A Proof-of-Concept Prototype of a Database Management System for Versioning and Distributed Storage of Semantic Containers in Project BEST

Submitted by
Johann Paul Stöbich, BSc

Submitted at
Institut für Wirtschaftsinformatik - Data & Knowledge Engineering

Supervisor
o. Univ.-Prof. Dipl.-Ing. Dr. techn. Michael Schrefl

Co-Supervisor
Ass.-Prof. Mag. Dr. Christoph G. Schütz

University
Johannes Kepler Universität Linz

10 2018

Master Thesis
to obtain the academic degree of
Master of Science
in the Master's Program
Wirtschaftsinformatik

Realized in cooperation with
FREQUENTIS

STATUTORY DECLARATION

Abstract

In air traffic management (ATM), information management is of major importance. The safety of aircraft, passengers, and goods depends on effective information management. System Wide Information Management (SWIM) aims at facilitating the information exchange between ATM stakeholders such as pilots and air traffic controllers. In order to realize the full potential of SWIM, the BEST project – a joint research effort between industry and academia within the SESAR joint undertaking of the EU Horizon 2020 framework – proposed the use of *semantic containers* for packaging ATM information and subsequently retrieving relevant information for specific tasks. This thesis presents a proof-of-concept implementation of a *semantic container management system* (SCMS) based on the concepts developed in the BEST project. In this regard, the main focus of this thesis is the development of a REST API which serves as the backend of the SCMS. For demonstration purposes, this thesis also provides a web frontend. The SCMS allows for the replication of *semantic containers* on multiple locations. To this end, the SCMS distinguishes between logical and physical containers. A shared RDF database, which could be fully replicated, serves as a registry of logical containers; it allows for searching of containers that fit a specific information need. Each location then keeps a local registry of physical containers which consists of an RDF database for metadata management and an XML database for storing the actual data sets. Furthermore, each location may store multiple versions of a container, reflecting the evolution of the container contents over time.

Kurzfassung

Im Air Traffic Management (ATM) ist das Informationsmanagement von hoher Wichtigkeit. Die Sicherheit von Flugzeug, Passagieren und Waren hängt von einem effektiven Informationsmanagement ab. Das System Wide Information Management (SWIM) hat zum Ziel den Informationsaustausch zwischen den ATM Stakeholders, wie zum Beispiel den Piloten und den Fluglotsen, zu erleichtern. Um das volle Potential von SWIM zu realisieren, schlägt das BEST-Projekt – ein gemeinsames Forschungsprojekt zwischen der Industrie und der Wissenschaft im SESAR joint undertaking vom EU-Forschungsprogramm Horizon 2020 – die Verwendung von *Semantic Containers* für die Verpackung und die anschließende Auffindung für einen bestimmten Anwendungsfall der relevanten ATM Information vor. Diese Arbeit präsentiert eine proof-of-concept Implementierung eines *Semantic Container Management System (SCMS)*, die auf die im BEST-Projekt entwickelten Konzepte basiert. In diesem Zusammenhang, liegt der Hauptfokus dieser Arbeit auf der Entwicklung einer REST API, welche als das Backend des SCMS fungiert. Zu Demonstrationszwecken stellt die Arbeit auch ein Web Frontend zur Verfügung. Das SCMS erlaubt die Replikation von *Semantic Containers* auf verschiedene Orte. Zu diesem Zweck unterscheidet das SCMS zwischen logischen und physischen Containern. Eine gemeinschaftlich genutzte RDF-Datenbank, welche voll repliziert werden könnte, dient als eine Registrierungsdatenbank für logische Container. Diese ermöglicht das Aufsuchen von Containern, die ein bestimmtes Informationsbedürfnis decken können. Jeder Standort besitzt eine lokale Registrierungsdatenbank für physische Container welche aus einer RDF-Datenbank für die Metadatenverwaltung und einer XML-Datenbank für die Speicherung der tatsächlichen Datensets besteht. Des Weiteren könnte jeder Standort mehrere Versionen von einem Container speichern um die Evolution von ebendiesem Container über die Zeit abzubilden.

Table of Contents

List of Tables

List of Figures

List of Listings

1. Introduction

In air traffic management (ATM), stakeholders such as pilots and air traffic controllers require various kinds of information. For example, a pilot requires information about the current weather, which potentially affects the flight, and the conditions of the runways at the destination airport as well as information about airspace closures. In this regard, the term *situational awareness* refers to a "person's knowledge of particular task-related events and phenomena" [1]. A lack of situational awareness caused by insufficient information can lead to, e.g., the pilot losing control of the aircraft, infringement of closed airspace, collisions of aircraft, and navigation into turbulences or areas of strong winds [1]. Ultimately, a lack of situational awareness may adversely affect security of aircraft and passengers.

System Wide Information Management (SWIM) is a new concept for information management in ATM which is currently being developed. SWIM proposes a service-oriented architecture for information management in ATM where information services provide various applications with ATM information. SWIM relies on a number of standardized information exchange models, e.g., the Aeronautical Information Exchange Model (AIXM) [2] and the Weather Information Exchange Model (IWXXM) [3]. The SWIM registry serves as a directory of information services, allowing applications to find suitable information services.

The semantic container approach as developed by the BEST[1] project [4] provides a data-centric view on SWIM information services [5], [6]. The approach of BEST is to provide the data which is needed for a specific task in a semantic container. A semantic container comprises ATM information for a specific context. For example, a semantic container may contain all relevant weather information for a certain flight on a particular date. Semantic technologies serve to describe the contents of a container. This description, in turn, serves to find a container that satisfies the information need of an application. Containers can be derived from other containers by SWIM information services. In addition to the content description, a container may also keep record of its lineage. The safety-critical nature of air traffic operations requires the redundant allocation of containers at multiple locations. Furthermore, for auditability purposes, a container may keep track of the evolution of its contents through versioning.

This thesis presents a proof-of-concept database management system (DBMS) for versioning and distributed storage of semantic containers[2] – a *semantic container management system* (SCMS). The SCMS builds on the semantic container approach as described in Deliverables 2.1 [5] and 2.2 [6] of the BEST project. This thesis served as the fundamental of parts of Deliverable 3.2 [7] of the BEST project. Furthermore, the prototype developed in this thesis is the foundation for a demonstration at the International Conference on Knowledge Engineering and Knowledge Management (EKAW) [8]. The remainder of this chapter is organized as follows. Section 1.1 introduces the use case and running example. Section 1.2 gives a brief overview of the semantic container approach. Section 1.3 gives an overview of the developed SCMS. Section 1.4 describes the outline of the thesis.

1.1 Use Case and Running Example

In the following, consider the case of a pilot who requires information for a flight from Vienna's Flughafen Wien-Schwechat (LOWW) to New York's John F. Kennedy International Airport (KJFK); this use case serves as running example for the remainder of this thesis. The pilot

[1] Achieving the Benefits of SWIM by making smart use of semantic technologies

[2] In the following, we refer to semantic container simply as container.

requires information about the airports in Vienna and in New York, e.g., runway surface contamination and malfunctioning of navigation aids (see [9] for a list of operational scenarios). The pilot further requires information such as the flight route, the time when the flight is scheduled, the expected flight time, airspace closures, a list of airports for emergency landing. Finally, the pilot requires some country-specific information of the Northeastern United States of America (USA), the Eastern Canada, the Atlantic Ocean, Ireland, the United Kingdom, the Netherlands, Germany, and Austria.

The ATM Information which can be sent or received is standardized. The FAA Aeronautical Information Manual [10] gives an overview of the messages which are exchanged in aviation. In the running example of a flight from LOWW to KJFK the pilot requires Notices to Airmen (NOTAMs), Aviation Routine Weather Reports (METARs), Terminal Aerodrome Forecasts (TAFs), Significant Meteorological Information (SIGMETs), Airmen's Meteorological Information (AIRMETs), airport information, and geographical information about the Flight Information Regions (FIRs) for the flight from LOWW to KJFK. NOTAMs contain information about events, which can influence the pilot's decision on a flight. For example, it contains information about closed runways, taxiways, closed airports, or areaways. METARs and TAFs are weather reports. The difference is that METARs are actual measurements and TAFs are weather forecasts. SIGMETs and AIRMETs are as well weather reports. They provide information about a weather conditions which might be hazardous to all aircraft. SIGMETs are issued in case of severe weather events. For example, whenever there are volcanic ashes or tropical cyclones SIGMETs are issued. In contrast, AIRMETs are issued for significant weather events with a lower intensity. AIRMETs are designed to inform pilots about certain problematic weather conditions before the flight in order to enhance the safety of the flight. For instance, an AIRMET can contain information about turbulences or strong surface winds. Information about FIRs, include regional information to enhance the safety of the flight. Furthermore, it gives information about the organizations which can be contacted whenever aid is needed. Last but not least, the pilot requires information about the existence of airports and terminals.

The required information for a flight from LOWW to KJFK can be packaged into a composite container. A composite container has one or more sub-containers. These containers can be elementary or composite containers. An elementary container consists of data items such as NOTAMs and METARs. Figure 1 illustrates a composite container which contains relevant information for a flight from LOWW to KJFK. In particular, this composite container consists of a composite MET container with weather information for Europe and the USA, a composite NOTAM container with NOTAMs for Europe and the USA, and two elementary containers with information about airports and FIRs. The MET container in turn consists of a composite container for SIGMETs, a composite container for AIRMETs, and a composite container for metrological (MET) information for airports; where each of the composite containers are further separated into two elementary containers, one for Europe and one for the USA. The next section briefly presents the semantic container approach as developed in the BEST project.

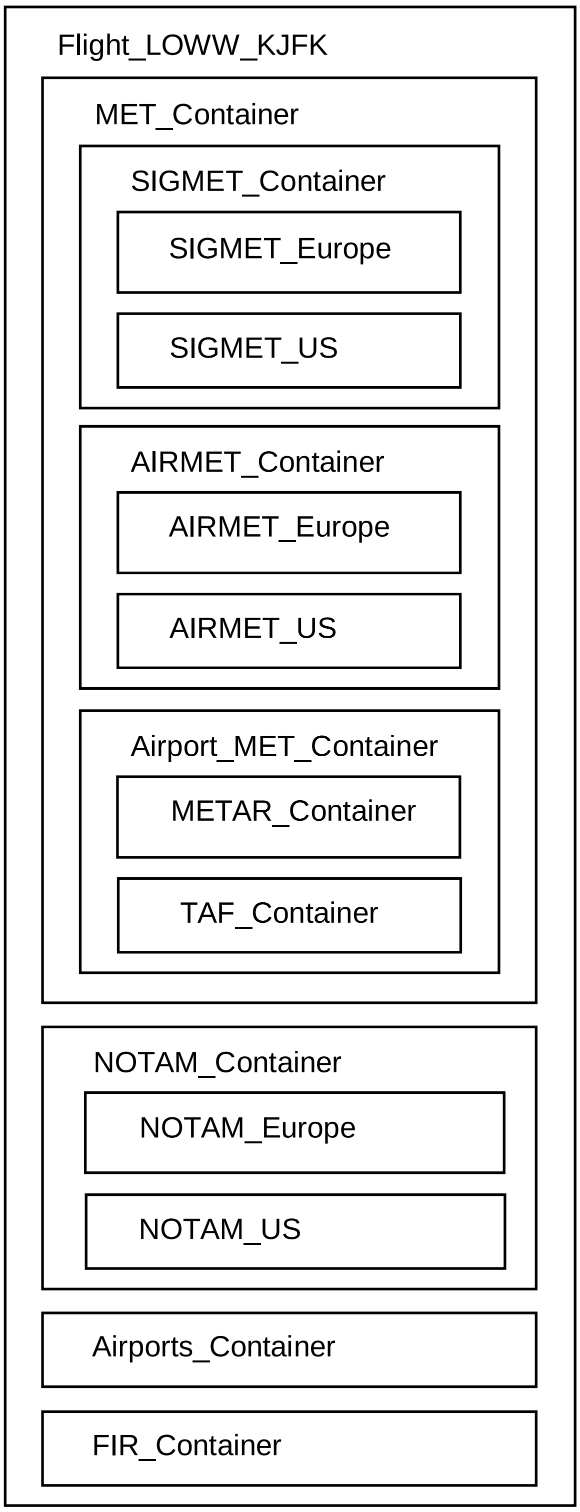


Figure 1: Container for flight between LOWW and KJFK

1.2 Background: The Semantic Container Approach

A container contains the information that is needed for a specific task, e.g., a flight. A container can have data from a single or from multiple SWIM information services. A container contains a set of data items, also referred to as content, and metadata, which consist of a semantic description of the content and administrative metadata. The content description serves to find a container for a specific task. Administrative metadata provide additional information, e.g., data format and encoding. The semantic container approach was developed in the BEST project [4]. The outcome of the BEST project that is relevant for this thesis comprises the Deliverables 1.1, 1.2, 2.1, 2.2, 3.1, and 3.2 (for the Deliverables see [11]). The Deliverable 1.1 investigates the use of ontologies to express the concepts of the ATM domain and Deliverable 1.2 is a proof-of-concept prototype that supports compliance evaluation of ontologies with the ATM Information Reference Model (AIRM) [12]. The developed AIRM ontology is further used as vocabulary to represent the metadata in the semantic container approach that is introduced in Deliverable 2.1. In Deliverable 2.2 concepts for data distribution and replication, consistency management, and provenance and composition of semantic containers are stated. The Deliverable 3.1 illustrates potential application of the semantic container approach and the Deliverable 3.2 demonstrates the semantic container approach by proof-of-concept applications; this thesis serves as the basis for parts of the Deliverable 3.2. In this section, the concepts of the semantic container approach that are relevant for the SCMS are explained.

The semantic description of a container’s content is referred to as semantic label, specifying a membership condition [5]. A semantic label consists of concepts which are taken from an ontology – one for every facet. A facet represents an aspect of the content, e.g., spatial and temporal relevance of the content. Each data item of the container’s content conforms to the specified membership condition. In order to be able to find a container, there must be a common understanding of the concepts used in the content description [13]. For example, there must be a common understanding when an aircraft needs to be classified as a “HeavyAircraft”. The semantic label represents the descriptive metadata which describe the content of a container.

The descriptive metadata describe the content of the data itself. This kind of metadata can be grouped into three categories: temporal, semantic, and spatial, each of which is divided into one or more facets. The spatial metadata provide insight about geographical scope of the data. For example, a specific container may only contain data about Austria. The temporal metadata provide information about the time, e.g., the time when the last update of the container was carried out [13]. The semantic metadata provide domain-specific information about the data [13].

Besides the descriptive metadata there exist the administrative metadata. The administrative metadata can be further separated into technical metadata, quality metadata, and provenance metadata [13]. The technical metadata store the data format, e.g., XML or JSON, the encoding, e.g., UTF-8, the size of the container, the location, and a checksum. The quality metadata store the data quality which is assessed by various metrics [13]. Furthermore, the administrative metadata consist of provenance metadata as described in Section 2.5.

For container discovery, one possibility is using subsumption reasoning [5]. As OWL ontologies have well defined axioms which relate the different classes with each other, one can create subsumption hierarchies. A subsumption hierarchy can be used to find a container with a specific information need [14]. A subsumption reasoner decides whether a class is more specific or more general than another [15]. For example, when a stakeholder requires the information regarding a flight from Vienna to New York on a particular date it is an information need. The information need can be expressed by formulating a membership condition in the same way as the descriptive metadata. The reasoner places the information need in the subsumption hierarchy of membership conditions of containers where the information need fits best. Therefore, the possible results of subsumption reasoning are either a full match, a most-specific subsumer or, additional filters [14]. When a container contains exactly the required data it is called a full match. If no full match is found, then the first more general container which contains the required information is matched. This container is called the most-specific subsumer. If no container is matched, then additional filters must be applied.

A container can be regarded from several perspectives: the distribution and replication perspective, the consistency perspective, and the provenance and composition perspective [6]. Distribution and replication refers to the distinction between logical and physical container as well as the allocation of containers at locations. Provenance and composition refers to the derivation of containers from other containers via SWIM services. Consistency refers to the versioning of containers and the synchronization between locations.

A container can be separated into a logical and a physical container. A logical container is a container that is independent of any physical location and a physical container is a container that is allocated at a specific location. Therefore, one logical container may have multiple physical containers. The main purpose of a physical container is to store the actual data [6]. Every physical container is primarily allocated at a specific location. The container can have a secondary copy allocated at other (secondary) locations. Every secondary location synchronizes with the primary location by using push or pull updates. A push update is whenever a primary copy pushes new changes to all secondary copies. A pull update is whenever a secondary copy pulls the updates from the primary copy. The replicated storage of the physical container on different locations increases the availability of the information and decreases the network load [6]. The logical registry, a shared database, contains the logical model (a RDF graph) – which is used as a registry of logical containers and the containers' metadata. The logical registry might be fully replicated in order to increase, e.g., availability. Furthermore, there exist metadata which are only relevant for a specific container at a physical location. These metadata are stored in the physical model of the physical registry. Every location has a physical registry which contains the physical model. For example, a physical container should contain at least the information when it was last synchronized with the primary copy, when it was last synchronized with a secondary copy, and when it last checked for new updates at a primary copy [6].

In the consistency perspective it is further described how a container is versioned [6]. Versioning allows to revert the physical container to a past state which ensures auditability [6]. The version generation follows the following principle. A physical container consists of a set of data items, e.g., NOTAMs [6]. Every physical container has an initial basic set. On every update, an additional data set is added, a delta set. A delta set can be viewed either as an extension of the existing basic set or it replaces the basic set completely [6].

A container has a certain structure. A container can be elementary or composite. An elementary container consists of data items, e.g., NOTAMs. Those containers can be combined to composites. A composite is a container which consists of multiple sub-containers. A container can be a homogenous container, which only contains data of the same type, e.g., METARs or a heterogeneous container which can contain data items of different types. A composite container can be either a heterogeneous or a homogenous container. Note that there are no heterogeneous elementary containers; all elementary containers are homogenous.

The provenance perspective describes how containers are derived from each other. A container can be derived from a primary source or from a secondary source by a service call [6]. A primary source is the source where the container takes preferably the data from. A secondary source can be specified in addition to a primary source. The secondary source is the source where the container takes its data from when the current primary source is not available anymore [6]. A data set is added to a physical container with an occurrence of a service call. This means that the service actually fetches a data set from the primary source or the secondary source and adds it to a physical container. A service call can occur multiple times [6]. The representation of provenance in the semantic container approach derives from PROV-O (see in Section 2.5).

As already mentioned, containers can be derived from each other. Containers can be combined, composed, filtered, derived, and enriched [16]. In a SWIM-based environment those functions will probably be hardcoded into the application which uses the data. In a BEST-based environment this is handled by the SCMS itself [17]. The combine operation produces a homogenous composite container out of multiple sub-containers. Those sub-containers must contain data items of the same data item type as the resulting homogenous container. In contrast, the compose operation produces a heterogeneous container. The sub-containers in this operation are allowed to be from a different data type. The filter operation transfers data items from one container to another. However, this operation can drop data items which does not match the filter. The derive operation produces exactly the same container as output, but an annotation type will be added. Finally, the enrich operation is a combination of the derive operation and the compose operation. The filter operation has been implemented separately as described in Deliverable 3.2 [7] of the BEST project; integration of this prototype into the SCMS is left to future work.

1.3 Semantic Container Management System: Overview

The SCMS serves for the distributed storage of containers on multiple locations. In order to set up one location of the SCMS, an operating system with a Java Virtual Machine, two Fuseki servers, and a BaseX server for executing the backend are required. Figure 2 gives an overview of the implemented SCMS architecture.

Each location runs an SCMS backend instance. An SCMS backend instance offers a REST interface which can be accessed from outside, uses a global and physical registry to store container metadata, and an eXtensible Markup Language (XML) database to store the actual data. The logical registry is shared between backend instances, possibly as a fully replicated database, whereas each backend instance maintains its own physical registry and XML database. In detail, the logical registry maintains the logical model which is an RDF graph depicting the domain of the SCMS. In the RDF graph, the shared metadata of the containers, e.g., the versions and the semantic label, are stored. On the other hand, the physical registry maintains the physical model. The physical model contains the metadata about the containers that are allocated at the particular SCMS instance to which the physical registry belongs to. For example, for every of the location's containers, the metadata of its data sets are stored. The actual content of a data set is saved in the XML database. The backend instances on different locations synchronize with each other. One location can pull the data from or push the data to another location. However, the physical model and the XML database are not fully replicated. So far, the location where the container should be replicated to can be chosen. To this end, every location keeps a registry of locations.

A web frontend serves to access the containers at a particular location. To this end, the frontend accesses the location's backend via REST interface. The REST interface provides different functionalities, e.g., adding a new container, retrieving the registered containers from the logical model, retrieving the data sets of a container from the location's physical model, or loading the data set's content from the XML database. If multiple locations are available, the frontend can switch between the different backend locations.

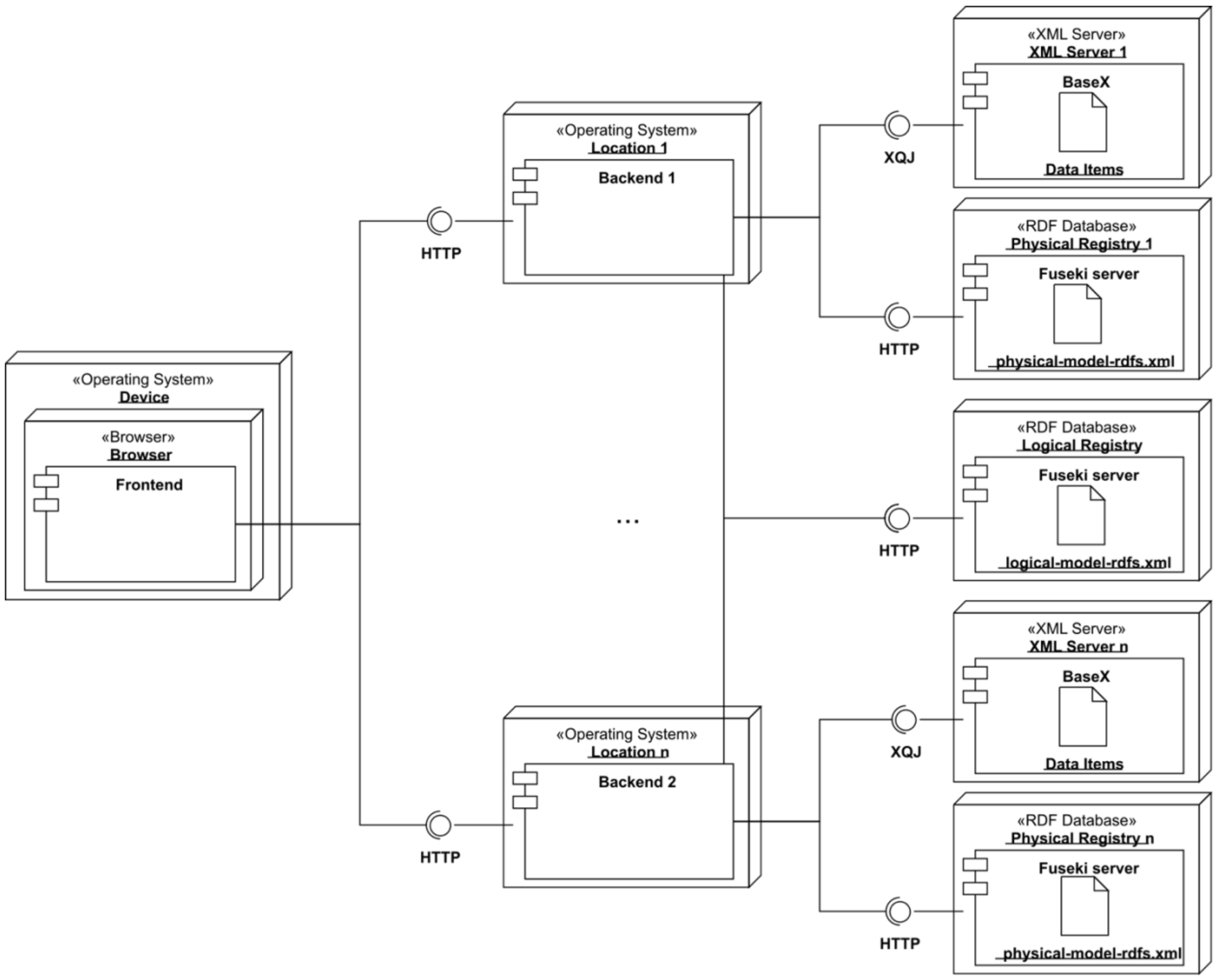


Figure 2: Components of the SCMS

1.4 Outline

The prototype of an SCMS for versioning and distributed storage of containers in the BEST project consists of multiple perspectives: the distribution and replication perspective, the consistency management perspective, and the provenance and composition perspective. Each of these perspectives is represented in each of the two components: the frontend of the SCMS and the backend of the SCMS.

The distribution and replication perspective separates the SCMS into a logical and a physical part. It separates the storage of the actual data from the storage of the information about the data. The consistency management describes a versioning mechanism for semantic containers. The provenance and composition perspective defines the structure of the containers. It distinguishes between a homogenous-composite and a heterogeneous-composite and between an annotated-elementary-container and an entity-elementary-container. Furthermore, the provenance perspective ensures that the service from which the data are issued is documented. In addition to these perspectives, each chapter describes how these services are implemented and explains the major implementation aspects. This overview can be seen in Table 1.

Table 1: Overview perspectives and views

Perspective / Component	Distribution and Replication	Consistency Management	Provenance and Composition
Backend	Section 3.2	Section 3.3	Section 3.4
Frontend	Section 4.1	Section 4.2	Section 4.3

The remainder of the thesis is structured as follows. First, the state of the art is described. The state of the art describes topics that are relevant in the remainder of this thesis. Next, the backend and the frontend with their perspectives are introduced. Finally, in the appendix, a detailed description of the REST interface of the backend, the development environment, and the used software are given.

2. State of the Art

In this chapter, the state of the art is described. This includes the technologies which are used in the implementation and reviews of essential topics. This chapter provides the foundation for the remaining parts.

2.1 Representational State Transfer

Representational state transfer (REST) is an architectural style for the development of web services. The SCMS proposed in this thesis adheres to the REST architectural style for implementing the backend. A REST interface can be invoked by various frontend applications which interact with the backend. This section gives an overview of the REST architectural style as described by Fielding and Taylor [18].

An application must fulfill six constraints in order to conform to the REST principles. An application must have a client-server architectural style, a client-cache-stateless-server style, a stateless communication between a client and a server, a uniform interface between components, a layered system style, and a code-on-demand style.

The client-server architectural style ensures that the concerns of the different parts of an application are separated. There are two concerns which exist. The concerns from the user perspective and the concerns from the data storage. This ensures that the way the data are stored is independent from the way the user uses it. Therefore, a REST application might be used across multiple platforms. Furthermore, the components can evolve independently.

The client-server interaction must be stateless. This means that each request from a client to a server contains all the necessary information which is needed from the server in order to fulfill the request. The session state is entirely kept on the client. This improves the scalability, the reliability, and the visibility. However, statelessness comes with some disadvantages as well: it may decrease network performance and the server does not have the full control over the state of the applications.

Because of the stateless client-server interaction, there might be a lot of network traffic. Therefore, REST requires to make use of a client-cache-stateless-server style. This requires that a response to a request must be saved in a cache, if the response is labelled as cacheable. If the response is not cacheable is must be labelled as non-cacheable.

A REST application must have a uniform interface between components. Therefore, the implementation can be decoupled from the services which it provides. In order to achieve a uniform interface REST relies on four constraints: identification of resources, manipulation of resources through representations, self-descriptive messages, and hypermedia as the engine of the application state. Every resource must have a unique identification and must be returned by using a specific representation. The client, however, will not get direct access to the resource. The client will only get a representation of the resource which it can modify. The format of the representation must be send to the client as well. In turn, the client must send self-descriptive messages to the server, e.g., the representations with some metadata which is required for the server to process the message. All data which is necessary to process a certain message must be sent. The set of representations stored by the client is referred to as application state. Because the set of representations might change when the client interacts with the server the application state changes as well. Therefore, a RESTful application can be seen as an engine which moves from state to state by using state transitions. This matches the way how users interact in a hypermedia browser.

The REST architectural style implies that the system must be a layered system. This means that the components of a system can be composed into hierarchical structures. For a layered system, there must be constraints such as that a component cannot “see” beyond its boundaries.

Within a REST application it is also allowed that executable code is downloaded from the server on demand. This is called the code-on-demand style. This, however, is not a strict requirement; a RESTful interface does not necessarily adhere to apply to this constraint.

2.2 Semantic Web Technologies

The semantic web technologies are of major importance in the BEST project. In the following, the technologies XML, RDF, HTML, and OWL ontologies are introduced as well as the semantic web, which offers a representation which can be interpreted by machines and humans.

The traditional web pages are designed to be read and interpreted by humans only. The semantic web, however, is an approach that the web can be read by machines as well as by humans. The goal of the semantic web is to express meanings so that an application as well as a human are able to interpret information on the internet. For example, the information where a certain container is allocated can be interpreted by a web application as well as by a human [19].

Before 2001, knowledge systems tended to be centralized [19]. In this setting, everyone uses the exact same definitions for concepts, which facilitates the handling of input. However, usually the number of questions which can be asked are limited so that the system can answer them reliably and if a lot of knowledge is managed in a centralized system handling that knowledge is challenging. According to Berners-Lee et al. [19], the semantic web must be decentralized with the advantage that a vast amount of data can be stored. This means that the semantic web is versatile. Therefore, the challenge of the semantic web is to provide a language which allows currently existing knowledge systems to be exported to the web.

For the semantic web two important technologies are the eXtensible Markup Language (XML) and the Resource Description Framework (RDF). XML is a markup language for representing the logical structure of a data file [15]. A markup language is a language which can be used to annotate or tag text fragments in a text file. Because the XML tags are not predefined, one can use XML also as a meta-language with which one can create another markup language [15]. For example, one can define the Hypertext Markup Language (HTML) by using XML.

In general, a valid XML file begins with an XML declaration. It contains the current XML version and optionally its encoding [15]. An XML file consists of exactly one root element which can have multiple sub-elements. An element always starts with an XML-start tag and an XML-end tag. A start tag and an end tag starts with < or </, has a name in the middle, and ends with > [15]. Between a start tag and an end tag, an arbitrary content can be placed. It can be empty, a string text, or further XML Elements [15]. Furthermore, an XML tag can have attributes. An attribute represents the metadata for an XML tag.

```xml
<rdf:RDF xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs = "http://www.w3.org/2000/01/rdf-schema#"
   xmlns:cims = "http://iec.ch/TC57/1999/rdf-schema-extensions-19990926#"
   xmlns:uml = "http://langdale.com.au/2005/UML#"
   xmlns:xsd = "http://www.w3.org/2001/XMLSchema#"
   xml:base = "http://best.frequentis.at/SemanticContainerManagementSystem" >
</rdf:RDF>
```

Listing 1: Namespaces available in the SCMS

To avoid conflicts between tags which are equally named and to create a common understanding of the XML documents semantics, every XML tag is assigned to a specific namespace [15]. One can see in Listing 1 some namespaces with the XML tag “rdf:RDF”. If one takes a look at the namespace “http://best.frequentis.at/ SemanticContainerManagementSystem” one can separate it into three parts: a schema which in this case is “http:”, a server name which is “best.frequentis.at”, and a path which in this case is “SemanticContainerManagementSystem”.

```
<rdf:Description rdf:about="#AdministrativeMetadata">
</rdf:Description>
```

Listing 2: Example XML-Element assigned to a URI

A namespace can be extended by a request and a fragment. One can see a fragment in Listing 2. The XML tag description has the fragment “#AdministravieMetadata”. This is a fragment of the “xml:base” path one can see in Listing 1. Those five things together represent the Uniform Resource Identifier (URI) [15]. One can see the available namespaces of the SCMS in Listing 1.

