Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Fakultät für Mathematik und Informatik
Institut für Informatik
Mensch-Computer Interaktion

Masterarbeit

im Studiengang Human-Computer Interaction
an der Universität Würzburg

SeMBa 2 - The Semantic Media Backend

Implementation and Performance Evaluation of a Remote Backend for Semantic Media Management

vorgelegt von
Eike Isermann, Matrikelnummer: 1734279

am 18. Mai 2017

Betreuer/Prüfer:
Prof. Marc Erich Latoschik, Informatik IX, Universität Würzburg
Dr. Dennis Wiebusch, Angewandte Kognitionspsychologie,
Universität Ulm

Picture taken from http://www.commitstrip.com/en/2014/04/15/the-original-code/

O., P., U.,
Time is valuable.
Thanks for letting me take some.

Eike

Abstract

Semantic technologies offer a solution to annotate data in a machine readable way. The computational capabilities of modern devices allow efficient reasoning upon these annotations to infer additional knowledge. By integrating semantic annotation into an applications storage model, users can annotate and retrieve data more efficiently. The Chair for HCI at the University of Würzburg is currently working on two teaching-related projects, that require the creation of individually related subsets of an asset library. Based on the project requirements, this thesis discusses the current state of semantic file annotation. The results are analyzed and integrated into software requirements for an application storage framework that allows domain specific structural, descriptive and administrative asset annotation. A software architecture is conceptualized, implemented and tested for functional and performance compliance. The resulting **Semantic Media Backend** is an extendable storage framework for developing arbitrary semantic media management applications.

Zusammenfassung

Semantische Technologien erlauben es, im Rahmen des intelligenten Wissensmanagements, Daten maschinenlesbar zu annotieren. Moderne Computer verfügen über die nötige Leistungsfähigkeit, implizites Wissen aus diesen Annotationen abzuleiten. Die Integration semantischer Annotation in anwendungsspezifische Datenmodelle erlaubt es Benutzern, Inhalte effizient zu annotieren und abzufragen. Der Lehrstuhl für HCI an der Universität Würzburg arbeitet zur Zeit an zwei Projekten zur internen Verbesserung der Abläufe beim Erstellen von Lehrmaterialien. Diese Projekte erfordern es, Inhalte einer Medienbibliothek zu individuellen Kollektionen zusammenzustellen und im jeweiligen Kontext miteinander zu verknüpfen. In Bezug auf die Projektanforderungen diskutiert diese Masterarbeit den aktuellen Stand zur semantischen Annotation im Dateimanagement. Die Ergebnisse bieten eine Grundlage zur Entwicklung von Softwareanforderungen für ein Softwareframework zu Datenannotation mit strukturellen, deskriptiven und anwendungsspezifischen Informationen. Es wird eine Softwarearchitektur entworfen, implementiert und auf die Erfüllung der funktionalen und leistungsbezogenen Anforderungen getestet. Das entstandene **Semantic Media Backend** ist ein erweiterbares Softwareframework zur Entwicklung von Anwendungen auf Basis semantischer Dateiverwaltung.

Contents

For reasons of readability the the following formatting rules are used in this work:

Classes are printed in bold sans serif font; *functions* are printed in italic sans serif font and attributes are printed in standard sans serif font. The same applies to the concepts used in ontologies.

List of Figures

List of Code Examples

1 Introduction

> *"In our fragmented age we particularly need to foster the modes of inquiry that seek bridges among ideas. The search for connections - for the relationships among the parts and the whole - is the driving force that advances knowledge."* (Ehrlich and Frey, 1995)

Over the course of millennia, mankind has evolved from autonomous groups of hunter-gatherers to a globalized society, populating the entire earth. Civilizations have established themselves on the realization, that exchange of knowledge and goods between these groups can form a more productive, survivable society. In today's world, a person is rather a single part of an interconnected collective organism than an autonomous individual (Maurer, Sapper, & Vienna, 2001). A main factor of this development is the evolution from fragmented information, that was relayed only within small social groups, to a collective knowledge fostered by written word, libraries and a scholar community. In the digital age, information is available in inexhaustible quantities. With mobile technology and internet access, a search engine delivers not one but millions of answers to a specific question, regardless of time or location. This constant access to knowledge emphasizes a need for the ability to interrelate information, as envisioned earlier by Bush et al. (1945), even further. Demands for people working in the quarternary sector are steadily increasing, reacting to a public knowledge base that doubles in size every two years. With less than ten percent of it being structured, improvements on an individual as well as technical level are imperative (Gantz & Reinsel, 2011).

1.1 Motivation

To cope with the ever growing body of information available at a persons fingertip, concepts and motivation of teaching at schools and universities have shifted over the past decades. Imparting a more conceptual approach of transfer, interrelation and

application has become more important than factual knowledge (Shulman, 1987). At least, reason and logic are capabilities that define and advance mankind. The ability to generate and evaluate knowledge by applying existing concepts, beliefs or practices is crucial to filter and evaluate ever new information available in the digital world. However, human capabilities are far too limited to process these vast amounts of data created every day. While computers may be able to access this data in the fraction of a time, they lack the concepts and understandings to add informational value by reasoning. The field of digital knowledge management is still in its infancy.

First practical attempts to enhance the digitally accessible information that forms the Internet with computer readable metadata have been initiated by Berners-Lee, Hendler, and Lassila (2001). Ontologies – formal conceptualizations of types, properties and their interrelations – facilitate structural, descriptive and administrative annotation of resources, even adding a second layer of what may be called "meta-metadata". Due to the strong formalism of these ontologies, the semantic information can be accessed by computers to infer additional knowledge. Although being beneficial to most people in our society, the idea remains largely unrealized, with only a slowly growing fraction of the World Wide Web being structured (Guha, 2013).

Leaving this grander scale of computers that exploit all publicly available information, semantic markup has possible applications in smaller individual or collaborative contexts. File systems on all common devices are hierarchically ordered, providing a rather physical data model instead of a conceptual representation. Proceedings in the direction of semantic file systems have not succeeded so far. In contrast, semantics are widely used on the individual application level. Photo libraries tag pictures based on their content by accessing embedded metadata or applying image recognition (Sigurbjörnsson & Van Zwol, 2008). Even in software development processes, semantic abstractions of architectural components are used to facilitate reusability and modularity (Wiebusch, 2016). Especially when handling bodies of media, semantic annotation can add value to an application in various use cases, such as filtering or content recommendations. Nevertheless, there is no available storage solution for applications that annotate and manage data based on a specific application domain. Therefore we will develop a framework to create domain specific models that serve data to arbitrary front end applications.

1.2 Example Scenario

The development of this thesis is based on two proposed software applications for the chair of HCI at the University of Würzburg. Despite their differing domains, both scenarios would benefit from a semantic media management solution.

Semantic Media Annotation RepresenTAtion and Presentation System (SMARTAPS)

Isermann (2013a) firstly assessed the challenges that are posed by the lack of automatic knowledge management for teaching applications. Authoring digital teaching materials is a repetitive task with inflexible results. The SMARTAPS toolchain (see Figure 1.1) describes a workflow for automatic generation of teaching materials, appropriate for varying target groups and teaching contexts. Assets are subject to extensive annotation with descriptive, structural and operational metadata. Authors can compose these learning units based on their "meaning in a specific context". Asset compositions then serve as a source for discrete build protocols that create arbitrary output formats. Main purpose of SMARTAPS would be the annotation and composition of presentation slides for reuse in different lectures. This workflow requires a framework for modeling the concepts of a teaching domain and managing resources and their respective semantic annotations. A first proof-of-concept application for managing and annotating teaching resources was developed by Isermann (2013b). The Semantic Media Backend (SeMBa'13) describes structural information of teaching presentations into based on an ontology.

E-learning in Science (EiS)

The EiS project (Korwisi & Mehn, 2016) aims to develop a web based platform for creating, modifying and operating e-learning scenarios. In an ongoing masters thesis, data management and study composer components are conceptualized and implemented. A semantic storage back end is deemed beneficial in designing a data model that supports hypertextual, context aware concatenation of test cases. For researching purposes it needs to be possible to define context-dependent intervention methods that are triggered by specified conditions. Being a web application, EiS requires a network accessible server application that delivers materials on the

fly, rather than SMARTAPS' build protocols. The EiS server application will be implemented in node.js and requires a solution to persist and dynamically access semantically linked test-case suites.

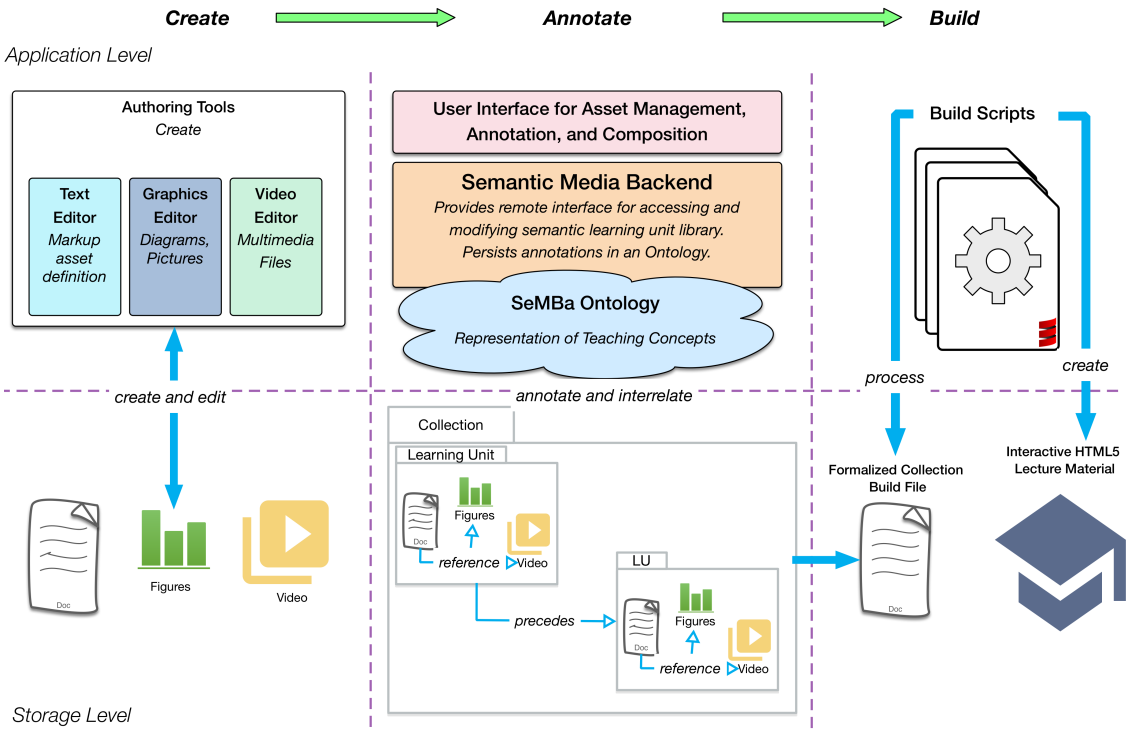


Figure 1.1. The Semantic Media Backend provides the core functionality for managing and composing learning materials with the SMARTAPS toolchain.

1.3 Goal

The main goal of this thesis is to use semantic technologies to enhance the capabilities of media management applications. Therefore a more versatile version of the Semantic Media Backend shall be designed, implemented, and tested. Following the steps of an iterative design cycle, the lessons learned from earlier proof-of-concept implementations (Isermann, 2013b, 2016) are to be reviewed and incorporated.
Modern work environments heavily rely on collaboration and spatial separation of coworkers. Multiple remote users need to be able to work on multiple libraries simultaneously. The chosen architecture of previous SeMBa versions lacked in performance due to its single threaded implementation and ineffective data serialization. Limiting the use of ontologies on collection representation, inhibited efficient search and filtering mechanisms.

Instead of being tightly coupled to a specific use case and graphical user interface, this new approach aims to provide a framework for defining and accessing semantic data models in media management applications. Although the development is inspired by two concrete examples, reusability of the application for future use cases is kept in mind. With the Chair of Human-Computer Interaction taking patronage of the EiS platform, the range of possible applications for a Semantic Media Backend has widened. By fostering synergies between both projects SeMBa will serve data models for EiS, whereas EiS will provide a generalized graphical user interface for dual use in SMARTAPS. This shift calls for a new set of requirements adhering to both stakeholder projects; SMARTAPS and EiS.

To provide an improved back end solution for both projects, the following steps have been identified:

1. Assessment of SeMBas current state, extraction of lessons learned.
2. Research of state of the art approaches for semantic file management.
3. Elicitation of new requirements, based on the lessons learned and impact of the EiS collaboration, with a focus on:
   - Remote access to back end functionality.
   - Performance improvements by introducing concurrent program execution.
   - Reusability of the application for arbitrary use cases.
   - Maintainability of the application architecture to facilitate functional extension of the framework.
4. Redesign and implementation of the back end architecture to fulfill requirements.
5. Evaluation of non-functional performance requirements. Iterative data model improvements if required.

1.4 Structure

This thesis is structured in six chapters reflecting the software design process. Existing approaches, that may facilitate the development of a semantic media back

end solution, are presented and analyzed in Chapter 2. Related work deals with basic principles of providing computer interpretable context on information. Existing levels of semantic file management solutions are presented, ranging from application to file system level. Regarding the concrete use cases, benefits of semantic annotation to knowledge management and existing concepts are elaborated. The results of previous SeMBa iterations are discussed.
In Chapter 3 concrete software requirements are derived after an analysis of project stakeholders and respective use cases.
Following the requirement engineering, Chapter 4 covers the definition of prerequisites for a back end application architecture that fulfills the non-functional requirements. Based on these prerequisites the application design is developed and illustrated. Chapter 5 describes the implementation of this software concept and potentially used external libraries.
Results of the implementation process are evaluated in Chapter 6. It includes testing of functional requirements as well as non functional measures with a focus on application performance.

Results of the thesis are summarized and discussed in Chapter 7. Base on the discussion, possible topics for future work and extension of the framework are given.

2 Related Work

Digital knowledge management has become an important issue for sustaining progress in todays society (Sher & Lee, 2004). While mostly applied to business contexts, teaching and e-learning are to be seen as an important part of knowledge management (Maurer et al., 2001). This chapter shows the general principles and approaches to tackle the growing need of contextualizing information in a teaching domain. It summarizes the previous work and techniques used in implementing a semantic media backend as part of the SMARTAPS project.

2.1 Contextualization of Information

2.1.1 Semantic Web

The transformation of existing information into new and explicit knowledge, requires reasoning about the presented data. One of the hallmarks of mankind is the ability to form connections between the many pieces that comprise all knowledge (Siemens, 2006). A main premise for this process of knowledge combination is the perception of information, i.e. presenting it in a form comprehensible for humans. A discrepancy arises when using digital technology to present these data as such comprehensible presentation for humans may not be interpretable by a machine. The main aspect of digital content creation is providing access and visualization rather than extracting its meaning and providing the means to create additional knowledge based on inference.

Addressing this issue in relation to the World Wide Web, Berners-Lee et al. (2001) described a "Semantic Web" where human-readable content is enriched with information to be processed by software agents. By providing the Resource Description Framework (RDF) to express meaning, a machine-readable state of information is

reached. The standardized syntax of RDF employs statements in the form of subject-predicate-object triplets to establish object to object or object to value relations on existing content. When combining this syntax with a vocabulary that is based on unambiguous Universal Resource Identifiers (URI) for all parts of a statement, software is enabled to make assertions about the type, properties and interrelationships of presented content. Albeit the benefits of such web of interconnected information, especially in the wake of an internet of things, the Semantic Web did not establish itself. Still, less than 10% of the worldwide knowledge is categorized in any way and over 30% of a workday are spent searching for information (Feldman, 2004; Gantz & Reinsel, 2011).

2.1.2 Ontologies

The structural representation of knowledge formed by RDF statements can be seen as a graph. Nodes represent entities (or values) whereas edges describe the relation between these values. An example would be the Wikipedia entry for "Donald J. Trump" having the relation "is President of" to the entry "United States of America". This relation in itself does not offer a base for interpreting machine to provide implicit information. For being able to infer that Donald Trump is a human being, commander of the US forces and must be born in the United States, a conceptualization of the used semantics is required.

> *"An ontology is a formal explicit specification of a shared conceptualization for a domain of interest."*
> (Gruber et al., 1993, p. 199-220)

With a formal, unambiguous description of the existing concepts for a domains terminology, machines can provide inference capabilities that deduct implicit knowledge from explicit statements (Staab & Studer, 2013).

Opposed to the simple tagging of data, the process of engineering meaningful ontologies requires an increased level of maintenance. In many circumstances, human language is ambiguous and words may represent different concepts in varying domains. This limits the possibility of merging domain ontologies. As an example, the aforementioned "is President of" relation in its context implies that the United States are a country and Donald Trump is a statesman. Regarding an ontology that describes corporate structures, the same relation may be applied to CEO of

a company, with fundamentally different implications. This leads to the need for restrictions on the type of concepts a relational assertion may describe. Only in combination with axioms that provide rules to constrain the interpretation of statements, valid knowledge can be inferred (Gruber et al., 1993).
The same applies to different words that express the same concept. While obvious to a human reader, this similarity has to be formally expressed in an axiom to be interpretable.

This high amount of different implications, expresses the complexity of employing ontologies to thoroughly describe a domain.

2.2 Semantic File Management

Concepts of enabling computers to process more of the available data by providing machine readable metadata are not limited to globally accessible information. When moving to the more local environment of PCs and mobile devices, there have been several approaches for organizing data in a more content aware fashion, mostly based on the concepts of Berner-Lee's Semantic Web. Proposed solutions may be roughly categorized as being implemented on a system- or application-wide level.

2.2.1 Application Based File Management

Media applications like Apple Photos use image processing for object and facial detection. Providing a semantic based data organization allows users to browse images based on their contents, location or date (Kwok & Zhao, 2006). In most cases these digital asset management applications use a proprietary library folder that separates all source files as well as the metadata files from the original file system. This limits access to the generated information across different applications, whereas information on a file system level fosters integration into all applications on the system. Although software developers may allow interoperations between their applications, for example a calendar and mail application, these application-conglomerates still remain isolated islands in a sea of data.
Metadata tags such as EXIF (Tesic, 2005) or ID3 (Nilsson, 2000) can be read across different applications, but do only provide human readable semantics. Processing their implications relies on the developers interpretation of such semantics rather

than a formal definition. On one hand, this increases the freedom of using metadata tags for an application specific use case. On the other hand, it limits the reusability of its results.

2.2.2 Semantic File Systems:

Since the introduction of Multics (Corbató & Vyssotsky, 1965), file systems and their respective interfaces in operating systems rely on a hierarchical structure to store and localize files. Limited features for information management in hierarchical file systems force users to devise a custom taxonomy for storing and accessing data. Although the effort for consequently maintaining such structure is comparable to content based knowledge management, it lacks the benefits. A users internal model is rather about *what* kind of file he is looking for, contrary to *where* he would have stored it. Based on this assumption Faubel and Kuschel, 2008 present a solution for automatic retrieval of meta information and path projection into a hierarchical file system based on semantic data.

Personal Information Management (PIM) should facilitate file retrieval by relying on a set of attributes, tailored to the users ability to recall them, instead of the actual physical location. These attributes have been identified to be related to content, visual elements or types (Blanc-Brude & Scapin, 2007). Search engines of modern operating systems generate search indices to provide results based on file metadata but do not take the existing folder taxonomy or context into account. Chirita, Gavriloaie, Ghita, Nejdl, and Paiu, 2005 present a conceptual desktop search architecture that allows informational queries by employing an underlying RDF graph. For example, downloaded files may be linked to corresponding e-mail conversations as a user tends to remember the context in which a file was acquired rather than the location it was put in.

TagFS: Bloehdorn, Görlitz, Schenk, Völkel, et al., 2006 argue that even elaborated directory hierarchies in file system oppose the natural structure of information. Adhering to strict rules in categorization tends to pose to much effort, resulting in frustrated users breaking their commitment. With hard links being the only possibility to cover multiple semantically correct locations of a file, missing support for

different orthogonal dimensions, navigational aid or implicit classification, hierarchical storage is deemed unfit. Bloehdorn et al., 2006 present TagFS, a virtual file system exposed using the WebDAV protocol. By assigning multiple tags that may be structured hierarchically (i.e., "rock" as a subtag of "music"), it is possible to search and organize files using elaborate queries, keeping the level of complexity at the user's discretion. TagFS maintains a storage backend layer with the files being kept in opaque locations on the servers file system as well as a RDF database for storing tag assignments. The TagFS program logic exposes files and dynamically created virtual folders related to the files based on the users query. However, TagFS' usage of RDF is only limited to the creation of structural tag hierarchies. It does not support the creation of object-to-object statements that describe the relations between different classes or instances.

WinFS: Microsoft has been researching new approaches for storing data since the early 1990s. The Object File System (OFS) was bound to bring "Information on your Fingertips" according to a 1994 keynote by Bill Gates. It would bring object orientation to the file system, encapsulating data and meta-information into a single container file. OFS was the foundation of Windows Future Storage (WinFS) (Rizzo, 2004), an extension bringing relational capability to the hierarchical NTFS file system. Still using an inaccessible NTFS stream at the core, WinFS would incorporate the concept of items and interconnectivity for structuring data. Extendable schemata for different document types are used for describing the content and possible relations of these types. By embedding this concept into the file system, all applications can access and interpret these meta information. WinFS has never been released and its current state is unclear, although Bill Gates states the concept is bound to reemerge ("Billionaire baron Bill Gates still mourns Vista's stillborn WinFS," 2013).

2.2.3 Semantic Desktop

> *"A Semantic Desktop is a device in which an individual stores all her digital information like documents, multimedia and messages. These are interpreted as Semantic Web resources, each is identified by a Uniform Resource Identifier (URI) and all data is accessible and queryable as RDF graph. Resources from the web can be stored and authored content*

> *can be shared with others. Ontologies allow the user to express personal mental models and form the semantic glue interconnecting information and systems. Applications respect this and store, read and communicate via ontologies and Semantic Web protocols. The Semantic Desktop is an enlarged supplement to the user's memory."* (Sauermann, Bernardi, and Dengel, 2005)

Introduced by Decker and Frank, 2004 the idea of a "file system agnostic" Semantic Desktop was refined by Sauermann et al., 2005. The concept addresses a paradigm shift from the desktop and paper metaphor currently employed in operating systems to a more information aware interconnected interface. Using standards developed as part of the Semantic Web they propose a standardized framework for desktop applications. All applications that utilize this framework utilize a central RDF repository and semantic search service. Existing data sources are integrated using adapters, user behavior and background information deliver additional contextual information about the personal mental model. Once enforced, this approach of a single controller interface for all semantic applications facilitates cross application, context-aware personal information management. As the underlying model is similar to the Semantic Web, individual knowledge and ideas can be shared to others from the personal desktop computer. The concepts of Sauermann et al., 2005 where put into practice in Handschuh, Möller, and Groza, 2007. The "Networked Environment for Personal, Ontology-based Management of Unified Knowledge" (NEPOMUK) project aims to provide a standardized, operating system and programming language agnostic, framework for semantic data exchange and application integration. NEPOMUK reference implementations were made available under Linux for the K desktop environment (KDE) as well as the Zeitgeist semantic event logger for Gnome.
A proof of concept implementation in Java provides a front end application for file organization, search, automatic recommendation and collaborative work. Information is stored using the Sesame RDF triple store framework. Automatic file content extraction sets a foundation for semantic annotations.

2.2.4 Discussion

While not all reasoning capabilities of an ontology model may be necessary for semantic file management, the formalism of ontologies provides a structured approach

to store and retrieve data. Compared to simple file tagging, ontologies facilitate the extensive definition a domain's concepts. As seen in the description of TagFS, RDF introduces features like sub-classes to form more specific taxonomies. Only with the addition of an ontological layer upon this hierarchy that is formed by tag based annotation, the ambiguity of classifications within these taxonomies can be reflected. The SMARTAPS project aims for a set of interconnected applications to foster collaborative information retrieval, as well as authoring and building of teaching materials. It shares common requirements with NEPOMUK. Therefore, the experiences and concepts should act as a landmark in redesigning the Semantic Media Backend. However, the goal of providing a universal solution that annotates all data used by a system proved to have a negative impact on overall performance, decreasing the popularity of NEPOMUK.

The different levels of OS integration that were presented, lead from novel file systems over system wide virtual file system layers to application specific storage solutions. The development and maintenance of a new file system with an integrated ontology seems to be too complex to be practical at this moment, as seen by the disappearance of WinFS. Virtual file systems such as TagFS or NEPOMUK either limit the usage of ontologies or come with other disadvantages such as the negative performance impact. Application specific solutions do not share this problem but are limited to a single application that makes use of the data.

Another approach would be mixing the concepts of proprietary groups of applications that employ a coherent storage concept, with those of the semantic desktop. A flexible framework for building semantic media management applications, based on interchangeable domain specific ontologies, would fill a gap between both solutions.

2.3 Knowledge Management & Semantically Grounded E-Learning

Solutions presented in section 2.2 have the common goal of dealing with the vast body of fragmented, interconnected information that defines modern mankind. Civ-

ilization is based on a shift from autonomous individuals, struggling to survive on what they can gather and make on their own, to a collective organism that distributes physical work and produce (Maurer et al., 2001).
This shift also occurs regarding the most valuable resource of the modern age: knowledge. Causes for competitive business advantages of organizations can partly be measured in the capability to capture, manipulate and create knowledge (Van Buren, 2002). Yet, knowledge is an elusive asset, strongly tied to the individual possessing it. Its importance for success and progress to organizations and mankind, as well academic interest in the process of sharing and persisting knowledge, established the field of knowledge management. Main points of knowledge management include the global sharing of existing information, utilizing individual expertise and managing innovation to shorten product development cycles and solve problems. These considerations can also be applied to a teaching context by integrating the domains of e-learning, mobile devices, remote collaboration and context-awareness into the process of learning material creation (Maurer et al., 2001).

Material classification based on ontologies facilitates retrieval, interaction and management of teaching assets. Existing taxonomies can be easily extended, fostering the required flexibility for developing fields of research. Classical learning materials can be broken down into atomic, self contained units, enriched with descriptive information about their contents. This process creates reusable learning objects, that may be aggregated into varying new materials (Wiley, 2003).

