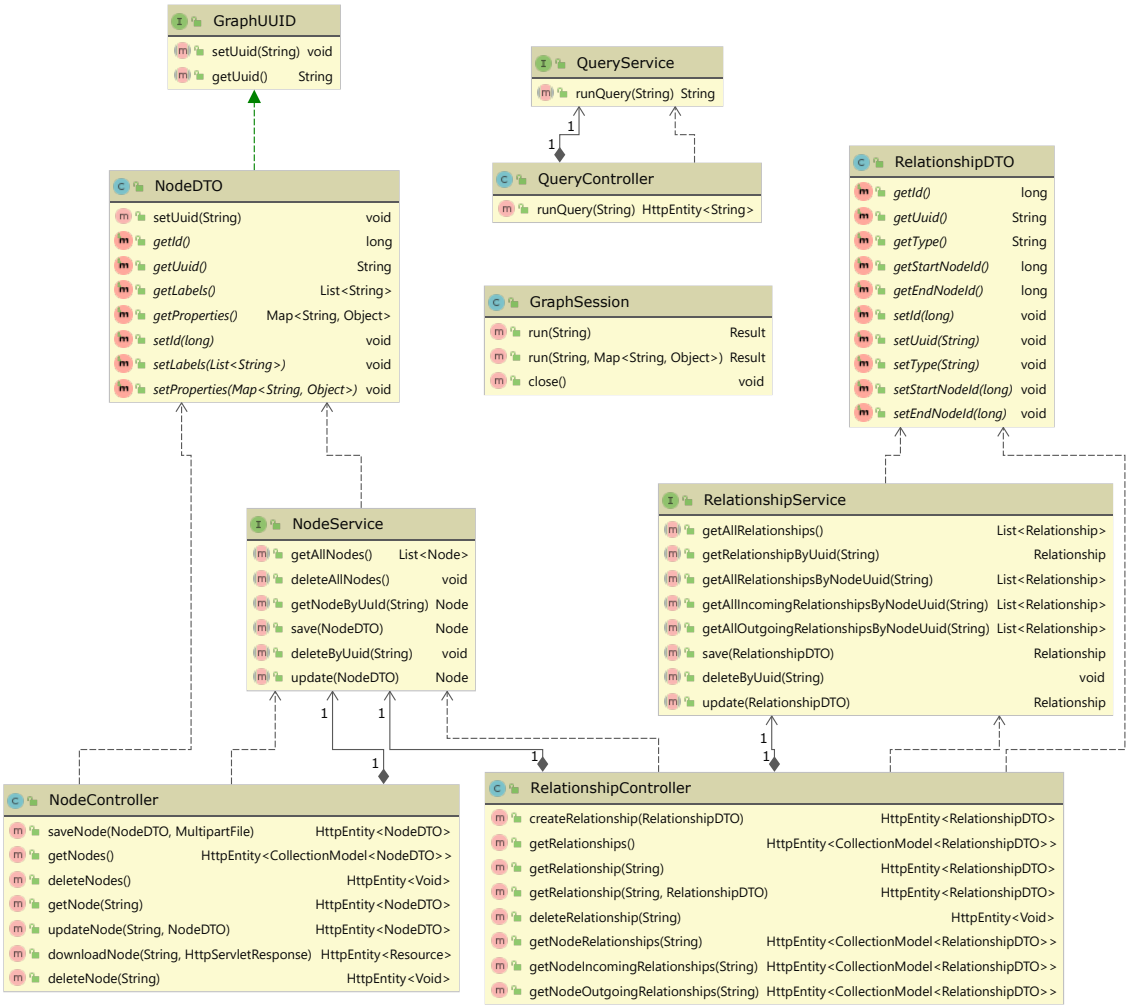


Figure 6.5: Class diagram that shows the connections between the logical components of the metadata module.

module and uses streaming to prevent the file from being cached on its own deployment server. Once a positive response is received from the data module, it stores the metadata node therefore, ensuring the data integrity.

To create a relationship between two metadata nodes, the *RelationshipDTO* model can be supplied which is defined as a typed relationship with a single type and a referenced start as well as a referenced end node in form of IDs. A JSON representation as shown in Listing 6.2 can be used to create an input relationship where the metadata node with *ID=1* is an input for the metadata node with *ID=2*.

Listing 6.2 A metadata relationship model as JSON representation.

```
1     {
2         "id" : 4,
3         "uuid" : null,
4         "startNodeId" : 1,
5         "endNodeId" : 2,
6         "type" : "Input"
7     }
```

Ontology Module

The ontology module provides the functionality to semantically enrich the graph metadata by using an ontology. The ontology can be used to define classes, properties, and relationships of graph nodes. Each of them is defined by a graph node itself which has a name property that defines the name of the class, property, or relationship. To model a class hierarchy, a *subclass-of* relationship is supported between two class nodes. By using an ontology, metadata graph nodes as presented in the metadata module become instances of the defined class nodes and include their class name as label. The instances of property nodes are key-value properties on the metadata nodes and cannot be instantiated as standalone graph element. The relationship nodes are instantiated by graph relationships and require a start and end node to be instantiated.

With this in mind, the ontology module manages an *ontology*, *ontology node*, *ontology class*, *ontology property* and *ontology relationship* resources. The *ontology* resource is representing the whole ontology in form of a file which uses the Turtle [9] syntax of the RDF [10] standard model. The file can be uploaded by using a HTTP POST request, parsed and recreated in the graph-database by using nodes and relationships. This allows to manage the ontology and instances in the same place and leverages native graph database technologies to perform fast queries and indexing. To ensure the integrity of the ontology and available metadata nodes, an ontology can only be created when the graph database is empty. An ontology can also be viewed and deleted as a whole supporting HTTP GET and DELETE requests, respectively. As shown in Figure 6.6, the methods of the API endpoints are exposed by the *OntoController* class which uses the *OntoService* interface definition to delegate the actual work. The implementation of the service class was done in a separate class and can throw a *GraphNotEmptyException* or a *WrongFileFormatException*. The exceptions are handled by the *OntologyControllerAdvice* class which is a RESTful controller itself. The implementation class of the service is used by the controller via dependency injection and is omitted in the class diagram for readability reasons. Since Neo4j recommends the use of UUIDs for referencing graph nodes, the *OntologyUtil* class was created to provide utility methods to add an UUID to each node that is created from the ontology.

The *ontology node*, *ontology class*, *ontology property* and *ontology relationship* resources are used to expose additional functionality to manage metadata. By having classes defined as a hierarchy and endpoints to reference a specific class, a new endpoint was created that leverages ontology by providing the capability to show all instances of a specific class and its subclasses. Each of the resources expose a RESTful API to show the classes, properties, relationships, and all nodes

[9] https://www.w3.org/TR/turtle/
[10] https://www.w3.org/RDF/

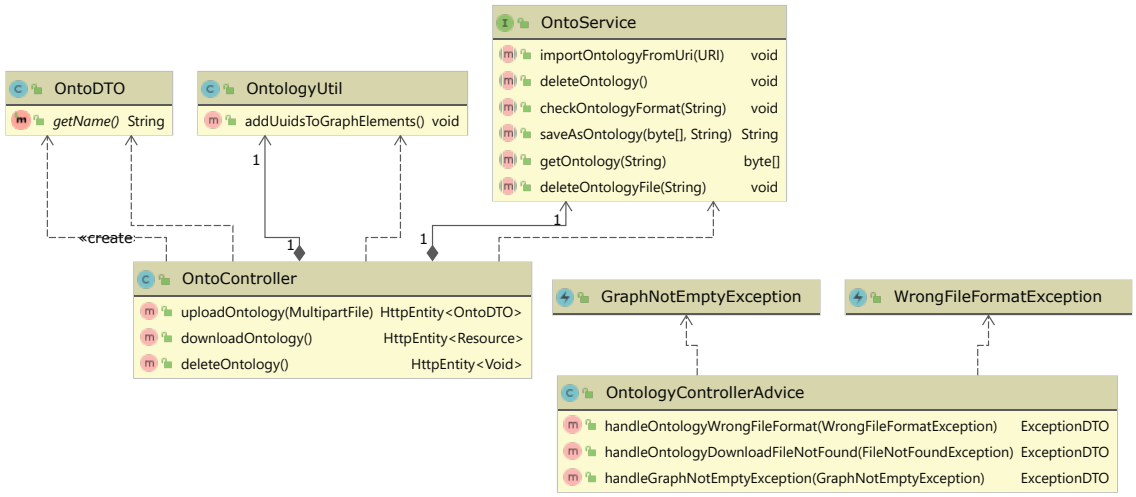


Figure 6.6: Class diagram that shows the connections between the logical components of the metadata module which are responsible for managing an ontology for the graph-based metadata.

defined by an ontology. These can be linked by using hypermedia links which allows to query each subset of the ontology individually. To prevent the modification of the created ontology these endpoints are read only by allowing HTTP GET requests only. A new endpoint in the form */v1/metadata/onto/classes/uuid/instances* provides the mentioned capability to show or delete all instances of a specific class as well as creating a new metadata node as instance of the class referenced by its UUID. A complete overview of the prototype's ontology endpoints is shown in Table A.4. As shown in Figure 6.7, this was implemented by creating a new controller class for each resource which implements the methods that make up the REST API. Each of these controllers uses a service class dedicated to its resource. A Data Transfer Object was created for class, property and relationship nodes which is a subclass of the *NodeDTO* that was shown in Listing 6.1.

Plugin Module

The plugin module bundles the functionality of the plugin manager and plugin type specific process executors. This means it needs the capabilities to show the available plugins to clients, allow them to start a plugin, stop a running plugin and monitor information about the current plugin executions. By applying the REST architectural style, we can identify that the service needs to manage the two resources *plugins* and *processes* where the latter belongs to the former.