XML is designed for the correct representation of documents. Data can be stored in a structured way within those documents in a hierarchical order. Nonetheless, the data cannot be linked together as it is intended by the idea of the semantic web. Within the semantic web multiple knowledge bases have to be linked together to create a broader knowledge base. To fulfill this purpose RDF was introduced [15]. RDF is a formal language for the description of structured data. With RDF, one can share data within the semantic web without losing its original meaning. RDF is designed for data integration and processing.

RDF represents the data as a directed graph. In a graph, edges and nodes are connected together. In RDF, such a connection is called a relation. By using RDF, one can state that a container is allocated at a specific location. In this context, the container and the location are resources and the relation is called allocatedAt [15]. Such a relation in RDF would be, e.g., example that the “Flight_LOWW_KJFK” container is “allocatedAt” Linz. In order to identify resources and relations uniquely, RDF requires to use URIs. Every resource has a well-defined URI which allows that a specific resource can be used multiple times within an RDF graph and in the semantic web. The resource is then always uniquely identified. RDF allows further to annotate a resource with a literal. A literal is always of a specific data type. These data types are well defined.

RDF is used to store information about related things. However, it can happen that two different URIs identify what in reality is one and the same object. A solution to this problem is an ontology – an ontology is an instance of a conceptual structure which can be represented by an RDF graph [20]. An ontology consists of a taxonomy and inference rules. A taxonomy defines classes of individuals which can exist in a system and the relations among them [19]. For example, in the SCMS there exists a “HeterogenousCompositeLogicalVersionedContainer” class containing all heterogeneous composite logical versioned containers. This is a specific class of an ontology which might relate to a location where the container is allocated at. Inference rules can be used to deduce information [19]. For example, given that heavy-weight airplanes are all airplanes over 50 tons, inference rules can deduce that an airplane that weights 60 tons is a heavy-weight airplane.

There exist two different OWL ontology specifications: OWL1 and OWL2 where OWL2 is the successor of OWL1. Some syntactic expressions, constructs for properties, extended datatype capabilities, simple metamodeling capabilities, and extended annotations as well as other innovations have been changed or added [21]. Other innovations are that with OWL2 one can declare entities before their first usage, more properties are added, Unified Resource Identifiers (URIs) are changed to Internationalized Resource Identifiers (IRIs) with an extended character set, and the import process of other ontologies is specified more precisely [21]. According to Patel-Schneider, [22] the majority of changes between OWL1 and OWL2 are additions with a few exceptions.

An OWL ontology consists of entities, expressions, and axioms [20]. Entities are the primitive terms of an OWL ontology, every of them is identified by an IRI, which are differentiated into seven types: individuals, classes, datatypes, literals, object properties, data properties, and annotation properties [20]. In OWL2 individuals represent the actual objects in a specific domain. There are two types of individuals: named individuals and anonymous individuals. Named individuals are identified by using a unique IRI whereas anonymous individuals are identified by a local node ID [20]. For this reason, an anonymous individual can only be used within an ontology. Classes represent sets of individuals. In OWL2 datatypes define the possible data values. A literal is the combination of a data value and a data type. A literal consists of a data value to which the data type IRI is appended. Both of them are written in lexical form; this leads to the following format: *“scms”^^datatypeIRI* [20]. Furthermore, object, data, and annotation properties are defined. Object properties connect different individuals, data properties connect individuals with literals, and annotation properties can be used to describe different concepts in more detail.

In OWL2 there exist two types of expressions: property expressions and class expressions [20]. Property expressions are further separated into object property expressions and data property expressions [20]. An object property expression is used to define a relationship between two individuals whereas a data property expression represents a relationship between an individual and a literal. A class expression specifies a membership condition that is constructed by using property expressions and classes [20]. Each individual which satisfies this membership condition is an instance of the class expression. Therefore, a class expression represents a set of individuals. Class expressions can be formulated by using boolean connectives, propositional connectives, enumerations of individuals, object property restrictions, object property cardinality restrictions, data property restrictions, and data property cardinality restrictions [20].

Finally, OWL2 defines axioms. Axioms are statements which define what is true for a specific domain. Axioms can make statements about classes, object properties, and data properties. For example, in the OWL2 Web Ontology Language Structural Specification [20] the following types of axioms are defined: the subclass, the equivalent classes, the disjoint classes, and the disjoint union of class expressions axioms. The subclass axiom states that one class is more specific than another. The equivalent classes axiom states that multiple classes are semantically equal to each other. The disjoint classes axiom specifies that all mentioned classes are disjoint and the disjoint union of class expression says that a union of classes is disjoint to another class. Furthermore, with axioms one can define custom datatypes, can specify keys, and can define assertions.

In future, the real power of the Semantic Web is reached when multiple software systems collect published data, process the collected data, and re-publish the results on the Semantic Web. This results in the effect that the knowledge can be transferred from any software system to any other software system [19]. Because every element is identified by an URI, software systems are able to send some messages to this object and therefore might retrieve some status updates from it.

2.3 Distributed DBMS

In this thesis, a distributed DBMS for storing containers was developed – the SCMS. This section describes the theoretical concepts behind distributed database management systems. Therefore, this section describes the foundations for the development of the prototype. The theoretical concepts which are relevant for a distributed DBMS are described by Ceri et al. [23] and Gilbert et al. [24]; the following summarizes their work.

Distributed databases have multiple advantages over non-distributed databases. Some companies decide in favor of distributed databases because they fit their organization better. In general, it is questionable whether to have large computing centers if the organization is distributed over multiple locations. This is true for the BEST project, because the ATM information must be available, e.g., in planes as well in airports. Furthermore, distributed databases can be used for interconnecting the existing databases. Therefore, a DBMS can be built on top of the existing system. Moreover, if organizations are growing rapidly it can be a reason to decide in favor of distributed databases. For example, if a new system is required at a different airport. This also reduces the communication overhead. Less network traffic will be needed to retrieve a certain information if the database is distributed. This will also have an effect on the performance of the database. One can use separate processors for every local distribution and there is no need to transfer information over a long distance. Finally, the reliability and the availability will increase.

A distributed DBMS is a system which helps during maintenance and creation of distributed databases. A typical distributed DBMS provides functionalities for accessing the database by an application, for offering a degree of distribution transparency, for administrating the database, and for concurrent control and recovery of the system. The feature for accessing the data is the most important feature of a DBMS. The access to the data can be through the DBMS system directly or through an auxiliary program. The difference is that in one case the DBMS provides the functionality itself whereas in the other case the DBMS uses another program to access it. Furthermore, one can differentiate the DBMS into a homogenous and a heterogeneous system. The difference between those two is that in a homogenous system only one DBMS exists which will be executed at every instance. In a heterogeneous system however, there will be multiple different DBMS which communicate with each other. It is obvious that a heterogeneous DBMS is more complex because one must also translate between different data models.

According to Ceri et al. [23], a DBMS consists of a site-independent schema and a site-dependent schema. The site-independent schema consists of a global schema, a fragmentation schema, and an allocation schema. The site-dependent schema consists of a local mapping, the DBMS, and the local database where the data are stored. One can see the different types in Figure 3. The global schema defines which data can be stored in the distributed database. It does neither determine how the data are fragmented nor where they are stored. This is defined by the fragmentation and by the allocation schema. The fragmentation schema describes in which parts the distributed database is separated. In other words, it defines in which part the global schema will be split up. Finally, the allocation schema defines at which location the different fragments are allocated. Data can be allocated either redundantly or non-redundantly at a specific location.

In general, there are two types of fragmentation. One can fragment a database horizontally or vertically. A horizontal fragmentation is when a single schema is split into multiple parts by using the values of their tuples. For example, this can be when the information for New York is stored in the database for New York and the information for Vienna is stored in Vienna. However, both fragments do have the same schema. A vertical fragmentation is whenever a global schema is fragmented by using the schema's attributes. However, one must ensure that the different schemas can be joined together so that a reconstruction is possible. In general, the fragmentation must be designed so that the global relation can be recreated. Composite containers in the SCMS can be considered as fragmented data structures, possibly both horizontally and vertically.

Because of the different levels, one can define different transparencies. The transparencies define which distribution details can the user see when interacting with a distributed database. For example, does the user even recognize that he is dealing with a distributed database or must the user know the exact site[3] where the queried data resides. According to Ceri et al. [23], there are four levels of transparencies which can be distinguished. First, the fragmentation transparency, second, the location transparency, third, the local mapping transparency, and fourth, no transparency at all. If a DBMS provides *fragmentation transparency*, then the user can execute a select statement without knowing anything about the fragmentation of the DBMS. Therefore, one does not need to know of which fragments the DBMS consists. One can issue a query without knowing anything about the database. A DBMS provides *location transparency* but not *fragmentation transparency* whenever the user needs to know which fragments in a database exist in order to get a result from a query. One has to target the fragment specifically. When a DBMS provides *local mapping transparency* the user of a DBMS must further know at which location a specific fragment is allocated.

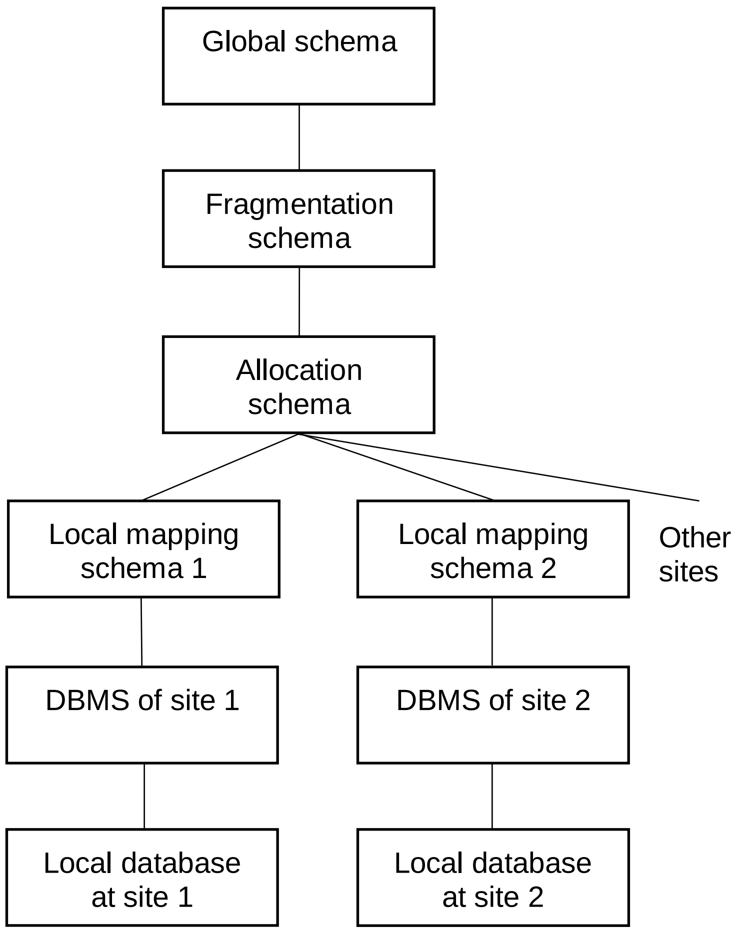


Figure 3: Fragmentation and allocation of a DBMS [23]

[3] In the following, we refer to site and location as synonyms.

Distributed databases contain three desired features: consistency, availability, and partition-tolerance (CAP) [24]. Availability means that every request sent to a distributed database must always result in a response. This is an important feature and mostly desired by the customers of distributed databases. If a distributed database is once connected to the network, then it should always be available. Distributed databases are required to be consistent. This means that whenever a transaction is executed, either it is fully executed or reverted so that either the new state or the old state is available. Finally, it is desired that the DBMS allows some sort of fault tolerance. If something crashes, it is still desired that the components function as given. The CAP theorem states that it is feasible to achieve two of the three desired properties. However, it is impossible to achieve all of them.

2.4 Transaction Management

In the SCMS, the transaction management is of relevance, e.g., during the synchronization and allocation process of containers. For example, the SCMS must ensure that each data item is available at each location where a container is allocated. Whenever a data item is added to the SCMS it is a transaction, because the operation must be executed either fully or it must be reverted, it must not have an effect if the data item is added concurrently or serially, when it was successfully added the data item must be persisted, and the data item must be only available to other programs when it was inserted successfully. In this thesis, a custom approach is employed to deal with transactions that cannot be executed on all locations, e.g., because the location might not be available. In this case, the SCMS works with “degenerated” data sets. This section gives a theoretical background to transaction management as described by Ceri and Pelagatti [23].

A transaction consists of one or more operations. The transaction’s operations must be executed atomic, durable, serializable, and isolated. A transaction is atomic when it is either executed fully or is not executed at all. Therefore, an atomic transaction can be either successful or must be reverted. A transaction must be reverted when, e.g., the system crashes or the transaction is aborted which can happen, e.g., because some constraints are not satisfied or it is forced by the system itself. For example, in the SCMS a data item must be registered in the physical model and must be stored in the XML database. If one of the operations fails, the other one must be reverted. A transaction is considered as completed when it is committed. Durability means that when a transaction is committed the result must be preserved and stored so that it cannot be lost even when, e.g., the system crashes. Furthermore, whenever a transaction is executed concurrently it must lead to the same result as if the transaction is executed serially. Finally, a transaction must be isolated. This means that the effects of a transaction must be only available to other applications when it is finished. The transaction management ensures that each transaction fulfills these properties and is executed efficiently, concurrently, and reliably.

In distributed databases the challenge is, e.g., that distributed transactions are executed reliably, concurrently, and efficiently. In distributed DBMS, the operations of a transaction are executed by one or more processes. At each site, where the transaction should be executed, at least one process must be in place. These processes are called agents. For each transaction, there exists a superior agent, the root agent. The root agent manages the execution of a transaction: It must start and abort or commit the transaction. In general, a transaction in a distributed DBMS is executed in the following way: The root agent requests agents at different sites. An agent is in charge of handling the transaction at the respective site by, e.g., forwarding the commands to a local transaction manager and exchanging messages with the other agents including the root agent. The local transaction manager can execute local begin_transaction, commit, or abort primitives. The root agent starts the whole transaction by creating a begin_transaction primitive. Each agent forwards the primitive to the local transaction manager that is able to execute the transaction as a local atomic transaction. The local atomic transaction modifies the local storage at the respective site. A local atomic transaction can be either successful or not successful. The agents must inform the root agent about the transaction results and then the root agent can decide whether to commit or abort the transaction. A transaction can be only committed by the root agent, if all local transaction managers could carry out the local atomic transactions successfully; otherwise it must be aborted. When the root agent decides to commit or abort a transaction it informs the agents which in turn commands the local transaction managers to commit or undo the local atomic transaction. In the SCMS, a data set which is added at a primary location must be distributed to the secondary locations. The primary location acts as a root agent which is responsible for distributing the data set to the secondary locations. Each secondary location represents an agent which is responsible for storing the data set properly.

The 2-phase-commitment protocol states a way how the decision-making process can work. The decision-making process refers to the process to decide whether to abort or commit a transaction. The 2-phase-commitment protocol can handle site failures, lost messages, and network partitions. It ensures that the sites of a DBMS agree on one decision and execute the respective operations. During the first phase, the root agent asks the participants; an agent and a local transaction manager at one site are together referred to as a participant; for commitment. Each participant has to return READY. This means that each participant has already written the log entries and is prepared to commit the transaction even when failures occur. When the root agent received all answers (READY or ABORT) from the participants or a timeout expired, it decides to either commit or abort the whole transaction. If one participant did not reply or sent an abort message, the whole distributed transaction must to be aborted. In the second phase, the root agent writes its decision to a log and informs the participants about the result. If a participant receives the information it can process the result and subsequently delete the local recovery information about the transaction. Furthermore, each participant sends an acknowledgment message to the root agent. If the root agent received all acknowledgment messages from the participants, it can delete the information about the outcome of the transaction as well. Instead of employing a 2-phase-commitment protocol, the SCMS works with “degenerated” sets and therefore the SCMS is not fully consistent.

2.5 Provenance and Information Quality

A container contains information of a specific quality level which was issued from a specific service. The stakeholders, who consume the information of a container, want to know of which quality the stored data in a container is and where the data came from. Therefore, the quality of the data and its origin must be documented by the SCMS. For the documentation of the data quality, the eight quality dimensions which are taken from Batini et al. [25] are explained. For the documentation of the provenance, a common model – the PROV family of documents – is stated. The PROV Data Model (PROV-DM) is used as the fundamental for the representation of provenance in the SCMS. This section refers to the following parts of the PROV family of documents: to the overview [26], to the introduction of the PROV-DM [27], to the PROV-DM [28], and to the ontology which is a serialization of the PROV-DM [29].

The data quality can be assessed by using the following data quality dimensions [25]: accuracy, completeness, redundancy, readability, accessibility, consistency, usefulness, and trust. The accuracy can be differentiated into syntactic accuracy and semantic accuracy. For example, “Vienna” is syntactically correct whereas “Viena” is not. It would be a semantical inaccuracy if someone asserts that “John F. Kennedy International Airport is in Vienna”. The completeness of a cluster measures how complete a specific information is. This depends on whether one assumes an open or a closed world. In a closed world, the completeness can be measured by using the metrics value completeness, tuple completeness, attribute completeness, and relation completeness. However, if one assumes an open world one can only estimate the completeness. Another dimension of data quality is redundancy; the higher the data quality the lower the redundancy. Furthermore, the data must be readable. The data quality is higher when the information is available for multiple user groups. People who have difficulties with reading a comprehensive text, are deaf, or are blind must also be able to access a specific kind of information. The data are consistent when there is no contradiction to all properties of interest. Finally, data quality takes into account that the data must be useful, and the sources must be trustable. Trust includes that the source is reliable, has reputation, and is believable.

Provenance, or the origin of data, represents the description of how the given data was collected. For example, provenance can be used to identify which organization or individual owns the right over a specific object or provenance can be used to assess the trustworthiness of a specific source by identifying the origin of the data. The provenance information about an object is considered as metadata of a specific object. We employ the following definition of provenance [28]: "Provenance is defined as a record that describes the people, institutions, entities, and activities involved in producing, influencing, or delivering a piece of data or a thing.".

One can view provenance from different perspectives: from the process-centered, from the object-centered, and from the agent-centered perspective. The process-centered perspective focuses on how the information was created. This means which steps are involved in creating this information. The object-centered perspective shows of which parts a document is assembled. For example, it focuses on the parts a container is assembled from. The agent-centered perspective focuses on which organization or individual created the different parts.

The PROV family of documents share a common provenance model: the PROV-DM. It is especially designed to provide provenance for heterogeneous environments, e.g., for the web. The PROV family consists of multiple serializations of the PROV-DM. There exist serializations for linked data (OWL2), for XML, and for PROV-N, a notation which was developed to be human readable. Furthermore, the PROV family provides constraints which help developers to create valid provenance models. Moreover, the PROV family defines provenance access and queries, which specifies how data can be accessed by using standardized protocols over the web, contains a mapping between Dublin Core and the provenance OWL2 serialization, has an extension for dictionaries, defines how provenance information can be linked across provenance bundles, and contains and extension which clarify the specification.

The PROV-DM consists of core structures, an extended structure, and a qualified structure. The core structures of the provenance model consist of an entity, an activity, and an agent. An entity is a thing which can be physical, digital, or conceptual. It can exist in a real as well as in an imaginary world. An activity is somewhat that acts on an entity and can occur in a certain time period. Thus, an activity can produce and use entities. If the production process of an entity is finished then it is stated that an entity was generated by an activity. Furthermore, activities can exchange information with each other. This means that an activity informs another activity. The same thing can happen with entities. When an entity is created, the creation process can be influenced by another entity. For example, for the creation of a container one needs multiple data sets – the container and the data sets are entities. Finally, an agent is something that has the responsibility for an activity. One can see these coherences in Figure 4.

An agent is something that takes the responsibility for an activity. This means that an agent has a role in a specific activity. Agents can delegate their function to other agents. Therefore, an agent can act on behalf of another agent. Furthermore, an agent can be accountable for the existence of a specific entity.

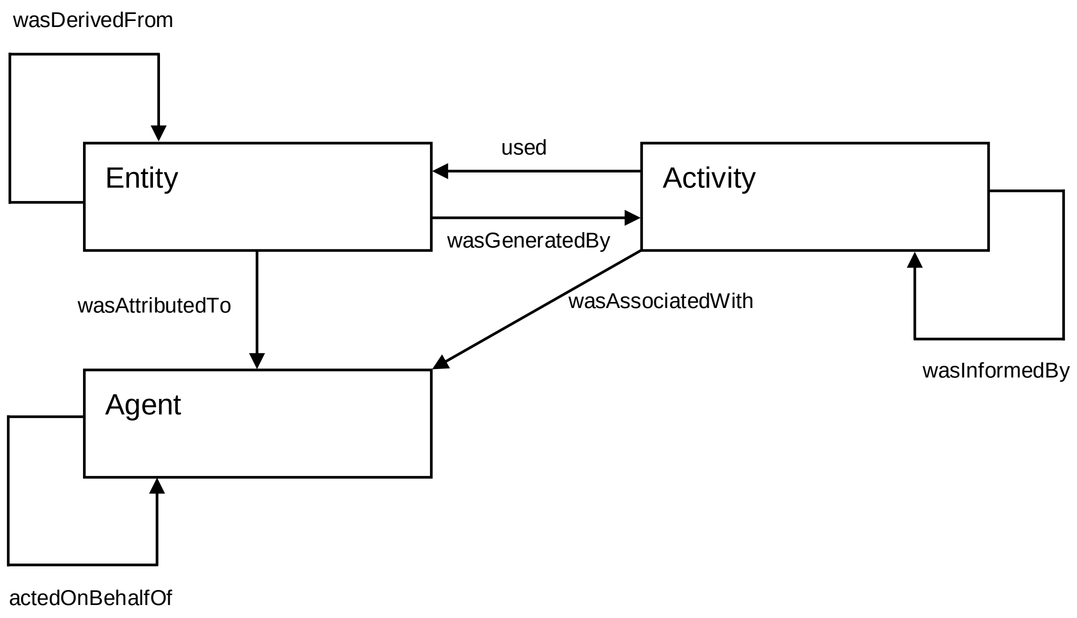


Figure 4: PROV-DM core [27]

The PROV-DM contains extended structures. This is designed to support more advanced structures. The basic provenance model can be extended by using four mechanisms: subtyping, expanded relations, optional identification, and further relations. Subtyping can be applied to an entity, activity, and agent. It can be used to specify these more closely. Another possibility to extend the core structure are expanded relations. An expanded relation is a subtype of a relation. With this concept, one can specify a relationship between objects more closely. Furthermore, one can extend the core of the PROV-DM by using optional identifications. With these one can identify instances of an association between two elements. Finally, the PROV-DM allows to specify further relations.

Within the section extended structures the PROV-DM does not only define mechanisms to extend the given core structure but introduces two more concepts, namely bundles and collections. A bundle is a defined set of provenance information. To a bundle further provenance information can be attached. This enables that provenance information can be attached to itself – provenance information. A collection represents an entity which itself consists of multiple entities. For example, the provenance of a heterogeneous container consists of the provenances of the sub-containers.

A serialization of the PROV-DM is the provenance ontology, PROV-O. It expresses the PROV-DM in OWL2 language. The PROV-O cannot only be used to serialize provenance information but can also be adapted to represent domains and application specific provenance information. It facilitates interoperability between different provenance models. PROV-O contains three different categories with three different levels of detail. These three levels are called starting terms, expanded terms, and qualified terms. PROV-O maps the classes and relations of the PROV-DM into OWL2 classes and properties. All PROV-O terms have the namespace “prov: http://www.w3.org/ns/prov#”.

The PROV-O’s starting terms contain the same classes as the PROV-DM core structure: an agent, an activity and an entity. The relations of the PROV-DM are extended by two properties which can be attached to an activity: a start time and end time.

Another category of the PROV-O are the expanded terms. First of all, the expanded terms contain multiple subtypes for the classes agent and entity. An agent is divided into person, organization, and softwareAgent. The entity is subtyped into collection, bundle, and plan. In addition, more properties are added by using subtyping or adding. There is an additional property called wasInflucendedBy. It states that an agent, entity, or activity is influenced by another agent, entity, or activity. The property wasDerivedFrom is subtyped into the following properties: wasQuotedFrom, wasRevisionOf, and hadPrimarySoruce. They are specializations which describe how entities are related with each other. Second, the properties specialzationOf and alternativeOf allow to describe the relation between two entities. The first one describes if an entity is a generalization of another entity and the second one can be used to link entities together which describe the same or different aspects of the same thing. Third, one can add a value to an entity by using the property of the same name. The value is an OWL2 literal. The location of an entity, activity, and agent can be stated by using the class location and the property atLoctation. Fourth, PROV-O enables to annotate the time where an entity was generated and invalidated. Therefore, the properties generatedAtTime and invalidatedAtTime exist. Furthermore, the activity which generated or invalidated the entity can be annotated by using properties. The properties are called wasInvalidatedBy and wasGeneratedBy as well as the inverse properties generated and invalidated. Fifth, because activities can also be started or stopped by specific entities, the properties wasStartedBy and wasEndedBy are created.

The last category of the PROV-O specification are the qualified classes and properties. They are defined to provide additional information about a relation. RDF only allows binary relations. This means that a property can only connect one instance with another. It is not possible to qualify or annotate a relation. Therefore, if one wants to have a qualified relation a new class must be generated and a new instance of the class is needed which can be annotated by using further properties [30]. The PROV-O comes with these classes which allow to qualify a relation. A detailed specification of the classes and properties can be found at [29] which are used for qualifying the relation.

2.6 Versioning

For several reasons, one might want to assess which data was in a container at a certain point in time. Therefore, the containers are versioned. With the SCMS, one can inspect the versions which existed at a certain point in time.

According to Beech and Mahbod [31], a version control must fulfill the following requirements: it must support the creation of a version of an object and the unique identification of a version, the system must be able to reference a specific version, the system must be able to cause new actions when a new version was generated and versioning must be integrated in a collaborative design environment [31]. Transaction mechanism, ownership, and access right privileges are further parts of the environment.

A specific individual version is a snapshot of an object state [31]. Each object is uniquely identified by an object identifier. Every individual version represents a state of an object which might be related to the last preceding state of it. In addition to individual versions, there are also generic versions. A generic version holds an abstraction of an object. The abstraction knows the existing individual versions of the object. That is, it contains a set of versions of an object that exist. Furthermore, a generic version knows how these versions are related with each other and it might hold some information about itself [31]. There exists only one generic version of an object for multiple individual objects [31]. One can see the relation between two individual versions and its generic version in Figure 5.

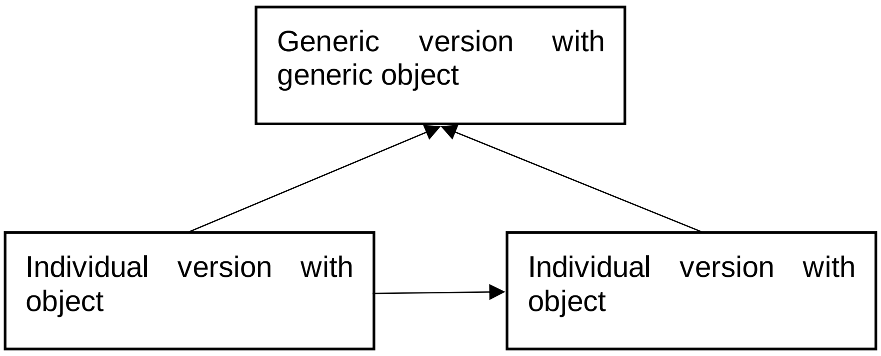

Generic version with generic object
Individual version with object
Individual version with object

Figure 5: Relationship between two individual versions and a generic version [31]

As there are individual versions and generic versions [31], there are also two types how versions can be referenced. Whenever a version of an object is referenced generically, the system itself decides which individual version is returned. Normally, this would be the current state of the object. However, one can access also an individual version as well.

In order to distinguish which individual version leads to which individual version, a simple time stamp is not sufficient. Because one object can have two versions which evolve simultaneously, one needs to record the predecessor and successor of every version [31]. This leads to graph structure.

According to Beech et al., three methods of version generation exist. The first method is to check out an individual version, modify the object and then check it in again [31]. This process leads to a new individual version. This allows that a single object can be modified by multiple individuals or processes at the same time. In order to do so, an object of an individual version can be forked into multiple objects. If those objects are modified, they represent an individual version again [31]. To create one single object out of multiple objects of the same type one must merge them together. A merge of two objects will lead to a new induvial version. The second method is to 'freeze' an object in order to do modifications on it. By using this approach, only one can modify a single object and therefore it is not possible that two copies of a single object are evolving concurrently. During the first and the second method only new individual versions are created when a file is modified. The generic version stays the same. However, the third method is to create a new version out of non-versioned objects. This leads to a new generic version which contains one individual version. The object itself is the first individual version [31].

In the context of versioning, a delta is a set of changes which lead from one version to another [32]. A change can be an addition, a deletion, or a modification. One can have different granularities on which the changes are recorded. For example, if a character is changed in one line, one can store the exact character and its exact position or one can store the line number and the full line which was changed [32]. The use of deltas is a space time tradeoff. Instead of storing a version as a full snapshot, which consumes a lot of space but no computing power, is required for reconstruction, one can store a version by using the deltas (changes) which were made from a predecessor or successor. The second approach needs less space but requires more computing power to construct a certain version, especially when multiple deltas are involved. It must be mentioned that the delta approach does not make full snapshots superfluous, however it reduces the required number of full snapshots [32]. Delta sets can be stored as forward deltas or as reverse deltas. The difference is that by using reverse deltas one take a higher version and undo the deltas whereby using forward deltas one take a lower version and redo the deltas [32]. In the SCMS, a delta set is a set of data items. It represents the delta between two container versions.

2.7 System Wide Information Management

SWIM defines a service-oriented architecture for exchanging ATM information. The data items which are stored in a container might be issued from SWIM information services. Furthermore, containers can serve as inputs for SWIM information services.

ATM stakeholders, e.g., pilots or air traffic controllers, require information. This information can be diverse. For example, it can be flight information, weather information, and NOTAMs. Currently this information is provided via a paper-based system or with custom information systems where the participants use individual communication protocols in order to transfer the data [13]. The stakeholders are limited to local custom applications where they have to agree the terms of service with every service provider.