Tane, Schmitz, Stumme, Staab, and Studer, 2003 present the Courseware Watchdog, an ontology-based tool for finding and organizing learning materials. The Courseware Watchdog framework exploits Edutella, a peer-to-peer network for distributed educational resources enhanced with RDF metadata (Nejdl et al., 2002). By facilitating the KAON ontology framework (Bozsak et al., 2002), these metadata are aggregated into an ontology of the use-case specific domain. The Watchdog enables lecture authors to visualize and browse these asset ontologies. A focused web crawler suggests related materials and highlights changes or updates to the domain. This is possible even for unannotated web resources by matching results against the ontology and applying a relevance score. All key domains of gathering material for fast evolving fields of studies are covered by the framework: *Browsing* model and content, *finding and assessing* relevant material, *querying* based on semantic annotations, *clustering* based on conceptual relations and *evolution* of domain specific

terms (Tane et al., 2003).

However, existing solutions tackle the challenges of gathering sources and complete lectures from large scale resource pools rather than covering the process of authoring semantically connected lectures based on learning objects as aimed by the SMARTAPS project. Stojanovic, Staab, and Studer, 2001 show the concept an e-learning portal providing web based trainings based on the users context that has not been implemented. Nevertheless, the research topics and approaches covered in sections 2.2 and 2.3 suggest, that a back end for such authoring toolchain should make use of existing Semantic Web concepts and domain specific ontologies to create lectures based on atomic learning objects. A steady shift to collaborative, mobile workplaces emphasizes the need for a cross platform remote backend (Maurer et al., 2001).

2.4 Semantic Media Backend 2013

2.4.1 SeMBa'13

In Isermann, 2013a and Isermann, 2013b a proof of concept application for the Semantic Media Backend as part of the SMARTAPS toolchain was presented. Implemented in Scala, SeMBa'13 offered basic functionality to investigate the possibilities and pitfalls of managing and aggregating media files into relational collections.

Figure 4.6 depicts an architectural overview of SeMBa'13. A modular approach, that employs the Model-View-Adapter (MVA) pattern, was chosen for a strong separation of concerns, hence maintainability. Assets are imported into the Media Library module as **MediaItems**. All descriptive and operational information about source files is persisted in an XML library file. **CollectionItems** are stored similarly but reference to an ontology file as source. These ontology files are solely used to map relational links between **MediaItems** that are included in the specific collection. Relations are assigned using a graphical editor panel and mapped to the ontology. All modules and services of the application make use of the Publisher/Subscriber pattern for an event driven architecture. Nevertheless, the application is single threaded and does not provide external access.

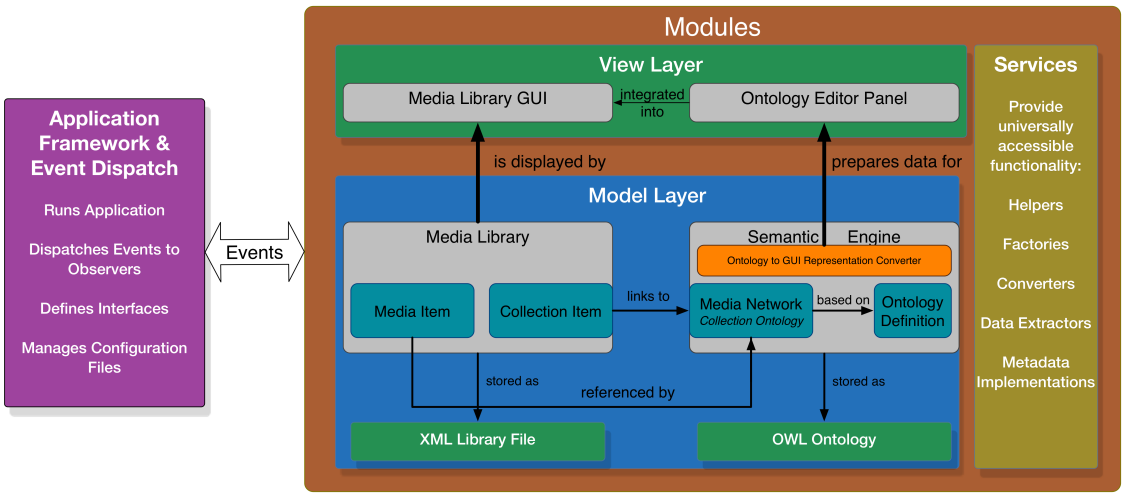


Figure 2.1. The first SeMBa'13 proof of concept application followed a modular design that allowed the implementation of different output modules to present or access the data. Synchronization with the model layer was realized by a Publisher/Subscriber event pattern.

2.4.2 Addition of a Remote API

Subsequent to providing a proof of concept user interface in Semba'13, a bachelor's thesis aiming for a more refined GUI was carried out (Gidt, 2015). Objective of this thesis was to develop editor views for the media library and collection ontologies based on usability design principles. Isermann, 2016 supported this work by implementing an external API for SeMBa'13.

Complex software systems are commonly divided into layers, separating application concerns. The three principal layers are encapsulating presentation, business logic and data source. This separation of concerns facilitates maintainability by fostering the exchange of single layers while keeping the system in its whole operational. Moving from a local GUI application to a new user interface can be achieved by simply adding another presentation layer (Fowler, 2002). When multi-layered applications become multi-tiered, viz. not only logically but spatially separated in the form of client-server relationships, one major challenge needs to be addressed: With layers being executed in different address spaces, control flow between these tiers has to be ensured. Additionally, for purposes of improved programmer productivity and application efficiency, layers might be implemented in different programming languages (Lamb, 1987).

Interface Description Languages (IDL) provide a framework facilitating language agnostic serialization and transfer of structured data (Nestor, Wulf, & Lamb, 1981).

In combination with remote procedure calls (RPC) it is possible to expose back end functionality to client applications. (Birrell & Nelson, 1984)
A remote API for the SeMBaGUI project was implemented using the Apache Thrift framework (Slee, Agarwal, & Kwiatkowski, 2007). It enables the application to request the current state of the media library, perform import/export operations and filter assets based on assigned metadata. Furthermore, an additional metadata model was introduced, allowing users to create custom fields specific to their domain. Metadata entries are not part of the semantic engine, but rather serialized as XML files.

2.4.3 Lessons Learned

Main ideas of the SeMBa'13 implementation turned out to be helpful in providing a back end for the SMARTAPS project. The layered architecture allowed facilitated the replacement of the original GUI with an external API. The separation of front and back end is a reasonable approach for a work environment relying on collaboration and mobility. Structuring media collections as ontologies offers a dynamic taxonomy adjustable to the required domain. Over the course of the implementation a set of improvable design choices emerged:

- A storage solution resting upon up to three separate files per item results in high amounts of file I/O operations. The gain of allowing manual edits of specific library entries is overshadowed in performance issues that arise with growing numbers of items. Additionally, the Apache Jena framework used for handling all relational data struggles in dealing with these high numbers of separate ontologies. This may have various reasons and will be investigated further over the course of this thesis.
- SeMBa'13 has been implemented as a single threaded application. Due to the time-consuming nature of operations like search or file I/O the application gets blocked although a user could work simultaneously.
- SeMBa'13 does not support multiple clients working on one server. Only one media library can be opened at a time. Users might have multiple libraries separating original content from external assets.

- Separating **MediaItem** definitions as non-semantical informations and semantic collections impedes filtering and search functionality. Different approaches are needed for searching the content of collections and filtering library contents. Adjusting metadata to a specific domain would be easier based on fitting taxonomies.
- The main effort in setting up a repository of teaching assets is comprehensible and thorough annotation. SeMBa'13 does not offer automatic extraction and propagation of existing metadata.
- The Apache Thrift framework used for connecting the front to the back end only supports unidirectional method invocation. Although the SeMBa'13 implementation uses an event system to implement the publisher/subscriber pattern, these events can not be propagated to client applications. Clients need to resort to a polling mechanism for being synchronized with the state of the server.

These main drawbacks identified in preliminary work will be considered in the following process of requirement engineering and application design for this thesis.

3 Requirement Engineering

This chapter covers the compilation of requirements for developing a production version of the Semantic Media Backend. Functional and non-functional requirements will be identified and specified based on an elicitation process. This process includes determining stakeholders, analyzing their current challenges, setting up visionary scenarios for tackling these challenges and summarizing the findings.

3.1 Elicitation

The idea of a Semantic Media Backend has originates in the SMARTAPS project described in Chapter 1. Hence, the main focus of development shall lay on the challenges and requirements of this project. The SMARTAPS project - accordingly the chair of HCI at the University of Würzburg - can be identified as a mandatory primary stakeholder (Razali & Anwar, 2011). E-learning in Science is another project carried out by the Chair of HCI, it aims to provide a platform for developing and executing e-learning studies. As a core requirement, both project rely on an ontology that grounds certain concepts to describe the data they present.

1. A *logical* or *descriptive* component, that allows the annotation of individual resources in the form of semantically grounded metadata. Allowing the user to create and access content related information about a resource but also application related information that an application requires to process a resource.
2. A *presentational* or *structural* component, that allows the aggregation and interconnection of resources in a specific context. Allowing the user to create arbitrary semantically grounded sequences of resources.

As described in Section 1.2, there are possible synergies between SMARTAPS and EiS that may be exploited by providing a unified back end solution. This makes EiS a secondary stakeholder. Being intended a sole back end for semantic data management, SeMBa requires a front end for user interaction. These front ends

need to have access to SeMBas functionality. A first graphical user interface will be implemented at the chair of HCI as part of a thesis by Kristof Korwisi. Extended stakeholders include possible developers taking care of extending and improving SeMBa in the future. Over the course of the SeMBa project, problems and demands for these focus groups have been gathered by discussion as well as analyzing the first prototype application.

3.1.1 SMARTAPS Motivation and Goal

Teaching and literature research are main aspects of academic work. Today, digital resources are an important asset in fulfilling these tasks. With growing numbers in resources and co-workers, the need for efficient management of these resources increases. Currently the chair of HCI employs a set of Git repositories for storing resources. The current toolchain heavily relies on LaTeX for authoring presentations. Lectures are composed of modules that form coherent learning units covering one topic. This allows for lecture presentation files to just import a series of different modules forming a comprehensible lecture. The main advantage of this approach is having reusable modules and presentations that automatically adapt to changes in such modules when being rebuild.
Nevertheless this structure comes with certain drawbacks. Organization of files is achieved in a hierarchical folder structure that may not be consistent with a mental model of every user. In combination with a lack of means for indexing and tagging resources this inconsistent model panders duplicate research or authoring of modules. An additional factor is the favored concept of microlearning (Hug, Lindner, & Bruck, 2005). A reduction of granularity of the used modules from at least 15 minutes per module to the more appropriate five to six minutes introduces additional challenges in file organization.
Growing and fluctuating staff in an interdisciplinary chair leads to a number of different approaches to authoring teaching materials. The current toolchain requires sufficient knowledge of LaTeX or willingness to adapt to the technique instead of using preexisting slides (e.g. PowerPoint). An important factor in this regard is the missing concept of "What you see is what you get" (WYSIWYG), inhibiting sequence visualization for presentations. These factors lead to further fragmentation of the knowledge base the chair relies on. Shared by all used authoring tools is missing support for designing multi-dimensional presentations. Certain audiences

may require examples, in-depth explanations or additional information on a learning unit. These relations can neither be represented in LaTeX nor using common presentation software.

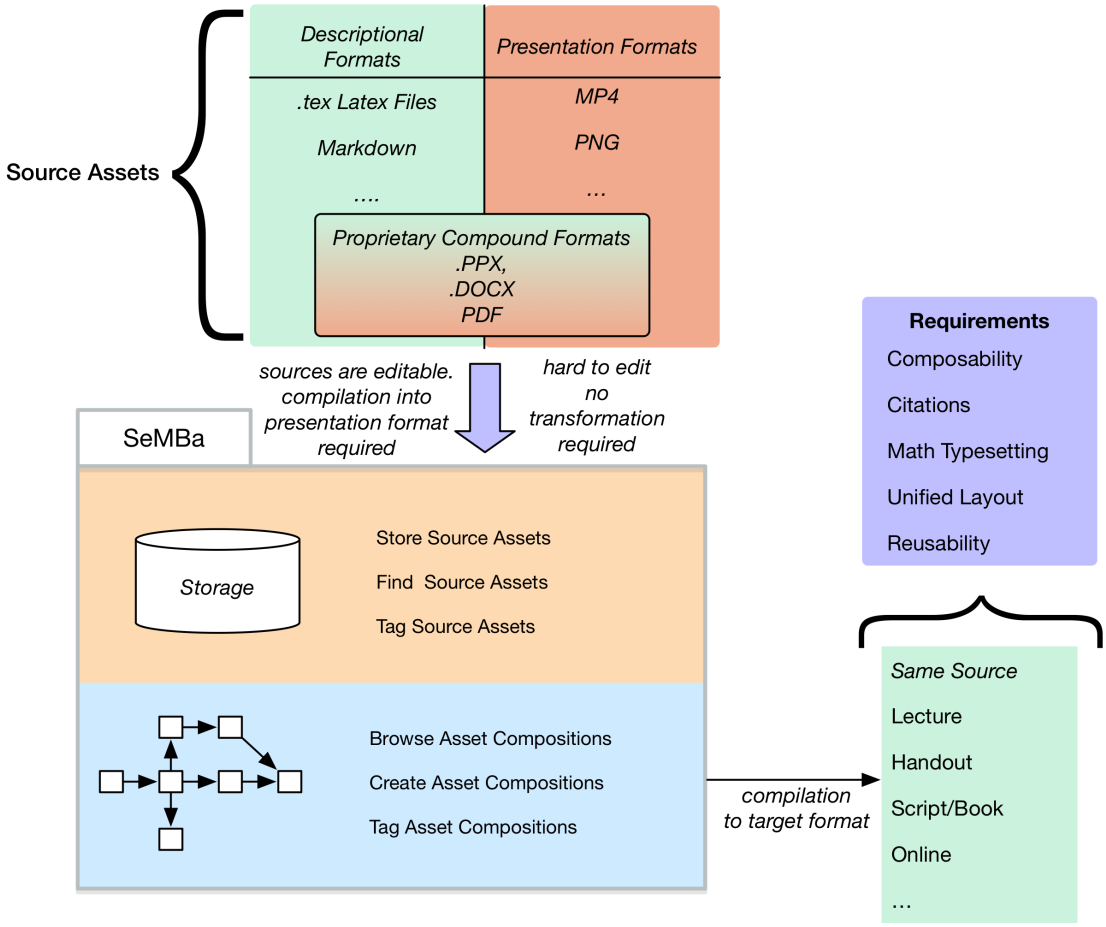


Figure 3.1. SMARTAPS requires a storage and authoring concept that moves from proprietary compound formats to a solution that offers unified management, composition and provision of teaching assets.

As depicted in Figure 3.1 a SMARTAPS toolchain requires SeMBa to provide two main functionalities; semantic *asset management* and semantic *asset composition*. Within these functionalities the current challenges may be addressed. Asset management should facilitate storing, tagging, finding and sharing of resources. Asset composition aims to provide means for creating new sets of resources by introducing semantic relations between existing assets. SeMBa should provide an interface for building various media representations based on these compositions.
Assets may be categorized into descriptional formats, that describe a presentation

format and require transformation to be displayable. An example would be the current toolchain that relies on .tex files which are compiled into PDF slides. Presentation formats are in most cases hard to edit but do not require transformation to be displayed. WYSIWYG formats such as PowerPoint PPX aim to combine these characteristics but require proprietary software. The goal of SMARTAPS is to provide an output format that is at the same time editable and presentable. The current approach is based on a markup language that describes contents in HTML5, meets the requirements stated in Figure 3.1 and can be directly viewed through a browser. When compiling a SeMBa collection into this descriptional/presentable markup format, descriptional assets require transformation whereas presentational assets can be directly embedded.

3.1.2 SMARTAPS Visionary Scenarios

Academic Staff Assembling a Library

Max is a research assistant at the chair of HCI in Würzburg. He is tasked with transferring the existing Latex lectures and modules as well as the referenced media files on real-time interactive systems (RIS) to the Semantic Media Backend. For teaching materials, the chair prefers to import assets that can be described as a coherent learning unit. Max checks if resources can be broken down into smaller bits, and does so if required. When setting up the library, Max chooses an ontology maintained by the chair as a base terminology. This ontology extends the SeMBa base with concepts of learning units, modules, lectures as well as a specified set of custom metadata and relations. After starting the batch import, SeMBa moves the source files into its library structure. Metadata values are automatically extracted from the files and saved. If an extracted metadata tag does not yet exist, it is automatically added to the ontology. While the import is underway Max already starts checking the finished items. He reviews metadata and the thumbnails created for each file. If the library base ontology contains required custom metadata that cannot be extracted automatically, he assigns the values accordingly.

A Lecturer Preparing Materials

As Max is importing files, Dr. Weber is on the train to Würzburg, returning from a conference. She still has to prepare the lecture on RIS for the upcoming semester. The reorganization of the teaching toolchain comes at a convenient time for Dr. Weber. The field she is teaching on evolves rapidly and she already wanted to overhaul the existing lecture for quite some time. Dr. Weber recognizes that her assistant is nearly finished with importing and tagging the existing materials. She queries the semantic back end for source materials she imported during the last months. Her query for **Papers** regarding RIS, that are not referencedBy any **Module** returns a promising paper on multimodal fusion. Using the chairs new markdown language for creating interactive slides, she creates a module on multimodal fusion techniques, imports, and tags it in SeMBa.

By this time Max is finished importing the existing lecture. He recreated the original sequence of slides with simple follows relations using the graphical editor. Dr. Weber decides to add the new module as an excursus to the lectures chapter on virtual reality interfaces. A hasExcursus relation does not exist in the teaching ontology used by the chair yet. She uses the Protégé Editor to add a new concept to SeMBa and reloads the library. Now the modules can be semantically connected in SeMBas graphical editor.

Because she introduced a new relation to the ontology, the build script that creates a presentation from the collection requires an update. Dr. Weber defines the handling of hasExcursus relations during the build process. After running the script, she will be able to react on a motivated audience, by giving a short excursus on the click of a button.

3.1.3 EiS Motivation and Goal

Offering interactive e-learning scenarios is beneficial to the teaching process. It provides an environment where learners can practice and apply their knowledge while teachers are able to assess the results. At the Chair of HCI, the "E-learning in Science" platform is developed as part of an ongoing masters thesis (Korwisi, Work in Progress) . This web application is intended to facilitate semantically backed creation, management and delivery of e-learning scenarios. Users can access

the web service and complete e-Learning scenarios that dynamically adapt to their behavior. Being intended as a platform for research, different intervention methods can be associated with a scenario execution to perform studies. EiS requires a back end that maps its internal data model to a persistent semantic representation.

3.1.4 EiS Visionary Scenario

Liam is a researcher planning to evaluate a new didactic learning method he developed. He adds the concepts of this new method to the domain ontology of the semantic back end. Using the EiS application he composes a user study to measure the success of the new concept. To start his authoring session Liam chooses an existing scenario and loads it into the study composer. As the scenario already contains the semantic information that describe different paths through an e-learning session he doesn't need to edit any structural annotations. He selects the individual components of the scenario and adds an "identical content" relation to its correlating component designed after the new methodology. He now annotates the scenario with a study configuration for comparing methodologies, setting the interval and content of timed feedback prompts. He repeats that step for different scenarios to minimize unwanted effects.
When it is time to execute the study, Liam assigns a methodology to each registered test person. When starting the experiment, EiS queries the semantic back end for scenarios that contain component representations in all designed methodologies. Based on the preset, test subjects are only presented with their respective representation. Interventions are triggered based on the scenario configuration.

3.1.5 Front End Developer Motivation and Goal

Different primary stakeholders and possible additional use cases, for example possible data management for the InterMem project (Gall, Lugrin, Wiebusch, & Latoschik, 2016), inhibit development of an exclusive user interface. Mobile access and different modes of data presentation need to be covered. This capability can be ensured by a modular approach, where the Semantic Media Backend is able interact with various front end modules using a standardized protocol. Specific use case and environments can require different programming languages to be adequately addressed, a remote

back end should be capable of providing functional access and data representations for a broad set of languages.

3.1.6 Front End Developer Visionary Scenario

Being an avid long distance runner, Emily is always looking for ways to enhance this sometimes monotonous activity. She wants her music on the run to match her experience. Her plan is to develop a mobile application that processes her positional and physiological data for providing a situationally adequate soundtrack. She defines her use case of SeMBa as the selection of a specific genre or artist based on this data. As she is a Windows Mobile user, Emily decides to implement her application in C#. SeMBa provides her with a library for setting up a remote client connection and the documentation to grasp the concepts and API functionality she can expect. After choosing SeMBa, Emily enhances an ontology describing music genres she found in the Internet. She adds concepts of different moods, landscapes and levels of exhaustion relating them to each other and the different genres. She implements the client code for tagging her existing media and querying based on sensor data, using the unified representation of library content SeMBa provides. After importing her ontology and configuration to SeMBa she starts developing the user interface of her application.

3.1.7 Future Back End Developer Motivation and Goal

Equivalent to the complexity of the real world concepts it helps to describe, semantic software struggles to cover all aspects of its theoretical capability. This realization led Gennari et al. (2003) to choose a modular, plug-in centered architecture for the popular ontology editor Protege. After becoming a main part of the teaching toolchain at the chair of HCI, new functional requirements are bound to come up. In this context the development of SeMBa, over the course of a master thesis, should be seen as laying a solid foundation for an extendable software solution. Taking this need into account for further development, a main non-functional requirement for SeMBa should be maintainability:

> *"The ease with which a software system or component can be modified to change or add capabilities, correct faults or defects, improve performance*

> *or other attributes, or adapt to a changed environment.* [...] *3. The capability of the software product to be modified."* ISO/IEC, 2010

The possibility for introducing new functional capabilities, fixing software errors and reusing components is important for the long term success of such a platform. Hence, all design decisions regarding the software architecture should be assessed for their influence on software maintainability. Additionally, a comprehensible documentation facilitates access to the concepts and structure of the implementation.

3.2 Analysis

Similar to the stakeholders themselves, the use cases for SeMBa may be separated into two categories.

1. All primary stakeholders are identified as users of a successful SeMBa integration into a working software toolchain. As a module in this set of tools, SeMBa provides the functionality to store and annotate arbitrary media files. Although EiS and SMARTAPS originate in a teaching context, they operate in different domains. This need for a customizable semantic grounding additionally fosters the use in further applications (e.g. section 3.1.5). Annotation is not only needed to describe singular entities, but also to establish relations between them. Collections of files, assembled based on these relations, are provided for compilation by build scripts.
2. The second category of use cases is defined by the developers working with SeMBa (secondary stakeholders). In combination with the lessons learned compiled in section 2.4, this group mainly accounts for non-functional requirements. SeMBa services need to offer an interface to a variety of programming languages to support different software purposes. Performance has to be sufficient for internal storage operations as well as remote interface calls. To ensure the compliance with these requirements performance measurements have to be conducted as part of the development process.

3.3 Design Requirements

Based on the elicitation process in section 3.1, software requirements for a Semantic Media Backend can be defined. For a system intended to be only used in collaboration with a front end, differentiation between functional and non-functional requirements is subject to discussion. As they are main parts of development, concurrency and the API for accessing SeMBa have been designed as functional requirements. Individual requirements for concepts required by the primary stakeholders are not part of the design requirements. They shall be discussed when developing the ontologies configuring SeMBa for the specific use case.

3.3.1 Functional Requirements

Functional Requirements: SeMBa Data Model		
FR-	**1**	**SeMBa relies on a basic predefined terminology to offer its functionality.**
FR-	1.1	All terminology shall be mapped in a base ontology used on all SeMBa libraries.
FR-	1.2	Users shall be able to extend the base ontology adding the respective concepts of the desired domain.
FR-	1.3	A library represents a set of Resources that share a physical and thematic domain.
FR-	1.4	MediaItems are Resources that represent individual files that are part of a library.
FR-	1.5	Collections are Resources that aggregate sets of other Resources which share a specific purpose.
FR-	1.6	The terminology shall define the base classes for descriptive, structural and administrative Resource annotations.

Functional Requirements: SeMBa Data Model			
FR-		1.7	Annotations shall either be represented as object-to-object or object-to-value relations.
FR-	**2**		**Authors shall be able to create new types of annotations and concepts for domain specific libraries.**
Functional Requirements: Resource Annotation			
FR-	**3**		**The system shall offer the possibility to annotate library contents.**
FR-		3.1	Users shall be able to add descriptive metadata to MediaItems and Collections.
FR-		3.2	Descriptive metadata shall be expressed by object-value or object-object relations.
FR-		3.3	Users shall be able to add structural metadata to contents of a collection.
FR-		3.4	Structural metadata shall be expressed by object-object relations that are only valid as part of a specific collection.
FR-		3.5	Users shall be able to add administrative metadata to Resources.
FR-	**4**		**The use of annotations shall be bound to their specific domain.**
Functional Requirements: Resource Management			
FR-	**5**		**The system shall offer the possibility to create new MediaItems.**
FR-		5.1	Users shall be able to import any type of file into SeMBa as a MediaItem.

Functional Requirements: Resource Management			
FR-		5.2	Users shall be able to remove any MediaItem and all of its related information from a library.
FR-		5.3	Users shall be able to trigger the import of multiple files or folders at once.
FR-	**6**	**The system shall offer the possibility to create new Collections.**	
FR-		6.1	Users shall be able to create empty Collection containers.
FR-		6.2	Users shall be able to add any existing MediaItem to a Collection.
FR-		6.3	Users shall be able to add any existing Collection to a Collection.
FR-		6.4	The system shall avert recursive Collections by prohibiting the addition of a Collection to itself.
Functional Requirements: File Import Process			
FR-	**7**	**The system shall offer the possibility to retain the original metadata of imported files.**	
FR-		7.1	Metadata shall be extracted automatically from imported files.
FR-		7.2	Extracted metadata shall be converted into existing descriptive SeMBa annotations.
FR-		7.3	If no SembaMetadata counterpart is available for a given tag, a new one shall be created based on this tag.
FR-		7.4	Users shall be able to limit the range of automatically retrieved metadata tags.