This means a *Plugin* as well as a *PluginProcess* class were created to handle any managed plugin information. A plugin object has a reference to the actual executable plugin file in form of an absolute filesystem path. As shown in Figure 6.8 the *Plugin* objects holds an implementation of a *PluginType* interface that defines how a specific plugin type can be executed. By using a registry to register any classes that implement the interface, a service can look up the registry of supported plugin types and instantiate the appropriate plugin with a specific type. This way additional plugin types can be easily supported by simply implementing the *PluginType* interface and registering the class in the *PluginClassRegistry*. The *PluginService* defines the methods needed for the controller

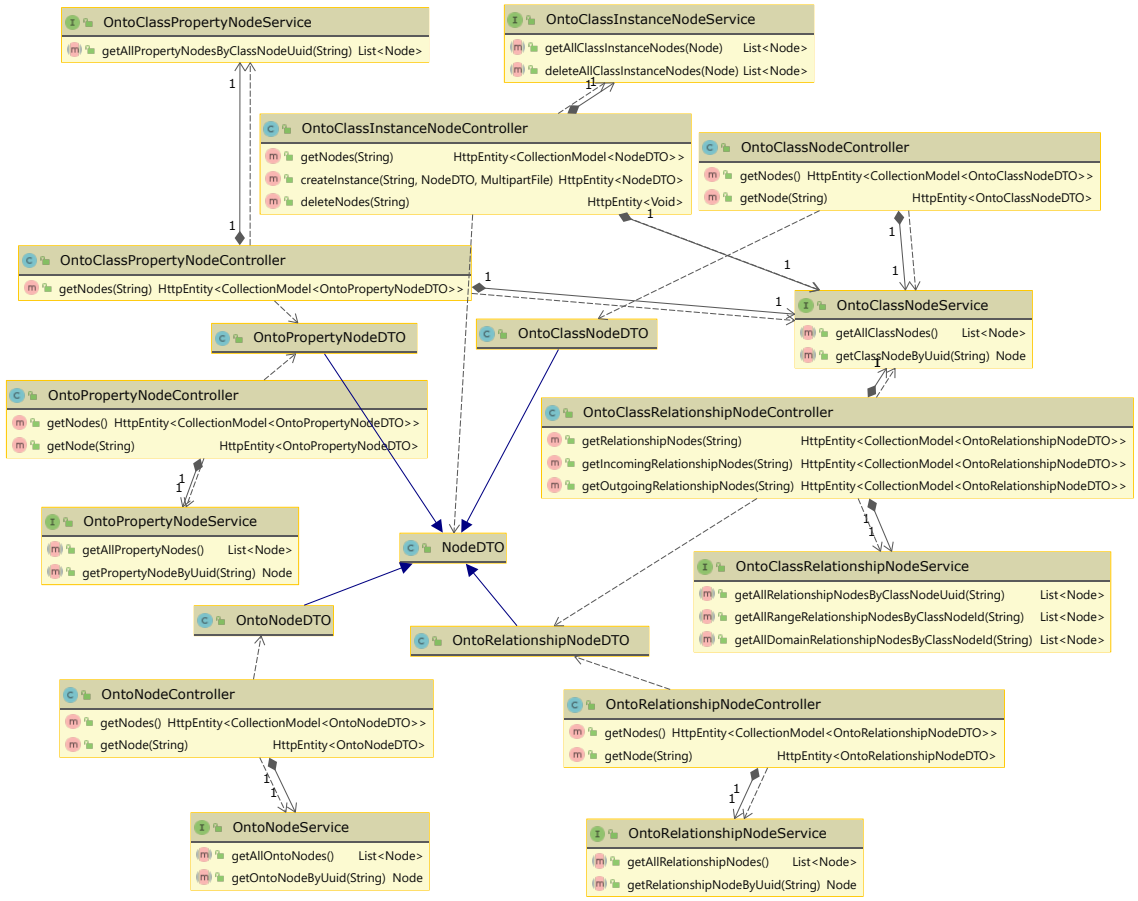


Figure 6.7: Class diagram that shows the connections between the logical components of the ontology module which are responsible for the metadata management by using an ontology.

classes. In this prototype implementation the executable plugins are stored in a predefined plugin directory on the local filesystem where the plugin module are deployed. However, to fully support the scaling potential described in the concept architecture this needs to be replaced with API calls to an external filesystem. However, this change would make the prototype unnecessarily complex while the overall dependencies to any other classes would remain unchanged.

The *PluginProcessService* gets passed the *Plugin* objects which include information about how they have to be executed. This way any plugin can be executed by the same process executor service instead of implementing a specific service for each plugin type. In this implementation the *PluginProcessService* is implemented by using OS tools from the underlying Java platform to start, stop and monitor the processes of a plugin. Plugins are started by the Java *ProcessBuilder* class and information on running processes are retrieved by using the Java *ProcessHandle* class. It allows to pass information to a new process via the standard input and by using environment variables. This allows to distinguish between information, necessary for the plugin to operate correctly and additional user supplied input. The former, which include connection details about the data lake and graph database where the data and metadata are stored respectively, is shared to the plugins

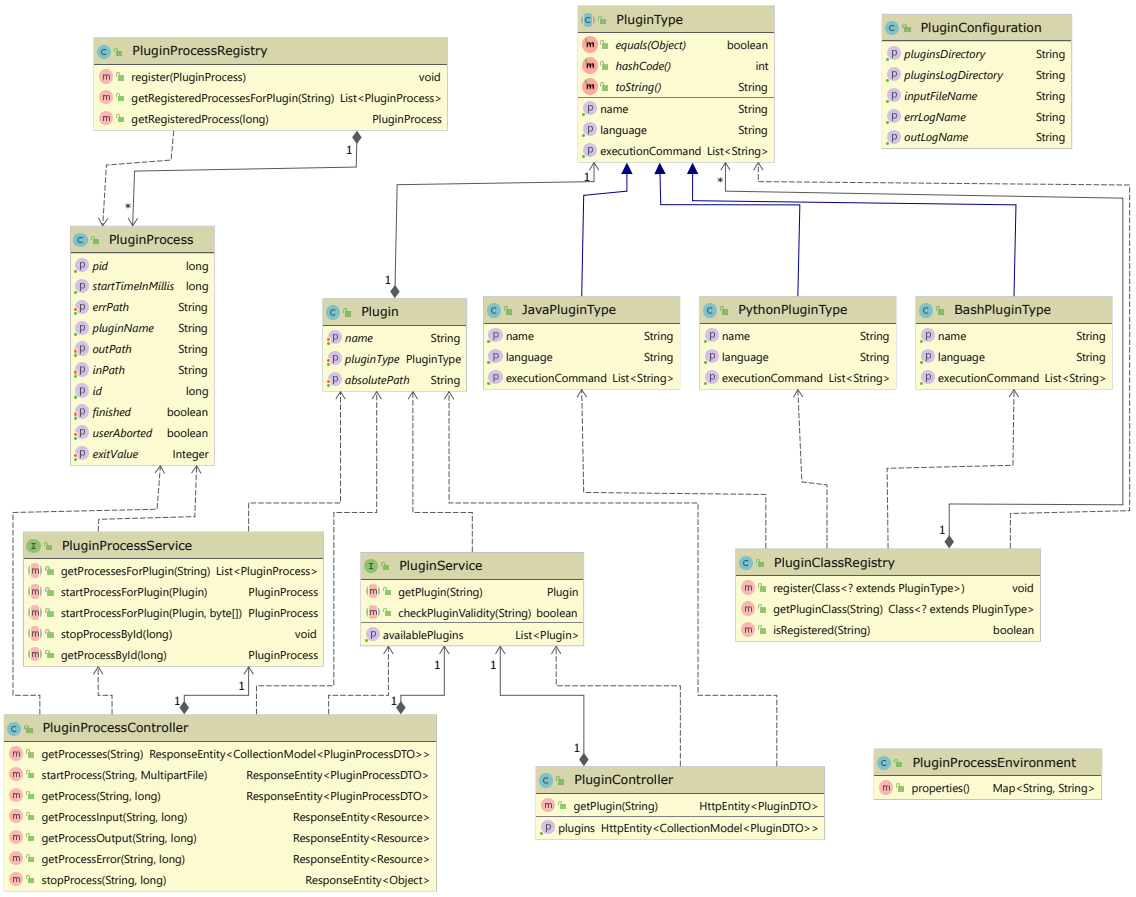


Figure 6.8: Class diagram that shows the connection between the logical components of the plugin module.

in form of environment variables. The plugins can use OS tools of their supported language to retrieve environment variables to gain information about how to connect to the data lake and the graph-database. This is important because the plugins need to use the same data lake, used to store CAE data, to transform these data files and store the final result in the same data lake. This way the graph-based metadata management system can manage the plugins without any further configuration. The latter information that plugins might need is user input to know on which data files a transformation step has to be executed and which parameters should be used. This is achieved by passing information about the data items to operate on to the plugin via the standard input. Since data items in the data lake are linked to metadata nodes in the graph database via a filepath property, UUIDs of metadata nodes can be passed as input to tell a plugin which CAE files shall be transformed.

The output and error logging of the plugins is written to the filesystem in form of simple text files in a directory that is named by the UUID of the process that it belongs to. Plugin processes are stored in an in-memory registry in form of the *PluginProcessRegistry* class that stores plugin processes using a Process Identifier (PID). To prevent leaking any OS sensitive information such as used PIDs, a randomized UUID is used instead of the PID of the underlying OS. However, to support full horizontal scalability, the *PluginProcessService* must use API calls to external executor services to

start the processes on a remote server. The plugin processes and their output as well as error logs would also need to be stored in an external database to be present even after system failure or when decommissioning a single executor service. Like the plugin storage location decision, this is done to reduce the complexity of the prototype implementation because the overall concept of the classes and their dependencies would not change.

To allow public clients to use the plugin and plugin process capabilities, a web interface in form of a RESTful API was created. This is done by creating a controller class for each resource and using annotations of the Spring Framework on their methods to make each method available as a web service endpoint. The controllers use the defined model classes as REST resources and delegate the plugin application logic to the services. In order to support the HATEOAS property of REST, specific HATEOAS helper components were implemented that are responsible to add hypermedia links of related resources to the model representation of the *Plugin* and *PluginProcess* classes. In case of the former, links are added to its own URI and the URI of its processes. In the latter case, links are added that point to the URI of itself, as well as to the input, output, and error files. The controller classes also handle the transformation of the model representations depending on the mime-type that is defined by the HTTP Content-Type and Accept headers for requests and responses respectively. The start plugin process functionality is mapped to a HTTP POST method and an additional file can be included in the request that defines the input passed to the plugin when creating a new process. Each new request starts a new process of a plugin which means simultaneous execution of the same plugin is supported. A HTTP DELETE method is used to stop a specific plugin process which shall not have any effect if the specified plugin process is already stopped. A HTTP GET request is used to query information about available plugins, plugin processes, plugin process input, output, and error logs. A complete overview of the prototype's plugin endpoints is shown in Table A.4.