For this reason, the SWIM concept describes a service-oriented architecture where service providers offer their services to ATM stakeholders. The service providers must offer their data in compliance with standard information exchange models [33]. Some examples for information exchange models are the Flight Information Exchange Model (FIXM), AIXM, and IWXXM. Using standardized models leads to the advantage that stakeholders can consume the information from different service providers without adapting to a specific format of a service provider. On the other hand, service providers can offer their services to multiple stakeholders without adapting their services as well. Therefore, the SWIM concept can be viewed as a system of systems which connects service providers with stakeholders [33]. Service providers can register their applications in the SWIM registry [33]. The stakeholders then can query the existing applications by specifying the desired information, Quality of Service (QoS), timeliness, and accuracy. Then SWIM connects the provider and the stakeholder by using a message broker. There are two types of communications: a peer-to-peer communication for time-critical requests and publish/subscribe communication for less critical demands [33].

The SWIM concept envisions the following components: Registry & Directory, Messaging & Brokering, Adaptation, Information Assurance, and System Management [33]. The Registry & Directory represents the core component. This component provides a registry of the services. In the registry, the services are stored along with the data they provide. Furthermore, the Registry & Directory stores the information where the services are located at and who is authorized to change some data. The Messaging & Brokering is responsible to ensure that applications can share data with each other. It acts as an intermediary so that the applications do not need to know each other. The Adaption component offers functionalities where the data can be formatted, adapted, converted, translated, packaged, and so forth. The Information Assurance component provides services which ensures the confidentiality, authentication, information integrity, etc. and finally, the SWIM concept envisions a System Management component which can, e.g., manage the performance or the security.

3. Semantic Container Management System: Backend

The backend is the backbone of the prototype. It is in charge of managing the provenance, the composition, the consistency, the distribution, and the replication. To achieve this, the backend of the SCMS provides RESTful services. The backend of the SCMS runs on one or more locations. Every location is represented by multiple server processes. One server process provides the interface to applications, e.g., the frontend, one keeps the data consistent with other locations, one provides the physical registry, and one provides the logical registry.

First in this section, the system configuration is described. The system configuration refers to the example setup of the backend used in this thesis. Next, the concepts which are used for realizing the thesis are stated. The concepts are separated into three perspectives: the distribution and replication perspective, the provenance and composition perspective, and the consistency management perspective. There is a small introduction to each perspective and then the classes which correspond to this perspective and their coherences are explained.

3.1 System Configuration

In the backend of the SCMS there are two major parts. The logical model and the physical model. The logical model contains the registry of the logical containers, the services providers, the services, the facets, the ontologies, the administrative metadata, the semantic label as well as the assignments between the containers and the administrative metadata. The logical registry stores the logical model which is shared for all locations of the SCMS. In practice, there might be multiple replications of the logical registry; however, one logical registry which hosts the logical model for one or multiple locations is sufficient. In this prototype, there is one logical registry which is realized with a Fuseki server.

In comparison to the logical model, the physical model is not location-independent. In short, it stores the actual data which is managed by the backend of SCMS. Each of these locations of the SCMS has its own physical registry where the physical model is stored and a BaseX server, where the data of the allocated containers are stored. For example, it contains the NOTAMs and TAFs which are stored at a specific location. The management processes, e.g., the PULL-Updater, are also required at each location. Management processes are processes which are executed in the background and provide functionalities that must be carried out periodically. The physical registry is realized as well with a Fuseki server.

The data which is contained in the physical model differs from location to location. It registers the physical containers, references the container's data sets, and the container's physical versions. These data are stored in the physical model of a certain location only if the container was allocated at that location. The references to the data sets and the physical versions are allocated with the container. Because of this information, the physical model can be used to, e.g., identify the data sets that are stored in a container, the container's current version, and the container's historic versions. The actual content of the data sets – the data items – are stored in an XML server called BaseX. The physical model holds references to the data sets that contain the data items. The XML server archives XML files under a specific path and allows retrieving and querying them. In the physical model only the path to these data sets is stored.

In Figure 6 one can see an example setup with two locations. In this example, the SCMS's processes run on three virtual machines. This is only an example. It is possible that each process might be executed on its own virtual machine as well as that all processes – including the logical registry – share one virtual machine. The later was used to test the SCMS backend.

The only access from outside to the SCMS backend is the RESTful interface. It provides services which offer information about the current state of the location including the data that is stored in the physical registry and in the XML database. But it also provides services which return information from the logical registry which is shared between different locations. Furthermore, the RESTful interface allows to modify or to add information as well.

The RESTful interface exposes only insert and get statement for the logical model. This constraint makes the prototype simpler. In this thesis, it is assessed whether the concepts envisioned in the BEST project can be realized by using semantic technologies. Therefore, to only allow insert and get statements is sufficient for this prototype.

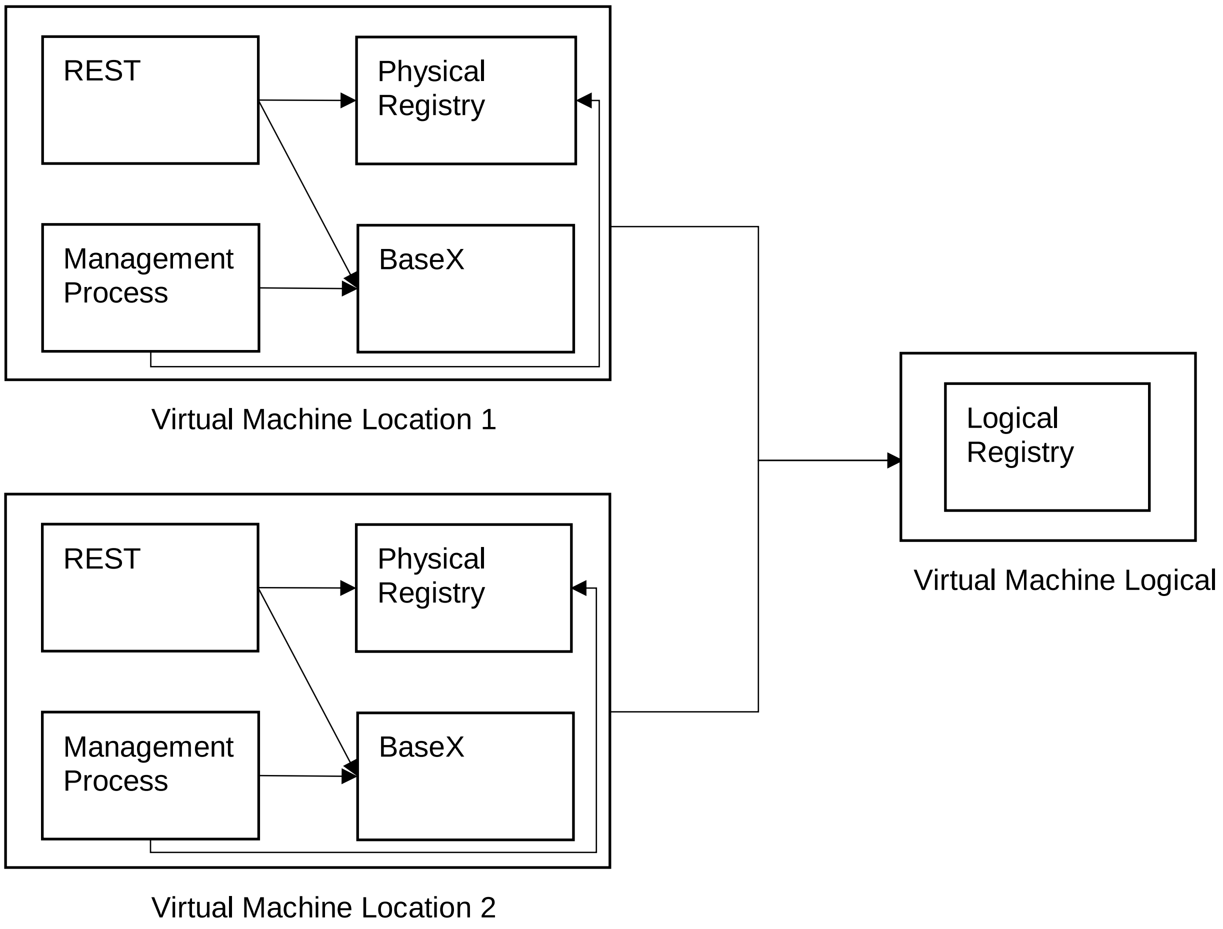


Figure 6: Architecture of the SCMS backend

For convenience, the SCMS provides an application (“App.java”) which starts multiple locations on one single machine. A physical registry, an instance of the backend, and an XML database are started for each location and one logical registry is started for all instances. In the SCMS, the amount of locations, the locations themselves, and the logical registry can be configured by using a configuration file named “config.properties”. This configuration file allows to configure a locations array that stores the information for each location and properties for the logical registry.

The configuration file consists of the following properties: The properties which can be configured for the logical registry are the host and the port of the logical registry: the logicalHost and the logicalPort. The logical registry is realized with a Fuseki server. Each location which should be started must be registered in the locations array. Each location consists of a RESTful interface, a management processes, a physical registry which is as well realized with a Fuseki server, and an BaseX server. For the configuration of the RESTful interface, the locations array contains the internal host and port where the backend is hosted (privateRestHost, privateRestPort) and the external host and port where it can be accessed from outside (restHost, restPort). For the BaseX server and the physical registry, the hosts and the ports can be configured. In addition, the BaseX server requires an arbitrary database name, username, and password. If one wants to know which other locations exist, one can query the locations array by using the RESTful interface but only the properties restHost and restPort are returned.

As the applications which interact with the backend of the SCMS might want to know with which location they are interacting with or which other locations exist as well, the backend provides functionalities to return this information. This information can be retrieved by using the get methods described in Table 2.

Table 2: Paths of the basic structure

REST Path	Method	Description
/config/locations	GET	This service returns the locations where the SCMS is supposed to be running. This method is designed to provide the applications with common information about other locations.
/config/location	GET	This service returns the current location.

3.2 Distribution and Replication

To increase the availability and the partition tolerance, the data are stored at multiple locations. The distribution and replication is divided into two different types: a logical and a physical distribution and replication. The information in the logical registry is available to all locations whereas each location has its own physical registry. The reason why there is a logical and a physical replication is that the information, which exists about a certain container, is separated from the actual data which are stored in a container. The logical model is globally available and can be accessed at every location. The logical model – stored in the logical registry – contains the information which exists about a container and the physical model – stored in the physical registry – contains the references to the data, e.g., to the NOTAMs and to the TAFs and to meta-information about it. The container's data are also stored at each location in a separate database. This means the content of different containers is stored at different locations. If a container is available at a certain location, it is allocated there.

The logical model contains information about the semantics of the content of a container, the provenance, and the structure of a container. For example, the logical model stores the name of a container, the site where it is allocated, the administrative and descriptive metadata, and the different versions which exist. The logical registry might be fully replicated. This means that multiple logical registries exist side by side which then must be synchronized with each other.

Each instance of the SCMS has a physical model that is stored in the physical registry. The physical model stores metadata about the allocated containers and data sets. The data set's content – the data items – is stored in an XML database. Not all data sets of all containers are stored at a specific location but only the data sets of the containers which are allocated at the specific location. For example, the physical model stores if it is a degenerated or regular data set or stores additional information about a specific data set. In the diagram in Figure 7 one can see the coherence between a container in the logical model and a container in the physical model.

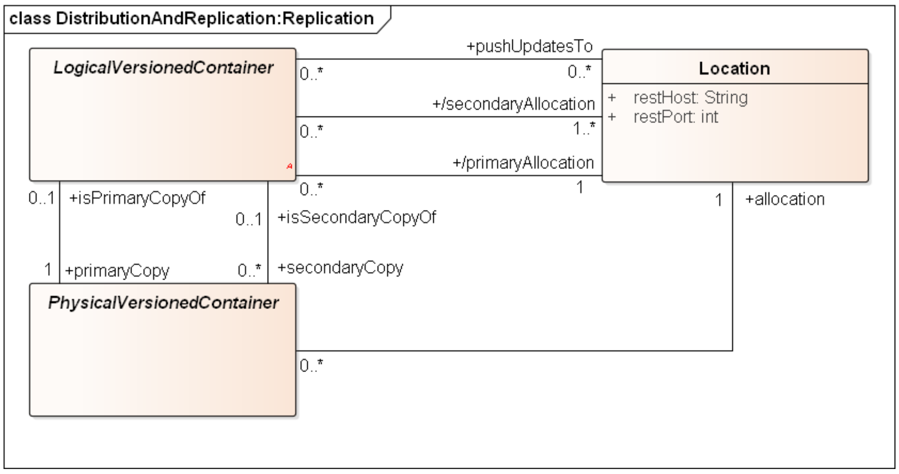


Figure 7: Distribution and replication

The advantage of this approach is that every location of the SCMS is aware of every available container. This is possible due to the fact that the logical model is stored in a single RDF graph in the logical model which is available for every SCMS instance. Therefore, a search for a container can be done with a certain information need. Another advantage of this approach is that it optimizes the storage usage. Not every container which exists logically must be allocated physically at every location and therefore the data sets do not have to be replicated.

The metadata of a container consist of administrative metadata and of a semantic label. An overview can be seen in Figure 8. The administrative metadata are further grouped into technical metadata and quality metadata. What technical metadata and what quality metadata are is explained in Section 1.2. Each administrative metadata, which can be referenced by a container, must first be added to the system. For example, the technical metadata “lastUpdatetime” must first be stored in the subsumption hierarchy as a subclass of technical metadata. Then in a next step one can associate the technical metadata “lastUpdatetime” with a container by using a concept that represents a literal. Such a concept represents a value, e.g., “2016-01-02”. Each concept is then stored as a subclass of Literal. One specific value, e.g., “2016-01-02” can be referenced more often.

In the logical model of the SCMS each container is stored with a semantic label. The semantic label contains the descriptive metadata of a container. The descriptive metadata can be grouped into three groups of facets: semantic, temporal, and spatial facets. A closer description of the different facet types be found in Section 1.2. Each facet must originate from an ontology. The ontologies where the facets are taken from are stored in the logical model as a named graph. In an ontology, the characteristics of facets are defined. For example, the facet “AircraftFacet” of the ontology “AeronauticalOntology” can have among others the following characteristics: “Aircraft”, “HeavyAircraft”, “AircraftWithHighWingSpan”, “Seaplane”, “LightAircraft”, and “Helicopter”. For example, one can assign the facet “AircraftFacet” by using one of the characteristics, e.g., “HeavyAircraft” to a container. How containers, facets, and concepts interrelate can be seen as well in Figure 8.

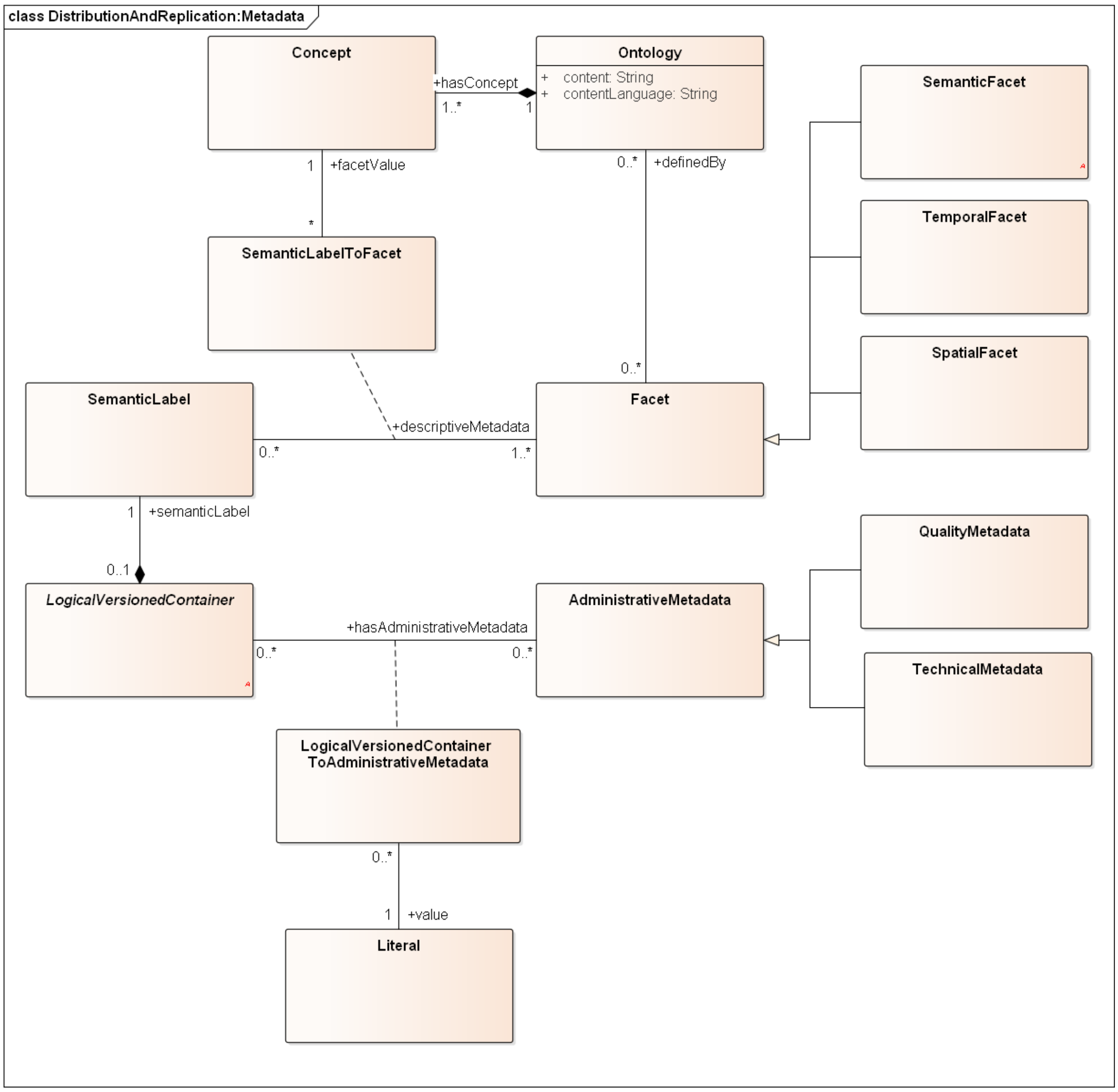


Figure 8: The container's metadata

The backend of the SCMS provides several operations to retrieve allocated containers, to allocate containers, to search for a container, and to add administrative and descriptive metadata. These operations are listed in Table 3.

Table 3: Paths for distribution and replication

REST Path	Method	Description
/container/allocate	GET	This service returns the containers that are allocated at a specific location.
/container/allocate/ {containerName}	GET POST	With this service, one can GET a container that is allocated at the location where the request is sent to. It will return every version of a container with its data sets. This means that it will return the information of a container stored in the

REST Path	Method	Description
		logical repository. The POST service will initiate a transfer of a container including its versions to the logical repository of the location where the request is sent to.
/metadata/administrativ	GET	This service returns all administrative metadata. It is a list of all registered metadata. For each administrative metadata, the name and type are returned.
/metadata/administrative /{administrativeName}	POST	This service registers administrative metadata data in the container management system. Therefore, the method needs the type of the administrative metadata and its name.
/metadata/ontology	GET	This service returns a list of ontologies. However, only the names of the OWL ontologies are returned.
/metadata/ontology /{ontologyName}	POST	This service adds an OWL ontology to the SCMS. For each ontology, a unique name must be specified, a file which contains the ontology must be given, and the representation must be set. Ontologies with the following representations can be added: CSV, JSON-LD, N3, NQ, N-Quads, NT, N-Triples, RDF/JSON, RDFNULL, RDFTHRIFT, RDF/XML, TriG, TriX, TSV, TTL, and Turtle.
/metadata/ontology /{ontologyName}/facet	GET	With this service, the possible facets of a registered ontology can be received. As the types of the facets are unknown, the facets are returned as Unknown Facets. The type is defined when a facet is registered in the SCMS.
/metadata/facet /{facetName}	POST	With this service, a facet can be added to the SCMS. A facet must be taken from an ontology and its type must be specified while adding. There are three different types: Semantic Facet, Spatial Facet, and Temporal Facet.
/metadata/facet	GET	This service returns the registered facets. For each facet the type, the name, and the ontology where it is taken from are returned.
/metadata/facet /{facetName}/concept	GET	This service returns a list of concepts for a specific facet. A concept represents a characteristic of a facet. This is required while specifying a facet and its characteristic for a container.
/container/search /{informationNeed}	GET	This method returns a list of containers. It is ensured that each container contains the needed information. The necessary information is specified in the information need.

3.3 Consistency Management

The consistency management comprises, e.g., the synchronization of physical containers and the versioning of containers. Each container has a primary allocation which contains the desired state of a container. The primary location is the first location where a container is allocated. If a container is allocated at another location it is called a secondary allocation. Each secondary allocation synchronizes itself with the primary allocation by either pulling the data from the primary location (PULL-Updates) or by receiving the changes from the primary allocation (PUSH-Updates). To receive PUSH-Updates a container must register itself at the primary allocation. It is stored in the logical model which locations receive PUSH-Updates. Whenever a container is registered for PUSH-Updates, the primary allocation issues a PUSH-Update whenever a data set is added to it.

The data of a container is represented as data sets. A data set contains one or more entities of a specified entity type, the data items. For example, a NOTAM container can have a data set which contains one or multiple NOTAMs. A data set can also contain metadata about its content. Only elementary containers can have data sets. If one wants to know the data sets of a composite container, one has to build the union of all data sets contained in the composite's elementary containers.

In Figure 9 one can see a container with two physical allocations and its versions and how they evolved over time. One of them is the primary allocation and the other is a secondary allocation. Both containers have already one version “V1” with the basic set “1” in the beginning. This is because every container must have at least a basic set in this prototype. A basic set represents the initial data which might be extended by delta sets. When a container is allocated at a secondary allocation all versions are transferd to that location as well. This is the reason why the container at the secondary allocation also has the basic set with the version “V1”. The coherence between the versions an the data sets can be seen in Figure 10.

If a data set is added to a container, it will always result in a new version. This versioning granularity was choosen for this prototype because it has benefits when it comes to testing the versioning algorithms. In the final BEST application, one might use different granularities for versioning. For example, one might create a new version every hour or after 100 megabytes of data was added to a container.

In Figure 9 the pimary container receives the “Delta Set 2”. This results in a new version “V2”. From the moment in which the “Delta Set 2” was added to the primary allocation, it is a regular set. A regular set contains data items that originate from a prefered source called primary source that is specified or dynamically determined for a logical container. A version which only contains regular sets is called a regular version. As “V1” is containg the sets 1 and 2 which are both regular sets, it is a regular version. When the secondary allocation is syncroniced with the primary allocation the version “V2” is transferred to the secondary allocation.

When now the primary allocation is not available due to disturbances, the secondary allocations will not be updated either. Therefore, delta sets can be added to secondary allocations as well to minimize the effects. These delta sets that are added to a secondary allocation might be degenerated sets. A degenerated set is a set which might not contain the desired content with the desired data quality. It contains data items that might originate from an alternative source called secondary source that is specified or dynamically determined for a logical container. To complete, a primaray allocation might have degenerated sets too. If a version contains a degenerated set it is a degenerated version. In Figure 9 one can see the degenerated version “V3-DEG” with the regular sets “1” and “2” and the degenerated set “3-DEG”. However, the aim is always to have the data available at every location with the predefined quality. Therefore, as soon as the primary allocation is available and a new data set is added to it, the version “V3” is transferred to the secondary allocation. The mechanisms are put in place as explained before.

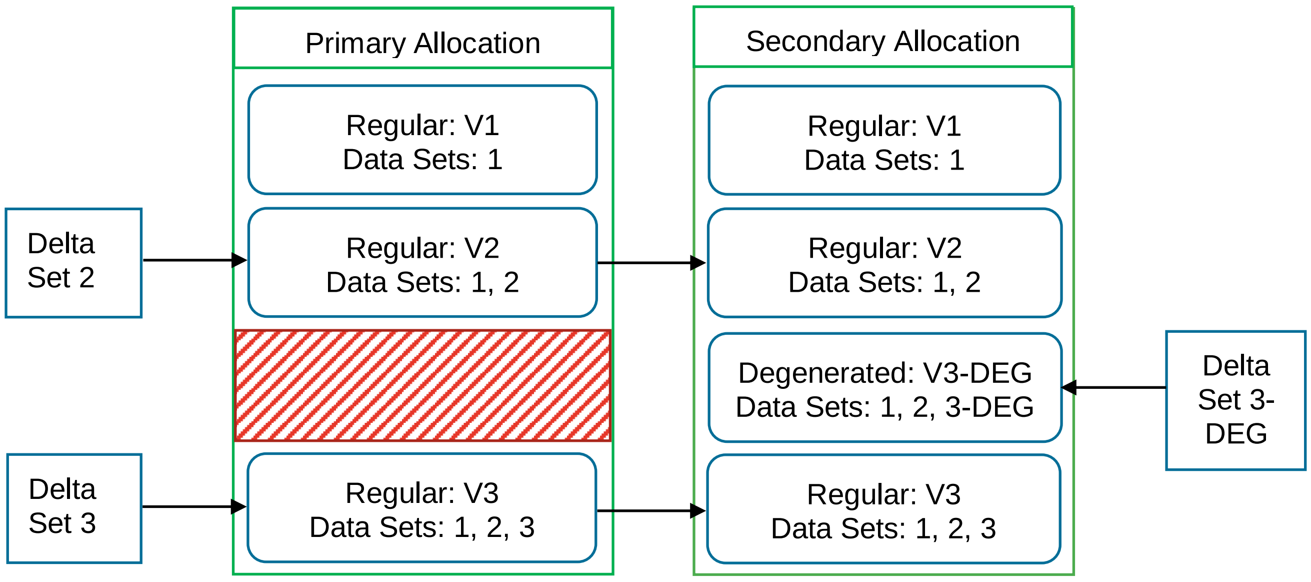


Figure 9: Versioning of containers with degenerated versions and data sets

Whenever a new regular version arrives at a secondary allocation via PULL or PUSH updates, the desired state is inherited and the degenerated version is dropped. However, the consistency management registers every version which ever existed on a certain location. Therefore, one has to query a specific location to see the degenerated versions which existed there. Degenerated versions are not deployed on to other allocations.

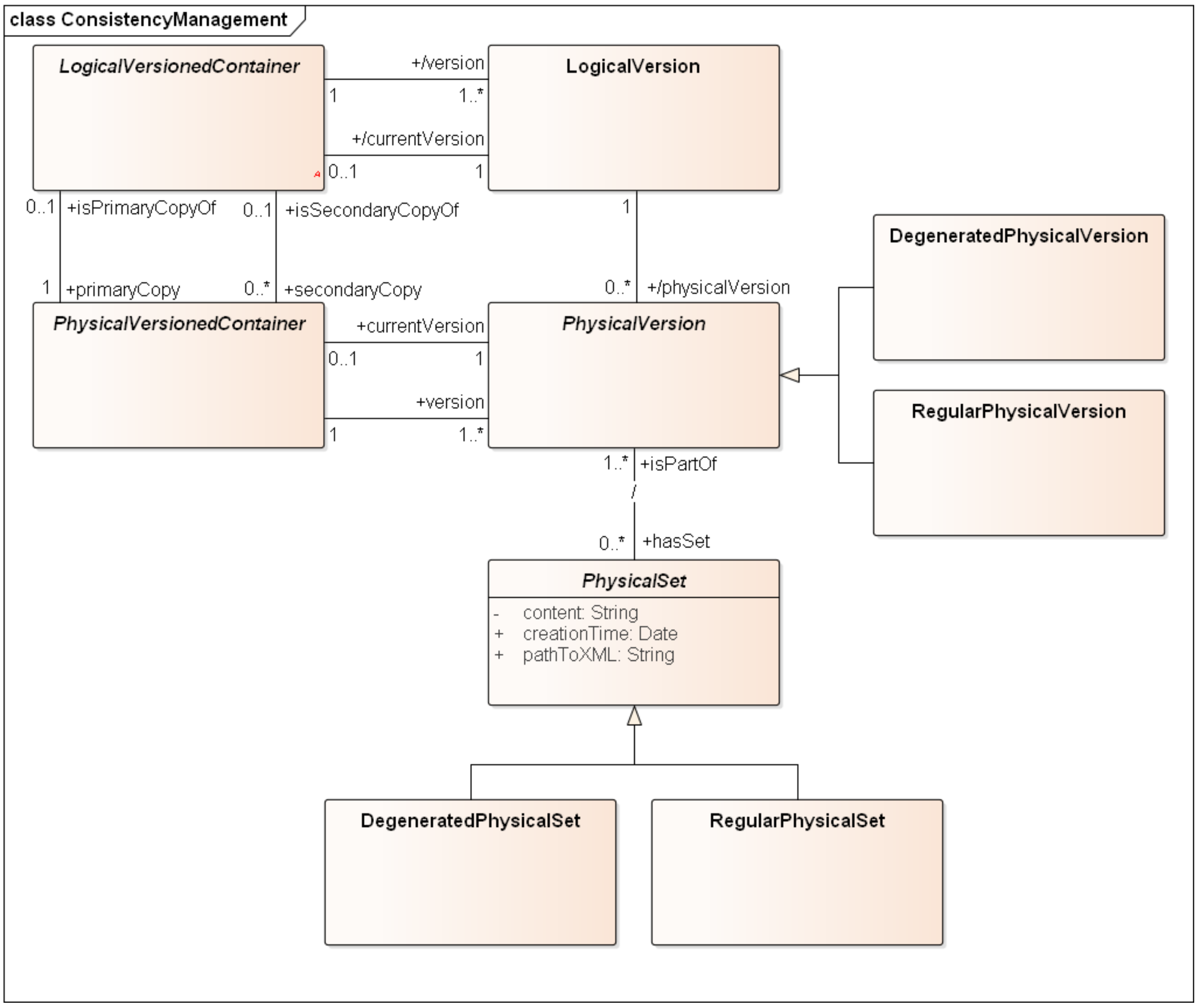


Figure 10: Consistency Management

The update manager is in charge of handling the PULL-Updates and keeping the different containers synchronized. This means that whenever a new version at the primary allocation is created, this version is transferred to the secondary allocations. The synchronization of the containers is realized with RESTful services and an update manager at every location. This update manager is in charge of pulling the new data from the primary allocation of a container to the secondary allocation of a container that is allocated at the update manager's location. In Table 4, one can see the different service calls provided by the backend of the SCMS to inspect versions, to allocate containers, to view data sets, and to inspect whether a container is allocated or not.