Functional Requirements: File Import Process			
FR-	**8**		**The system shall offer the possibility to create thumbnail images of imported files.**
FR-		8.1	Thumbnails shall be retrieved for at least the following file types: Image, PDF, Text
FR-		8.2	If a file type is not supported, a user-selected default image shall be used.
FR-		8.3	The system shall facilitate the easy integration of thumbnail generators for additional file types.
Functional Requirements: Filtering of Library Contents			
FR-	**9**		**The system shall offer the possibility to apply filters to the contents of a library.**
FR-		9.1	Users shall be able to find Resources based on their descriptive annotations.
FR-		9.2	Users shall be able to find Resources based on their structural annotations inside a Collection.
FR-		9.3	Users shall be able to find Collections based on the MediaItems they contain.
FR-		9.4	Users shall be able to combine multiple search queries into one filter.
FR-		9.5	Users shall be able to find Resources based on free text search inputs.
Functional Requirements: Remote API			
FR-	**10**		**The system shall provide a network accessible API.**

Functional Requirements: Remote API		
FR-	10.1	The API shall provide client code for as many target languages as possible.
FR-	10.2	The API shall provide universal data structures for all required internal concepts: Library, Resources, CollectionContents, Annotations.
FR-	10.3	The API shall provide a self contained data model that provides all information related to a collection and its contents. This model shall facilitate the implementation of custom applications that provide a visual representation of contents or compile them into a presentation format.
FR-	10.4	The API shall provide functions to acquire and modify the current state of the library, based on requirement groups **FR-2**-**FR-3**.
FR-	10.5	The API shall provide an update functionality that notifies to notify front end applications of changes to the library
Functional Requirements: Concurrency		
FR-	**11**	**The system shall provide capabilities for collaborative work.**
FR-	11.1	Multiple users shall be able to work on one library simultaneously.
FR-	11.2	A user shall be able to work on multiple libraries simultaneously.
FR-	11.3	Users shall be able to perform multiple, asynchronous tasks at a time.
FR-	11.4	Users shall be able to work on already imported library items while a batch import of items is being processed.

3.3.2 Non-Functional Requirements

Non-Functional Requirements: Performance			
NFR-	**1**		**The system shall handle libraries that contain 10000 resources within the following boundaries.**
NFR-		1.1	A library shall be initialized within 5 seconds.
NFR-		1.2	Synchronous remote function calls shall be completed in under 200ms.
NFR-		1.3	Remote functions that require more than 200ms shall be defined as asynchronous.
NFR-		1.4	Asynchronous remote function calls shall not block program execution.
NFR-		1.5	Depending on complexity of a remote query, completion time shall be reasonable.
Non-Functional Requirements: Platform Compatibility			
NFR-	**2**		**The system shall be executable on macOS as well as Windows or Linux based computers.**
Non-Functional Requirements: Documentation			
NFR-	**3**		**The developer shall provide a detailed documentation for integration of the remote API.**
Non-Functional Requirements: Maintainability			
NFR-	**4**		**The system shall be based on a modular architecture, fostering later implementation of extended functionality.**
NFR-	**5**		**Application modules that are subject to possible extensions (e.g., storage, data extraction) shall facilitate abstractions for improved reusability.**

Non-Functional Requirements: Maintainability		
NFR-	**6**	**The system shall offer means of behavioral configuration for easy adaption of novel use cases within existing functionality.**

4 Concept

Chapter 3 presented the requirements for an extensible software framework supporting semantic media management. Based on these requirements this chapter illustrates a software architecture for implementing such framework. The first section introduces the basic software techniques to tackle the constraints of a collaborative remote back end. Figure 4.1 shows a proposed layered design that leverages these techniques. The subsequent sections Model Layer, Program Logic Layer and Remote API develop the internal architectures for each proposed layer, to give an overall understanding of the application design.

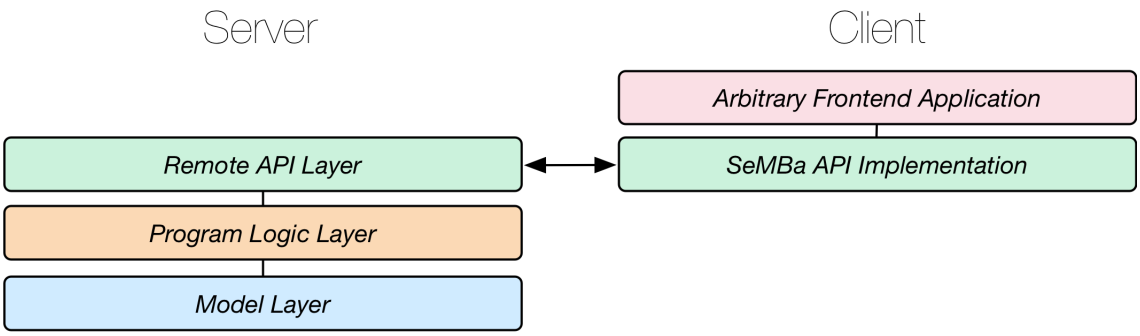


Figure 4.1. SeMBa will rely on a layered client-server model and provide an API connection layer for both tiers of the application.

4.1 Prerequisites

4.1.1 Client-Server Model

The Semantic Media Backend is expected to serve as a data management and service solution for various software applications with different use cases. This requirement dictates a differentiation between content and presentation. SeMBa will not provide a presentation front end, but equip application developers with a client side data model for GUI integration. In software engineering this logical *separation of concerns*

is a proven design principle (E. W. Dijkstra, 1982). Dividing a computer program based on its logical purpose fosters maintainability and reusability (Laplante, 2007). Object-oriented programming languages facilitate this separation through the usage of classes and objects. Spatial separation of front and back end requires some type of client-server application architecture. Introducing a layered design further increases the separation of concerns fostering high cohesion and low coupling (Buschmann, Meunier, Rohnert, Sommerlad, & Stal, 1996), and thusly improving the reusability and maintainability of the product (Microsoft, 2009).

Figure 4.1 shows the proposed multitier client-server architecture for this software project. Communication between tiers will rely on a request-response messaging pattern common for such architectures. Front ends will be implemented as remote tiers that serve as a presentation layer. They may be implemented as rich, smart or thin clients, as long as they adhere to the mandated messaging protocol. SeMBa itself will act as a server application, consisting of all remaining layers required for providing functionality to the user. A *remote API layer* exposes the interface to front ends and other external applications. It defines available message types and procedures. The *program logic layer* incorporates the application facade, manages and contains the library representations and their respective functionality. Data access is achieved via the *model layer*. Concepts for each layer of the back end applications will be illustrated in the upcoming sections.

4.1.2 Actor Model

One of the main requirements for SeMBa is providing services for multiple users to multiple libraries simultaneously. A user may try to perform a search for specific library items, while a second user is importing new media into a different library. If SeMBa relied on single threaded process execution, the search would be blocked until all operations of the file import are finished. Bearable for a single user solution, this behavior is not acceptable for serving more than one client. Introducing multiple threads of execution into a single process offers multiple advantages, especially when relying on extensive I/O operations. Such multi-threaded back end is still responsive to user input as requests are handled in individual threads. On systems with multiple processors or cores load can be distributed, decreasing execution time. Even in cases where operations may take some time nonetheless, a non-blocking server

application is at least able to provide initial feedback, acknowledging the request thusly improving user experience.

Introduced by Hewitt, Bishop, and Steiger (1973), the actor model is a conceptual model for developing concurrent applications.

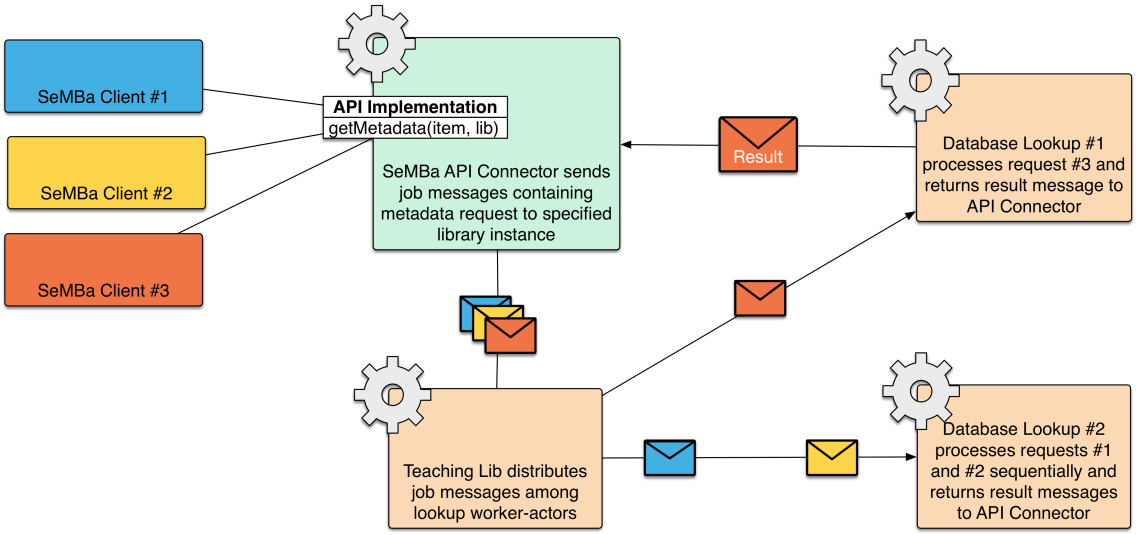


Figure 4.2. The actor model facilitates straightforward modeling of simultaneous client requests, like several data lookups.

Actors are small, self-contained computational units communicating by messages. An actor requires the functionality to create new actors, send messages to other actors and react on received messages. Unlike objects in object-oriented programming languages, there is no shared state between actors. This limits the behavior inflicted by receiving a message to updating the private state and sending a new message, containing the processing results.

When executing each actor in a designated thread this concept allows for multiple operations requested by multiple users to run simultaneously. Besides the possibility to comfortably model a back end, capable of concurrent interaction with multiple clients, the actor model offers a feasible approach concerning additional challenges (Roestenburg, Bakker, & Williams, 2015):

- A layered software architecture, especially when being distributed among multiple tiers, requires layers to communicate using some kind of messaging protocol. An actor based logic layer can be integrated into such message driven design, even allowing distribution of a single layer among different tiers.

- In figure 4.2 a small example shows how a process can be modeled using actors. Mirroring the asynchronous nature of the real world, requests are broken down in smaller individual operations and actors. This natural structure fosters comprehensible application design and fault tolerance (Nash & Waldron, 2016).
- Regarding maintainability of the software, a modular approach is predetermined by the actor model. Low coupling and the absence of shared state facilitate changes to modules even within the different layers, adapting the software to upcoming use cases (Wiebusch, 2016).

The main advantage of an actor based back end is the inherent capability to maintain a concurrent state. As messages are processed serially by each actor, and an actors access is **limited to its own state**, this *single-threaded-illusion* (Nash & Waldron, 2016) prevents race conditions. In systems where different actors require access to the same data, a transactional model that mediates between these actors and the shared state needs to be designed.

Although discouraged, especially when working with external libraries, it's not always possible to design all software components as actors, compromising this concurrent state (Tasharofi, Dinges, & Johnson, 2013). The following section will illustrate the challenges of integrating an actor based logic layer with possible non-actor libraries for handling data and API operations.

Concurrency

As described in the previous paragraph, SeMBa shall serve multiple clients at once. A state of multiple threads, concurrently progressing in their flow of control, can lead to several problems regarding data consistency. Imagine two clients working on the same SeMBa library at once. In a collaborative effort they are authoring a presentation. Right as Client A sets a follows relation between to slides, Client B decides to delete said slide from the collection. Besides the logical implications of these instructions leading to a result where – regardless the order of execution - no relation is created, it could also affect the program itself. Assume the following timeline of process steps:

1. **Thread A:** Retrieve both **CollectionItem** A and **CollectionItem** B from data model.
2. **Thread B:** Retrieve **CollectionItem** B from data model.
3. **Thread A + B:** Check if **CollectionItems** A+B are not NULL.
4. **Thread B:** Send `delete CollectionItem B` to data model.
5. **Thread A:** Send `connect CollectionItem A, Relation C, CollectionItem B` to data model.

Although item B pointed to a valid object in memory at step 3, it is removed from memory before **Thread B** executes its next instruction, leading to a NULL reference. These race conditions, where concurrent operations are executed on a shared memory, lead to nondeterministic behavior. Not always leading to an invalid memory access but rather concurrent write operations overwriting each other, this erratic behavior is hard to detect despite its tremendous impact.

Strictly adhering to the actor model eliminates these issues. Actors do not share state and execute code sequentially. Before processing an incoming message, all reactions to a current message are executed, guaranteeing thread-safe consistency of the internal state.

However, sometimes the effort of adapting parts of a software to the actor model is not feasible (Tasharofi et al., 2013). These cases often concern external libraries for persistent storage of data, i.e. the model layer of an application. In computationally intensive applications a data model could be wrapped into a single actor, serving as a daemon for thread-safe data access while keeping the benefit of concurrent execution. Programs that require constant read access, or rely on other capabilities of a model layer, need an additional solution for concurrent computation. Achieving integrity among shared data has been a research topic since the beginnings of concurrent computing (E. Dijkstra, 1965; E. W. Dijkstra, 1968). Dijkstra proposed the concept of mutual exclusion, ensuring that only one thread may enter a *critical section*, accessing the shared resource, at a time - possibly blocking other threads. Mutual exclusion still is a core concept in coordinating concurrent access to shared resources. The associated issues, such as deadlocks where all resources are being blocked by threads, waiting for each other to complete (see *Dining philosophers problem* (Hoare, 1978)), immensely complicate development of concurrent applications. Since the 1970s these issues have been addressed by several solutions, includ-

ing inter-process communication (Chandy & Misra, 1984), arbitrators coordinating the locks, or partial ordering of resources. Nevertheless, a discussion of techniques for achieving uninterruptible modification of shared state is beyond the scope of this paragraph. It demonstrates the need for a model layer, that is capable of providing read access to multiple processes, while moderating the execution of critical sections through mutual exclusion. Section 4.2 will present a concept for such multiple reader/single writer access scheme of the model layer.

4.2 Model Layer

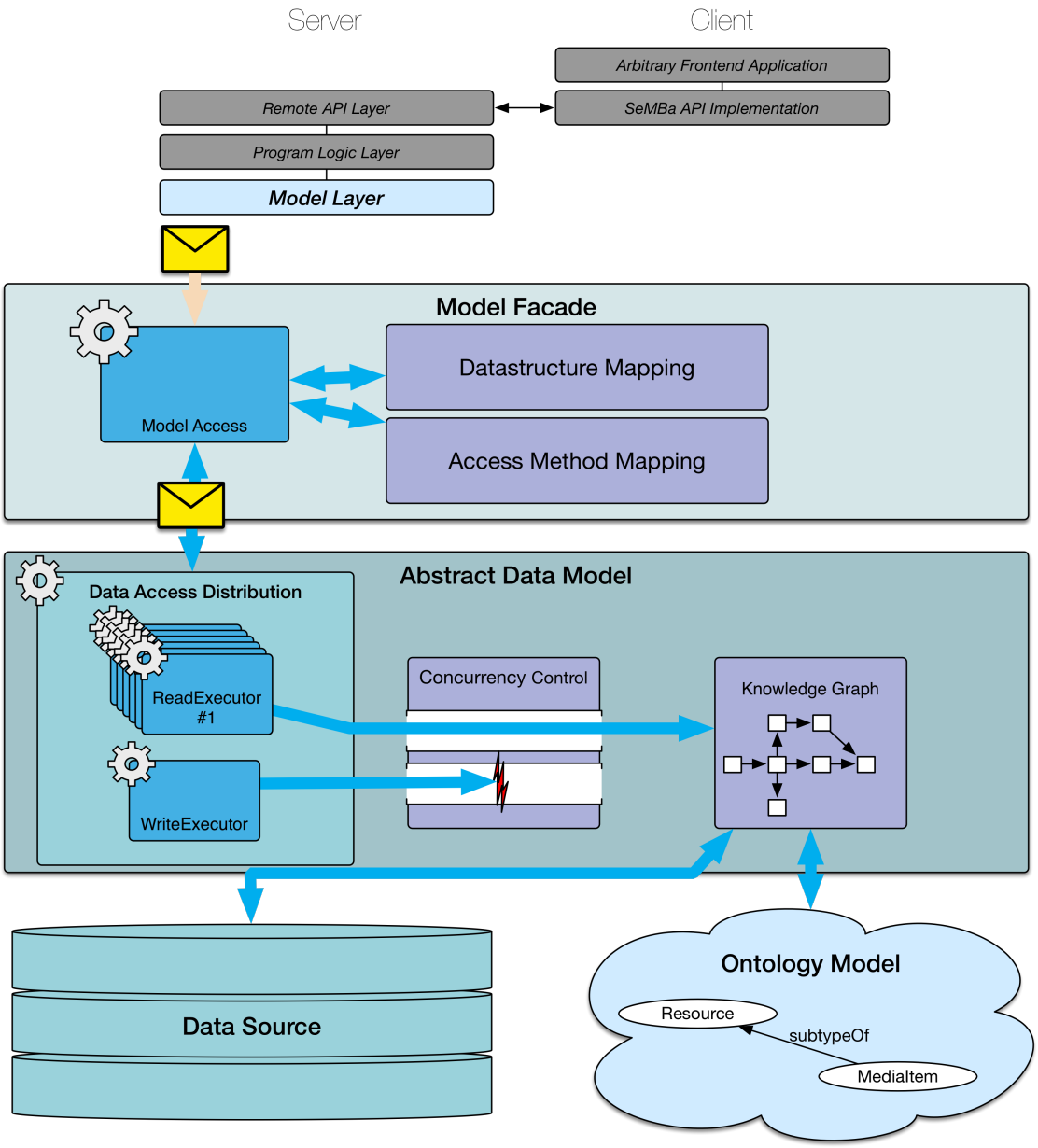


Figure 4.3. Conceptual overview of the SeMBa Model layer, providing access the semantic application state.

As elaborated in chapter 3, and fixated in requirement group **FR-1**, the semantic back end shall rely on an ontology. This ontology provides the terminology for semantically describing general and domain specific attributes of the managed media. The SeMBa model layer consists of four major functional clusters that

1. define a domain specific terminology for the concepts required for a given use case,
2. persist the assertional statements that described the managed media based on the terminology,
3. provide a runtime representation of the persisted ontology including access to reasoning capabilities,
4. provide a model abstraction that facilitates integrating different means of persistent data sources and ensures thread safety.

The internal representation of the ontology will be handled by an external library, which mandates a paradigm change from the actor model to a concurrent, object oriented structure. Figure 4.3 illustrates the structure and control flow of the SeMBa model layer. In the following paragraphs, the main contents of the diagram; Ontology Model, Data Source, Abstract Data Model and the Model Facade are described.

4.2.1 Ontology Model

The terminological component of SeMBas ontology can be separated into two categories. Use-case specific conceptualization that can be arbitrarily defined by the application developer and the SeMBa base model. This base provides the main **Semba Concepts** for the back end implementation. Each library that is managed by SeMBa is represented by a dedicated ontology rooted in this base ontology. **Semba Concept** classes and base attributes and relations are defined as follows.

Semba Concept Class	Covered Reqs.	Description
Resource	FR-1.1 - 1.3	The displayable contents of a library are called **Resource**. Hence, for each **Resource** the ontology has to contain assertions about its name and visual representation. **Resources** may be instances of two different subclasses, **MediaCollection** or **MediaItem**. Items are representations of a single, physical asset that has been imported to the library and contain an additional assertion that describes the source file location. Collections represent unions of **Resources** that aggregate their structural annotations regarding a specific context.
Metadata Type	FR-(3.1, 3.2, 3.5)	Following the W3C guidelines for n-ary relations in the Semantic Web (Noy et al., 2006), **MetadataTypes** are wrapper classes for descriptive object-to-value metadata. For each domain, different subclasses of **MetadataType** can be defined by the author. To annotate a **Resource**, an object-to-object relation to a unique instance of the desired **MetadataType** is added to the model. The instance itself is then annotated with arbitrary object-to-value properties. Relations about the similarity of different **MetadataTypes** to external ontology concepts or the type of data (number, text, date ...) add richness to the model. Two subclasses of **MetadataType** are used to differentiate automatically extracted **GeneratedMetadata** from **CustomMetadata** that are created by an author and specific to a domain.

Semba Concept Class	Covered Reqs.	Description
Collection Item	FR-(1.5, 3.4, 6.2, 6.3)	A **Collection Item** is used to represent an individual occurrence of a **Resource** inside a **MediaCollection**. This wrapping methodology is used to facilitate multiple occurrences of the same **Resource** instance in a single collection. A **CollectionItem** contains the contextual relations between **Resources** at their specific location in the collection, but still reflects all descriptive annotations of the original **Resource**. For example, in a teaching domain, a collection could be a non-linear presentation with multiple paths, that are based on the target audience. In this case the same slide may come up several times in different contexts.
Annotation	**Covered Reqs.**	**Description**
Extendable Annotations	FR-1.6	SeMBa acknowledges three types of relations for describing library contents. **Collection Relations** model the context dependent relationship between **Resources** inside a collection. In contrast, **General Relations** are used to describe universally valid **Resource**-to-**Resource** links. This could be a research paper linking to all its references that are also imported in SeMBa. A third subclass of **Semba Relations**, **Descriptive Relation** is used to link **Resource**-to-**MetadataType** instances. All three types can be subclassed by an author to describe more domain specific relations, for instance to model an *excursus*-**CollectionRelation** inside a teaching presentation.

Annotation	Covered Reqs.	Description
Critical Annotations		Critical are those annotations that are required for running the software and may not be extended or changed. They are either object-to-object or object-to-value relations that describe the physical location of source files, values related to a **MetadataType** or map **CollectionItems** to the **Resource** they represent.

These concepts of the base ontology are mandatory for all libraries modeled in SeMBa. Altering or extending them requires changes to the program logic layer. However, users can build upon this base to achieve the granularity and extent of covered information for a required use case.

4.2.2 Data Source

All of the data created and modified by SeMBa is part of the assertion component of an ontology. Information inside this knowledge graph is expressed in the form of RDF triples, statements adhering to a subject-predicate-object syntax. The data source module is responsible for serializing these statements and persist them on the hard drive.

Approaches for storing RDF data are manifold and include human readable formats such as the original W3C XML-based specification (W3C et al., 2014), formats increasing readability like Turtle (W3C, 2008) as well as binary triple store databases that are optimized for querying large datasets with high performance (Rohloff, Dean, Emmons, Ryder, & Sumner, 2007). As long as it adheres to the interface defined by the abstract data model, a data module is free to implement any kind of storage solution.

4.2.3 Abstract Data Model

In order to achieve uniform access to different data sources an abstract data model is introduced. It defines as a schema for all data source module implementations. It solves the following issues in making the conceptual ontology model accessible to the application, and works in content coupling with the model facade.

Data Representation: Data source modules are expected to provide access to persisted application data in the form of RDF triples. The abstract data model processes these triples to reflect their actual structure. Triples are integrated into a single graph, following the subject-predicate-object syntax where predicates represent graph edges connecting the vertices. The resulting knowledge graph is accessed by the model facade and physically persisted as required.

Concurrency control: As elaborated in section 4.1 the data layer requires some kind of concurrency control. While handling multiple read processes at the same time can be considered thread safe, users are also expected to modify the media library, i.e. the knowledge graph. However, no thread can be allowed to access the data structure while a modification is written into the data structure due to the mentioned concurrency problems.

A common solution for this problem is a multi-reader lock. By adding conditional logic to the sequential behavior of simple mutual exclusion, multiple readers or a single writer are granted access to the data structure. The abstract data model requires data source models to implement locking functionality for read and write processes. This ensures thread safety while fostering performance through concurrent read processes.

4.2.4 Model Facade

This data access module for the program logic layer is designed following the facade pattern (Schmidt, 1999). Functions required to access the data structure are combined into simplified methods. It maps the structure of the graph provided by the abstract data model to the conceptual ontology model. This mapping is achieved in two consecutive steps:

Knowledge Graph to Ontology: The abstract data model's graph structure is wrapped with actual concepts of an ontology. This step introduces first-class objects for classes, their instances and properties, by applying reasoning to the graph statements. It provides functions to modify these object and translates modifications into statements that can be written into the abstract data model.

Ontology to Internal Data Structure: This access component mediates between program logic and data layer. Following the actor model's messaging paradigm it wraps ontology objects into immutable messages for use in SeMBas logic layer and external API. The component provides the internal interface for retrieving and modifying the ontology model. Section 4.4 elucidates the concept of the internal data structure.