6.2 Hypermedia-Driven Prototype

The prototype adheres to the REST constraints and, therefore, it uses HATEOAS. It provides hypermedia links in responses which indicate the allowed state transitions that can be initiated by clients when receiving the resource. This allows clients to explore the API without the need for external documentation. The prototype uses the Hypertext Application Language (HAL) [11] format to represent links which means a *_links* object is added to each resource model which includes a typed relation object which in turn holds the referenced URI. This allows even human users to navigate the API by using a normal web browser and simply following hypermedia links.

Any resources which requires authentication links to the authentication endpoint if no valid authentication method is supplied. The authentication endpoint that manages the *token* resource links to the metadata, ontology, and plugins endpoints where the metadata endpoints are split into the *node*, *relationship* and *query* resources. Graph nodes and relationships are tightly connected and link to each other via multiple relations. The *node* resource also links to the *data* resource. The *ontology class*, *ontology property* and *ontology relationship* resources are also interconnected and linked to by the *ontology* resource. The *ontology class* resource in turn links to the metadata

[11] https://tools.ietf.org/html/draft-kelly-json-hal-00

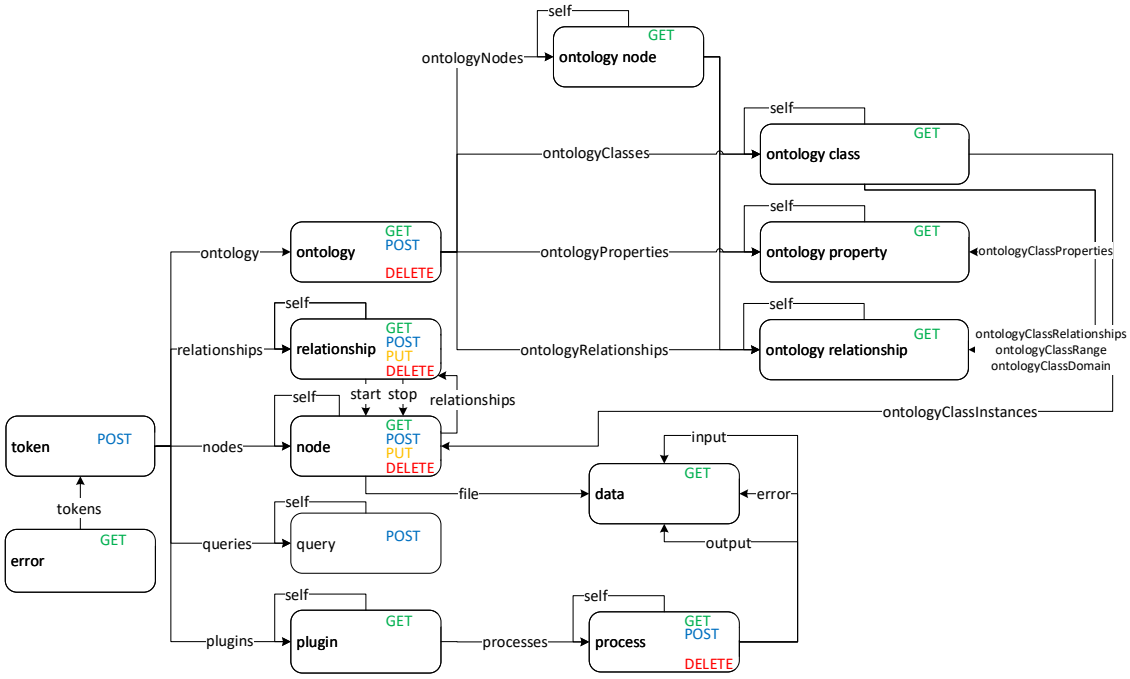


Figure 6.9: State diagram of the implemented graph-based metadata management in form of named relations for each resource.

node resource. The *plugin* resource links to the *process* resource which links to the *data* resource. Figure 6.9 shows a complete state diagram where endpoints are grouped to their respective resources and hypermedia links are represented by named relations.

6.3 Build Automation

The system is composed of multiple independently developed modules. This makes it easy to deploy the components on different physical servers to have a distributed system of multiple small components. Developing independent modules and decoupling them by using a loosely coupled interface in the form of REST allows us to scale each module independently which means the architecture fits the needs of small user bases just as much as it fits the needs of large user bases. However, since the prototype implementation does not need the scaling capabilities, all components are deployed on the same physical server to reduce the systems complexity.

This was done by using Gradle and Spring Boot [12] to create a single executable Java file which starts all the modules at the same time on the same server. If the scaling capabilities are indeed needed, an executable file needs to be created for each module in order to start them individually.

[12] https://spring.io/projects/spring-boot

In either case it is useful to automate the build and start up process needed to start the application which is a key component to quickly start up the whole system in a certain environment. This is useful for local testing, integration tests and to start up an entire production environment. To decouple the application from the underlying platform, the application is wrapped in a container. This is implemented by using Docker [13] as container engine which allows the application to run on the same platform while the container itself is portable across many different OSs. Plugins that should be made available via the API can easily be included in this container by adding a copy statement into the container build script. In case the scaling is required and individual executable Java files are created, each individual file must be wrapped in its own container. The dependencies and each individual service can then be configured by using a container orchestration tool such as Kubernetes [14] that uses IasC to deploy all of the individual applications as a single interconnected system of containers with build in scaling capabilities of individual containers.

6.4 Testing

The graph-based metadata management system was developed with separation of concerns in mind which is shown by the individual modules. This means each utility class can be tested using unit tests while the remaining classes can be tested with integration tests. This is done by starting a test web server that only includes the files of a single module and using a test web client to do black box testing of the web API. Any required test data or additional services are mocked to keep the test scope of the integration tests small. This creates flexibility in creating additional system tests where all the modules are started, and the mocks are replaced by the actual databases that hold the test data. This way each class can be tested in the first phase to ensure the methods work as intended. The integration tests verify the functionality of the individual components by testing the API endpoints. Additional system tests use a near production setup of the whole system and validate the system against different use cases.

While each of the tests can be automated, it is quite cumbersome to automate system tests since the use cases might be quite complex and require additional user guidance. Unit and integration tests are implemented as automated tests for the prototype to verify the implementation and to meet a certain level of quality. System tests have also been made but were not automated. To support the manual execution of the API calls, Postman [15] was used to manage all available URIs in a central knowledge repository. Chapter 7 describes in detail how these system tests look like and how they were conducted.

[13] https://www.docker.com/
[14] https://www.digitalocean.com/products/kubernetes/
[15] https://www.postman.com/

6.5 Documentation

The prototype is documented in several ways to help software architects, developers and clients to understand how it works and how to use it. Software architects make lots of Architectural Decisions (ADs) which impact the structure of the overall system. By capturing the decision rationale, future architects and developers gain a better understanding of why the system is structured as it is which helps when a system needs to evolve [JVAH07]. Software ADRs are documented as Architectural Decision Records (ADRs) in the Markdown Architectural Decision Records (MADR) [KAZ18] format. This way the prototype can quickly evolve and eventually become an enterprise ready system if needed.

To further support the developers each service class which provides the business functionality is thoroughly documented to make sure future developers can reuse its functionality. Additionally, the Java coding guidelines [16] were followed and it was focused on clean code development with self-descriptive class, variable and method names.

Clients do not have any insights into the source code and need additional external documentation as guidance on how to use the RESTful API of the graph-based metadata management system. One of the strengths of REST is its uniform interface that uses HATEOAS. This allows clients by just knowing the address of the server to explore the service on its own. Any requested resource links to related resources which can be used to initiate state transitions. For example, in the case of the graph-based metadata system, the node resource can be requested which gives a list of all graph nodes where each graph node includes links to itself, the relationship resource and the file resource. This makes the API of the system extensible without the need to explicitly document the new resources as the client simply retrieves links to them.

However, a human user might still be interested in a more readable way in form of an external API documentation. Therefore, a complete API documentation is deployed alongside the graph-based metadata management system which describes the available URIs, request parameters as well as gives sample request and responses. To keep the documentation up to date, the API endpoints as well as sample requests and responses are generated by the test client and server of the integration tests. The API documentation is deployed on the same server as the graph-based metadata management and can be found by using */docs/index.html* as URI.

6.6 Continuous Integration

The idea of developing separate decoupled components and start them as individual services is great in terms of maintainability, extensibility, scalability, and availability. However, this means an overhead in the build, testing, deployment, and management phases as each step needs to be done for each component individually. In a fast-paced development environment where new iterations of a product need to be developed continuously this can become a drawback of the selected architecture. However, the concept of CI helps to tackle these problems by using a CI server to automatically call these routines.

[16] https://www.oracle.com/technetwork/java/codeconventions-150003.pdf

The concept of CI uses a single source code repository where all developers commit their code to. The build process of the individual projects and the overall system should be automated which can be done by using a dedicated build tool. Any tests should use a near production environment but shall not have any external dependencies, e.g. an in-memory test database should be started on demand rather than using a fixed test database set up for every test. This improves the overall reliability of the tests and makes them comparable across different test runs. A CI server is used to automatically pick up commits in the source code repository which always triggers a CI build. [FF06]

Each CI build uses the build automation to build the whole project with each individual module. Once the build step is completed each module is tested individually while system tests run afterwards. A CI can include additional analysis or set up steps like static code analysis checks to ensure a certain quality level. Any artifacts that were built should be archived so they can easily be accessed by others when needed [FF06]. Due to the automation and the ability to quickly set up the system in a specific environment, CI allows to continuously deliver a production ready system. In fact, CI/CD builds can go even further and automate the deployment of the production ready system which is called continuous deployment [AP18]. This way, the overhead of having individual modules is minimized.