Table 4: Paths for consistency management

REST Path	Method	Description
/container/{containerName} /version	GET	This service returns the versions which are stored in the logical scheme of the SCMS.
/container/{containerName} /version/{versionName}	GET	This service returns a specific version which is stored in the physical scheme of the SCMS. This means that the data sets which are stored in a specific version are

		returned as well.
/container/allocate /{containerName}/physicalset	GET	This returns a list with the container's basic set and delta sets. The basic set will be the first element in the returned list.
/container/allocate /{containerName}/physicalset /{physicalSetName}	GET, POST	The GET service returns the content of a data set. This service is designed to inspect the content of a specific data set at specific instances of the SCMS. Therefore, it will interact with the logical repository. The POST service adds a new data set to a container. However, this will add a data set at a specific location. It initiates the PULL synchronization process if the location where the data set was added to is a primary copy.
/container/allocate /{containerName}/version /{versionName}	POST	This service is designed for internal use only to exchange data between the different SCMSs. It is used to deploy a specific version at a specific location.
/container/notallocate	GET	This service returns the not allocated containers. The not allocated containers are the containers which exist in the logical repository, however do not have a physical copy at this specific location and thus do not exist in the physical repository.
/container/allowedtoallocate	GET	This service returns the containers which are allowed to be allocated. These are the topmost containers of a composite and the elementary containers. The reason why only the topmost container of a composite container can be allocated is that a composite container and its components must be allocated at the same time.
/container/{containerName} /rootcontainer	GET	This service returns the root container for a container. The root container is the topmost container in the container hierarchy.

3.4 Provenance and Composition

The basic structure is an elementary container which stores the actual data of the SCMS. There can be an annotated elementary container and an entity elementary container. For every elementary container either annotated or not annotated, an entity type must be stated. The entity type defines the content which is stored in the container. In addition, every annotated elementary container has an annotation type. A container can be assembled to composite containers. There are two types of composite containers. A container which only contains containers of the same data type is called homogenous composite container. If a composite contains containers of different data types, it is called a heterogeneous composite container. This relationship can be seen in Figure 11. Figure 11 focuses purely on the logical part of the model. However, the logical, the elementary, the composite, the homogenous composite, the heterogeneous composite, the entity elementary, and the annotated elementary container exist in the physical model as well.

Every container can have sources where it gets its data from. The data are transferred from the sources to the container by a service. In general, the container usually retrieves its data from the primary source. In special cases where the primary source is not available, the container can retrieve its data from a secondary source. However, to implement this feature a service must be implemented as well, which transfers the data from a source to a container. This has not been done in the development of the prototype.

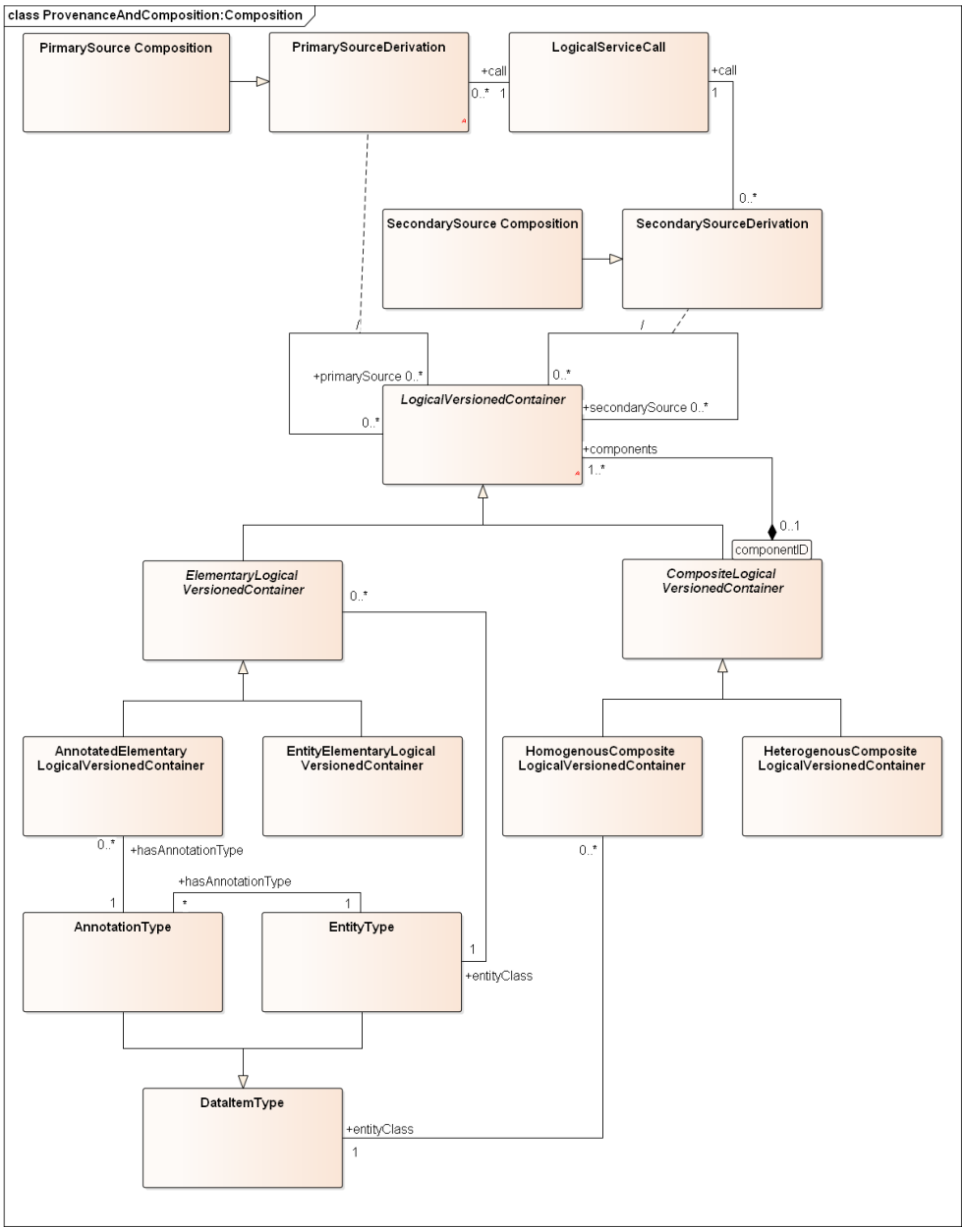

class ProvenanceAndComposition:Composition
PirmarySource Composition
PrimarySourceDerivation
LogicalServiceCall
+call
0..*
1
+call
1
0..*
SecondarySource Composition
SecondarySourceDerivation
+primarySource 0..*
0..*
0..*
LogicalVersionedContainer
+secondarySource 0..*
+components
1..*
0..1
componentID
ElementaryLogical VersionedContainer
0..*
CompositeLogical VersionedContainer
AnnotatedElementary LogicalVersionedContainer
EntityElementaryLogical VersionedContainer
HomogenousComposite LogicalVersionedContainer
HeterogenousComposite LogicalVersionedContainer
0..*
+hasAnnotationType
0..*
+hasAnnotationType
1
*
1
AnnotationType
EntityType
1
+entityClass
DataItemType
+entityClass
1

Figure 11: Composition

The provenance documents, e.g., which service added which data set to a container. In Figure 12 one can see the relationship between a service and the data set which is added to a container. A service is provided by a service provider. This can be an organization or a person which offers a web service in the SWIM environment. Such a service provider can offer multiple services. A service provider might run a service at multiple physical servers. The physical servers are registered as well. A service can add data to multiple containers and can be executed for each container with different parameters. Therefore, a service which is executed with well-defined parameters on a container is called a service call. A service call – which represents a possible and already parameterized invocation of a service – can again be executed multiple times for a container. For example, a copy service can copy the data multiple times from one container to another. Every execution is called a service occurrence. For service calls and service call occurrences one can differentiate between the logical model and the physical model. In the logical model a service call states that a service can be executed for a container whereas in the physical model it states that a service interacted with this location one or more times. A service occurrence in the logical model refers to the occurrences in the primary allocation whereas in the physical model a service occurrence can be associated with the data set it added. A physical occurrence only exists when the data set was actually added at this location. The physical service, the logical service, and the service provider are concepts of the logical model. As one can see in Figure 12 the service and service calls can be composed to composites as it can be done with containers.

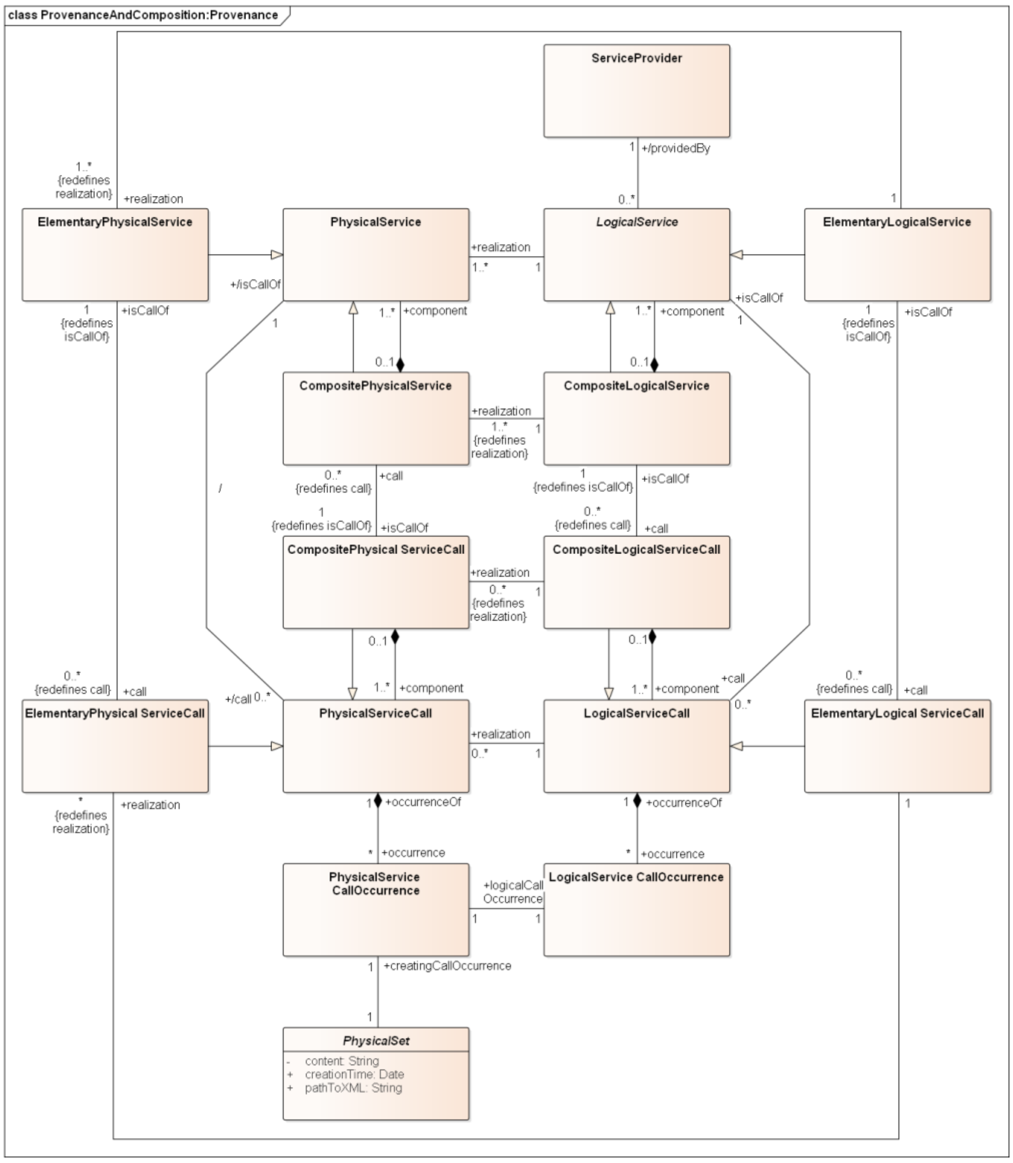


Figure 12: Provenance

In Table 5, one can see the paths that are offered by an application to add containers and services to the REST interface. As mentioned before, the application is limited to the registration of service providers and services. However, the RDF graph of the logical model and the RDF graph of the physical model provide classes and properties for storing the provenance.

Table 5:Paths for composition and provenance

REST Path	Method	Description
/container	GET	This service returns a list of containers from the logical model. As every container must be registered in the logical model, this service gives an overview about all containers in the SCMS.
/container /{containerName}	GET, POST	The GET service returns the information about a specific container from the logical model. The container which is returned has the following metadata: the container type, the name, the annotation type, the primary allocation, the secondary allocations, pushes updates to, the semantic label with the facets, the administrative metadata, the components, the versions, and the current version. However, the transferred data for a container can differ according to the container type. For example, an entity elementary container does not contain an annotation type. The POST service allows to add a specific container to the SCMS. For every container which is added, a logical and a physical container must be specified. The logical container contains the metadata whereas the physical container contains the data sets.
/service	GET	This service gives an overview about all registered services in the logical model.
/service /{serviceName}	GET, POST	The GET service returns a detailed information about a service. For a service, the type, the service name, the service provider, the parameters, and its realizations are returned. The POST service adds a fully specified service to the SCMS.

3.5 Implementation Details

In this section, the technologies and tools which are used are explained and it is justified why a certain technology or tool is used. Furthermore, the structure of the backend and the major classes are explained.

3.5.1 Technologies and Tools

In order to realize the backend, Java was chosen as programming language. Java has a high cross-platform compatibility and it enjoys high popularity [34]. Therefore, it is one of the most popular programming languages for developing web applications. It can be run on a development client as well as on a server. Furthermore, it has a broad set of functionalities [34]. When it comes to performance, Java has a throwback. It is faster than JavaScript in specific Node.js but slower than Go. Though performance is not relevant for this prototype this can be neglected.

For developing RESTful applications in Java, several frameworks are available. Those frameworks are concrete implementations of JAX-RS, the Java API for RESTful web services [35]. In 2013, nine frameworks of these kind existed. From these frameworks, Jersey was chosen for realizing the RESTful interface. Jersey is Oracle's implementation of JAX-RS. In this survey [35], Jersey lacks a transaction support, does not support other protocols beside HTTP, and does not have extended resource types. However, Jersey scores well in the categories documentation, support for HTTP, and handling of big files. In the remaining categories, e.g., suitable for server application, modeling of REST API, and caching the framework scores average results [35].

Because Jersey is primarily designed for servlets, one needs additional software to create a standalone application. In this prototype, the Grizzly HTTP Server is used as a host [35]. Grizzly's goal is to allow developers to build scalable and robust servers [36]. In the background, it uses the Java New I/O API (NIO) which has an advanced thread management. Furthermore, Grizzly offers extended framework components, e.g., Web Frameworks (HTTP/S) and WebSockets.

In this prototype, the data, which is transferred from an application to the RESTful service via HTTP, must be organized in JSON format [37]. Jersey supports integrations for the following formats: JSON, XML, and Multipart. To enable the support for one of these formats, one has to use an external module. The used module must be set in the Jersey configuration. The JSON format was chosen because it is faster and uses less resources than XML [38]. Multipart is a format which allows multiple formats to be transferred in one request. Jersey comes with an integration for the following modules that enable JSON: MOXy, Java API for JSON Processing, Jackson, and Jettison [37]. MOXy and Jackson represent the easiest ways to convert JSON to Java objects [37]. Therefore, in the prototype Jackson is used to covert Java objects to JSON objects and vice versa.

The data sets which are stored by the SCMS are in XML format and therefore an XML database is used to store the content of the data sets. As XML Database BaseX was chosen. In this report [39], the XML databases eXist, BaseX, Senda, and Oracle Berkeley DB are compared regarding their performance. BaseX is the fastest XML database. However, the conventional MySQL database is faster than the XML databases in executing queries. Furthermore, BaseX is the 4th popular XML database out of 8 [40]. However, BaseX is the best pure XML database in this ranking [40].

For accessing the XML database the XQuery Java API (XQJ) is used [41]. XQJ enables Java applications to connect to various XML databases. XQJ defines a set of interfaces and classes. With those it empowers a Java application to submit XQuery expressions and to consume their results. XQJ is based on the model of Java Database Connectivity (JDBC). In comparison, JDBC enables Java applications to connect to various SQL databases [41].

Because in the BEST project semantic technologies are used in order to represent containers the SCMS also needs an RDF Triple Store database where the data can be stored. For the prototype, the Apache Jena Framework is chosen. According to Voigt et al. [42], the Apache Jena Framework, OWLIM Lite, BigData RDF Database, and OpenLink Virtuoso are the only triple stores that are available for free, can handle more than a 100 million triples, can reason over data, support the SPARQL Protocol And RDF Query Language (SPARQL) 1.1, and are built for Java. According to Voigt et al. [42], it depends on the project setup which one should be chosen. For the prototype, the Apache Jena Framework was chosen. It comes with a server called Fuseki.

In order to create the UML diagrams Enterprise Architect is used. Enterprise Architect [43] comes with a wide range of functionalities, e.g., class diagrams and object diagrams. The Fuseki server requires an initial RDF data structure or model which describes the initial set of triples. This initial RDF graph is exported from the class diagram from Enterprise Architect. With Enterprise Architect, one can define which classes, which attributes, and which associations are part of a schema. Two schemas are created from the class diagram which are then exported to

OWL2, one is the logical model for the logical registries and one is the physical model for the physical registries.

The RESTful interface as well as the Java model of the application are documented with Swagger [44]. Swagger is an open source tool for designing, building, documenting, testing, and standardizing RESTful interfaces. It offers open source tools for code generation, editing, and inspection. Swagger is used because it currently has the largest community, power, and tools over its competitors RAML and API Blueprint [45]. Swagger requires the documentation in a certain format so that it can be interpreted by the tools Swagger provides. This format is specified in the OpenAPI specification, formally known as Swagger specification [46]. OpenAPI allows one to define a title, a description, and a version for the RESTful interface. The servers hosting the RESTful interface can be documented as well. However, the core part of the specification deals with the documentation of the different RESTful paths which exist in the application. For every RESTful path the parameters, the request body, and the responses can be documented. Furthermore, the format of the service input and output can be described [46]. As the REST interface uses the Java model as the input and output data structures, the Java model is documented with Swagger as well. For the documentation of the RESTful interface and its model, refer to Appendix C.

A big advantage of documenting the API with the OpenAPI specification is that the documentation can be generated from Java classes by using a maven plugin. This plugin (com.github.kongchen.swagger-maven-plugin) is able to generate a specification out of a RESTful application from the JAX-RS annotations and extended Swagger annotations.

3.5.2 Structure

The SCMS is hierarchically structured and can be grouped into four big parts: Model and common classes, Data Access Objects (DAO), REST interface, and the application and its startables. These parts are clearly separated. The Java model represents the domain of the SCMS. For each RDF class of the logical and physical model exists a counterpart in Java [6]. The DAO load the data from the logical and the physical model of the database and then create Java objects from it. The so created objects are a reflection of the content in the database. The DAO can persist the data contained in the Java objects as well. As already mentioned, the DAO provide the capabilities to store the Java model in the database or to reload it from there. The REST interface interacts with the DAO. For example, it uses the DAO to load a specific container from the database or to store a newly created container. The application and its startables provide the functionality to boot the backend of the SCMS. Each of those parts are described below. One can see an overview in Figure 13.

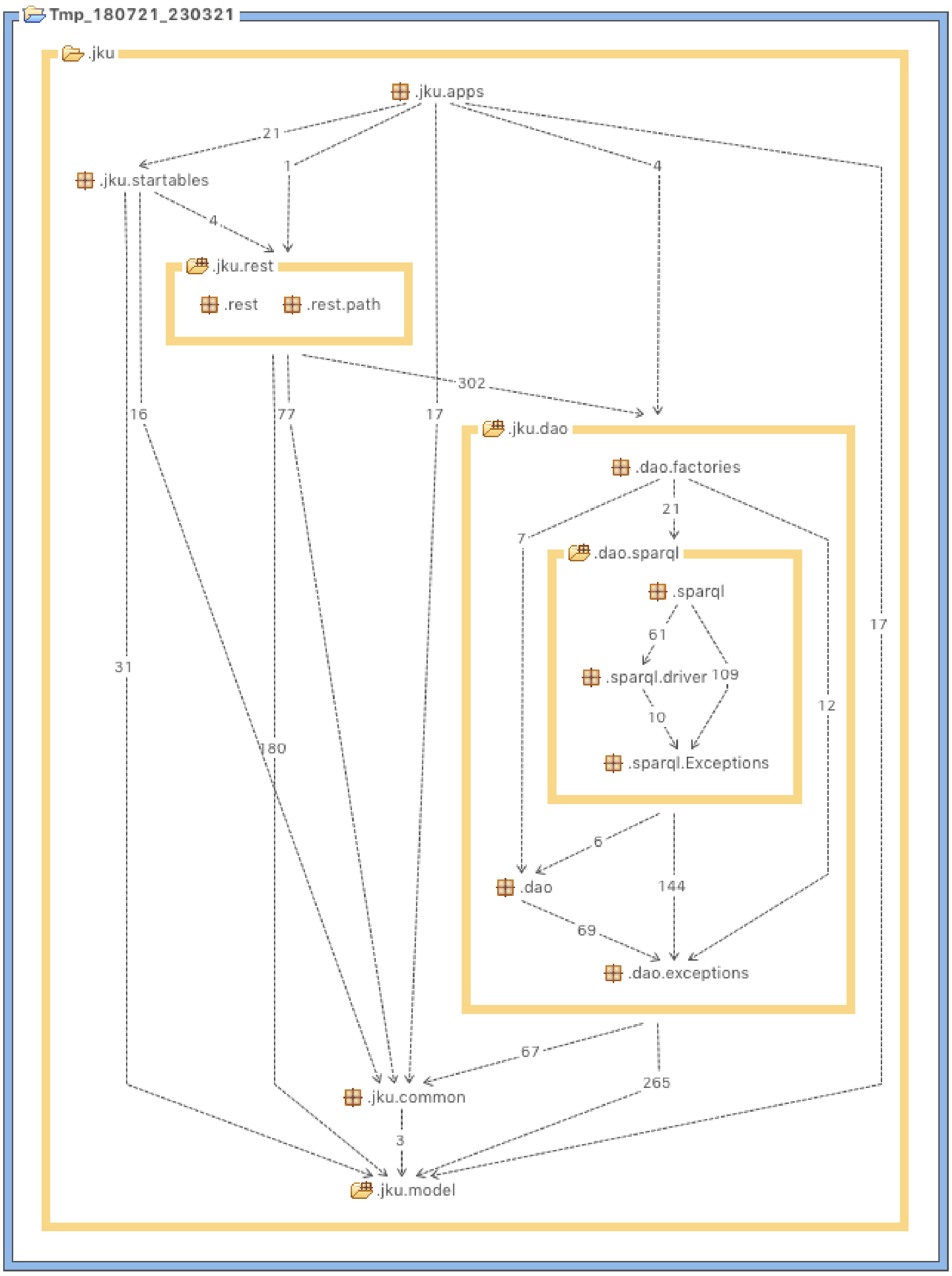


Figure 13: Package structure of the backend

3.5.2.1 Java Model and Common Classes

The Java model represents the data structures which are known by SCMS. One can see the Java classes in Figure 7, in Figure 8, in Figure 10, in Figure 11, and in Figure 12. Those classes are used by all other parts of the implementation because they represent the domain model. The domain model was developed in Enterprise Architect and then exported as an ontology which is used by the logical and physical registry. In Java, however the model is reimplemented by using Java classes. The model of the SCMS is further separated into four paths:

org.jku.model.compositon, org.jku.model.consistancymanagement, org.jku.model.distand-replication, and org.jku.model.provenance. For example, the namespace org.jku.model.composition contains the following classes: DataItemType, EntityType, AnnotationType, LogicalVersionedContainer, ElementaryLogicalVersionedContainer, CompositeLogicalVersionedContainer, AnnotatedElementaryLogicalVersionedContainer, EntityElementaryLogicalVersionedContainer, HeterogenousCompositeLogicalVersioned-Container, and HomogenousCompositeLogicalVersionedContainer. The Java model is documented like the REST services as well with Swagger because the REST services use it as input and output structures. Therefore, a detailed documentation of the model can be found in Appendix C. To generate this specification, the Java classes and properties are annotated with Swagger annotations.

The common package contains utility functions. It includes functions which help loading the properties from the property files or which help sending an HTTP requests. These methods can be used by all packages except the model package. The utility methods know only the model classes.

3.5.2.2 Data Access Objects

The DAO are responsible for the communication with the database [47]. They encapsulate the logic which is necessary to get, update, delete, or add resources in a database. In the prototype, the DAO offer operations on the following resources: location, logical service, logical versioned container, physical service, and physical versioned container.

According to the DAO pattern, one needs to implement an interface for each resource. This interface must offer the functionalities which can be executed on a specific resource. For example, if a resource can be added and updated but not deleted and retrieved, the interface must provide two functions: one add and one update function.

In this prototype, for each of these interfaces there exists only one concrete implementation namely the implementation for SPARQL and XQuery. This means that each request is transferred to a SPARQL query except for those requests, which need access to the data sets' content. Those requests consist of both, an XQuery and a SPARQL statement. The advantage of this approach is that only queries are generated. Therefore, the databases could be exchanged in the background as long as they support XQuery and SPARQL. The functionality which sends the query to an RDF database is encapsulated in a class called RdfDbDriver and the functionality which sends a query to an XML database is encapsulated in a class called XmlDbDriver. The RdfDbDriver uses Apache Jena and the XmlDbDriver uses XQJ [41] to interact with either the Fuseki server or BaseX. This decoupling provides further flexibility.

If one requests to insert a physical container with one data set in the databases, then a SPARQL query for the physical registry and an XQuery for inserting the content of the data set for the BaseX server is generated. The queries are then executed with either the RdfDbDriver or the XmlDbDriver. Those are located in the package org.model.dao.sparql.driver.

To enhance the query generation the "SparqlObjectHelper.java" is created. This class is able to convert Java objects to SPARQL quires which are able to update, load, and insert objects in the RDF database. To insert an object one needs to pass the object itself to the "SparqlObjectHelper.java". From then, a SPARQL insert statement is generated where the object itself and its properties are inserted into the RDF graph. The object itself is stored as an RDF class. Java uses reflection to retrieve the properties of the object. If a primitive type is encountered, the value of the property is stored as literal. In Java the primitive types are int, double, float, long, byte, char, short, and boolean [48]. As each primitive type has its wrapper class, those classes are treated as primitive classes as well. In addition to the primitive types, strings and dates are also stored as literals. If a property of a complex type is encountered, it is stored as a class in the subsumption hierarchy. This results in a recursion. According to the OWL standard, an instance of a class would be considered as an individual. However, because a subsumption hierarchy must be generated [5], an individual of a class is added as a subclass to the object hierarchy.

The “SparqlObjectHelper.java” can reverse that action. It can generate a Java object of a RDF class in a subsumption hierarchy. Furthermore, it can set the properties of the Java object by reading the values of the properties from the database. However, the class type of the object and its unique identifier must be known beforehand. For example, if one wants to load a container from the database one needs to specify the type, e.g., ElementaryLogicalVersionedContainer and its identifier which is the container’s name.

The user only uses the interfaces that are provided by the DAO to interact with the database. An object factory hides the concrete DAO implementations from the user. For example, if at one time in future, the data should not be persisted in a database which operates with SPARQL and XQuery, but in a relation database only another implementation of the DAO interfaces must be added. The factory can now switch between those two implementations seamlessly. Only the factory knows the concrete implementation.

3.5.2.3 REST Interface

The REST interface represents the part where external applications access the SCMS. It is also the REST interface which is used by the frontend. The REST interface offers the following paths:

- config (access to the location configuration)
- container (provides functionalities which contain actions on a container)
- metadata (the metadata which are provided about the container at a specific location)
- service (the services which are registered in the semantic SCMS)

Each path represents a resource which can be accessed from outside. Each path offers get, put, post, delete, head, and option operations as well as an access to resources. The resources can have sub-collections as well. For example, containers store data sets or versions as sub-collections.

The paths of the REST interface are structured differently, in comparison to the structure of the BEST project. The path “container” contains all operations which can be executed on a container. This can be operations of the distribution and replication (container/allocation), of the provenance and composition, and of the consistency management perspective. The path “metadata” contains solely operations for the distribution and replication perspective. One can add ontologies and facets by using the services this path provides. Finally, with the path “service” one can register service providers which can be used to replicate containers. One can find the precise definition of the interface in Appendix C.

The REST interface provides methods which are available to the public and can be called by using HTTP methods. In Listing 3, one can see an implementation of an HTTP-GET method of the path container. The GET service in the listing must be called with the path parameter container name (line 14). Next, the required container is loaded (line 17). Internally, this method uses the DAO to retrieve the container. Finally, the container is converted to a JSON object (line 18) and is returned (line 19).

In order to create a service, one has to annotate the method with either @GET, @POST, @PUT, @DELTE, @HEAD, and @OPTIONS [49]. This means that the method responds to the given HTTP request. Furthermore, one has to define the path of the resource with the @PATH annotation. Whenever a HTTP request is done, the path-method combination is recovered and the method is executed. If the path contains a path parameter, e.g., /container/{containerName} (name written in brackets), one can specify with an annotation @PathParam in the method signature where the parameter will be handed over. With the annotations @Consumes and @Produces one can specify the format of the content which can be consumed or is produced by the service [49].

```
/** * Gets all LogicalVersionedContainers.
* @return This service returns the logical container specified by the container name.
* @throws IOException */
@GET
@Path("{containerName}")
@ApiOperation(code = 200, produces = "application/json", value = "Returns a specific
logical versioned container", notes = "This service returns the logical container
specified by the container name. This service loads all properties of a logical
container as well.", response = LogicalVersionedContainer.class)
@ApiResponses(value = {
   @ApiResponse(code=400, message="There was an error in the request."),
   @ApiResponse(code=404, message=" The logical container was not found by the given
container name."),
   @ApiResponse(code=500, message="An unknown error occurred.")
})
@Produces(MediaType.APPLICATION_JSON)
public Response getLogicalVersionedContainerPath(@ApiParam(value = "The name of the
logical container which should be returned.", required = true) @PathParam("containerNam
e") String containerName) {
    String containerNamePrepared = Utils.unprepareString(containerName);
    try {
      LogicalVersionedContainer container = getLogicalVersionedContainer(containerNameP
repared);
      String returnJson = Utils.getDefaultObjectMapper().writeValueAsString(container);
      return Response.ok(returnJson, MediaType.APPLICATION_JSON).build();
    }
    catch (BadRequestError e) {
      Log.error(getClass(), "", e);
      return Response.status(Response.Status.BAD_REQUEST).entity(Utils.convertStringToE
rrorJson(e.getMessage())).build();
    }
    catch (JsonProcessingException | InternalServerError e) {
      Log.error(getClass(), "", e);
      return Response.status(Response.Status.INTERNAL_SERVER_ERROR).entity(Utils.conver
tStringToErrorJson("An unknown error occurred.")).build();
    }
    catch (NotFoundError e) {
        Log.error(getClass(), "", e);
        return Response.status(Response.Status.NOT_FOUND).entity(Utils.convertStringToE
rrorJson("The logical container was not found by the given container name.")).build();
  }
}
```

Listing 3: RESTful service implemented with Jersey

For documentation purposes, each service is annotated with @ApiOperation and @ApiResponses and each parameter with @ApiParam. @ApiOperation gives an overview about the service, e.g., a value (short description or name) and a note (long description). The @ApiResponses describes which results can be expected from the service and the @ApiParam annotation describes the parameters.