4.3 Program Logic Layer

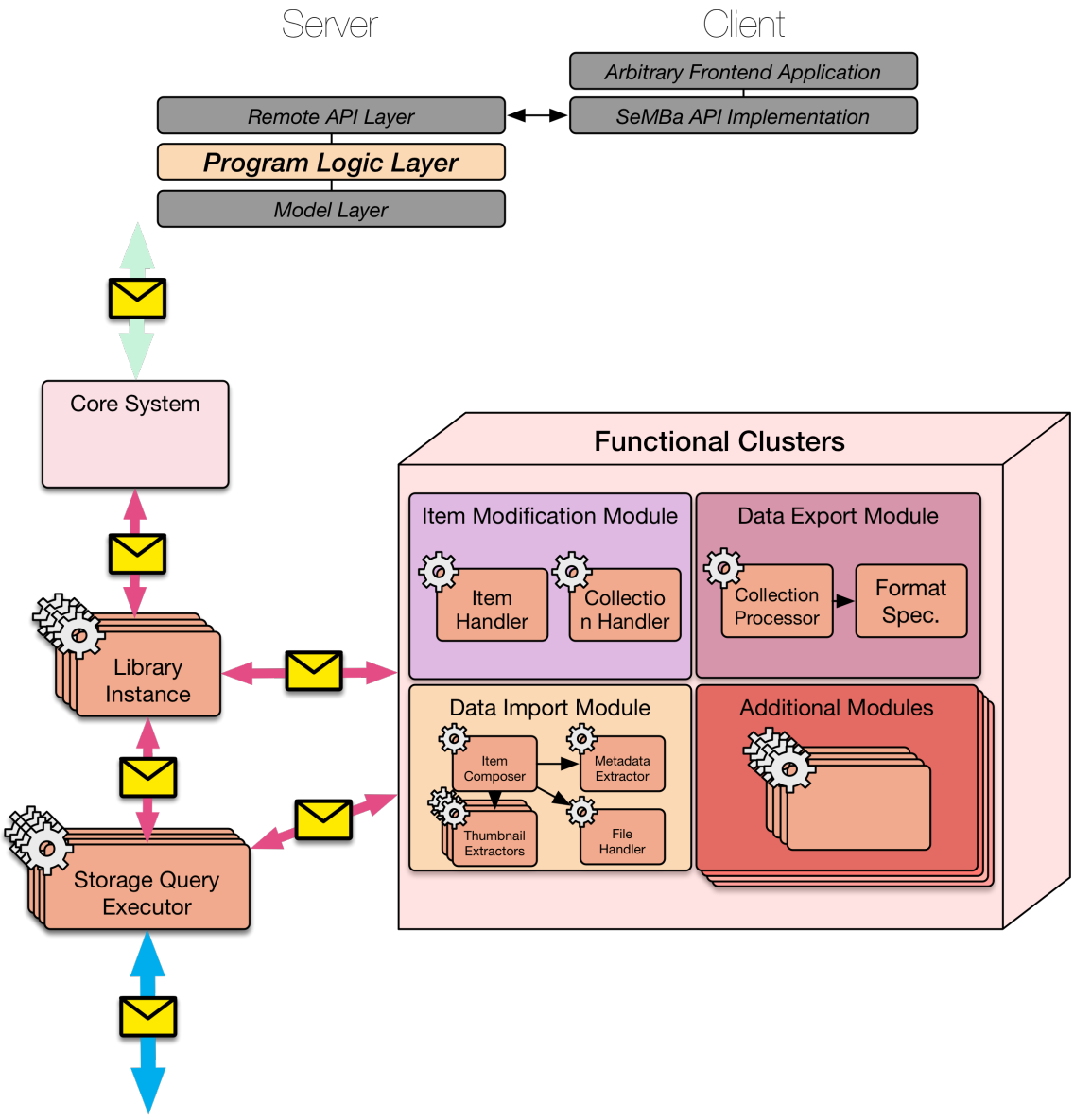


Figure 4.4. Depiction of the SeMBa logic layer. The core system relays requests to the specified library. Requests are either split into subtasks and distributed among functional clusters or directly forwarded to the storage query executor for accessing the data layer.

SeMBas program logic layer implements the core functionality of the system. In compliance with requirement **FR-11** *"The system shall provide capabilities for col-*

*laborative work.*it implements the actor model described in 4.1. Illustrated in Figure 4.5 is the internal structure of the logic layer. The layer's functional clusters and their tasks are designed as follows.

4.3.1 Core System

Serving as an entry point, the core system handles the execution and startup of all software components. After successful initialization of all layers the main core system's main purpose is library and session management. In order to achieve this functionality it keeps track of initialized SeMBa libraries and connected client sessions. This ensures that only one instance of a library is loaded even if multiple clients request loading it. Similarly libraries can be closed after all clients have disconnected.

4.3.2 SeMBa Library

SeMBa libraries represent a single library ontology. A **LibraryInstance** actor exposes the library to the core system and remote API layer. It contains the local configuration of the library and establishes a connection to the model layer. All API calls are forwarded to and handled by the **LibraryInstance** actor. Data access and modification components are modeled as child actors. Depending on its nature, a task is broken down into single tasks forwarded to the respective actors.

4.3.3 Data Import

Importing new media into a SeMBa library includes various differing tasks. Upon receiving the job to import a file, a supervising actor distributes these task among a series of separate components.

File Handling: Resources handled by SeMBa are copied into a proprietary folder structure by default. A file handling component sets up the required directory structure and prevents naming conflicts. Furthermore original file names are used to create URI-compliant resource identifiers.

Metadata Extraction: An extraction component analyzes the imported file and provides the import module with the available metadata values. New metadata tags

are propagated to the data layer to be included as **Generated Metadata** in case the library is not configured to only include predefined metadata keys.

Thumbnail Extraction: To provide graphical user interfaces with iconic file representation a thumbnail component is established. The file is analyzed and forwarded to the **ThumbActor** specified for its content. If the file format is not recognized a generic user defined thumbnail is returned.

Item Composition: Results of the file analysis are gathered by the supervisor and propagated to the data layer. After a successful write to the model, the **LibraryInstance** actor is notified.

4.3.4 Data Export

Data structures serving as a source for the compilation of **MediaCollections** into novel output formats are subject to different requirements than standard API calls. For latency reasons, data structures transmitted to remote clients are partitioned and contain only required information. For example, a **CollectionItem** only carries the reference to a **MediaItem**. An additional query is required to access the **SembaMetadataType** associated with that item. In contrast to this approach, a build script should be provided with all essential descriptive, structural and administrative information about a build source for easy processing. The data export module preprocesses a requested collection and provides the build script with a traversable data structure containing the required information.

4.3.5 Item Modification

The item modification module groups actors that are responsible for all functionality that affects the state of existing library contents. This includes the handling of collections and their contents as well as descriptive annotations of distinct items.

4.3.6 Search

Possible search queries range from string based full-test searches to more elaborated filters based on combining several subject-predicate-object statements of RDF

graphs. The search module decreases the effort for API users and internal development by wrapper functions for frequent query types. It mediates between the graph search syntax of the data layer and user inputs.

4.4 Remote API

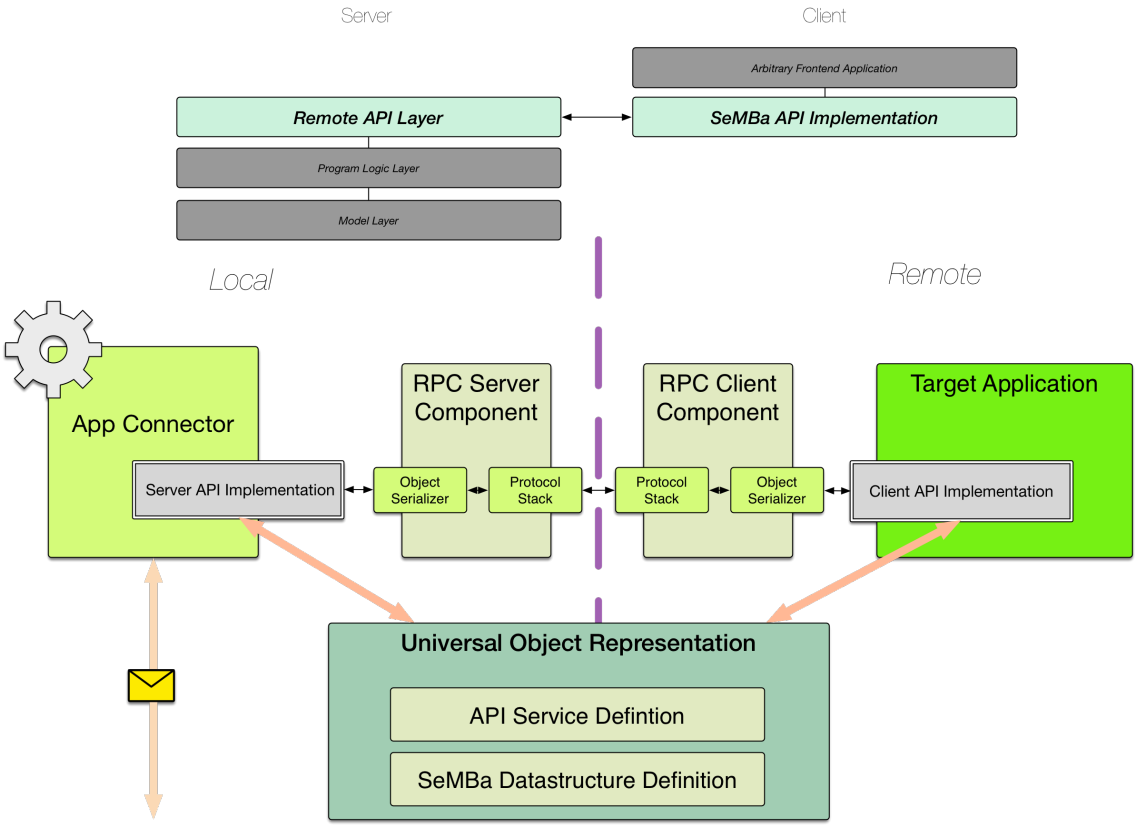


Figure 4.5. The API layer is comprised of client and server implementation providing the SeMBa services. An enclosing AppConnector actor ensures integration into the SeMBa architecture. Defined data structures are accessible from all target platforms.

Requirement group **FR-10** *"The system shall provide a network accessible API."* defines that SeMBa needs to facilitate location-independent access to its services for a wide of programming languages. A client-server architecture, as suggested in Section 4.1, facilitates this remote access. It allows arbitrary front ends to implement the SeMBa client layer and request-response pattern into their application architecture.

Birrell and Nelson (1984) introduced the concept of remote procedure calls (RPC), allowing distributed programs to execute procedures in remote address spaces. Functions exposed by a RPC client appear similar to local implementations. When invoked, a function call is wrapped in a message by a marshalling component and transmitted to the server application. Received messages are unpacked by the server implementation and the respective function is invoked with transmitted parameters. Results are packed and transmitted accordingly.

SeMBa applies this concept to achieve a back end solution that can be comfortably integrated in a wide array of client applications. Figure 4.5 illustrates the internal structure of the Remote API layer, its main components are defined as follows.

4.4.1 RPC Server Component

The RPC server component is responsible for managing connections from client applications. It provides the network protocols, marshalling component and low level implementations for inter-process communication.

4.4.2 Application Connector

Function calls from client applications need to be propagated to the respective **LibraryInstance**. The **Application Connector** is an actor designed to convey function calls and return the results to the RPC server component. Client requests are divided in two categories, those solely accessing the state of the system and those mutating it. Handling of these functions differs substantially.

4.4.3 State Affecting Functions

SeMBa is designed with concurrency in mind. If a client request changes the state of a library, other clients might be currently working on the same data. For this reason the changes of side effecting functions need to be propagated to all affected clients. When opening a library, clients subscribe to an update service maintained by the **Application Connector**. State changing functions do not return the actual result but rather a message acknowledging the validity of the request or an error in case of invalid parameters. Once the request has been processed and results are persisted

the **Application Connector** publishes an update message to all subscribers of the affected library. This approach maintains the consistent global state required for a working concurrent application.

Functions Without Side Effect: Functions without side effects do not need to be handled similarly as their results do not concern other clients. This category includes simple accessor functions as well as filter or search operations on larger amounts of data. When receiving such request it is forwarded to the respective library in the data layer. As waiting for the result of a computationally intensive query could potentially block the **Application Connector**, the method callback is deferred in a separate thread until computation of the result is complete.

4.4.4 Universal Data Representation

To facilitate inter-process communication across different software platforms, a program language-independent approach to model data structures is required. Nestor et al. (1981) introduced a concept providing such common base; the Interface Description Language (IDL). When implementing remote procedure call applications, IDLs provide the definition of services and data types for automatic generation of platform specific code. This capability is ensured by an IDL specification being *precise*, *representation independent*, *language independent*, *maintainable* and *portable* (Nestor et al., 1981).

With layers, actors and remote procedure calls, all main architectural design patterns applied by SeMBa rely on message passing. Exploiting this commonality eliminates the need for a second conversion at API level, by using generated IDL data structures as the internal representation of SeMBa concepts. The model layer's facade component provides objects that may be directly serialized by the RPC marshalling component.

4.5 Architecture Overview

The previous sections proposed a layered software architecture for a semantic media back end, that fulfills the requirements defined in Chapter 3. The general functionality of these layers has been presented and elaborated.

Figure 4.6 depicts the overall application design. Adhering to the applied software design patterns, coupling between layers and modules is loose due to dedicated message end points and an overall message based communication. For application support a fourth package is introduced to cover the cross cutting concerns. Globally required variables, functions and objects of the software – such as configuration data, exception handling or general helper functions – can be accessed from within all layers. This also covers the universal object representation that is logically assigned to the API layer, as it is part of the RPC component. These representations will also serve as internal data structures, they have to be used within all layers. Other messages that are not strictly tied to a layer are also defined in a global object.

In addition to the layered architecture, the universal object representation and the ontology model are both defined detached from the actual software implementation. This fosters maintainability by allowing manipulation of software functionality without a need for code access. Introducing new functional clusters to the data logic should be similarly straightforward.

Chapter 5 will cover the implementation of this proposed design.

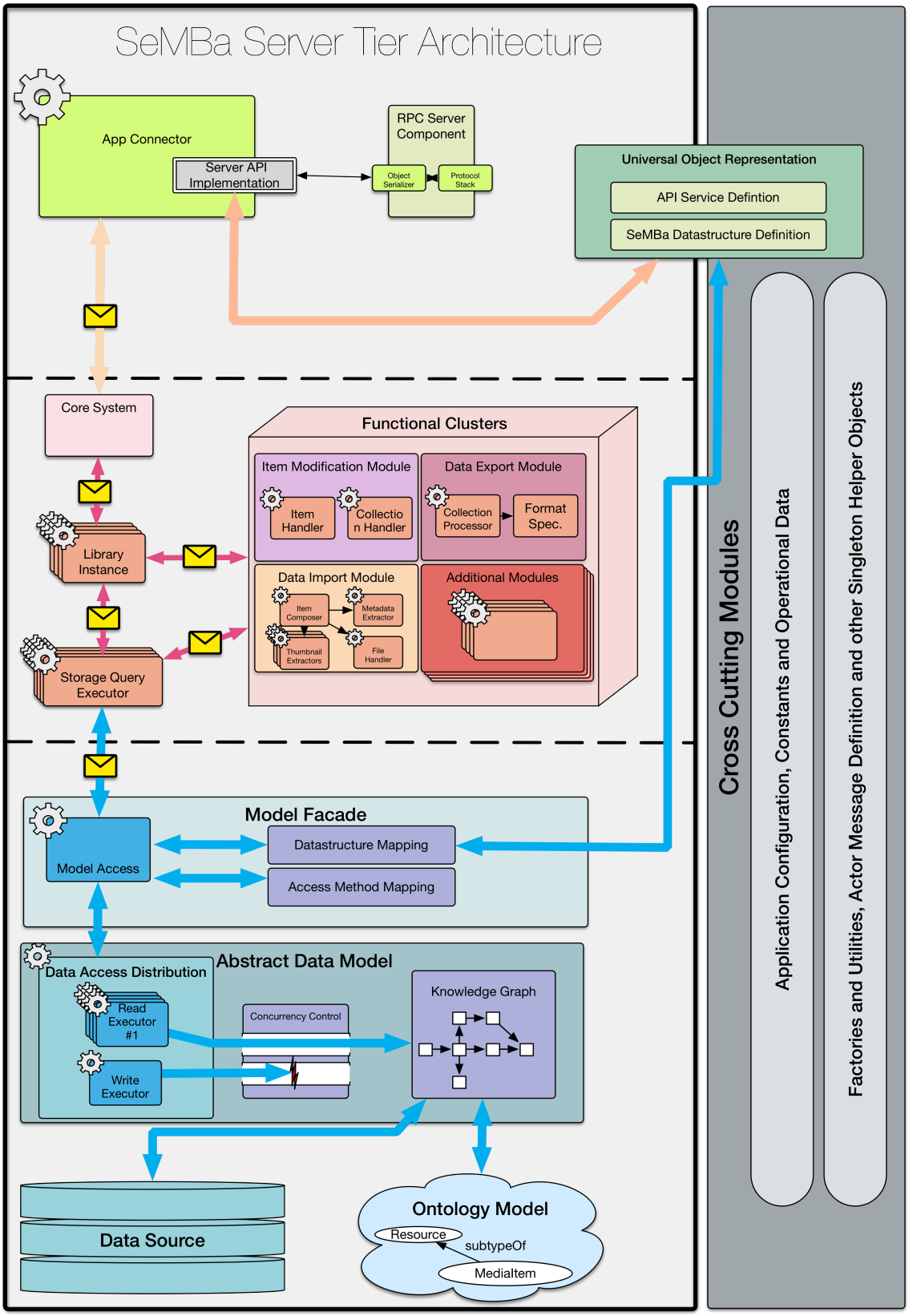


Figure 4.6. Architectural overview of SeMBa's server application tier. Communication between layers is message based and limited to dedicated actors.

5 Implementation

This chapter covers the implementation of the layer based software concept developed in Chapter 4. In accordance with the non functional requirements, it focuses on the implementation aspects facilitating application maintainability. External libraries and data structures will be introduced within their respective sections.

5.1 General Implementation

5.1.1 Programming Language

Because this thesis describes a complete redevelopment of the original SeMBa'13 application, the choice of programming language was not predetermined. However, the software design presented in Chapter 4 established additional non functional requirements. The chosen language should facilitate the implementation of actors as well as provide access to a wide base of existing software libraries. Mandatory external libraries should provide support for RDF graphs, remote procedure calling and metadata extraction.

The Scala programming language by Odersky et al. (2004) is an object-functional programming language, combining functional and object oriented programming paradigms. Based on the Java Virtual Machine it is compiled to Java Bytecode before execution and offers compatibility to existing Java applications. Other advantages of Java such as its platform independence, garbage collection, and profiling for easy prototyping are also preserved. Compared to Java code, the conciseness of Scalas functional style (e.g. iteration over collections), adds to its understandability, hence maintainability (Dubochet, 2009). The focus on immutable data structures and pure functions facilitates comprehensible, concurrent programming.

Regarding the need for RDF, RPC and file analysis implementations, Scala benefits from the ubiquity of Java libraries that offer solutions for all mandatory features

listed above. SeMBa makes extensive use of *Apache Jena, Apache Tika, Apache PDFBox, Google Protocol Buffers, Google gRPC* and *ScalaPB*. Justification, description and references for the libraries will be given within the implementation context.

The conceptual architecture for SeMBa is heavily based on the approach of modeling all functionality inside actors to promote a message driven, extensible and concurrent design. Since version 2.10, Scala includes the Akka actor toolkit by default (Roestenburg et al., 2015). Akka is an implementation of the actor model, it allows the user to build large systems of actors that communicate by passing messages. Core functionalities such as message passing, mailbox queuing, thread scheduling and fault recovery are provided by the toolkit, but may be customized to fit a specific use case. All of an actors behavior relies on the function `receive : PartialFunction[Any, Unit]`. Each message that is received by an actor implementation during its lifecycle is sequentially passed as an argument to this partial function. As seen in code example 5.1, a typical receive function applies pattern matching to define the actors behavior upon different types of messages. With *receive*s return type `Unit` all replies to a message need to be explicitly sent to the receiving counterpart which limits actor interaction to message passing. If shared state is avoided, or itself wrapped in an actor, this limitation and the sequential order of message processing ensure thread safe execution of multiple actors at once.
With the inclusion of Akka, the wide array of available libraries and its object-functional paradigm, Scala is an optimal choice for developing a maintainable, concurrent framework for semantic data management.

Code Example 5.1. A Scala code example of an actors receive function replying on a certain message.

```
1 class SecretActor extends Actor {
2
3 override def receive = {
4     case "What can I get you?" => sender ! "Martini. Shaken - Not stirred."
5     case _      =>  sender ! "Bond. James Bond."
6   }
7 }
```

5.1.2 Project Structure

All code related to the Semantic Media Backend has been implemented using the IntelliJ IDE ("IntelliJ IDEA the Java IDE," 2017) with the sbt build tool ("sbt - The interactive build tool." 2017) for dependency management. The project structure follows the sbt guidelines with a separation of `test` and `main` packages. To facilitate navigation inside the project, packages for all layer implementations and functional clusters have been created.

Application wide settings are stored in an XML file within the application folder. For each library that is created by the application, a new folder with the following contents is written to the disk:

1. A config.xml to define the used ontologies, storage model, language and other library specific settings.
2. A data folder to hold all imported source files and thumbnails, sorted by their respective URI.
3. A library specific ontology to store additional terminological data. (see Section 5.2.1)
4. An ontology folder to hold the data used by the **SembaStorageComponent** for storing assertional statements about the library. (see Section 5.2.3)

5.1.3 Stackable Modifications with ActorFeatures

A core idea behind the conceptual architecture of SeMBa, visualized in Figure 4.6, is functional extensibility by introduction of additional functional clusters. The implementation details of such features are referenced in Section 5.4.1. Even if they apply logic to different kinds of messages, some functionality between all actors of the concept is shared: The **AppConnector** processes a RPC request, sends a message to the respective **LibraryInstance** where it is relayed to the responsible module. The results of all actor executions need to be collected and returned to the **AppConnector**. While this behavior of *"accept - apply - collect"* could be directly integrated into each actor implementation, a centralized definition reduces boilerplate code and supports maintainability.

In Scala, code reuse is greatly enhanced by the introduction of **Traits**. Unlike normal class inheritance in Java, **Traits** can be mixed into multiple classes, to share a behavior although they inherit different superclasses. This technique allows to apply a stackable trait pattern, where multiple **Traits** override the definition of an actors `receive : PartialFunction [Any, Unit]` to inject their defined behavior (Odersky, Spoon, & Venners, 2010, p.226-230) .
In SeMBa, all actors that can be extended with such feature, mix in a base-trait called **ActorFeatures**. As seen in Code Example 5.2 the abstract receive function of an actor is overridden by the trait, implementing actors are required to define a wrapped function instead.

Code Example 5.2. ActorFeatures defines an abstract function wrappedReceive for implementing classes. Stackable traits override the receive function to inject their behavior.

```
1  trait ActorFeatures extends Actor {
2    def wrappedReceive: Receive
3
4      def wrappedBecome(r: Receive) = {
5          wrappedReceive = r
6      }
7      override def receive: Receive = {
8        case x => if (wrappedReceive.isDefinedAt(x))
9        wrappedReceive(x) else unhandled(x)
10     }}
```

If a message is not handled by any of the feature traits, the base classes *wrappedReceive* is called or an exception is thrown. Further illustrations of the inheritance hierarchy and linearization for a **LibraryInstance** using **ActorFeatures** is given in 5.4.1.

5.1.4 JobHandling Trait for Distributing Requests

The *"accept - apply - collect"* process of distributing tasks that are connected with a client request and aggregating their results, mentioned in Section 5.1.3, unites all actors with a functional role in SeMBa. For this purpose, a set of message abstractions to represent requests and replies is introduced.

Code Example 5.3. All tasks relayed between actors inherit **Job**. Sending a **Job** always leads to receiving a JobReply that contains a ResultArray. Specific results can be extracted from the ResultArray.

```
1  trait Job {
2    var jobID: UUID = UUID.randomUUID()
3    def newId() = jobID = UUID.randomUUID()
4  }
5  case class JobReply(job: Job, result : ResultArray)
6  case class JobResult(val content: ResultContent)
7  trait ResultContent{val payload: Any}
8
9  class ResultArray(input: ArrayBuffer[JobResult] = ArrayBuffer[JobResult]())
10 {
11   var results = input
12   def get[T <: ResultContent]( aClass: Class[T]): T = {...}
13   def getAll[T <: ResultContent]( aClass: Class[T]): ArrayBuffer[T] = {...}
14   def processUpdates(): ArrayBuffer[UpdateMessage] = {...}
15   def extract[T]( aClass: Class[T]): T = {...}
16  }
```

As shown in Code Example 5.3 all requests between actors mix in the **Job** trait, facilitating pattern matching and identification by UUID. Replies on the other hand contain a **ResultArray** with all results a job has aggregated on its path as well as a reference to the original job. The class **ResultArray** wraps a statically typed **Array[JobResult]** with convenience functions for extracting specific types of **ResultContent** or even their generic payload.

All actors shown in Figure 4.6 are required to adhere to this strategy of accepting a (sub)task and finally giving a reply to the sender. By mixing in the **JobHandling ActorFeature** (see Code Example 5.4), this behavior is reused and task management mostly automated.

Inside its *receive* function, an actor employs pattern matching to check if it is designed to process the received **Job**. If so, the **Job** is accepted and its JobID is stored with a reference to the sender. If subtasks need to be distributed among other actors, a new **Job** is created, linked to the original **Job** and added to a list of subtasks

that are waiting for completion. Upon receiving a **JobReply**, the results are merged into a **ResultArray**. If all subtasks have been completed, a **JobReply** containing a reference to the original **Job** and all contents of the **ResultArray** are sent to the original sender. By overriding *finishedJob()*, this behavior can be modified to fit more specific needs.

In case subtasks need to be executed sequentially, a function *reactOnReply()* is executed when receiving a **JobReply**, this enables the developer to create a new subtask based on the results of the currently received reply. This is also possible if the results of multiple tasks are required to initiate a subtask, by creating "Job Clusters". For further understanding a partial implementation of this trait can be reviewed in Code Example 5.4. By facilitating sequential or parallel execution and result aggregation of tasks, the **JobHandling** trait gives a feature developer the tools to quickly design complex operations without the need for mutable shared state, keeping a high level of concurrency and maintainability.

Code Example 5.4. The **JobHandling** trait maps incoming **JobReplies** to their original **Job**. After a reply for all pending subtasks has been received, a finish-operation is executed.

```
1  trait JobHandling extends Actor with ActorFeatures {
2    var waitingForCompletion = Set.empty[(UUID, Job)]
3    var originalSender = mutable.HashMap[UUID, ActorRef]()
4    val jobResults = mutable.HashMap[UUID, ResultArray]()
5
6    override def receive: Receive = {
7      case reply: JobReply => handleReply(reply)
8      case x => super.receive(x)
9    }
10   def handleReply(reply: JobReply): Boolean = {
11     var completed = false
12     val entry = waitingForCompletion.find(_._1 == reply.job.jobID)
13     if (entry.isDefined) {
14       val originalJob = entry.get._2
15       val newResults = reply.result
16       jobResults.apply(originalJob.jobID).add(newResults.results)
17       // React on a reply before a Job may be finished to trigger further subtasks.
18       reactOnReply(reply, originalJob, jobResults.apply(originalJob.jobID))
19       waitingForCompletion -= entry.get
20       completed = !waitingForCompletion.exists(_._2.jobID == entry.get._2.jobID)
21       if (completed) {
22         val jobMaster = originalSender.apply(entry.get._2.jobID)
23         val resultBuffer = jobResults.apply(originalJob.jobID)
24
25         // Call finishedJob before deleting references.
26         finishedJob(originalJob, jobMaster, resultBuffer)
27         jobResults.remove(originalJob.jobID)
28         originalSender.remove(originalJob.jobID)
29       }}
30     completed
31   }
32
33   def acceptJob(newJob: Job, sender: ActorRef): Job = {
34     jobResults.put(newJob.jobID, new ResultArray)
35     originalSender.put(newJob.jobID, sender)
36     newJob
37   }
38 {...}
39 }
```

5.2 Model Layer

5.2.1 OWL Base Ontology Model

The **SeMBa Concepts** described in Chapter 4.2 are modeled using the Web Ontology Language (OWL) (W3C, 2009). Presented by the World Wide Web Consortium, OWL provides a formalized syntax for describing RDF resources. With a wide spread availability (Wang, Parsia, & Hendler, 2006) OWL has become the defacto standard for ontologies in the semantic web. Existing editors such as Protégé (Knublauch, Fergerson, Noy, & Musen, 2004) can be used to efficiently model new base ontologies for SeMBa. The specified exchange format Turtle provides a human readable serialization of ontologies. Even though SeMBa does not require much of OWLs full language specification, features like the transitive or inverse properties are beneficial for modeling complex item collections. If **Resource** B follows **Resource** A, and precedes is the inverse of follows, *A - precedes B* can be inferred. For further explanations on the choice of OWL for media collections see Isermann (2013b).