The prototype includes a CI/CD build script in the form of a Jenkinsfile which is used in conjunction with a Jenkins [17] server. The source code of the implementation is versioned in a GitLab [18] repository [19]. The CI/CD server pulls the repository for changes and starts a new build once it has found new commits. This means each project is build and tested by using Gradle as the build tool. Test results are published to be available to the user in form of a HTML report. The static code analysis tool SonarQube [20] is used to ensure a certain level of code quality and is called in the next step of the build pipeline. The results are once again validated and checked against a defined quality gate. The quality gate passes if new code meets the following requirements of 1. at least 80% test coverage 2. less than 3% duplicated lines 3. not introducing new bugs 4. not introducing new vulnerability issues 5. not introducing new code smells. The latter refers to any code snippets that are known to be bad practice. If the code passes the quality gate, an executable Java file is created by using Gradle to automatically bundle the individual modules as a single Spring Boot application which is getting archived in order to still be available after the build. The build script continues by creating a Docker image based on the executable Java file. This image is automatically pushed to an external artifact repository implemented by a Sonatype Nexus Repository [21]. This way a production ready Docker image is available on a Docker registry which can be easily pulled by anyone interested in starting a production system of the graph-based metadata management API. The prototype also supports continuous deployment by including a deployment step in the build script where the Docker image is pulled and started on the production server. This ensures that the deployed prototype meets a certain level of quality and does not introduce new bugs when redeployed.

[17] https://www.jenkins.io/
[18] https://about.gitlab.com/
[19] https://gitlab-as.informatik.uni-stuttgart.de/ma_architektur/metadata-api
[20] https://www.sonarqube.org/
[21] https://www.sonatype.com/nexus/repository-oss

7 Validation

Chapter 5 introduced a concept of a software architecture for the graph-based metadata management which has been implemented as a prototype as described in Chapter 6. In order to validate that the architecture fulfills the requirements defined in Chapter 4, the prototype is validated against its use cases. The next sections describe how the validation process works, how it was executed and what results were measured. Section 7.1 shows the test set up and the tools used for the validation. Section 7.2 describes the preparation for the validation and introduces the validation plan for each use case of Section 4.1. Section 7.3 describes the execution of the validation plans and presents the measured results.

7.1 Environment

The architecture of the graph-based metadata management is being validated by running the use cases defined in Section 4.1 against the implementation of the prototype. The prototype is deployed on a Virtual Machine (VM) of the university servers. The network connection between the VM and test computer where the files will be uploaded from is limited to 35 Mbps for upload and 100 Mbps for download. The data lake in form of Hadoop is deployed as a cluster configuration on two separate VMs on the same university server. Neo4j as well as PostgreSQL are also deployed on their own VMs on the university server. In the first step, a validation plan was created that lists all steps that are necessary to fulfill each use case in form of user actions. In the next step, the validation plan is executed by mapping the user actions to HTTP requests that are executed against the API of the prototype. The Unix command line tool cURL [1] is used as API client because it allows to easily change the HTTP request methods. However, each request can also be executed by using a graphical web browser and changing the HTTP request methods as needed in the developer tools. Each command to create the HTTP request as well as the server's HTTP responses is documented. Additionally, the total round-trip time is measured. This is done by prefixing the *cURL* command with the built-in Unix command *time*.

7.2 Preparation

To be consistent between the use cases, the same data files and metadata information has been used. CAE test files were simulated by generating binary files. A 10 GB binary file was generated to simulate a CAE 3D model file and a 10 MB CSV file was generated to simulate a test result file. Additionally, a small Java application was created to be used as test plugin for executing the use

[1] https://curl.haxx.se/

cases. More details on the Java test plugin and the full source code is shown in Appendix A. The data lake as well as the graph database were cleared before executing the first use case. However, the use cases *Visualizing Graph-based Metadata* and *Transforming CAE Data* were executed on basis of the data of the use case *Linking CAE Data with Graph-based Metadata* to prevent duplicating the steps. A validation plan was created for each use case which are described in the following three subsections.

Validation Plan: Linking CAE Data with Graph-based Metadata

Step 1 Login to the graph-based metadata management system

Step 2 Create an ontology

Step 3 Store the CAE test file in the graph based-metadata management system

Step 4 Add the creation date to the CAE model

Step 5 Add the number of voxels to the CAE model

Step 6 Add the material to the CAE model

Step 7 Store the simulation result CSV file in the graph based-metadata management system

Step 8 Add the simulation process to the graph based-metadata management system

Step 9 Create an input relationship from the 3D model to the simulation process

Step 10 Create an output relationship from the simulation process to the simulation result

Step 11 Query all 3D models and test files

Validation Plan: Visualizing Graph-based Metadata

Step 1 Retrieve all metadata nodes and relationships from the graph-based metadata management system

Step 2 Select a metadata graph node of a 3D model and show its connected metadata

Step 3 Download the linked file of the selected metadata node

Step 4 Allow iterating over the graph by showing all relationships and their end nodes of the selected graph node and repeat from step 3

Validation Plan: Transforming CAE Data

Step 1 Login to the graph-based metadata management system

Step 2 List all available plugins

Step 3 Start the sample plugin with a list of node IDs

Step 4 Check whether the plugin has finished

Step 5 Show the output of the plugin process

Step 6 Check the error log of the plugin process

7.3 Execution and Measurement

The validation plan was executed by mapping each step to one or more request URLs. The requests and their corresponding server responses were documented as well as their round-trip time was measured. The requests and replies are documented in JSON format although XML is also supported. They are also shortened to focus on the relevant parts of each request and reply. The complete execution plans for the use cases *Linking CAE Data with Graph-based Metadata*, *Visualizing Graph-based Metadata* and *Transforming CAE Data* is documented in Table A.5, Table A.6 and Table A.7, respectively. The following three sections describe each execution plan in more detail.

Execution Plan: Linking CAE Data with Graph-based Metadata

The complete execution can be found in Table A.5

Step 1 The login mechanism of the graph-based metadata management system is mapped to a URL pointing to a tokens resource. It requires to send a valid username and password via a HTTP POST request and generates an API access token with a validity of 30 days.

Step 2 An ontology can be easily created by uploading an ontology file to the URL of the ontology resource. The prototype only supports ontology files in the Turtle syntax which can be used to define classes, relationships and properties as shown in Listing A.2. The ontology will then be modeled in the graph database and the response includes links to guide the client to its created resources.

Step 3 A CAE data file is managed by sending a metadata model and attaching the file to the request. This way a metadata node exists for each CAE file in the data lake. The metdata and its file are linked by the server which automatically adds a *originalFileName* and *filePath* property. Additional properties, which should not be considered to be metadata, can be added by specifying them in the request model such as the name property. An ID and UUID property is added by default to any node and relationship during their creation. It also provides links to further explore the relationships of the node and a direct download link for its linked CAE file.

Step 4-6 To create additional metadata nodes, the same endpoint can simply be reused. By omitting a file attachment, the file properties are not set by the server and only a graph node of the given type is created. To connect nodes a request is sent to the relationship resource where the ID of the start and end node is specified as well as the type of the relationship.

Step 7-8 The procedure of creating a node linked to a CAE file is the same as already described for the model file and has been applied in Step 7 again. The creation of any node that does not require additional CAE files is the same as the already mentioned creation of metadata nodes. For example, in Step 8 of the execution plan the process node has been created by simply specifying a different label in the request model and following sending it to the same URL of the already mentioned metadata nodes resource.

Step 9-10 Connecting the process nodes also follows the same pattern as connecting metadata nodes and specifying a different type for the relationship.

Step 11 Using the same labels, relationship types and property names as in the ontology that was uploaded, all data nodes can be queried by using the ontology class resource. The first request lists all ontology classes which is used to find the UUID of the data class node. This can then be used to query any node that is an instance of the defined class.

Execution Plan: Visualizing Graph-based Metadata

The complete execution can be found in Table A.6

Step 1 A visualization client requires information about all nodes and relationships of a graph in order to visualize the whole graph. Since the graph-based metadata system offers an API that exposes nodes and relationships on different URLs, two requests are needed by a visualization client. The visualization client has to match the information of the graph relationship models to create a visual representation of the graph. However, it only needs to store information about the UUID and the name of each node and relationship.

Step 2 Additional information can be queried when the user selects a single node by performing a GET request of the nodes resource by using the stored UUID. It also includes links to allowed state transitions in form of resource URIs.

Step 3 A file download can be made available to the user by simply embedding the file URL that the server included in its response when querying a data node.

Step 4 To provide the user with a more detailed view of a single node and all its relationships with neighboring nodes, a single request can be made by following its *relationships* hypermedia link. This will display incoming and outgoing relationships of the selected node, but requests are also available to show only outgoing or incoming relationships. This way all the neighboring nodes can be displayed and selected by the user which allows him to iterate over the graph. The process will go to Step 3 and repeats whenever the user selects a new node.

Execution Plan: Transforming CAE Data

The complete execution can be found in Table A.7

Step 1 The login step is the same as in the first execution plans shown in Table A.5.

Step 2 All available plugins will have to be placed in the plugins directory by a system administrator before they can be used. A list of all plugins can then be queried by sending a GET request to the plugins resource. This reveals that the sample plugin defined in Listing A.1 is available to be managed and shows its type and language in the response. It also includes links to the processes of the plugins to guide the client to the available state transitions.

Step 3 The plugin can be started by sending a POST request to the process resource and specifying the name of the plugin that should be started in the plugins URI. To send input to the plugin, an additional file can be uploaded as attachment and its lines will be fed into the standard input of the plugin process. This way a user can specify which data and metadata nodes a plugin shall operate on by sending the corresponding UUIDs as input. The connection details of the graph database as well as the data lake will be shared in environment variables with the plugins. The response includes a random PID, the start time as well as flags indicating the status of the execution. These include a return value which is only applicable if the plugin process has already finished, a *finished* and *userAborted* flag. The response also includes links to the input, output, and error file resources.

Step 4 To check the status of a plugin process, a simple GET request on the plugin process resource can be used.

Step 5 The output of a plugin process can be checked by following the link of its output relation in the plugin process resource that was just described.

Step 6 Similarly, any plugin process output sent to the standard error can be viewed by following the link of the error relation of the plugin process resource.