If one requires a Java object as parameter which is not a path parameter, one can simply define it in the method signature. It will be transferred in the body of the HTML request (only one object can be specified for a service). The caller of the service must send the object in JSON format to the interface, because Jersey can map JSON string into Java classes by using the Jackson Object Mapper. Therefore, the developer does not need to take care about the conversion between JSON and Java objects.

3.5.2.4 Application and its Startables

The package “startables” provides the classes and the methods to boot a BaseX server (“BaseXBootable.java”), a Fuseki server (“FusekiBootable.java”), the REST interface by using a Grizzly server (“JersyGrizzlyBootable.java”), and a pull updater (“PullUpdaterBootable.java”).

This pull updater is in charge of synchronizing the secondary allocated container at the location where the pull updater is executed with its primary allocation. The pull updater considers only secondary allocated containers which are not registered for PUSH-Updates. It periodically compares the versions of the primary allocation with the versions of the secondary allocation. It is to be noticed that the primary allocated container represents the desired state which should be mirrored to the secondary allocations. If a difference is encountered, the pull updater transfers the newest version from the primary allocation and adds it to the secondary allocation.

The backend provides the following executable applications: the SCMSApp, the SwaggerApp, and two test applications. The SCMS app is able to start multiple instances of the SCMS backend at one server. The number of instances is configured in a configuration file. For each instance of the SCMS the app starts one physical registry which stores the physical model, one BaseX server, one Grizzly server, and one PULL-Updater. Those applications represent a backend at one location as described in Section 3.1. Furthermore, the SCMSApp starts one logical registry. In this prototype, there exists only one logical registry, no matter how many instances are booted.

4. Semantic Container Management System: Frontend

The frontend is the graphical interface of the SCMS which runs in a browser. The frontend is developed for two major reasons. With the help of the frontend, the users can familiarize themselves with the SCMS. They can execute all REST services the backend provides without prior knowledge about the backend. For example, they can query the containers, act as a service and add data to containers, or deploy the containers at different locations. The second reason why a frontend was developed is because with a working frontend the developers can test the backend and can assess if the backend works as expected. The aim of the frontend is that with it the developers and the users can execute every REST service the backend provides.

First, the main operations which are provided by the frontend are explained. They are as well separated into distribution and replication, consistency management, and provenance and composition concerns. This section ends with describing the general structure of the frontend and the related implementation aspects and decisions.

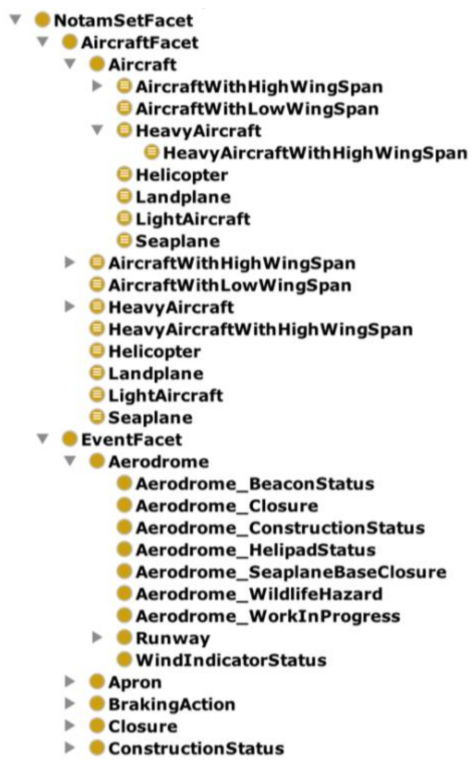
4.1 Distribution and Replication

The distribution and replication management is in charge of replicating the content, especially the containers' data to different locations. To transfer a container to a different location is part of the distribution and replication management. This process is called allocation. Every container is registered in the logical model. For example, each container is registered in the logical registry and the locations where the container is allocated are registered as well. However, the data are actually stored in the physical model and in the XML database. During an allocation, the logical and the physical model have to be adapted. For more details see Section 3.2.

In this section, it is explained how to, e.g., add metadata to a container, to register a new container, and to search for a container by using the interface the fronted provides. Each action is explained step by step and for each operation a figure is provided. The example data shown in the figures is taken from the example stated in Section 1.1.

4.1.1 Register an Ontology

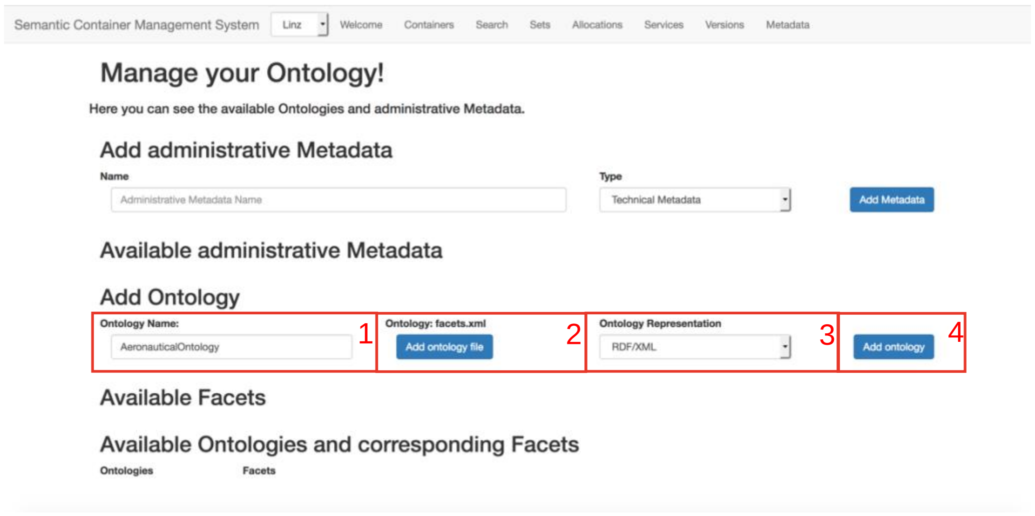
In the frontend of the SCMS an ontology can be added under the menu entry "Metadata". An ontology can contain multiple facets with the corresponding concepts. One can see in Figure 14 an ontology with two facets the "AircraftFacet" and the "EventFacet" with its corresponding concepts. This ontology states an example of how an ontology for metadata management in the aeronautical domain could look like. It was developed by Sumereder [50].



```
▼ NotamSetFacet
  ▼ AircraftFacet
    ▼ Aircraft
      ▶ AircraftWithHighWingSpan
        AircraftWithLowWingSpan
      ▼ HeavyAircraft
          HeavyAircraftWithHighWingSpan
        Helicopter
        Landplane
        LightAircraft
        Seaplane
    ▶ AircraftWithHighWingSpan
      AircraftWithLowWingSpan
    ▶ HeavyAircraft
      HeavyAircraftWithHighWingSpan
      Helicopter
      Landplane
      LightAircraft
      Seaplane
  ▼ EventFacet
    ▼ Aerodrome
        Aerodrome_BeaconStatus
        Aerodrome_Closure
        Aerodrome_ConstructionStatus
        Aerodrome_HelipadStatus
        Aerodrome_SeaplaneBaseClosure
        Aerodrome_WildlifeHazard
        Aerodrome_WorkInProgress
      ▶ Runway
        WindIndicatorStatus
    ▶ Apron
    ▶ BrakingAction
    ▶ Closure
    ▶ ConstructionStatus
```

Figure 14: Ontology with facets and concepts

This ontology can be added by using the function “Add Ontology” in the menu “Metadata”. One can see in Figure 15 how an ontology can be added. First the ontology name must be chosen (1), next an ontology file must be selected (2), then the representation must be selected (3), and finally the ontology can be added to the system (4). The ontology then will be available for all locations because it is stored in the logical model and not in the physical model of the selected location.



Manage your Ontology!

Here you can see the available Ontologies and administrative Metadata.

Add administrative Metadata

Name: Administrative Metadata Name

Type: Technical Metadata

Available administrative Metadata

Add Ontology

Ontology Name: AeronauticalOntology

Ontology: facets.xml

Ontology Representation: RDF/XML

Available Facets

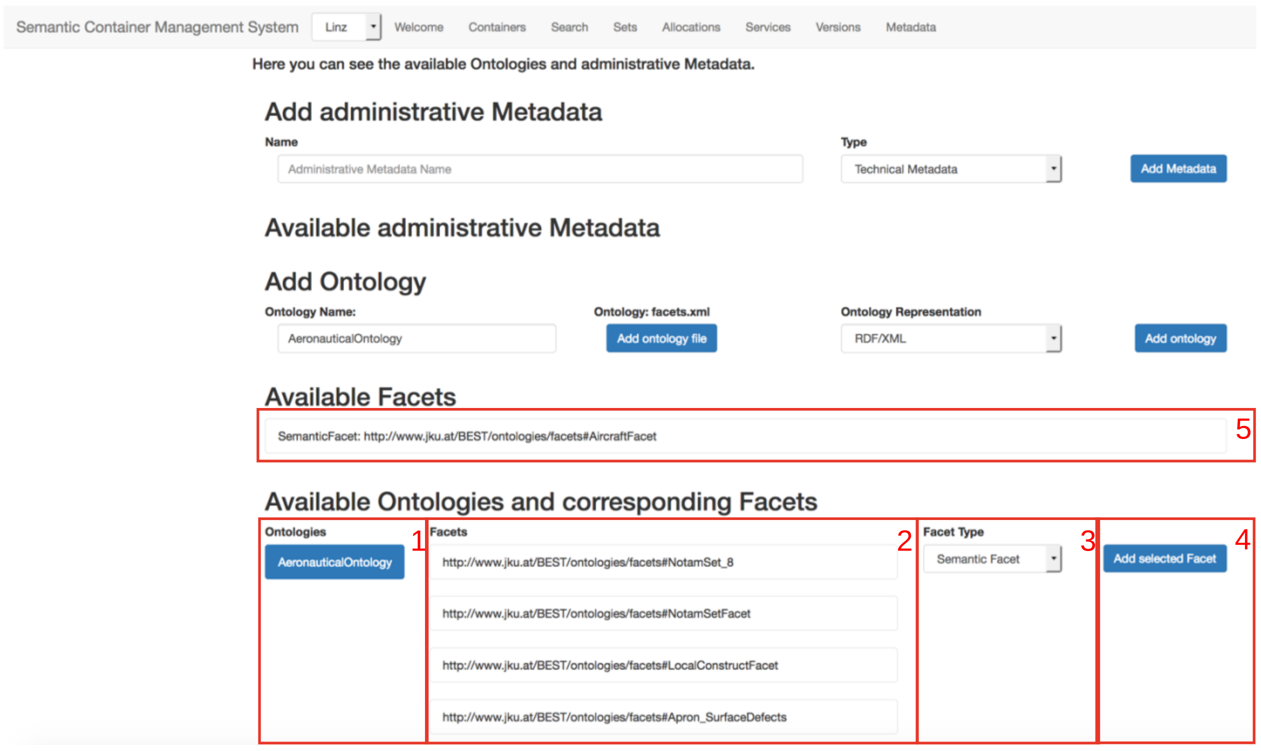
Available Ontologies and corresponding Facets

Ontologies	Facets

Figure 15: Add an ontology

4.1.2 Register a Facet

A facet is a superclass of an ontology which has different concepts (subclasses). One can see an ontology with facets in Figure 14. If one wants to add a facet, e.g., the "AircraftFacet" to the logical model of the SCMS one needs to select the ontology first (1) where one wants to take the facet from. Next, one has to select the facet itself which one wants to add (2). Furthermore, the type of the facet must be specified (3). One can choose between "Semantic Facet", "Spatial Facet", and "Temporal Facet". Finally, the facet can be added to the SCMS with a click on the button "Add selected Facet" (4). The facets are also stored in the logical model of the SCMS and therefore available for all locations once they are added. If a facet has been added, it appears under "Available Facets" (5). This process can be seen in Figure 16.


Here you can see the available Ontologies and administrative Metadata.

Add administrative Metadata

Name: Administrative Metadata Name

Type: Technical Metadata

Available administrative Metadata

Add Ontology

Ontology Name: AeronauticalOntology

Ontology: facets.xml

Ontology Representation: RDF/XML

Available Facets

SemanticFacet: http://www.jku.at/BEST/ontologies/facets#AircraftFacet

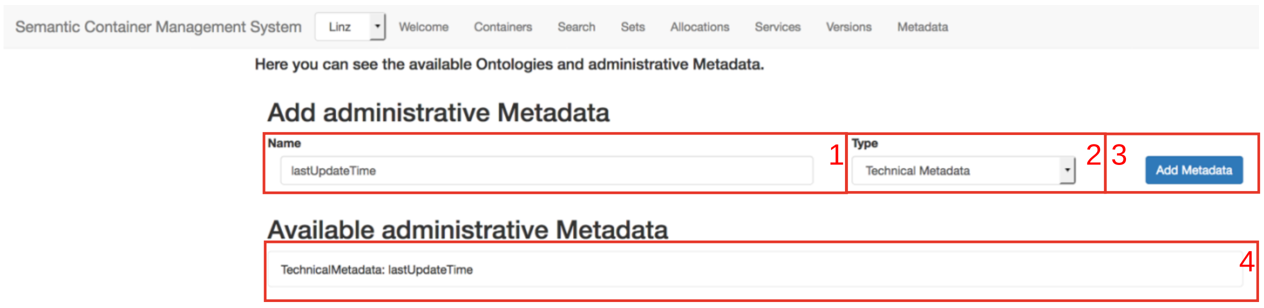
Available Ontologies and corresponding Facets

Ontologies	Facets	Facet Type
AeronauticalOntology	http://www.jku.at/BEST/ontologies/facets#NotamSet_8	Semantic Facet
	http://www.jku.at/BEST/ontologies/facets#NotamSetFacet	
	http://www.jku.at/BEST/ontologies/facets#LocalConstructFacet	
	http://www.jku.at/BEST/ontologies/facets#Apron_SurfaceDefects	

Figure 16: Add a facet to an SCMS

4.1.3 Register an Administrative Metadata

One can register administrative metadata as well. An administrative metadata represents a specific information, e.g., the time when the last update was done. To add administrative metadata a name must be specified (1) and a type must be selected (2) as one can see in Figure 17. For administrative metadata, there exist "Technical Metadata" and "Quality Metadata". Finally, the metadata can be added by clicking the "Add Metadata" button (3). Once added, the administrative metadata appear under the administrative metadata label (4). The administrative metadata are also part of the logical model.



Here you can see the available Ontologies and administrative Metadata.

Add administrative Metadata

Name: lastUpdateTime 1

Type: Technical Metadata 2

3 Add Metadata

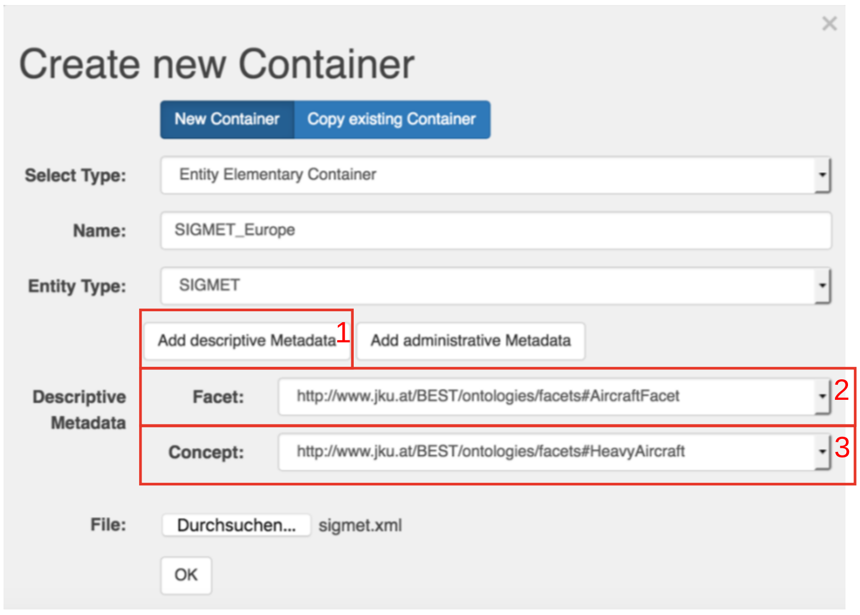
Available administrative Metadata

TechnicalMetadata: lastUpdateTime 4

Figure 17: Administrative metadata

4.1.4 Use Facets and Administrative Metadata when creating a Container

Whenever a container is created under the menu entry “Containers” → “Create new container” one can specify the facets and administrative metadata. In order to specify a facet, one has to add a descriptive metadata (1) to a container during its creation process. Next, one has to specify the facet which one wants to create (2), and finally one has to choose a concept from the list (3) as one can see in Figure 18.



Create new Container

Select Type: Entity Elementary Container

Name: SIGMET_Europe

Entity Type: SIGMET

Descriptive Metadata

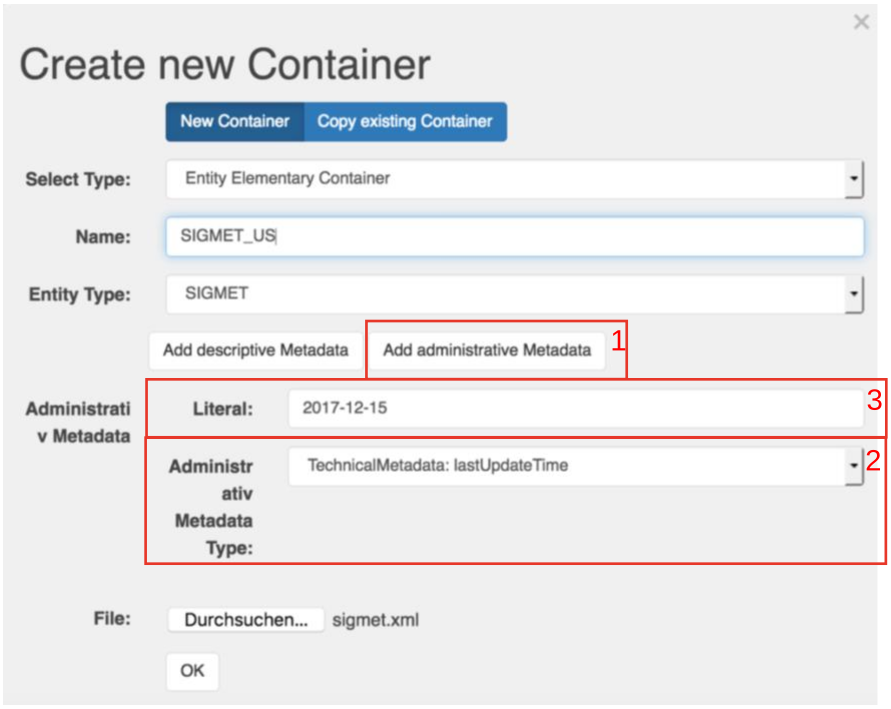
Facet: http://www.jku.at/BEST/ontologies/facets#AircraftFacet 2

Concept: http://www.jku.at/BEST/ontologies/facets#HeavyAircraft 3

File: Durchsuchen... sigmet.xml

Figure 18: Specify a facet

If one wants to add administrative data to a container one has to click on “Add administrative Metadata” (1). Next, one can choose the administrative metadata from the list of predefined administrative metadata (2). Finally, one can set its characteristics by setting its literal (3). One can specify an arbitrary non-empty value there. Now the administrative facet can be added as seen in Figure 19. There can be as many administrative metadata as necessary. If the literal exists, the SCMS references to that existing literal otherwise a new literal is created. The administrative metadata are also stored in the logical model and therefore available for all locations.


Create new Container

New Container | Copy existing Container

Select Type:	Entity Elementary Container
Name:	SIGMET_US
Entity Type:	SIGMET

Add descriptive Metadata | Add administrative Metadata

Administrativ Metadata:

Literal:	2017-12-15
Administrativ Metadata Type:	TechnicalMetadata: lastUpdateTime

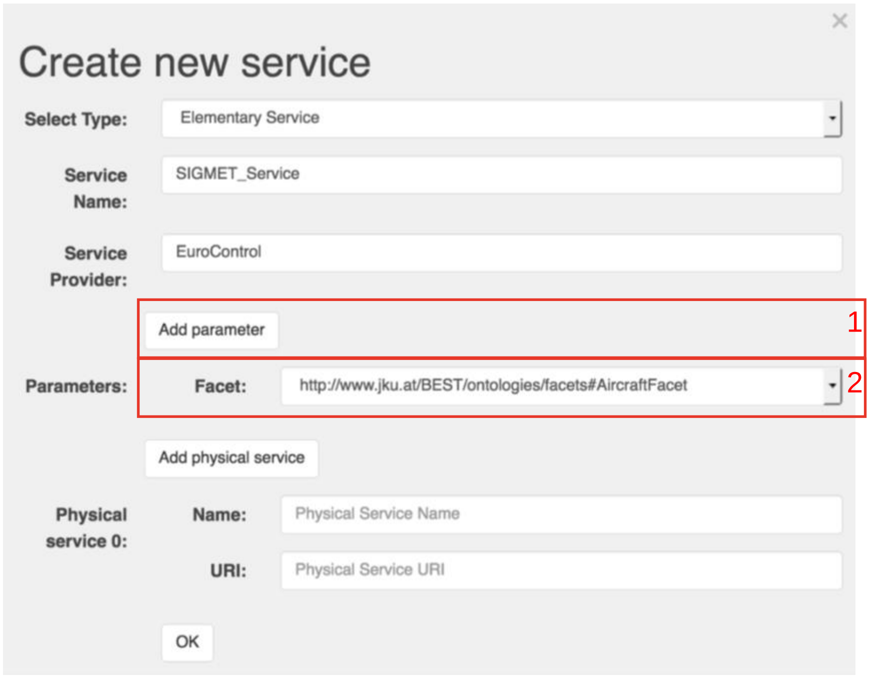
File: Durchsuchen... sigmet.xml

OK

Figure 19: Add administrative data to a container

4.1.5 Use Facets as Parameters for Service

Parameters can be defined for a service as well. These parameters are facets taken from an ontology. The concepts of the facet are characteristics. For example, one can register a service that requires a concept from the “AircraftFacet”. A concept of the “AircraftFacet” can be, e.g., “HeavyAircraft” and “Seaplain”. First, one has to click on the “Add parameter” (1) button and next one can choose the facet (2) as it can be seen in Figure 20. How a service can be created can be seen in Section 4.3.4.


Create new service

Select Type:	Elementary Service
Service Name:	SIGMET_Service
Service Provider:	EuroControl

Add parameter

Parameters: Facet: http://www.jku.at/BEST/ontologies/facets#AircraftFacet

Add physical service

Physical service 0:

Name:	Physical Service Name
URI:	Physical Service URI

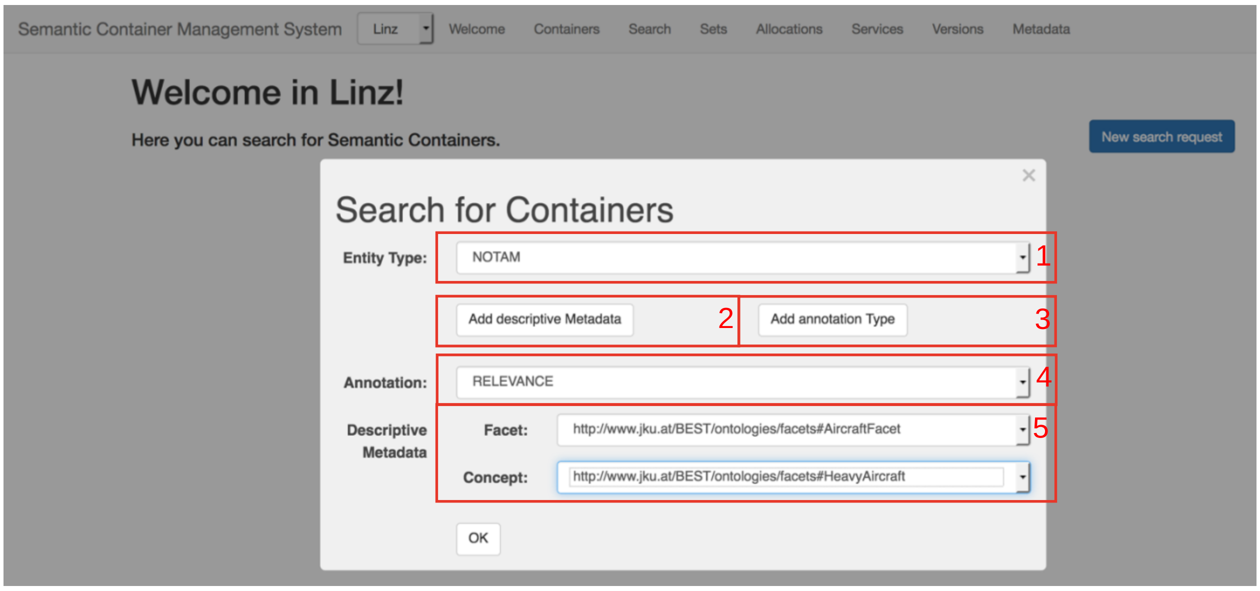
OK

Figure 20: Facets as parameters

4.1.6 Retrieve a Container Using Facets

If one wants to recover a container one can use the functionality under “Search” → “New search request”. If the button gets clicked, one can enter an information need. An information need specifies the information needed by the stakeholder. The containers can be filtered based on an entity type, annotation types, or/and specific facets. One can see the search function in Figure 21. The search algorithm works as follows:

- The containers can be filtered by their entity type (1). If the entity type should not be considered NONE can be selected. Other options are: NOTAM, METAR, TAF, SIGMET, FIR, AIRMET, and ATM Information.
- Next, the containers can be filtered by their annotation type. An annotation type filter can be added by pressing “Add annotation type” (3). When pressed, an annotation type can be selected (4). There can be no, one, or more filters for annotation types.
- Furthermore, the containers can be filtered by the descriptive metadata which describe their content. If no descriptive metadata are specified in the information need, the descriptive metadata are not considered in the search. Descriptive metadata can be added by pressing “Add descriptive Metadata” (2). For example, if a container is annotated with the concept “HeavyAircraft” of the facet “AircraftFacet”, it can be recovered by a search for the facet “AircraftFacet” with a concept which is either “HeavyAircraft” or a subclass of “HeavyAircraft” (5). If a container is annotated by two descriptive metadata and one searches a container with one descriptive metadata, then the container will not be returned because then the container cannot represent a superset. For example, if a container is annotated with the descriptive metadata “AircraftFacet: HeavyAircraft” and “EventFacet: Aerodome” and one needs a container which contains information about heavy aircrafts this container is not returned because it does represent a subset of the needed data. However, if there is a search for “AircraftFacet: HeavyAircraft” and “EventFacet: Runway” this container would be returned. This container fulfills the information need because “HeavyAircraft” is equal to “HeavyAircraft” and the concept “Aerodome” is a superclass of the concept “Runway”. This means that only containers which have a broader content than it is specified by the information need are returned.



Welcome in Linz!

Here you can search for Semantic Containers.

Search for Containers

Entity Type: NOTAM

Annotation: RELEVANCE

Descriptive Metadata

Facet: http://www.jku.at/BEST/ontologies/facets#AircraftFacet

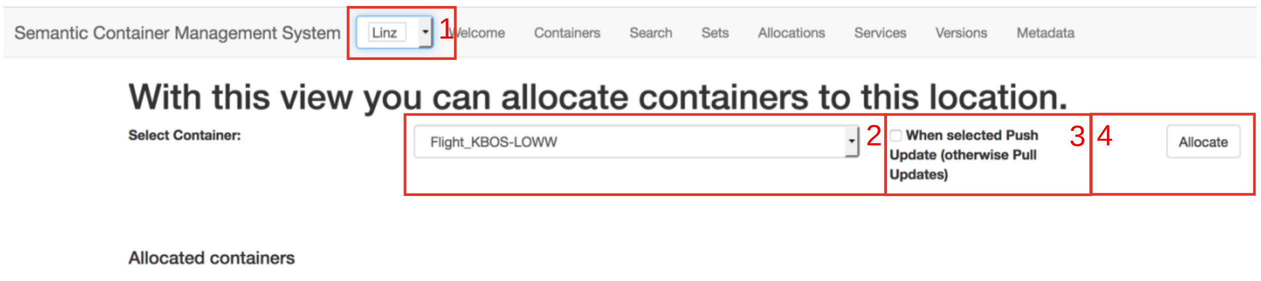
Concept: http://www.jku.at/BEST/ontologies/facets#HeavyAircraft

Figure 21: Example search

4.1.7 Allocate a Container

A container is allocated at a location when the physical data which is contained by a container is available at this location. If one allocates a container at a specific location, the data are copied to that location in an initial process. From now on, the data keep synchronized with either a PULL or the PUSH mechanism.

One can allocate a container under “Allocations”. First, one has to choose the location where one wants to allocate a container (1). Next, one has to choose the container which should be allocated (2). Then one can select whether the PULL or the PUSH mechanism should be used for synchronization (3). When the checkbox is selected then the updates are pushed to this location, otherwise it will be synchronized with the PULL mechanism. Finally, the container can be allocated (4). This can be seen in Figure 22.



With this view you can allocate containers to this location.

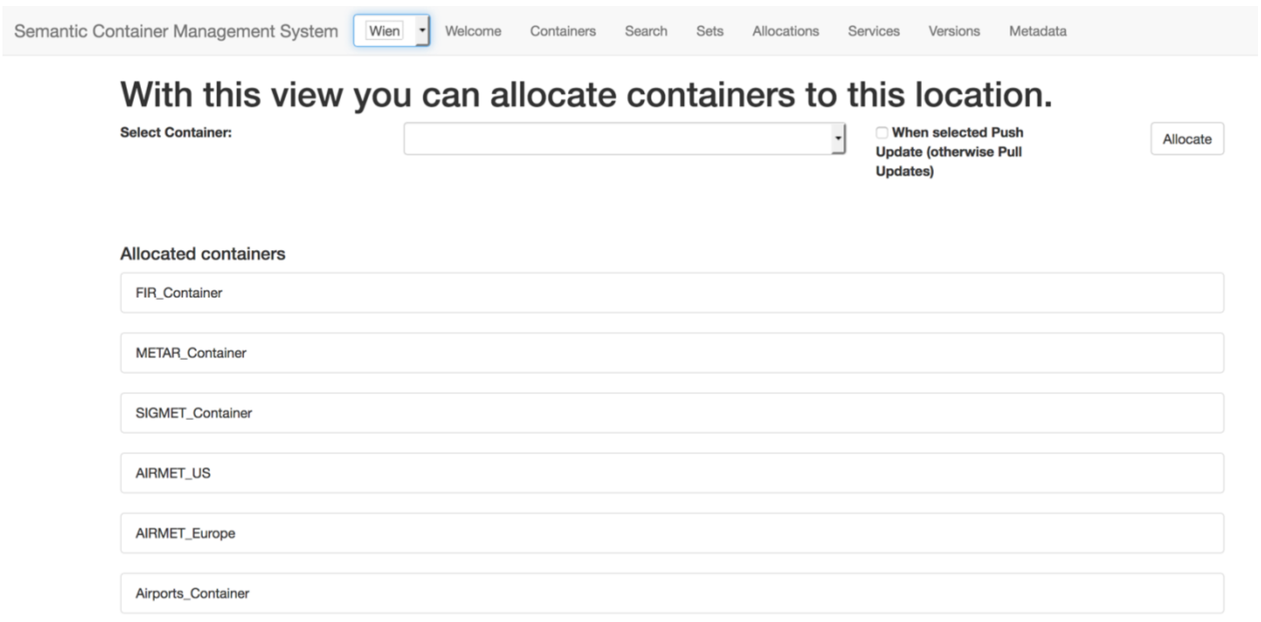
Select Container: Flight_KBOS-LOWW

When selected Push Update (otherwise Pull Updates)

Allocated containers

Figure 22: Allocate a container

A container is only available for allocation if it is not allocated at that specific location and it is a root container. This means that a container which is available for allocation may not be part of a composite. Therefore, only entity or composite containers which are not part of a composite are available for allocation. This is due to the constraint that composites are only allowed to be allocated as an entity. If one presses “Allocations” in the menu, one can see the allocated containers at a specific location as one can see in Figure 23.