OWL is designed to support a modular structure, by using an `owl:import` syntax existing ontologies can be included in a document. This feature facilitates the design of a base ontology that may be extended by more domain specific taxonomies or even existing ontologies from the semantic web. With SeMBa an application or content developer creates the terminological component using an ontology editor. The application then creates assertional statements describing imported files, based on the provided concepts. Figure 5.1 illustrates the structure of a prototypic ontology for using SeMBa in a teaching context. The base ontology layer defines the different `SeMBa Concepts` required for defining **Resources** and basic **Relations**. The domain specific layer adds granularity by introducing subclasses to model the different types of **Media Collections** in a teaching context. **CollectionRelations** representing the flow inside a presentation are introduced.

These upper two layers are designed to be immutable for the application, typically they are provided as a web reference. This ensures, that all instances of SeMBa using a specific domain ontology have access to the same set of concepts. The library definition ontology is a local document importing the upper ontology layers. As the terminological component of SeMBa, it is continuously updated with definitions for generated **MetadataTypes**. This differentiation of immutable concept definition

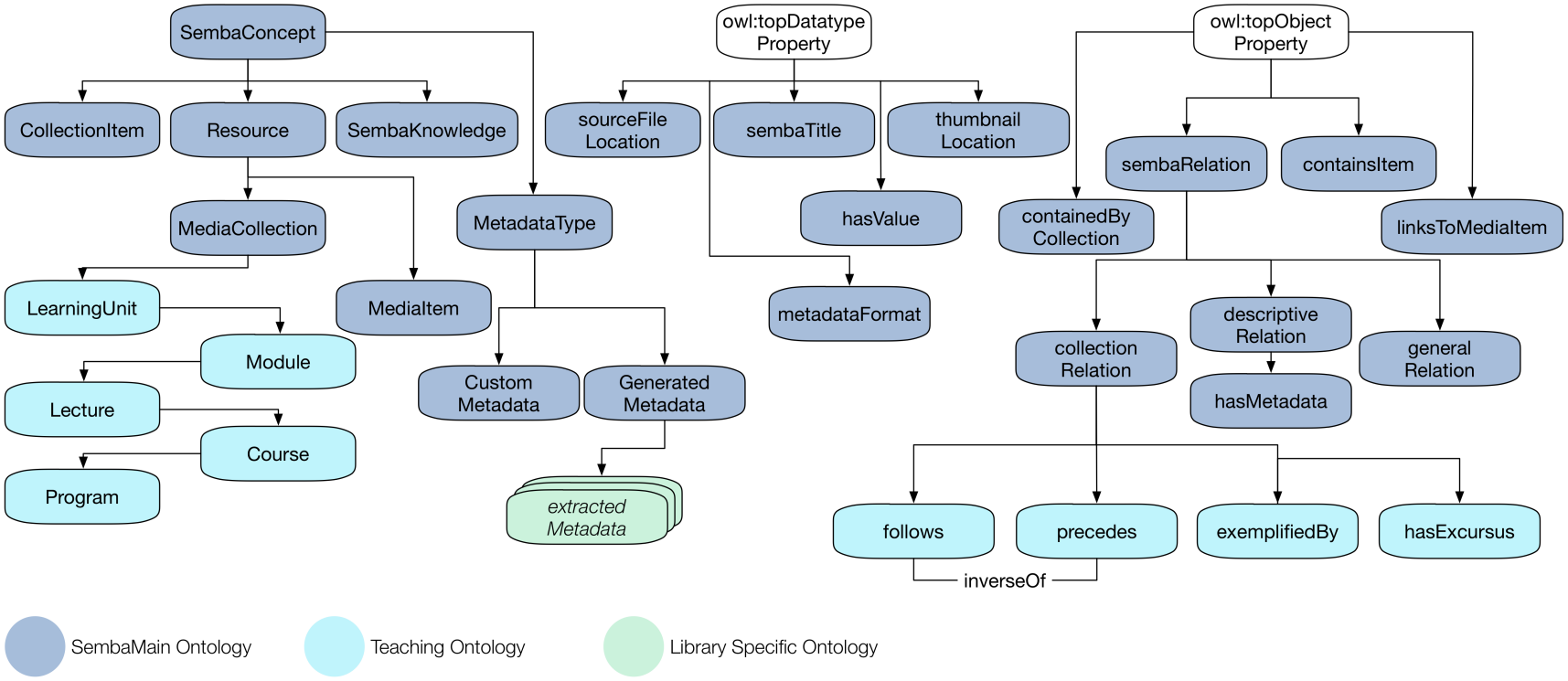


Figure 5.1. Ontology layout of a teaching library. SeMBa relies on OWLs capability to import existing ontologies into new ones. Regardless of domain, all libraries share the same base terminology. Additional concepts and assertions can be added on domain level. The library layer contains concepts based on the library contents.

and mutable subclasses of concepts facilitates the exchange between different SeMBa applications that are based on the same domain ontology.

5.2.2 Model Facade

SeMBas knowledge graph combines the OWL base ontology model with the assertional statements about contents of the library. The mapping of stored statements to the graph and provision of read/write methods is carried out by an RDF/OWL API. The data access functionality of SeMBa is built around the Apache Jena library.

Apache Jena

Even for a wide spread programming language like Java libraries for reading and writing serialized RDF data are sparse. The three most prominent solutions are the OWL API, developed at the University of Manchester (Horridge & Bechhofer, 2011), the Protégé-OWL API developed at the University of Stanford (Knublauch et al., 2006) and Apache Jena (Carroll et al., 2004), maintained by the Apache Foundation. All of these libraries are able to provide an object oriented mapping of ontological concepts. However, as a project supported by the Apache Foundation, Jena comes with an active base of developers, issue management and comprehensive documentation. The project also contains several persistence solutions and SPARQL support which will be elaborated in upcoming sections.

Apache Jena is built in a layered structure providing different levels of abstractions, it can be reviewed in Figure 5.2. The internal RDF graph data structure is wrapped into a **Model** instance by the RDF API. **Models** can be used to add and remove statements from the graph and query for certain **Classes**, **Properties** or **Literals**.

The higher level **InfModel** adds inference capabilities to a **Model** by binding a schema to the graph and deducting new statements through reasoning. When using the OWL specification as the reasoning schema, an **InfModel** becomes an **OntModel**. As the highest abstraction, **OntModels** provide wrappers for all required OWL concepts such as **Classes**, **Individuals** or **ObjectProperties** and a wide range of convenience functions while the basic RDF graph structure remains intact. To influence the performance, **OntModels** can be bound to a series of OWL reasoners. These range from only acknowledging transitive/reflexive **rdfs:subProperty** and

rdfs:subClassOf annotations up to an incomplete subset of the *OWL Full* specification. This allows SeMBa to take advantage of the convenient Ontology API and still respect the performance requirements.

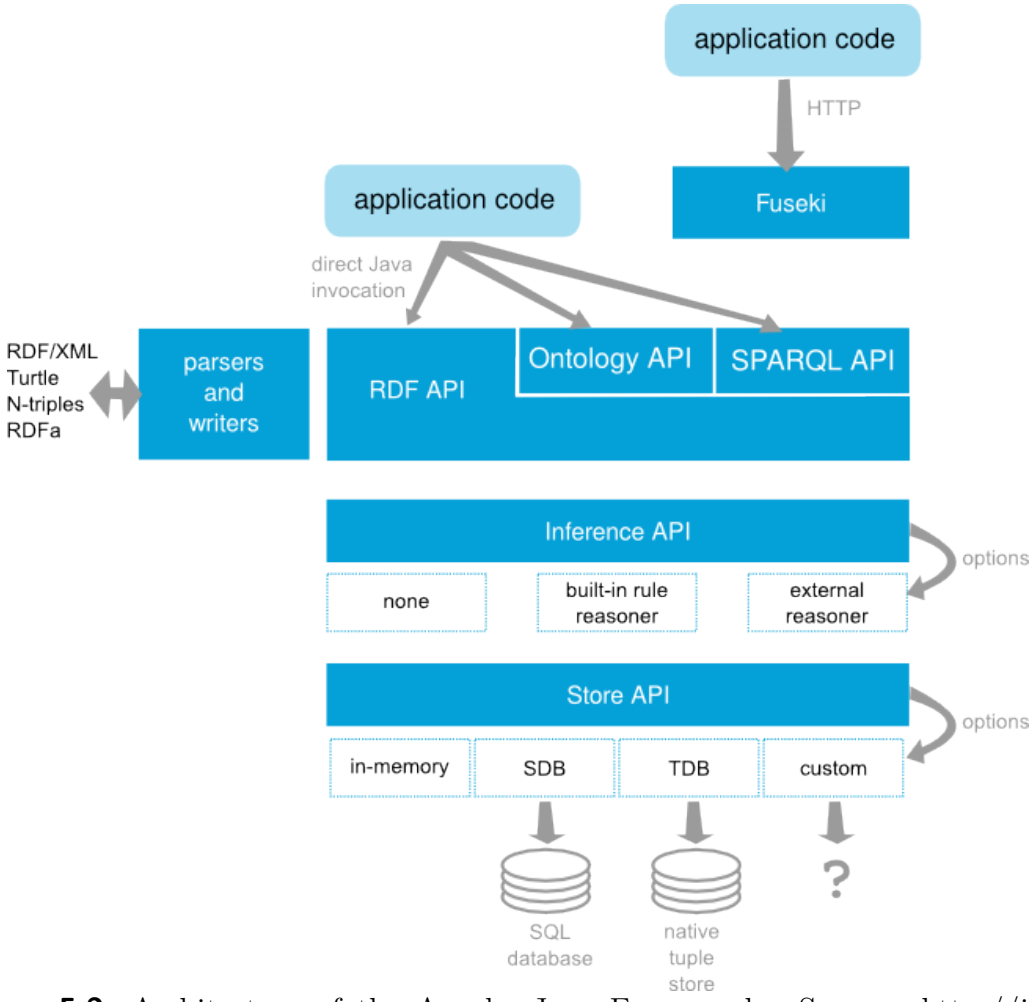


Figure 5.2. Architecture of the Apache Jena Framework. Source: http://jena.apache.org/about_jena/architecture.html

Access Methods

Although the Ontology API provides abstractions for fairly easy access to the RDF graph describing a library, the required code is still bulky and would require to be written multiple times for different model requests. The model facade includes a singleton object **AccessMethods**, that provides reusable convenience functions for ontology operations. Code Example 5.5 shows a retrieval function that is passed an **OntModel** and returns an SeMBa API representation of all **Resources** that are

part of a library. While most retrieval functions return external representations, modification functions may return Jena objects for keeping the ability to chain modifications.

Code Example 5.5. The **AccessMethods** are built to conveniently access **OntModel** instances to retrieve or modify external datastructure representations.

```
1  object AccessMethods {
2    ⋮
3    def retrieveLibContent(model: OntModel, config: LibInfo): LibraryContent = {
4      // Initialze return Value – Empty library content
5      var content = LibraryContent().withLib( Library ( config . libURI ))
6
7      // Retrieve all Individuals that are Resources: Items / Collections
8        val ontClass = model.getOntClass(SembaPaths.resourceDefinitionURI)
9        var individuals = Option(model. listIndividuals (ontClass))
10       if ( individuals . isDefined ) {
11         var iter = individuals . get
12
13         //Convert Individuals to External Representation and add to LibraryContent
14         while (iter . hasNext) {
15           val resource: Individual = iter . next()
16           content = content.addLibContent((resource.getURI, DatastructureMapping.wrapResource(
17                resource , model, config ))))}}
18     content}
19
20   def retrieveMetadata(item: String , model: OntModel): ItemDescription = {...}
21   ⋮
22 }
```

Datastructure Mapping

Outside of the model layer, the SeMBa API datastructures described in Section 5.3.2 are used to represent ontology concepts. A **Datastructure Mapping** singleton object provides the **AccessMethods** with conversion functions for all required objects. In Code Example 5.5 line 16 a Jena **Individual** is converted to a Semba API **Resource**. Conversion functions are always passed a reference to the Jena object and

the containing ontology model. They query the model for all required information and return a SeMBa API object constructed using these information.

5.2.3 Abstract Data Model

The Abstract Data Model is responsible for handling concurrent read/write access to the Jena model. As SeMBa is designed to support different types of RDF serialization the abstract class **SembaStorageComponent** defines a set of functions to adhere to the access scheme defined in Section 5.2.4. The concurrency model of multiple readers or a single writer is enforced by the combination of this scheme and the specific implementation of a **SembaStorageComponent**.

As seen in Code Example 5.6 a storage solution for SeMBa can be implemented quite freely. The first main requirement for a **SembaStorageComponent** is provision of separate access to the terminological and assertional component of the RDF graph in the form of an Jena **OntModel**. The `perform_*[T](f: => T): T` functions are designed as exclusive access points to a storage component. They execute a passed read or write function inside a thread safe environment with multiple readers or a single writer and return its resulting value. To exemplify this behavior one could pass the `retrieveLibContent ( SembaStorageComponent.getABox, config)` function shown in Code Example 5.5 to a **SembaStorageComponent**.*performRead*. The storage component would wait for a read lock, execute the query and convert all **Resource** individuals and return a Semba API **LibraryContent**. A companion object of **SembaStorageComponent** is used to create new instances of a specific implementation. By accessing a **LibInfo** configuration object the required storage solution is evaluated and based on the implementation details either a new instance or the reference to a singleton object is created.

Code Example 5.6. A **SembaStorageComponent** is required to provide Jena **Models** and a concurrency strategy regardless of the implemented storage solution.

```
1  abstract class SembaStorageComponent {
2
3    /** @return an OntModel containing all asserted/deducted statements about the library content. */
4    def getABox: OntModel
5
6    /** @return an OntModel containing all asserted/deducted statements about the library concepts. */
7    def getTBox: OntModel
8    /** Should only be used for read access, increased query performance.
9      * @return an OntModel containing only the asserted statements
10     */
11   def getABoxNoReasoning: OntModel
12   def saveABox(model: OntModel)
13   def saveTBox(model: OntModel)
14   /**
15     * Sets a Read Lock. Executes the query if no write lock is set.
16     * @param f A query function without any side effects to the model.
17     * @tparam T The return type of the Query function.
18     * @return An instance of T
19     */
20   def performRead[T](f: => T): T
21   /**
22     * Sets a Write Lock. Executes the query if no lock is set.
23     * @param f A query function that changes the state of the model.
24     * @tparam T The return type of the Query function.
25     * @return An instance of T
26     */
27   def performWrite[T](f: => T): T
28   /** Called on Startup. Initializes the persisted DataModel */
29   def initialize
30 }
```

5.2.4 Accessing the Storage Component

The general concept of concurrently accessing the data model by calling the read-/write functions of the **SembaStorageComponent** was manifested in the previous section. To add a second layer of concurrency control and incorporate the locking

base, direct access scheme into the actor model, the actors **SembaStorage**, **ReadExecutor** and **WriteExecutor** are introduced.

Storage Operation Messages

All functional clusters of the logic module are able to send **Jobs** that extend the abstract class `Storage [...] Request(val operation: (SembaStorageComponent => JobResult))` to the model layer. A request contains a unary partially applied function `operation` that is defined by the developer of the originating SeMBa module. When passed a **SembaStorageComponent** this function returns a **JobResult** that is returned to the module. The complexity of such `operation` can range from applying a single **AccessMethod** to the storage components *performRead* function to chaining multiple *performWrite* calls that include custom calls of the Jena **OntModel** API. The definition of a **StorageWriteRequest** can be seen as part of a detailed walkthrough in Code Example 5.11 of Section 5.4.2.

SembaStorage and Executors

The **SembaStorage** is parent to an actor pool of **ReadExecutors** and a single **WriteExecutor** actors. In their constructor these actors are passed a reference or individual instance of the required **SembaStorageComponent**, depending on its implementation. **SembaStorage** distributed incoming requests to these executors, using a round-robin routing scheme. The differentiation of multiple read and a single write executor ensures that read requests are handled prioritized, even without knowing the specific lock strategy of the storage component. This is important, as read requests are typically blocking on the client side. The actual functionality of executors is trivial. The partially applied function *SembaStorageComponent => JobResult* of the request is applied to the local **SembaStorageComponent** and the JobResult is automatically returned to the sender by the **JobHandling** trait.

5.2.5 Storage Component Implementations

As depicted in Figure 5.2, the Jena Store API offers different serialization formats and allows the implementation of custom storage models for RDF triples. The **SembaStorageComponent** class facilitates this customization. The different iterations

of SeMBa, over the course of this project, included two fundamentally different solutions for handling the data model. As performance testing (see Chapter 6) suggested essential flaws for the Jena in-memory model, the triple database remained the only pursued option.

Jena In-Memory Model

Solely relying on the ability to serialize Jena **Models**, the first implementation of this project used RDF/XML files to describe each library component. When adding a new **Resource** to the library, an OWL ontology is created and stored on disk in human readable turtle notation. The ontology imports the library base terminology and can be used autonomous from other **Resource** descriptions. Querying a single **Resource** for its metadata or a **MediaCollection** for the contained relations benefits from this architecture because the amount of statements describing a **Resource** is typically low. However, the implementation needs to address several complex issues:

Querying the entire library:
Opposed to single **Resources**, querying the contents of a library requires a union of all models. Jena provides methods to merge references to existing **Models** into such unions. New assertions about **Resources** are written into this union model instead of the respective **Resource** model, interfering with the introduced separation of data. Bypassing this behavior means using the union for read queries, create a list of all single **Resource** models that require a write and accessing them one by one. A modification of the union model triggers the reasoning process of all individual models to evaluate changes. Even when binding a most basic OWL schema this process has an impact on performance when handling larger amounts of **Resources**.

Library Initialization:
For every **Resource** a serialized XML file is loaded from the file system. The file is parsed and converted into the Jena graph structure. After creating a new **OntModel** the OWL reasoner is bound and executed. To facilitate access to the autonomous **Resource** model it is added to a map with the **Resource** URI as a key. The created model is then added to the union, which requires a unique write lock to prevent concurrency issues.

Concurrency Control:
Jena provides a MRSW locking strategy by marking critical sections as either read or write locked. When obeying to this lock contract by handling all model access inside a critical section with the correct lock, the strategy is thread-safe. However the multitude of interwoven **Resource** models calls for additional logic on top of this strategy. For instance when writing to a single **Resource** model, the union model enters a write lock as well, other **Resource** models may be read but not written.

Conclusion:
Although the first iteration of SeMBa using the in-memory model is functional, it poses a risk to successfully implement the non-functional requirements. The complex handling of access to different **OntModel** instances is prone to implementation errors, that might not be easily detected due to the nature of concurrent processes. The high number of **OntModel** instances with distinct reasoners accounts for high memory consumption and low performance. Due to these restrictions, a second storage solution based on a triple store database was implemented.

Triple Store

The Jena Tuple Database (TDB) is a storage implementation, developed as part of the Jena project. Its detailed implementation can be reviewed in Owens, Seaborne, Gibbins, et al. (2008). In general terms, TDB creates identifiers for each RDF term in a graph and stores statements in three B+ tree indexes. These indexes form Subject-Predicate-Object, Predicate-Subject-Object and Object-Subject-Predicate statements for faster access based on a specific query. Depending on the available heap space the store is either completely mapped to memory or cached based on access frequency. An additional fourth identifier is used to create a named graph to accommodate storing multiple **Models** in one TDB instance. TDB implements the Jena graph interface by providing an RDF dataset that can be queried for a single named model or a union of several named graphs.

To allow concurrent database access TDB implements a transaction mechanism based on write-ahead logging. Write access is limited to a single transaction at a time that is applied to the database after being committed. This facilitates a design where presence of a write lock does not inhibit starting a read transaction. Hence the presented state always reflects the actual, transaction free main database.

This allows SeMBa to perform larger atomic write processes without affecting read performance.

TDB Based SembaStorageComponent Implementation:
TDBs integration into the Jena framework facilitates a straightforward implementation of **DatasetStorage**, the TDB **SembaStorageComponent**. Code Example 5.7 shows a partial implementation of **DatasetStorage**. Upon initialization the library definition ontology is read from file and a union is formed with the existing tBox model in storage. This allows users to import library specific concepts from other SeMBa instances, facilitating the migration of items between libraries sharing the same base ontologies. Execution of *performWrite* starts a transaction by specifying the **ReadWrite** lock, performs the operation and ends the transaction. Exceptions during a transaction are not handled internally due to Akkas *"let it crash"* philosophy. If an exception occurs an **ErrorResult** is returned by the supervisor of the **WriteExecutor** and a new Executor is started. The abundance of any internal state assists this behavior. Access to the assertional model is realized by getting the assertional and terminological models from the named graph and adding the terminology as a schema to the assertional model. Changing the **OntModelSpec** used for creating the assertional model affects the extent of reasoning.

With the **DatasetStorage** implementation, each **Executor** holds a unique instance of the storage component. TDB does not limit the amount of simultaneous connections to a certain database. The transactional model prevents concurrency issues.

Code Example 5.7. DatasetStorage implements the abstract **SembaStorageComponent** using Jenas Tuple Database.

```
1  class DatasetStorage(config: LibInfo) extends SembaStorageComponent {
2    val tBoxName = "tBox"
3    val aBoxName = "aBox"
4    val loc = new File(new URI(config.rootFolder + config.constants.storagePath)).getAbsolutePath
5    val data = TDBFactory.createDataset(loc)
6  ⋮
7    override def getABox(): OntModel = {
8      val schema = data.getNamedModel(tBoxName)
9      val aBoxModel = ModelFactory.createOntologyModel(OntModelSpec.OWL_MEM_RDFS_INF,
10       data.getNamedModel(aBoxName))
11     aBoxModel.addSubModel(schema)
12     aBoxModel}
13 ⋮
14   override def performWrite[T](f: => T): T = {
15     var retVal = None: Option[T]
16     data.begin(ReadWrite.WRITE)
17     try {
18       retVal = Option(f)
19       data.commit()}
20     finally data.end()
21     retVal.get}
22   ⋮
23 }
```

5.3 Bidirectional Remote API

Isermann (2016) implemented a remote API for the SeMBa'13 proof of concept application. However, its lack of bidirectional communication between server and client pointed out the requirement for an improved API concept. The use of remote procedure calling based on an IDL is still favored, so a new framework has to be selected for implementing the API.

5.3.1 Google Protocol Buffers & gRPC

After the implementation of SeMBa'13, Google introduced gRPC (Ryan, 2015), an RPC protocol stack for its data serialization format Protocol Buffers (Google, 2008). Protocol Buffers was developed to provide a simple IDL for describing structured data, that can be compiled into a compact binary format. It offers code generators for over ten programming languages including Java, which facilitated development of a native third-party Scala compiler (TrueAccord, 2014). In combination with gRPC, Protocol Buffers offers some key points that qualify it as the RPC framework for SeMBa:

- Bidirectional streaming facilitates sending and receiving of larger datasets or temporally related information such as the sequence of database updates. By keeping an active stream over its complete lifecycle the server is able to communicate update messages to the client.
- Message serializations are backward compatible as they don't carry an access schema; fields are referenced by indices. Older client implementations can still communicate with more recent servers and vice versa.
- Implementations can mix asynchronous and synchronous procedures.
- The workflow of using gRPC is fast end easy to use: describing types and services in a concise IDL, compiling the source code and finally implementing the service interfaces. This facilitates rapid prototype iterations.
- Sending library contents over a remote connection results in large numbers of objects tht describe individual items. Protocol Buffers binary messages are up to 10 times smaller and can be parsed up to 100 times faster than XML based message formats (Google, 2017).

5.3.2 SeMBa Data Structures

To give the front end developer access to the application state, Protocol Buffer messages are used not only for communication, but as the main data structure mapping the ontology contents. Messages are defined in *.proto* files as seen in Code Example 5.8. The ScalaPB compiler generates a case class for each object. Due to their immutable state and Scalas pattern matching, these data structures are suitable for use in a concurrent actor system such as SeMBa.

Code Example 5.8. Protocol Buffer messages consist of numbered name-value pairs. Values can be scalar types, compounds such as maps or arrays or other messages. The shown **CollectionItem** contains the URIs of itself, the collection it is a part of and the **Resource** it represents. The relations map holds a list of **CollectionItem** URIs to each **CollectionRelation** assigned to this **CollectionItem**.

```
1  message CollectionItem {
2      string uri = 1;
3      string libraryResource = 2;
4      map<string, RelationValue> relations = 3;
5      string parentCollection = 4;
6      Library lib = 5;
7  }
8  message RelationValue {
9      repeated string destination = 1;
10 }
11 service SembaAPI {
12     rpc openLibrary (LibraryRequest) returns (LibraryConcepts) {}
13     rpc requestContents (Library) returns (LibraryContent) {}
14     rpc requestCollectionContent(Resource) returns (CollectionContent){}
15     rpc addToLibrary (SourceFile) returns (VoidResult) {}
16     rpc subscribeUpdates (UpdateRequest) returns (stream UpdateMessage) {}
17     ⋮
18 }
19 ⋮
```

Diagram 5.3 illustrates the composition of Protocol Buffer messages to describe a SeMBa library. Messages are kept fine grained to reduce network load by only transmitting requested information. For example, a **Resource** does not carry all metadata information as field. When required, an **ItemDescription** for a distinct **Resource** is created and passed to the client. The same strategy applies to state updates that are relayed to all clients. By encapsulating contents and concepts in fine grained units they can be replaced individually to update the local client-state of the library.