8 Conclusion and Outlook

The goal of this thesis was to create a software architecture for a graph-based metadata management concept for CAE data. First, the context of CAE in product development as well as the need for a graph-based metadata management has been described. Afterwards, the graph-based metadata management concept of Ziegler et al. (2020) has been described and the need for a metadata management system with a uniform interface has been stated. Use cases have been presented that show how future clients could interact with the metadata management system. These were used to extract software architecture requirements as well as challenges for the graph-based metadata management system. Afterwards, different software architecture concepts for the graph-based metadata management system have been presented by using the architectural styles *Client-Server* (2-Tier, 3-Tier and N-Tier), *SOA*, *REST* and *Microservices* (Message-Oriented and Hypermedia-Driven). Each architecture has been discussed in reference to the previously defined use cases and requirements. The presented Client-Server architectures showed drawbacks in scalability, extensibility and evolvability but excelled at performance and simplicity. In contrast, the presented SOA architecture focused on scalability, extensibility, evolvability and performance but it could become highly complex and is very inflexible due to its standardization. The presented RESTful architecture is great for scalability, extensibility, evolvability, simplicity and availability but imposes a communication and administration overhead, as well as makes some sacrifices regarding performance. The presented *microservice* architectures could overcome the administration overhead and shared the same advantages of the presented RESTful architecture. However, the asynchronous nature of the message-oriented microservices approach is not required by most parts of the graph-based metadata management system and imposes an additional communication overhead. The hypermedia-driven microservices approach had similar advantages with less communication overhead but did not enforce the data integrity of the graph-based metadata management system between different microservices.

Overall, the RESTful architecture and the microservice architecture were the most promising architectures regarding the use cases and requirements. Based on these findings, a final architecture has been introduced as a mix of the REST architectural style and the hypermedia-driven microservices architectural style. The concept architecture was used to implement a prototype as a proof of concept. Finally, the software architecture concept was validated by validating the prototype against the defined use cases. This was done by creating a validation plan and mapping each validation step to a URL of the prototype. These requests have been executed and their requests and responses have been documented as well as the round-trip time has been measured. By showing that each use case can be handled by the implementation of the prototype in a reasonable time, it can be concluded, the selected software architecture is a well-suited choice for the graph-based metadata management.

Outlook

The architecture has been designed with extensibility, evolvability and scalability in mind. Therefore, it can be used in an enterprise environment to integrate with the engineer's product development processes. It would be interesting to conduct a use case study in a real CAE-based product development project to see the overall benefit of the uniform interface and how the architecture scales. Additionally, the uniform interface of the graph-based metadata management system could be integrated in existing applications to speed up product development processes. For example, CAE computer applications could automatically upload data while simultaneously annotating them with its metadata. This could go even further by designing all product development processes and all of their software applications around the graph-based metadata management interface. This creates new possibilities such as continuous monitoring and continuous analysis. The former can be used to monitor data versions and their simulation results while the latter describes an analysis pipeline where a data upload triggers transformation and analysis steps.

A Appendix

API Endpoints of the Prototype

The following four subsections show a detailed overview of each of the prototype's API endpoints. A complete API documentation with more detailed descriptions as well as example requests and responses is bundled with the prototype and can be accessed on the deployment server using the URI */docs/index.html*.

Authentication Endpoint

Table A.1 provides an overview of the prototype's endpoint that provides the capability to generate API access tokens for clients of the graph-based metadata management system.

Metadata Endpoints

Table A.2 provides an overview of the prototype's endpoints that provide access to the following plugin capabilities, respectively.

1. Show all available metadata nodes.
2. Create a new metadata node.
3. Show information of a specific metadata node.
4. Update the information of a specific metadata node.
5. Delete a specific metadata node.
6. Download the CAE file that is referenced by a specific metadata node with the given UUID.
7. Show all relationships of a specific metadata node.
8. Show all incoming relationships of a specific metadata node.
9. Show all outgoing relationships of a specific metadata node.

HTTP Method	URI	Path Parameters
POST	/v1/tokens	

Table A.1: An overview of the API endpoints to access the authentication functionality.

HTTP Method	URI	Path Parameters
GET	/v1/metadata/nodes	
POST	/v1/metadata/nodes	
GET	/v1/metadata/nodes/{uuid}	node UUID
PUT	/v1/metadata/nodes/{uuid}	node UUID
DELETE	/v1/metadata/nodes/{uuid}	node UUID
GET	/v1/metadata/nodes/uuid/file	node UUID
GET	/v1/metadata/nodes/{uuid}/relationships	node UUID
GET	/v1/metadata/nodes/{uuid}/relationships/in	node UUID
GET	/v1/metadata/nodes/{uuid}/relationships/out	node UUID
GET	/v1/metadata/relationships	
POST	/v1/metadata/relationships	
GET	/v1/metadata/relationships/{uuid}	relationship UUID
PUT	/v1/metadata/relationships/{uuid}	relationship UUID
DELETE	/v1/metadata/relationships/{uuid}	relationship UUID
GET	v1/metadata/queries	

Table A.2: An overview of all API endpoints to access the metadata functionality.

10. Show all available relationships between metadata nodes.
11. Create a new relationship between two metadata nodes.
12. Show information of a specific relationships between two metadata nodes.
13. Update the relationship between to metadata nodes.
14. Delete a specific relationship between two metadata nodes.

Ontology Endpoints

Table A.3 provides an overview of the prototype's endpoints that provide access to the following plugin capabilities, respectively.

1. Create an ontology in the graph database by uploading a file which specifies the ontology.
2. Download a file that specifies the existing ontology.
3. Delete the existing ontology from the graph database.
4. Show all graph nodes that represent properties, relationships or classes defined by the ontology.
5. Show information about a specific graph node that represents a property, relationship or class defined by the ontology.
6. Show all graph nodes that represent properties defined by the ontology.

7. Show information about a specific graph node that represents a property defined by the ontology.
8. Show all graph nodes that represent relationships defined by the ontology.
9. Show information about a specific graph node that represents a relationship defined by the ontology.
10. Show all graph nodes that represent classes defined by the ontology.
11. Show information about a specific graph node that represents a class defined by the ontology.
12. Show all graph nodes that represent properties that are defined for a specific class defined by the ontology.
13. Show all graph nodes that represent relationships that are defined for a specific class defined by the ontology.
14. Show all graph nodes that represent relationships that are defined in the range of a specific class which is defined by the ontology.
15. Show all graph nodes that represent relationships for which a specific class is defined in the domain defined by the ontology.
16. Show all metadata graph nodes that are instances of a specific class or any of its subclasses defined by the ontology.
17. Create a metadata node as instance of a specific class defined by the ontology.
18. Delete all metadata nodes that are instances of a specific class defined by the ontology.

Plugin Endpoints

Table A.4 provides an overview of the prototype's endpoints that provide access to the following plugin capabilities respectively.

1. Show all available plugins.
2. Show plugin information.
3. Show all running and previously run processes of the specified plugin.
4. Start a new plugin process with an optional file as plugin input.
5. Show information about a specific plugin process.
6. Stop a specific plugin process.
7. Show the input of a specific plugin process that was provided when the plugin process was started in form of a text file.
8. Show and download the output of a specific plugin process in form of a text file.
9. Show and download the error log of a specific plugin process in form of a text file.

HTTP Method	URI	Path Parameters
POST	/v1/metadata/onto	
GET	/v1/metadata/onto	
DELETE	/v1/metadata/onto	
GET	/v1/metadata/onto/nodes	
GET	/v1/metadata/onto/nodes/{uuid}	ontology node UUID
GET	/v1/metadata/onto/properties	
GET	/v1/metadata/onto/properties/{uuid}	property node UUID
GET	/v1/metadata/onto/relationships	
GET	/v1/metadata/onto/relationships/{uuid}	relationship node UUID
GET	/v1/metadata/onto/classes	
GET	/v1/metadata/onto/classes/{uuid}	class node UUID
GET	/v1/metadata/onto/classes/{uuid}/properties	class node UUID
GET	/v1/metadata/onto/classes/{uuid}/relationships	class node UUID
GET	/v1/metadata/onto/classes/{uuid}/relationships/range	class node UUID
GET	/v1/metadata/onto/classes/{uuid}/relationships/domain	class node UUID
GET	/v1/metadata/onto/classes/{uuid}/instances	class node UUID
POST	/v1/metadata/onto/classes/{uuid}/instances	class node UUID
DELETE	/v1/metadata/onto/classes/{uuid}/instances	class node UUID

Table A.3: An overview of all API endpoints to access the ontology functionality for enhanced metadata management.

Sample Plugin

Listing A.1 shows the implementation of a Java program which is used to test the plugin functionality of the graph-based metadata management. It reads the standard input and writes the content to the standard output. In the next step, it checks the connection details to the graph database and the data lake. Afterwards, it writes some test output to the standard error and pauses for some time.

Sample Ontology

Listing A.2 shows the class definitions of *Data*, *Metadata* and *Process* nodes of the graph-based metadata management in form of an ontology. *Data* nodes can have *Input* relationships with *Process* nodes which in turn can have *Output* relationships with *Data* nodes again. *Data* as well as *Process* nodes can have relationships of type *Metadata* with *Metadata* nodes. All node types have a *name* property but *Data* nodes have an additional *originalFileName* and *filePath* property.

HTTP Method	URI	Path Parameters
GET	/v1/metadata/plugins	
GET	/v1/metadata/plugins/{plugin}	plugin name
GET	/v1/metadata/plugins/{plugin}/processes	plugin name
POST	/v1/metadata/plugins/{plugin}/processes	plugin name
GET	/v1/metadata/plugins/{plugin}/processes/{uuid}	plugin name, and UUID of process
DELETE	/v1/metadata/plugins/{plugin}/processes/{uuid}	plugin name, and UUID of process
GET	/v1/metadata/plugins/{plugin}/processes/{uuid}/in	plugin name, and UUID of process
GET	/v1/metadata/plugins/{plugin}/processes/{uuid}/out	plugin name, and UUID of process
GET	/v1/metadata/plugins/{plugin}/processes/{uuid}/err	plugin name, and UUID of process

Table A.4: An overview of all API endpoints to access the plugin functionality.

Execution Plans

This section shows the execution plans for each validation plan defined in Chapter 7. The execution plans show which requests are sent to the prototype of the graph-based metadata system to achieve the outcome described by each step. The response from the prototype is shown on the opposite side to understand the interaction flow. The time until a response was received is documented for each step. Requests and replies are sometimes shortened to fit on the page which is indicated by three consecutive dots (. . .). Additionally, the process IDs and the UUIDs of graph nodes as well as graph relationships are shortened for readability reasons. To shorten the request and response links, the host *https://ipvs-vm-4:8080* is substituted by the placeholder *{host}*. The execution plans for the use cases defined in Section 4.1.1, Section 4.1.2 and Section 4.1.3 are defined in Table A.5, Table A.6 and Table A.7 respectively.