With this view you can allocate containers to this location.

Select Container:

When selected Push Update (otherwise Pull Updates)

Allocated containers

FIR_Container

METAR_Container

SIGMET_Container

AIRMET_US

AIRMET_Europe

Airports_Container

Figure 23: Allocation view with allocated containers

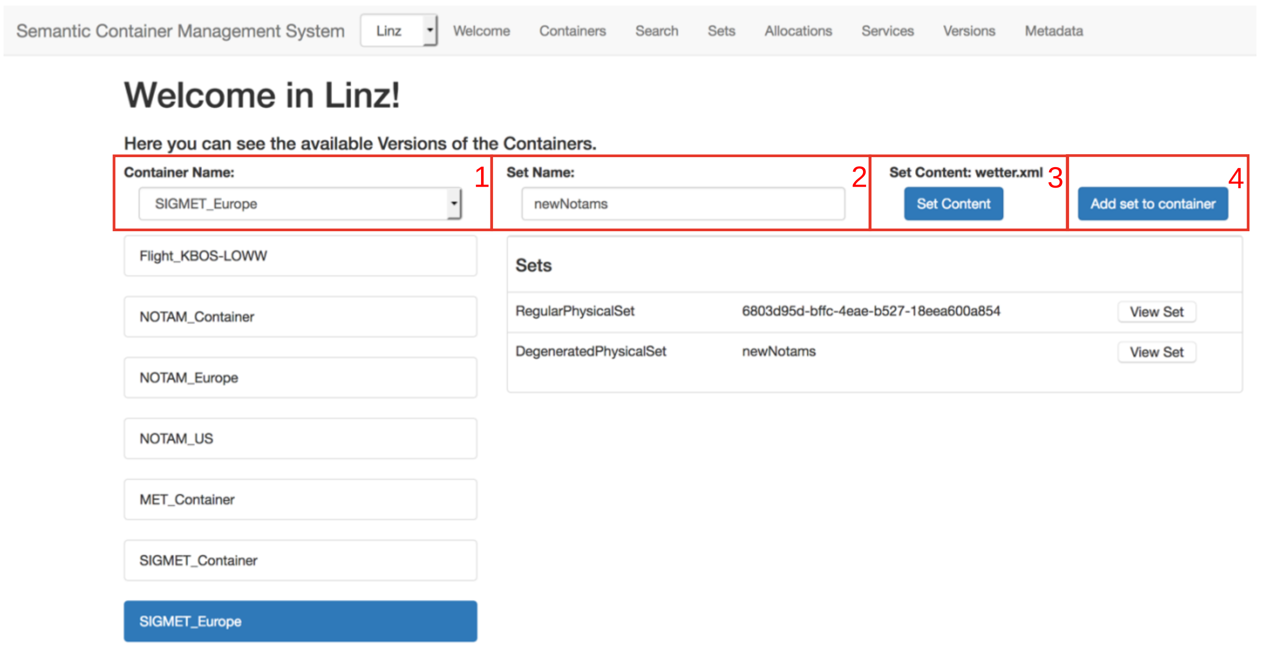
4.2 Consistency Management

The consistency management is about how the data are synchronized between the different allocations. When a container is created at a specific location then the container is inserted in the logical model and in the physical model of that location. The logical information is shared between all locations. The information in the physical model is only specific to a certain location. The first location where a container is created is the primary allocation. This means that this is the root location which contains the desired state. If it is possible, the secondary allocated containers are synchronized to this state. A secondary allocation is an allocation were the content of the primary allocation is replicated to when possible. For more details refer to Section 3.3.

In this section, the consistency management as it can be triggered by the frontend is stated. It is explained how to, e.g., add data sets to a container at the secondary allocation, add data sets at the primary allocation, and to inspect versions.

4.2.1 Add Data Set to a Container at the Secondary Allocation

If one wants to add a data set to a container one has to go to the menu entry “Sets”. Then one has to choose a container where the data should be added (1). It is only allowed to add data to elementary containers. Next, one has to specify the data set name (2) and its content (3). Finally, one can add the container (4). Because the container is a secondary allocated container, this results in a degenerated set. This can be seen in Figure 24.



Welcome in Linz!

Here you can see the available Versions of the Containers.

Container Name: SIGMET_Europe

Set Name: newNotams

Set Content: wetter.xml

- Flight_KBOS-LOWW
- NOTAM_Container
- NOTAM_Europe
- NOTAM_US
- MET_Container
- SIGMET_Container
- SIGMET_Europe

Sets

RegularPhysicalSet	6803d95d-bffc-4eae-b527-18eea600a854	View Set
DegeneratedPhysicalSet	newNotams	View Set

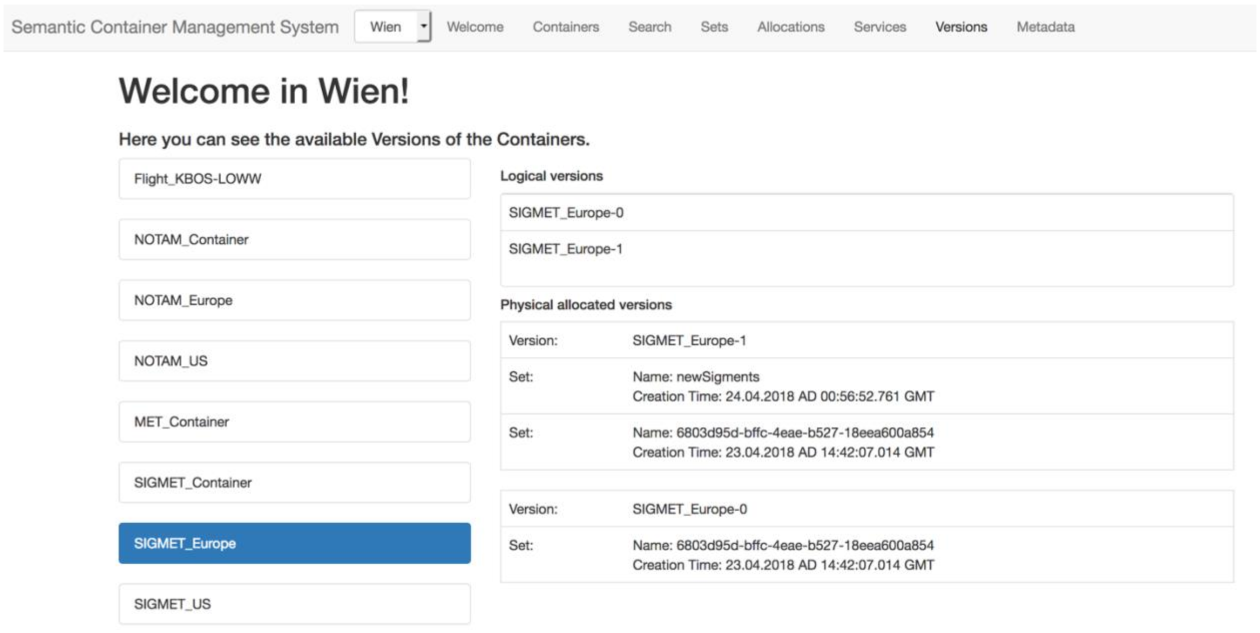
Figure 24: Add data set to a secondary allocation

4.2.2 Add Data Set to a Container at the Primary Allocation

The same procedure as in Section 4.2.1 can be repeated at the primary location of the container. In this prototype, if a data set is added to a primary location, the data set is a regular set. The primary container always represents the state which is transferred to the secondary containers if possible. The primary location and the secondary locations are registered in the logical model and therefore can be seen in the tab “Containers”.

4.2.3 Inspect Versions

In order to inspect which versions existed for one container at a specific location one can use the “Versions” view. By clicking on one container the versions which existed of a specific container will be shown. Those versions are separated into physical versions and logical versions. The logical versions are loaded from the logical model whereas the physical versions are loaded from the physical model of the location accessed by the frontend. The data sets can only be viewed from the physical model because the data sets are not included in the logical model because they are location specific. In Figure 25, one can see that the container has two versions. One version with one data set and one version with two data sets. The versions never differ from the versions which are registered in the logical model.



Here you can see the available Versions of the Containers.

Logical versions

SIGMET_Europe-0
SIGMET_Europe-1

Physical allocated versions

Version:	SIGMET_Europe-1
Set:	Name: newSigments Creation Time: 24.04.2018 AD 00:56:52.761 GMT
Set:	Name: 6803d95d-bffc-4eae-b527-18eea600a854 Creation Time: 23.04.2018 AD 14:42:07.014 GMT

Version:	SIGMET_Europe-0
Set:	Name: 6803d95d-bffc-4eae-b527-18eea600a854 Creation Time: 23.04.2018 AD 14:42:07.014 GMT

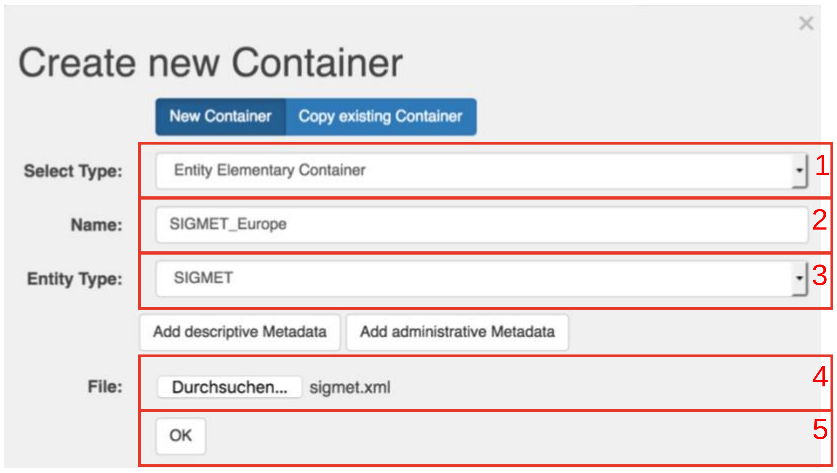
Figure 25: Inspect versions

4.3 Provenance and Composition

Provenance and composition deals with the registration of containers, the composition of containers, and the registration of service provides, services, and service calls. With the frontend, it is possible to register containers in the SCMS. Every elementary container contains the data of one single data type, e.g., METEAR. However, in practice a container must provide the information for a specific information need. To fulfill this requirement, it is necessary to combine containers together. Furthermore, it is necessary to document from where the data come from. This can be done by documenting the provenance of the data. Therefore, services can be registered in the system. The following part describes how to create containers and register services in the SCMS.

4.3.1 Create an Elementary Container

An elementary container is the basic element which is managed by the SCMS. This is the part which actually contains the data. There are two types of elementary containers. An entity elementary container and an annotated elementary container. The difference is that an annotated elementary container can have annotations as well. An annotation represents a tool to specify the content of an elementary container more precisely, not only by using the entity type. One can add a container by pressing at “Containers” → “Create new container”. First one must choose the type (1) as one can see in Figure 26. Then the name (2) and the entity type (3) must be specified. Finally, the initial content also called basic set (4) must be specified. Finally, the container can be added (5).

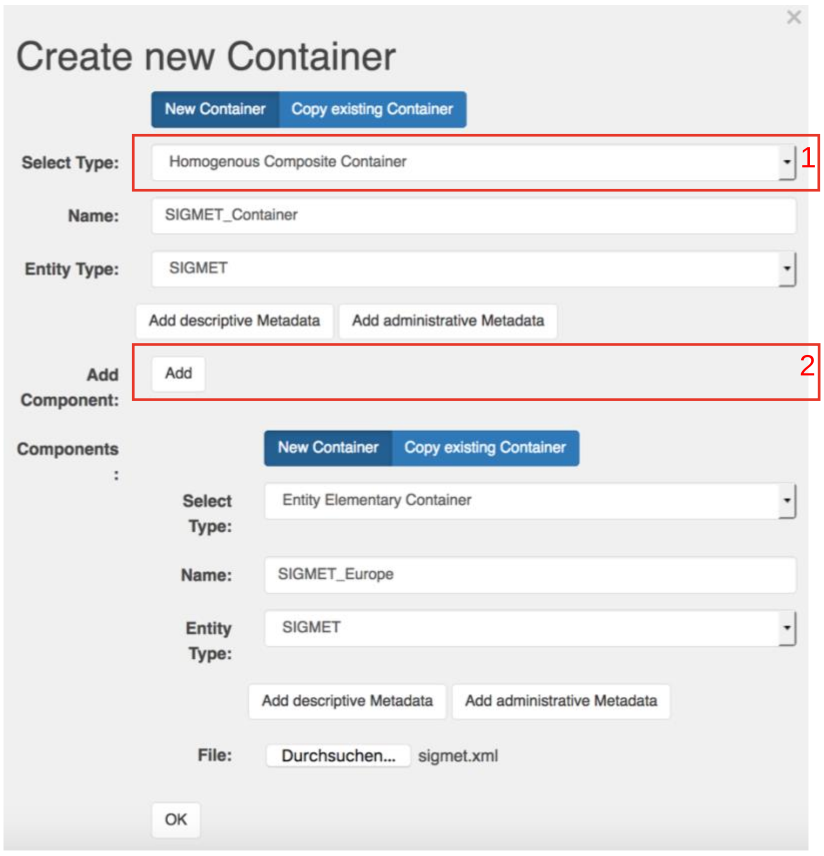


Select Type: Entity Elementary Container
Name: SIGMET_Europe
Entity Type: SIGMET
File: sigmet.xml

Figure 26: Creation of an elementary container

4.3.2 Create a Composite Container

Whenever one or more containers are combined it is called a composition. There are two types of compositions. There is a homogenous composition which only contains containers of the same entity type. On the other hand, there is a heterogeneous composition. A heterogeneous composition consists of containers of different entity types. Every composition is exposed into either a homogenous container or a heterogeneous container. A composite container represents an undividable entity. Therefore, a composition must be specified during the creation process and cannot be created later on. First, one has to select the container type either “Homogenous Composite Container” or “Heterogeneous Composite Container” (1) and then one has to add one or more sub-containers (2) as it can be seen in Figure 27. The sub-containers can be compositions or elementary containers. The process of specifying an elementary container can be seen in Section 4.3.1.



Create new Container

New Container | Copy existing Container

Select Type:	Homogenous Composite Container
Name:	SIGMET_Container
Entity Type:	SIGMET

Add descriptive Metadata | Add administrative Metadata

Add Component: Add

Components:

New Container | Copy existing Container

Select Type:	Entity Elementary Container
Name:	SIGMET_Europe
Entity Type:	SIGMET

Add descriptive Metadata | Add administrative Metadata

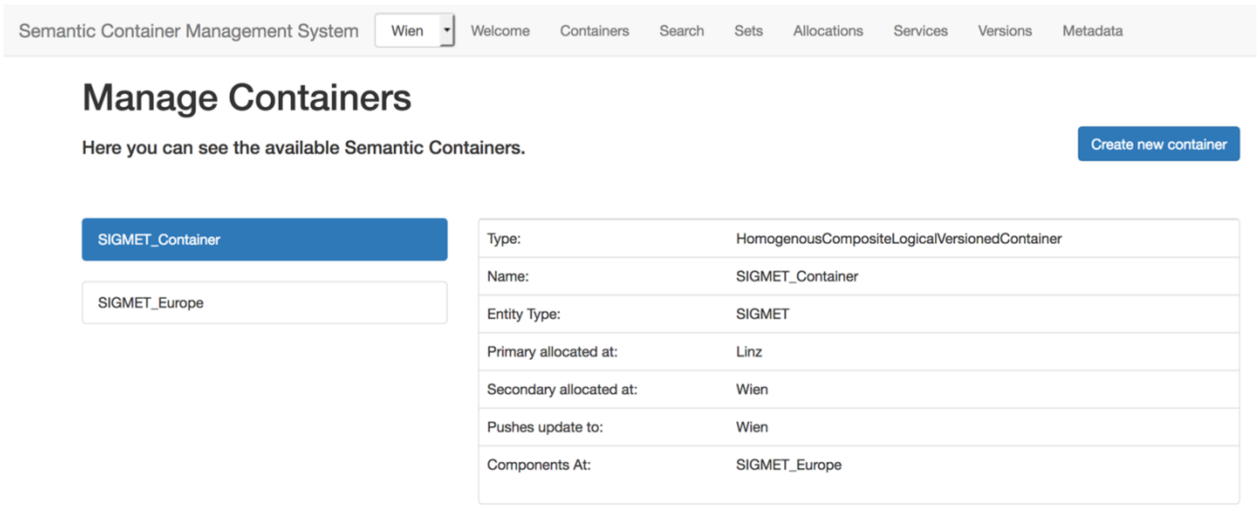
File: Durchsuchen... sigmet.xml

OK

Figure 27: Creation of a homogenous container

4.3.3 Inspect Containers

If one wants to inspect the containers which exist in the SCMS, one has to navigate to “Containers” in the menu. There is a full list of containers and the corresponding information. This information includes the container type, the name, the primary allocation, the secondary allocations, where it pushes updates to, and if it is a composite container it shows the composites as well. This information is location-independent because it is stored in the logical model. One can see an example overview in Figure 28.



Manage Containers

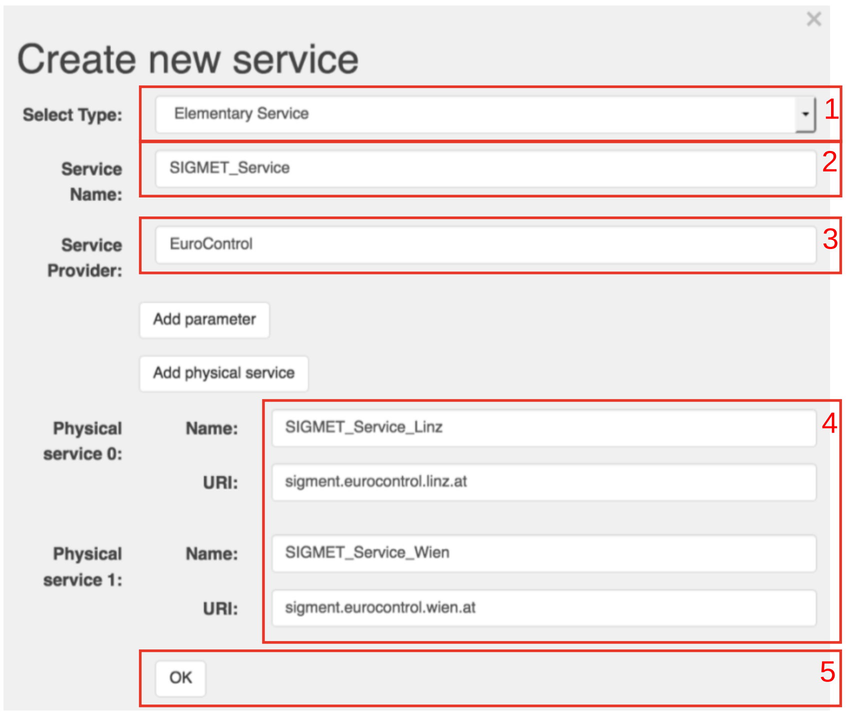
Here you can see the available Semantic Containers.

Type:	HomogenousCompositeLogicalVersionedContainer
Name:	SIGMET_Container
Entity Type:	SIGMET
Primary allocated at:	Linz
Secondary allocated at:	Wien
Pushes update to:	Wien
Components At:	SIGMET_Europe

Figure 28: Container overview

4.3.4 Register a Service

In the SCMS, one can register a service as well. Therefore, one has to choose "Services" → "Create new service" in the menu. One can choose the service type (1) as it can be seen in Figure 29. There exist elementary and composite services. A composite service is a service which is composed from different elementary services. To create a new service, one has to enter a service name (2), specify the service provider (3), and the physical locations of the services (4). Every physical location consists of a URI where the service operates and a name. Finally, the button "OK" (5) must be pressed.



Create new service

Select Type: Elementary Service

Service Name: SIGMET_Service

Service Provider: EuroControl

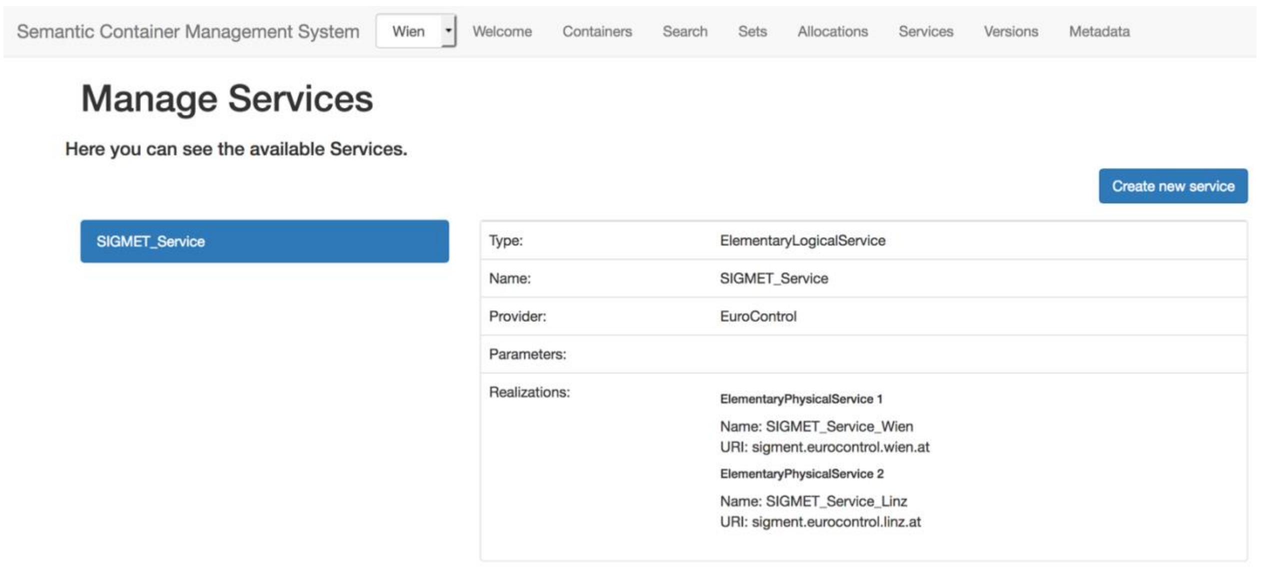
Physical service 0: Name: SIGMET_Service_Linz URI: sigment.eurocontrol.linz.at

Physical service 1: Name: SIGMET_Service_Wien URI: sigment.eurocontrol.wien.at

Figure 29: Add a service

4.3.5 Inspect Services

If one clicks on "Services" the registered services show up. In this view, one can see a list of the registered services and if one clicks on a specific service the detailed information of a service shows up. This information contains the service type, the name, the provider, the parameters, and the realizations of the service. One can see an example in Figure 30.



Manage Services

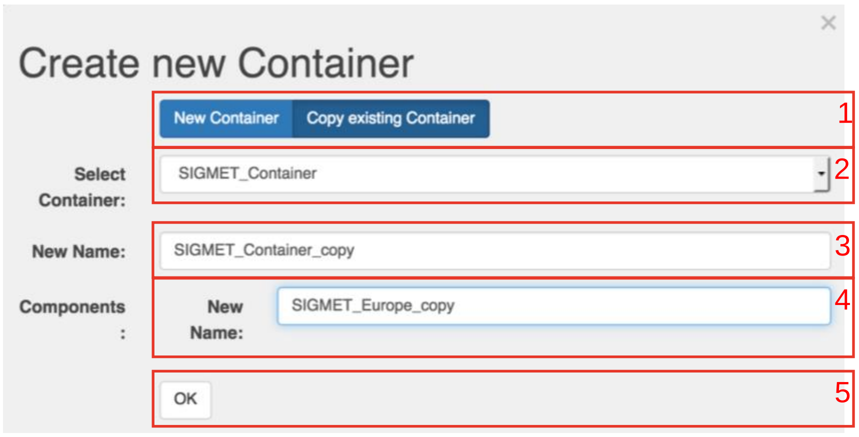
Here you can see the available Services.

Type:	ElementaryLogicalService
Name:	SIGMET_Service
Provider:	EuroControl
Parameters:	
Realizations:	ElementaryPhysicalService 1 Name: SIGMET_Service_Wien URI: sigment.eurocontrol.wien.at ElementaryPhysicalService 2 Name: SIGMET_Service_Linz URI: sigment.eurocontrol.linz.at

Figure 30: Inspect services

4.3.6 Copy an Existing Container

The SCMS allows the copying of an existing container. This means that the copied container has another container as source where the data comes from. A container can be copied if one clicks on “Containers” a then on “Create new container”. Then one can select whether an existing container should be copied or a new container should be created (1). Next, one can select the container which should be copied (2). Then one can specify a new name for the container (3) and its components (4) which must differ from the original names. Finally, if one clicks on “OK” the container will be copied (5). This can be seen in Figure 31.



Select Container: SIGMET_Container

New Name: SIGMET_Container_copy

Components: New Name: SIGMET_Europe_copy

Figure 31: Copy an existing container

4.4 Implementation Details

The frontend is designed according to the Single-Page Paradigm [51]. In Single-Page applications the state of the application is stored on the client side. Furthermore, the client is self-contained, this means a Single-Page application runs completely inside the browser and can choose the services which it might invoke freely. On the other side, the server is responsible to expose an interface which can be invoked form a client which must not be necessarily a Single-

Page application. In the initial step, the start page is transferred to the client which includes JavaScript, HTLM, and CSS files. These files are executed in the client's web browser. From now on, the server is only evolved when the client requests new data or wants to add, update, or delete data on the server. Before the Single-Page Paradigm evolved, the client's browser was only used to present the web site which was completely rendered at the sever [51]. This had the effect that on every request the server had to render the JavaScript, HTML, and CSS files. This in turn led to an increase of the server load and the network traffic. However, those systems are less complex and the web site provider has the full control over the environment and the consistence of the web page [51]. With the Single-Page Paradigm, the web developer must trust the rendering process on the client side, especially that it works correctly even when the data changes [51]. Despite this disadvantage, the Single-Page Paradigm has more advantages over traditional paradigms [51]. Applications which are designed using the Single-Page Paradigm reveal a higher usability and can work without a network connection, when no data needs to be requested.

4.4.1 Technologies and Tools

Molin [52] compares several Single-Page Application Frameworks. Three major frameworks compete each other: React, AngularJS, and Angular 2. The findings are that the frameworks are overall similar. Molin uses a criteria catalog to assess the capability of those frameworks. By using those criteria, you can see that the biggest difference between those frameworks is that AngularJS has the most stackoverflow followers and has the biggest community behind it [52]. Therefore, AngularJS was chosen. AngularJS is an extensible framework for building web applications [53]. With AngularJS, one can separate an application into different parts, each of those managed by a controller. A controller holds the data and represents the logic of how the application interferes with each other. It is written in JavaScript. Every controller is responsible for a specific view. The view knows its controller however, the controller does not know its view. Changes of the variables in the controller are automatically propagated to the view by using a two-way data binding. AngularJS handles the manipulation of the Document Object Model (DOM) [53]. Another feature of AngularJS represents services. A service represents a substitutable object. It is wired together by dependency injection. A service can be used to hold code which is shared throughout the application [54]. Furthermore, AngularJS allows the create modules. A module can be seen as containers for different parts of the application [55]. Modules can contain controllers, services, etc. and can depend on other modules. The recommended structure for modules is: a module for each feature, a module for each reusable component, and an application level module hence AngularJS does not have a main method which wires the application together [55].

For the visual interface a Cascading Style Sheets (CSS) framework was chosen. A CSS framework provides solutions to common problems. Therefore it is used to solve new problems of the same structure [56]. Such a framework usually contains an CSS source code, which handles the positioning of the elements, style definitions for HTML elements, CSS classes and solutions for browser incompatibilities [56]. Bootstrap is a technical well implemented framework with a huge popularity, which is also its biggest strength. Bootstrap offers as many resources as its four biggest competitors combined [56]. For example, resources are articles, books, tutorials, and third-party plugins. Therefore, bootstrap was chosen in the implementation.

For developing, a standard text editor was used. The application was executed and debugged in Firefox. In slant, a website specialized on ranking, Firefox was ranked as the best browser for web development followed by Google Chrome [57].

4.4.2 Structure

The “index.html” represents the main site of the web application. In the “index.html” the JavaScript files of the external modules as well as the internal modules are included. The different CSS files which are relevant for the frontend are referenced as well. In the basic files of an AngularJS application, one must define the application level module. This module is called “scms” standing for “Semantic Container Management System”. It is defined in a file called “app.js”. One particular file is the “services.js”. As the name indicates, in this file the module “scms.services” is defined. It provides services which are used over the whole application. These services are called: “serverRestLocation”, “requests”, “fileservices”, and “infodialog” service. The “serverRestLocation” service provides the current URI of the backend. The “requests” service contains all possible requests to the backend which are used in the fronted. This has the advantage that when a service of the backend is called from multiple places in the frontend and the interface of the backend changes, one must only update one place in the frontend. Furthermore, it increases the testability. Every HTTP-request is encapsulated in its own method in the “requests” service. The “fileservices” service provides one utility: it allows to read a file from the current system and stores its content in a JavaScript variable. Finally, the “infodialog” service opens an “infoDialog”. An “infoDialog” is a simple dialog with only a header and a content.