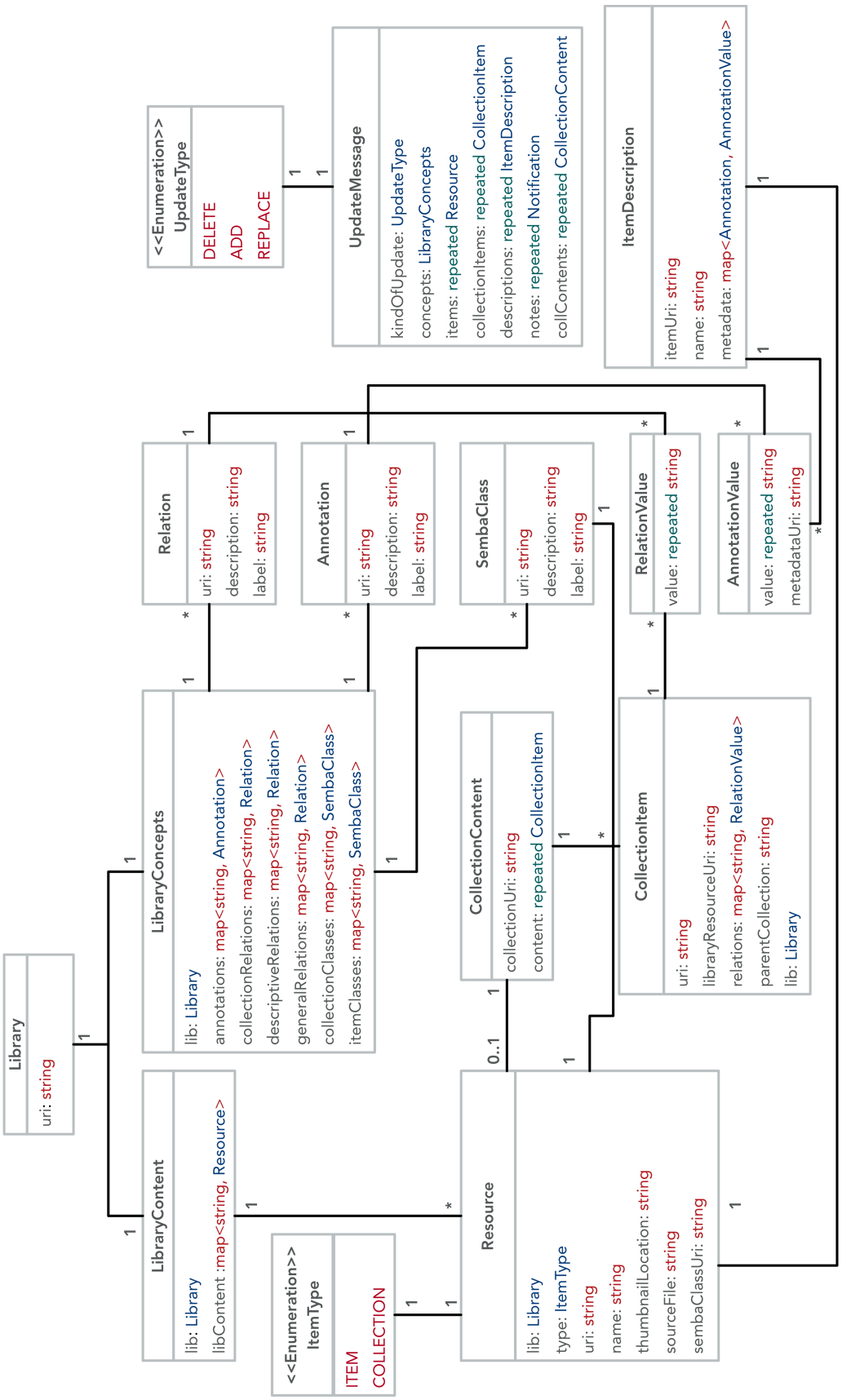


Figure 5.3. The basic structure of SeMBas Protocol Buffer messages. Outside of the model layer these classes are used to represent the state of a library.

5.3.3 SeMBa Remote Procedures

When executing a local API function, the developer can expect a call to always succeed in a certain amount of time. In contrast, remote procedure calls are subject to packet loss and network travel time. To minimize the impact of these factors, remote APIs should offer coarser grained functions then a local implementation would do (Fowler, 2014).
The approach of the SeMBa remote API is to directly map functional requirements to remote procedures. This behavior separates application logic from the front end implementation and provides an accessible interface for the application developer. An excerpt from the API definition for SeMBa can be seen in Code Example 5.8. It shows the differentiation between state affecting and pure functions. Read-only requests return actual SeMBa data structures that can be processed or displayed. Any modifications to the library return a **VoidResult** as an acknowledgment of the remote procedure call. After such call has been executed the results are wrapped in an **UpdateMessage** and sent through an update stream. With gRPC, clients can choose to execute API calls asynchronously or synchronously. This Publish/-Subscribe pattern for – possibly time consuming – state affecting functions allows synchronous function calls with asynchronous execution at server side without blocking issues. It also ensures that all registered clients receive updates and maintains a coherent shared state.

5.3.4 Application Connector

Processing of requests and distribution of update messages is handled by the **AppConnector** actor. It contains a nested class **ApiServer** that implements the gRPC **SembaAPI** service stubs.

ApiServer Implementation

Upon initialization of the **AppConnector** the nested class **ApiServer** is initialized. It builds a gRPC server that listens to the port specified in the application configuration and binds a **SembaApiImpl** to that server. **SembaApiImpl** overrides all functions that are defined in the **SembaApi** service. All function calls return a **Future**

of the specified return type that is returned to the caller on completion. The implementation the service does not differ much between functions. Each call message contains the ID of the affected library from which the responsible **LibraryInstance** actor is resolved. Using the akka *ask* pattern an **SembaAPICall Job** is created and sent to the **LibraryInstance**. The resulting future is mapped to the functions return type. Code Example 5.9 shows the implementation of a SPARQL query forwarded to the respective library.

Code Example 5.9. The implementation of all gRPC server stubs is based on resolving the **LibraryInstance** actor and *ask*ing it to return a **JobResult** of the required type.

```
1 /** Executes the [[SparqlQuery]] on the referenced OWL Model and returns all Items that match.
2         * lib is resolved by main Application object, then asked for a SparqlFilter JobResult.
3         * @param request Request containing the query as a [[String]]
4         * @return List of all matching Resources
5         */
6       override def sparqlFilter(request: SparqlQuery): Future[FilterResult] = {
7         val lib = app.get(request.getLibrary.uri)
8         ask(lib, SparqlFilter(request)).mapTo[FilterResult]
9       }
```

Update System Implementation

The **AppConnector** actor has the sole responsibility of distributing **UpdateMessages** that result from changes to a library model. It maintains maps of all registered clients for each **LibraryInstance** as well as the the update streams they have created when calling *subscribeUpdates()* (see Code Example 5.8). When receiving an update, all registered client streams are resolved. Each update is transmitted to the client by calling the *onNext()* function of the stream.

5.4 Program Logic Layer

To ensure a high level of maintainability, the program logic layer provides a framework for implementing and adding new functional clusters to a SeMBa application.

The main two classes are the **Application** object and the **LibraryInstance**.

The **Application** serves as the application entry point and **LibraryInstance** registry. It creates a new **LibraryInstance** in the SeMBa *ActorSystem* if the **AppConnector** receives an *openLibrary()* call and the library is not yet part of the registry. Further logic is required to keep track of the number of client sessions connected to each library and destroying any **LibraryInstances** that are not used by any open session.

The abstract class **SembaBaseActor** is the base class for mixing in all SeMBa **ActorFeatures**. All **LibraryInstance** actors that represent one distinct data model connection extend this base class and an arbitrary amount of feature traits. The foundation **LibraryInstances** lies in the **SembaBaseActor** itself mixing **ActorFeatures** and **JobHandling** to provide the correct *Actor.receive()* implementation and **Job** processing.

The third mix in – **Initialization** – modifies the behavior of a **LibraryInstance** to only react to incoming API messages after all feature subsystems have been fully initialized. For instance, the Jena in-memory model implementation needs to load and import all single ontologies before queries can be executed correctly. Incoming API messages are stashed and processed once an **InitializationComplete** message was received from all feature implementations and the original *receive* behavior has been restored. Apart from these interfaces for additional features, a **LibraryInstance** initializes a **Config** by parsing an XML configuration file. It contains all library specific settings and paths for use within the model layer and features. The following section goes over the general concept of implementing such features, using the example of the **ResourceCreation** module.

5.4.1 Semba Feature Traits

Figure 5.4 shows a class diagram of the currently implemented features for a **LibraryInstance**. While complexity may differ, all of them share three components:

- **ActorFeatures Trait:** Using the stackable trait pattern, this trait extends **SembaBaseActor**. The behaviors for feature initialization, receiving API **Job**

messages, and the corresponding *finishedJob()* are injected into the actor. This is achieved by abstract overriding the respective functions with the feature specific logic and calling the *super()* function to execute the other implementations.

- **JobHandling Actor:** Receives **Job** messages forwarded by the respective feature trait and handles them according to the feature requirement. Depending on the specific case this could be creating a single **StorageOperation** or a distributed **Job** including multiple batch processes sent to further actors of the feature implementation.
- **StorageOperation Singleton:** **StorageOperation** messages have been explained in Section 5.2.4. This singleton contains all case class definitions that extend **StorageOperation** and are intended to be sent to the query executors. It also aggregates the correlating static (SembaStorageComponent => JobResult) function definitions to facilitate easy access for modifications. In combination with the predefined **AccessMethods** this design is aimed to preserve the *Don't Repeat Yourself* (DRY) principle by keeping logically related information in one place for reuse (Hunt, 2000).

5.4.2 Feature Example: ResourceCreation

This section presents the implementation and control flow of the actual SeMBa feature for importing files. As shown before, the guidelines for implementing a feature are standardized. Important aspects of other functional implementations will be discussed in Section 5.4.3.

A Data Import Module was designed to implement the functional requirement groups **FR-5, FR-7** and **FR-8**, dealing with file import. The described control flow of processing an API request for a single PDF import is shown in Figure 5.5.

ResourceCreation Trait

When mixed into a **SembaBaseActor**, the **ResourceCreation** feature trait initializes a **ResourceCreator** actor and adds its reference as a variable.

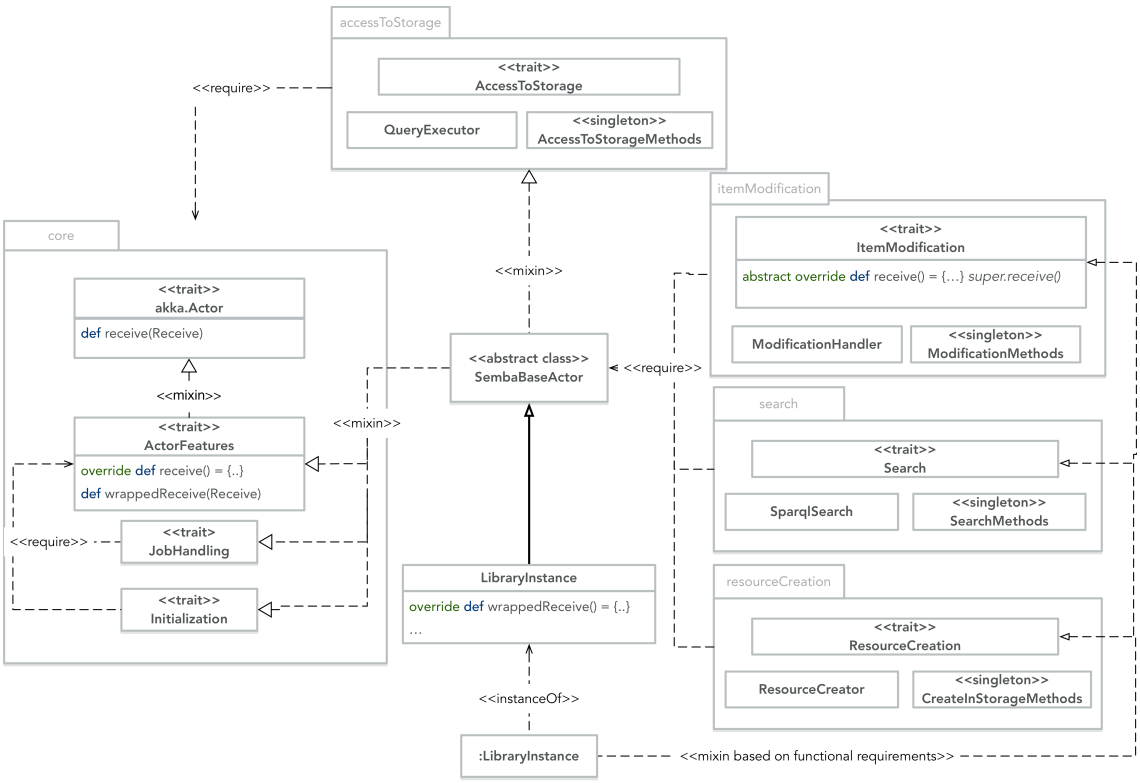


Figure 5.4. ActorFeatures serves as a base trait for stackable modifications in SeMBa. All **LibraryInstances** mix in the **SembaBaseActor** trait that provides job handling capabilities as well as model initialization. Optional features add new behavior for additional API calls by abstract overriding the **LibraryInstance**.*receive()* function. Functional implementations inside each feature package are not part of this simplified diagram.

The *receive()* function is overridden with a partial function matching the case class **AddToLibrary** – an API message containing the **SourceFile** SeMBa data structure. After checking if the the **SourceFile** is not empty but contains either a file path, raw data or **NewCollection** definition, an acknowledgment or error message is returned to the API and the message forwarded to the **ResourceCreator**.

The same **AddToLibrary** case is injected to the *finishedJob* behavior. All **UpdateMessages** are extracted from the aggregated **ResultArray** for the job and forwarded to the **AppConnector** for propagation to subscribers.

ResourceCreator Batch Job Distributor

An **AddToLibrary** message contains either the path to a source file/folder to be imported as **MediaItems** or a **NewCollection** specification to be imported as a

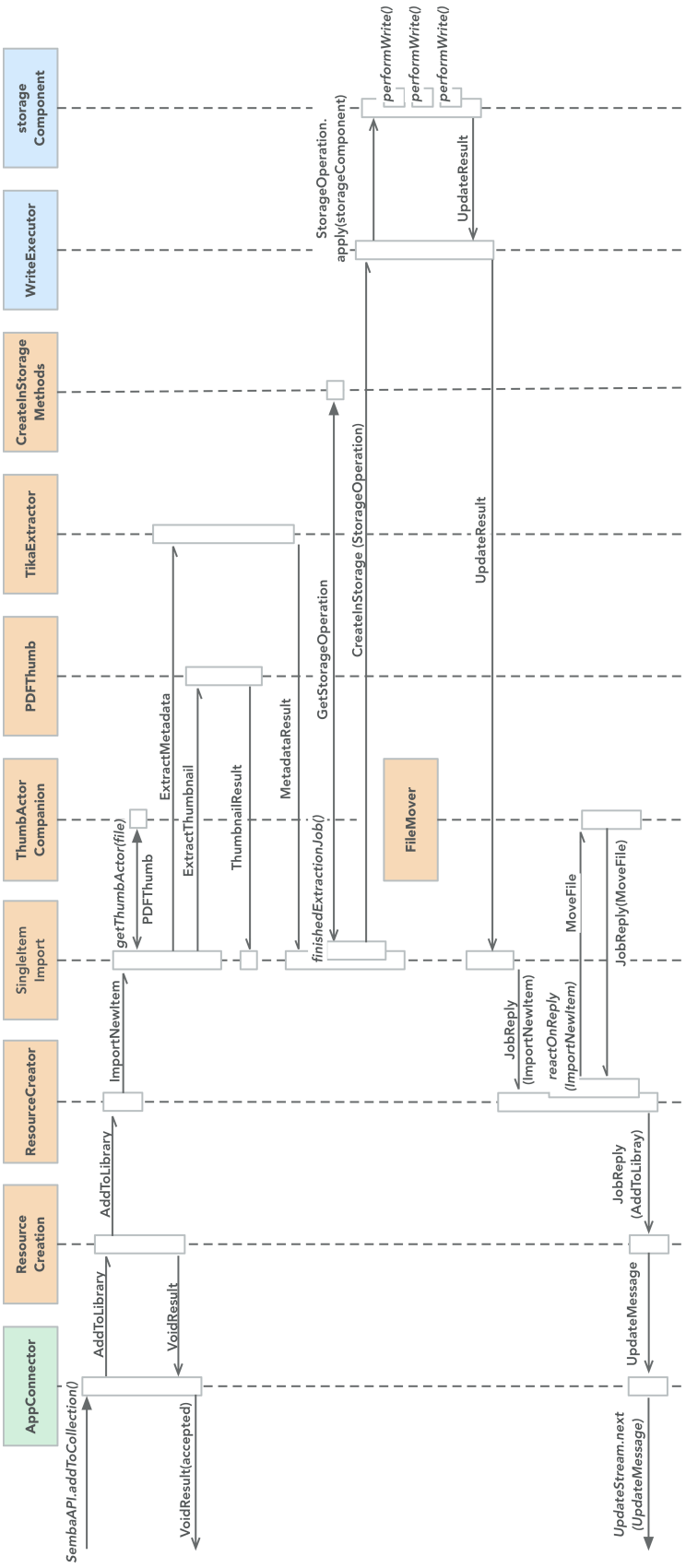


Figure 5.5. Sequence Diagram showing the control flow for an *AddToLibrary()* API call for a PDF document. The **Resource-Creation** trait receives the API message and creates a Job that is processed further by the feature logic.

MediaCollection. On initialization, the **ResourceCreator** creates two actor pools of import executors; **SingleItemImport** and **SingleCollectionImport**. Depending on the content of **AddToLibrary** the **Job** is propagated to either one of these actor pools. If an item path links to a folder, it is recursively scanned and a **ImportNewItem Job** is created for each file. In this case a single **ImportNewItem** is sent to the **SingleItemImport** pool.

After a successful import, the **ResourceCreator** receives a **JobReply** from the executor. The *reactOnReply()* function creates a subtask for moving the source file of an item to the newly created resource folder in the library data store. After the subtask is finished all results are returned to the **ResourceCreation** trait.

SingleItemImport Logic

Importing a new file can be partitioned into a two step process with three logical operations. The first step is file analysis or *Data Extraction*. Existing metadata of the file needs to be read and processed for adding it to the ontology. To provide the front end with a displayable representation of the **Media Item** a thumbnail needs to be created. After this data has become available, it has to be transferred to the ontology in a second step.

The **SingleItemImport** implementation reflects this process by creating a new **ExtractionJob** cluster upon receiving an import message. An **ExtractThumbnail** and a **ExtractMetadata** subtask are created for the **ExtractionJob** and distributed to the respective extraction components.

After all subtasks of an **ExtractionJob** are finished, the overridden *finishedJob()* function from the **JobHandling** trait aggregates the results, creates a **CreateInStorage** storage operation, and sends it to the query pipeline.

Data Extraction

Thumbnail Creation:
As stated in functional requirement **FR-5.1**, users can import arbitrary files. To simplify the extension of supported file types for thumbnail generation, an abstract

class **ThumbActor** is introduced. It contains the logic for handling **Jobs** and creating required directory paths. Concrete **ThumbActors** just need to implement a `def createThumbnail(thumb: ExtractThumbnail): ThumbnailResult` function that generates a JPEG picture and writes it to the provided path. Currently, SeMBa supports thumbnail generation actors for PDF, pictures and text files. Other file types are annotated with a default thumbnail. A VLC based video/audio thumbnail extractor was part of SeMBa'13 and could be easily ported to the new architecture.

A companion object of the abstract **ThumbActor** ensures, that the PDF document that is imported in Diagram 5.5 is sent to the **PDFThumb** instance. The object registers each concrete **ThumbActor** implementation as a variable, following the pattern `MIMETYPES_*DOCUMENTTYPE* = (Props[ConcreteActor],Set(CompatibleMimetypeStrings)`. During initialization for each implementation a pool of ten actors is created, using the provided **Props**. The static method `getThumbActor(file: File ): ActorRef` is used to analyze the file and return a reference to the responsible actor. If no implementation is available a **GenericThumbActor** is returned. It provides the default thumbnail image for the given file type.

Regarding the imported PDF document, the **PDFThumb** actor uses the Apache PDFBox library (Apache-Foundation, 2014) to load and render the first page of a document. The resulting **ImageBuffer** is resized to the configured thumbnail dimensions and gets written to disk.

File Analysis:
SeMBa employs the Apache Tika library (Apache, 2017) to analyze the type and descriptive metadata of files. Tika is a Java library designed to parse arbitrary files and provides interfaces to extract text contents, language, metadata or file types. As seen in Code Example 5.10, the file is analyzed and an **ItemDescription** is returned. Instead of URIs the metadata names are preserved in the **ItemDescription.metadata** map. This facilitates possible filtering of the used data or or type conversion based on ontological definitions for certain metadata types at a later point in the import process.

Code Example 5.10. Apache Tika is used to extract a **Metadata** object containing a Map of *Tag-Value* pairs. Metadata are converted into a SeMBa **ItemDescription** and propagated to the import process.

```
1  def analyzeFile(src: File): ResultContent = {
2      val parser = new AutoDetectParser()
3      val metadata = new Metadata()
4      val stream: FileInputStream = new FileInputStream(src)
5      var description = new ItemDescription()
6      //Parse file and add extracted information to metadata object
7      parser.parse(stream, new DefaultHandler, metadata)
8
9      //Map metadata to Annotationvalue and add K-V pair to return value.
10     metadata.names().foreach( x => description = description.addMetadata(
11       (x, new AnnotationValue(metadata.getValues(x)))))
12
13     //If no title was extracted use filename without extension as name
14     description = description.update(_.name :=
15       Option(metadata.get(TikaCoreProperties.TITLE))
16         .getOrElse(TextFactory.omitExtension(src.getAbsolutePath)))
17     stream.close()
18     MetadataResult(description)
19   }
```

CreateInStorageMethods Singleton

After all preceding tasks have been completed, the **SingleItemImport** actor instantiates a **CreateInStorage** storage operation. Code Example 5.11 shows the definition of the message and a condensed version of the partially applied function *createInStorage()* that is executed in the model layer. The execution of *createInStorage()* can be separated in three steps, each modeled as a distinct *Storage.performWrite()* transaction. It returns an **UpdateMessage** with the **UpdateType**.ADD.

1. **Create a Resource individual in the ontology:** By calling *AccessMethods.createItem()* a new individual of the provided **OntClass** is created. This can be a subclass of **MediaItem** or **MediaCollection**. The item is annotated with the required datatype properties for its source file location, thumbnail lo-

cation and name. After creating the individual it is converted to a **Resource** SeMBa datastructure. The return value is added to the **UpdateMessage**.

2. **Generate new MetadataTypes in the terminological component:** Each extracted metadata tag is passed to *AccessMethods.generateMetadataType()*. If the name does not already exist as a **GeneratedMetadata**, it is transformed into a valid URI and a new class is created. Only in this case the function returns an **Option[OntClass](Some)**, otherwise the value is **None**. All newly created **MetaTypes** are converted into SeMBa **Annotations** and added to the **LibraryConcepts** variable of the **UpdateMessage**.
3. **Add hasMetadata statements to the Resource:** All *Tag–Value* pairs of the extracted metadata are converted to valid *MetadataType–Value* pairs and added as ***Resource***–*hasMetadata*–***MetadataType*** and ***MetadataType***–*hasValue*–*Value* statements to the assertional component of the ontology. The resulting **ItemDescription** is added to the **UpdateMessage**.

Code Example 5.11. The **CreateInStorage** request and the corresponding function are part of the **CreationStorageMethods** singleton. They combine different **AccessMethods** to add a new **Resource** to the ontology and annotate it with the extracted metadata.

```
1  case class CreateInStorage(itemType: ItemType, ontClass: String, fileName: String,
2   desc: ItemDescription, config: LibInfo, thumb_path: String, id: UUID)
3    extends StorageWriteRequest(createInStorage({ /*parameters*/}, _ ))
4
5  def createInStorage(itemType: ItemType, ontClass: String, fileName: String,
6        desc: ItemDescription, config: LibInfo, thumb_path: String, id: UUID,
7        storage: SembaStorageComponent): JobResult = {
8    var update = UpdateMessageFactory.getAddMessage(config.libURI)
9    update addItems storage.performWrite(
10         AccessMethods.createItem(storage.getABox, desc.name, ontClass, fileName,
11          config, thumb_path, id)})
12   //Iterate over all Metadata names, call generateMetadataType,
13   //add to map if new property is returned.
14   update = update withConcepts storage.performWrite({
15        itemDescription.metadata.keys.map(key =>
16          AccessMethods.generateMetadataType(key, storage.getTBox, config))
17            .foldLeft(Map.empty[String, Annotation])
18              case (annotations, property) =>
19              if (property.isDefined) annotations.updated(property.get.getLocalName,
20                DatastructureMapping.wrapAnnotation(property.get, config))
21              else annotations})
22   {/* Update ItemDescription to contain URIs instead of tags.
23    *Add DataPropertyValues by calling AccessMethods.updateMetadata*/ { ...}}
24   JobResult(StorageWriteResult(update))}
```

Using the item import process as a detailed example underlines the design choice of defining both, the **AccessMethods** and the feature-related **StorageOperations** as singleton objects. The **CreateInStorageMethods** messages and functions are also used by the **SingleItemImport** with differing parameters. The employed **AccessMethods** are reused by the **ItemModification StorageOperations** to update existing **MetadataTypes**.

With execution of the **CreateInStorage** message, the import process is completed.

The **UpdateMessage** is part of the **ResultArray** referenced by the original **AddToLibrary** API request, extracted and propagated by the **AppConnector**.

5.4.3 Other Feature Implementations

To implement the requirement groups **FR-3** & **FR-4** *Annotation*, **FR-9** *Filtering* and **FR-10.4** *Remote API: Acquire the current state of the Library*, three additional features have been designed. This section presents their main concepts, the basic architecture is similar to **ResourceCreation**.

AccessToStorage

The **AccessToStorage** feature is responsible for initializing the model layer and passing the fundamental read requests directly to the query pipeline. During startup it instantiates the actors building the model layer and registers it with the query pipeline. Basic API functions such as *openLibrary()*, **getCollectionContent()** or *getMetadata()* are implemented as atomic **AccessMethods**. These functions are simply wrapped into a unary function that is required for a **StorageOperation** case class. API requests carry the class type to be extracted after finishing a Job for easy return type extraction from the **ResultArray**.

Search

Apart from the Java API used by the **AccessMethods** implementation, Jena contains ARQ, a query engine for processing SPARQL Protocol And RDF Query Language (SPARQL) requests (Prud, Seaborne, et al., 2006). With SPARQL, RDF datasets can be accessed or modified, using a pattern of triple statements. Code Example 5.12 shows a query that selects all **Collections** that are a **Lecture** and contain a **CollectionItem** that links to a certain **Resource** and has an preceeds **CollectionRelation** to an arbitrary **CollectionItem**. By building statement patterns, SPARQL queries can contain a certain level of reasoning logic themselves. This minimizes the amount of model reasoning and fosters performance.

Front end developers can offer users a intuitive solution to build own queries or just provide a series of pre-composed query schemes. The API function for a query filter

is passed a **SparqlQuery** object containing the query string and a list of the variable names that need to be returned. Adhering to the feature design, the **SparqlSearch** actor relays the query to the model layer by creating a **StorageOperation** from the **SearchMethods** singleton.

After a successful query, a **FilterResult** is returned to the client. It contains a list of **ResultEntries** that map the variables to their value for each query result. Regarding Code Example 5.12, the client would receive a **FilterResult** that contains the URI of all **MediaCollections** that are a **Lecture** and contain a **CollectionItem** that represents currentLibrary:item_1 and exemplifies an arbitrary item inside that collection.

Code Example 5.12. A SPARQL query as used in the performance measurements of Chapter 6. Only collections that are a **Lecture** and contain a **CollectionItem** with the listed annotations are returned.

```
1  PREFIX semba: <http://www.hci.uni-wuerzburg.de/ontologies/semba/semba-main.owl#>
2  PREFIX teaching: <http://www.hci.uni-wuerzburg.de/ontologies/semba/semba-teaching.owl#>
3  PREFIX currentLibrary  <http://www.foolib.de/lib#>
4  SELECT ?collection
5  WHERE {
6    ?collectionItem  a  semba:CollectionItem;
7      semba:linksToMediaItem  currentLibrary:item_1 ;
8      teaching:isExampleOf ?anyItem;
9      semba:containedByCollection  ?collection .
10   ?collection  a  teaching:Lecture.
11 }
```

The current implementation also allows to perform free text searches for literal values. A future implementation could exploit the Jena ARQ compatibility with Apache Lucene (Białecki, Muir, Ingersoll, & Imagination, 2012), an indexing library that facilitates high performance free text search.

ItemModification

All requirements regarding the modification of existing ontology statements are implemented as part of the **ItemModification** feature. It covers updating of **Resource**

metadata, creation and removal of relations or deleting items from the library. As these storage operations alter the state of the model, all **ItemModification** requests return an **UpdateMessage**.

5.5 Conclusion

This Chapter covered the final implementation of the concept presented in Chapter 4. The main difference to the first iteration – the model layer implementation – was shown by discussing the Jena in-memory model. In its current state, SeMBa provides a framework for developing new or modify existing features. Developers are not required to implement complex message flow logic, as it is handled by the presented **ActorFeatures** and the **JobHandling** trait. The implementation facilitates the design of complex process sequences by introducing clustered tasks and quasi callbacks via the *reactOnReply()* function.
The process of implementing further functionality is reduced to message definition for gRPC, **AppConnector** adaptation and sealed implementation of the new feature module. Reusable behavior may be added to the **AccessMethods** singleton. Regarding the exchangeable storage solution, the **SembaStorageComponent** offers an interface for adding arbitrary solutions, for instance SQL or noSQL mappings of RDF triple stores.
These key points help in fulfilling requirement group **NFR-4** *Maintainability* for an extensible software framework. The requirement group **NFR-1** *Performance* will be tested and discussed in Chapter 6.

6 Application Testing & Benchmarks

In order to validate the implementation of the requirements specified in 3, two different testing approaches were chosen. Functional requirement specifications are covered by a client-side blackbox test suite. Aside from serving as a system test for the initial requirements, it may also be used as a smoke test for future feature integrations. Testable non-functional requirements, namely requirement group **NFR-1** *Performance*, were measured in a series of benchmarks. This chapter covers the implementation and execution of both testing approaches as well as a discussion of the results. However, the specification and implementation of a formal test cycle, including automated unit-, integration-, system- and user acceptance tests is beyond the scope of this thesis. The implemented tests are rather to be seen as a means to ensure general usability of the API. They also provide a reference implementation of the API functionality for future developers.

6.1 Client-Side Blackbox Tests

SeMBas gRPC component provides client API implementations for ten different programming languages. Additional compilers such as the employed ScalaPB add more possible target languages. These client implementations serve as the only front end for the media management capabilities of SeMBa. Thusly, automated system tests need to be executed on the client side, instead of mocking API calls, to cover all layers of the SeMBa core application.

6.1.1 Scala Client and Model Implementation

To reflect the use case of a front end application that is based on the SeMBa data model, a rudimentary client data layer and API connection was implemented in Scala. An architectural overview of the implementation is shown in Figure 6.1. The

abstract class **AbstractSembaConnection** handles the connection to the remote server application. It implements all functions that are provided by the generated client stub. All direct API calls are executed using the blocking RPC implementation to provide immediate feedback to the client. In case of non-state affecting functions the return value is a SeMBa datastructure containing the requested data. For state affecting functions a **VoidResult** containing the acknowledgment and internal **Job** identifier for the request is returned. Upon completion, the result of the call is propagated asynchronously using an **UpdateMessage**. An update is mapped to the same **Job** id for client-side tracking purposes. Implementations of **AbstractSembaConnection** are required to provide a PartialFunction [UpdateMessage, Any], that is passed to the *subscribeForUpdates()* stream observer. This update function contains the application specific handling of incoming update messages.

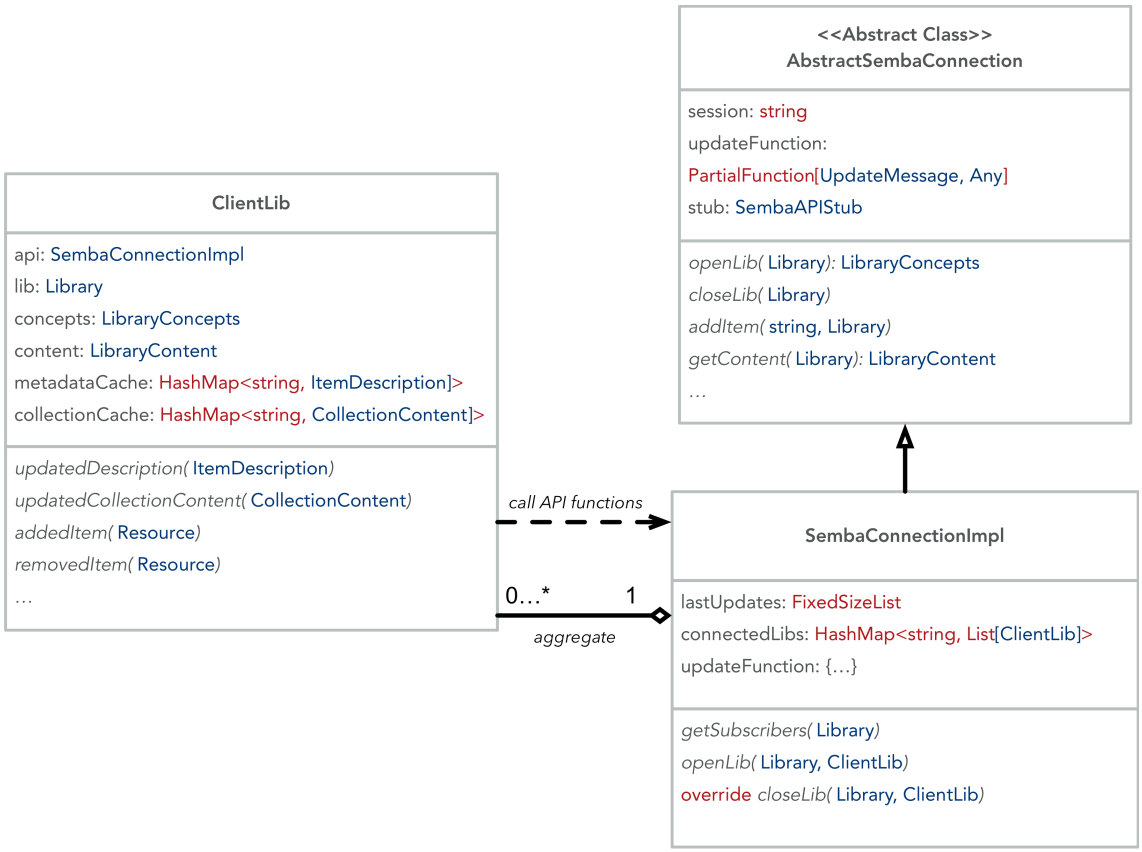


Figure 6.1. The **AbstractSembaConnection** implements all remote procedure calls provided by the gRPC client stub. **SembaConnectionImpl** adds application specific logic such as behavior for incoming **UpdateMessages**. **ClientLib** instances rely on a SembaConnection to keep their model synchronized with the SeMBa model state.

Regarding this reference client implementation, a class **SembaConnectionImpl** ex-

tends **AbstractSembaConnection**. All **ClientLib** instances that represent the state of an individual SeMBa library require an existing **SembaConnectionImpl**. To distribute updates, it maintains a map of all connected **ClientLibs** and the library URI they represent. Code Example 6.1 shows an excerpt of the updateFunction, passing the contents of an **UpdateMessage** to all **ClientLibs** that are registered for the SeMBa library affected by the message. Apart from the update distribution a **SembaConnectionImpl** also adds a history of the latest received **UpdateMessages** and an automated opening and closing of SeMBa library instances based on the currently connected **ClientLibs**.

Code Example 6.1. This partially shown updateFunction is registered as a callback for the UpdateMessage stream. For each incoming update, the relevant fields are passed as an argument to a matching function of each subscribed **ClientLib** instance that updates its internal state. As all Protocol Buffer message fields are **Options**, the *updateValueOperation* is only executed if a field is set.

```
1  val updateFunction: PartialFunction[UpdateMessage, Any] = {
2      case update: UpdateMessage => {
3        val subs = getSubscribers(update.lib)
4        update.kindOfUpdate match {
5          case UpdateType.REPLACE => {
6            updateValueOperation(subs, update.descriptions,
7                (s: ClientLib, v: ItemDescription) => s.updatedDescription(v))
8            updateValueOperation(subs, update.items,
9                (s: ClientLib, v: Resource) => s.updatedItem(v))}
10           ⋮
11       }
12       lastUpdates.append(update)}}
13
14   def updateValueOperation[T, A <: ClientLib](subscribers: List[A],
15                       values: Seq[T], f: ((A,T) => Unit)) = {
16     values.foreach( value => subscribers.foreach(subscriber => f(subscriber, value)))
17     }
```

To connect to a SeMBa library, a **ClientLib** is instantiated with the file path of a library folder and a SembaConnection instance as parameters. During initialization the object registers itself with the SembaConnection. It also retrieves and stores the **LibraryConcepts** and **LibraryContents** of the SeMBa library.

Apart from containing the contents of a library, a **ClientLib** serves three purposes:

1. Wrapping API calls of the used SembaConnection into more conveniently accessible functions. Required parameters are reduced by retrieving information from the stored **LibraryContent** and **LibraryConcepts**.
2. Defining the methods that are called by the SembaConnections update functions to synchronize the local library with the SeMBa library.
3. Caching of retrieved **ItemDescriptions** and **CollectionContents** to accelerate local access times. Cached representations are also covered by the **UpdateMessage** mechanism.

Each SembaConnection can be used by multiple **ClientLibs** that may represent the same or different SeMBa instances. Multiple SembaConnections can be connected to one SeMBa server. This design facilitates convenient implementation of front ends, test suites or console applications by granting synchronized library access through a single object.

6.1.2 Testsuite Implementation

Based on the described client model, a series of test cases was implemented using the ScalaTest framework (Venners, 2017). All test cases are sequentially executed on the same **ClientLib** instance to closely reflect actual user behavior. To underline this approach, the test case design resembles a behavior driven testing (Solis & Wang, 2011). Testcases are implemented as user *scenarios* and aggregated into *features* that mirror a requirement group. Each individual test case is structured using a *Given - When - Then* syntax that facilitates traceability of the individual steps. It also allows new developers to quickly adjust to the application design and requirements. Code example 6.2 shows the implementation of the scenario *"Adding Metadata to an Item"* of the feature *"Metadata Modifications"*. As this is an asynchronous operation, the eventually operator checks the SembaConnection update history in regular intervals. If the id of a received update matches the call id, the test assertions are carried out. If there is no update within a defined period of time, the testcase fails.

Code Example 6.2. Exemplary implementation of a behavior driven test case. After adding new Metadata to a library Resource, an **UpdateMessage** is expected. If received, the contents of the message as well as the updated model state of the **ClientLib** are checked.

```
1  scenario("Adding Metadata to an Item") {
2          When("Adding a Metadata Tag and updating the name")
3          val add = ItemDescription(name = changedItemName)
4            .addMetadata((updatedMetaKey, newAnnotationValue))
5          val msg = MetadataUpdate().withItem(mediaItem).withAdd(add)
6          var update = Traversable.empty[UpdateMessage]
7          val jobID = testLib.saveMetadata(msg).id
8
9          Then(" an Update Message should be received")
10         eventually {
11           update = clientApi.lastUpdates. filter (x => x.jobID === jobID)
12           update.size should be(1)}
13
14         And("an updated ItemDescription should part of the UpdateMessage.")
15         assert(update.head. descriptions .nonEmpty)
16
17         And(" the local description for this item should contain the added metadata values")
18         val description = openMetadata.get(update.head.descriptions.head.item)
19         assert( description .name === changedItemName)
20         newAnnotationValue.value.foreach(x =>
21           assert( description .metadata(updatedMetaKey).value.contains(x)))
```

6.2 Internal Performance Measurements

To track message flows and execution time inside the SeMBa actor system, the additional actor feature **Benchmarking** was implemented. Using console outputs to measure the execution time of a function is a quick way for gain an insight to its complexity and possible programming errors. However, in a concurrent actor system that constantly handles messages in parallel threads this approach is not feasible. Mapping console outputs that occur in arbitrary order to different API calls requires additional effort, and output messages influence performance themselves. The **Benchmarking** trait can be mixed into any JobHandling and overrides

the *acceptJob* and *finishedJob*() functions. When accepting the **Job**, a the current system time and the UUID of the **Job** are stored in a map. After it has been finished, either internally or if all subtask results have been aggregated, a **BenchmarkResult** containing the start/end timestamps, the **Job** ID and name, the original ID of the API call and the class of the processing actor are sent to a **BenchmarkActor**. In case of the SeMBa implementation, there is a configuration setting in each libraries config.xml. If the setting is activated a **BenchmarkActor**, responsible for this library, is created and propagated to all **Benchmarking** actors. During runtime it aggregates a map of all API calls and the respective list of **BenchmarkResults** for this call. When receiving a **WriteResults** message, the data is parse to an XML format and written to disk. The structure of a benchmark result XML can be reviewed in Code Example 6.3. The chosen XML schema can be imported into Microsoft Excel for Windows which offers filtering based on all contained XML values, fast metric calculation like waiting times for individual messages the average processing time of a **Job** at a certain actor.

Code Example 6.3. Results of a SeMBa performance measurement, serialized an XML format. The schema can be imported into Microsoft Excel for further filtering and visualization.

```
1  <Benchmark date="05.05.17 00:51">
2    <Job name="OpenLib" id="3578b092-d121-4699-a878-fb20fd9c28a1">
3        <BenchmarkResult>
4        <JobID>3578b092-d121-4699-a878-fb20fd9c28a1</JobID>
5        <JobDescription>OpenLib</JobDescription>
6        <MeasuredAt>LibraryInstance</MeasuredAt>
7        <AcceptedAt>132229047213182</AcceptedAt>
8        <FinishedAt>132229189607545</FinishedAt>
9        <TimeInMillis>142</TimeInMillis>
10     </BenchmarkResult>
11         ⋮
12   </Job>
13   <Job name="RequestContents" id="16cd3d01-7074-4f7a-bea3-6f6a2069d07a">
14     ⋮
15   </Job>
```

6.3 Results

All tests described in this section were executed on an Intel Core i5 2600K with 8GB RAM running Windows 10. Functional tests runs were also performed on a 2011 MacBook Air running macOS Sierra. The used library contained ~10.500 individual items described in 2.5 million asserted RDF statements. This included 10.100 **MediaItems** and 400 **MediaCollections**. All collections were filled with 20 **CollectionItems** representing randomly selected **MediaItems**. Each **CollectionItem** was assigned 5 random connections to random **CollectionItems** inside the same collection, resulting in 40.000 **CollectionRelation** statements for the library. The overall specification for test execution is explained in the following paragraphs and listed in Table 6.1.

Figure 6.2 shows the result of an individual functional test run using the TDB storage solution. Apart from functional integrity, test case execution times show, that the functional requirements for synchronous function calls have been met. Even with the added overhead of library maintenance, testcase setup and assertions, these times are within the required range of **NFR-1** *Performance*. However, abnormal spikes of up to eight seconds in the item import process required the use of the more granular **Benchmarking** solution.

The internal performance benchmark was carried out using the same library containing 10000 individual items to simulate a database size that is expected in a production environment. Motivated by the ScalaTest results, three scenarios where devised. Each scenario was performed for at least 1000 cycles to measure the average execution times. Initially these scenarios were intended to be tested on models, the triple store as well as the in-memory model. However, the latter solutions lack of performance made these measurements obsolete. While importing 1000 files was feasible, SPARQL queries for libraries >500 individuals did not return a result in 24 hours. The threshold of 200ms for a simple query was reached with not more than 10 individuals. This critical flaw lead to the decision to abandon the in-memory model altogether and revise the software architecture. In the following, the three scenarios, testing results and consequences for development are discussed. In-memory measurements did not perform the same number of cycles and are just for reference.

Similar to the Berlin SPARQL benchmark performed by Bizer and Schultz (2009),

a pool of test data was defined for each measurement to compensate for the caching algorithm of the TDB triple store. The found results are not comparable to this paper, as the SeMBa ontology is subject to more dynamic ontology changes and reasoning processes.

Measurement Graphics

Table 6.1. Specification of the SeMBa library and test environment for execution of functional and performance related tests.

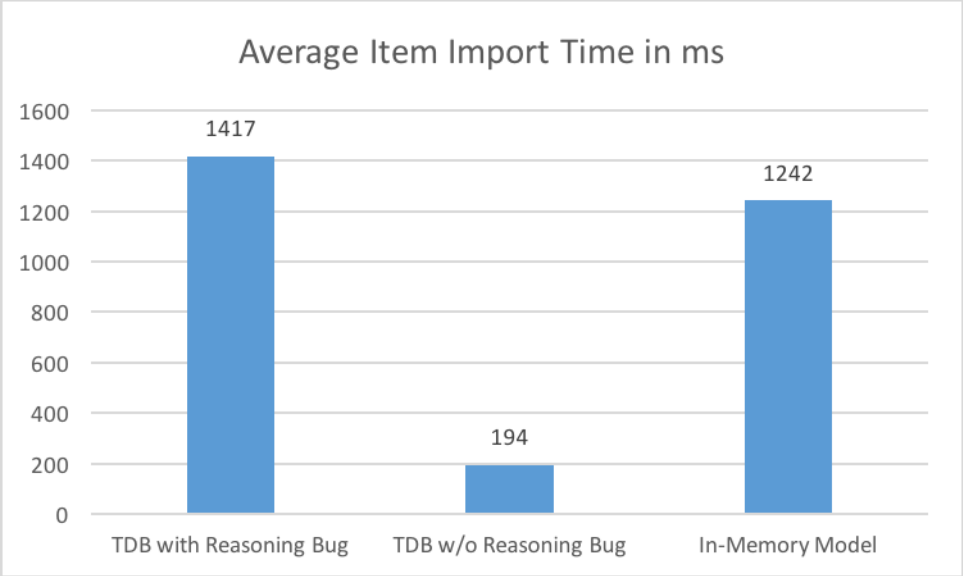
Benchmark Configuration	
Test Environment	Windows 10 PC with Intel Core i5 2600K processor and 8GB RAM
Size of TDB index	1.7GB size on disk with 2.5 million asserted/inferred RDF statements
MediaItems	10500 imported items based on a set of 10 binary and raw text source files.
Resource Annotations	347 unique **GeneratedMetadata** classes with an average of 25 values per **MediaItem**
MediaCollections	400 MediaCollections that contain 20 CollectionItems each.
CollectionRelations	5 outgoing connections per CollectionItem resulting in 40.000 **CollectionRelations**
Iterations	10 functional test runs, 1000 API calls per performance scenario

ClientTest: 17 total, 17 passed 12.50 s

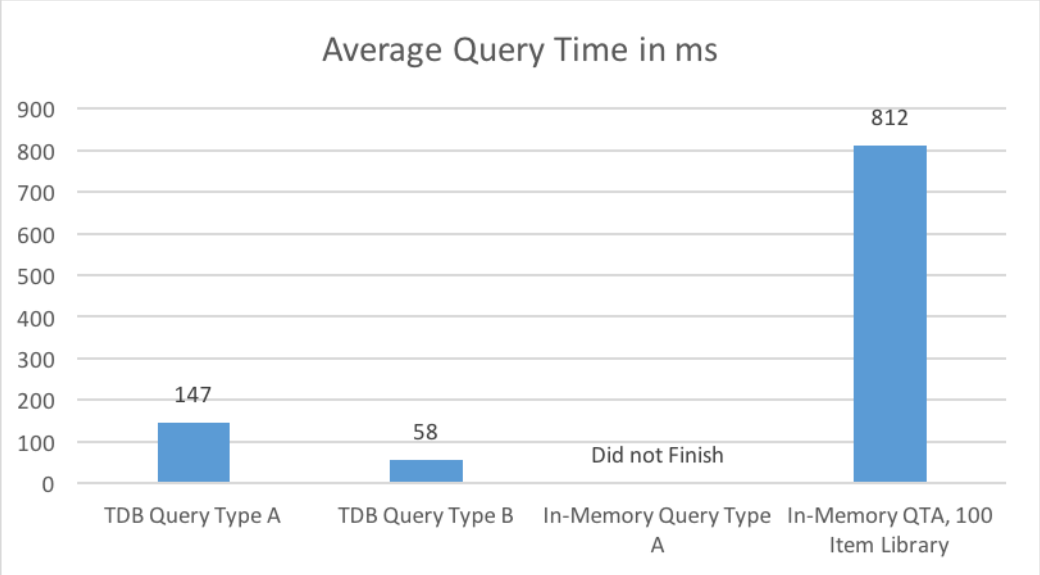
Collapse | Expand

ClientTests		12.50 s
Feature: SeMBa Initialization		4.69 s
Scenario: The client registers a Session on the remote server	passed	500 ms
Scenario: The client requests to open a remote library	passed	4.19 s
Feature: Item Creation		1.83 s
Scenario: Item Import	passed	1.79 s
Scenario: Collection Creation	passed	41 ms
Feature: Collection Handling 1: Setup		399 ms
Scenario: Adding an Item to a Collection	passed	127 ms
Scenario: Adding an Item item twice to a Collection	passed	99 ms
Scenario: Connecting two Items in a Collection	passed	88 ms
Scenario: Adding a second Connection between two Items	passed	85 ms
Feature: Metadata Modifications		346 ms
Scenario: Metadata Retrieval	passed	31 ms
Scenario: Adding Metadata to an Item	passed	154 ms
Scenario: Removing Metadata from an Item	passed	161 ms
Feature: Sparql Search		31 ms
Scenario: Simple Search for name	passed	15 ms
Scenario: Filter Library Contents for MediaItems that are part of a Collection with a certain Relation	passed	16 ms
Feature: Collection Handling 2: Removal		492 ms
Scenario: Removing a connection between two Items	passed	109 ms
Scenario: Removing a CollectionItem which is a connection destination	passed	221 ms
Scenario: Removing a MediaItem Item which represents a CollectionItem	passed	162 ms
Feature: Concurrency Check		4.71 s
Scenario: Adding an Item using a second Client Connection to the same library	passed	4.71 s

Figure 6.2. Test run protocol for a SeMBa library containing 10.500 items and 2.5 million statements.



(a) Average import time for 1000 iterations of a folder containing 10 files. File types were mixed binary and text based files, with about 350 distinct metadata tags.



(b) Average execution time for SPARQL query types A (search for a given metadata value) and B (search for all collections that contain a given media item with a given relation). The in memory model did not complete the query with a library containing 10000 individuals. For comparison, in-memory query time for 100 individuals is shown.

Figure 6.3. Results of two benchmark scenarios, performed on an intel i5 2600K with 8GB Ram. The library contained 15000 individuals upon measurement execution.

Item Import Performance

The testcase execution times for adding new items to a SeMBa library showed to be inconsistent, with values ranging from two to eight seconds. To investigate this low performance and especially the high variance of the values, the import process was logged for 100 API import calls of a folder containing 10 files. A review of the results in Excel showed that thumbnail generation and metadata extraction processes are constant for a single import. However, the *createItemInStorage* method at the **StorageWriteExecutor** varied drastically in its execution times. In combination with the single writer policy this lead to an additional bottleneck for batch imports, as subsequent **StorageWriteRequests** had to wait for the lock to release.
Further analysis of the involved functions revealed the creation of metadata value statements to be origin of the delay. For each value a new statement is added sequentially to the ontology model. Most steps that are part of the creation process require a reasoner to add inferred statements about class relationships or relations to the model. In this case, the bound reasoner infers new statement after each change to the ontology. However, for the simple object-to-value relations of the metadata model, this is not required. Nevertheless, the process was performed with an active reasoner, that was triggered after every addition of the up to 40 extracted metadata values. By disabling the reasoning process for metadata addition, execution times were decreased from average an 1417ms to 194ms per item. Without the detailed timings of the benchmark results, tracing this origin would have been more time-consuming. Figure 6.3a shows the average import times per media item with activated and deactivated reasoner.

SPARQL Query Performance

Filters can be applied to SeMBa library for various reasons. As the results are generally used to change the view of a library, they are executed synchronously. Nevertheless, a fixed requirement for execution times below 200ms may not be reached in all cases. Pérez, Arenas, and Gutierrez (2006) elucidate that the complexity and statement sequence of a query has a high influence on its duration. For this benchmark two rather simple queries, that match the use case of a library filter, were defined. Query type A is a search for items that contain a certain value for a

metadata tag, similar to a free text search. Query type B returns all **MediaCollections** that contain a **CollectionItem** that represents a **MediaItem** X and contains a relation of type Y to an arbitrary destination object. For each type 1000 cycles were performed with query variables being taken randomly from a pool. Figure 6.3b shows the measurements for both query types.

The results show, that the SPARQL performance of TDB is sufficient for the SeMBa performance requirements in the context of these simple, use case related queries. However, different queries or growing library content may decrease performance in some cases.

Metadata Retrieval Performance

A third series of measurements was executed to test the stability of one of the most relevant API calls of SeMBa. Requests for the metadata of a library resource might be executed by a graphical front end whenever a user hovers the mouse over an item representation. To verify the results of the automatic test case execution, the metadata of each item in the test library were requested iteratively. Measurement evaluation showed, that the average execution time for the request is 11ms, with all measurements ranging between 7ms and 66ms. Compared with 1000 looped requests on a library containing 10 items, changes are negligible. All measurements are within the range defined by **NFR-1.2** *"Synchronous remote function calls shall be completed in under 200ms."*.

6.4 Conclusion

This chapter described the implementation and execution of functional and performance related test environments for SeMBa. The measurements provided by the **Benchmarking** trait assisted in the analysis and removal of a performance bottleneck.
The deactivation of the reasoning process during the metadata addition does not affect the external application state, as each model transaction is bound to its own reasoner. Since the the process that adds the metadata statements has no access to any inference the use of some Jena **OntModel** convenience functions is inhibited and individual statements have to be added using the **Model** abstraction. As the

data property **hasValue**, that connects a value to a **SembaMetadataType** has no implications on any reasoning process, the overall integrity of the model for read-requests is not affected.
After this issue has been solved, all results are within the expected range and conform to the non-functional requirements. Further implementation of the in-memory storage model was stopped after first measurements revealed its critical inability to perform low latency SPARQL queries.

7 Conclusion

7.1 Results

The main motivation for this project was to develop a maintainable back end application for use in a teaching related content management and authoring toolchain. This thesis presented SeMBa, a software framework that facilitates semantic media management for arbitrary front end applications. The current state in semantic file management in regard to teaching applications has been discussed. Based on this discussion, the requirements for a semantic media back end with a focus on lecture authoring were elicited. A software solution for a generic, extensible framework was conceptualized and implemented in Scala, using Apache Jena as an RDF representation library. Domain specific concepts can be added to the used ontology model to extend functionality. The resulting software underwent functional and performance related tests to minimize further development efforts, especially regarding front end applications. The logic implementation is designed to foster the development of additional features, with a focus on code reusability.

SeMBa facilitates the annotation of arbitrary files based on the concepts defined in an OWL base ontology. Each SeMBa library loads an individual ontology which imports the base and additional domain specific ontologies extending it. The resulting pool of object-to-object and object-to-value properties provides annotations for files managed by the application. To store the terminological and assertional RDF statements, two different storage solutions were implemented and compared. Apache Jena TDB emerged as the more reliable solution in terms of performance and code complexity.

The Protocol Buffer & gRPC API supplies remote procedures and semantically grounded native data structures for multiple programming languages. A Scala reference implementation of the client side model based on these data structures was presented in chapter 6. The client application model is synced with the back end model and provides simple abstractions of the API functions.

To further improve SeMBas maintainability, the logic layer is designed around a job handling logic that regulates message flows and result aggregation for API calls. The stackable trait pattern facilitates modular implementation of individual functional clusters and *'plug-n-play'* addition of modules to library instances. For the analysis of message flows and processing times for each involved actor, a benchmarking module can be activated during development. Benchmark results are serialized as XML files. The XML schema is compatible with Microsoft Excel for improved visualization and processing.

7.2 Future Work

SeMBa is intended to be an extensible framework with multiple areas of use, but mainly targeted at the EiS and SMARTAPS project. In this context, three different categories of future work can be identified.

7.2.1 Client Applications

Main Application GUI & Collection Editor

Visualizing the library contents and annotations is the next logical step of the SMARTAPS development process. The console application and automated tests showed that a client application can provide stable, synchronized and reasonably fast access to the SeMBa data model. Data bindings between client model objects and graphical elements can be used to build reactive GUI applications without the need for extensive business logic. A combined collection editor for EiS and SMARTAPS will be developed as part of an ongoing masters thesis (Korwisi, Work in Progress). The application is implemented in node.js and deployed as a web server on the same machine as the SeMBa server application. Users can connect using their web browser and collaboratively work on semantic collections or import new assets.

Build Application for Converting MediaCollections

A core requirement for the SMARTAPS toolchain is the conversion of semantically connected **MediaCollections** into a unified presentation format. After deciding on a format and defining a conversion process, an application can be developed to execute this process, using the information available to the client model. As opposed to developing this build tool as a server feature, an external application that takes use of the API may run on any remote machine in arbitrary parallel instances without affecting the server performance. Additionally, an external build application would not be subject to possible changes in the server model, the required testing pipeline and a deployment process. This facilitates rapid development iterations on a fixed API functionality.

7.2.2 Internal Improvements

Extensive Performance Analysis:

With the help of the **Benchmark** trait, some existing bottlenecks, such as item import, have already been identified and fixed. The feature offers a tool for tracing message flow and latency inside the actor system. The definition of further metrics, such as time spent in mailbox queues, could help locating further optimization possibilities. Prioritized mailboxes might be investigated to ensure quick processing of **JobReplies**, although an actor might be clogged by multiple **Jobs**. This possibility has not been tested yet.
Another influence on performance is the size of the used actor pools for accelerated batch processing. With to many parallel processes running on CPUs with limited cores, the scheduling overhead outweighs possible performance improvements. However, especially for the **WriteExecutor** decreased processing times could be observed, although, the locking mechanism limits database access to one actor at a time.

Application Security

gRPC already offers the functionality to limit access to SSL connections with individual client authentification. In order to make SeMBa securely accessible via

the World Wide Web, the **AppConnector** and client implementations have to be adapted to implement a user authentification protocol.

Implementation of a Formal Testing Cycle

If the results of this work are to be integrated into a newly developed toolchain for asset management at the chair of HCI, application resilience is a critical factor. Setting up a formal testing procedure would ensure that updates to the software do not compromise the asset library. As the project is already hosted on GitLab, an integration pipeline could facilitate continuous integration testing. The existing smoke test could be enhanced to include boundary tests and invalid inputs.

Graphical User Interface for Server Application Management

At this point the SeMBa server application is configured using XML files for application wide and library dependent settings. Internal functions, such as writing benchmark results, are executed via the remote API by using a **SembaConnection** instance in the command line shell. A GUI application would allow live monitoring of benchmark results and statistics, changing application configuration or the variety of enabled features for a library. It could be directly attached to the application instead of using the public API.

Exploration of Additional StorageComponents

This thesis tested two different solutions for persisting the SeMBa ontology model. Apache Jenas **Graph** interface provides an easy to use interface for implementing additional storage solutions. The Berlin Benchmark (Bizer & Schultz, 2009) suggests, that mappings to a relational database like MySQL might increase performance. Additionally, the choice of reasoning capabilities has a major influence. A custom reasoner based on a small set of required rules that replaces the currently applied OWL rule set should be able to increase overall performance of the application.

Semantic Type Annotations for Metadata

Currently, all metadata values are stored as string literals inside the SeMBa ontology. By adding a type annotation for each metadata class, values could be mapped to the correct data type during retrieval. ScalaPB allows the addition of implicit conversion parameters to Protocol Buffer definitions. By exploiting both functionalities, server side logic would be enabled to process, filter or sort numerical values as intended.

Extension of the SeMBa Base Ontology

Existing ontologies, such as the Dublin Core terms or "Friend of a Friend" (FOAF) (Brickley & Miller, 2007), could add capabilities to the base ontology. The Dublin Core ontology adds descriptions and structure for all DC metadata tags. FOAF may be used to describe author relations for collections as well as imported source materials.

7.2.3 Feature Additions

Additional Thumbnail Generators

The **ResourceCreation** feature creates thumbnails for raw text, images and PDF files. With research of efficient libraries for retrieving Office and multimedia thumbnails, the implementation of additional generators only requires extension of the **ThumbActor** class and definition of a *createThumbnail* method. This would enhance the user experience of front end applications with low effort.

Free Text Search Index

Apache Jena is prepared to attach an Apache Lucene instance to create a free text index for all ontology statements. This index increases query performance for free text searches, such as filtering based on metadata values. There are certain hurdles for the implementation in a dynamic ontology with constant write operations, as the index would require periodical updates. Benefits and drawbacks of this approach need to be evaluated.

Content Analysis with Tika

Aside from retrieving metadata and file type information, Apache Tika can also be used to retrieve the content of text based file formats. Especially with the addition of a free text search engine, this capability could be used to filter library items based on their actual content. An exaggerated version of this approach may use text mining technologies (Aggarwal & Zhai, 2012) to map the text contents to ontological concepts and introduce additional assertions.

Content Exchange Module

As two different SeMBa libraries are based on matching ontologies, the export and import of terminological and assertional statements is possible. An additional module could convert partial storage entries into a serialization format such as Turtle and facilitate the import of these files into the SeMBa ontology. This approach would allow merging contents created on different server applications, for instance on local devices without network access.

7.3 Discussion

Since its beginning in 2011, the Semantic Media Backend has undergone several iterations with four major reimplementations as the result of changed requirements and performance or functional constraints. Over the course of this thesis and the preceding HCI project, it became more and more evident that a sufficient back end application for research and production environments requires a more generic approach than presented in Isermann (2013b). An application that satisfies multiple stakeholders with different functional requirements needs a modular design that is maintainable enough to be extended by varying developers as a team project. This assumption is underlined by the extensive list of possible fields for future work on the application.

The most ambitious part of the development process was to find a set of requirements that fulfills needs of both major stakeholder projects and to define a software architecture that incorporates the required aspects of concurrency, performance and

extensibility. Low coupling, that is inherent with the actor models and gRPC's message based communication, is a cornerstone of this design. Scala's mixin class composition further facilitates the implementation of new features.

The most prominent lesson learned during development is the need to verify the technical feasibility of design and tool choices early on through individual unit and component tests. The first iteration of this SeMBa version relied on the in-memory storage model which was deeply integrated into the entire application structure. Although functional requirements were met, the implementation could not cope with the high amounts of data and low latency requirements introduced during performance testing. This setback lead to a new concept and implementation that is presented in this work. For reference, the earlier version may be consulted on the attached storage medium.

In summary, the current version of the Semantic Media Backend is a semantically grounded media management solution, developed on established design patterns and principles. Its core is an extensible software framework, that allows the addition of domain specific functionality. In combination with a GUI application that binds to the client model and a build-module for *MediaCollections*, it is equipped to provide the data model for the SMARTAPS toolchain.

Bibliography

Aggarwal, C. C. & Zhai, C. (2012). *Mining Text Data*. Springer Science & Business Media.

Apache. (2017). *Apache Tika - A Content Analysis Toolkit*. Retrieved from https://tika.apache.org/

Apache-Foundation. (2014). *PDFBox, Java PDF processing Library*. Retrieved May 15, 2017, from http://www.pdfbox.org

Berners-Lee, T., Hendler, J., & Lassila, O. (2001, May). The Semantic Web. *Scientific American*, *284*(5), 34–43. Retrieved from http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21

Białecki, A., Muir, R., Ingersoll, G., & Imagination, L. (2012). Apache Lucene 4. In *Sigir 2012 workshop on open source information retrieval* (p. 17).

Birrell, A. D. & Nelson, B. J. (1984). Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems (TOCS)*, *2*(1), 39–59.

Bizer, C. & Schultz, A. (2009). The Berlin Sparql Benchmark.

Blanc-Brude, T. & Scapin, D. L. (2007). What Do People Recall About Their Documents?: Implications for Desktop Search Tools. In *Proceedings of the 12th international conference on intelligent user interfaces* (pp. 102–111). IUI '07. Honolulu, Hawaii, USA: ACM

Bloehdorn, S., Görlitz, O., Schenk, S., Völkel, M., et al. (2006). Tagfs- Tag Semantics for Hierarchical File Systems. In *Proceedings of the 6th international conference on knowledge management (i-know 06), graz, austria* (Vol. 8).

Bozsak, E., Ehrig, M., Handschuh, S., Hotho, A., Maedche, A., Motik, B., ... Stojanovic, L., et al. (2002). KAON—towards a large scale Semantic Web. In *International conference on electronic commerce and web technologies* (pp. 304–313). Springer.

Brickley, D. & Miller, L. (2007). *FOAF Vocabulary Specification 0.91*. Citeseer.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Chichester, UK: Wiley.

Bush, V. et al. (1945). As We May Think. *The Atlantic Monthly*, *176*(1), 101–108.

Carroll, J. J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., & Wilkinson, K. (2004). Jena: Implementing the Semantic Web Recommendations. In *Proceedings of the 13th international world wide web conference on alternate track papers & posters* (pp. 74–83). ACM.

Chandy, K. M. & Misra, J. (1984). The Drinking Philosophers Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *6*(4), 632–646.

Chirita, P. A., Gavriloaie, R., Ghita, S., Nejdl, W., & Paiu, R. (2005). Activity Based Metadata for Semantic Desktop Search. In *European semantic web conference* (pp. 439–454). Springer.

Corbató, F. J. & Vyssotsky, V. A. (1965). Introduction and Overview of the Multics System. In *Proceedings of the november 30–december 1, 1965, fall joint computer conference, part i* (pp. 185–196). ACM.

Decker, S. & Frank, M. (2004). The Social Semantic Desktop. *Digital Enterprise Research Institute, DERI Technical Report May*, *2*, 7.

Dijkstra, E. W. (1968). Cooperating Sequential Processes. In *The origin of concurrent programming* (pp. 65–138). Springer.

Dijkstra, E. W. (1982). On the Role of Scientific Thought. In *Selected writings on computing: a personal perspective* (pp. 60–66). Springer.

Dijkstra, E. (1965). Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, *8*(9), 569.

Dubochet, G. (2009). Computer Code as a Medium for Human Communication: Are Programming Languages Improving? In *Proceedings of the 21st working conference on the psychology of programmers interest group* (LAMP-CONF-2009-001, pp. 174–187). University of Limerick.

Ehrlich, T. & Frey, J. S. (1995). *The Courage to Inquire: Ideals and Realities in Higher Education*. Indiana University Press.

Faubel, S. & Kuschel, C. (2008). Towards Semantic File System Interfaces. In *Proceedings of the 2007 international conference on posters and demonstrations-volume 401* (pp. 20–21). CEUR-WS. org.

Feldman, S. (2004, March). The High Cost of not Finding Information. electronic magazine. Retrieved from http://www.kmworld.com/Articles/Editorial/Feature/The-high-cost-of-not-finding-information-9534.aspx

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc.

Fowler, M. (2014). *Microservices and the First Law of Distributed Objects*. Retrieved May 15, 2017, from https://martinfowler.com/articles/distributed-objects-microservices.html

Gall, D., Lugrin, J.-L., Wiebusch, D., & Latoschik, M. E. (2016). Remind Me: An Adaptive Recommendation-Based Simulation of Biographic Associations. In *Proceedings of the 21th international conference on intelligent user interfaces (iui)*. ACM.

Gantz, J. & Reinsel, D. (2011). Extracting Value from Chaos. *IDC iView*, 1–12.

Gennari, J. H., Musen, M. A., Fergerson, R. W., Grosso, W. E., Crubézy, M., Eriksson, H., ... Tu, S. W. (2003). The Evolution of Protégé: an Environment for Knowledge-Based Systems Development. *International Journal of Human-computer studies*, *58*(1), 89–123.

Gidt, S. (2015). *SeMBa GUI*. Universität Würzburg.

Google. (2008). *Protocol Buffers: Google's Data Interchange Format*. Retrieved May 15, 2017, from https://opensource.googleblog.com/2008/07/protocol-buffers-googles-data.html

Google. (2017). *Protocol Buffers: Encoding*. Retrieved May 15, 2017, from https://developers.google.com/protocol-buffers/docs/encoding

Gruber, T. R. et al. (1993). A Translation Approach to Portable Ontology Specifications. *Knowledge acquisition*, *5*(2), 199–220.

Guha, R. V. (2013). Light at the end of the tunnel. In *Talk at the 12th international semantic web conference (iswc), sydney* (Vol. 10).

Handschuh, S., Möller, K., & Groza, T. (2007). The NEPOMUK Project – on the Way to the Social Semantic Desktop.

Hewitt, C., Bishop, P., & Steiger, R. (1973). Session 8 Formalisms for Artificial Intelligence A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Advance papers of the conference* (Vol. 3, p. 235). Stanford Research Institute.

Hoare, C. A. R. (1978). Communicating Sequential Processes. In *The origin of concurrent programming* (pp. 413–443). Springer.

Horridge, M. & Bechhofer, S. (2011). The OWL API: A Java API for OWL Ontologies. *Semantic Web*, *2*(1), 11–21.

Hug, T., Lindner, M., & Bruck, P. A. (2005). Microlearning: Emerging Concepts, Practices and Technologies after E-Learning. *Proceedings of Microlearning*, *5*, 3.

Hunt, A. (2000). *The Pragmatic Programmer*. Pearson Education India.

Isermann, E. (2013a, June). *Establishing the Foundation for a Semantic Presentation Toolchain.* Universität Würzburg.

Isermann, E. (2013b, December). *SeMBa: The Semantic Media Backend of SMARTAPS.* Universität Würzburg.

Isermann, E. (2016). *SeMBa - Implementation of a Remote API for Frontend Applications.* Universität Würzburg.

ISO/IEC. (2010). IEEE, Systems and Software Engineering–Vocabulary. *ISO/IEC/IEEE 24765: 2010 (E)) Piscataway, NJ: IEEE computer society, Tech. Rep.*

Knublauch, H. et al. (2006). Protégé-OWL API Programmer's Guide.

Knublauch, H., Fergerson, R. W., Noy, N. F., & Musen, M. A. (2004). The Protégé OWL Plugin: An open Development Environment for Semantic Web Applications. In *International semantic web conference* (pp. 229–243). Springer.

Korwisi, K. (Work in Progress). *Preliminary Title: E-learning in Science (EiS) - an E-Learning Platform for Research Purposes.* (Master's thesis, Universität Würzburg).

Korwisi, K. & Mehn, C. (2016, September). *A management system for the E-learning in Science platform.* Universität Würzburg.

Kwok, S. & Zhao, J. L. (2006). Content-based object organization for efficient image retrieval in image databases. *Decision Support Systems*, *42*(3), 1901–1916.

Lamb, D. A. (1987, July). IDL: Sharing Intermediate Representations. *ACM Trans. Program. Lang. Syst. 9*(3), 297–318

Laplante, P. A. (2007). *What Every Engineer Should Know About Software Engineering (What Every Engineer Should Know).* Boca Raton, FL, USA: CRC Press, Inc.

Maurer, H., Sapper, M., & Vienna, S. (2001). E-Learning has to be seen as part of General Knowledge Management. In *Proceedings of ed-media* (pp. 1249–1253).

Microsoft. (2009). *Microsoft Application Architecture Guide* (2nd). Microsoft Press.

Nash, M. & Waldron, W. (2016). *Applied Akka Patterns: A Hands-On Guide to Designing Distributed Applications.* " O'Reilly Media, Inc.".

Nejdl, W., Wolf, B., Qu, C., Decker, S., Sintek, M., Naeve, A., ... Risch, T. (2002). EDUTELLA: a P2P networking infrastructure based on RDF. In *Proceedings of the 11th international conference on world wide web* (pp. 604–615). ACM.

Nestor, J. R., Wulf, W. A., & Lamb, D. A. (1981). *IDL-Interface Description Language: Formal Description.* Carnegie-Mellon University. Department of Computer Science.

Nilsson, M. (2000). ID3 Tag Version 2.4. 0 – Main Structure. *http//www. id3. org/id3v2*.

Noy, N., Rector, A., Hayes, P., & Welty, C. (2006). Defining n-ary Relations on the Semantic Web. *W3C working group note*, *12*(4).

Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., ... Zenger, M. (2004). *An Overview of the Scala Programming Language*.

Odersky, M., Spoon, L., & Venners, B. (2010). *Programming in Scala, A Comprehensive Guide* (2nd). Artima Inc.

Billionaire baron Bill Gates still mourns Vista's stillborn WinFS. (2013, February). Retrieved May 15, 2017, from http://www.theregister.co.uk/2013/02/12/bill_gates_reddit_ama/

IntelliJ IDEA the Java IDE. (2017). Retrieved May 15, 2017, from https://www.jetbrains.com/idea/

sbt - The interactive build tool. (2017). Retrieved May 15, 2017, from http://www.scala-sbt.org/

Owens, A., Seaborne, A., Gibbins, N., et al. (2008). Clustered TDB: A Clustered Triple Store for Jena.

Pérez, J., Arenas, M., & Gutierrez, C. (2006). Semantics and Complexity of SPARQL. In *International semantic web conference* (pp. 30–43). Springer.

Prud, E., Seaborne, A. et al. (2006). SPARQL query language for RDF.

Razali, R. & Anwar, F. (2011). Selecting the Right Stakeholders for Requirements Elicitation: a Systematic Approach. *Journal of Theoretical and Applied Information Technology*, *33*(2), 250–257.

Rizzo, T. (2004). WinFS 101: Introducing the new Windows File System. *Longhorn Developer Center Home: Headline Archive: WinFS*, *101*, 1–5.

Roestenburg, R., Bakker, R., & Williams, R. (2015). *Akka in Action*. Manning Publications Co.

Rohloff, K., Dean, M., Emmons, I., Ryder, D., & Sumner, J. (2007). An Evaluation of Triple-Store Technologies for Large Data Stores. In *Otm confederated international conferences" on the move to meaningful internet systems"* (pp. 1105–1114). Springer.

Ryan, L. (2015). *gRPC Motivation and Design Principles*. Retrieved May 15, 2017, from http://www.grpc.io/blog/principles

Sauermann, L., Bernardi, A., & Dengel, A. (2005). Overview and Outlook on the Semantic Desktop. In *Proceedings of the 2005 international conference*

on semantic desktop workshop: next generation information management d collaboration infrastructure-volume 175 (pp. 74–91). CEUR-WS. org.

Schmidt, D. C. (1999). Wrapper Facade – A Structural Pattern for Encapsulating Functions within Classes.

Sher, P. J. & Lee, V. C. (2004). Information Technology as a Facilitator for Enhancing Dynamic Capabilities through Knowledge Management. *Information & management*, *41*(8), 933–945.

Shulman, L. (1987). Knowledge and Teaching: Foundations of the new Reform. *Harvard educational review*, *57*(1), 1–23.

Siemens, G. (2006). *Knowing Knowledge*. Lulu.com.

Sigurbjörnsson, B. & Van Zwol, R. (2008). Flickr Tag Recommendation based on Collective Knowledge. In *Proceedings of the 17th international conference on world wide web* (pp. 327–336). ACM.

Slee, M., Agarwal, A., & Kwiatkowski, M. (2007). Thrift: Scalable cross-language services Implementation. *Facebook White Paper*, *5*(8).

Solis, C. & Wang, X. (2011). A Study of the Characteristics of Behaviour Driven Development. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on* (pp. 383–387). IEEE.

Staab, S. & Studer, R. (2013). *Handbook on Ontologies*. Springer Science & Business Media.

Stojanovic, L., Staab, S., & Studer, R. (2001). eLearning based on the Semantic Web. In *Webnet2001, world conference on the www and internet, october 23-27, 2001, orlando, florida, usa*.

Tane, J., Schmitz, C., Stumme, G., Staab, S., & Studer, R. (2003, November). The Courseware Watchdog: an ontology-based Tool for Finding and Organizing Learning Material. In *Fachtagung "mobiles lernen und forschen"*. Kassel. Retrieved from http://www.kde.cs.uni-kassel.de/schmitz/publ/mobillernen.pdf

Tasharofi, S., Dinges, P., & Johnson, R. E. (2013). Why do Scala Developers mix the Actor Model with other Concurrency Models? In *European conference on object-oriented programming* (pp. 302–326). Springer.

Tesic, J. (2005). Metadata Practices for Consumer Photos. *IEEE MultiMedia*, *12*(3), 86–92.

TrueAccord. (2014). *ScalaPB Protocol Buffer Compiler for Scala*. Retrieved May 15, 2017, from https://scalapb.github.io/

Van Buren, M. (2002). Fusion of E-Learning & Knowledge Management. In *Title academy of human resource development conference proceedings (honolulu, hawaii, february 27-march 3, 2002). volumes 1 and* (p. 335). ERIC.

Venners, B. (2017). *ScalaTest*. Retrieved from www.scalatest.org

W3C. (2008). Triple Language. *W3C Team Submission*, *14*.

W3C. (2009). OWL 2 Web Ontology Language Document Overview.

W3C et al. (2014). Rdf 1.1 concepts and abstract syntax.

Wang, T. D., Parsia, B., & Hendler, J. (2006). A Survey of the Web Ontology Landscape. In I. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, ... L. M. Aroyo (Eds.), *The semantic web - iswc 2006: 5th international semantic web conference, iswc 2006, athens, ga, usa, november 5-9, 2006. proceedings* (pp. 682–694). Berlin, Heidelberg: Springer Berlin Heidelberg

Wiebusch, D. (2016). *Reusability for Intelligent Realtime Interactive Systems*. BoD–Books on Demand.

Wiley, D. A. (2003). *Connecting Learning Objects to Instructional Design Theory: A Definition, a Metaphor, and a Taxonomy*.

Ehrenwörtliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht.

Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Würzburg, 18.05.2017

Eike Isermann