Listing A.1 An executable Java class used as sample plugin to test the interaction with the graph-based metadata management system.

```
1  import java.util.Map;
2  import java.util.Scanner;
3  import java.util.concurrent.TimeUnit;
4
5  class Plugin {
6      public static void main(String[] args) throws InterruptedException {
7          System.out.println("----Plugin Started----");
8
9          System.err.println("Test Error Output");
10
11         Map<String, String> env = System.getenv();
12         System.out.format("Found %d environment variables%n", env.size());
13         for (String envName : env.keySet()) {
14             System.out.format("%s=%s%n",
15                     envName,
16                     env.get(envName));
17         }
18
19         System.out.println("Input Content");
20         Scanner scanner = new Scanner(System.in);
21         int lineCount = 0;
22         while (scanner.hasNext()) {
23             lineCount++;
24             String line = scanner.nextLine();
25             System.out.format("Line %d: %s%n", lineCount, line);
26         }
27
28         if (args.length > 0) {
29             System.out.println(String.format("Sleeping for %s minutes", args[0]));
30             TimeUnit.MINUTES.sleep(Long.valueOf(args[0]));
31         } else {
32             System.out.println("Sleeping for 1 minute");
33             TimeUnit.MINUTES.sleep(1);
34         }
35
36         System.out.println("----Plugin Finished----");
37     }
38 }
```

Listing A.2 A sample ontology defining the classes, relationships and properties of the graph-based metadata management by using RDF as description framework.

```
@prefix mapi: <https://www.metadata-api.org/mapi#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

<https://www.metadata-api.org/> rdf:type owl:Ontology .

#################################################################
#    Classes
#################################################################

mapi:Data a owl:Class ;
      rdfs:label "Data" .

mapi:Process a owl:Class ;
      rdfs:label "Process" .

mapi:Metadata a owl:Class ;
      rdfs:label "Metadata" .

#################################################################
#    Object Properties
#################################################################

mapi:Input a owl:ObjectProperty ;
      rdfs:label "Input" ;
      rdfs:domain mapi:Data ;
      rdfs:range mapi:Process .

mapi:Output a owl:ObjectProperty ;
      rdfs:label "Output" ;
      rdfs:domain mapi:Process ;
      rdfs:range mapi:Data .

mapi:Has a owl:ObjectProperty ;
      rdfs:label "Has" ;
      rdfs:domain mapi:Data, mapi:Process ;
      rdfs:range mapi:Metadata .

#################################################################
#    DataType Properties
#################################################################

mapi:Name a owl:DatatypeProperty ;
      rdfs:label "name" ;
      rdfs:domain mapi:Data, mapi:Metadata, mapi:Process .

mapi:OriginalFileName a owl:DatatypeProperty ;
      rdfs:label "originalFileName" ;
      rdfs:domain mapi:Data .

mapi:FilePath a owl:DatatypeProperty ;
      rdfs:label "filePath" ;
      rdfs:domain mapi:Data .
```

Validation Step

HTTP Requests

Step 1 - *82 ms*

```
curl '{host}/v1/tokens' -i -X POST \
  -H 'Content-Type: application/json' \
  -H 'Accept: application/json' \
  -d '{
    "username": "admin",
    "password": "123456"
  }'
```

HTTP Replies

```
HTTP/1.1 200 OK

{
  "token": "eyJh. . . m5uQ",
  "expires": "2020-11-05T18:14:45.000+0000",
  "links": [ {
    "rel": "self",
    "href": "{host}/v1/tokens"
  } ]
}
```

Step 2 - *411 ms*

```
curl '{host}/v1/metadata/onto' -i -X POST \
  -H 'Content-Type: application/octet-stream' \
  -H 'Authorization: Bearer eyJh. . . m5uQ' \
  -H 'Accept: application/json' \
  -F 'file=@ontology.ttl;
      ↪ type=application/octet-stream'
```

```
HTTP/1.1 201 Created
Location: {host}/v1/metadata/onto/

{
  "name": "ontology.ttl",
  "_links": {
    "self": {
      "href": "{host}/v1/metadata/onto"
    },
    "ontologyNodes": {
      "href": "{host}/v1/metadata/onto/nodes"
    },
    "ontologyClasses": {
      "href": "{host}/v1/metadata/onto/classes"
    },
    "ontologyProperties": {
      "href": "{host}/v1/metadata
          ↪ /onto/properties"
    },
    "ontologyRelationships": {
      "href": "{host}/v1/metadata
          ↪ /onto/relationships"
    }
  }
}
```

Step 3 - *44 m 53.64 s*

```
curl '{host}/v1/metadata/nodes' -i -X POST \
  -H 'Content-Type: application/json' \
  -H 'Authorization: Bearer eyJh. . . m5uQ' \
  -H 'Accept: application/json' \
  -F 'file=@TestModel;
      ↪ type=application/octet-stream' \
  -F 'node={"labels":["Data"],
      ↪ "properties":{"name":"3D Test
      ↪ Model"}}'
```

```
HTTP/1.1 201 Created
Location: {host}/v1/metadata/nodes/e78a2e

{
  "id": 1,
  "uuid": "e78a2e",
  "labels": [ "Data" ],
  "properties": {
    "uuid": "e78a2e",
    "name": "3D Test Model",
    "originalFileName": "TestModel",
    "filePath": "/user/ubuntu/raw/. . . "
  },
  "_links": {
    "self": {
      "href": "{host}/v1/metadata /nodes/e78a2e"
    },
    "relationships": {
      "href": "{host}/v1/metadata
          ↪ /nodes/e78a2e/relationships"
    },
    "file": {
      "href": "{host}/v1/metadata
          ↪ /nodes/e78a2e/file"
    }
  }
}
```

Step 4 - *89 ms* and *127 ms*

```
curl '{host}/v1/metadata/nodes' -i -X POST \
  -H 'Content-Type: application/json' \
  -H 'Authorization: Bearer eyJh. . . m5uQ' \
  -H 'Accept: application/json' \
  -F 'node={"labels":["Metadata"],
       ↪ "properties":{"name": "Creation
       ↪ Date", "creationDate": "2020-10-06
       ↪ 13:34:042"}}'
```

```
HTTP/1.1 201 Created
Location: {host}/v1/metadata/nodes/0aA26l

{
  "id": 2,
  "uuid": "0aA26l",
  "labels": [ "Metadata" ],
  "properties": {
    "uuid": "0aA26l",
    "name": "Creation Date",
    "creationDate": "2020-10-06 13:34:042"
  },
  "_links": {
    "self": {
      "href": "{host}/v1/metadata /nodes/0aA26l"
    },
    "relationships": {
      "href": "{host}/v1/metadata
           ↪ /nodes/0aA26l/relationships"
    }
  }
}
```

```
curl '{host}/v1/metadata/relationships' -i -X
     ↪ POST \
  -H 'Content-Type: application/json' \
  -H 'Authorization: Bearer eyJh. . . m5uQ' \
  -H 'Accept: application/json' \
  -d '{
    "startNodeId": 1,
    "endNodeId": 2,
    "type": "Has"
  }'
```

```
HTTP/1.1 201 Created
Location: {host}/v1/metadata/relationships/b04cfa

{
  "id": 3,
  "uuid": "b04cfa",
  "type": "Has",
  "startNodeId": 1,
  "endNodeId": 2,
  "_links": {
    "self": {
      "href": "{host}/v1/metadata
           ↪ /relationships/b04cfa"
    },
    "start": {
      "href": "{host}/v1/metadata/nodes/b04cfa"
    },
    "end": {
      "href": "{host}/v1/metadata/nodes/b04cfa"
    }
  }
}
```

Step 5 - *79 ms* and *95 ms*

```
curl '{host}/v1/metadata/nodes' -i -X POST \
  -H 'Content-Type: application/json' \
  -H 'Authorization: Bearer eyJh. . . m5uQ' \
  -H 'Accept: application/json' \
  -F 'node={"labels":["Metadata"],
       ↪ "properties":{"name": "Voxels",
       ↪ "voxels": "324004238"}}'
```

```
HTTP/1.1 201 Created
Location: {host}/v1/metadata/nodes/a3bjk4

{
  "id": 4,
  "uuid": "a3bjk4",
  "labels": [ "Metadata" ],
  "properties": {
    "uuid": "a3bjk4",
    "name": "Voxels",
    "voxels": "324004238"
  },
  "_links": {
    . . .
  }
}
```

```
curl '{host}/v1/metadata/relationships' -i -X
     ↪ POST \
  -H 'Content-Type: application/json' \
  -H 'Authorization: Bearer eyJh. . . m5uQ' \
  -H 'Accept: application/json' \
  -d '{
    "startNodeId": 1,
    "endNodeId": 4,
    "type": "Has"
  }'
```

```
HTTP/1.1 201 Created
Location: {host}/v1/metadata/relationships/6dfsdf

{
  "id": 5,
  "uuid": "6dfsdf",
  . . .
  "_links": {
    . . .
  }
}
```

Step 6 - *81 ms* and *66 ms*

```
curl '{host}/v1/metadata/nodes' -i -X POST \
  -H 'Content-Type: application/json' \
  -H 'Authorization: Bearer eyJh. . . m5uQ' \
  -H 'Accept: application/json' \
  -F 'node={"labels":["Metadata"],
       ↪ "properties":{"name": "Material",
       ↪ "material":"matte finish"}}'
```

```
HTTP/1.1 201 Created
Location: {host}/v1/metadata/nodes/3fhkh2

{
  "id": 6,
  "uuid": "3fhkh2",
  "labels": [ "Metadata" ],
  "properties": {
    "uuid": "3fhkh2",
    "name": "Material",
    "material": "matte finish"
  },
  "_links": {
    . . .
  }
}
```

```
curl '{host}/v1/metadata/relationships' -i -X POST \
  -H 'Content-Type: application/json' \
  -H 'Authorization: Bearer eyJh. . . m5uQ' \
  -H 'Accept: application/json' \
  -d '{
    "startNodeId": 1,
    "endNodeId": 6,
    "type": "Has"
  }'
```

```
HTTP/1.1 201 Created
Location: {host}/v1/metadata/relationships/93fgd9

{
  "id": 7,
  "uuid": "93fgd9",
  . . .
  "_links": {
    . . .
  }
}
```