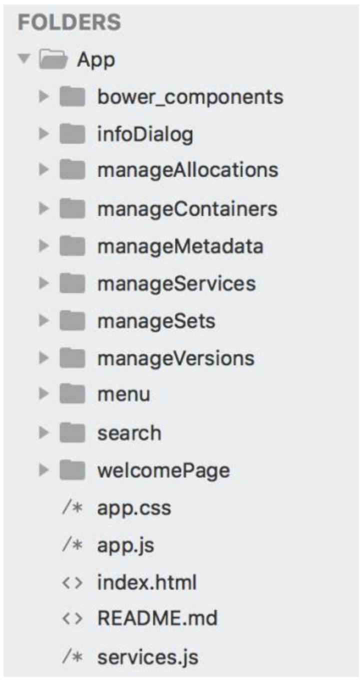


Figure 32: Frontend folder structure

In the frontend, one differentiates between internal or external modules. Internal modules are written especially for the frontend of the SCMS whereas external modules are third-party software. The internal modules can be further classified into page or utility modules. A page module is a module which contains only one controller. This module is then used to provide the content for a specific page. Every page has an entry in the menu. One can find a detailed list of modules in Table 6. Every internal module is implemented in a separate folder, except if the internal module represents a dialog which is used in a page module. In Figure 32 one can see the folder structure. One can see that the application module in “app.js”, the “index.html”, and the “services.js” are in the base folder. They represent either top-level modules or services which are used throughout the application. The submodules are defined in the folders.

Table 6: Modules of the frontend of the SCMS

Module Name	Type	Description
ngRoute	external	With the “ngRoute” module one can link URIs to specific controllers or views. In the frontend of the SCMS the module is used to assign a URI to the different page modules. This works in the following way: inside “app.js” “ngRoute” is configured. For each and every route a different view is assigned. The view in turn knows its controller. Whenever the URI is changed the “ngRoute” replaces a certain div (div annotated with ng-view) in the “index.html” with the related view. Therefore, “ngRoute” realizes the navigation. The following mappings are defined: “/” to “welcomePage/welcomePage.html”, “/manageContainers” to “manageContainers/-manageContainers.html”, “/manageSets” to “manageSets/manageSets.html”, “/manageVersions” to “manageVersions/manageVersions.html”, “/manageAllocations” to “manageAllocations/manage-Allocations.html”, “/manageServices” to “manageServices/manageServices.html”, “/manageMetadata” to “manageMetadata/manage-Metadata.html”, and “/search” to “search/search.html”
ngDialog	external	The module “ngDialog” opens up a dialog in an AngularJS application. “ngDialog” is used whenever a dialog is required in the application. In order to open up a dialog, one needs to specify a dialog view and a corresponding dialog controller.
scms.menu	internal utility	The menu module is visible in the frontend at its top. It implements the navigation logic. The menu makes use of “ngRoute” for changing the view. Furthermore, if one switches between different locations, the menu updates the backend location in the “service.js”.
scms.infoDialog	internal utility	In the frontend, the majority of dialogs require only a head and a content. The module “scms.infoDialog” consists of a controller and a view which then can be opened with “ngDialog”. The controller has only two properties a title and a head. Those properties are represented in the view of the dialog. In the frontend, there is also a utility method which opens up the dialog. It is the “infodialog” service in “service.js”.
scms.welcomePage	internal page	The welcome page module represents the initial page of the frontend. It is the first page a user will browse to. It displays a list of available locations.
scms .manageContainers	internal page	This page visualizes the available logical containers. One can explore the containers and request additional metadata about a logical container, e.g., its type, its

Module Name	Type	Description
		entity type, its primary allocations, and its secondary allocation. Furthermore, one can open up the dialog which is responsible for creating a new container.
scms .newContainerDialog	internal utility	This module contains two views and two controllers. Every view knows its corresponding controller. It has two views because one view represents the dialog view, which is the entry point for "ngDialog". The second view represents the input form for a single container. It is included in the dialog view as well as in itself (recursive) as a sub-view. This hierarchy is needed, because if one adds a composite container with a component, in fact the data for two containers must be entered. Therefore, the input form in the second view is separated from the actual dialog view and can be reused multiple times. In AngularJS, a view can reference another view by using "ngInclude".
scms .manageAllocations	internal page	This module consists of one controller and one view. It shows the current allocated containers at a specific location. It allows not allocated containers to be allocated at the corresponding location.
scms .manageServices	internal page	This module consists of one controller and one view. It shows the current services which are registered in the SCMS. Furthermore, the dialog which allows one to add new services can be opened from this module.
scms .newServiceDialog	Internal utility	This module consists of two views and two controllers. The first view is the view which is opened by "ngDialog". The second view represents a form where the data for the service can be entered. A view can reference another view by using "ngInclude", a mechanism provided by AngularJS. Whenever the user adds subservice the second view is replicated. This is the same principle as in the module "scms.newContainerDialog".
scms .manageMetadata	internal page	This module consists of one view and one controller. This module shows the available administrative metadata and the available facets. Furthermore, it allows one to register new faces and administrative metadata. Because of the fact that facets have to be taken from an ontology, this view allows one to register ontologies as well.
scms.manageSets	internal page	This module consists of one view and one controller. One can see the container with its data sets. This module provides functionalities which open an "infoDialog". The "infoDialog" shows the content of the data sets.
scms	internal	This module consists of a view and a controller. It

Module Name	**Type**	**Description**
.manageVersions	page	allows one to inspect the version and the data sets a specific version contained.
scms.services	internal page	This module contains the services which are available throughout the application. Those services are described in the beginning of Section 4.4.2.
scms.search	internal page	The module “scms.search” consists of one view and one controller. It displays the result of the query which can be entered in the search dialog.
scms.searchDialog	internal utility	The module “scms.searchDialog” consists of one view and one controller. One can specify which entity type, which annotation type, and which descriptive metadata a returned container must have. The descriptive metadata consist of facets. For every facet which should be searched for, a concept must be specified.

5. Summary and Future Work

This thesis describes the prototypical implementation of an SCMS as envisioned in the BEST project. The SCMS backend is a DBMS designed for the distributed storage of semantic containers. The SCMS backend offers features to register ontologies, to register facets out of an ontology, to create containers, to allocate containers, to add data sets to a container, and to register services. Furthermore, the SCMS backend offers features to query the data stored at a specific location. For example, it offers features to retrieve the registered administrative data and facets, to retrieve a container using facets, to inspect versions, and to inspect containers. These features are available at every location where an instance of the SCMS backend runs. To keep the data consistent between the locations, the instances synchronize with each other and, for some information, use a shared registry which might be replicated as well. Moreover, every SCMS backend has a RESTful interface that functions as application programming interface for third-party applications. With this RESTful interface, the functions described above can be invoked.

The SCMS frontend is an application that is able to access the SCMS backend at different locations. It uses the RESTful interface of the SCMS backend and is currently the only application to do so. With this frontend, one can demonstrate the interactions which in future will be done by third-party applications.

The SCMS backend and the SCMS frontend demonstrates the concepts developed in the BEST project in Deliverable 2.2 [6]. Specifically, this prototype demonstrates how semantic containers can be used in the ATM domain. The prototype further estimates the feasibility of an SCMS. To demonstrate feasibility, the SCMS went online and the use case as described in Section 1.1 was conducted. For this system test, two SCMS backends were started at one server. However, it appeared as if the two SCMS backends ran on two different servers. This is possible with the help of an Integration Reverse Proxy, which hides the physical infrastructure of the server from external parties [58]. Furthermore, the SCMS frontend was hosted by an Apache HTTP Server, which was behind the same Integration Reverse Proxy [59]. The outcome of this thesis is one foundation of the Deliverable 3.2 [7] which describes prototypes that demonstrate the concepts which are envisioned in the BEST project. Furthermore, this prototype is the foundation for a demonstration at the EKAW conference [8].

This thesis is limited to the features described in the beginning of Section 5. Features of the Deliverable 2.2 [6] which are not supported by the SCMS are, e.g., metadata management for physical containers and a service reference when a container is derived by using the copy function. Besides the functional limitations, the prototype is limited to get and insert statements. This allows simpler synchronization mechanisms and is sufficient in regard to the demonstration purpose of the proof-of-concept prototype. The limitation that only get and insert statements are allowed implies that containers and data sets can only be added and obtained. The fact that the shared logical registry is not replicated represents another limitation. In order to realize a shared logical registry that is fully replicated, effective techniques for RDF database replication must be employed which is left to future work. The shared logical registry is implemented as a RDF database and therefore there exist synchronization mechanisms which are not the focus of this thesis.

In the future, a prototype must be developed which not only demonstrates the concepts envisioned in the BEST project by using semantic technologies but also evaluates the concepts in regard to, e.g., the performance (see [60]), the storage usage, the system's behavior when handling a big volume of data, the behavior under big loads, and the integration of third-party applications. In future, the semantic container approach might be used beyond the ATM domain because it is applicable to all use cases where data need to be bundled and distributed. Therefore, it might be suitable in other domains, e.g., logistics.

6. References

[1] SKYbrary, “Situational Awareness,” *SKYbrary*, 23-Oct-2016. [Online]. Available: https://www.skybrary.aero/index.php/Situational_Awareness. [Accessed: 20-Jul-2018]

[2] Eurocontrol and FAA, “AIXM 5.1.1 - data model (UML).” 15-Apr-2016 [Online]. Available: http://aixm.aero/document/aixm-511-data-model-uml. [Accessed: 29-Aug-2018]

[3] ICAO and WMO, “IWXXM 2.1.1 - data model (UML).” 15-May-2018 [Online]. Available: http://schemas.wmo.int/iwxxm/2.1. [Accessed: 29-Aug-2018]

[4] SINTEF, Frequentis, University of Linz, Slot Consulting, and Eurocontrol, “Project Description,” *BEST*, 2016. [Online]. Available: http://www.project-best.eu/project.html. [Accessed: 19-Jul-2018]

[5] C. Schuetz, B. Neumayr, and M. Schrefl, “Deliverable 2.1: Techniques for ontology-based data description and discovery in a decentralized SWIM knowledge base.” 09-Mar-2018 [Online]. Available: http://www.project-best.eu/downloads/D2.1%20Techniques%20for%20ontology-based%20data%20description%20and%20discovery%20in%20a%20decentralized%20SWIM%20knowledge%20base%20(1).pdf. [Accessed: 19-Jul-2018]

[6] C. Schuetz, B. Neumayr, M. Schrefl, and E. Gringinger, “Deliverable 2.2: Ontology-based techniques for data distribution and consistency management in a SWIM environment.” 17-May-2018 [Online]. Available: http://www.project-best.eu/downloads/D2.2%20Ontology-based%20techniques%20for%20data%20distribution%20and%20consistency%20management%20in%20a%20SWIM%20environment.pdf. [Accessed: 19-Jul-2018]

[7] E. Gringinger, C. Fabianek, and C. Schütz, “Deliverable 3.2: Prototype SWIM-enabled Applications.” 08-May-2018 [Online]. Available: http://www.project-best.eu/downloads/D3.2%20Prototype%20SWIM-enabled%20Applications.pdf. [Accessed: 29-Aug-2018]

[8] E. Gringinger, C. Fabianek, C. G. Schuetz, J. Stöbich, B. Neumayr, and M. Schrefl, “A Proof-of-Concept Implementation of a Semantic Container Management System for Air Traffic Management,” in *Proceedings of the 21st International Conference on Knowledge Engineering and Knowledge Management - Posters & Demos*, Nancy, France, Accepted for publication. To appear.

[9] Federal Aviation Administration, “Federal NOTAM System Airport Operations Scenarios.” Dec-2010 [Online]. Available: https://notams.aim.faa.gov/FNSAirportOpsScenarios.pdf. [Accessed: 28-Aug-2018]

[10] Federal Aviation Administration, “Aeronautical Information Manual.” 12-Oct-2017 [Online]. Available: https://www.faa.gov/air_traffic/publications/media/aim.pdf. [Accessed: 10-Apr-2018]

[11] SINTEF, Frequentis, University of Linz, Slot Consulting, and Eurocontrol, “Publications,” *BEST - SESAR*. [Online]. Available: http://project-best.eu/publications.html. [Accessed: 05-Sep-2018]

[12] S. Wilson and R. Suzic, “AIRM Primer.” 19-Sep-2018 [Online]. Available: https://www.eurocontrol.int/sites/default/files/content/documents/sesar/8.1.3.d47-airm-primer-v4.1.0.pdf. [Accessed: 18-Sep-2018]

[13] I. Kovacic *et al.*, “Ontology-based data description and discovery in a SWIM environment,” in *2017 Integrated Communications, Navigation and Surveillance Conference (ICNS)*, 2017, pp. 5A4-1-5A4-13 [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8011928&isnumber=8011878. [Accessed: 18-Jul-2018]

[14] B. Neumayr, E. Gringinger, C. Schuetz, M. Schrefl, S. Wilson, and A. Vennesland, “Semantic Data Containers for realizing the full potential of system wide information management,” in *2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)*, 2017, pp. 1–10 [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8102002&isnumber=8101896. [Accessed: 18-Jul-2018]

[15] P. Hitzler, M. Krötzsch, S. Rudolph, and Y. Sure, *Semantic Web: Grundlagen*. Springer-Verlag, 2008 [Online]. Available: https://link.springer.com/book/10.1007/978-3-540-33994-6. [Accessed: 24-Mar-2017]

[16] E. Gringinger, C. Schuetz, B. Neumayr, M. Schrefl, and S. Wilson, “Towards a value-added information layer for SWIM: The Semantic Container Approach,” in *2018 Integrated Communications, Navigation, Surveillance Conference (ICNS)*, 2018, pp. 3G1-1-3G1-14 [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8384870&isnumber=8384823. [Accessed: 19-Jul-2018]

[17] E. Gringinger, B. Neumayr, C. Fabianek, and A. Savulov, “Deliverable 3.1: Prototype Use Case Scenarios.” 20-Mar-2018 [Online]. Available: http://www.project-best.eu/downloads/D3.1%20Use%20Case%20Scenarios%20(2).pdf. [Accessed: 19-Jul-2018]

[18] R. T. Fielding and R. N. Taylor, “Principled Design of the Modern Web Architecture,” *ACM Trans. Internet Technol.*, vol. 2, no. 2, pp. 115–150, May 2002 [Online]. Available: http://doi.acm.org/10.1145/514183.514185. [Accessed: 12-Jun-2018]

[19] T. Berners-Lee and J. Hendler, “Publishing on the semantic web,” *Nature*, vol. 410, p. 1023, Apr. 2001 [Online]. Available: http://dx.doi.org/10.1038/35074206. [Accessed: 24-Mar-2017]

[20] B. Motik, P. F. Patel-Schneider, and B. Parsia, “OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax (Second Edition),” *w3c*, 11-Dec-2012. [Online]. Available: https://www.w3.org/TR/2012/REC-owl2-syntax-20121211/. [Accessed: 13-Jul-2018]

[21] C. Golbreich and E. K. Wallace, “OWL 2 Web Ontology Language New Features and Rationale (Second Edition),” *w3c*, 11-Dec-2012. [Online]. Available: https://www.w3.org/TR/owl2-new-features/. [Accessed: 15-Jul-2018]

[22] P. F. Patel-Schneider, “Differences between OWL 1 and OWL 2,” *w3c*, 01-Apr-2009. [Online]. Available: http://lists.w3.org/Archives/Public/public-owl-wg/2009Apr/0014.html. [Accessed: 15-Jul-2018]

[23] S. Ceri and G. Pelagatti, *Distributed Databases*. Singapore: McGRAW-Hill, 1984.

[24] S. Gilbert and N. Lynch, “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services,” *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002 [Online]. Available: http://doi.acm.org/10.1145/564585.564601. [Accessed: 02-Jul-2018]

[25] C. Batini and M. Scannapieco, *Data and information quality: dimensions, principles and techniques*. Cham: Springer, 2016 [Online]. Available: https://doi.org/10.1007/978-3-319-24106-7. [Accessed: 03-Jul-2018]

[26] P. Groth and L. Moreau, “PROV-Overview,” *w3c*, 30-Apr-2013. [Online]. Available: http://www.w3.org/TR/2013/NOTE-prov-overview-20130430/. [Accessed: 03-Apr-2017]

[27] Y. Gil and S. Miles, “PROV Model Primer,” *w3c*, 30-Apr-2013. [Online]. Available: https://www.w3.org/TR/prov-primer/. [Accessed: 06-Jul-2018]

[28] L. Moreau and P. Missier, “PROV-DM: The PROV Data Model,” *w3c*, 30-Apr-2013. [Online]. Available: https://www.w3.org/TR/prov-dm/. [Accessed: 03-Apr-2017]

[29] T. Lebo, S. Sahoo, and D. McGuinness, “PROV-O: The PROV Ontology,” *w3c*, 30-Apr-2013. [Online]. Available: https://www.w3.org/TR/prov-o/. [Accessed: 06-May-2018]

[30] L. Dodds and I. Davis, *Linked Data Patterns: A pattern catalogue for modelling, publishing, and consuming Linked Data*. 2012 [Online]. Available: http://patterns.dataincubator.org. [Accessed: 09-Jul-2018]

[31] D. Beech and B. Mahbod, “Generalized version control in an object-oriented database,” in *Fourth International Conference on Data Engineering*, 1988, pp. 14–22 [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=105441&isnumber=3237. [Accessed: 17-Jul-2018]

[32] W. F. Tichy, “RCS—a system for version control,” *Software: Practice and Experience*, vol. 15, no. 7, pp. 637–654, Jul. 1985 [Online]. Available: https://doi.org/10.1002/spe.4380150703. [Accessed: 17-Jul-2018]

[33] J. S. Meserole and J. W. Moore, “What is System Wide Information Management (SWIM)?,” in *2006 ieee/aiaa 25TH Digital Avionics Systems Conference*, 2006, pp. 1–8 [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4106233&isnumber=4106228. [Accessed: 18-Jul-2018]

[34] Y. Ueda and M. Ohara, “Performance competitiveness of a statically compiled language for server-side Web applications,” in *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*, 2017, pp. 13–22 [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7975266&isnumber=7975258. [Accessed: 22-Jul-2018]

[35] M. Fischer, K. Képes, and A. Wassiljew, “Vergleich von Frameworks zur Implementierung von REST-basierten Anwendungen,” Jun. 2013 [Online]. Available: https://d-nb.info/1043482024/34. [Accessed: 22-Jul-2018]

[36] Project Grizzly, “Project Grizzly: Goals of the Grizzly Framework,” *Project Grizzly*. [Online]. Available: https://javaee.github.io/grizzly/overview.html. [Accessed: 18-Aug-2018]

[37] “Chapter 9. Support for Common Media Type Representations,” *Jersey documentation*. [Online]. Available: https://jersey.github.io/documentation/latest/media.html#json.jackson. [Accessed: 18-Aug-2018]

[38] N. Nurseitov, M. Paulson, R. Reynolds, and C. Izurieta, “Comparison of JSON and XML data interchange formats: a case study,” *Caine*, vol. 9, pp. 157–162, 2009 [Online]. Available: https://www.cs.montana.edu/izurieta/pubs/caine2009.pdf. [Accessed: 18-Aug-2018]

[39] S. M. Freire, E. Sundvall, D. Karlsson, and P. Lambrix, “Performance of XML Databases for Epidemiological Queries in Archetype-Based EHRs,” in *Scandinavian Conference on Health Informatics 2012*, Linköping; Sweden, 2012, pp. 51–57 [Online]. Available: http://www.ep.liu.se/ecp/article.asp?issue=070&volume=&article=009. [Accessed: 22-Jul-2018]

[40] Solid IT GmbH, “DB-Engines Ranking - popularity ranking of native XML DBMS,” *DB-engines*, Sep-2018. [Online]. Available: https://db-engines.com/en/ranking/native+xml+dbms. [Accessed: 22-Jul-2018]

[41] M. Van Cappellen, Z. H. Liu, J. Melton, and M. Orgiyan, “XQJ: XQuery Java API is Completed,” *SIGMOD Rec.*, vol. 38, no. 4, pp. 7–13, Jun. 2010 [Online]. Available: http://doi.acm.org/10.1145/1815948.1815950. [Accessed: 22-Jul-2018]

[42] M. Voigt, A. Mitschick, and J. Schulz, “Yet Another Triple Store Benchmark? Practical Experiences with Real-World Data,” in *Proceedings of the 2nd International Workshop on Semantic Digital Archives, Paphos, Cyprus, September 27, 2012*, 2012, pp. 85–94 [Online]. Available: http://ceur-ws.org/Vol-912/paper7.pdf

[43] SparxSystems Software GmbH and atomicboy.tv, “Enterprise Architect 14.1,” *SparxSystems Europe*, 20-Aug-2018. [Online]. Available: https://www.sparxsystems.de/uml/neweditions/. [Accessed: 01-Sep-2018]

[44] SmartBear Software, “The Best APIs are Built with Swagger Tools | Swagger.” [Online]. Available: https://swagger.io/. [Accessed: 10-Aug-2018]

[45] T. Johnson, “Overview of REST API specification formats | Document REST APIs,” *I’d rather be writing*. [Online]. Available: https://idratherbewriting.com/learnapidoc/pubapis_rest_specification_formats.html. [Accessed: 10-Aug-2018]

[46] SmartBear Software, “Basic Structure | Swagger,” *swagger*. [Online]. Available: https://swagger.io/docs/specification/basic-structure/. [Accessed: 10-Aug-2018]

[47] Sun Microsystems, “Core J2EE Patterns - Data Access Object,” *Oracle*, 29-Mar-2013. [Online]. Available: http://www.oracle.com/technetwork/java/dataaccessobject-138824.html. [Accessed: 22-Jul-2018]

[48] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, and D. Smith, *The Java® Language Specification: Java SE 10 Edition*, vol. 10. 2018 [Online]. Available: https://docs.oracle.com/javase/specs/jls/se10/jls10.pdf. [Accessed: 13-Aug-2018]

[49] Oracle, “Package javax.ws.rs (Java(TM) EE 7 Specification APIs),” *JavaX Documentation*. [Online]. Available: https://docs.oracle.com/javaee/7/api/javax/ws/rs/package-summary.html. [Accessed: 18-Aug-2018]

[50] R. Sumereder, “OWL Ontologie für NOTAM Sets,” Johannes Kepler Universität Linz, Austira, 2016.

[51] J. Kuuskeri, “Experiences on a design approach for interactive web applications,” in *Proceedings of the 2nd USENIX conference on Web application development*, Portland, OR, USA, 2011, pp. 87–98 [Online]. Available: http://static.usenix.org/event/webapps11/tech/final_files/webapps11_proceedings.pdf#page=95. [Accessed: 31-Aug-2018]

[52] E. Molin, “Comparison of Single-Page Application Frameworks,” Oct. 2016 [Online]. Available: http://www.nada.kth.se/~ann/exjobb/eric_molin.pdf. [Accessed: 10-Aug-2018]

[53] Google, “AngularJS — Superheroic JavaScript MVW Framework,” *AngularJS*. [Online]. Available: https://angularjs.org/. [Accessed: 21-Jul-2018]

[54] Google, “AngularJS: Developer Guide: Services,” *AngularJS*. [Online]. Available: https://docs.angularjs.org/guide/services. [Accessed: 21-Jul-2018]

[55] Google, “AngularJS: Developer Guide: Modules,” *AngularJS*. [Online]. Available: https://docs.angularjs.org/guide/module. [Accessed: 21-Jul-2018]

[56] N. Jain, “Review of different responsive CSS front-end frameworks,” *Journal of Global Research in Computer Science*, vol. 5, no. 11, pp. 5–10, Nov. 2014 [Online]. Available: http://jgrcs.info/index.php/jgrcs/article/viewFile/949/607. [Accessed: 21-Jul-2018]

[57] “9 Best browsers for web development as of 2018,” *Slant*. [Online]. Available: https://www.slant.co/topics/7143/~browsers-for-web-development. [Accessed: 21-Jul-2018]

[58] P. Sommerlad, “Reverse Proxy Patterns,” in *Proceedings of the 8th European Conference on Pattern Languages of Programms (EuroPLoP ’2003)*, Irsee, 2003, pp. 431–458 [Online]. Available: https://hillside.net/europlop/HillsideEurope/Papers/EuroPLoP2003/2003_Sommerlad_ReverseProxyPatterns.pdf. [Accessed: 31-Jul-2018]

[59] Documentation Group, “Apache HTTP Server Project.” [Online]. Available: https://httpd.apache.org/. [Accessed: 01-Aug-2018]

[60] G. Brataas, B. Neumayr, C. G. Schuetz, and A. Vennesland, “Towards Scalability Guidelines for Semantic Data Container Management,” in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, New York, NY, USA, 2018, pp. 17–20 [Online]. Available: http://doi.acm.org/10.1145/3185768.3186302. [Accessed: 04-Sep-2018]

A. Development Environment

In the following, the development environment is described. The development environment includes the computer and the operating system used. Furthermore, the programs which are used in order to create the frontend and the backend are stated in that list.

The computer which is used in order to develop the SCMS for versioning and distributing is a MacBook Pro (Retina, 13-inch, Mid 2014) with an 2,6 GHz Intel Core i5, an 8 GB DDR3 Random Access Memory, and an Intel Iris 1536 MB video card.

The language which is used in order to create the backend was Java. The used Java Development Kit and Java Runtime Environment version is 1.8.0_51. The backend is developed with Eclipse IDE for Java Developers; Version: Oxygen.3a Release (4.7.3a) and Build id: 20180405-120. This build of Eclipse IDE comes with a build-in package management system called maven.

The development environment which is used in order to create the frontend is called Sublime Text (Build 3143). Sublime does not come with a built-in package management system, therefore bower (version 1.8) was used. The runtime environment for the frontend is Firefox Quantum (version 59.0.2). Firefox is also used in order to debug the webpage.

B. List of Third-Party Software

In the following, a list of third-party libraries which are used to realize the SCMS is given. Each library is stated with the corresponding version and license. In Table 7 one can see the libraries which are used in the frontend, in Table 8. one can see the libraries used in the backend. However, this does not include the development tools which are used to create the SCMS.

Table 7: Frontend libraries

Library	Version	License
angular	1.5.8	The MIT License (MIT)
angular-cookies	1.5.8	MIT
angular-route	1.5.8	MIT
bootstrap	3.3.7	MIT
jQuery	1.5.8	Open Source License from JS Foundation and other contributors
ng-dialog	0.6.4	MIT

Table 8: Backend libraries

Library	Version	License
jersey-container-grizzly2-http	2.26	Common Development and Distribution License (CDDL) Version 1.1
jersey-container-grizzly2-servlet	2.26	CDDL Version 1.1
jersey-hk2	2.26	CDDL Version 1.1
jackson-jaxrs-json-provider	2.9.3	Apache License Version 2.0
swagger-jersey2-jaxrs	1.5.18	Apache License Version 2.0
apache-jena-libs	3.6.0	Apache License Version 2.0
jena-fuseki-embedded	3.6.0	Apache License Version 2.0
basex	8.6.7	The 2-Clause BSD License
xqj-api	1.0	https://github.com/ligasgr/intellij-xquery/blob/master/licenses/xqj-api-license.txt
basex-xqj	9.0	Apache License Version 2.0
slf4j-api	1.7.5	MIT
slf4j-log4j12	1.7.5	MIT

C. REST Interface

The REST interface of the SCMS provides services which can be accessed by, e.g., ATM clients. These services allow clients to, e.g., add containers, to add datasets to a container, to query the containers, and to view the metadata of containers. There are four groups of services: the config, the container, the metadata, and the service group. The config group offers services to query the metadata of SCMS locations. The container group offers services which are linked to actions on a container like add, remove, or allocate a container. The metadata group offers services that allow to register and query ontologies, facets, and administrative metadata and further allow to query the facet's concepts. Finally, the service group offers services to access and modify provenance-related information. These groups are used by naming the paths.

C.1. Application Programming Interface

C.1.1 /config/location

The path “/config/location” contains a GET operation. The operation of the path “/config/location” is described in the following section.

C.1.1.1 GET: Returns information about the current SCMS location

This service returns the information about the current location. The location name, the URI, and the port of the REST interface of the current location are returned.

Response

Status Code	Reason	Response Model
200	Successful operation	Location
500	An unknown error occurred.	-

C.1.2 /config/locations

The path “/config/locations” contains a GET operation. The operation of the path “/config/locations” is described in the following section.

C.1.2.1 GET: Returns information about all SCMS locations

This service returns information about all locations that run an SCMS. The name of the location, the URI, and the port are returned by this service.

Response

Status Code	Reason	Response Model
200	Successful operation	array[Location]
500	An unknown error occurred.	-

C.1.3 /container

The path “/container” contains a GET operation. The operation of the path “/container” is described in the following section.

C.1.3.1 GET: Returns all registered logical versioned containers

This service returns all logical versioned containers that are registered in the logical registry. It sorts them so that all containers are followed by their sub-containers, if they have some. This service solely returns the containers' names and types. Other to the container related attributes, e.g., the semantic label, will not be returned.

Response

Status Code	Reason	Response Model
200	Successful operation	array[LogicalVersionedContainer]
500	An unknown error occurred.	-

C.1.4 /container/allocate

The path “/container/allocate” contains a GET operation. The operation of the path “/container/allocate” is described in the following section.

C.1.4.1 GET: Returns meta information about all allocated versioned containers

This service returns meta information about all versioned containers that are allocated at this location. The containers are either composites or elementary containers.

Response
Content-Type: application/json

Status Code	Reason	Response Model
200	Successful operation	array[LogicalVersionedContainer]
500	An unknown error occurred.	-

C.1.5 /container/allocate/{containerName}

The path "/container/allocate/{containerName}" contains a GET and a POST operation. The operations of the path "/container/allocate/{containerName}" are described in the following section.

C.1.5.1 GET: Returns a specific allocated physical versioned container

This service returns a specific allocated physical versioned container. The physical container contains all data sets that are replicated to this location.

Request
Content-Type: application/json

Name	Located in	Required	Description	Schema
containerName	path	yes	The name of the physical versioned container which should be returned.	string

Response
Content-Type: application/json

Status Code	Reason	Response Model
200	Successful operation	PhysicalVersionedContainer
400	There was an error in the request.	-
404	The physical versioned container is either not allocated at this location or does not exist.	-
500	An unknown error occurred.	-

C.1.5.2 POST: Allocates a specific physical container

This service allocates a specific physical container by adding it to the physical registry and by storing the container's data sets in the XML database of this location.

Request
Content-Type: application/json

Name	Located in	Required	Description	Schema
containerName	path	yes	The name of the physical container which should be allocated.	string
pushUpdatesTo	body	yes	This specifies whether the primary source pushes or an update manager pulls the updates to this allocation.	

Response
Content-Type: application/json

Status Code	Reason	Response Model
400	There was an error in the request.	-
404	The container was not found.	-
409	The physical container already exists at this location.	-
500	An unknown error occurred.	-

C.1.6 /container/allocate/{containerName}/physicalset

The path "/container/allocate/{containerName}/physicalset" contains a GET operation. The operation of the path "/container/allocate/{containerName}/physicalset" is described in the following section.

C.1.6.1 GET: Returns all data sets of an allocated physical container

This service returns the basic set and the delta sets. The basic set is always on index 0.

Request
Content-Type: application/json

Name	Located in	Required	Description	Schema
containerName	path	yes	The name of the physical container where the data sets should be returned.	string

Response
Content-Type: application/json

Status Code	Reason	Response Model
200	Successful operation	array[PhysicalSet]
400	There was an error in the request.	-
404	The physical container was not found at this location.	-
500	An unknown error occurred.	-

C.1.7 /container/allocate/{containerName}/physicalset/{physicalSetName}

The path "/container/allocate/{containerName}/physicalset/{physicalSetName}" contains a GET and a POST operation. The operations of the path "/container/allocate/{containerName}/physicalset/{physicalSetName}" are described in the following section.

C.1.7.1 GET: Returns a specific data set of an allocated physical container

This service returns a specific data set of an allocated physical container. The data set's name is specified in the path.

Request
Content-Type: application/json

Name	Located in	Required	Description	Schema
containerName	path	yes	The name of the physical container where the data set should be returned.	string
physicalSetName	path	yes	The name of the data set which should be returned.	string

Response
Content-Type: application/json

Status Code	Reason	Response Model
200	Successful operation	PhysicalSet
400	There was an error in the request.	-
404	The container or the data set was not found.	-
500	An unknown error occurred.	-

C.1.7.2 POST: Adds a specific data set to an allocated physical container

This service adds a specific data set to an allocated physical container.

Request
Content-Type: application/json

Name	Located in	Required	Description	Schema
containerName	path	yes	The name of the physical container where the data set should be added.	string
physicalSetName	path	yes	The name of the data set which should be added.	string
physicalSet	body	yes	The data set.	PhysicalSet

Response
Content-Type: application/json

Status Code	Reason	Response Model
400	There was an error in the request.	-
404	The container was not found.	-
409	The data set already exists.	-
500	An unknown error occurred.	-

C.1.8 /container/allocate/{containerName}/version/{versionName}

The path "/container/allocate/{containerName}/version/{versionName}" contains a POST operation. The operation of the path "/container/allocate/{containerName}/version/{versionName}" is described in the following section.

C.1.8.1 POST: Adds a physical version to an allocated physical versioned container

This service adds a specific physical version to a physical versioned container. This service should be used when a logical version was already created; however, the physical version was not deployed at this location. If the version is the current version then data sets are taken over by the allocated physical versioned container.

Request
Content-Type: application/json

Name	Located in	Required	Description	Schema
containerName	path	yes	The name of the physical versioned container where the physical version should be added.	string
versionName	path	yes	The name of the physical version which should be added.	string
physicalVersion	body	yes	The physical version.	PhysicalVersion

Response
Content-Type: application/json

Status Code	Reason	Response Model
400	There was an error in the request.	-
404	The physical versioned container was not found at this location.	-
409	The physical version already exists.	-
500	An unknown error occurred.	-

C.1.9 /container/allowedtoallocate

The path "/container/allowedtoallocate" contains a GET operation. The operation of the path "/container/allowedtoallocate" is described in the following section.

C.1.9.1 GET: Returns all containers that are allowed to be allocated

This service returns the containers that are allowed to be allocated. A container is allowed to be allocated when it is not allocated at this specific location and the container is either the highest hierarchy level of a composition or an elementary container that is not part of a composition. Solely the containers' names and types are returned.

Response

Status Code	Reason	Response Model
200	Successful operation	array[LogicalVersionedContainer]
500	An unknown error occurred.	-

C.1.10 /container/notallocate

The path "/container/notallocate" contains a GET operation. The operation of the path "/container/notallocate" is described in the following section.

C.1.10.1 GET: Returns all not allocated containers

This service returns all containers that are not allocated at this location. Every registered container will be taken into account; no matter if the container is part of a composite. Solely the containers' names and types are returned.

Response

Status Code	Reason	Response Model
200	Successful operation	array[LogicalVersionedContainer]
500	An unknown error occurred.	-

C.1.11 /container/search/{informationNeed}

The path "/container/search/{informationNeed}" contains a GET operation. The operation of the path "/container/search/{informationNeed}" is described in the following section.

C.1.11.1 GET: Returns all containers that match an information need

This service returns all containers that match the specified information need. With the information need, the containers can be filtered by entity type, annotation type, and by facets.

Request
Content-Type: application/json

Name	Located in	Required	Description	Schema
informationNeed	path	yes	Contains the needed data for filtering the containers.	InformationNeed

Response
Content-Type: application/json

Status Code	Reason	Response Model
200	Successful operation	array[LogicalVersionedContainer]
400	There was an error in the request.	-
500	An unknown error occurred.	-

C.1.12 /container/{containerName}

The path "/container/{containerName}" contains a GET and a POST operation. The operations of the path "/container/{containerName}" are described in the following section.

C.1.12.1 GET: Returns a specific logical container

This service returns the logical container specified by the container name. This service loads all properties of a logical container as well.

Request

Name	Located in	Required	Description	Schema
containerName	path	yes	The name of the logical container which should be returned.	string

Response

Status Code	Reason	Response Model
200	Successful operation	LogicalVersionedContainer
400	There was an error in the request.	-
404	The logical container was not found by the given container name.	-
500	An unknown error occurred.	-

C.1.12.2 POST: Adds a logical and the corresponding physical versioned container

This service adds a container pair (a logical versioned container and its corresponding physical versioned container) to the location.

Request
Content-Type: application/json

Name	Located in	Required	Description	Schema
containerName	path	yes	The name of the container which should be added.	string
containerPair	body	yes	The logical and the physical container.	PhysicalLogicalContainerPair

Response
Content-Type: application/json

Status Code	Reason	Response Model
400	There was an error in the request.	-
409	The logical container already exists.	-
500	An unknown error occurred.	-

C.1.13 /container/{containerName}/rootcontainer

The path “/container/{containerName}/rootcontainer” contains a GET operation. The operation of the path “/container/{containerName}/rootcontainer” is described in the following section.

C.1.13.1 GET: Returns the root container of a composite

If the container is part of a composite, it will return the root container. Otherwise it will return an empty JSON.

Request

Name	Located in	Required	Description	Schema
containerName	path	yes	The name of the container of which the root container should be returned.	string

Response

Status Code	Reason	Response Model
200	Successful operation	LogicalVersionedContainer
400	There was an error in the request.	-
404	The container from which the root container should be returned was not found by the given container name.	-
500	An unknown error occurred.	-

C.1.14 /container/{containerName}/version

The path "/container/{containerName}/version" contains a GET operation. The operation of the path "/container/{containerName}/version" is described in the following section.

C.1.14.1 GET: Returns the logical versions of a logical versioned container

This service returns all logical versions of a specific logical versioned container with all its details. It returns the information from the logical model as well as the information from the physical model.

Request

Name	Located in	Required	Description	Schema
containerName	path	yes	The name of the logical versioned container where the logical versions should be returned.	string

Response

Status Code	Reason	Response Model
200	Successful operation	array[PhysicalLogicalVersionsPair]
400	There was an error in the request.	-
404	The logical versioned container was not found by the given container name.	-
500	An unknown error occurred.	-

C.1.15 /container/{containerName}/version/{versionName}

The path "/container/{containerName}/version/{versionName}" contains a GET operation. The operation of the path "/container/{containerName}/version/{versionName}" is described in the following section.

C.1.15.1 GET: Returns a specific physical version of an allocated physical versioned container

This service returns a specific physical version of an allocated physical versioned container.

Request

Name	Located in	Required	Description	Schema
containerName	path	yes	The name of the physical versioned container where the physical version should be returned.	string
versionName	path	yes	The name of the physical version.	string

Response

Status Code	Reason	Response Model
200	Successful operation	PhysicalVersion
400	There was an error in the request.	-
404	The physical versioned container or the physical version was not found.	-
500	An unknown error occurred.	-

C.1.16 /metadata/administrativ

The path "/metadata/administrativ" contains a GET operation. The operation of the path "/metadata/administrativ" is described in the following section.

C.1.16.1 GET: Returns all registered administrative metadata

This service returns all registered administrative metadata of the SCMS.

Response

Status Code	Reason	Response Model
200	Successful operation	array[AdministrativeMetadata]
400	There was an error in the request.	-
500	An unknown error occurred.	-

C.1.17 /metadata/administrativ/{administrativeName}

The path "/metadata/administrativ/{administrativeName}" contains a POST operation. The operation of the path "/metadata/administrativ/{administrativeName}" is described in the following section.

C.1.17.1 POST: Registers an administrative metadata

This service registers an administrative metadata in the SCMS. Each registered administrative metadata can be used to describe the content of a container.

Request
Content-Type: application/json

Name	Located in	Required	Description	Schema
administrativeName	path	yes	The name of the administrative metadata which should be added.	string
administrativeMetadata	body	yes	The administrative metadata which should be added.	AdministrativeMetadata

Response
Content-Type: application/json

Status Code	Reason	Response Model
400	There was an error in the request.	-
409	The administrative metadata already exists.	-
500	An unknown error occurred.	-

C.1.18 /metadata/facet

The path "/metadata/facet" contains a GET operation. The operation of the path "/metadata/facet" is described in the following section.

C.1.18.1 GET: Returns all registered facets

This service returns all registered facets. A facet belongs to a specific OWL ontology.

Response

Status Code	Reason	Response Model
200	Successful operation	array[Facet]
500	An unknown error occurred.	-

C.1.19 /metadata/facet/{facetName}

The path "/metadata/facet/{facetName}" contains a POST operation. The operation of the path "/metadata/facet/{facetName}" is described in the following section.

C.1.19.1 POST: Registers a facet

This service registers a facet in the SCMS. A facet is a class of an ontology which represents a temporal, a spatial, or a semantic characteristic.

Request
Content-Type: application/json

Name	Located in	Required	Description	Schema
facetName	path	yes	The name of the facet which should be added.	string
facet	body	yes	The facet which should be added.	Facet

Response
Content-Type: application/json

Status Code	Reason	Response Model
400	There was an error in the request.	-
409	The facet already exists.	-
500	An unknown error occurred.	-

C.1.20 /metadata/facet/{facetName}/concept

The path “/metadata/facet/{facetName}/concept” contains a GET operation. The operation of the path “/metadata/facet/{facetName}/concept” is described in the following section.

C.1.20.1 GET: Returns the concepts of a facet from an OWL ontology

This service returns all possible concepts for a facet of a specific OWL ontology. This means that all child classes of the facet defined in the OWL ontology are returned.

Request

Name	Located in	Required	Description	Schema
facetName	path	yes	The name of the facet where the concepts should be returned.	string

Response

Status Code	Reason	Response Model
200	Successful operation	array[Concept]
400	There was an error in the request.	-
404	The facet was not found.	-
500	An unknown error occurred.	-

C.1.21 /metadata/ontology

The path “/metadata/ontology” contains a GET operation. The operation of the path “/metadata/ontology” is described in the following section.

C.1.21.1 GET: Returns all stored OWL ontologies

This service returns all stored OWL ontologies of the SCMS.

Response

Status Code	Reason	Response Model
200	Successful operation	array[Ontology]
400	There was an error in the request.	-
500	An unknown error occurred.	-

C.1.22 /metadata/ontology/{ontologyName}

The path “/metadata/ontology/{ontologyName}” contains a POST operation. The operation of the path “/metadata/ontology/{ontologyName}” is described in the following section.

C.1.22.1 POST: Stores an OWL ontology

This service stores an OWL ontology in the SCMS. It is stored in the logical registry as a named graph.

Request
Content-Type: application/json

Name	Located in	Required	Description	Schema
ontologyName	path	yes	The name of the OWL ontology which should be added.	string
ontology	body	yes	The OWL ontology which should be added.	Ontology

Response
Content-Type: application/json

Status Code	Reason	Response Model
400	There was an error in the request.	-
409	The OWL ontology already exists.	-
500	An unknown error occurred.	-

C.1.23 /metadata/ontology/{ontologyName}/facet

The path “/metadata/ontology/{ontologyName}/facet” contains a GET operation. The operation of the path “/metadata/ontology/{ontologyName}/facet” is described in the following section.

C.1.23.1 GET: Returns facets of an OWL ontology

This service returns all possible facets of a specific OWL ontology.

Request

Name	Located in	Required	Description	Schema
ontologyName	path	yes	The name of the OWL ontology where the facets should be returned.	string

Response

Status Code	Reason	Response Model
200	Successful operation	array[Facet]
400	There was an error in the request.	-
404	The OWL ontology was not found by the given ontology name.	-
500	An unknown error occurred.	-

C.1.24 /service

The path “/service” contains a GET operation. The operation of the path “/service” is described in the following section.

C.1.24.1 GET: Returns all registered services

This service returns all services that are registered by the SCMS with their service providers.

Response

Status Code	Reason	Response Model
200	Successful operation	array[LogicalService]
400	There was an error in the request.	-
500	An unknown error occurred.	-

C.1.25 /service/{serviceName}

The path “/service/{serviceName}” contains a GET and a POST operation. The operations of the path “/service/{serviceName}” are described in the following section.

C.1.25.1 GET: Returns a specific registered service

This service returns the specified service if it is registered in the SCMS.

Request

Name	Located in	Required	Description	Schema
serviceName	path	yes	The name of the service which should be returned.	string

Response

Status Code	Reason	Response Model
200	Successful operation	LogicalService
400	There was an error in the request.	-
404	The specified service was not found.	-
500	An unknown error occurred.	-

C.1.25.2 POST: Registers a service

This service registers a service in the SCMS.

Request
Content-Type: application/json

Name	Located in	Required	Description	Schema
serviceName	path	yes	The name of the service which should be added.	string
logicalService	body	yes	The service which should be added.	LogicalService

Response
Content-Type: application/json

Status Code	Reason	Response Model
400	There was an error in the request.	-
409	The service already exists.	-
500	An unknown error occurred.	-

C.2. Definitions

C.2.1 InformationNeed

This represents the data which is needed to query the SCMS for containers.

Name	Description	Schema
semanticLabelToFacets	This represents the descriptive metadata (an array of facets with the corresponding concepts) with which the containers are filtered.	array[SemanticLabelToFacet]
entityType	This represents the entity type with which the containers are filtered. If an entity type is given, only the containers which have the same entity type are returned (except for the entity type “NONE”).	EntityType
annotationType	This represents the annotation type with which the containers are filtered.	AnnotationType

C.2.2 AnnotatedElementaryLogicalVersionedContainer

Derived from: ElementaryLogicalVersionedContainer

This represents an elementary logical versioned container that is annotated. An annotation can be applied to a container in addition to an entity type in order to describe the container's content more precisely.

Name	Description	Schema
hasAnnotationType	This represents the annotation type with which the container is annotated.	AnnotationType

C.2.3 AnnotationType

Derived from: DataItemType

An annotated elementary logical versioned container has an entity type and an annotation type. For example, it can have the entity type “NOTAM” and the annotation type “Importance”. An annotation type represents a subset of an entity type.

C.2.4 CompositeLogicalVersionedContainer

Derived from: LogicalVersionedContainer

A composite logical versioned container is a special type of logical versioned container. It consists of one or more logical versioned container.

Name	Description	Schema
components	These are the components of the composite logical versioned container.	array[LogicalVersionedContainer]

C.2.5 DataItemType

An elementary versioned container and a homogenous versioned container have a well-defined entity type that is used to specify the type of data items they are allowed to contain. To specify the allowed data items more closely, an annotation type can be used. The data item type represents either an annotation type or an entity type.

Name	Description	Schema
className	This represents the name of the data item type (annotation type or entity type).	string

C.2.6 ElementaryLogicalVersionedContainer

Derived from: LogicalVersionedContainer

An elementary logical versioned container contains the metadata of a container. In addition to the metadata of a logical versioned container, the entity type must be specified. Its physical counterpart stores the actual data.

Name	Description	Schema
entityClass	This refers to the entity type of the data items the container is allowed to contain.	EntityType

C.2.7 EntityElementaryLogicalVersionedContainer

Derived from: ElementaryLogicalVersionedContainer

This represents an entity elementary versioned container of the logical model.

C.2.8 EntityType

Derived from: DataItemType

The entity type specifies the type of data items an elementary versioned container or a homogenous versioned container is allowed to contain. The entity type can be grouped into subsets by using annotations. There might exist multiple annotation types for an entity type.

Name	Description	Schema
hasAnnotationTypes	This list references the annotation types that are used to divide the entity type into subsets.	array[AnnotationType]

C.2.9 HeterogenousCompositeLogicalVersionedContainer

Derived from: CompositeLogicalVersionedContainer

A heterogenous composite logical versioned container is a composite logical versioned container that must be composed of containers of different entity types.

C.2.10 HomogenousCompositeLogicalVersionedContainer

Derived from: CompositeLogicalVersionedContainer

A homogenous composite logical versioned container is a composite logical versioned container that must be composed of containers of the same entity type.

Name	Description	Schema
entityClass	This is the entity type of a homogenous composite logical versioned container. It is not required because it is given by its components.	DataItemType

C.2.11 LogicalVersionedContainer

This represents a versioned container in the logical model. A logical versioned container contains the metadata of a container but not its real data sets. The data sets are deposited in a physical versioned container. For every logical versioned container, at least one physical versioned container must exist at any location.

Name	Description	Schema
isComponentOf	-	LogicalVersionedContainer
className	This represents the class ID of a logical versioned container. It contains the container's name.	string
primarySource	If the container is derived from a source (or from other containers by using a service), their primary source is specified here.	LogicalVersionedContainer
secondarySources	If the container is derived from a source (or from other containers by using a service), their secondary sources are specified here.	array[LogicalVersionedContainer]
primaryAllocation	This property stores the location where the container is primarily allocated.	Location
secondaryAllocations	This property stores the locations where the container is secondary allocated.	array[Location]
pushUpdatesTo	This property stores where the container's primary allocation should push its updates to (to which secondary allocations).	array[Location]
semanticLabel	The semantic label which contains the descriptive metadata of the container, e.g., information about the contained contents.	SemanticLabel
hasAdministrativeMetadata	The administrative metadata store, e.g., the data quality of the container.	array[LogicalVersionedContainerToAdministrativeMetadata]
versions	This represents the container's versions. In this prototype, every time a new data set is added a new version is generated.	array[LogicalVersion]
currentVersion	This represents the container's current version.	LogicalVersion

C.2.12 DegeneratedPhysicalSet

Derived from: PhysicalSet

A degenerated data set exists whenever a data set contains data items which might be not of the desired data quality or are not issued by the desired service.

C.2.13 DegeneratedPhysicalVersion

Derived from: PhysicalVersion

A degenerated physical version is a physical version that contains at least one degenerated set. A degenerated physical version is not represented in the logical model by a logical version.

C.2.14 LogicalVersion

This represents a container's version in the logical model. For every regular physical version at the container's primary allocation a logical version exists. The version's data sets are referenced in the physical model.

Name	Description	Schema
versionName	This represents the name of the logical version.	string

C.2.15 PhysicalSet

This class represents a data set. It stores the metadata of a data set which belongs to a container or to a container's version in the physical model. The data set's content is stored in the XML database of the location.

Name	Description	Schema
className	This represents the data set's ID or name.	string
creatingCallOccurrence	This refers to the call which created this data set.	PhysicalServiceCallOccurrence
creationTime	This is the time when the data set was created.	string (date-time)
pathToXML	This is the path to the data set stored in the XML database.	string
content	This represents the content of a data set. It is only populated when the content is loaded from the XML database.	string

C.2.16 PhysicalVersion

This represents a physical version of a container. A physical version is location specific. It references the version's actual data sets that are stored in the XML database of the location.

Name	Description	Schema
versionName	This is the name of a physical version.	string
hasSets	This refers to the data sets a physical version consists of.	array[PhysicalSet]

C.2.17 RegularPhysicalSet

Derived from: PhysicalSet

A set is called a regular physical set whenever it is added regularly to the primary allocation.

C.2.18 RegularPhysicalVersion

Derived from: PhysicalVersion

A regular physical version is a physical version that only contains regular sets.

C.2.19 AdministrativeMetadata

This represents an administrative metadata. This references the type of administrative metadata that can be stored. The administrative metadata are divided into quality and technical metadata. Administrative metadata can be, e.g., the time, when the last update happened.

Name	Description	Schema
className	This is the name and unique ID of an administrative metadata.	string

C.2.20 AnnotatedElementaryPhysicalVersionedContainer

Derived from: ElementaryPhysicalVersionedContainer

An annotated elementary physical versioned container is a container which represents the physical counterpart of an annotated elementary logical versioned container.

C.2.21 CompositePhysicalVersionedContainer

Derived from: PhysicalVersionedContainer

A composite physical versioned container is a special type of physical versioned container. It consists of one or more physical versioned container. A composite physical versioned container represents an indivisible unit. Whenever a composite physical versioned container is allocated it must be allocated at once; this means that the components cannot be allocated separately.

Name	Description	Schema
components	These are the components of a composite physical versioned container.	array[PhysicalVersionedContainer]

C.2.22 Concept

A concept represents a characteristic of a facet. A concept must be taken from an ontology where it is represented as a class. In this ontology, the concept's class must be a child of the facet's class.

Name	Description	Schema
conceptName	This is the name of a concept. The concept's name must correspond to the class's name in the ontology.	string

C.2.23 ElementaryPhysicalVersionedContainer

Derived from: PhysicalVersionedContainer

An elementary physical versioned container is a special type of physical versioned container. A physical elementary container contains the actual data. It stores which data are in the specific container and which path must be used in order to retrieve the data from the XML database.

Name	Description	Schema
basicSet	This represents the basic (initial) set of an elementary physical versioned container. Every container must have an initial content in this prototype.	PhysicalSet
deltaSets	This represents the content of an elementary physical versioned container. It holds the data sets which are contained by this container beside the basic set.	array[PhysicalSet]

C.2.24 EntityElementaryPhysicalVersionedContainer

Derived from: ElementaryPhysicalVersionedContainer

An entity elementary physical versioned container is the physical counterpart of an entity elementary logical versioned container.

C.2.25 Facet

A facet represents a dimension of an ontology. Each facet has its characteristics. A characteristic represents a subclass of a facet in that ontology. A facet is either a semantic, a spatial, or a temporal facet.

Name	Description	Schema
facetName	This is the name or ID of a facet. The facet's name must match the corresponding class name in the ontology.	string
definedBy	This refers to the ontology where the facet was taken from.	Ontology

C.2.26 HeterogenousCompositePhysicalVersionedContainer

Derived from: CompositePhysicalVersionedContainer

A heterogenous composite physical versioned container is the physical counterpart of a heterogenous composite logical versioned container.

C.2.27 HomogenousCompositePhysicalVersionedContainer

Derived from: CompositePhysicalVersionedContainer

A homogenous composite physical versioned container is the physical counterpart of the homogenous composite logical versioned container.

C.2.28 Literal

This class represents a literal. A literal is a value that is represented as a string.

Name	Description	Schema
className	This is the value and the unique name of a literal.	string

C.2.29 Location

This represents a location that runs an instance of the SCMS.

Name	Description	Schema
locationName	This is the unique name of the location.	string
restHost	This is the host of the public rest interface of this location.	string
restPort	This is the port of the public rest interface of this location.	integer (int32)

C.2.30 LogicalVersionedContainerToAdministrativeMetadata

With this class, an administrative metadata can be associated with a container by specifying a literal. A literal represents a concrete value for a type of administrative metadata.

Name	Description	Schema
className	This is the unique identifier for the association class.	string
value	This is the reference to the literal.	Literal
hasAdministrativeMetadata	This is the reference to the type of administrative metadata to which the literal (value) is assigned.	AdministrativeMetadata

C.2.31 Ontology

This class represents an ontology with its content or concepts. Further, this class contains the metadata about the ontology.

Name	Description	Schema
ontologyName	The unique name of an ontology.	string
hasConcepts	These are the concepts which are provided by an ontology. The concepts are the ontology's classes. This field is required when the content is not set.	array[Concept]
content	The content of an ontology (ontology in a specific language). This field is required when the property has concept is not set (e.g., a new ontology is added to the SCMS).	string
contentLanguage	The language in which the content is represented.	string

C.2.32 PhysicalVersionedContainer

The major difference between a physical and a logical container is that a physical container contains the real data whereas the corresponding logical container contains the meta information about it. Whenever a container is allocated at a specific location a physical container is created at that specific location. Therefore, for one logical versioned container multiple physical versioned containers can exist - at maximum one at each location. The information about the existence of a physical container is stored in the physical registry at each location where it is allocated to. The data are synchronized between the locations.

Name	Description	Schema
className	This is the unique name of a physical versioned container. It matches the name of the corresponding logical versioned container.	string
versions	This contains the physical versioned container's versions that are allocated at this location of the SCMS.	array[PhysicalVersion]
currentVersion	This is the current version.	PhysicalVersion

C.2.33 QualityMetadata

Derived from: AdministrativeMetadata

This represents a type of administrative metadata which refers to the data quality of a container's data.

C.2.34 SemanticFacet

Derived from: Facet

This represents a semantic facet. Semantic facets are facets that are neither spatial nor temporal facets.

C.2.35 SemanticLabel

This represents a semantic label for a container. Every container has a semantic label. It stores all facets that are associated with the container. Along with every facet, the for a container selected characteristic - the concept - is stored.

Name	Description	Schema
className	The unique ID of the semantic label.	string
semanticLabelToFacet	The with the semantic label and therefore with the container associated facets. This property can be empty.	array[SemanticLabelToFacet]

C.2.36 SemanticLabelToFacet

This connects a semantic label with a facet. Furthermore, a concept must be specified for every connection. The concept represents the facet's characteristic that is specified for the container that the semantic label belongs to.

Name	Description	Schema
className	This is the unique identifier of a semantic label.	string
descriptiveMetadata	This references the facet.	Facet
facetValue	This represents the facet's characteristic that is specified for the container that the semantic label belongs to.	Concept

C.2.37 SpatialFacet

Derived from: Facet

This represents a spatial facet. Spatial facets are facets that represent a location.

C.2.38 TechnicalMetadata

Derived from: AdministrativeMetadata

The represents a type of administrative metadata which references the technical information of a container, e.g., last update time.

C.2.39 TemporalFacet

Derived from: Facet

This represents a temporal facet. Temporal facets are facets that refer to a point or period of time.

C.2.40 UnknownFacet

Derived from: Facet

This represents an unknown facet. In an ontology, the facet's type is unknown. However, once a facet is added from an ontology to the SCMS the type is known. Therefore, an unknown facet represents a facet candidate in an OWL ontology.

C.2.41 CompositeLogicalService

Derived from: LogicalService

With a composite logical service, one can register a service which consists of multiple subservices. A subservice can either be a composite or an elementary logical service.

Name	Description	Schema
components	The components a composite logical service contains.	array[LogicalService]

C.2.42 CompositeLogicalServiceCall

Derived from: LogicalServiceCall

This represents a composite logical service call on a composite container.

Name	Description	Schema
components	This are the components of a composite logical service call.	array[LogicalServiceCall]

C.2.43 CompositePhysicalService

Derived from: PhysicalService

This represents a composite physical service. It consists of multiple components. A component can either be a composite or an elementary physical service.

Name	Description	Schema
components	The components of a composite physical service.	array[PhysicalService]

C.2.44 CompositePhysicalServiceCall

Derived from: PhysicalServiceCall

This represents a composite physical service call. A composite physical service call refers to a physical service call on a composite container because multiple services are required to execute it – one service call is required for every of the composite container’s containers.

Name	Description	Schema
component	The components of a composite physical service call.	array[PhysicalServiceCall]

C.2.45 ElementaryLogicalService

Derived from: LogicalService

This represents an elementary logical service. An elementary service references the actual service.

C.2.46 ElementaryLogicalServiceCall

Derived from: LogicalServiceCall

This represents an elementary logical service call for an elementary logical service.

C.2.47 ElementaryPhysicalService

Derived from: PhysicalService

An elementary physical service represents a service instance.

Name	Description	Schema
serviceURI	This refers to the location where a physical service is provided.	string

C.2.48 ElementaryPhysicalServiceCall

Derived from: PhysicalServiceCall

An elementary physical service call refers to the possibility that an elementary physical service call occurs at a location and adds, deletes, or modifies a data set.

C.2.49 LogicalService

This class represents a logical service in the SCMS. A service is an application which can add data to a container. The difference between a logical service and a physical service is that a logical service refers to the service whereas a physical service refers the concrete service instance. Every service is provided by a service provider.

Name	Description	Schema
serviceName	This represents the unique name of the service.	string
providedBy	This represents the person or organization that provides the service.	ServiceProvider
realizations	This represents the physical realizations of this services.	array[PhysicalService]
parameter	This represents the parameters which must be specified for each service call.	array[Facet]
serviceCalls	This represents the logical service calls of this service.	array[LogicalServiceCall]

C.2.50 LogicalServiceCall

A logical service call is whenever a service is applied to the container. This means that this class connects a service with a container. If a service is assigned to a container this implies that this service might be used to add data to the referenced container.

Name	Description	Schema
serviceCallName	This represents the unique ID of a service call.	string
serviceCall-Occurrences	This represents the service call's occurrences.	array[LogicalService-CallOccurrence]
parameters	This represents the specific parameters that are applied to every execution of this service call. Every parameter that is defined for a service must be specified here.	SemanticLabel

C.2.51 LogicalServiceCallOccurrence

This references an occurrence of a logical service call. In this prototype, every time a service adds a regular set to a container it is registered as a logical service call. For every logical service call, there must be at least one physical call.

Name	Description	Schema
serviceCallOccurrenceName	This represents the unique name of a service call occurrence.	string

C.2.52 PhysicalService

The difference between a physical and a logical service is that a physical service references the actual location where a service exists. A logical service represents only the existence of a service. Therefore, a logical service can have multiple physical services.

Name	Description	Schema
serviceName	This represents the unique name of a service. It must correlate with the name of the corresponding logical service.	string

C.2.53 PhysicalServiceCall

The difference between a physical and a logical service call is that the physical service call is location specific whereas the logical service call is location independent.

Name	Description	Schema
serviceCallName	This represents the unique name of a physical service call. It must match the name of the logical service call.	string

C.2.54 PhysicalServiceCallOccurrence

This references an occurrence of a physical service call. In this prototype, every time a service adds a data set to a container the process is registered as a physical service call. A physical service call occurrence is a physical service call that adds a data set to a physical container.

Name	Description	Schema
serviceCallOccurrenceName	This represents the unique name of a service call occurrence.	string
occurrenceOf	This represents the service call the occurrence refers to.	PhysicalServiceCall

C.2.55 ServiceProvider

A service provider is an organization or a person (in PROV-DM called entity) that offers a service.

Name	Description	Schema
serviceProviderName	The name of the service provider.	string

C.2.56 PhysicalLogicalContainerPair

This represents a wrapper object for a physical and a logical container. Furthermore, it contains a list of wrapper objects that represent the container's components. It is used whenever it is required to transfer a physical and a logical container at once.

Name	Description	Schema
logicalVersionedContainer	The logical versioned container.	LogicalVersionedContainer
physicalVersionedContainer	The physical versioned container.	PhysicalVersionedContainer
components	This represents the components whenever it is a composite container.	array[PhysicalLogicalContainerPair]

C.2.57 PhysicalLogicalVersionsPair

This represents a wrapper object for a list of physical and a list of logical versions. It is used whenever it is required to transfer the physical and the logical versions at one.

Name	Description	Schema
logicalVersions	This represents a list that contains the logical versions.	array[LogicalVersion]
physicalVersions	This represents a list that contains the physical versions.	array[PhysicalVersion]