Step 7 - *2.73 s*

```
curl '{host}/v1/metadata/nodes' -i -X POST \
  -H 'Content-Type: application/json' \
  -H 'Authorization: Bearer eyJh. . . m5uQ' \
  -H 'Accept: application/json' \
  -F 'file=@SimulationResult-20201001.csv;type=application/octet-stream' \
  -F 'node={"labels":["Data"],"properties":{"name":"Simulation Result"}}'
```

```
HTTP/1.1 201 Created
Location: {host}/v1/metadata/nodes/2d05ws

{
  "id": 8,
  "uuid": "2d05ws",
  "labels": [ "Data" ],
  "properties": {
    "uuid": "2d05ws",
    "name": "Simulation Result",
    "originalFileName": "SimulationResult-20201001.csv",
    "filePath": "/user/ubuntu/raw/. . . "
  },
  "_links": {
    . . .
  }
}
```

Step 8 - *70 ms*

```
curl '{host}/v1/metadata/nodes' -i -X POST \
  -H 'Content-Type: application/json' \
  -H 'Authorization: Bearer eyJh. . . m5uQ' \
  -H 'Accept: application/json' \
  -F 'node={"labels":["Process"],
      ↪ "properties":{"name":"Simulation
      ↪ Process"}}'
```

```
HTTP/1.1 201 Created
Location: {host}/v1/metadata/nodes/8pj4q

{
  "id": 9,
  "uuid": "8pj4q",
  "labels": [ "Process" ],
  "properties": {
    "uuid": "8pj4q",
    "name": "Simulation Process"
  },
  "_links": {
    "self": {
      "href": "{host}/v1/metadata/nodes/8pj4q"
    },
    "relationships": {
      "href": "{host}/v1/metadata/nodes/8pj4q
          ↪ /relationships"
    }
  }
}
```

Step 9 - *83 ms*

```
curl '{host}/v1/metadata/relationships' -i -X
    ↪ POST \
  -H 'Content-Type: application/json' \
  -H 'Authorization: Bearer eyJh. . . m5uQ' \
  -H 'Accept: application/json' \
   d '{
    "startNodeId": 1,
    "endNodeId": 9,
    "type": "Input"
  }'
```

```
HTTP/1.1 201 Created
Location: {host}/v1/metadata/relationships/z6md9c

{
  "id": 10,
  "uuid": "z6md9c",
  . . .
  "_links": {
    . . .
  }
}
```

Step 10 - *60 ms*

```
curl '{host}/v1/metadata/relationships' -i -X
    ↪ POST \
  -H 'Content-Type: application/json' \
  -H 'Authorization: Bearer eyJh. . . m5uQ' \
  -H 'Accept: application/json' \
  -d '{
    "startNodeId": 9,
    "endNodeId": 8,
    "type": "Output"
  }'
```

```
HTTP/1.1 201 Created
Location: {host}/v1/metadata/relationships/nsdf65

{
  "id": 11,
  "uuid": "nsdf65",
  . . .
  "_links": {
    . . .
  }
}
```

Step 11 - *52 ms* and *46 ms*

```
curl '{host}/v1/metadata/onto/classes' -i -X GET \
  -H 'Accept: application/json'
```

```
HTTP/1.1 200 OK

{
  "_embedded": {
    "ontologyClasses": [ {
      "id": 1,
      "uuid": "3zu5fd",
      "labels": [ "Resource", "Class" ],
      "properties": {
        "name": "Data",
        "label": "Data",
        "uri": "{host}/mapi#Data",
        "uuid": "3zu5fd"
      },
      "_links": {
        "self": {
          "href": "{host}/v1/metadata/onto/properties/3zu5fd"
        },
        "ontology": {
          "href": "{host}/v1/metadata/onto"
        },
        "ontologyClassProperties": {
          "href": "{ontoClasses}/3zu5fd/properties"
        },
        "ontologyClassRelationships": {
          "href": "{ontoClasses}/3zu5fd/relationships"
        },
        "ontologyClassRange": {
          "href": "{ontoClasses}/3zu5fd/relationships/range"
        },
        "ontologyClassDomain": {
          "href": "{ontoClasses}/3zu5fd/relationships/domain"
        },
        "ontologyClassInstances": {
          "href": "{ontoClasses}/3zu5fd/instances"
        }
      }
    },
    . . . ]
  },
  "_links": {
    "self": {
      "href": "{host}/v1/metadata/onto/classes"
    }
  }
}
```

```
curl '{host}/v1/metadata/onto/classes
    ↪ /3zu5fd/instances' -i -X GET \
  -H 'Accept: application/json'
```

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 1682

{
  "_embedded": {
    "metadataNodes": [ {
      "id": 1,
      "uuid": "e78a2e",
      "labels": [ "Data" ],
      "properties": {
        "name": "3D Test Model",
        . . .
      },
      "_links": {
        . . .
      }
    }, {
      "id": 8,
      "uuid": "2d05ws",
      "labels": [ "Data" ],
      "properties": {
        "name": "Simulation Result",
        . . .
      },
      "_links": {
        . . .
      }
    } ]
  },
  "_links": {
    "self": {
      "href": "{host}/v1/metadata/onto
          ↪ /classes/3zu5fd/instances"
    }
  }
}
```

Table A.5: Execution plan for the use case *Linking CAE Data with Graph-based Metadata.*

Validation Step

HTTP Requests	HTTP Replies

Step 1 - *60 ms* and *54 ms*

```
curl '{host}/v1/metadata/nodes' -i -X GET \
  -H 'Accept: application/json'
```

```
HTTP/1.1 200 OK

{
  "_embedded" : {
    "metadataNodes" : [
      . . .
    ]
  },
  "_links" : {
    "self" : {
      "href" : "{host}/v1/metadata/nodes"
    }
  }
}
```

```
curl '{host}/v1/metadata/relationships' -i -X
    ↪ GET \
  -H 'Accept: application/json'
```

```
HTTP/1.1 200 OK

{
  "_embedded" : {
    "metadataRelationships" : [
      . . .
    ]
  },
  "_links" : {
    "self" : {
      "href" : "{host}/v1/metadata/relationships"
    }
  }
}
```

Step 2 - *44 ms*

```
curl '{host}/v1/metadata/nodes/2d05ws' -i -X
    ↪ GET \
  -H 'Accept: application/json'
```

```
HTTP/1.1 200 OK

{
  "id" : 1,
  "uuid" : "e78a2e",
  . . .
  "_links" : {
    "self" : {
      . . .
  }
}
```

Step 3 - *960 ms*

```
curl '{host}/v1/metadata/nodes/2d05ws/file' -i -X GET
```

```
HTTP/1.1 200 OK
Content-Disposition: attachment; filename=SimulationResult-20201001.csv

...
```

Step 4 - *75 ms*

```
curl '{host}/v1/metadata/nodes/2d05ws/relationships' -i -X GET \
  -H 'Accept: application/json'
```

```
HTTP/1.1 200 OK

{
  "_embedded" : {
    "metadataRelationships" : [
      ...
    ]
  },
  "_links" : {
    "self" : {
      "href" : "{host}/v1/metadata/relationships"
    }
  }
}
```

Table A.6: Execution plan for the use case *Visualizing Graph-based Metadata.*

Validation Step

HTTP Requests	HTTP Replies

Step 1 - *142 ms*

```
curl '{host}/v1/tokens' -i -X POST \
  ...
```

```
HTTP/1.1 200 OK

...
```

Step 2 - *61 ms*

```
curl '{host}/v1/metadata/plugins' -i -X GET \
  -H 'Authorization: Bearer eyJh...m5uQ' \
  -H 'Accept: application/json'
```

```
HTTP/1.1 200 OK

{
  "_embedded" : {
    "plugins" : [ {
      "name" : "Plugin.jar",
      "type" : {
        "name" : "jar",
        "language" : "java"
      },
      "_links" : {
        "self" : {
          "href" : "{host}/v1/metadata/plugins
              ↪ /Plugin.jar"
        },
        "processes" : {
          "href" : "{host}/v1/metadata/plugins
              ↪ /Plugin.jar/processes"
        }
      }
    } ]
  },
  "_links" : {
    "self" : {
      "href" : "{host}/v1/metadata/plugins "
    }
  }
}
```

Step 3 - *78 ms*

```
curl '{host}/v1/metadata/plugins/Plugin.jar
    ↪ /processes' -i -X POST \
  -H 'Authorization: Bearer eyJh. . . m5uQ' \
  -H 'Content-Type: text/plain' \
  -H 'Accept: application/json' \
  -F 'file=@nodes.txt;type=text/plain'
```

```
HTTP/1.1 202 Accepted

{
  "id" : 1,
  "startTimeInMillis" : 1601468828051,
  "startTime" : "2020-10-06 20:54:50.290",
  "finished" : false,
  "userAborted" : false,
  "exitValue" : null,
  "_links" : {
    "self" : {
      "href" : "{host}/v1/metadata/plugins
          ↪ /Plugin.jar/processes/1"
    },
    "input" : {
      "href" : "{host}/v1/metadata/plugins
          ↪ /Plugin.jar/processes/1/in"
    },
    "output" : {
      "href" : "{host}/v1/metadata/plugins
          ↪ /Plugin.jar/processes/1/out"
    },
    "error" : {
      "href" : "{host}/v1/metadata/plugins
          ↪ /Plugin.jar/processes/1/err"
    }
  }
}
```

Step 4 - *30 ms*

```
curl '{host}/v1/metadata/plugins/Plugin.jar
    ↪ /processes/1' -i -X GET \
  -H 'Authorization: Bearer eyJh. . . m5uQ' \
  -H 'Accept: application/json'
```

```
HTTP/1.1 200 OK

{
  "id" : 1,
  "startTimeInMillis" : 1601468828051,
  "startTime" : "2020-10-06 20:54:50.290",
  "finished" : true,
  "userAborted" : false,
  "exitValue" : 0,
  "_links" : {
    . . .
  }
}
```

Step 5 - *44 ms*

```
curl '{host}/v1/metadata/plugins/Plugin.jar
    ↪ /processes/1/out' -i -X GET \
  -H 'Authorization: Bearer eyJh. . . m5uQ' \
  -H 'Accept: text/plain'
```

```
HTTP/1.1 200 OK
Content-Disposition: attachment;
    ↪ filename=1-output.log

----Plugin Started----
Found 9 environment variables
org.neo4j.driver.authentication.password=. . .
org.neo4j.driver.authentication.username=neo4j
hadoop.hdfs.address=192.168.221.54
hadoop.hdfs.username=ubuntu
hadoop.hdfs.files.root=/user/ubuntu
org.neo4j.driver.uri=bolt://192.168.221.42:7687
hadoop.hdfs.port=9000
hadoop.hdfs.files.storage=raw
spring.profiles.active=production
Input Content
Line 1: 2d05ws
Sleeping for 1 minute
----Plugin Finished----
```

Step 6 - *22 ms*

```
curl '{host}/v1/metadata/plugins/Plugin.jar
    ↪ /processes/1/err' -i -X GET \
  -H 'Authorization: Bearer eyJh. . . m5uQ' \
  -H 'Accept: text/plain'
```

```
HTTP/1.1 200 OK
Content-Disposition: attachment;
    ↪ filename=1-error.log

Test Error Output
```

Table A.7: Execution plan for the use case *Transforming CAE Data.*

Bibliography

[AP18] S. Arachchi, I. Perera. "Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management". In: *2018 Moratuwa Engineering Research Conference (MERCon)*. IEEE. 2018, pp. 156–161. DOI: 10.1109/mercon.2018.8421965 (cit. on p. 83).

[Bar13] D. K. Barry. *Web Services, Service-Oriented Architectures, and Cloud Computing: The Savvy Manager's Guide*. 2nd ed. Morgan Kaufmann Publishers Inc., 2013. ISBN: 978-0123983572 (cit. on pp. 45, 46).

[BBNT18] A. Beheshti, B. Benatallah, R. Nouri, A. Tabebordbar. "CoreKG: a knowledge lake service". In: *Proceedings of the VLDB Endowment* 11.12 (2018), pp. 1942–1945 (cit. on p. 34).

[BCK12] L. Bass, P. Clements, R. Kazman. *Software architecture in practice*. 3rd ed. Addison-Wesley Professional, 2012. ISBN: 978-0321815736 (cit. on pp. 25, 33, 35).

[BEK+00] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, D. Winer. *Simple object access protocol (SOAP) 1.1*. 2000 (cit. on p. 46).

[BHL01] T. Berners-Lee, J. Hendler, O. Lassila. "The semantic web". In: *Scientific american* 284.5 (2001), pp. 34–43 (cit. on p. 19).

[BKT00] P. Buneman, S. Khanna, W.-C. Tan. "Data provenance: Some basic issues". In: *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer. 2000, pp. 87–93. DOI: 10.1007/3-540-44450-5_6 (cit. on p. 15).

[CCMW+01] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, et al. *Web services description language (WSDL) 1.1*. 2001 (cit. on p. 46).

[Cur04] E. Curry. "Message-oriented middleware". In: *Middleware for communications* (2004), pp. 1–28. DOI: 10.1002/0470862084.ch1 (cit. on p. 56).

[DHSW02] E. Duval, W. Hodgins, S. Sutton, S. L. Weibel. "Metadata principles and practicalities". In: *D-lib Magazine* 8.4 (2002), pp. 1082–9873 (cit. on p. 17).

[DRC+14] D. Dai, R. B. Ross, P. Carns, D. Kimpe, Y. Chen. "Using property graphs for rich metadata management in hpc systems". In: *2014 9th Parallel Data Storage Workshop*. IEEE. 2014, pp. 7–12. DOI: 10.1109/pdsw.2014.11 (cit. on p. 17).

[Erl16] T. Erl. *Service-oriented architecture: analysis and design for services and microservices*. Prentice Hall Press, 2016. ISBN: 978-0133858587 (cit. on pp. 45, 46).

[Fan15] H. Fang. “Managing data lakes in big data era: What’s a data lake and why has it became popular in data management ecosystem”. In: *2015 IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems (CYBER)*. June 2015, pp. 820–824. DOI: 10.1109/CYBER.2015.7288049 (cit. on pp. 18, 19).

[FF06] M. Fowler, M. Foemmel. *Continuous integration*. 2006 (cit. on p. 83).

[FL14] M. Fowler, J. Lewis. “Microservices a definition of this new architectural term”. In: *URL: http://martinfowler. com/articles/microservices. html* (2014), p. 22 (cit. on pp. 54, 55).

[Fow02] M. Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 978-0321127426 (cit. on p. 68).

[FT00] R. T. Fielding, R. N. Taylor. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine Irvine, 2000 (cit. on pp. 50, 51).

[GGH+19] C. Giebler, C. Gröger, E. Hoos, H. Schwarz, B. Mitschang. “Leveraging the Data Lake: Current State and Challenges”. In: *International Conference on Big Data Analytics and Knowledge Discovery*. Springer. 2019, pp. 179–188. DOI: 10.1007/978-3-030-27520-4_13 (cit. on p. 19).

[GLN+05] J. Gray, D. T. Liu, M. Nieto-Santisteban, A. Szalay, D. J. DeWitt, G. Heber. “Scientific data management in the coming decade”. In: *Acm Sigmod Record* 34.4 (2005), pp. 34–41. DOI: 10.1145/1107499.1107503 (cit. on p. 15).

[HW04] G. Hohpe, B. Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004. ISBN: 978-0321200686 (cit. on pp. 57, 58).

[JVAH07] A. Jansen, J. Van Der Ven, P. Avgeriou, D. K. Hammer. “Tool Support for Architectural Decisions”. In: *2007 Working IEEE/IFIP Conference on Software Architecture (WICSA’07)*. Ieee. 2007, pp. 4–4. DOI: 10.1109/wicsa.2007.47 (cit. on p. 82).

[KAZ18] O. Kopp, A. Armbruster, O. Zimmermann. “Markdown Architectural Decision Records: Format and Tool Support.” In: *ZEUS*. 2018, pp. 55–62 (cit. on p. 82).

[MAC05] P. Mohagheghi, B. Anda, R. Conradi. “Effort estimation of use cases for incremental large-scale software development”. In: *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* IEEE. 2005, pp. 303–311. DOI: 10.1145/1062455.1062516 (cit. on p. 25).

[Mal18] R. Mall. *Fundamentals of software engineering*. 5th ed. PHI Learning Pvt. Ltd., 2018. ISBN: 9789388028035 (cit. on p. 34).

[Mar18] R. C. Martin. *Clean Architecture: A Craftsman’s Guide to Software Structure and Design*. Robert C. Martin series. Prentice Hall, 2018. ISBN: 978-0132350884 (cit. on pp. 33, 34).

[Mat17] C. Mathis. “Data lakes”. In: *Datenbank-Spektrum* 17.3 (2017), pp. 289–293. DOI: 10.1007/s13222-017-0272-7 (cit. on p. 18).

[NH19] M. Needham, A. E. Hodler. *Graph Algorithms: Practical Examples in Apache Spark and Neo4j*. O’Reilly Media, Inc., 2019. ISBN: 978-1492047681 (cit. on pp. 19, 20).

[NJ04] S. S. Nair, V. Jeeven. “A brief overview of metadata formats”. In: *DESIDOC Journal of Library & Information Technology* 24.4 (2004). DOI: 10.14429/dbit.24.4.3629 (cit. on p. 18).

[NMMA16] I. Nadareishvili, R. Mitra, M. McLarty, M. Amundsen. *Microservice architecture: aligning principles, practices, and culture*. O’Reilly Media, Inc., 2016. ISBN: 9781491956250 (cit. on pp. 55, 60).

[NRMK09] T. Nixon, A. Regnier, V. Modi, D. Kemp. “Web Services Dynamic Discovery (WS-Discovery), version 1.1”. In: *OASIS Standard Specification* (2009) (cit. on p. 47).

[Olu14] H. S. Oluwatosin. “Client-server model”. In: *IOSR Journal of Computer Engineering (IOSR-JCE)* 16.1 (2014), p. 67. DOI: 10.9790/0661-16195771 (cit. on pp. 35, 36, 39, 43).

[PD10] H.-S. Park, X.-P. Dang. “Structural optimization based on CAD–CAE integration and metamodeling techniques”. In: *Computer-Aided Design* 42.10 (Oct. 2010), pp. 889–902. ISSN: 0010-4485. DOI: 10.1016/j.cad.2010.06.003 (cit. on p. 15).

[PWA13] C. Pautasso, E. Wilde, R. Alarcon. *REST: Advanced Research Topics and Practical Applications*. Springer, 2013. ISBN: 978-1-4614-9299-3 (cit. on p. 50).

[Ril17] J. Riley. “Understanding metadata”. In: *National Information Standards Organization* (2017), p. 23 (cit. on p. 17).

[RLSB12] M. Rosen, B. Lublinsky, K. T. Smith, M. J. Balcer. *Applied SOA: service-oriented architecture and design strategies*. John Wiley & Sons, 2012 (cit. on p. 45).

[RWE15] I. Robinson, J. Webber, E. Eifrem. *Graph databases: new opportunities for connected data*. O’Reilly Media, Inc., 2015. ISBN: 978-1491930892 (cit. on pp. 19, 20).

[Sch13] R. F. Schmidt. *Software engineering: architecture-driven software development*. Newnes, 2013. ISBN: 9780124078789 (cit. on p. 33).

[Sim16] M. Simic. “Clover: Property Graph based metadata management service”. In: Jan. 2016 (cit. on p. 34).

[SKA15] A. Sharma, M. Kumar, S. Agarwal. “A complete survey on software architectural styles and patterns”. In: *Procedia Computer Science* 70 (2015), pp. 16–28. DOI: 10.1016/j.procs.2015.10.019 (cit. on pp. 35, 36).

[Smi12] B. Smith. “Ontology”. In: *The furniture of the world*. Brill Rodopi, 2012, pp. 47–68 (cit. on p. 20).

[Sul+19] S. Sulova et al. “The Usage of Data Lake for Business Intelligence Data Analysis”. In: *Conferences of the department Informatics*. 1. Publishing house Science and Economics Varna. 2019, pp. 135–144 (cit. on p. 18).

[Wel03] C. Welty. “Ontology research”. In: *AI magazine* 24.3 (2003), pp. 11–11 (cit. on p. 20).

[ZRKM20] J. Ziegler, P. Reimann, F. Keller, B. Mitschang. “A Graph-based Approach to Manage CAE Data in a Data Lake”. English. In: *Procedia CIRP: Proceedings of the 53rd CIRP Conference on Manufacturing Systems (CIRP CMS 2020)*. Elsevier, 2020. DOI: 10.1016/j.procir.2020.04.155 (cit. on pp. 15, 18, 22).

All links were last followed on November 12, 2020.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature