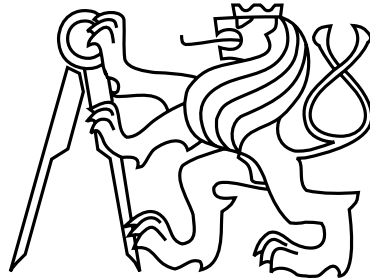


Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering



Diploma Thesis

**Integration of Heterogeneous Data Sources Based
on a Catalog of Master Entities**

Bc. Ondřej Pánek

Supervisor: Mgr. Martin Nečaský, Ph.D.

Study Program: Open Informatics

Field of Study: Software Engineering

May 7, 2015

Aknowledgements

I would like to thank Mgr. Martin Nečaský, Ph.D. for his willing help and devoted time when supervising this thesis. I extend my gratitude to Ing. Ondřej Straník for valuable consultations and helpful advice from practice. My deepest thanks goes to my girlfriend, family and friends for their support in my studies.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.

I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague on May 11, 2015

.....

Abstract

This diploma thesis addresses the problem of integrating heterogeneous data sources. Its aim is to design and implement a general and configurable integration system capable of providing simple and transparent access to data integrated from multiple heterogeneous data sources. The architecture is based on a modified wrapper-mediator principle extended by a catalog keeping the metadata and thus forming the integrated global schema. Since the architecture follows the RESTful principles, clients can query the integrated data via a standard, transparent, easy-to-use and self-descriptive API.

keywords: integration, heterogeneous data source, wrapper, mediator, catalog, transparent access, RESTful architecture

Abstrakt

Tato diplomová práce se zabývá problémem integrace heterogenních datových zdrojů. Jejím cílem je navrhnout a implementovat obecný a konfigurovatelný integrační systém schopný poskytnout jednoduchý a transparentní přístup k datům integrovaným z více heterogenních datových zdrojů. Architektura je založena na principu wrapper-mediator rozšířeném o katalog, který uchovává metadata a formuje tak integrované globální schéma. Jelikož architektura dodržuje principy RESTful, klienti systému se mohou dotazovat na integrovaná data prostřednictvím standardního, transparentního, snadno použitelného a samopopisujícího API.

klíčová slova: integrace, heterogenní datové zdroje, wrapper, mediator, katalog, transparentní přístup, RESTful architektura

Contents

1	Introduction	1
2	Background	3
2.1	Problem Statement	3
2.2	Interoperability	4
2.2.1	About interoperability	4
2.2.2	Interoperability Difficulties	4
2.2.2.1	Autonomy	5
2.2.2.2	Distribution	5
2.2.2.3	Heterogeneity	5
2.2.2.4	Instability	5
2.2.3	Heterogeneity	5
2.2.3.1	System level	6
2.2.3.2	Syntactic level	6
2.2.3.3	Structural level	6
2.2.3.4	Semantic level	7
2.2.4	Interoperability Approaches	7
2.2.4.1	Standardization	7
2.2.4.2	Federation	7
2.2.4.3	Mediation	8
2.2.4.4	SOA	9
2.2.4.5	Ontology and Semantic Web	9
2.3	Master Data Management	11
3	Related Works	13
3.1	Selected Integration Systems	13
3.1.1	TSIMMIS	13
3.1.2	Virtual Query System for SemanticLIFE	14
3.1.3	OWSCIS	16
3.1.4	Virtual-Q	17
3.2	Discussion	19
4	Analysis	23
4.1	Basic Vision	23
4.2	Key Features	24

4.3	Major Problems	24
4.3.1	Different Identifiers	24
4.3.2	Traffic Minimization	24
4.3.3	Identification of weak entities	25
4.4	Glossary	25
4.5	Principle and Business Process	26
4.6	Requirements	26
4.6.1	Functional Requirements	27
4.6.2	Non-Functional Requirements	28
4.7	Use Cases	30
4.7.1	Client's Perspective	30
4.7.2	Administrator's Perspective	30
4.8	Domain Model	33
5	Architecture	35
5.1	Architecture Overview	35
5.2	Components of the Architecture	38
5.2.1	Catalog	38
5.2.2	Wrapper Module	38
5.2.3	Mediator Module	39
5.2.4	Querying Web Service	40
5.2.5	Catalog Web Service	41
5.2.6	Admin Client	41
5.3	REST	41
5.3.1	Constraints and properties of REST	41
5.3.2	RESTful system architecture	42
6	Design	45
6.1	Catalog	45
6.1.1	Master Entity and Attribute Types	45
6.1.2	Owned Master Entity Types	46
6.2	Wrapper Module	46
6.2.1	Principles of Wrapper's Work	47
6.2.2	Building Wrappers	48
6.2.3	Dependent Keys	48
6.2.4	Cache	48
6.3	Mediator Module	49
6.3.1	Mediation Process	49
6.3.2	Waiting for Dependent Keys	51
6.3.3	Lazy Loading	52
6.3.4	Filtering	53
6.3.5	Stability	53
6.3.6	Usage Statistics	53
6.4	Querying Web Service	53
6.4.1	REST API Design	53
6.4.2	Response Building	55

6.4.3	Cache	55
6.4.4	Errors Handling	55
6.5	Catalog Web Service	55
6.5.1	Public API	55
6.5.2	Admin API	56
6.6	Admin Client	57
6.6.1	Dashboard	57
6.6.2	Data Sources Management	58
6.6.3	Wrappers Management	58
6.6.4	Master Entity Types Management	59
6.7	Database Design	59
7	Implementation	61
7.1	Used Platform and Technologies	61
7.1.1	Spring Framework	61
7.1.1.1	About Spring	61
7.1.1.2	Dependency Injection	61
7.1.1.3	Aspect-Oriented Programming	62
7.1.2	REST	62
7.1.3	ORM	63
7.2	Multithreading	64
7.3	Implementation of Wrappers	66
7.4	Cache	67
7.4.1	Wrapper Cache	67
7.4.2	HTTP Cache Control	68
7.5	Exceptions Handling and Logging	69
7.6	Admin Client	70
8	Testing	73
8.1	Unit and Integration Tests	73
8.2	Requirements Fulfillment	74
8.2.1	Functional Requirements	74
8.2.2	Non-Functional Requirements	74
8.3	Demonstration on Real Data	75
8.3.1	Connected Data Sources	76
8.3.2	Defined Master Entity Types	77
8.3.3	Implementation of Wrappers	77
8.3.4	Performance Tests	79
9	Conclusion	83
9.1	Future Work	84
	Bibliography	85
A	List of Abbreviations	89

B	Installation Manual	91
B.1	System Installation	91
B.1.1	Database Preparation	91
B.1.2	Deployment	91
B.2	Example on Real Data	92
B.2.1	ARES RDB	92
B.2.2	ARES REST	92
B.2.3	System Configuration	92
C	Administration Manual	93
C.1	Login	93
C.2	Dashboard	93
C.3	Entity Types	95
C.3.1	Overview	95
C.3.2	Add Master Entity Type	95
C.3.3	Edit Master Entity Type	97
C.3.4	Remove Master Entity Type	97
C.4	Data Sources	97
C.4.1	Overview	97
C.4.2	Add Data Source	97
C.4.3	Edit Data Source	99
C.4.4	Remove Data Source	99
C.4.5	Clear Cache	99
C.5	Wrappers	99
C.5.1	Overview	99
C.5.2	Add Wrapper	99
C.5.3	Wrapper Implementation	99
C.5.3.1	SQL	101
C.5.3.2	REST	101
C.5.3.3	CSV	101
C.5.3.4	SOAP	102
C.5.3.5	Other	102
C.5.3.6	Deployment	102
C.5.4	Edit Wrapper	102
C.5.5	Remove Wrapper	102
C.5.6	Clear Cache	102
D	User Manual	103
D.1	API Entry Point	103
D.2	Catalog Web Service	103
D.3	Querying Web Service	104
E	DVD Content	107

List of Figures

3.1	TSIMMIS architecture based on wrappers and mediators (taken from [11]) . . .	14
3.2	The architecture of the Virtual Query System (taken from [21])	15
3.3	OWSCIS architecture (taken from [13])	17
3.4	Virtual-Q - overview of the main parts (taken from [47])	18
3.5	The architecture of the Virtual Query Engine (taken from [47])	18
4.1	Process of data retrieval and integration	26
4.2	Basic schema of the system	27
4.3	Use cases from the client's perspective	31
4.4	Use cases associated to the catalog management	31
4.5	Use cases from associated to the data sources management	32
4.6	Use cases from associated to the wrappers management	32
4.7	A composite use case to integrate a new data source with all the circumstances	33
4.8	Analytical domain model	34
5.1	Basic overview of the architectural components	36
5.2	Deployment schema of the integration system	37
5.3	Catalog	38
5.4	Wrapper Module	39
5.5	Mediator Module	40
5.6	Querying Web Service	41
6.1	Design classes of the Catalog	46
6.2	Design classes of the Wrapper Module	47
6.3	Design classes for caching of wrappers' results	49
6.4	Design classes of the Mediator Module involved in mediation by key	50
6.5	Sequence diagram of mediation by key	51
6.6	Design classes specific for mediation all	52
6.7	Design classes of the Querying Web Service	54
6.8	Design classes of the public part of the Catalog Web Service	56
6.9	Design classes of wrappers administration	57
6.10	Database design	60
7.1	Page for master entity types management	70
7.2	Dialog for editing a master entity type	72

8.1	Data sources in the example of use	76
8.2	Master entity types in the example of use	78
8.3	Performance tests results - lazy loading off, cache off	79
8.4	Performance tests results - lazy loading on, cache off	80
8.5	Performance tests results - lazy loading on, cache on	80
8.6	Performance tests results - average times of all situations	81
C.1	Login page	93
C.2	Dashboard	94
C.3	Error log	94
C.4	Master entity types management	95
C.5	Add a master entity type dialog	96
C.6	Specification of attribute types	96
C.7	Data sources management	98
C.8	Add a data source dialog	98
C.9	Wrappers management	100
C.10	Add a wrapper dialog	100

List of Tables

3.1	Comparison of selected integration systems	20
8.1	Results of performance tests	81

Chapter 1

Introduction

This diploma thesis addresses the problem of integrating heterogeneous data sources. Organizations of any size need to effectively work with data to support business requirements and nowadays, enormous amounts of data are available over public or corporate networks. However, the rapid growth of the Internet as well as information technology in general led to a coexistence of different ways of data storing and providing, and that brought certain restrictions on working with multiple different data sources.

The aim of this thesis is to design and implement a general and configurable integration system capable of providing simple and transparent access to data integrated from multiple heterogeneous data sources. Among other essential requirements belong supporting of a wide range of data sources, simple extensibility with new ones and orientation on essential business entities, internally called master entities. An important condition is to be able to work with original data without the need to preprocess them and store in a more suitable storage.

Several related works are introduced but since no existing system supports all the desired features, a different approach is proposed. Nevertheless, the integration system uses the best practises from the existing solutions as a modified wrapper-mediator principle and introduces a catalog component to store metadata forming the integrated global schema. As a result, a standard RESTful API is formed on top of the system offering both transparently integrated data and metadata from the catalog enabling to better understand the data and build dynamic client applications above them.

This text is organized as follows. Chapter 2 gives background information - states the problem and explains the issue of interoperability and master data management. In Chapter 3, several selected related works are introduced and compared. Next, Chapter 4 begins the development of the integration system with the analysis. In Chapter 5, the architecture of the integration system is proposed. Chapter 6 addresses the software design phase, Chapter 7 deals with the implementation part and Chapter 8 describes system testing. Finally, Chapter 9 concludes the diploma thesis including some future work remarks.

Chapter 2

Background

This chapter provides background information about the thesis' topic. Firstly, the problem statement is proposed in general. Then, the main part of the chapter is dedicated to the problem of interoperability, describes its difficulties with emphasis on heterogeneity and introduces various approaches to achieve it. At the end, a discipline called Master Data Management is briefly discussed.

2.1 Problem Statement

Nowadays, enormous amounts of data are available over public or corporate networks and a plenty of new on-line data sources are continuously appearing. People often access more or less structured data, which tend to be very dynamic. However, the rapid growth of the Internet as well as information technology in general led to a coexistence of different ways of data storing and providing, and that brought certain restrictions on working with multiple different data sources. Heterogeneity of data sources is an obstacle in the access to information, its processing and combination.

Organizations of any size need to effectively work with data to support their business requirements. Even at a single organization level, information from several different data sources is necessary for making business decisions. That usually is both time consuming and demanding of knowledge about the locations of required data and ways of retrieving them. Acquisition of information from heterogeneous data sources involves working with various data models and requires the ability to use different query methods and languages. In addition, in order to correctly and accurately interpret information from diverse data sources, one has to be cautious of its intended meaning.

In such an environment it is still more and more important to develop software systems, applications and tools that manage to combine relevant data from heterogeneous data sources and present them to the user in an understandable manner. Sometimes it is not possible to preprocess the data and move them into a more suitable storage in order to support their better linking and enrichment. Therefore, we often need applications based on original data integration in practice.

Data integration refers to the combination of data stored by various technologies in multiple disparate sources and providing users with a unified view of these data. As a result, the

unified view offers better accessible and more meaningful and valuable information. Combining technical and business processes, data integration overcomes the barriers in the form of system, syntactic, structural and semantic heterogeneity. One of the key requirements on a good integration solution consists in a fact that the end user or data consumer should be as little involved in the technical integration process as possible without having to be aware of the real data origin.

2.2 Interoperability

2.2.1 About interoperability

Data integration offers a possibility to transparently manipulate with information across multiple data sources. The term of data integration is a part of a wider concept of interoperability between systems.

According to [12], interoperability is the ability of two or more systems or components to exchange information and use the information that has been exchanged. Because of an increasing amount of independently developed information systems, more significant specialization in work demanding data reuse and analysis and a great diversity of paradigms in data representation and modelling, the interoperability is becoming a center of interest.

[35] introduces four degrees of interoperability which are intended to classify how structuring and automating the exchange and interpretation of data can enhance operational effectiveness.

- **Unstructured Data Exchange** – involves the exchange of human-interpretable unstructured data such as the free text extracted from operational estimates, analysis and papers
- **Structured Data Exchange** – involves the exchange of human-interpretable structured data intended for manual and/or automated handling, but requires manual compilation, receipt and/or message dispatch
- **Seamless Sharing of Data** – involves the automated sharing of data among systems based on a common exchange model
- **Seamless Sharing of Information** – an extension of the previous degree to the universal interpretation of information through data processing based on co-operating applications

2.2.2 Interoperability Difficulties

Interoperability remains a complicated problem and still creates new challenges for many areas of computer science. [45] presents main difficulties of interoperability - autonomy, distribution and heterogeneity. [13] emphasises the significance of a fourth aspect - instability.

2.2.2.1 Autonomy

Data sources are usually autonomous - managed separately and independently. If the information system bases on multiple sites - components, the autonomy becomes a major problem. The information system anatomy can be classified to three areas [4]:

- **Design anatomy** – includes the property of a component to have an independent design suitable for inner data representation, management and integrity constraints changeable at any point in time, which can lead to complication within the cooperating infrastructure
- **Communication autonomy** – means that a component can decide which other systems it communicates with
- **Processing autonomy** – ensures the independence of a component in the matters of scheduling and processing of incoming requests

2.2.2.2 Distribution

Physical location of data may become another obstacle in interoperability. Data could be distributed among multiple data sources, which could be spread over one or more computer systems. The request then cannot be answered by the data available from a single data source, but it has to merge the distributed answer fragments. In addition, the distribution methods may vary considerably (e.g. vertical vs. horizontal distribution in relation databases).

2.2.2.3 Heterogeneity

The major difficulty of data integration is the heterogeneity of information sources. The differences consist in areas from hardware platforms over various data models, way of data representation, query languages to the interpretation of information meaning. A more detailed description of heterogeneity is given in the next section.

2.2.2.4 Instability

The information sources appear every day, others change or disappear. Any minor change in data source schema, like adding or removing an attribute, influences the integration of these sources.

2.2.3 Heterogeneity

Heterogeneity arises in situations when there is a conflict in meaning, interpretation or intended usage of the same or related data [6]. Bridging the heterogeneity belongs to the most important tasks of data integration. Based on a research of various aspects of heterogeneity, [44] distinguishes four levels of heterogeneity - system, syntactic, structural and semantic.

2.2.3.1 System level

System level of heterogeneity involves technical differences on the lowest level of abstraction - hardware, operational system and communication systems and protocols. [44] divides system heterogeneity into two subareas:

- **Information system heterogeneity** – includes the heterogeneity of database management systems, data models, system capabilities such as concurrency control and recovery
- **Platform heterogeneity** – includes operation system related aspects (heterogeneity of file system, naming, file types, operation, transaction support), and hardware related aspects (heterogeneity of instruction set, data representation/coding)

2.2.3.2 Syntactic level

This level comprises of distinction in data models, languages and representations. Syntactic differences seen from the view of heterogeneity include:

- **Data model heterogeneity** – information can be modeled according to various paradigms (relational database, XML database, graph database, object database etc.), which brings differences in the semantics of used concepts
- **Interface heterogeneity** – this type of heterogeneity consists of different approaches to retrieve data including diverse query languages with their constraints and query restrictions (e.g. only certain queries are allowed, only certain conditions can be expressed, optional vs. required input parameters)
- **Format heterogeneity** – refers to different data types (e.g. integer vs. long) and data formats (e.g. 2014-12-24 vs. 24-12-2014)

2.2.3.3 Structural level

This level lies in a different data modelling. It comprises of *representation heterogeneity* involving data modelling constructs and *schematic heterogeneity*, which refers especially to structured databases.

- **Representation heterogeneity** – this heterogeneity is caused by inconsistencies in information modelling and representation in different schemas (e.g. the same information can be represented by one or more attributes)
- **Schematic heterogeneity** – is concerned with using of different elements of a data model to describe the information (relation vs. attribute name, attribute name vs. attribute value, relation vs. attribute value)

2.2.3.4 Semantic level

The meaning of data can be expressed in various ways, and that brings semantic heterogeneity. This level is focused on the correct interpretation of information, when the key aspect is to know its intended meaning. The goal is to properly evaluate which concepts are equivalent or related. [15] identifies three major reasons of semantic heterogeneity:

- **Confounding conflicts** – occurs when two pieces of information seem to have the same meaning but vary in reality, e.g. due to different temporal contexts
- **Scaling and units conflicts** – lies in using different units of measures or scales (different currencies, different granularity in scales etc.)
- **Naming conflicts** – consists of significant differences caused by naming schemes, a conflict usually occurs when using synonyms (different terms used to refer to the same concept) or homonyms (the same term used to refer to different concepts)

2.2.4 Interoperability Approaches

Various approaches to the interoperability have been introduced during the technological development.

2.2.4.1 Standardization

Standardization approach says that each cooperating system communicates using the same standard and has the same model for data representation merging all system domains. Design of such a complex data model is not only a very difficult task but also poorly resistant to changes. Any change, even of only one data source, reflected in the integrated model can significantly affect many others which have to be adopted subsequently.

2.2.4.2 Federation

This approach is based on a cooperation of independent heterogeneous databases which are connected by one or more virtual federated schemas [33]. A federated schema associates exported schemas of one or more data sources. Generally, two types of federations are distinguished - tightly coupled and loosely coupled [39].

In a **tightly coupled** federation, each source defines an optional subset of its schemas to be shared in a federation. Based on these schemas, the federation administrator then has to put them together into a global federated schema, which acts as an entry point for a transparent access to data without any need for knowledge of its real location.

On the other hand, a **loosely coupled** federation does not perform any global schema integration but incorporates a subset of schemas available in the federation for specific purposes. These federation schemas are created by users or administrators of local databases involved in the federation. User has to specify schemas of the used databases in order to access the data, which removes any levels of location transparency.

[45] extended the classical three-level database schema to a five-level architecture in order to support autonomy, distribution and heterogeneity of federated database systems. The five-level schema architecture consists of following layers:

- **Local schema** – a local conceptual schema on a database (federation component) level expressed by a native data model
- **Component schema** – derived from translating local schemas to a canonical or common data model, which allows to describe the heterogeneity of local schemas by a uniform representation and to refine them with some additional semantics (plays the integrating role in a tightly coupled federation)
- **Export schema** – defines a subset of component schemas intended to share in a federation, hence accessible to non-local users (may include access control mechanisms using filters)
- **Federated schema** – integrates multiple export schemas and keeps the information about distribution generated during the integration process, which is used to a translation of federated schema to export schemas
- **External schema** – determines a schema for end users/applications, can simplify the federated schema for some specific purposes including new integrity constraints or access control rules specification

2.2.4.3 Mediation

The mediation extends the federation approach for the integration of a large number of information sources that can be dynamically added, removed or often modified. The aim of the mediation approach is to benefit from the transparent access of a tightly coupled federation and the flexibility of a loosely coupled federation in order to provide a solution in data integration.

The mediation architecture is based on two types of software components - wrappers, designed for communication with data sources, and mediators, which are responsible for the integration and interfacing with users or applications.

Wrapper operates on top of a data source and maps its native representation into a common data model. In order to achieve such a mapping, wrapper must be able to convert a request of the common data model in a request that is specific for the particular source and the obtained data translate back to the common data model [10].

Mediator forms an integrated view on data from one or more wrappers. Its task is to process an incoming request, delegate it to appropriate wrappers and combine, integrate and abstract the obtained data. The mediators can cooperate with each other, for example to decompose a query into subqueries and schedule their execution plan. The rules in mediators determine how to map each data source into an integrated view.

As regards the specific types of heterogeneity, wrappers overcome the syntactic heterogeneity by hiding the inner data representation and providing the data independently of the nature of their source. Mediators bridge the gap in heterogeneity on the structural

level, since they map the data from wrappers into a homogeneous data structure. Semantic heterogeneity belongs also to a responsibility of mediators, so it must be taken into account [52]. However, heterogeneity on the semantic level makes the development of mediators a time consuming and complicated task.

2.2.4.4 SOA

SOA (Service Oriented Architecture) represents a flexible set of design principles used during the phase of system development and integration. It determines a way of integration of various applications and systems in a distributed world using multiple different platforms.

A service is generally implemented as a coarse-grained, discoverable software entity that exists as a single instance and interacts with applications and other services through a loosely coupled (often asynchronous), message-based communication model [3]. The services and their consumers communicate with each other by passing data in a shared format or by orchestrating an activity between multiple services to create advanced functionality.

Basic protocols enabling discoverability and describing the methods of using the web services are SOAP (Simple Object Access Protocol) [46], WSDL (Web Services Description Language) [55] or UDDI (Universal Description Discovery and Integration) [51]. The communication is commonly based on HTTP (Hypertext Transfer Protocol) [23] and the data is usually transferred in standard formats like XML (Extensible Markup Language) [56], JSON (JavaScript Object Notation) [28] or YAML (YAML Ain't Markup Language) [2]. In recent years, an architectural style REST (Representational State Transfer) [9], which is based on a set of stateless web services designated to manipulate the resources through their representation, has been coming to the fore.

A set of web services can be independently built requiring more or less effort from any information source. These services then forms a system interface providing an access to the data and functionality. It is a very flexible solution leading to an interoperability of heterogeneous information sources.

2.2.4.5 Ontology and Semantic Web

According to [53], the Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries. It is a collaborative effort led by W3C with participation from a large number of researchers and industrial partners. It is based on the RDF (Resource Description Framework) [40]. The Semantic Web refers to a group of technologies that seek to build a global data space of semantically linked information, which would be well readable and processable by machines.

The above mentioned approaches to the interoperability focused especially on the syntactic and structural level of heterogeneity, while the semantic heterogeneity was slightly in the background. Overcoming the semantic heterogeneity, which would result in a correct interpretation of the meaning of information across each system, is considered to be the most difficult problem on the way to data integration. Tools of the Semantic Web, ontologies in particular, concentrate on this vital aspect of heterogeneity.

The term ontology stands for an *explicit specification of a conceptualization* [16]. *Conceptualization* refers to a set of objects, human commonly think they exist in the world of

interest. It represents an abstract model of how we think about the real things. *Explicit specification* denotes a hierarchically organized structure of explicitly named and defined concepts and relations between them. Such a structure defines the meaning of objects of the abstract model.

Generally, an ontology composes of a set of organized terms to describe the reality on a concept base. A dictionary provided by the ontology serves as a stable conceptual model to the information sources independent on the source schemas. The language of ontologies is expressive enough to address even a very complex knowledge. To define ontologies, languages as OWL (Web Ontology Language) [36] or RDFS (Resource Description Framework Schema) [41] are most often used.

The strong expressiveness allows to define a concept in a precise manner, which is processable and interpretable by computer systems. The interpretation of knowledge described in ontologies enables to translate the information sources into the common data model. All these features make the ontology an appropriate technology for data integration.

In the vast majority of integration approaches based on ontologies, an ontology is used for the explicit description of information sources semantics. But there are different ways of how to employ an ontology. [54] distinguishes three different directions: single, multiple and hybrid.

Single ontology approach

This approach uses one global ontology to provide a shared vocabulary for a description of knowledge. The global ontology may also be a combination of multiple specialized ontologies due to a better maintainability.

All information sources are mapped to one global ontology. An independent model of each source must be described by relating the objects of the source to the concepts of the global ontology. Based on these mappings, it is possible to identify semantic relations between information from different sources.

Single ontology approach is suitable in situations when all information sources provide nearly the same view on its domain. If a view of some information source differs from others, i.e. has another level of granularity, finding a minimal ontology becomes a problem. [17].

Multiple ontology approach

In a multiple ontology approach, each information source is described by its own ontology. This ontology could again be a combination of several ontologies, but the essence is that information sources share neither a dictionary nor a schema.

The advantage of this approach consists in no need for a common minimal ontology, which is often hard to assemble due to the heterogeneity in data granularity. On the other hand, the lack of a global dictionary makes it exceptionally difficult to compare different source ontologies. This shortage is solved by creating inter-ontology mappings which identify relations between the concepts of the compared ontologies. However, these mappings must take into account the different views of data, and that makes their definition very complex.

Hybrid approach

To overcome the drawbacks of both above mentioned approaches, a hybrid approach was introduced. Information sources are described by their own ontologies like in the multiple ontology approach, but in order to be comparable, these local ontologies are built upon one global dictionary. To describe a domain, this shared dictionary contains primitive terms, which can be combined to cover more complex knowledge on the local ontology level. This approach requires two kinds of mappings: 1) mappings between information sources and their local ontologies and 2) mappings between local ontologies and the global dictionary.

The advantage of the hybrid approach lies in an easy adding of new sources without the need to modify neither the shared dictionary nor the mappings. The presence of the global dictionary allows to compare local ontologies eliminating the disadvantage of multiple ontology approach. But the drawback of the hybrid approach is that existing ontologies cannot be easily reused because of the dependencies of local ontologies on the shared dictionary, so the local ontologies have to be developed from scratch.

2.3 Master Data Management

Master Data Management (MDM) is a collection of best data management practises that intend to support organization's business needs by providing access to a single view of the master data entities across the operational infrastructure [31]. [43] describes MDM as a business-driven practice of defining and maintaining consistent definitions of business entities and data about them across multiple IT systems and possibly beyond the enterprise. Master Data Management gained its name by the data it governs - master data.

Master data represents the most essential core business entities shared across multiple transactional applications. It involves the key objects that matter the most, which are the subjects of measuring, analysis and reporting processes. Master data forms a single version of the truth about these objects across the operational landscape [5]. Common examples of master data include the following:

- Customers
- Products
- Employees
- Suppliers
- Locations
- Contracts etc.

MDM has reached the stage that it is a mainstream activity today, because organizations increasingly need, among other things, a single view of their essential data to support their business decisions. The master data may often be enriched by information even beyond the organization level, which subsequently brings a need for the integration.

Chapter 3

Related Works

This chapter addresses several selected integration systems, including: TSIMMIS [10, 11], the Virtual Query System for SemanticLIFE [20, 21, 22], OWSCIS[14, 13] and Virtual-Q [48, 47, 34]. At the end, properties of the introduced systems are discussed resulting in a set of best practises to use within this thesis.

3.1 Selected Integration Systems

Firstly, several selected integration systems are described including the used approach, the architecture, a way of representing data model, a way of querying etc. All these systems work with original data querying original data sources every time a user makes a request (excluding caching).

There are numerous ways of integration that need to store the data in a different storage more suitable for purposes of integration, linking and enriching. These approaches generally provide more advanced integration possibilities and more complex operations on the data. On the other hand, they need to host and maintain an infrastructure for storing the data and manage the updates of original data in order to provide fresh and up-to-date information. Such approaches include e.g. data warehousing on an organization level or Linked Data [18] on a global level. However, this thesis is not concerned with such a situation.

3.1.1 TSIMMIS

TSIMMIS - The Stanford-IBM Manager of Multiple Information Sources - is a project whose main goal is to develop tools that facilitate the rapid integration of heterogeneous information sources including both structured and unstructured data [10].

This system is a typical example of the mediation approach (see Figure 3.1). Wrappers (or translators) stand above each data source converting application queries into source specific queries and translating obtained data into a common data model. Mediators are used for the integration part refining in some way information from one or more sources. To properly describe the common data model, TSIMMIS introduced its own simple and self-describing object model called OEM (Object Exchange Model) [37] and an SQL-like query language for requesting OEM objects named Lorel Query Language [1].

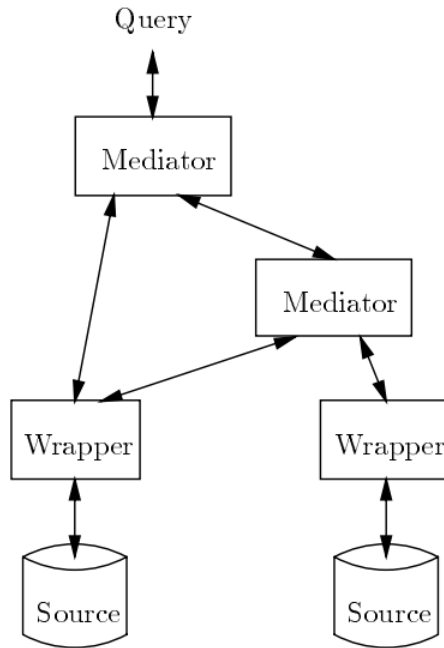


Figure 3.1: TSIMMIS architecture based on wrappers and mediators (taken from [11])

Both mediators and wrappers export the same interface taking a query as an input and returning OEM objects. Such a unified approach allows to access the data sources transparently either directly from the wrappers or the mediators. Hence, a new data source becomes useful as soon as a wrapper is supplied.

In TSIMMIS there is not a single global schema. Furthermore, mediators and wrappers are not required to provide objects according to a fixed schema or type, but each query determines the form of the matching objects' schema. No end user knows the global view of all the information handled by the system. Therefore, in order to build a new mediator or wrapper, it is necessary only to understand the data source that it will use.

TSIMMIS concentrates especially on flexibility, so it is well prepared for unexpected occurrences of heterogeneity. Accessing very diverse and different information which may frequently change its content or meaning is the basis of this system. The downside of such an approach is that in some cases, the integration must be performed manually by the end user. So, as the authors of this project say, TSIMMIS does not perform fully automated integration but rather provides a framework and tools to assist the users in information processing and integration efforts.

3.1.2 Virtual Query System for SemanticLIFE

SemanticLIFE is a framework for managing information of a human lifetime from the Semantic Web [20]. Its query system, the Virtual Query System [21] (VQS), focuses on retrieving information from huge ontology-based repositories in a very efficient, yet user-friendly way.

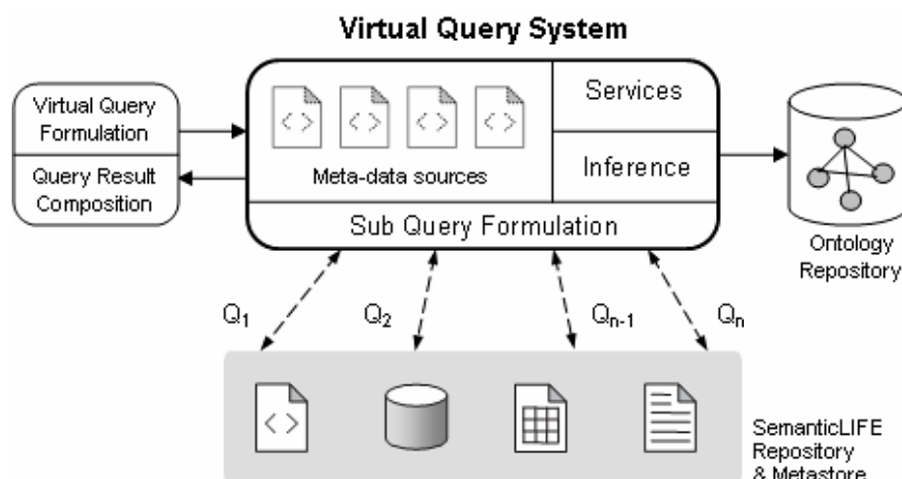


Figure 3.2: The architecture of the Virtual Query System (taken from [21])

The motivation of this project is based on an easement of querying semantic web applications without knowledge of RDF query languages. For this purpose, the Virtual Query Language [22] (VQL) was also developed.

The Virtual Query System uses the hybrid ontology approach simplified by the fact that the local ontologies are already provided by the ontology-based information sources, hence not required to be redeveloped from scratch. The global ontology offers a better understanding of the available data to users in order to enable formulating of non-ambiguous queries. Based on the mappings between the common global ontology and local ontologies, the VQS refines the queries and decomposes them to sub-queries for individual data sources.

The architecture of the VQS composes of six modules (see Figure 3.2):

1. **Meta-data sources:** Contains the metadata of data sources and acts as a virtual data source enabling the user to explore the semantics of the available data and form more specific queries.
2. **The Ontology Repository:** Comprises the global ontology, its mappings from the local ontologies and inference ontology.
3. **Sub-Queries Formulation:** Based on the ontologies mappings, this module transforms a virtual query to sub-queries for the particular data sources in a form of a specific RDF query language.
4. **The VQS Services:** Involves services dealing with generating mappings from local ontologies of newly added information sources to the global ontology, caching of queries and results and an interactive process of query refinement.
5. **Ontology-based Inference:** Provides a basis for analysis and deduction on the concepts of specified ontologies. This module helps the system with generating sub-queries based on the inference ontology.

6. **The Virtual Query User Interface:** A graphical user interface provides an overview of the system schema and helps the user to formulate queries.

The Virtual Query Language is used to model the query patterns and generate the virtual queries. The queries are coded in XML and specify schemas, sources, constraints and specific data. After a user formulates a virtual query with a help of the client-side tools, the VQS evaluates it on the foundation of the ontology-based services. The analyzed virtual query is sent to the Sub-queries formulation module, where the specific queries for each real data source are generated. Finally, the results are integrated and returned to the user.

3.1.3 OWSCIS

OWSCIS (Ontology and Web Service based Cooperation of Information Sources) is an ontology-based cooperation system between heterogeneous information sources [14]. It aims at transparent querying on the information sources in an integrated centralized way.

This system is a prime example of the hybrid ontology approach based on the mediation architecture. Each data source is wrapped by a local ontology that is mapped to the source schema in order to be linked to the actual information. That allows to preserve the source's autonomy, transparency and extensibility. Beside local ontologies, there is a global ontology that provides the mediation schema describing the semantics of the whole domain of interest. The global ontology represents a global model for all participating local ontologies and acts as a mediator. Therefore, another type of mappings between the global and local ontologies is required.

OWSCIS architecture is composed of the following modules and web services, each focused on a specific activity (see Figure 3.3) :

1. **Knowledge Base Module:** Encapsulates the global ontology and mappings with the local ontologies.
2. **Data Provider:** A wrapper above a data source described by a local ontology including the mappings to the source schema.
3. **Mapping Web Service:** Establishes mappings between local ontologies and the global ontology.
4. **Querying Web Service:** Serves to decompose queries into sub-queries and rebuild the results.
5. **Visualization Web Service:** Visualizes the global ontology and presents the results to the user.

A user specifies a query in SPARQL (SPARQL Protocol and RDF Query Language) [49], a query language for RDF, according to the global ontology. Based on the mappings between the local ontologies and the global one, OWSCIS decomposes the query and rewrites it to several SPARQL queries using the terms from the local ontologies. On the data provider level, the SPARQL query is translated to a specific query language of the underlying data source according to the mappings between the local ontology and the source schema. The

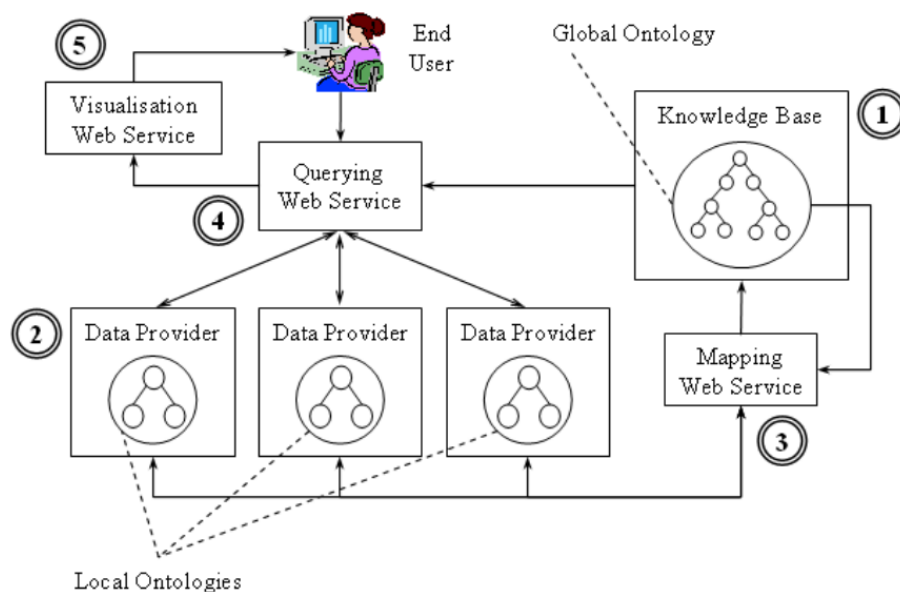


Figure 3.3: OWSCIS architecture (taken from [13])

results are then translated to RDF and returned to the integration processing. Finally, the combined RDF data is presented to the user.

OWSCIS supports, with certain limitations, SPARQL-to-SQL [50] query translation and SPARQL-to-XQuery [57] query translation, so it is able to work with relational databases and XML data sources.

3.1.4 Virtual-Q

Virtual-Q is a virtual query system whose creation is related to CALIMERA (Conference Advanced Level Information Management & Retrieval) framework [48] aiming at the enhancement of the information management, retrieval and visualization of recorded talks of scientific conferences.

This system is not a representative of any interoperability approaches described in the previous section per se, but studying its architecture we could notice some parallels with the mediation. Also, by the manner of working with data sources' schemas, this engine could be likened to the multiple ontology approach, although the local schemas do not have to always be ontologies. What is clear is that Virtual-Q does not have an established global schema.

The model of Virtual-Q system (see Figure 3.4) consists of four main parts: 1) The user interface, 2) the Virtual Query Engine (VQE), 3) the data and metadata storage and 4) the external reasoning models. VQE (see Figure 3.5), the essential part of the system, involves the following modules [47]:

1. **Query Receiver:** Receives the original user query and transforms it in the format used inside the Virtual Query Engine.

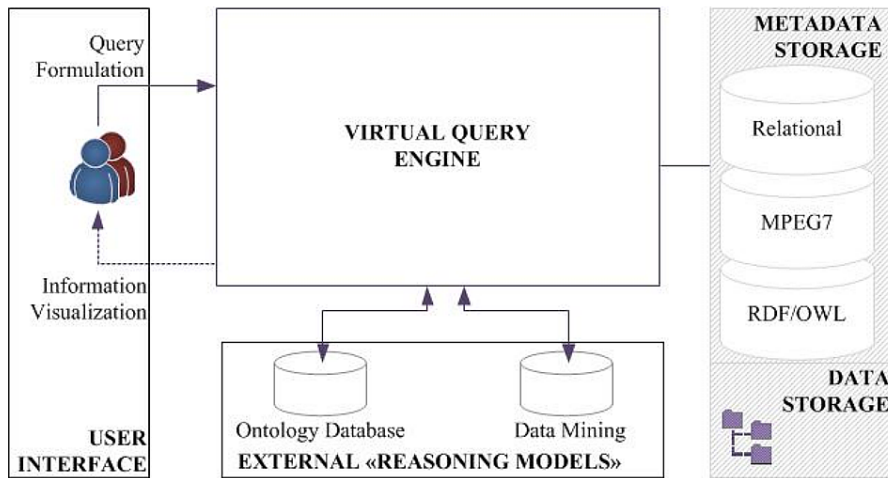


Figure 3.4: Virtual-Q - overview of the main parts (taken from [47])

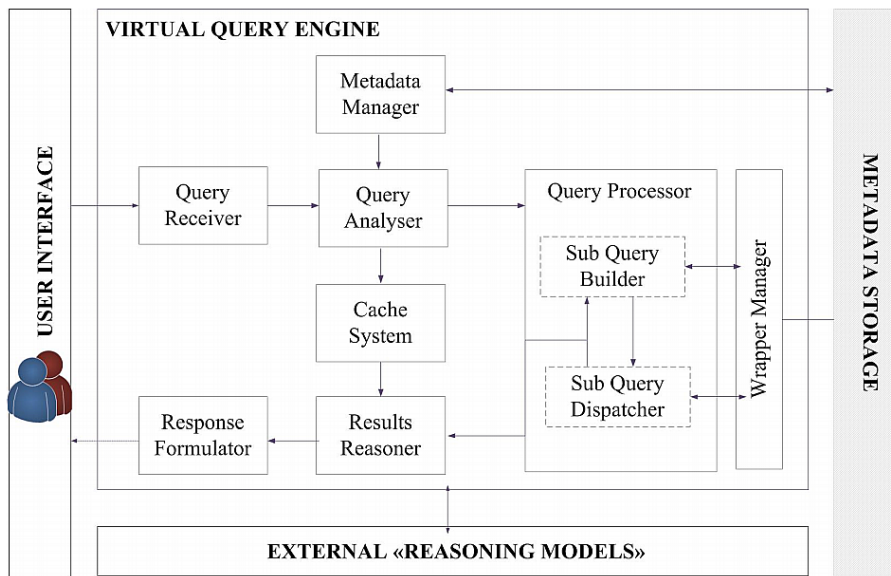


Figure 3.5: The architecture of the Virtual Query Engine (taken from [47])

2. **Query Analyzer:** Analyses the user query in order to retrieve the most precise information. Cooperating with the Metadata Manager and the Wrapper Manager modules, the Query Analyzer proceeds with accordance to this approach:
 - *Data sources schemes:* Each data is compliant to a schema, even the most basic one. The idea is to request the schemas of data sources, which is in most cases possible.
 - *Ranking the schemas:* Schemas could be described by different ways, some are stronger than others. So, the second idea is to rank the schemas for each theme in order to let the virtual query engine know the best schemas.
 - *Using a schema:* Since the analyzer knows the best schema for a given theme, it can use it to analyze a query and refine the creation of sub-queries. This could be done by e.g. exploiting the field names and using associate metadata.
3. **Query Processor:** Composed of the Sub Query Builder and the Sub Query Dispatcher modules. The Sub Query Builder generates a first query to be run on the most adequate data source and transfers this query called virtual query to the Sub Query Dispatcher. The Sub Query Dispatcher sends the virtual query to the corresponding wrappers, merges the results and passes them to the Results Reasoner module.
4. **Results Reasoner:** Reasons on the results with an assistance from external reasoning models.
5. **Response Formulator:** Converts the queries results into proper format and sends them to the user.
6. **Metadata Manager:** Manages the metadata, connected data sources, installed wrappers, schema rankings etc.
7. **Wrapper Manager:** Forwards queries to the appropriate wrappers.

The major aim is to provide an easy-to-use system by avoiding complicated administration work and automating as many processes as possible. It allows to simply add data sources at any time without a need to describe in advance their structure by the administrator. The method of querying is the easiest possible from the view of a user - a text field, which can be complemented by a theme/topic specification to improve the results [34]. The query is then wrapped in XML.

3.2 Discussion

The purpose of this diploma thesis is not to design an integration system that is better in all possible aspects than the existing ones but to propose a solution for querying the essential business entities of a specific domain from multiple heterogeneous data sources. This might also be interpreted as one of the disciplines of Master Data Management where a part of the information about organization's master data comes from data sources beyond the organization.

Each integration system introduced in the previous section has its pros and cons. The author of this text attempted to compare these approaches by several aspects that are important for the desired objective of this thesis. Reflecting the presented systems on these properties, Table 3.1 has been obtained. The comparing aspects include:

Table 3.1: Comparison of selected integration systems

	TSIMMIS	Virtual Query System for SemanticLIFE	OWSCIS	Virtual-Q
Approach	Mediation	Hybrid Ontology	Hybrid Ontology	Multiple Ontology (not exactly)
Data Model	OEM	XML, RDF, OWL	RDF, OWL	various
Global Schema	no	yes	yes	no
Query Language	LoREL	XML	SPARQL	XML
Degree of Automation	semi	manual	full	full
Quality of Answers	medium	high	high	low
Support of Data Sources	high	very low	low	medium

- **Approach** – says which of the interoperability approaches (presented in Section 2.2.4) the system is based on
- **Data Model** – determines the way of representing the common data model and/or data sources' schemas
- **Global Schema** – indicates whether the system keeps a global schema of its data model
- **Query Language** – identifies the method of querying the integration system
- **Degree of Automation** – measures the level of automation, the extent to which the user is needed to be involved in the process of integration
- **Quality of Answers** – estimates the level of accuracy, how the results correspond to what the user really asked for and how a user/application using the system can rely on the form and content of the answers
- **Support of Data Sources** – specifies how disparate sources could be appended to the integration system, especially from the perspective of the level of structure (struc-

ured, semi-structured, unstructured data) and methods of querying (SQL, XQuery, SPARQL, a web service call, preprocessing of unstructured data etc.)

According to the results of the comparison, no presented integration system can be considered as a versatile solution. Each of them has at least one handicap limiting its usage in the desired situation. We can notice some patterns proving that by fulfilling a specific requirement we might inevitably lose other advantages.

For example, looking at the aspects evaluation for OWSCIS, this system provides fully automated and high quality answers with a lot of querying possibilities thanks to SPARQL. But its bottleneck lies in a very limited range of supported data sources because of significant limitations in SPARQL translation. This means that requesting the support of a wide variety of data sources must be balanced with lesser possibilities in querying.

In addition, the comparison shows that in order to provide a high quality answer, the system should have a global schema. Having a global schema, the user can look at the global data model and submit a clear and precise query.

The pros and cons of OWSCIS has already been discussed. The Virtual Query System for SemanticLIFE works exclusively with ontology-based data sources and Virtual-Q is rather a full-text searching mechanism across various data sources returning a wide range of information often with a poor precision. The only referenced system supporting a large variety of data sources is the mediation-based TSIMMIS. Unfortunately, it lacks of standard technologies for querying and representing the data model and does not guarantee the fully automated information integration with high quality results.

To sum up, inspiring by the related works, the integration system developed within this diploma thesis should be built on the following principles:

1. provide a global schema in order to achieve full transparency and high quality answers
2. use the mediation-based approach
3. use standard technologies for querying and a description of the data model
4. restrict the possibilities of querying enough to enable support of wide variety of data sources

Chapter 4

Analysis

This chapter is dedicated to the first phase of the integration system development - the analysis. Firstly, a basic vision including the motivation and the system's purpose is given. Then, there are proposed key features of the system and major problems expected to occur during its development. After explaining specific terminology, the essential principles and processes of the system are introduced. An important part of the analysis is a list of functional and non-functional requirements followed by a specification of use cases and introducing a basic domain model.

4.1 Basic Vision

Today, a huge amount of data is available throughout the public or corporate networks. Organizations more and more need to have a unified view of their essential data in order to support their business. However, various types of heterogeneity of information sources build an obstacle in both data access and evaluation. The integration system should resolve this problem and transparently provide more valuable and better accessible information.

The purpose is to develop an integration system capable of merging data from remote heterogeneous data sources. The target solution should provide the highest possible level of automation and quality of answer and allow to work with nearly arbitrary data sources. From the user's point of view, the communication with particular information sources is fully transparent. The system should offer an easy-to-use standard way of querying to enable a flexible client application to be built upon.

As discussed in Section 3.2, no integration system is universal and well-suited for every purpose. Apparently, requiring some features we might lose some other possibilities. It is important to note that the system must be able to work with the original data without the need to preprocess them and store in a more suitable form or storage. Furthermore, the system is restricted with the requirements for support of various data sources including their fundamental or other limitations.

Among the specifics of this integration system belong orientation on the essential business entities (master data) and advanced configuration to overcome various access restrictions (regarding especially web services) in order to enable feasible behaviour and provide reasonable performance.

4.2 Key Features

The crucial features forming the system characteristics are:

- Support of a wide variety of data sources
- Simple extensibility with new data sources (no need to modify existing source code)
- Transparent integration approach
- Standard and easy-to-use way of querying
- Orientation on essential business entities (master entities)
- Advanced configuration regulating access to data sources (e.g. due to access limitations or high latency of web services)

4.3 Major Problems

This section presents the major problems expected to occur during development of the integration system that should be solved within this thesis.

4.3.1 Different Identifiers

One of the non-trivial obstacles the integration system ought to overcome are different identifiers. Nature of this problem lies in the fact that various data sources might identify the same entity by different keys, e.g. one data source identifies a user by his/her e-mail address while another data source identifies him/her by a personal identification number.

When such a situation occurs, it is important to choose the most suitable key as the main key within the integration system. In order to deal with the inconsistency, the system must know how each information source identifies the particular entity and keep relations between the keys. These relations may basically be of the two following types:

- **Dependency on the main key** – The primitive relation when the wanted key can be gained by a simple transformation of the main key (e.g. the main key contains leading zeros, the other one does not).
- **Dependency on another attribute** – In this relation, the wanted key is or can be derived from an attribute that is not the main key. Consequently, fetching data from a data source with such a key is dependent on fetching the needed attribute.

4.3.2 Traffic Minimization

One of the typical problems every integration system has to deal with is to minimize traffic between the system and data sources, resp. the system and its clients. The integration system should provide the answer in reasonable time and ought not to cause unnecessary

load to the data sources. Moreover, public data sources are usually limited with a number of accesses per a time unit so sensible communication is desirable.

There are several relevant possibilities of how to reduce traffic between the integration system and data sources, resp. clients. All of the followings will be designed and implemented in the integration system.

- **Cache** - Caching of responses from data sources should be an integral part of the integration system. In many cases, the system has to produce responses for identical requests in a short period of time and a cache would make the process a lot more efficient. Knowing the frequency of a data source's change, the system can combine cached data from such a source with fresh data from other ones. Cache control can be used on the system-client communication as well.
- **Filtering** - Next possibility of traffic reduction lies in filtering, i.e. sending a more specific query to a data source in order to reduce a number of results. Since the integration system should support a wide variety of data sources and each data source will surely have different possibilities of filtering as well as various filterable attributes, advanced requirements on configuration and integration mechanism arise.
- **Lazy Loading** - Lazy loading represents another way of performance increase and unnecessary traffic reduction. The principal consists in loading the particular information only when it is directly asked for. This approach is required especially in communication with public data sources providing data through restricted web services when the system must be careful when to call such an "expensive" service. Moreover, lazy loading addresses the issue of system availability because there are numerous situations when it is even not feasible to fetch all the data. Instead of assembling a complete response with all the information at once, the integration system should provide a lazy loading mechanism without breaking the integration transparency.

4.3.3 Identification of weak entities

Another problem which is needed to solve is identification of weak entities. A weak entity is dependent on another entity, i.e. to identify such a weak entity the another entity's identification is required. The design of querying the integration system must take this situation into account.

4.4 Glossary

Before a closer look at the processes, requirements and use cases, a short glossary of basic terms is introduced. Some terms may have a quite different meaning or a different level of granularity than in other domains or contexts so it is important to clarify them.

- **Master entity** – An essential business entity the integration system is oriented on that consists of a set of attributes. Clients query the system by specifying a type of the master entity and eventually a key or other additional information.

- **Catalog** – Metadata storage containing information about types of the master entities, their attributes and relations between each other.
- **Data source** – Information source that can be either local or remote offering data in various formats by various ways, e.g. a relational database, a CSV document, a web service etc.
- **Wrapper** – A software component standing above a data source responsible for querying, processing the results and mapping them in the global schema according to the metadata from the catalog.

4.5 Principle and Business Process

The system is oriented on master entities in terms of both integration and querying. That means the integration process builds instances of master entities composed from multiple data sources. Clients query the system to retrieve a master entity, resp. a collection of master entities by specifying information as a type of the master entity, a key, filters etc.

Figure 4.2 shows a schema of the integration system. The basic idea lies in creating wrappers above each data source, building a solid metadata storage in the form of a catalog and making an integration mechanism capable of building master entities from data of particular wrappers according to information from the catalog. A sequence of system processes is concisely illustrated in Figure 4.1.

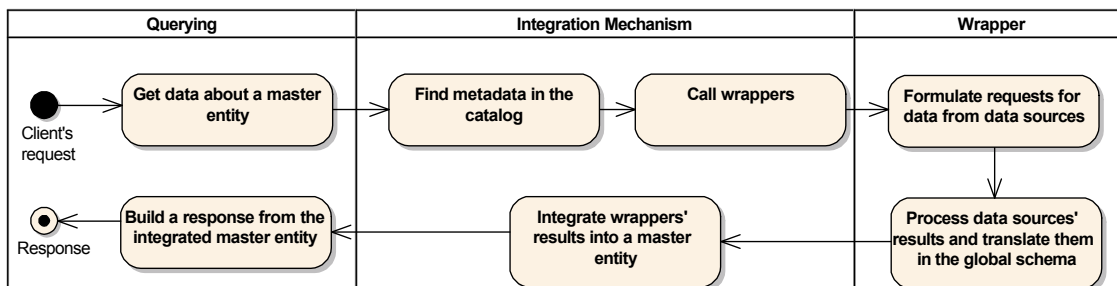


Figure 4.1: Process of data retrieval and integration

Above the integration system stands a UI client that dynamically generates a user interface according to metadata from the catalog and provides integrated data to end users. Within this diploma thesis, the integration system including an API for this UI client is to be designed, implemented and tested. However, development of the UI Client itself is a subject of a diploma thesis of Bc. Tomáš Hladík from the Faculty of Mathematics and Physics at the Charles University and is not a concern of this thesis.

4.6 Requirements

In this section the demands on the integration system are summed up in a set of functional and non-functional requirements. Functional requirements describe functions and behaviour

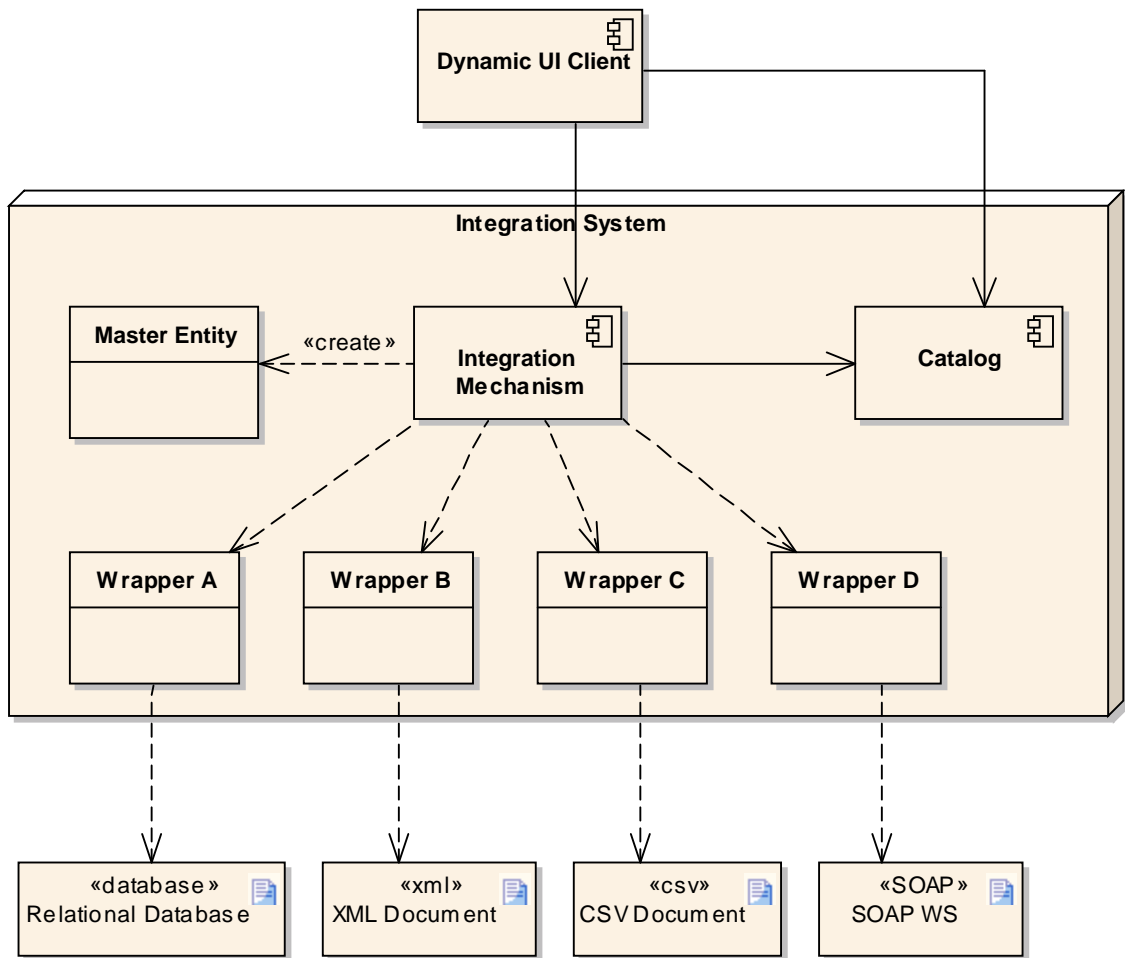


Figure 4.2: Basic schema of the system

of the system and its components. Non-functional requirements rather specify criteria, qualities and certain restrictions laid on the system and its development.

4.6.1 Functional Requirements

Functional requirements focus on the functions of the system, i.e. what the system is supposed to do. There are two primary perspectives we can look at the integration system functionalities from: the client's perspective (data accessing) and the administrator's perspective (system management).

Client's Perspective

1. The system shall enable to find a list of all master entities of a type.
2. The system shall enable to find a filtered list of master entities of a type.

3. The system shall enable to find a detail of a master entity.
4. The system shall enable to find a list of all types of master entities.
5. The system shall enable to find metadata about a type of a master entity.
6. The system shall enable to find metadata about an attribute type.

Administrator's Perspective

7. The system shall enable to register/edit/remove a data source.
8. The system shall register a name, a type, type-specific attributes and a cache validity time for each data source.
9. The system shall enable to define/edit/remove a master entity type.
10. The system shall register a name and a set of attribute types for each master entity type.
11. The system shall enable to define/edit/remove an attribute type of a master entity type.
12. The system shall register a name and fetching settings for each attribute.
13. The system shall enable to define/edit/remove a wrapper.
14. The system shall register a relation to a data source, a relation to a master entity type, a schema with mappings to attribute types and optionally a reference to implementation for each wrapper.
15. The system shall enable to clear cache for a data source.
16. The system shall enable to display usage information about data sources, master entity types and wrappers.
17. The system shall enable to display an error log.

4.6.2 Non-Functional Requirements

Integration Circumstances

18. The integration process shall be transparent to clients.
19. The system shall support a wide variety of data sources (at least relational databases, REST web services, SOAP web services, CSV documents).
20. The data model of the system (global schema) shall be described with a catalog in the form of a relational database or an XML document.
21. The system shall provide a solution for a situation when data sources use different keys for the same entities.

Demands on Design

22. The system shall support expandability with new data sources without needing to modify the system's source code.
23. The system architecture shall enable scalability.
24. The system architecture shall enable a transparent embedding of intermediaries.
25. The system interface shall provide a standard API to enable interoperability with other systems.
26. The system interface shall be explorable and self-describing.

Availability and Stability

27. Stability of the system shall be preserved if any data source fails or is not available.
28. The system maximal response time shall be set to 30 seconds to prevent too much long-standing load.
29. Incorrect client's input shall not put the system in an inconsistent state.
30. The system shall support a configurable lazy-loading mechanism to specify when an attribute should be fetched in order to enable feasible behaviour and better performance.

Security

31. The administration client shall not be accessible to an unauthorized user.
32. The web services for the administration purposes shall not be accessible to an unauthorized user.

Performance

33. The system performance shall be supported by caching of data retrieved from a data source.
34. The system performance shall be supported by cache control between the system and clients.

Documentation

35. The system documentation shall involve an installation manual.
36. The system documentation shall involve a manual for users (clients).
37. The system documentation shall involve a manual for system administrators.
38. The system documentation shall involve a documentation of source code.

4.7 Use Cases

Use cases capture what the actors are able to do with the system. There are two types of actors interacting with the integration system:

- **Client** – The UI Client or another software or human client querying the integration system.
- **Administrator** – The human administrator managing the integration system having extended the authorities.

4.7.1 Client's Perspective

An ordinary client can query the system for the integrated data and metadata. A diagram of relevant use cases can be seen in Figure 4.3.

4.7.2 Administrator's Perspective

Management of the catalog involves creating, updating, removing and finding master entity types including their attribute types and displaying the usage statistics (see Figure 4.4). The definition/update of a master entity type consists of the following actions:

1. Fill in the master entity type's name and its unique identification used for querying.
2. If the entity type is weak, select the entity type this type is dependent on.
3. For each attribute type specify its name, cardinality, type, whether it is a key and other settings used for intern processing.

Management of data sources includes creating, editing, removing and finding data sources, clearing their cache and displaying the usage statistics (see Figure 4.5). The definition/update of a data source consists of the following actions:

1. Fill in the data source's name.
2. Select a type of the data source. If the proper type is not available, select *other*.
3. Specify caching settings.
4. Specify type-dependent properties (e.g. information needed to establish a connection to a database).

Management of wrappers includes creating, editing, removing and finding wrappers, clearing their cache and displaying the usage statistics (see Figure 4.6). The definition/update of a wrapper consists of the following actions:

1. Fill in the wrapper's name.
2. Select a data source the wrapper communicates with.

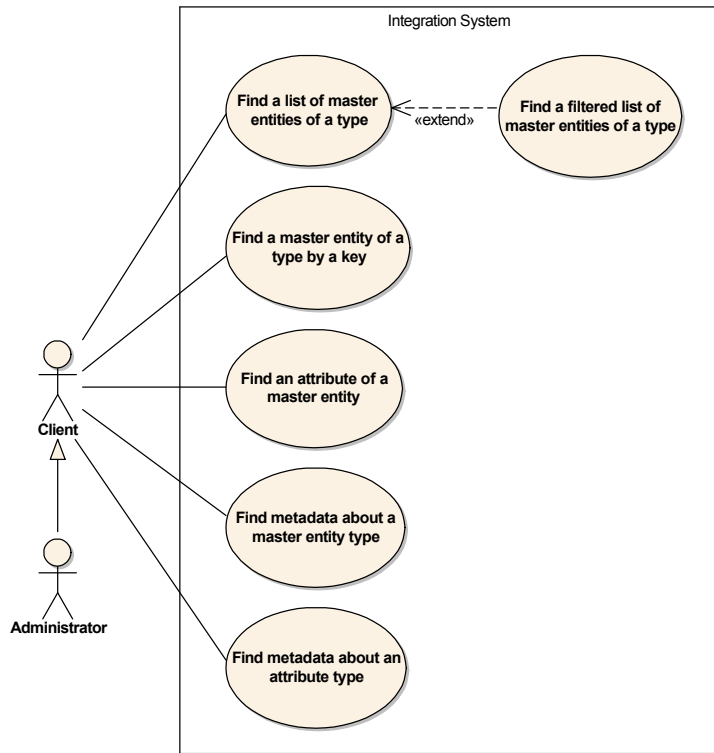


Figure 4.3: Use cases from the client's perspective

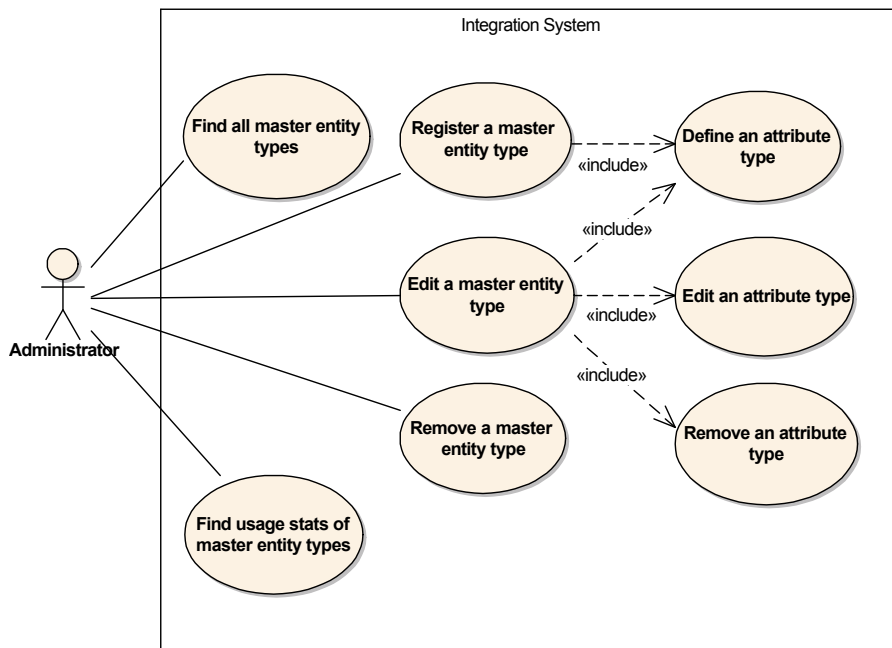


Figure 4.4: Use cases associated to the catalog management

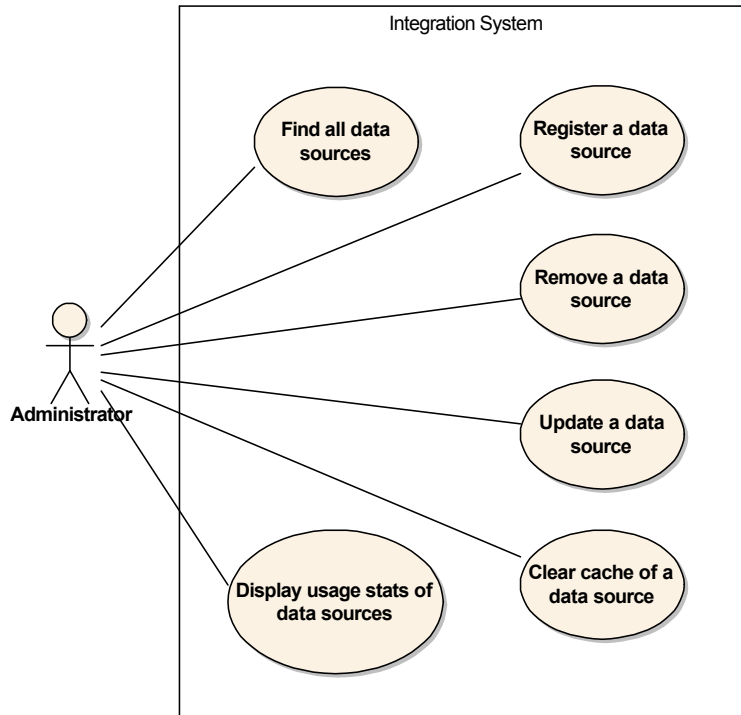


Figure 4.5: Use cases from associated to the data sources management

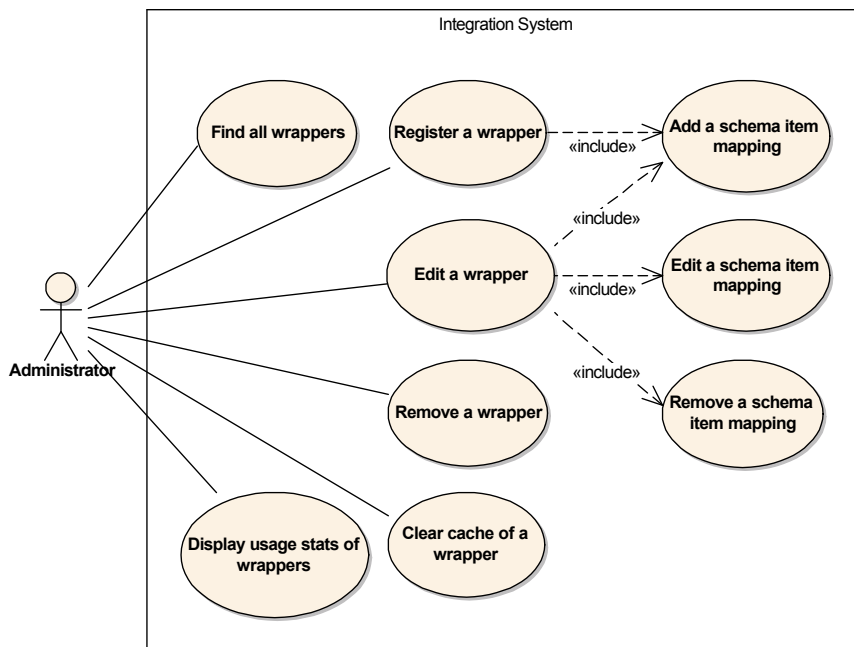


Figure 4.6: Use cases from associated to the wrappers management

3. Select a master entity type the wrapper fetches data for.
4. If there is an own implementation for the wrapper, specify the class name.
5. For each schema item, select an attribute type it maps to.

The above mentioned use cases still represent rather atomic operations. In reality, the administrator often needs to perform multiple atomic use cases to achieve a complex goal. A typical example of administrator's job is to integrate a new data source into the system, which requires the following tasks (see Figure 4.7):

1. Register a new data source.
2. If the data source brings a new master entity type, register it with its attribute types in the catalog.
3. Edit a master entity type to be enriched by data from the new data source.
4. Register one or more wrappers for the data source and map its or their schema to the global schema. If necessary, provide own implementations of the wrappers.

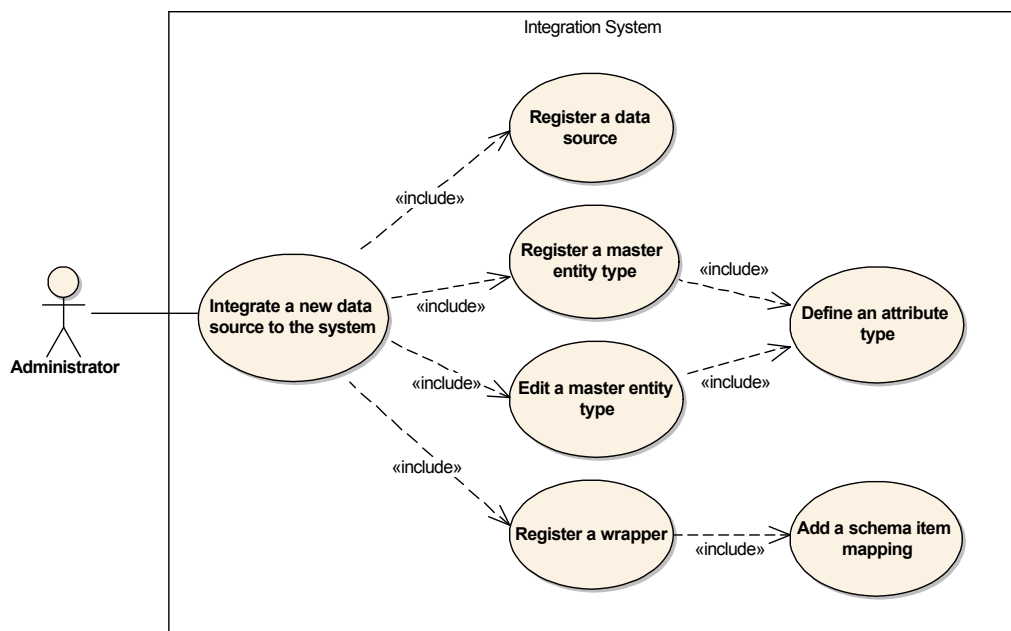


Figure 4.7: A composite use case to integrate a new data source with all the circumstances

4.8 Domain Model

A domain model diagram involving basic domain classes from a general, analytical point of view is shown in Figure 4.8. The diagram focuses on relationships between the core domain objects mentioned within the analysis.

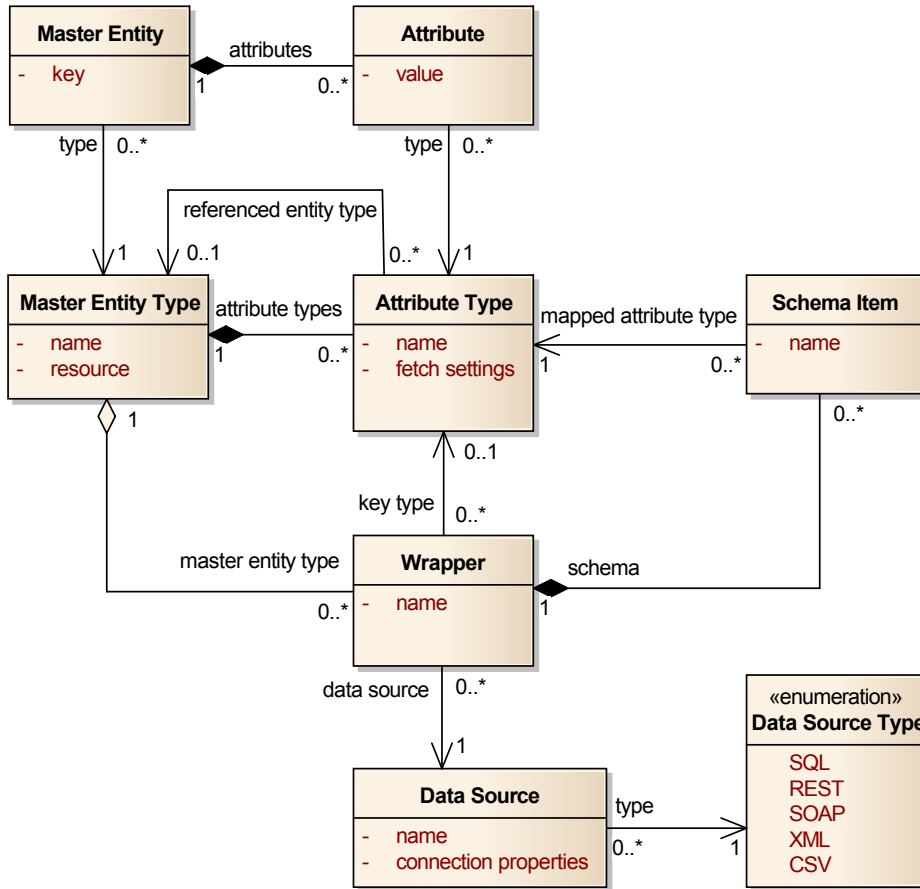


Figure 4.8: Analytical domain model

Master Entity composed of *Attribute* instances represents the object to be built during the integration process. Both of these classes are defined by types - *Master Entity Type* and *Attribute Type*, the essential objects specified in the catalog. These type classes determine all properties of master entities and their attributes and thus constitute the global schema of the integration system.

Each *Wrapper* stands above one *Data Source* and is able to request data and translate them into the global schema. To manage such a translation, *Wrapper* consists of *Schema Item* instances representing mappings between the schema of a data source and the global schema - *Attribute Type* instances. It is a common expected practice to build multiple wrappers above more complex data sources (e.g. relational database).

Chapter 5

Architecture

The design of the software architecture is a vital step in the software design process. The architecture describes the major system components and their relationships with the principles guiding the system design. It lays solid foundations which the system is built upon.

This chapter addresses the design of the architecture and the description of the particular system components with respect to the requirements derived from the analysis in Section 4.6. Thanks to knowledge gained from related works, the architecture is based on the best practices from existing solutions.

5.1 Architecture Overview

The architecture of the integration system is built upon the wrapper-mediator principle (see Section 2.2.4.3). A wrapper operates on top of a data source and maps its native representation into a common data model. A mediator processes incoming requests, delegates them to appropriate wrappers and combines, integrates and abstracts the obtained data. Such an approach allows to divide data accessing from the integration business logic leading to better maintainability. Also, thanks to flexibility of the data access it is possible to form a wrapper above a wide variety of data sources while the mediation part is completely independent on the data sources' paradigms.

Experience learned from the TSIMMIS (see 3.1.1) project says that the development of mediators could often be hard and time-consuming. To overcome this inconvenience, let us transfer the complexity from the mediator development into building a solid metadata storage (inspired by the Knowledge Base in OWSCIS (see 3.1.3)) by introducing a catalog. The catalog will maintain all metadata needed for the integration part of the process posing as the global schema. As a consequence, the mediator will then stand for a universal mechanism performing the integration according to the metadata obtained from the catalog. Therefore, to include a new data source to the integration system, the administrator has to create a wrapper together with its mapping to the global schema. The mediator does not need to be modified at all.

The essential idea of this integration system lies in master data orientation. From the user's point of view, the system will provide data about master entities specified with an

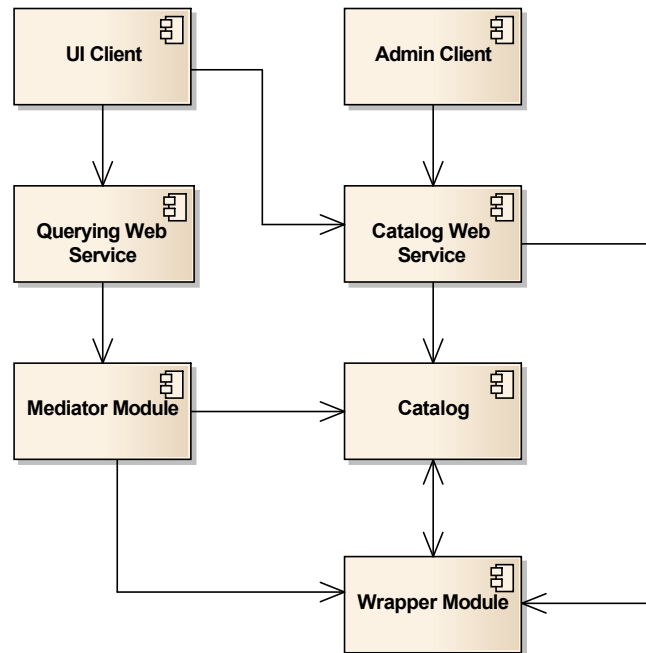


Figure 5.1: Basic overview of the architectural components

identifier. This draws an intuitive image of how the catalog could look like - a list of master entities definitions along with their relations and attributes.

The focus on master entities also indicates the way of querying the integration system. The desired solution requires an easy-to-use, standard method of communication between clients and the system. Since the possibilities of querying have to be restricted in order to work with various data sources with access limitations returning high quality answers, no complex query language is needed. Among the desirable features, which the integration system should dispose of, belong scalability and caching to achieve better performance. Therefore, REST seems to meet these requirements and so becomes a well-suited architectural style to be followed here.

Combining the wrapper-mediator and the REST architectural principles, a basic skeleton of the integration system architecture can be introduced. Figure 5.1 shows the seven modules and web services forming the system architecture:

1. **Catalog:** A central element of the integration system managing the metadata storage. It provides all the information about master entities and their attributes forming the global schema.
2. **Wrapper Module:** The bottom of the system is formed by the Wrapper Module containing the particular wrappers and their management. Wrappers are intended to communicate with data sources and translate their data representation into a common data model using mappings between data sources' schemas and the global schema.

3. **Mediator Module:** Based on the metadata from the catalog, the Mediator Module delegates the incoming requests on wrappers, processes the received data and returns an integrated master entity, resp. a list of master entities.
4. **Querying Web Service:** A web service providing the integrated data to the end user/application following the REST principles. It processes a request into a form the Mediator Module understands and transforms its answer to a standard, well transportable and processable format.
5. **Catalog Web Service:** This web service provides an API for managing of data sources, managing of master entities, managing of wrappers and creating mappings between data sources' schemas and the global schema. Then, it offers public metadata to the UI Client to enable to build a dynamic UI on top of the integration system.
6. **Admin Client:** A tool intended for system administrators serving for the catalog, data sources and wrappers management that communicates with the system via the Catalog Web Service.
7. **UI Client:** A client application intended for end users capable to build dynamic UI. It explores the global schema using the Catalog Web Service and submits the queries to the Querying Web Service.

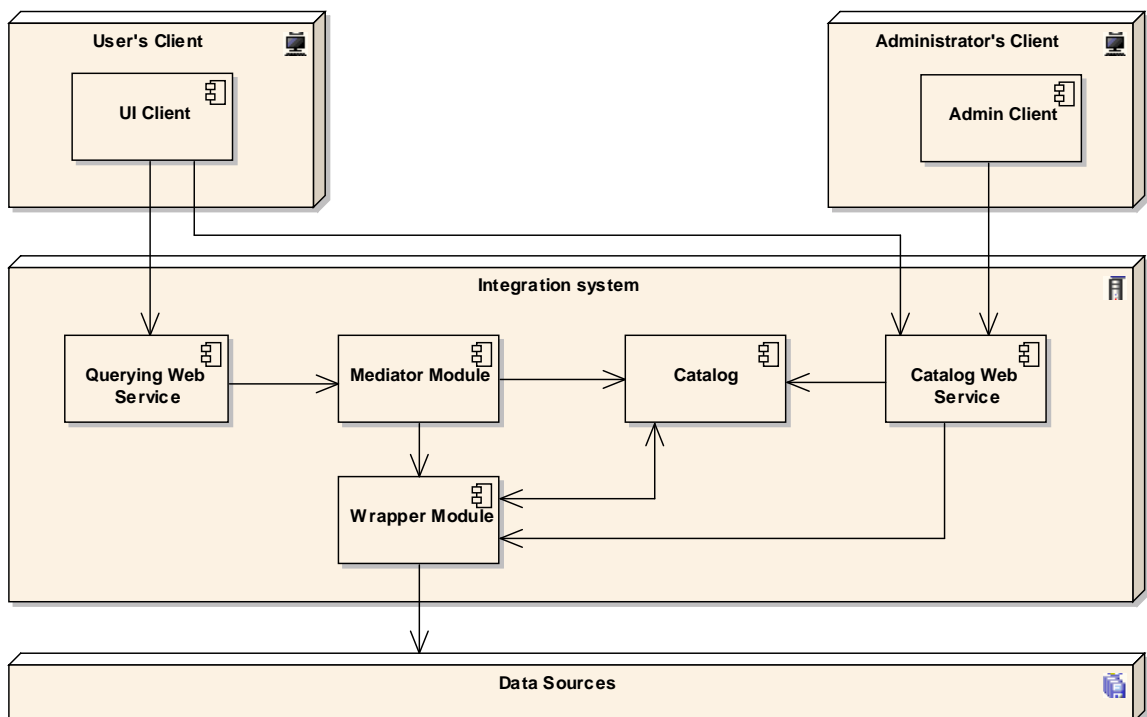


Figure 5.2: Deployment schema of the integration system

How to deploy the integration system components is shown in Figure 5.2. This diagram clearly illustrates the physical location of particular components.

As the basic system components and their relationships are found, we can look at them in a more specific way. However, the UI Client is not a concern of this thesis, so the following section describes only the components belonging to the subjects of this work.

5.2 Components of the Architecture

5.2.1 Catalog

This is the essential component of the system, a central knowledge repository needed for the integration process. It contains Catalog Service and Master Entity Type Manager (see Figure 5.3).

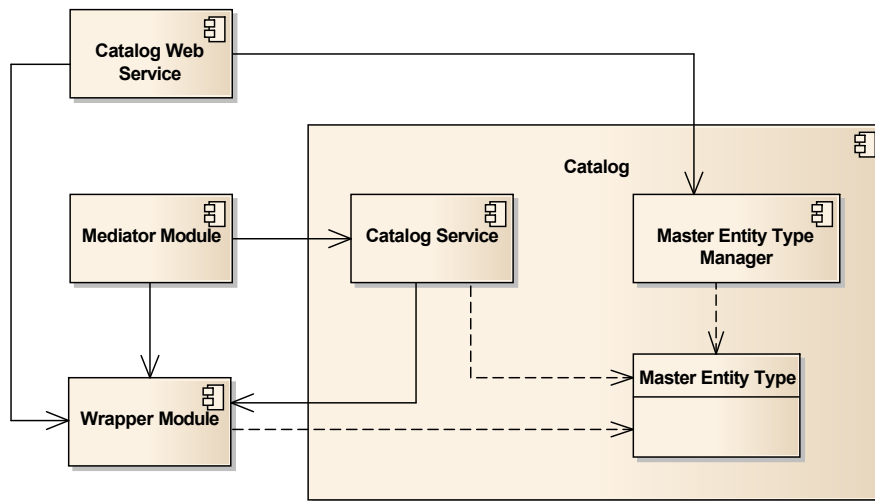


Figure 5.3: Catalog

The Catalog Service provides knowledge for internal purposes of the integration system. It prepares complete metadata needed by the Mediator Module including information about the registered wrappers.

The Master Entity Type Manager is responsible for the catalog management, i.e. definition of master entity types and its attribute types. It includes functionalities to create, edit, remove and retrieve the content of the catalog and is intended to be used by the Catalog Web Service.

5.2.2 Wrapper Module

This module takes care of the communication with data sources. It includes the Data Source Manager, the Wrapper Manager and particular wrappers and data sources definitions (see Figure 5.4).

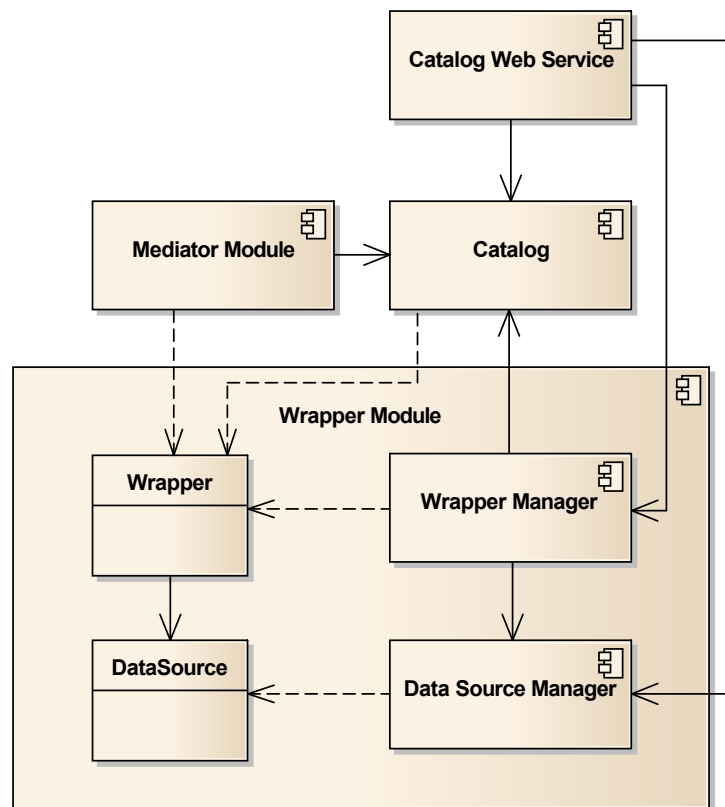


Figure 5.4: Wrapper Module

The data sources definition holds the information about the data sources which the integration system works with. Each data source is described by a type and type-specific information needed for establishing a connection.

The wrappers definition contains settings of all wrappers. For each wrapper, it includes an implementation of retrieving and processing the data and mapping its schema to the global schema. A common practice is to have more than one wrapper on top of a data source, especially regarding more complex data sources like a relational database.

The Data Source Manager provides functionalities to manage data sources definition intended to be used by the Catalog Web Service.

The Wrapper Manager is responsible for wrappers management including definitions of mappings from data sources' schemas to the global schema. Based on these mappings, the Catalog is able to find relevant wrappers and provides them along with necessary metadata to the Mediator Module.

5.2.3 Mediator Module

The Mediator Module has two main tasks - query decomposition managed by the Query Decomposer and result merging arranged by the Results Merger (see Figure 5.5). This

module is responsible for the integration process itself depending on information from the Catalog.

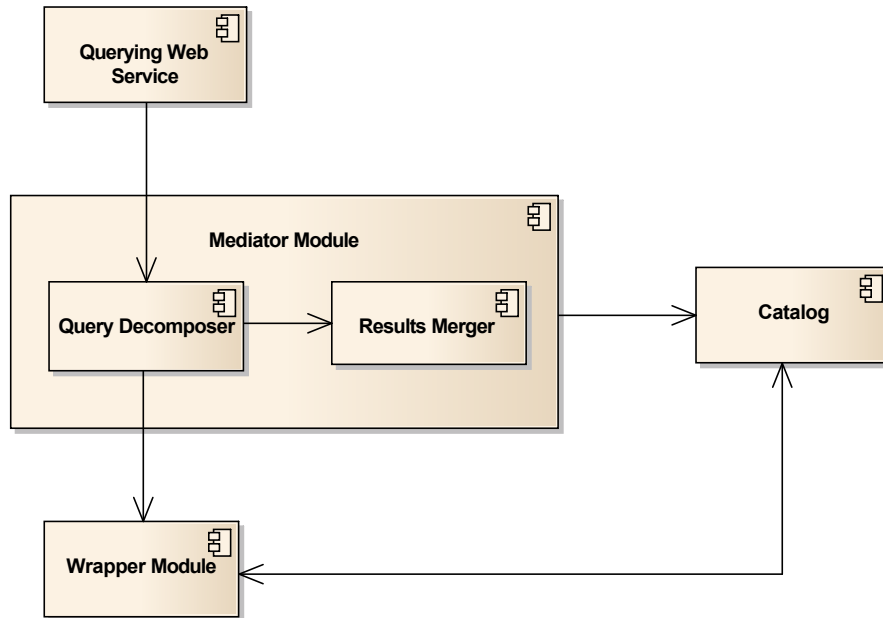


Figure 5.5: Mediator Module

The Query Decomposer receives a query from the Querying Web Service, then gains metadata about the master entity from the catalog and based on this information, it formulates the subqueries for the relevant wrappers.

The Results Merger collects responses returned from the wrappers, recomposes them in a master entity object, resp. a collection of master entity objects, which then sends back to the Querying Web Service.

5.2.4 Querying Web Service

This component plays the role of the entrance gate to the integration system that users or client applications communicate through. It is a RESTful web service (more information in Section 5.3.2) consisting of two main parts - the Query Processor and the Response Formatter (see Figure 5.6).

The Query Processor takes an incoming query formed simply by an HTTP request - a combination of a URI and optional HTTP headers. It processes all the useful information, i.e. a master entity identification, a key or other filters from the request and hands it over to the Mediation Module.

The Response Formatter serves for formatting the master entity object, respectively a list of master entity objects, returned from the Mediator Module to a standard, well transportable and processable format. As soon as the formatting is ready, the Querying Web Service returns an HTTP response with the integrated data to the client.

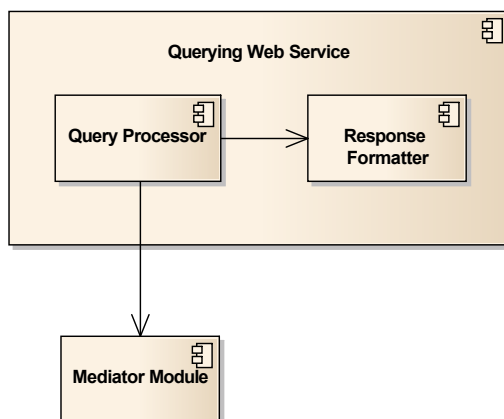


Figure 5.6: Querying Web Service

5.2.5 Catalog Web Service

The Catalog Web Service is a RESTful web service representing an API focused on catalog and wrappers management. First of all, it provides means for managing master entity types, attribute types, data sources and wrappers as well as creating mappings between the wrappers' schemas and the global schema. The public part of the API serves for accessing metadata from the catalog according to which the UI Client is able to build a dynamic user interface. Then, it offers secondary supporting functionalities for the Admin Client.

5.2.6 Admin Client

This component stands for an administration client application communicating with the system through the Catalog Web Service. It is intended to enable the administrator to easily manage the integration system, e.g. add a new data source with associated wrappers together with an extension of the global schema.

5.3 REST

5.3.1 Constraints and properties of REST

The architecture is based on the REST principles which have been widely accepted across the Web and adopted in the field of web services development. Among the great benefits of this approach belong good scalability, cacheability and an easy-to-use uniform interface. Noteworthy, these significant advantages bring a trade-off lying in a slightly lower performance due to the stateless communication and a standardized form of information transferring.

Roy Fielding [9] introduced specific constraints that compose the REST style:

1. **Client-Server** – Separation of clients from servers. Clients are not concerned with data storage and servers are not concerned with the user interface.

2. **Stateless** – The client-server interaction is stateless meaning that every request must contain all the information needed to be understood and processed. This indicates properties as visibility, reliability and scalability.
3. **Cache** – In order to compensate the performance loss of REST, one of the constraints refers caching. Responses should be labeled as cacheable or non-cacheable, the cacheable could be later reused by a client for an equivalent request.
4. **Uniform Interface** – This fundamental property of REST services decouples the architecture making it simpler and more intuitive. The uniform interface is defined by four constraints: identification of resources, manipulation of resources through representations, self-descriptive messages, and HATEOAS (Hypermedia As The Engine Of Application State).
5. **Layered System** – The layered system within this context ensures transparency of client-server communication enabling to place intermediaries in front of the server, e.g. shared caches, load balancing or a security proxy.
6. **Code-On-Demand** – The only optional constraint of REST saying that services might extend/pre-implement client's functionality by downloading and executing code as scripts or applets.

A web service can be characterized as RESTful if it fulfills all of these constraints [42]. If any of them is violated, the service cannot be considered RESTful.

5.3.2 RESTful system architecture

The design of the integration system architecture ought to be RESTful, so it should conform to each of the required constraints. From the look at the components introduced in the previous section, the **client-server** architectural style is evident. The UI Client and the Catalog Editor are the clients while the rest of the components stands on the server side separating the user interface from the integration logic and data access.

The communication between a client and the server is utterly **stateless**. The integration system does not hold any sessions or other information about a client's state. Each request contains all the required information as the master entity identification, its key or other additional information in HTTP header fields. In addition, since the Querying Web Service provides data exclusively for reading, not its creating, modifying or removing, every request is idempotent, i.e. can be called multiple times without changing the result.

In order to provide reasonable latency and enable more comfortable and fast-acting client's behaviour, **caching** should be applied to the integration system. It is convenient to label each response of the Querying Web Service as cacheable and thus notify the client that the response may be reused later after sending the equivalent request.

The subject of caching also relates to another constraint - a **layered system** architecture. Important parts of the system architecture are the intermediaries standing transparently between the server and the clients.

- One of them is a web cache which could be a third-party proxy caching HTTP responses. It saves HTTP requests and responses for some given time and in case the same request comes, the caching proxy serves it and thus saves the computing performance of the integration system.
- Another intermediary is a security one - an anti-DDoS (Distributed Denial of Service) proxy service protecting the integration system against DDoS attacks. This component guards not only the security of the system itself but also protects the data sources that are not secured with a suchlike technology.
- Based on the performance testing in practice, the architecture might be enhanced with a load balancer distributing workloads across multiple instances of the integration system.

The Querying Web Service respects the rules of a REST **uniform interface**. The data provided by the integration system are in fact resources - the master entities identified by a URI that are manipulated (only read in this case) through their representation (JSON). Each response is self-descriptive enough to be understood (MIME type, cacheability indicator, HTTP code etc.). The web service offers a single entry point capable of data exploration so the client needs to know only the root URI. Each resource (master entity) having a relation to another resource includes a hyperlink enabling its easy retrieval.

This section proves the system architecture to be RESTful, i.e. it fulfils all the required constraints of REST. Among the gained benefits belong better scalability, transparent cacheability and security, a uniform, easy-to-use and self-descriptive interface based on standard technologies and finally separation and decoupling of concerns leading to more effective and better maintainable development. In relation to the requirements laid on the system in the analysis (see Section 4.6.2), the RESTful interface guarantees the non-functional requirements 23, 24, 25 and 26.

Chapter 6

Design

This chapter refers to the design phase of software development proposing a specific description of how to implement the functionalities specified during the analysis (see Section 4). Within this chapter, the analytical model will be transformed to an object model, where there will figure not only entities of real world but software classes in particular.

The architecture of the integration system has already been proposed in previous Section 5. In this chapter, a more detailed design of particular software components, according to which it would be possible to implement the essential parts of the system, is introduced.

6.1 Catalog

The catalog is a central element of the integration system managing the metadata storage. It provides all information about master entity types and their attribute types forming the global schema.

6.1.1 Master Entity and Attribute Types

The class diagram in Figure 6.1 illustrates major classes participating in the catalog. The core classes of this component are `MasterEntityType` and `AttributeType` forming the global schema of the system. The `Catalog` interface defines necessary operations needed by other system components communicating with the catalog and the `ApplicationCatalog` stands for its referential implementation. To retrieve the underlying data about master entity types and their attribute types, the `ApplicationCatalog` uses a DAO (Data Access Object) class - `MasterEntityTypeDAO`.

Each master entity type is uniquely specified with a resource - a string used by REST to identify the resource type. Attribute types are uniquely identified with its master entity type and a name. An attribute type can be marked as `multiple` which means that instead of one value, it contains a list of values. There are two kinds of attribute types within the meaning of its possible values:

- **Literal** – value(s) of the attribute is/are literals
- **Referential** – value(s) of the attribute is/are references to another master entity

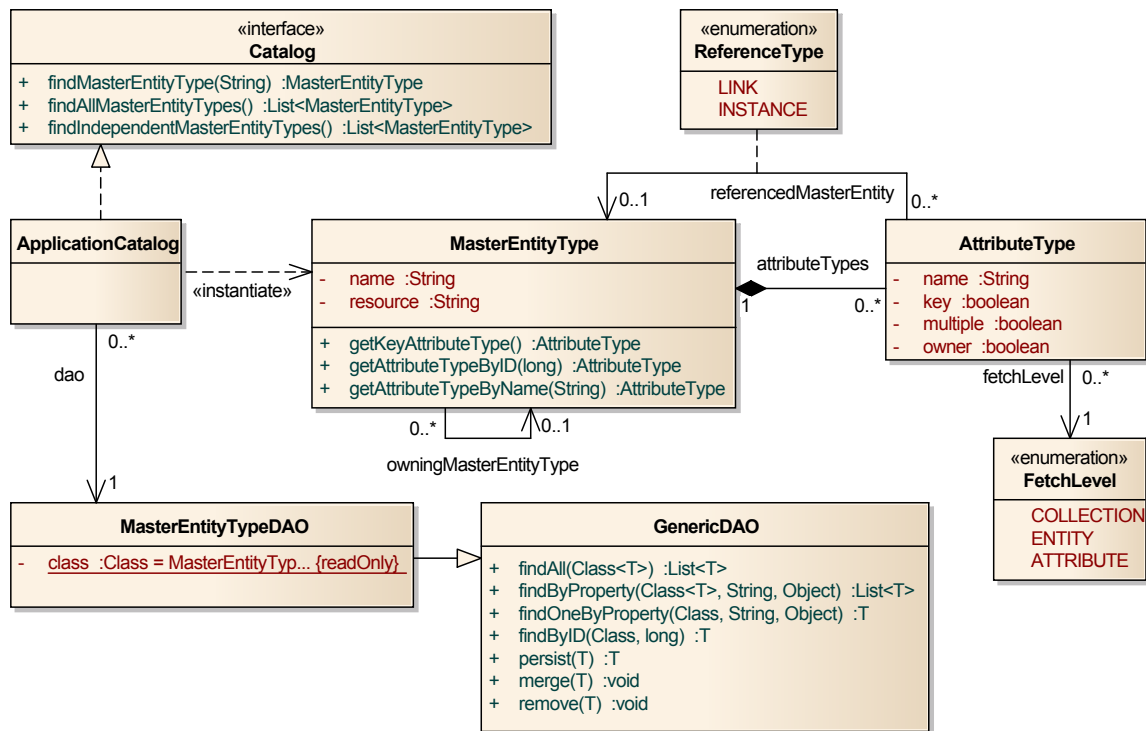


Figure 6.1: Design classes of the Catalog

6.1.2 Owned Master Entity Types

There are situations when a master entity type is dependent on another master entity type in the sense of identification, i.e. a master entity of such a type needs for its identification also the information about the entity which it is dependent on. It is an analogy to weak entities in terms of relational databases. Within this integration system, such a master entity type is called *owned*.

In the REST environment, each resource has to be identified by a unique URI, so the master entities of an owned type need the identification of their owner to build up its URI. That is the reason for the self-reference `owningMasterEntityType` in the `MasterEntityType` class. The `AttributeType` has a field `owner` indicating whether it stands for a reference to the owning master entity.

6.2 Wrapper Module

The wrappers are responsible for communicating with data sources and transforming the obtained data into the global schema specified by the catalog. Figure 6.2 shows a class diagram of the Wrapper Module containing the most important classes of this component.

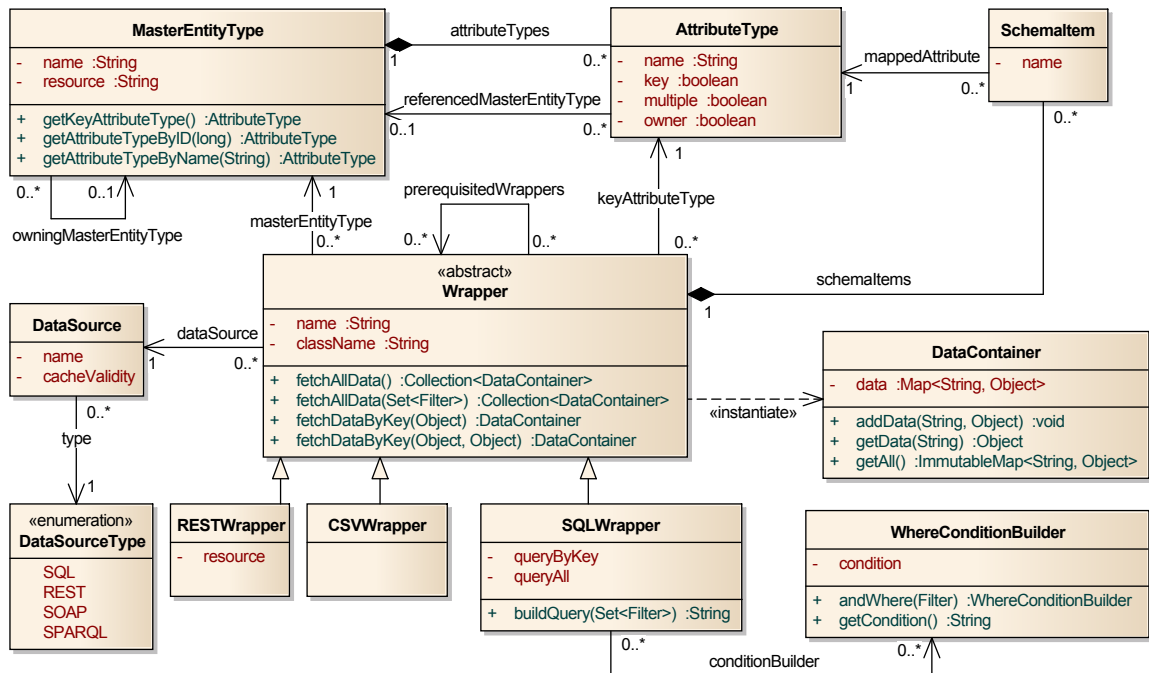


Figure 6.2: Design classes of the Wrapper Module

6.2.1 Principles of Wrapper's Work

Each wrapper stands above a data source and is connected to a master entity type whose attributes it fetches. In order to translate data obtained from the data source to the global schema = attributes of the master entity, wrappers carry these schema mappings within instances of the `SchemaItem` class.

Having to support various data sources means to work with various types of wrappers whose implementations might substantially differ. However, the mediator must not be concerned about these differences, so it is crucial that all wrappers comply with a common interface. Abstract class `Wrapper` represents such an interface defining a contract between the mediator and all wrappers:

- Fetch all data and return a collection of data containers.
- Fetch all data applying given filters and return a collection of data containers.
- Fetch data by a key and return a data container.
- Fetch data by a key and an owning master entity's key and return a data container.

All these operations return data in the form of an instance/a collection of instances of the `DataContainer` class which keeps its data in a key-value pair data structure. The keys stand for identification of attribute types so that mediator understands the received data and knows how to evaluate them.

6.2.2 Building Wrappers

The system provides referential implementations of wrappers for some types of data sources. However, the possibilities of providing default behaviour may differ a lot due to nature of the type of a data source. Generally, such a default wrapper can be implemented only for a data source with highly structured data where it is possible to submit a query returning the data in a fixed schema.

For example, system offers a default behaviour of wrappers for relational databases in the `SQLWrapper` class. To work properly, the wrapper needs to have defined SQL queries for fetching all rows, resp. a row by a key. The wrapper then maps the column names, resp. column labels specified with column aliases, to the global schema based on the schema item mappings. If a support of filtering is desirable, the only thing needed to implement is `buildQuery` method for building the SQL query overriding the SQL query for all rows while taking into account given filters.

Since the system should support nearly arbitrary data source, it is often necessary to implement an own wrapper. Sometimes a complete implementation from scratch is needed (but always has to extend the `Wrapper` class), but mostly it is enough only to override a method for results processing and exploit existing functionalities of the referential implementation for querying the data source. In some situations, the default referential implementation fully meets the requirements.

In any case, when a referential implementation does not fully satisfy the desired needs, a wrapper implementation has to be made and submitted to the integration system. The attribute `className` in wrappers then stands for a reference to such an implementation, so when the wrapper is needed the system loads the implementation from the given full class name. The possibility to create a wrapper with custom logic for retrieving and processing data in the form of a plugin is one of the essential features of the integration system allowing to support a wide range of data sources.

6.2.3 Dependent Keys

Not all wrappers can use the main key of a master entity type for fetching, some may need another attribute to identify the entity within the data source. The reason lies in different identification across data sources when the same entity is identified by different keys.

When the mediator calls a method to fetch data by key, it already has to pass the right key so obtaining the dependent key is a mediator's task. Nonetheless, the wrapper has to define what key it works with and for that, there is the relation `keyAttributeType`. When the integration process begins, the wrapper also needs to know which other wrappers it is dependent on, i.e. which wrappers fetch the attribute needed to assemble the dependent key. This information is kept within the `prerequisitedWrappers` self-reference.

6.2.4 Cache

In order to reduce traffic between the integration system and data sources, wrappers should cache their results. The reason why caching is dealt with on this level, i.e. caching of wrappers' results instead of mediator's results, is caused by various temporal validity of

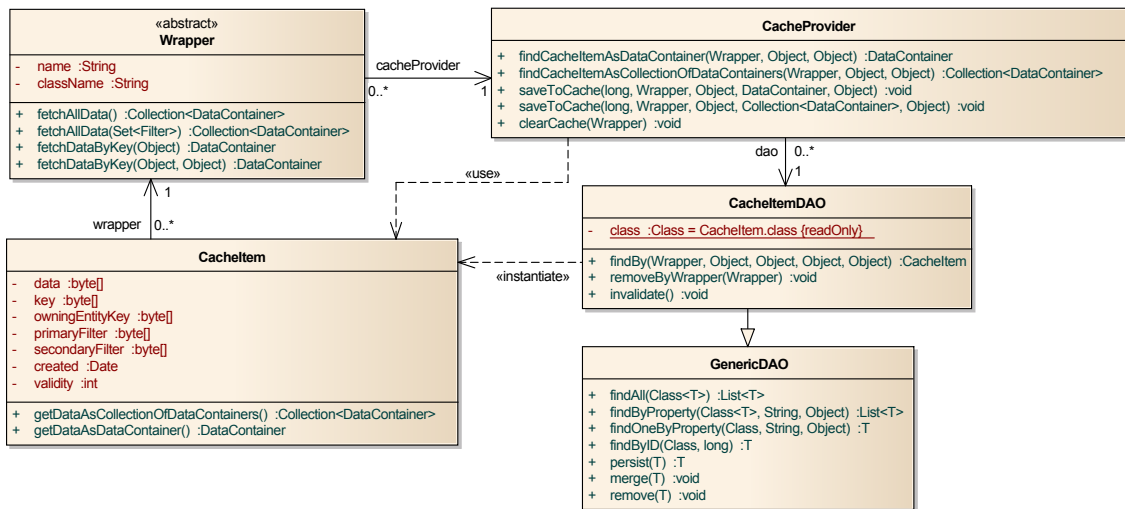


Figure 6.3: Design classes for caching of wrappers' results

particular data sources. Therefore, each data source keeps temporal cache validity telling their wrappers how long the cached results can be considered as valid.

A class diagram in Figure 6.3 shows the structure of classes involved in the caching process. A wrapper communicates with the cache via `CacheProvider` that manages instances of the `CacheItem` class stored in a local database. Saving the cached data requires all the necessary information to identify the request on the wrapper in order to distinguish an identical request next time. Besides a key or an owning entity's key, there are primary and secondary filters that have to be general enough to capture various wrappers' way of requesting data (e.g. SQL query, HTTP request etc.).

6.3 Mediator Module

Based on the metadata from the catalog, the Mediator Module delegates the incoming requests on wrappers, processes the received data and returns an integrated master entity, resp. a list of master entities. This component holds the fundamental business logic of the whole system.

6.3.1 Mediation Process

The process of mediation lies in an effective decomposition of incoming requests and merging particular parts of information into an integrated form - master entities. Figure 6.4 includes the most important classes involved in the mediation, in this case *mediation by a key*. Classes specific for *mediation all* are to be found in Figure 6.6.

To clarify the used terms in this area, it is important to explain the followings:

- **Mediation by a key** – Refers to a process of integration one master entity specified by a key

- **Mediation all** – Refers to a process of integration of a collection, i.e. a list of all master entities of a given type that can be filtered

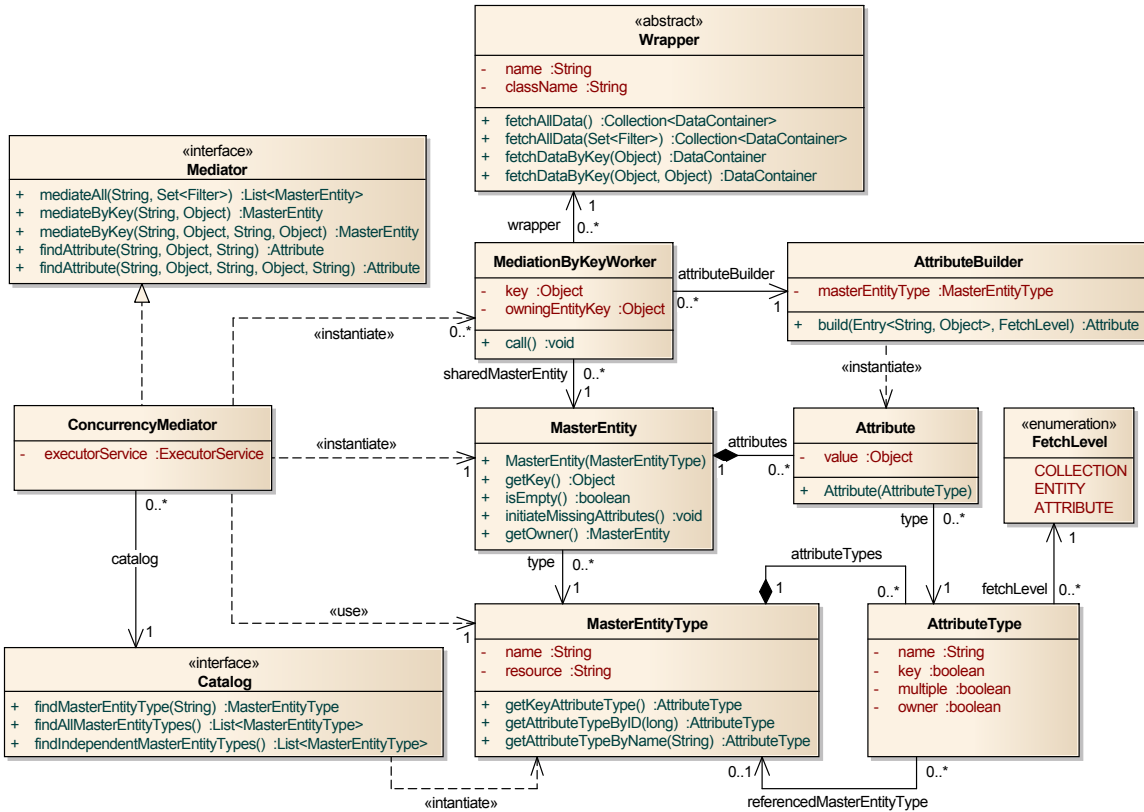


Figure 6.4: Design classes of the Mediator Module involved in mediation by key

The interface `Mediator` defines a contract for mediators, `ConcurrencyMediator` is its referential implementation focusing on an effective concurrent approach. After getting required metadata from the catalog, the mediator prepares a set of workers, one for each wrapper, and asynchronously calls them via the `ExecutorService`. The processes of mediation by a key and mediation all are slightly different from this point.

- Mediation by a key** – When mediating by a key, the mediator constructs an instance of `MasterEntity` and passes it to each worker (`MediationByKeyWorker`). Each worker then requests data from its wrapper, builds full-fledged attributes from the received data containers and adds them to the shared master entity. A slightly simplified process of mediation by a key is captured by a sequence diagram in Figure 6.5.
- Mediation all** – Mediation all gradually builds a collection of integrated master entities instead of just one instance. In this case, the mediator initializes an instance of `MasterEntitiesSharedCollection`, a thread-safe collection, in which the workers (`MediationAllWorker`) send partial master entities built from data of their wrappers. The shared collection either stores a new master entity or update an existing one.

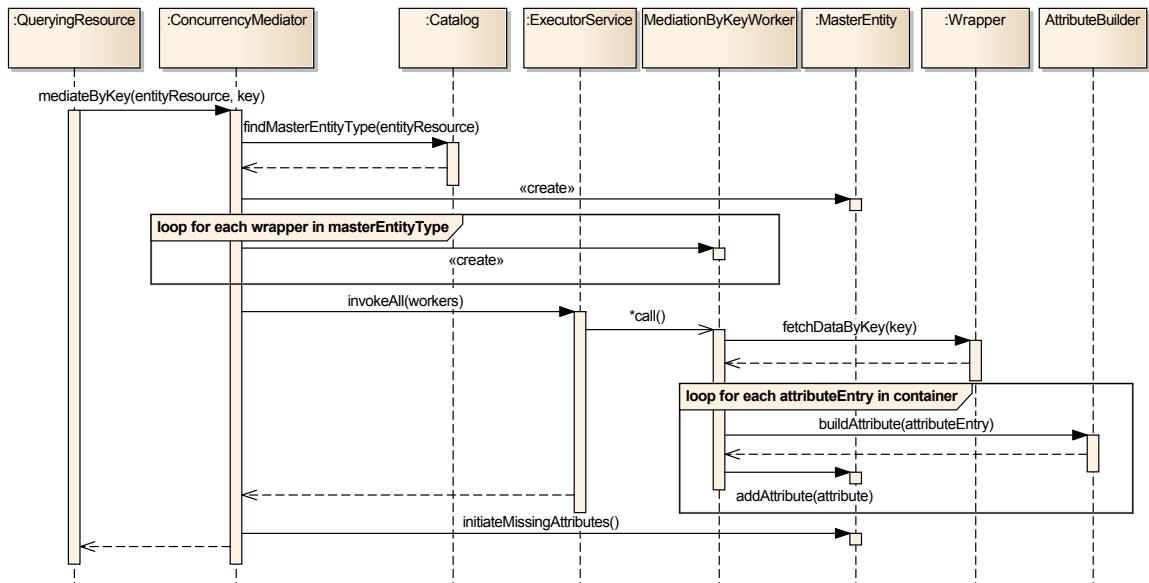


Figure 6.5: Sequence diagram of mediation by key

Among other mediator's operations belongs finding a specific attribute of given master entity, resp. owned master entity. The process is very similar to mediation by a key except that only wrappers required to fetch such an attribute are contacted.

6.3.2 Waiting for Dependent Keys

As described in Section 6.2.3, there can be wrappers whose data source uses different key than the main key within the system to identify an entity. Wrappers define what key they use and it is up to the mediator to pass the right key.

When mediating by a key, workers whose wrapper uses a dependent key, i.e. is dependent on results of another wrapper, must wait till the required attribute is fetched. Until then, it is not possible to fetch the data.

Moreover, the problem of dependent keys does not refer only to mediation by a key but concerns mediation all, as well. Unlike the mediation by a key, in this case it is possible to retrieve data from data sources immediately, but waiting is needed before merging. To properly recognize the same entities when merging data from multiple wrappers, the `MasterEntitiesSharedCollection` identifies each master entity with the main key. However, if a wrapper uses a dependent key, it is difficult to find the proper master entity in the collection. The solution lies again in waiting till the required attribute is fetched, then it is possible to find the master entity by this attribute and update it with data from the dependent wrapper.

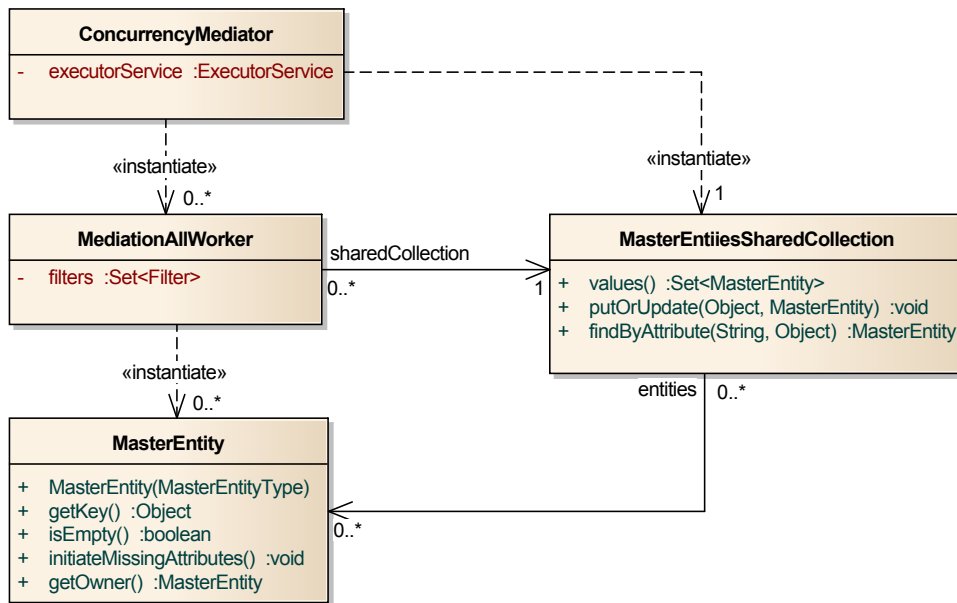


Figure 6.6: Design classes specific for mediation all

6.3.3 Lazy Loading

The vital feature of the integration system is to manage when an attribute should be fetched eagerly and when it is not so convenient or even feasible. Because a lot of data sources might be public web services with specific restrictions in access, the system should provide possibilities to control their workload. For example, when mediating all data of a master entity type, the system probably should not call a web service for each found entity. Advanced configuration of lazy loading should overcome these limitations.

The integration system supports three levels of fetching that each attribute type can be assigned to:

- **Collection Level** – Attribute of this level is always fetched. The mediator should contact each wrapper mapped to the attribute even when mediating all.
- **Entity Level** – Attribute is fetched only during mediation by a key. When mediating all, wrappers mapped to such an attribute are not contacted and the attribute remains with null value.
- **Attribute Level** – Attribute is fetched only when accessed directly. Neither mediation all nor mediation by a key requests data from wrappers mapped to this attribute.

The fetch level of the whole wrapper is determined by the most specific fetch level of mapped attribute types.

If an attribute stands for a reference to another master entity, there are two ways of how to represent its value:

- **Link** – Only a link to the master entity represented as its key.
- **Instance** – Value of the attribute is an embedded instance of the master entity.

6.3.4 Filtering

Another way of how to reduce the amount of unnecessarily transferred information is filtering. Among operations of wrappers belongs according to the interface fetching with filters. However, it is up to each wrapper whether it really implements the filtering so there is no guarantee that the retrieved data will really be filtered. Filtering in wrappers reduces the traffic between data sources and the system and if the wrapper does not implement filtering, it just returns all the data.

Therefore, filtering is also one of the mediator's task in order to reduce traffic between clients and the system and return the corresponding data according to a client's request. So, after building a collection of data sources, the mediator should filter the collection according to given filters.

6.3.5 Stability

The integration system cooperates with multiple data sources. Whenever any of the data sources does not respond or behaves unexpectedly, the system stays stable and returns data from the rest of the data sources. To achieve such stability, no mediation worker can endanger the mediation process so it must be treated as a single thread checked for any exceptions. Every exception caused by a third party is caught, logged into the database and clients are notified about unsuccessful communication with the data source along with data from other data sources.

6.3.6 Usage Statistics

Mediator together with its workers is the ideal place for tracking usage statistics. For every request, information about usage of the master entity type, all contacted wrappers and their data sources is registered and stored in the database.

6.4 Querying Web Service

A web service providing the integrated data to the end user/application following the REST principles. It processes a request into a form the mediator understands and transforms its answer to a standard, well transportable and processable format. The class diagram in Figure 6.7 includes the significant classes of this module.

6.4.1 REST API Design

Clients send HTTP requests that are processed by the `QueryingResource` class. From the given URI, this class translates the request to a function according to this mapping:

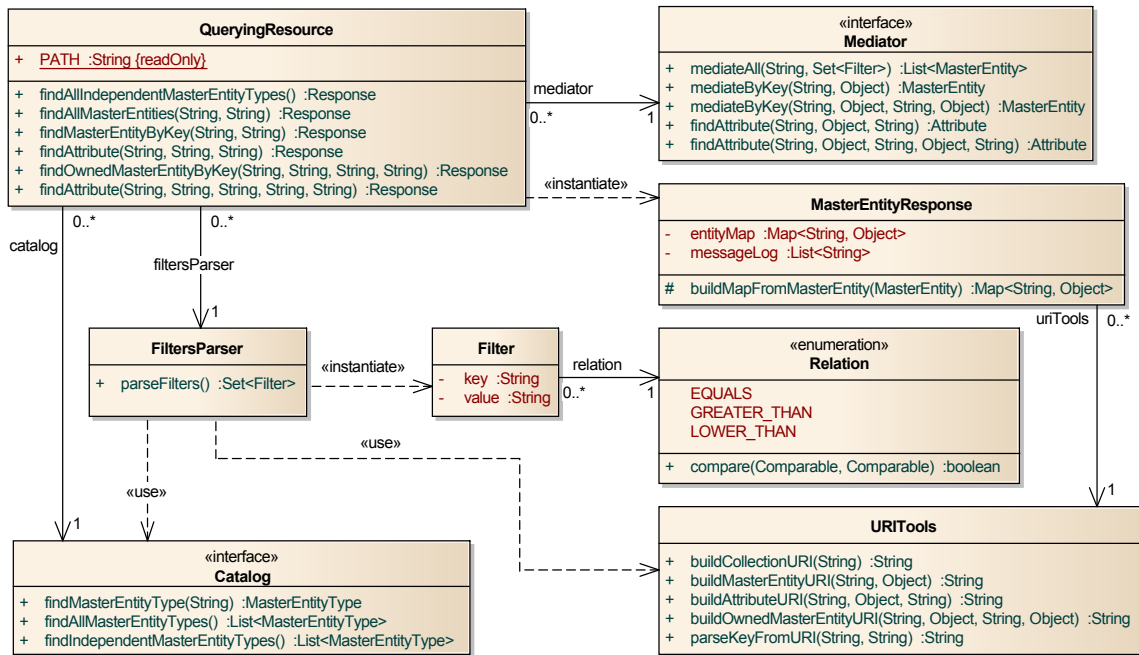


Figure 6.7: Design classes of the Querying Web Service

- **/data** – Acts as a crossroad serving a list of all independent master entities’ URIs in the form of the next pattern.
- **/data/{entityResource}** – Corresponds to a request for all master entities of a type identified with given **entityResource**. These requests support filtering within a custom HTTP header in the form:

$$\{\text{attributeName}\}\{\text{relation}\}\{\text{someValue}\}$$

where **relation** stands for =, > or <. Multiple filters can be separated by a semicolon. The **QueryingResource** parses these filters using **FiltersParser** and hands them over to the mediator as a set of the **Filter** instances.

- **/data/{entityResource}/{entityKey}** – Corresponds to a request for a detail of a master entity with given **entityResource** identified with **entityKey**.
- **/data/{entityResource}/{entityKey}/{attributeName}** – Corresponds to a request for an attribute named as **attributeName** which belongs to a specified master entity.
- **/data/{owningEntityResource}/{owningEntityKey}/{entityResource}/{entityKey}** – Corresponds to a request for a detail of an owned master entity.
- **/data/{owningEntityResource}/{owningEntityKey}/{entityResource}/{entityKey}/{attributeName}** – Corresponds to a request for an attribute named as **attributeName** which belongs to a specified owned master entity.

6.4.2 Response Building

The second task of this module is to build a response from the mediator's result that is either a master entity, a collection of master entities or an attribute. In every case, the main objective lies in proper evaluation of attributes' values, for which several rules have been stated:

- If the attribute was not fetched due to lazy loading, return its URI.
- If the attribute is a literal, return its value.
- If the attribute is a reference to a master entity, then either return the entity's URI (*link*) or recursively build a response for the entity (*instance*).
- For previous two rules also applies that if the attribute is multiple, then return a list of values.

6.4.3 Cache

The REST interface supports standard cache control via HTTP cache headers ([23]) instructing clients how to cache responses. Each response is sent with an ETag (Entity Tag) standing for the hash of the response. Clients are supposed to send this ETag along with next requests and if the current version has the same ETag, it indicates the response is the same as the client's cached copy. In that case, the system responses with 304 Not Modified.

6.4.4 Errors Handling

This module is also responsible for correct interpretation of possible problems.

- If the master entity type does not exist or the entity with given key was not found, return 404 Not Found.
- If the specified filters are invalid, return 400 Bad Request.
- If any internal exception is not properly caught, return 500 Internal Server Error.

6.5 Catalog Web Service

This web service provides an API for accessing and managing the catalog, wrappers and data sources including the schemas mappings.

6.5.1 Public API

The public part of the API serves for accessing metadata from the catalog according to which the UI Client is able to build a dynamic user interface. Public API returns only information useful for clients omitting configuration data intended only for internal system needs. The `CatalogResource` (see Figure 6.8) class offers metadata via the REST API according to the following way:

- **/catalog** – Returns a list of URIs referencing to metadata details for each master entity type in the catalog in the form of the next pattern.
- **/catalog/{entityResource}** – Returns metadata about the given master entity type including all of its attributes.
- **/catalog/{entityResource}/{attributeName}** – Returns metadata about the particular attribute.

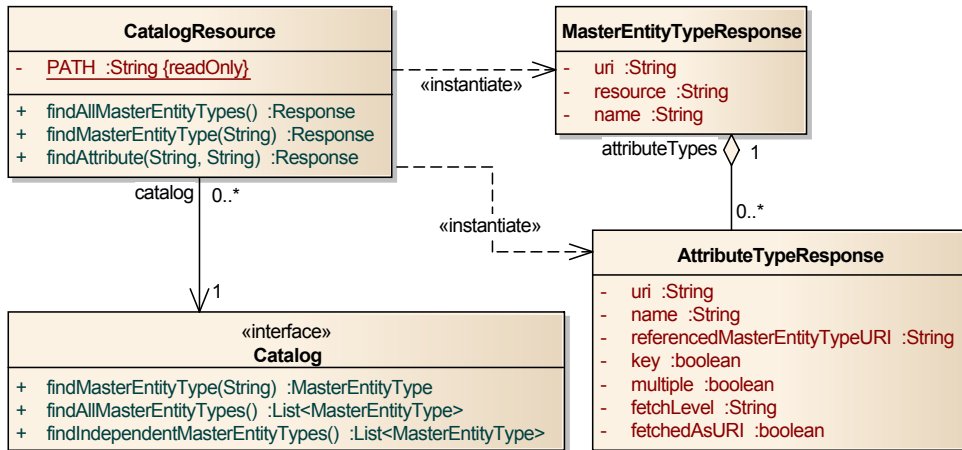


Figure 6.8: Design classes of the public part of the Catalog Web Service

6.5.2 Admin API

Admin API provides web services for management of the catalog (i.e. master entity and attribute types), wrappers and data sources. Access to these services is allowed only to authorized clients. The REST API is designed in this way:

- **/catalog/administration** – Returns a list of URIs referencing to root resources of particular administration areas in the form of the next pattern.
- **/catalog/administration/datasource** – Root resource for data sources management.
- **/catalog/administration/wrapper** – Root resource for wrappers management.
- **/catalog/administration/entitytype** – Root resource for master entity types management.
- **/catalog/administration/log** – Root resource for accessing logs.

Figure 6.9 displays the structure of design classes for wrappers management. The *WrapperService* class holding the essential business logic accesses the underlying data through

DAO classes. The obtained instances of wrappers together with their schema items are then transformed into DTO (Data Transform Object) objects standing for the resource representations. The architecture of other management areas is analogous.

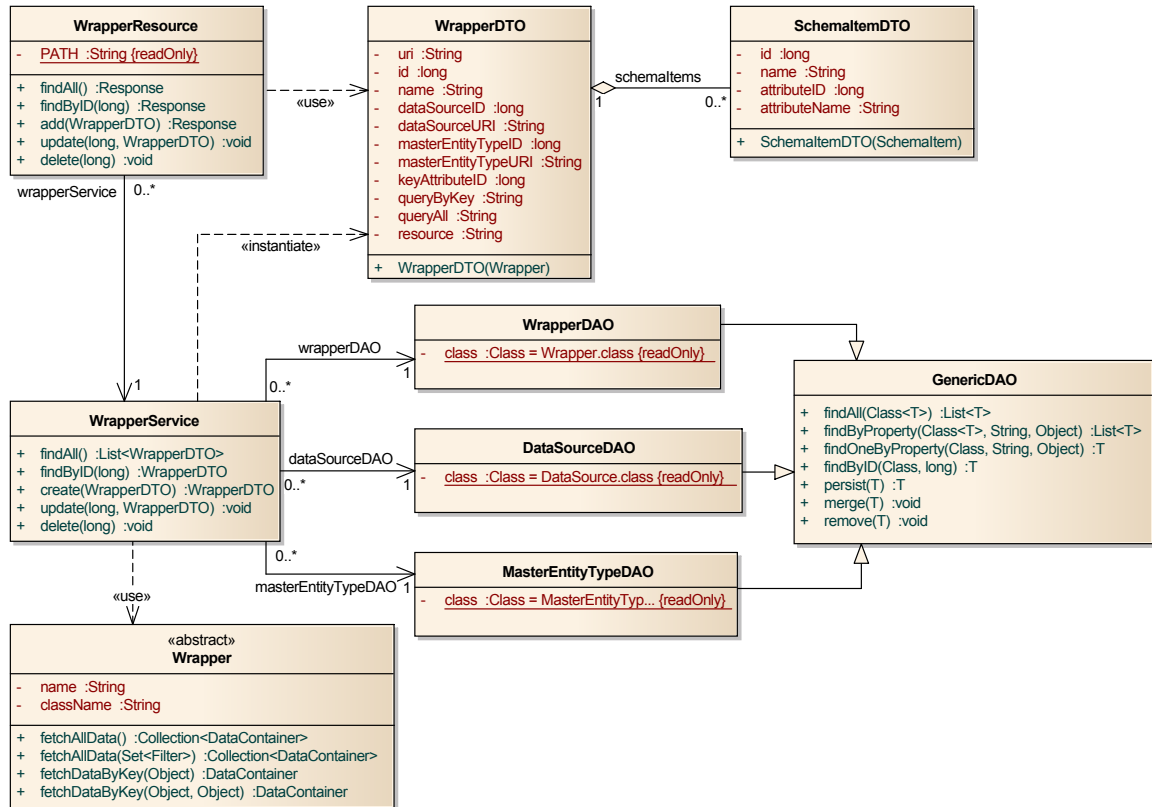


Figure 6.9: Design classes of wrappers administration

6.6 Admin Client

The Admin Client component stands for an administration UI client application communicating with the system through the Catalog Web Service. It is intended to enable the administrator to easily manage the integration system via a graphical user interface.

The client is a web application using the REST API respecting the HATEOAS principle. The application requires the user to be authorized. The structure of the UI is quite simple - a dashboard, data sources management, wrappers management and master entity types management.

6.6.1 Dashboard

The dashboard plays the role of a landing page providing useful information about the integration system. The following information should be present:

- Simple summary of numbers of integrated data sources, built wrappers and registered master entity types.
- Usage of particular data sources, wrappers and master entity types in the form of charts.
- Error log.

6.6.2 Data Sources Management

Within this page, the administrator can register a new data source, edit information about an existing one or remove it from the system. Also, there is a possibility to clear cache of a data source, i.e. clear cache of all wrappers above this data source.

For each data source, the administrator can specify:

- Name
- Type
- Cache validity (in seconds)
- Dynamic fields depending on the type (e.g. base URI for REST services)

6.6.3 Wrappers Management

This page enables to add a new wrapper, edit an existing one or remove it. Then, the administrator can clear cache of a wrapper. The dialog for creating/editing a wrapper dynamically updates available form inputs according to a type of selected data source (e.g. SQL queries for a relational database, resource identification for REST data sources etc.). An important part of the dialog is the schema mapping settings where the administrator maps wrapper's schema items to attribute types of the selected master entity type.

For each wrapper, the administrator can specify:

- Name
- Data source
- Master entity type
- Key attribute type
- Class (full name of the class with the implementation)
- Dynamic fields depending on the data source's type (e.g. SQL query by a key, SQL query for all rows for a relation database)
- List of schema mappings, each comprising of:
 - Schema item name
 - Attribute type

6.6.4 Master Entity Types Management

This is the place where the catalog is managed, which includes creating master entity types including their attribute types, editing existing ones or deleting them.

For each master entity type, the administrator can specify:

- Name
- Resource (used by the REST API)
- Owning master entity type
- List of attributes, each comprising of:
 - Name
 - Cardinality (single x multiple)
 - Fetch level (collection x entity x attribute)
 - Type (literal x reference)
 - Reference master entity type
 - Type of reference (link x instance)
 - Is key (marks the main key)
 - Is owner (only for owned entity types)

6.7 Database Design

The integration system must store the catalog, wrappers and data sources configuration in a persistent storage. For this purpose, a relational database has been chosen mainly because of the possibility of using an ORM (object-relational mapping) framework making both capturing of complicated relationships and interaction in general easy and comfortable. Also, it supports quick and well maintainable development. The database design is described by Figure 6.10.

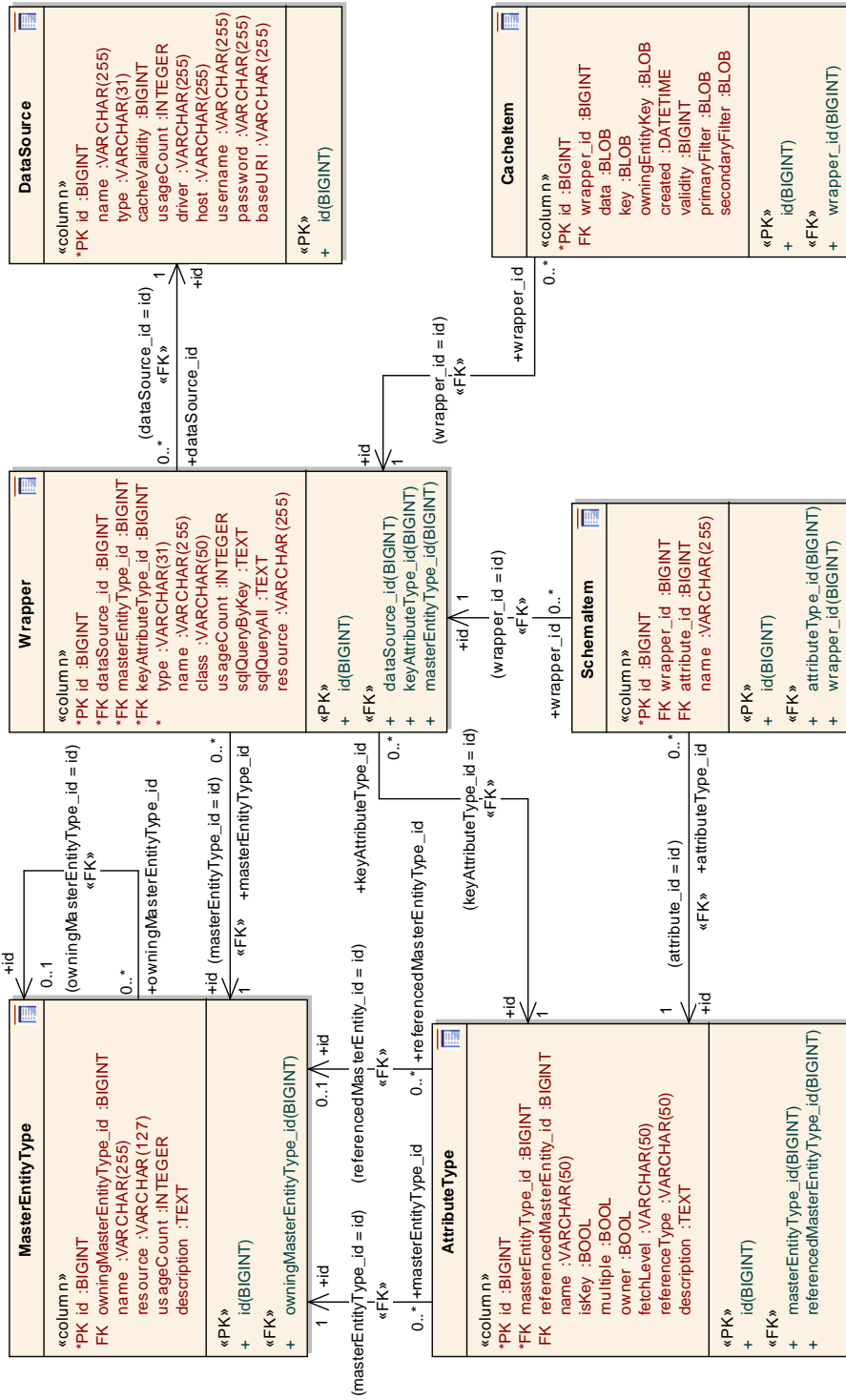


Figure 6.10: Database design

Chapter 7

Implementation

Within this chapter, the implementation part of the integration system development is described. The system has been implemented according to the design proposed in Chapter 6 and this chapter refers to several interesting challenges encountered during implementation. Also, the used platform, technologies and libraries are introduced including some basic followed principles.

7.1 Used Platform and Technologies

The integration system is implemented on the Java EE [24] platform using the Spring Framework [27]. The project's structure, dependencies and build are managed by Maven [32], a software project management and comprehension tool by Apache. Hence, the project is not dependent on a single IDE (Integrated Development Environment). The project's source code has been versioned in Git [7], a distributed version control system.

7.1.1 Spring Framework

7.1.1.1 About Spring

Spring Framework is an open source Java platform and the most popular application development framework for enterprise Java. Among others, it provides extensions for building Java EE applications that significantly facilitates software development. Thanks to its non-invasive character, Spring is easily integrable with other libraries.

Being a container itself, Spring does not require an EJB (Enterprise Java Beans) container such as an application server but a robust servlet container such as Tomcat is sufficient. The IoC (Inversion of Control) container and POJO-based (Plain Old Java Object) programming model leads to easier development and good programming practice.

7.1.1.2 Dependency Injection

The integration system uses the pattern called Dependency Injection (DI) to control dependencies between classes and thus lowering coupling. Components of the system are annotated

with the annotation `@Component` (or its more specific variants `@Repository`, `@Service` and `@Controller`) to the container as beans involving them in auto-detection in annotation-based configuration. Such annotated components can be injected to other classes managed by the container using the annotation `@Autowired`.

Code Listing 7.1 shows injection of the catalog implementation into the mediator implementation. Note, that the `ConcurrencyMediator` is dependent only on the `Catalog` interface, so a possible substitution of the implementation is very easy.

Code Listing 7.1: Dependency Injection - autowiring a catalog implementation

```

@Component
@Transactional
public class ApplicationCatalog implements Catalog {
    ...
}

@Service
public class ConcurrencyMediator implements Mediator {

    @Autowired
    private Catalog catalog;
    ...
}

```

7.1.1.3 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) brings aspects, modularization of concerns that cut across multiple classes, to the traditional Object-Oriented Programming (OOP). The *Aspect* can insert an *advice* (additional functionality) to a certain point during the execution of a program called *join point* that matches to a *pointcut* predicate. The process of linking aspects with application code is called *weaving* and can be done at compile time, load time or runtime.

The implementation of the integration system uses AOP to include objects created outside of the Spring container into DI. For example, domain objects are created either programmatically with *new* operator or by an ORM tool as a result of a database query, so the Spring container does not know about them. Annotating a class with `@Configurable`, the aspect weaving arranges that the container is aware of the class. In Code Listing 7.2, the `Wrapper` class is annotated with `@Configurable` and so it may be autowired with `CacheProvider` that is managed by the container.

7.1.2 REST

Java EE provides the JAX-RS [38] (Java API for RESTful Web Services) to support creating web services according to the REST architectural pattern. JAX-RS defines annotations for mapping a resource class, content negotiation, pulling information out of a request in method parameters and supports the HATEOAS principle.

Code Listing 7.2: Wrapper class' annotations

```

@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "type",
                    discriminatorType = DiscriminatorType.STRING)
@DiscriminatorValue(value = "default")
@Configurable
public abstract class Wrapper extends AbstractEntity<Long> {

    @Autowired
    @Transient
    protected CacheProvider cacheProvider;
    ...
}

```

The project uses the Jersey [26] library, an open source reference implementation of JAX-RS. Code Listing 7.3 shows how an HTTP request is mapped to a method for a master entity detail.

- `@GET` defines an HTTP method
- `@Path` specifies the mapping path relative to the class' base path
- `@Produces` indicates the content type of the response
- `@PathParam` maps dynamic parts of the path to particular arguments
- `@Context` return the entire context of the object

Code Listing 7.3: Mapping HTTP request on a method according to JAX-RS

```

@GET
@Path("/{entity}/{key}")
@Produces({MediaType.APPLICATION_JSON})
public Response findMasterEntityByKey(@Context Request request, @Context
    UriInfo uriInfo, @PathParam("entity") String entityResource,
    @PathParam("key") String key) {...}

```

7.1.3 ORM

The system works with a database that stores the catalog, wrappers and data sources configuration and other system data. To comfortably communicate with the database, it is recommended to use ORM, a programming technique for converting data between a relational database and an object-oriented programming language. The Java Persistence API [8] (JPA) provides a POJO persistence model for ORM. The system implementation uses the Hibernate ORM library [19] as the implementation of JPA.

Code Listing 7.2 contains the `Wrapper` class annotated as `@Entity` that also specifies the inheritance strategy for particular types of wrappers. Because the number of wrappers is not expected to be very high and their properties are generally similar, single table strategy keeping all of the types in a single table has been chosen. The particular types are distinguished by a discriminator column, which is in this case `type`. In Code Listing 7.4, there is an illustration of the discriminator value specification for the `SQLWrapper`, a default wrapper implementation for SQL data sources.

Code Listing 7.4: `SQLWrapper` class' annotations

```
@Entity
@DiscriminatorValue("sql")
public class SQLWrapper extends Wrapper {
    ...
}
```

In order to separate business logic from the data access itself, the system divides the code into service and DAO classes. A DAO class contains methods that either directly specify SQL queries or call the entity manager of Hibernate. In order to obey the DRY (don't repeat yourself) principle, the `GenericDAO` class has been implemented gathering the most frequent operations with entities as a common ancestor for all DAO classes. Code Listing 7.5 shows a sample function from the `GenericDAO` class to find entities by a property, where the usage of Java generics is worth mentioning.

Service classes call DAO classes in need of interaction with the database. All service classes are declaratively marked with the `@Transactional` annotation which means that each of their methods behaves as one transaction (see the `ApplicationCatalog` class in Code Listing 7.1). The advantage of this approach lies in communication with the database in transactions while exploiting generic methods of DAO classes. The disadvantage is the inability to catch database exceptions on the service layer.

Code Listing 7.5: Generic function to retrieve entities by a property

```
public <ENTITY> List<ENTITY> getByProperty(String property, Object value,
                                         Class<ENTITY> clazz) {
return getEntityManager()
    .createQuery("SELECT e FROM " + clazz.getSimpleName() +
                 " e WHERE e." + property + " = :value")
    .setParameter("value", value)
    .getResultList();
}
```

7.2 Multithreading

The process of mediation, i.e. delegating a query to wrappers and merging their results, is a suitable place to use multithreading. Generally, the most time-consuming part of the system is waiting for the results from data sources so parallelization should significantly increase the overall performance.

The `ConcurrencyMediator` creates a worker implementing `Callable` for each wrapper that is available for given master entity type and current fetch level. Then, it calls the `invokeAll` method on an instance of `ExecutorService` handing over a collection of the workers and specifying a maximum time to wait. The `ExecutorService` asynchronously calls all the workers and returns a list of `Future` instances. The method `get` on an instance of `Future` blocks the thread till the computation is done and retrieves its result. Calling it on all `Future` instances ensures that by the time the cycle ends, all of the threads will have completed their job (see Code Listing 7.6).

Code Listing 7.6: Asynchronous calling of workers and waiting for their results

```
for (Future<Boolean> result :
    executor.invokeAll(workers, TIME_LIMIT, TimeUnit.SECONDS)) {
    try {
        result.get();
    } catch (ExecutionException | CancellationException ex) {
        LOGGER.error(ex);
    }
}
```

A specific situation occurs when a wrapper is dependent on results of other wrappers (see Section 6.3.2). In such a case, the worker must wait till the other wrappers complete their job, i.e. the corresponding workers merge their results into a shared master entity, resp. a collection of master entities.

In Code Listing 7.7, the worker waits for other wrappers in a synchronized block using a monitor object that all workers share. When the required wrappers are done, the dependent key is loaded from the shared master entity. However, if it is not found, the worker cannot call its wrapper and is terminated. After a successful mediation process, the wrapper is marked as done and all waiting threads are notified (see Code Listing 7.8).

Code Listing 7.7: Waiting for dependent key

```
if (wrapper.usesDependentKey()) {
    try {
        synchronized (lock) {
            while (wrapper.mustWaitForPrerequisitedWrappers()) {
                lock.wait();
            }
        }
        loadDependentKey();
    } catch (InterruptedException | CannotFindDependentKeyException ex) {
        LOGGER.error(ex);
        return Boolean.FALSE;
    }
}
```

Code Listing 7.8: Marking a wrapper as done and notifying waiting threads

```

wrapper.done();
synchronized (lock) {
    lock.notifyAll();
}

```

7.3 Implementation of Wrappers

As described in Section 6.2.2, when needing to register a wrapper, the system administrator can either use a default wrapper for some types of data sources, create a subclass of a default wrapper and override only necessary parts or implement a wrapper from scratch extending the `Wrapper` abstract class.

For example, `SQLWrapper` provides a default behaviour for wrappers above a relational database. The administrator configures SQL queries and schema mappings and the wrapper works as expected - sends the SQL query to the database and processes the results translating the column labels on particular attributes according to the schema mappings (see Code Listing 7.9).

Code Listing 7.9: `SQLWrapper`'s default processing of multiple data

```

Collection<DataContainer> dataContainers = new HashSet<>();
while (results.next()) {
    DataContainer dataContainer = new DataContainer();
    for (int i = 1; i <= resultsMetaData.getColumnCount(); i++) {
        String column = resultsMetaData.getColumnLabel(i);
        Object value = results.getObject(i);
        dataContainer.addData(findMappedAttribute(column), value);
    }
    dataContainers.add(dataContainer);
}
return dataContainers;

```

When filtering is desired, the administrator only creates a subclass of `SQLWrapper` and overrides a method `buildQueryAll(Set<Filter>)`. Code Listing 7.10 shows such an overridden method, an example of building a where condition is contained in Code Listing 7.11.

Code Listing 7.10: Overridden method for building SQL query for fetching all data

```

@Override
protected String buildQueryAll(Set<Filter> filters) {
    return "SELECT id, email, name "
        + "FROM user "
        + buildWhereCondition(filters);
}

```

Code Listing 7.11: Building a where condition for filtering by e-mail

```

private String buildWhereCondition(Set<Filter> filters) {
    WhereConditionBuilder whereBuilder = new WhereConditionBuilder();
    SchemaItem schemaItem;
    for (Filter filter : filters) {
        schemaItem = findMappedSchemaItem(filter.getKey());
        if (schemaItem == null) {
            continue;
        }
        if (EMAIL.equals(schemaItem.getName())) {
            whereBuilder.andWhere(EMAIL, filter.getValue(),
                filter.getRelation());
        }
    }
    return whereBuilder.getCondition();
}

```

The essential rule when implementing a wrapper is to respect the global schema, i.e. the wrapper must strictly return data according to the attributes' properties. Besides the proper mapping on the attributes, it needs to comply the following:

- If an attribute is *multiple*, the wrapper must return a list (`List<Object>`) of values.
- If an attribute is a reference of type *link*, the wrapper must return the key of the referenced master entity.
- If an attribute is a reference of type *instance*, the wrapper must return a map collection (`Map<String, Object>`) representing the attributes of referenced master entity where the keys respects the schema of the master entity type.

When the default implementations are not enough and the administrator creates his/her own wrapper class, it is necessary to register it in the system which requires two steps:

1. Build a jar from own wrapper classes, go to a directory where the system is deployed and put the jar to WEB-INF/lib.
2. Specify the class name in the wrapper's configuration via the Admin Client.

7.4 Cache

7.4.1 Wrapper Cache

In order to reduce traffic between data sources and the system, the integration system supports caching of wrappers' results. The cached data is stored in the database in a binary form together with additional information required to identify the request (e.g. a key, an SQL query etc.). To serialize the data containers in the database and vice versa, `SerializationUtils` provided by Hibernate is used.

In order to respect cache validity, each time the cache is used its validity must be checked. Furthermore, a scheduled method was implemented to invalidate out-of-date cache items at regular time intervals (see Code Listing 7.12).

Code Listing 7.12: Scheduled method to invalidate out-of-date cache items

```
@Component
public class ClearCacheTask {

    @Autowired
    private CacheProvider cacheProvider;

    @Scheduled(fixedRate = 3600000)
    public void run() {
        cacheProvider.invalidateCache();
    }
}
```

7.4.2 HTTP Cache Control

HTTP Cache Control instructs clients how to cache responses. The `QueryingResource` counts an ETag from the mediator's result and sends it to a client within the HTTP response headers. The client is supposed to send this ETag together with next requests and if the current mediator's result has the same ETag, the client's cached copy should be used. JAX-RS provides a very comfortable way of dealing with HTTP cache control (see Code Listing 7.13).

Code Listing 7.13: HTTP cache control with ETag

```
...
CacheControl cacheControl = new CacheControl();
cacheControl.setMaxAge(CACHE_MAX_AGE);

EntityTag eTag =
    new EntityTag(Integer.toString(mediationResponse.hashCode()));

ResponseBuilder builder = request.evaluatePreconditions(eTag);
if (builder == null) {
    builder =
        Response.ok(new MasterEntityResponse(mediationResponse, uriTools));
    builder.tag(eTag);
}

builder.cacheControl(cacheControl);
return builder.build();
```

7.5 Exceptions Handling and Logging

To preserve stability of the integration system, it is necessary to properly work with exceptions and do not let a problem with one data source endanger not only the system as a whole but the particular request, as well. The basic principle is to shield the system from a worker's failure, the Code Listing 7.6 shows, among others, how each worker is checked within a try-catch block.

When an exception occurs, it should be caught and logged at the right place. A typical situation is when a data source is either unavailable or responds unexpectedly. When implementing a wrapper, it is important to take this into account and properly treat such a case. The Code Listing 7.14 shows such a treatment in the `RESTWrapper` class where a prepared response builder requests data from a data source.

Except catching exceptions, it is very useful also to log the encountered problems. The system uses Log4j library [30] by Apache for this purpose. In Log4J settings, two loggers are defined - a console log and a database log used to store error logs that are then provided by the Catalog Web Service for the Admin Client (see the settings in Code Listing 7.15).

Code Listing 7.14: Catching exceptions when contacting a data source in `RESTWrapper`

```
ClientResponse response;
try {
    response = builder.accept(accept).get(ClientResponse.class);
} catch (ClientHandlerException ex) {
    String message = "Couldn't connect to REST service at " + resourceURI;
    LOGGER.error(message);
    throw new QueryFailedException(ex, message);
}

if (response.getStatus() != 200) {
    String message = "REST service at " + resourceURI + " responded with
        HTTP status code " + response.getStatus();
    LOGGER.error(message);
    throw new QueryFailedException(message);
}
```

Code Listing 7.15: Log4j settings for logging errors in the database

```
log4j.appender.DB=org.apache.log4j.jdbc.JDBCAppender
log4j.appender.DB.URL=jdbc:mysql://localhost:3306/integration
log4j.appender.DB.driver=com.mysql.jdbc.Driver
log4j.appender.DB.user=user
log4j.appender.DB.password=password
log4j.appender.DB.sql=INSERT INTO LOGS (dated, logger, level, message)
    VALUES('%d{yyyy-MM-dd HH:mm:ss}', '%C{1}', '%p', '%m')
log4j.appender.DB.layout=org.apache.log4j.PatternLayout
log4j.appender.DB.Threshold=ERROR
```

7.6 Admin Client

The Admin Client has been implemented as a JavaScript application communicating with the REST API of the Catalog Web Service via AJAX (Asynchronous JavaScript and XML). Particular HTML pages are served by JSP (Java Server Pages) for what the security settings requires the user to be authorized by logging in via a login form.

The front-end implementation is based on jQuery¹ and Bootstrap², a popular HTML, CSS and JS framework for developing responsive web applications. Among other used libraries belong morris.js³ for charts rendering, Font Awesome⁴ as an iconic font and Data-Tables⁵ adding advanced interaction controls to tables. The UI is fully responsive enabling positive user experience on a device of nearly any display size.

#	Name	Resource	Options
1	Company	company	Edit Delete
2	Firm	firm	Edit Delete
3	Address	address	Edit Delete
4	Company Seat	seat	Edit Delete
5	Bank Account	account	Edit Delete
6	CTIA Inspection	inspection	Edit Delete
7	CTIA Sanction	sanction	Edit Delete
8	EU Grant	grant	Edit Delete

Figure 7.1: Page for master entity types management

Figure 7.1 illustrates a page for master entity types management. Pages for data sources management and wrappers management are implemented in a very similar way. The table containing all registered master entity types supports searching, pagination and sorting by any column. The *Refresh* button updates the table content. *Add Entity Type* opens a dialog

¹<https://jquery.com>

²<http://getbootstrap.com>

³<http://morrisjs.github.io/morris.js>

⁴<http://fontawesome.github.io/Font-Awesome>

⁵<https://www.datatables.net>

to define a new master entity type, *Edit* opens the same pre-filled dialog to edit an existing master entity type and *Delete* opens a dialog to confirm removing of the master entity type. The definition of the table enhancement by DataTables is shown in Code Listing 7.16.

Code Listing 7.16: Setup of master entity types table with DataTables

```
EntityController.prototype._defineTable = function () {
  this._table.dataTable({
    language: {
      search: "_INPUT_",
      searchPlaceholder: "Search...",
      emptyTable: "No entity types registered."
    },
    columns: [
      {data: "id"},
      {data: "name"},
      {data: "resource"},
      {data: "id", width: "155px"}
    ],
    columnDefs: [
      {
        targets: 3,
        render: function (data, type, row) {
          return '<button class="btn btn-primary" data-toggle="modal"
            data-target="#editEntity" data-id="' + data + '>
            <i class="fa fa-edit "></i> Edit</button>\n\
            <button class="btn btn-danger" data-toggle="modal"
            data-target="#confirmDelete" data-id="' + data + '>
            <i class="fa fa-trash"></i> Delete</button>';
        }
      }
    ]
  });
};
```

In Figure 7.2, there is a sample dialog for editing a master entity type. The master entity type is called *Company*, the REST API identifies it as a resource *company* and it is an independent entity type. The attribute type *identificationNumber* is the entity's key, a literal with a single cardinality that is always fetched. On the other hand, *addresses* has multiple cardinality and represents references to master entities of type *Address* that are fetched on the entity level (not fetched when accessing a collection). Not checking *Fetch as URI(s)* means that the reference is of the instance type, i.e. contains an embedded master entity and not just a link in the form of its URI.

Edit Entity
✕

Name*

Resource*

Description

Owned (weak) entity

Attributes

Name* ✕

Description

Cardinality

Single Multiple

Fetch Level

Type

Literal Reference

Key

Name* ✕

Description

Cardinality

Single Multiple

Fetch Level

Type

Literal Reference

Referenced Entity*

Fetch as URI(s)

Name* ✕

Description

Cardinality

Single Multiple

Fetch Level

Type

Literal Reference

Key

Figure 7.2: Dialog for editing a master entity type

Chapter 8

Testing

This chapter refers to the testing of the implemented integration system. Firstly, the usage of unit and integration tests is generally introduced. Verification of fulfillment of the requirements specified in the analysis is a next step. At the end, the system usage is demonstrated on real data including performance tests.

8.1 Unit and Integration Tests

The system has been ceaselessly tested with unit tests during development using the JUnit [29] library. However, a lot of these unit tests do not fully respect the unit testing characteristics as isolation and maximal simplicity and overgrow rather to integration tests. So, the system is covered with both unit tests focused on particular methods and more complex integration tests that deal with a cooperation between system classes and components hitting the real database.

Since the application is dependent on the Spring container for the dependency injection, the integration tests must be aware of the context, as well. The Spring Framework provides a set of annotations to use in test classes enabling to work with the Spring context. In Code Listing 8.1, there is a base test class for all context aware tests. The annotation `@RunWith` sets a Spring class to run the tests and `@ContextConfiguration` specifies a location with the context configuration. Moreover, within `@TransactionConfiguration` is configured that all transactions should be rolled back in order not to modify the database and hence, `@Transactional` makes each test method a transaction.

Code Listing 8.1: Annotations of a base test class

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {
    "/WEB-INF/context/application-context.xml"})
@TransactionConfiguration(defaultRollback = true,
    transactionManager = "txManager")
@Transactional
public abstract class ContextAwareTest {
    ...
}
```

Areas with the largest code and decision coverage involve:

- CRUD operations with master entity types and attribute types
- CRUD operations with data sources
- CRUD operations with wrappers and schema items
- Mapping between wrappers' schemas and attribute types
- Building attributes from wrappers' results
- Mediation workers with stubbed wrappers
- Properties of master entities and attributes
- Building responses from the mediation results before serialization to JSON
- Parsing and processing of filters
- Working with cache

8.2 Requirements Fulfillment

Basic testing of the integration system lies in the evaluation of requirements fulfillment specified in the analysis (see Section 4.6).

8.2.1 Functional Requirements

All functional requirements have been met, the system provides a REST API offering all desired functionalities from the *client's perspective* (1-6) and the Admin UI serves for executing all the required actions from the *administrator's perspective* (7-17).

8.2.2 Non-Functional Requirements

Regarding the non-functional requirements, the system also fulfilled all. The *integration circumstances* are satisfied because:

- The integration process is fully transparent, a client does not need to know the data sources. (18)
- The system supports a wide variety of data sources including the required ones with a potential to cover a lot more. (19)
- The catalog is in the form of a relational database. (20)
- The problem of different keys across data sources is solved by the dependent keys mechanism (see Sections 6.2.3 and 6.3.2) (21).

Most of the *demands on design* - scalability (22), support of intermediaries (23), a standard (24), explorable and self-descriptive (25) API are met thanks to the used REST architecture. The remaining one lying in easy expandability with new data sources (26) is also fulfilled. To connect to a new data source, the administrator must register it, create one or more wrappers, expand the global schema by adjusting the master entity type's properties and eventually copy the jar with implementations of wrappers to the WEB-INF/lib system's directory. None of the operations requires a change in the integration system's source code (see Sections 6.2.2 and 7.2).

As regards the requirements on *availability and stability*, testing proved them to be met.

- Stability of the system is not endangered if any data source fails, is not available or responses unexpectedly mainly thanks to the shielding of the mediation process from particular mediation workers (see Section 7.1.3). Within the testing, a data source has been turned off, changed its schema or returned an empty result. The way of a proper logging of the exception depends on the wrapper's implementation. (27)
- The maximal response time has been tested by shutting down one of the data sources and measuring time. It can be modified by changing the time limit of threads of the mediation workers. (28)
- As regards incorrect client's input, various forms of invalid URIs and the HTTP header with filters have been successfully tested. The JAX-RS mechanism manages to filter non existing URI patterns, the system validates the existence of master entity's resource and the filter parsing mechanism also dispose of a strict validation. (29)
- The system provides a three-level lazy loading mechanism enabling to specify when an attribute should be fetched (see Section 6.3.3). (30)

The *security* regarded requirements have also been subjected to tests proving that an unauthorized user does not have access neither to the Admin UI client (31) nor the secured REST API (32) intended only to administration purposes.

Performance of the system is supported both by caching of wrappers' results (33) reducing traffic between data sources (more in the following Section 8.3.4) and the system and HTTP cache control (34) instructing clients how to cache the responses and thus saving bandwidth, as well.

Finally, requirements on the *documentation* are met. An installation manual (35), a user manual (36) and an administration manual (37) are included in the appendices of the thesis. The documentation of the source code (38) in Javadoc[25] is contained in the attached DVD.

8.3 Demonstration on Real Data

The demonstration of the system usage aims to simulate a real situation the system has been developed for in the domain of joint-stock companies in the Czech Republic. Six data sources offering real data in various forms (SQL, SOAP, REST, CSV, XLS) are connected to build eight types of master entities via sixteen wrappers.

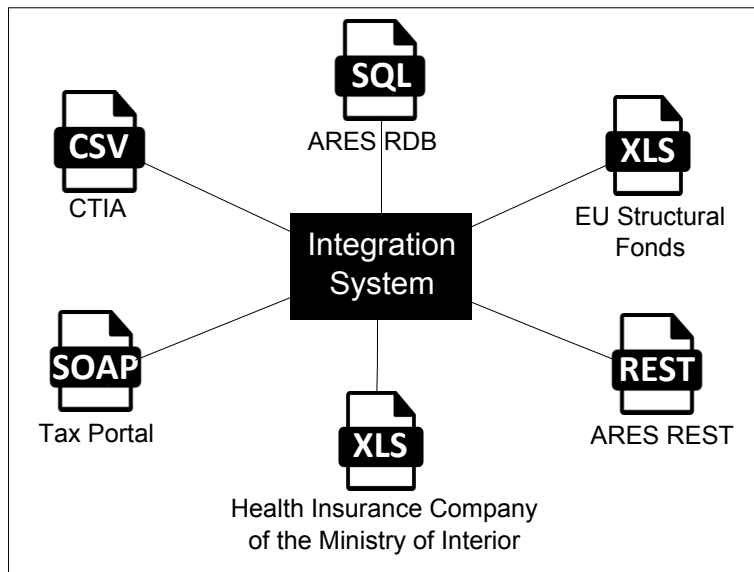


Figure 8.1: Data sources in the example of use

8.3.1 Connected Data Sources

Within the example of use, the system integrates the following data sources (see Figure 8.1 for a basic schema of the data sources and their types):

1. **ARES RDB** – The primary data source is a local relational database with all joint-stock companies in the Czech Republic created from an information system called ARES¹ (Access to Registers of Economic Subjects / Entities) managed by the Ministry of Finance of the Czech Republic. It simulates a private base data source (e.g. a database of company’s clients) to be enriched by public data.
2. **Tax Portal** – The Tax Portal² managed by the General Financial Directorate provides a SOAP web service for determining the reliability of the VAT (value-added tax) payers and their bank accounts.
3. **ARES REST** – A REST web service providing information about company’s subjects (shareholders, statutory authorities, a board of directors, a supervisory board etc.). In fact, this data source was built as a REST API above a part of the ARES RDB to simulate the problem of dependent keys and creating wrappers for a REST web service.
4. **EU Structural Fonds** – The Ministry of Regional Development provides a list of beneficiaries from all operational programs within the European structural funds³ in the form of an XLS document.

¹<http://wwwinfo.mfcr.cz/ares/ares.html.en>

²http://adisspr.mfcr.cz/adistc/adis/idpr_pub/dpr_info/ws_spdph.faces

³<http://www.strukturalni-fondy.cz/en/Informace-o-cerpani/Seznamy-prijemcu>

5. **Czech Trade Inspection Authority (CTIA)** – The Czech Trade Inspection Authority, an administrative government institution monitoring and inspecting business and individuals, publishes among others data about inspections and imposed sanctions as CSV documents⁴.
6. **Health Insurance Company of the Ministry of Interior** – A list of payers whose total dept for public health insurance and penalties in the Health Insurance Company of the Ministry of Interior is higher than 100 000 CZK as an XLS document⁵.

Except the ARES REST that identifies the companies by a surrogate key created specially for this simulation and known only to the REST RDB, all data sources identify the companies by a registration number specified in the Commercial Register.

8.3.2 Defined Master Entity Types

The key master entity type is the *Company* whose attributes are fetched from all the mentioned data sources. Other master entity types are practically parts of the *Company* that a client can further explore. A class diagram capturing the attribute types of particular master entity types and relationships between each other is to be seen in Figure 8.2.

1. **Company** – A joint-stock company located in the Czech Republic with a unique company registration number.
2. **Firm** – Represents a firm formed by a company with temporal validity (in other words it catches various names of the company during its existence).
3. **Subject** – A subject with a relationship to a company (shareholders, statutory authorities, a board of directors, a supervisory board etc.). It could be either a natural person or a legal entity (e.g. another joint-stock company).
4. **Address** – Address of a company or a subject.
5. **Bank Account** – A bank account belonging to a company.
6. **EU Grant** – A grant from European structural funds.
7. **CTIA Inspection** - An inspection executed by the CTIA.
8. **CTIA Sanction** - A sanction imposed within an inspection executed by the CTIA.

8.3.3 Implementation of Wrappers

Most wrappers are built above the ARES RDB because it contains data for three master entity types (*Company*, *Firm* and *Address*). The wrappers are basically of two types:

- a) Wrappers focused on a particular entity using its key when fetching by a key, e.g. *Firm Wrapper* fetching either all firms or a firm by a key.

⁴<http://www.coi.cz/cz/spotrebitel/open-data-database-kontrol-sankci-a-zakazu>

⁵<http://www.zpmvcr.cz/platci/dluznici/>

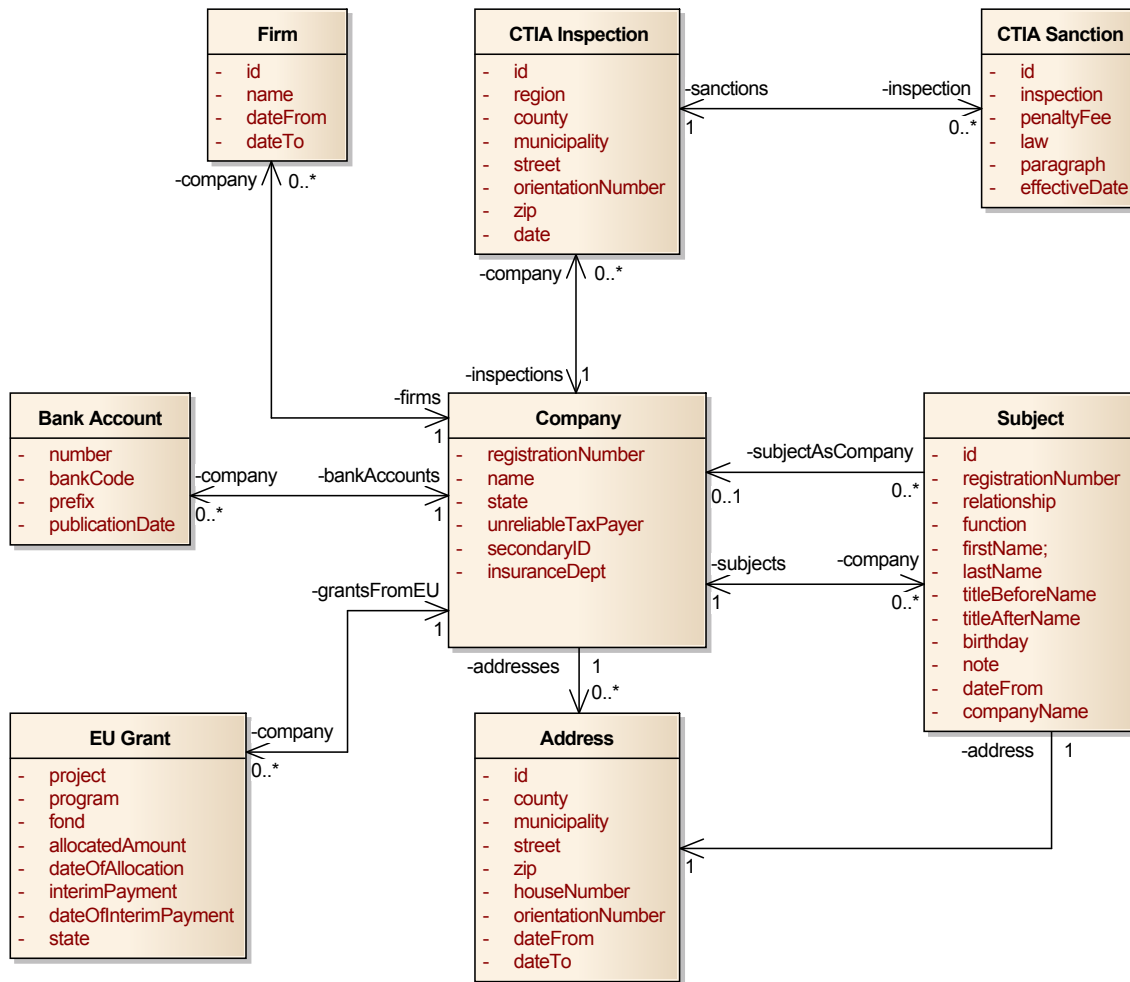


Figure 8.2: Master entity types in the example of use

- b) Wrappers fetching multiple referenced master entities of another master entity (mostly *Company*). Such wrappers do not implement fetching all data, but focus only on particular multiple references, especially of the *Company*, e.g. *Firms by Company Wrapper* returning all firms by a company's key.

All own implementations of wrappers significantly exploit functionalities provided by the default implementations, especially for SQL, REST and CSV data sources. SQL wrappers of the first type overrides only the method for filters processing (see Section 7.2). Results processing in wrappers of the second type must have been manually implemented because the multiple attributes often stand as embedded instance reference types so the mapping on the global schema is more difficult.

8.3.4 Performance Tests

The integration system with real data described above underwent performance tests to prove the influence of lazy loading and caching. However, as the analysis already emphasises, the purpose of lazy loading is not related only to the performance circumstances but supports the system availability by making the communication with data sources feasible.

The performance test cases lie in querying the integration system for a company's detail 10 times, once in sequence and once in parallel. A test client makes HTTP requests for the following URI:

`http://{baseURI}/{context}/rest/data/company/26185610`

The tests were evaluated on three various configurations of the system depending on cache and lazy loading settings. The results are available in Table 8.1, all times are in milliseconds.

1. In the first case, caching of wrappers' results is turned off and lazy loading is set to fetch all data about the company (except the data from the Tax Portal that is a public web service whose restrictions do not allow more than 4 parallel requests, which is after all a perfect example why lazy loading is a necessary part of the system). See Figure 8.3 and first and second row in Table 8.1.

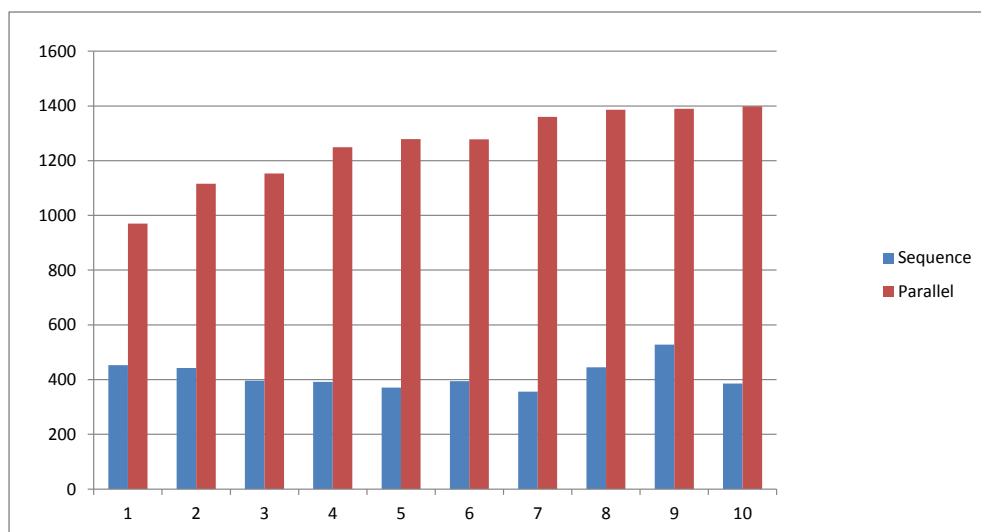


Figure 8.3: Performance tests results - lazy loading off, cache off

2. The second case queries the integration system with lazy loading set to fetch only data from ARES RDB and ARES REST while other data sources are not contacted (i.e. attributes fetched from the other data sources have the fetch level set to *attribute*). See Figure 8.4 and third and fourth row in Table 8.1.

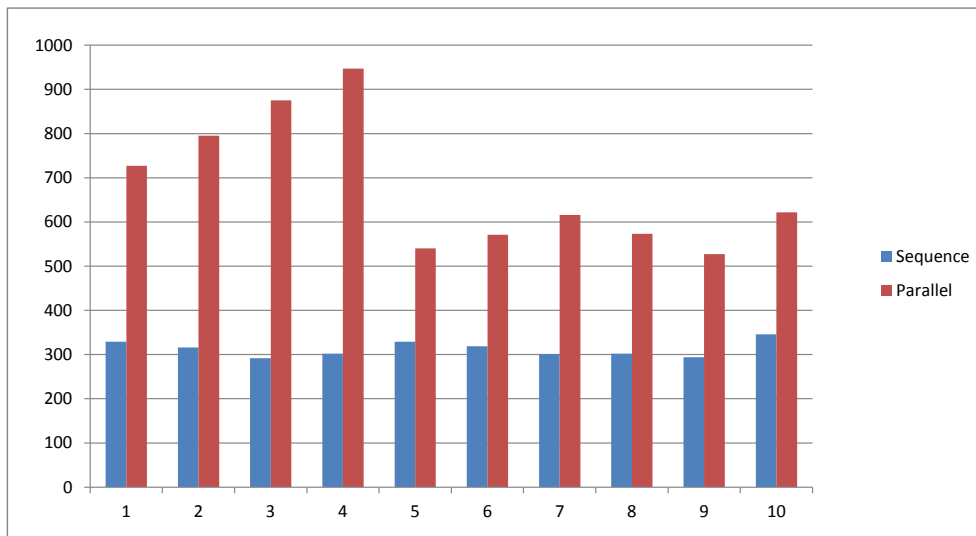


Figure 8.4: Performance tests results - lazy loading on, cache off

3. Within the third case, caching of all wrappers' results is turned on. As seen from the sequence case, the first response takes more time because the results are not cached yet, the next responses are much faster. See Figure 8.5 and fifth and sixth row in Table 8.1.

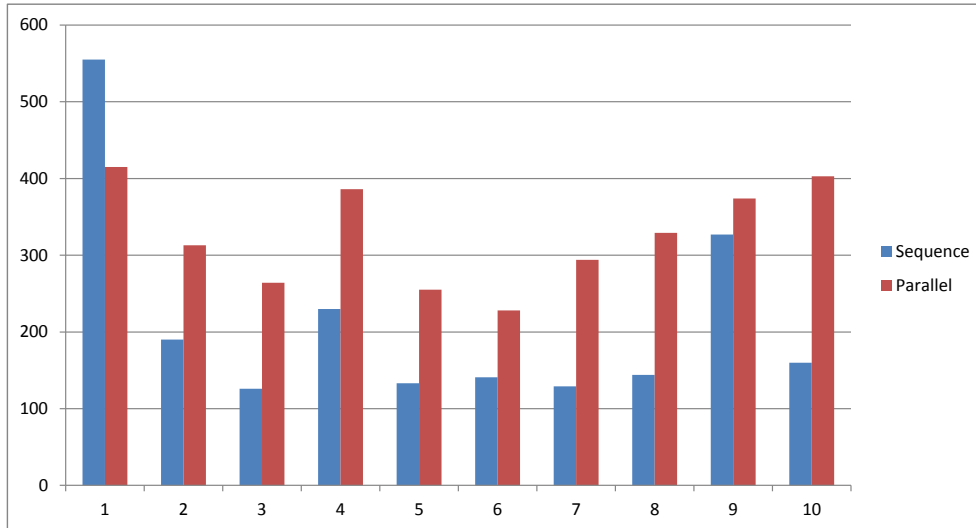


Figure 8.5: Performance tests results - lazy loading on, cache on

Table 8.1: Results of performance tests

Cache	LL ¹	Type	1	2	3	4	5	6	7	8	9	10
no	no	S ²	516	629	655	722	255	228	294	329	374	403
no	no	P ³	970	1116	1153	1249	1279	1278	1360	1386	1390	1398
no	yes	S	329	316	292	302	329	319	301	302	294	346
no	yes	P	727	795	875	947	540	571	616	573	527	622
yes	yes	S	555	190	126	230	133	141	129	144	327	160
yes	yes	P	415	313	264	386	255	228	294	329	374	403

¹ Lazy Loading

² Sequence

³ Parallel

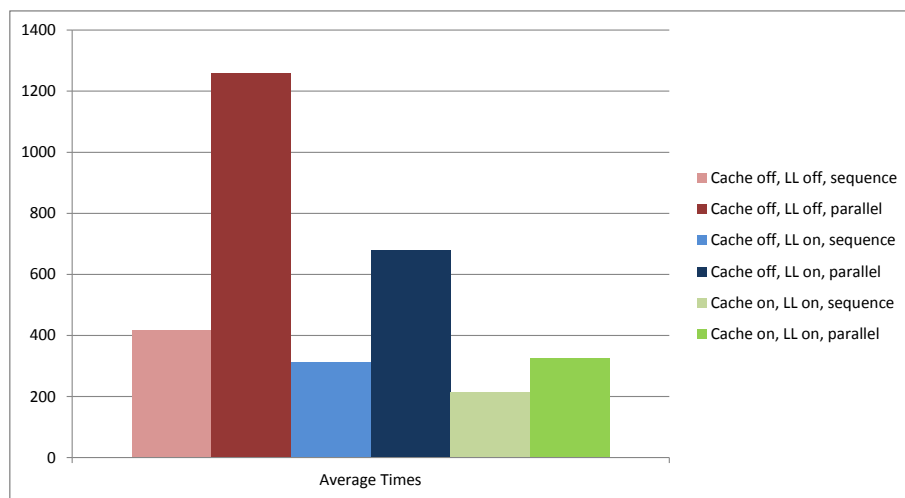


Figure 8.6: Performance tests results - average times of all situations

As expected, the performance tests prove that cache and lazy loading have an important influence on system performance. In this case, having both turned on saves about a half the time in sequence and three quarters in parallel. The more time a data source's response takes, the greater influence these instruments have. The testing also confirms lazy loading, within the meaning of requesting particular information when a client really needs it, to be worth using.

Chapter 9

Conclusion

This diploma thesis proposed an approach to the integration of heterogeneous data sources oriented on the essential business entities - master entities. As a result, a general and configurable integration system capable of integrating a wide range of information sources was developed. The system provides a single view on data from multiple data sources in a transparent way while enabling to easily include a new data source. Settings of the system can be managed via a user-friendly administration client application.

The architecture was based on a wrapper-mediator principle modified to enable easy extensibility with new data sources. The mediator stands for an independent mechanism performing the integration of data received from wrappers. Building wrappers, components querying the data sources and processing the obtained data that respect the common interface, represents the way of easy extending the integration system with information sources. To describe the integrated global schema, a catalog keeping metadata comprising of master entity and attribute types was designed.

Respecting all the necessary REST constraints, the architecture can be considered as RESTful. As a consequence, the integration system supports scalability, transparent security and cacheability and clients query the integrated data via a uniform, easy-to-use and self-descriptive interface based on standard technologies. The resource oriented API is designated by the contents of the catalog. Therefore, the system is very adaptive requiring no changes in the existing source code after modification of the global schema.

Several interesting problems identified in the analysis had to be solved. Firstly, a mechanism to overcome an obstacle in the form of different identifiers across data sources was designed lying in building a structure of dependent wrappers the mediator complies when forming the subqueries. By requiring to support a wide range of data sources, the integration system had to take into account access restrictions to make the querying of data sources not only more effective but sometimes even feasible. That is the reason why a lazy loading technique was developed enabling to set when particular data should be fetched and when it is, on the contrary, not convenient. The communication of the system in both directions, i.e. with data sources and clients, was supported by caching.

To demonstrate the benefits of the integration system, an example using the real data was provided. Within this sample, a relational database of all joint-stock companies in the Czech Republic originating from the Ministry of Finance was linked to various heterogeneous

data sources providing data about the reliability of the VAT payers and their bank accounts, EU structural funds beneficiaries, the inspections and sanctions made by the Czech Trade Inspection Authority or the debtors of the Health Insurance Company of the Ministry of Interior. So, all the interesting information is thanks to the integration system accessible from one REST API in a completely transparent way.

9.1 Future Work

At present, a dynamic client for the integration system developed within this thesis begins to emerge as a subject of another diploma thesis. Therefore, support and possible upgrades of the system become one of the needed future works. Among the expected requirements, which will arise from the related project's analysis, belongs an extension of the metadata provided by the catalog to support data visualization.

Meanwhile, areas to enhance include improvements of default wrappers' implementations, automatic schema mappings for embedded master entities and more detailed statistics to show on the admin dashboard.

Bibliography

- [1] S. ABITEBOUL, D. QUASS, J. MCHUGH, J. WIDOM, and J. WIENER. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1:68–88, 1997.
- [2] O. BEN-KIKI, C. EVANS, and B. INGERSON. YAML Ain't Markup Language (YAML™) Version 1.1. W3C, <http://yaml.org/spec/1.1/current.pdf>, 2005.
- [3] A. BROWN, S. JOHNSTON, and K. KELLY. *Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications*. Rational Software Corporation, 2002.
- [4] S. BUSSE, R.-D. KUTSCHE, U. LESSER, and H. WEBER. Federated Information Systems: Concepts, Terminology and Architectures. *ACM Computing Surveys (CSUR) - Special issue on heterogeneous databases*, 22(3):183–236, 1999.
- [5] D. BUTLER. *Master Data Management*. Oracle Corporation, 2011.
- [6] J. CARDOSO and A. SHETH. *Semantic Web services, processes and applications*. NY: Springer, New York, 2006.
- [7] S. CHACON and B. STRAUB. *Pro Git*. Apress, 2014.
- [8] L. DE-MICHIEL. JSR 317: Java Persistence API, Version 2.0. Version 2.0, Final Release. Sun Microsystems, <https://jcp.org/aboutJava/communityprocess/final/jsr317>, 2009.
- [9] R. T. FIELDING. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.
- [10] H. GARCIA-MOLINA, J. HAMMER, K. IRELAND, Y. PAPAKONSTANTINOY, J. ULLMAN, and J. WIDOM. Integrating and Accessing Heterogeneous Information Sources in TSIMMIS. In *Proceedings of the AAAI Symposium on Information Gathering*, pages 61–64, 1995.
- [11] H. GARCIA-MOLINA, Y. PAPAKONSTANTINOY, D. QUASS, A. RAJARAMAN, Y. SAGIV, J. ULLMAN, V. VASSALOS, and J. WIDOM. The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems*, pages 117–132, 1997.

- [12] A. GERACKI, F. KATKI, L. MCMONEGAL, B. MEYER, J. LANE, P. WILSON, J. RADATZ, M. YEE, H. PORTEOUS, and F. SPRINGSTEEL. *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*. IEEE, 1991.
- [13] R. GHAWI. *Ontology-based cooperation of information systems contributions to database-to-ontology mapping and XML-to-ontology mapping*. Doctoral dissertation, Universit'e de Bourgogne, 2010.
- [14] R. GHAWI, T. POULAIN, G. GOMEZ, and N. CULLOT. OWSCIS: Ontology and Web Service Based Cooperation of Information Sources. *Signal-Image Technologies and Internet-Based System, 2007. SITIS '07. Third International IEEE Conference on*, pages 246–253, 2007.
- [15] C. H. GOH. *Representing and Reasoning about Semantic Conflicts in Heterogeneous Information Systems*. Doctoral dissertation, MIT, 1997.
- [16] T. R. GRUBER. A translation approach to portable ontology specifications. *Knowledge Acquisition - Special issue: Current issues in knowledge modeling*, 5(3):573–582, 1993.
- [17] T. R. GRUBER. Toward Principles for the Design of Ontologies Used for Knowledge Sharing. *International Journal of Human-Computer Studies - Special issue: the role of formal ontology in the information technology*, 43(5-6):907–928, 1995.
- [18] T. HEATH and C. BIZER. *Linked Data: Evolving the Web into a Global Data Space*. Synthesis Lectures on the Semantic Web: Theory and Technology, 1:1, 1-136. Morgan & Claypool, 1st edition, 2011.
- [19] Hibernate. Red Hat, <http://hibernate.org>, 2015.
- [20] H. H. HOANG, A. ANDJOMSHOAA, and A. M. TJOA. Towards a New Approach for Information Retrieval in the SemanticLIFE Digital Memory Framework. *2006 IEEE/WIC/ACM International Conference on Web Intelligence*, 2006.
- [21] H. H. HOANG and T. M. NGUYEN. Ontology-based Virtual Query System for the SemanticLIFE Digital Memory Project: Concepts, Designs and Implementation. *World Academy of Science: Engineering and Technology*, 2006.
- [22] H. H. HOANG and A. M. TJOA. The Virtual Query Language for Information Retrieval in the SemanticLIFE Framework. *Web Information Systems Modeling*, 2006.
- [23] Hypertext Transfer Protocol – HTTP/1.1. Network Working Group, <http://www.w3.org/Protocols/rfc2616/rfc2616.html>, 1999.
- [24] JSR 342: Java Platform, Enterprise Edition 7 (Java EE 7) Specification. Oracle, <https://jcp.org/en/jsr/detail?id=342>, 2011 (updated in 2013).
- [25] Javadoc 5.0 Tool. Oracle, <http://docs.oracle.com/javase/1.5.0/docs/guide/javadoc/index.html>, 2010.
- [26] Jersey. Oracle, <https://jersey.java.net>, 2015.

- [27] R. JOHNSON, J. HOELLER, K. DONALD, C. SAMPALLEANU, R. HARROP, and T. RISBERG. *Spring Framework Reference Documentation*. Pivotal Software, 2011.
- [28] JSON-RPC 2.0 Specification. JSON-RPC Working Group, <http://www.jsonrpc.org/specification>, 2010 (updated in 2013).
- [29] JUnit. JUnit Team, <http://junit.org>, 2014.
- [30] Apache Log4j 2. The Apache Software Foundation, <http://logging.apache.org/log4j/2.x>, 2015.
- [31] D. LOSHIN. *Master data management*. Amsterdam: Elsevier/Morgan Kaufmann, 2009.
- [32] Apache Maven. The Apache Software Foundation, <https://maven.apache.org>, 2014.
- [33] D. MCLEOD and D. HEIMBIGNER. A federated architecture for database systems. *National Computer Conference, 1980*, 1980.
- [34] E. MUGELLINI, P. S. SZCZEPANIAK, M. CHIARA PETTENATI, and M. SOKHN. 7th Atlantic Web Intelligence Conference, AWIC 2011, Fribourg, Switzerland, January 26-28, 2011. In *Advances in Intelligent Web Mastering - 3*. Springer-Verlag Berlin Heidelberg, 2011.
- [35] NATO Intelligence, Surveillance, and Reconnaissance (ISR) Interoperability Architecture (NIIA): VOLUME 1: Architecture Description, 2005.
- [36] OWL Web Ontology Language Overview. W3C, <http://www.w3.org/TR/owl-features/>, 2004.
- [37] Y. PAPAKONSTANTINOY, H. GARCIA-MOLINA, and J. WIDOM. Object exchange across heterogeneous information sources. In *Proceedings of the Eleventh International Conference on Data Engineering*. IEEE Comput. Soc. Press, 1995.
- [38] S. PEERICAS-GEERTSEN and M. POTOCIAR. JSR-000339 The Java API for RESTful Web Services. Oracle, <https://www.jcp.org/aboutJava/communityprocess/final/jsr339/index.html>, 2013.
- [39] C. RAY. *Distributed database systems*. New Delhi: Dorling Kindersley, 2009.
- [40] RDF 1.1 Primer. W3C Working Group. W3C, <http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624>, 2014.
- [41] RDF Schema 1.1. W3C, <http://www.w3.org/TR/rdf-schema/>, 2014.
- [42] L. RICHARDSON and S. RUBY. *RESTful Web Services*. O'Reilly Media, Inc, 2008.
- [43] P. RUSSOM. *Next Generation Master Data Management*. The Data Warehousing Institute, 2012.
- [44] A. SHETH. *Changing Focus on Interoperability in Information Systems: from System, Syntax, Structure to Semantic*. Norwell: Kluwer Academic Publishers, 1999.

- [45] A. SHETH and J. LARSON. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3), 1990.
- [46] SOAP Version 1.2 Part 0: Primer (Second Edition). W3C, <http://www.w3.org/TR/soap/>, 2007.
- [47] M. SOKHN. *Ontology driven framework for multimedia information retrieval in P2P network*. Doctoral dissertation, T'el'ecom ParisTech, 2011.
- [48] M. SOKHN, E. MUGELLINI, O. A. KHALED, and A. SERHROUCHNI. Conference knowledge modeling for conference-video-recordings querying. *In Proceedings of the International Conference on Management of Emergent Digital EcoSystems*, 2009.
- [49] SPARQL Query Language for RDF. W3C Recommendation. W3C, <http://www.w3.org/TR/rdf-sparql-query/>, 2008.
- [50] Information Technology - Database Language SQL. Second Informal Review Draft. ISO/IEC 9075:1992, Database Language SQL. Digital Equipment Corporation. Maynard, Massachusetts, <http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt>, 1992.
- [51] UDDI Spec Technical Committee Draft. Oasis, http://www.uddi.org/pubs/uddi_v3.htm, 2004.
- [52] U. VISSER, H. STUCKENSCHMIDT, H. WACHE, and T. VÖGELE. Enabling Technologies for Interoperability. *14th International Symposium of Computer Science for Environmental Protection, Bonn, Germany*, pages 35–46, 2000.
- [53] W3C. W3C Semantic Web Activity. <http://www.w3.org/2001/sw>, 2013. [cited on 19/11/2014].
- [54] H. WACHE, T. VÖGELE, U. VISSER, H. STUCKENSCHMIDT, G. SCHUSTER, H. NEUMANN, and S. HUBNER. Ontology-Based Integration of Information — A Survey of Existing Approaches. *IJCAI-01 Workshop: Ontologies and Information Sharing*, pages 108–117, 2001.
- [55] Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. W3C, <http://www.w3.org/TR/wsd120>, 2007.
- [56] Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C, <http://www.w3.org/TR/xml>, 2008.
- [57] XQuery 1.0: An XML Query Language (Second Edition). W3C, <http://www.w3.org/TR/xquery>, 2010.

Appendix A

List of Abbreviations

AJAX Asynchronous JavaScript and XML

AOP Aspect-Oriented Programming

API Application Programming Interface

CALIMERA Conference Advanced Level Information Management & Retrieval

CRUD Create Retrieve Update Delete

CSS Cascading Style Sheets

CTIA Czech Trade Inspection Authority

DAO Data Access Object

DI Dependency Injection

DRY Don't Repeat Yourself

DTO Data Transfer Object

EJB Enterprise Java Beans

ETag Entity Tag

GUI Graphical User Interface

HATEOAS Hypermedia As The Engine Of Application State

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

IDE Integrated Development Environment

IoC Inversion of Control

Java EE Java Enterprise Edition

JAX-RS	Java API for RESTful Web Services
JPA	Java Persistence API
JS	JavaScript
JSON	JavaScript Object Notation
LL	Lazy Loading
MDM	Master Data Management
OEM	Object Exchange Model
OOP	Object-Oriented Programming
ORM	Object-Relational Mapping
OWL	Web Ontology Language
OWSCIS	Ontology and Web Service based Cooperation of Information Sources
POJO	Plain Old Java Object
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
REST	Representational State Transfer
SOAP	Simple Object Access Protocol
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
TSIMMIS	The Stanford-IBM Manager of Multiple Information Sources
UDDI	Universal Description Discovery and Integration
UI	User Interface
URI	Uniform Resource Identifier
VAT	Value-Added Tax
VQE	Virtual Query Engine
VQL	Virtual Query Language
VQS	Virtual Query System
WSDL	Web Services Description Language
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

Appendix B

Installation Manual

The installation manual consists of two parts. The first part instructs how to install the system itself on an application server or a servlet container. The second one describes a process of putting the example of the demonstration on real data into operation.

B.1 System Installation

This section refers to the installation of the integration system itself including the database preparation and deployment on a server.

B.1.1 Database Preparation

The integration system uses a local relational database to store the data sources, wrappers, master entity types, cache and logs.

1. Create a database in your RDBMS (relational database management system).
2. Specify the driver's class name (e.g. `com.mysql.jdbc.Driver` for MySQL), the URL of the database (e.g. `jdbc:mysql://localhost:3306/databaseName`), the username and the password in `WEB-INF/properties/jdbc.properties` and `log4j.properties`.
3. If using a different RDBMS than MySQL, update the property `jpa.platform` in `WEB-INF/properties/jpa.properties` and add a dependency to a Java implementation of the connector for your RDBMS in `pom.xml`.

B.1.2 Deployment

The system is based on the Spring Framework being a container itself, so no EJB container as the application server is necessary but a robust servlet container itself such as Tomcat is sufficient.

Since the structure, dependencies and build process are managed by Apache Maven, building of the system is very simple.

1. Go to the system root directory and run `mvn clean install package` in the command line.
2. Deploy `target/integration-1.0.war` to your server. Alternatively, if a Maven plugin supports deployment to your server directly you can enhance the `mvn` command.

To include any jars with implementations of wrappers, copy them to the `WEB-INF/libs` directory within the deployed war.

B.2 Example on Real Data

To prepare the example of the system demonstration on real data, you must install a local ARES RDB, deploy the ARES REST application, copy a jar with wrappers to the system and import a database with the system configuration. Prepare the files `ares.sql`, `subject-1.0.war`, `wrappers.jar` and `integration-ares.sql`, all located on the attached DVD.

B.2.1 ARES RDB

To prepare the ARES RDB, install the PostgreSQL RDBMS in version at least 9.4 together with the pgAdmin.

1. Open the pgAdmin and create a database called *ares*.
2. Click on Plugins/PSQL Console in the top menu bar.
3. Type `\i /path/to/ares.sql` where you replace `/path/to` by the proper path to the `ares.sql` containing the PostgreSQL database located on the attached DVD.
4. Press Enter.

B.2.2 ARES REST

Deploy the `subject-1.0.war` located on the attached DVD to your server.

B.2.3 System Configuration

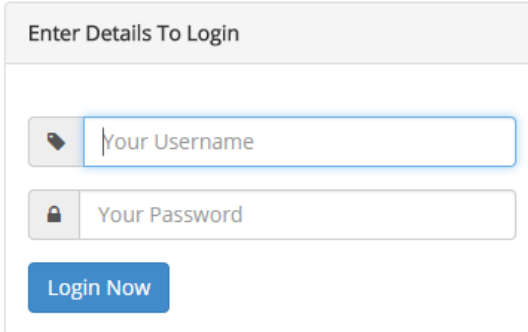
1. Copy the `wrappers.jar` to the `WEB-INF/libs` directory within the deployed war.
2. Import the `integration-ares.sql` to the system database created in the step B.1.1.
3. Restart the application.
4. Open the Admin UI, log¹ in as *admin* using the password *integration* and go to the data sources management. Check that the host of the ARES RDB is correct as well as the base URI in the ARES REST.

¹To change the credentials, modify `WEB-INF/context/spring-security.xml`

Appendix C

Administration Manual

Admin Client : Login
(Login yourself to get access to the administration)



Enter Details To Login

Your Username

Your Password

Login Now

Figure C.1: Login page

C.1 Login

Before the system can be used for offering data to clients, it must be properly configured. Firstly, the administrator has to log in to the Admin Client application with correct credentials on a page shown in Figure C.1.

C.2 Dashboard

The landing page provides a simple dashboard (see Figure C.2). On top, the numbers of integrated data sources, built wrappers and used master entity types are displayed. Below, there are charts showing usage of particular data sources, wrappers and master entity types.

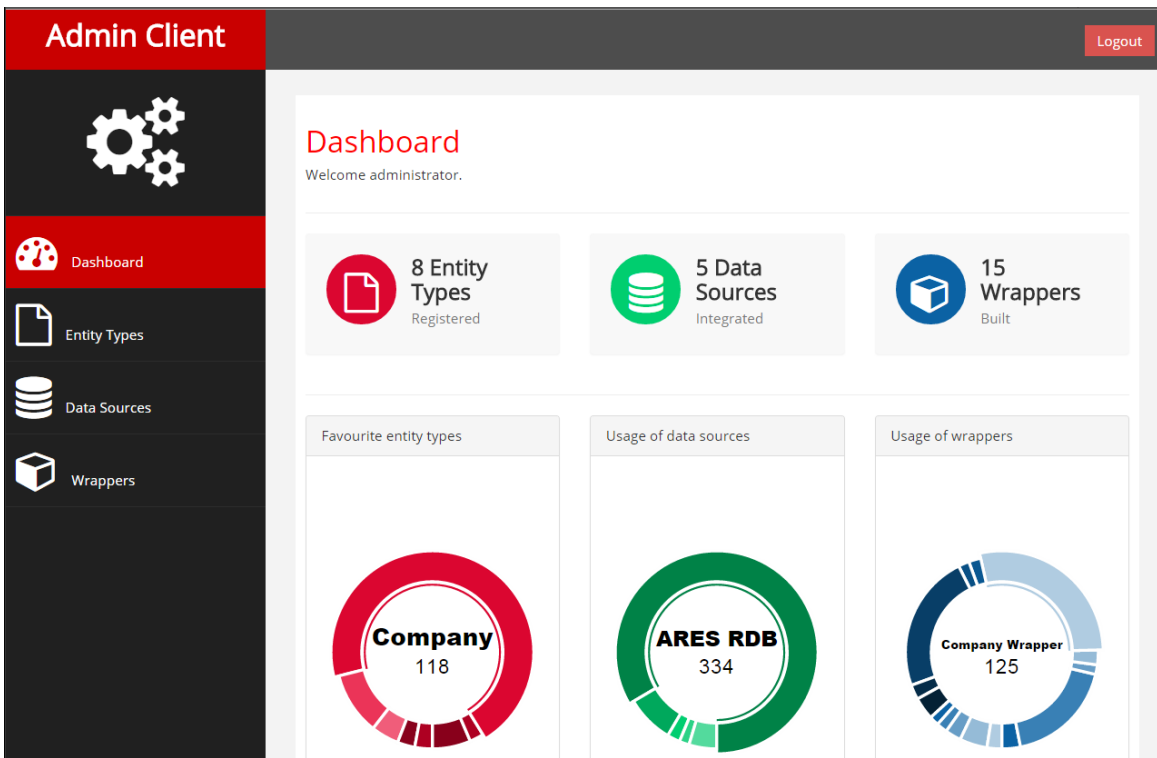


Figure C.2: Dashboard

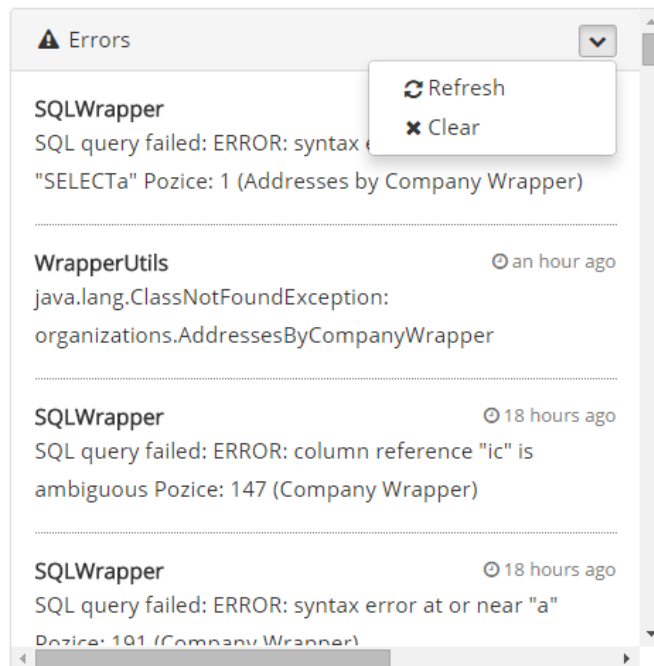


Figure C.3: Error log

Finally, bottom of the page includes an error log that can be refreshed or cleared (see Fig C.3). The dashboard can be enhanced with more interesting information in future.

C.3 Entity Types

C.3.1 Overview

By selecting *Entity Types* from the left menu, you come to a master entity types management page showing all registered master entity types (see Figure C.4). There, you can add a new master entity type, edit an existing one or remove it from the system. This is the place where the system global schema is specified.

The screenshot shows the 'Admin Client' interface. The main content area is titled 'Entity Types' and contains a table of registered master entity types. The table has columns for '#', 'Name', 'Resource', and 'Options'. The 'Options' column contains 'Edit' and 'Delete' buttons for each row. The table lists 8 entries: Company, Firm, Address, Company Seat, Bank Account, CTIA Inspection, CTIA Sanction, and EU Grant. The page also includes a search bar, a refresh button, and a pagination control showing 'Showing 1 to 8 of 8 entries'.

#	Name	Resource	Options
1	Company	company	Edit Delete
2	Firm	firm	Edit Delete
3	Address	address	Edit Delete
4	Company Seat	seat	Edit Delete
5	Bank Account	account	Edit Delete
6	CTIA Inspection	inspection	Edit Delete
7	CTIA Sanction	sanction	Edit Delete
8	EU Grant	grant	Edit Delete

Figure C.4: Master entity types management

C.3.2 Add Master Entity Type

To add a new master entity type, click on *Add Entity Type* button and a dialog will appear (see Figure C.5). Fill in the master entity type's name and resource (an identifier for the REST interface). Optionally, provide a description to better explain the circumstances to the clients using the Catalog Web Service for getting the metadata. If you want the entity type to be owned, check the *Owned (weak) entity* and select the owning master entity type.

The essential task is to properly specify the attribute types (see Figure C.6). For each attribute, fill in its name, cardinality, fetch level and type and eligibly provide a description.

The 'Add Entity' dialog box includes the following elements:

- Name***: A text input field with a small icon to its right.
- Resource***: A text input field.
- Description**: A larger text area for detailed notes.
- Owned (weak) entity**: A checkbox option.
- Attributes**: A section header above an **Add Attribute** button.
- Close** and **Save**: Action buttons at the bottom right.

Figure C.5: Add a master entity type dialog

Attributes

The attribute specification panels include the following configuration options:

- Name***: Text input field.
- Description**: Text area.
- Cardinality**: Radio buttons for **Single** (selected) and **Multiple**.
- Fetch Level**: Dropdown menu with **Collection** (left) and **Entity** (right).
- Type**: Radio buttons for **Literal** and **Reference** (selected).
- Referenced Entity***: Dropdown menu with **Address** selected.
- Fetch as URI(s)**: Checkbox option.
- Key**: Checkbox option (present only in the right panel).

Add Attribute button is located below the panels.

Figure C.6: Specification of attribute types

If the type is *Literal*, you can mark this attribute as a key. On the other hand, for the *Reference* type specify the reference master entity type and whether it should be fetched as a URI or as a nested instance. If the master entity type has been marked as owned, you can then specify an attribute of a *Reference* type to be the owner. In any case, note that:

- *Multiple* attributes contain a list of values (a list of literals, URIs or nested instances)
- Fetch level determines when the attribute should be fetched, i.e. when the wrappers mapped to the attribute should be contacted.
 - *Collection* – the attribute is fetched always (suitable for general data that can be fetched collectively)
 - *Detail* – the attribute is fetched when accessing the master entity’s detail or the attribute directly but not when accessing the collection of master entities
 - *Attribute* – the attribute is fetched only when it is accessed directly (suitable for attributes whose retrieval is demanding or somehow restricted, e.g. web services by a key)

C.3.3 Edit Master Entity Type

For editing a master entity type and its attribute types, click on *Edit* button in the table. Editing is done via an analogous dialog as when adding.

C.3.4 Remove Master Entity Type

For removing a master entity type from the system, click on *Delete* button in the table. Removing a master entity type is possible only when no registered wrapper is using it.

C.4 Data Sources

C.4.1 Overview

By selecting *Data Sources* from the left menu, you come to a data sources management page showing all registered data sources (see Figure C.7). There, you can add a new data source, edit an existing one or remove it from the system. Also, you can manually clear cache of a data source, i.e. clear cache of each wrapper standing above the data source.

C.4.2 Add Data Source

To add a new data source, click on *Add Data Source* button and a dialog will appear (see Figure C.8). Select a data source’s type (SQL Database, CSV document, SOAP service etc.) and fill in its name. If the desired type is available, select *Other*. Optionally, you can specify cache validity (number of seconds to store data from this data source in cache). If the field is empty, the system will not cache data from this data source at all. Based on the selected type, some other fields can be shown. For example, if you desire to exploit the

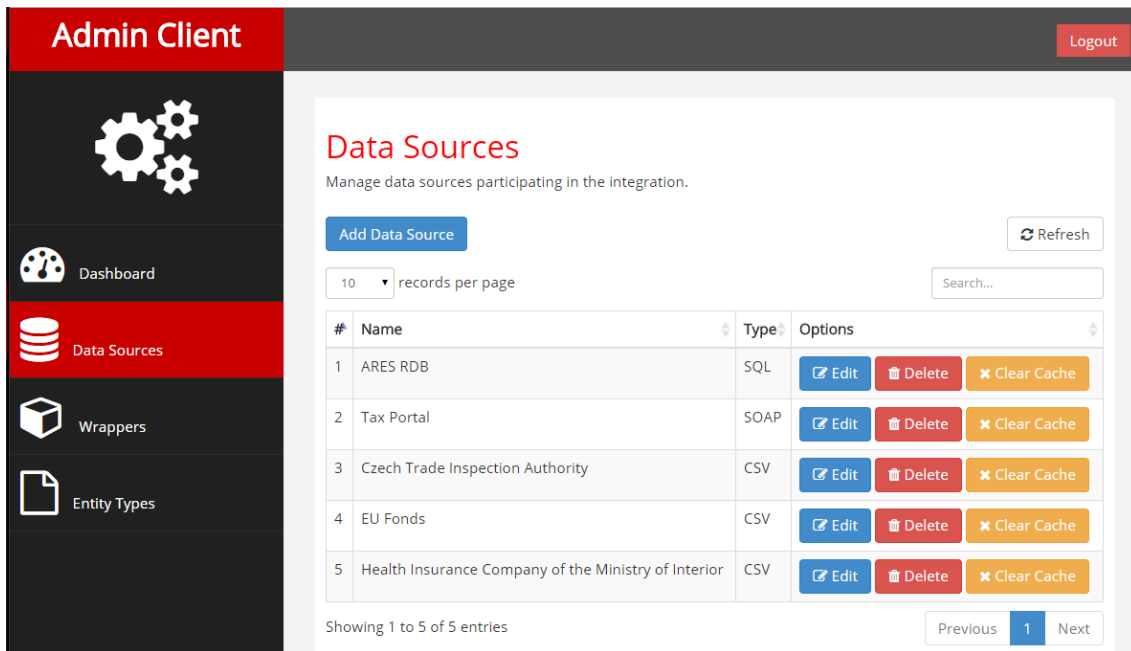


Figure C.7: Data sources management

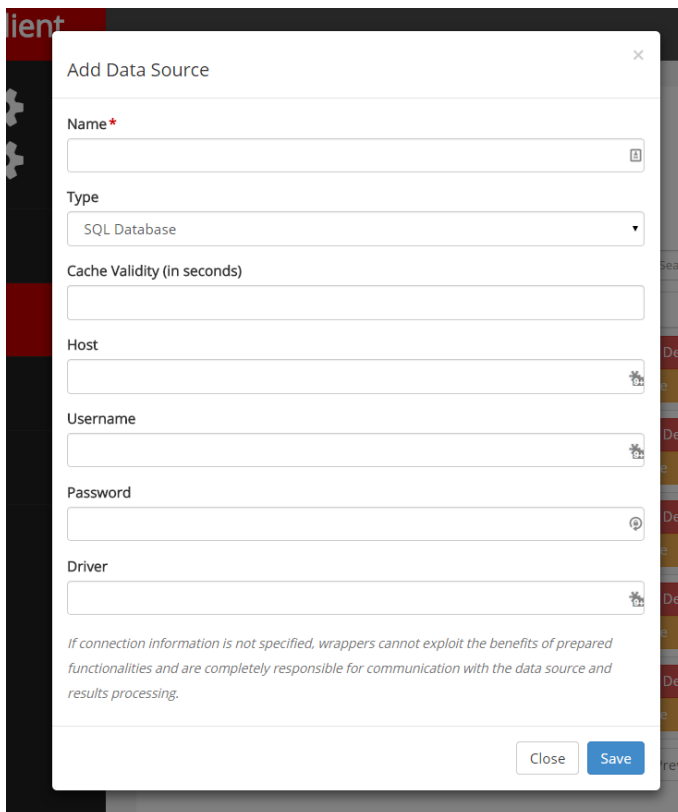


Figure C.8: Add a data source dialog

benefits of default behaviour for wrappers above a relational database, you should specify the host (e.g. jdbc:postgresql://192.168.2.1/database), a username, a password and a full class name of a driver (e.g. org.postgresql.Driver).

C.4.3 Edit Data Source

For editing a data source, click on *Edit* button in the table. Editing is done via an analogous dialog as when adding.

C.4.4 Remove Data Source

For removing a data source from the system, click on *Delete* button in the table. Removing a data source is possible only when no registered wrapper is using it.

C.4.5 Clear Cache

To manually clear data source's cache, click on *Clear Cache* button in the table and cache of all wrappers standing above the data source will be invalidated.

C.5 Wrappers

C.5.1 Overview

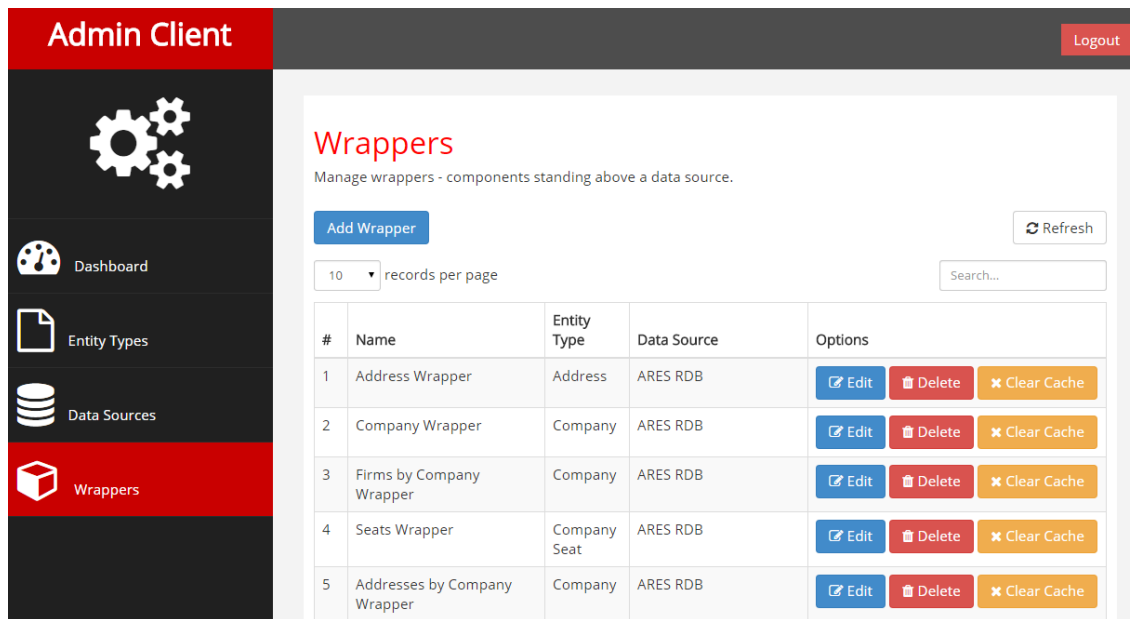
By selecting *Wrappers* from the left menu, you come to a wrappers management page showing all registered wrappers (see Figure C.9). There, you can add a new wrapper, edit an existing one or remove it from the system. Also, you can manually clear cache of a wrapper.

C.5.2 Add Wrapper

To add a new wrapper, click on *Add Wrapper* button and a dialog will appear. Fill in its name and choose a data source the wrapper should work with. According to a type of the selected data source, the dialog may be extended by some new fields, e.g. a specification of SQL queries for a relational database (see Figure C.10). Choose a master entity type whose attributes are to be fetched by the wrapper. Then, select an attribute type used as a key when fetching from the underlying data source. If you have created a class with wrapper's implementation, specify its full name. Finally, provide mappings between the wrapper's schema and the attributes by typing a schema item's name and selecting a corresponding attribute type for each desired schema item.

C.5.3 Wrapper Implementation

In a situation when the default implementation is not enough for your purpose or is not available for the type of the selected data source, you must write a Java class with your implementation. Such a class must inherit from the `Wrapper` class or one of its descendants (e.g. `SQLWrapper`, `CSVWrapper`).



Admin Client Logout

Wrappers

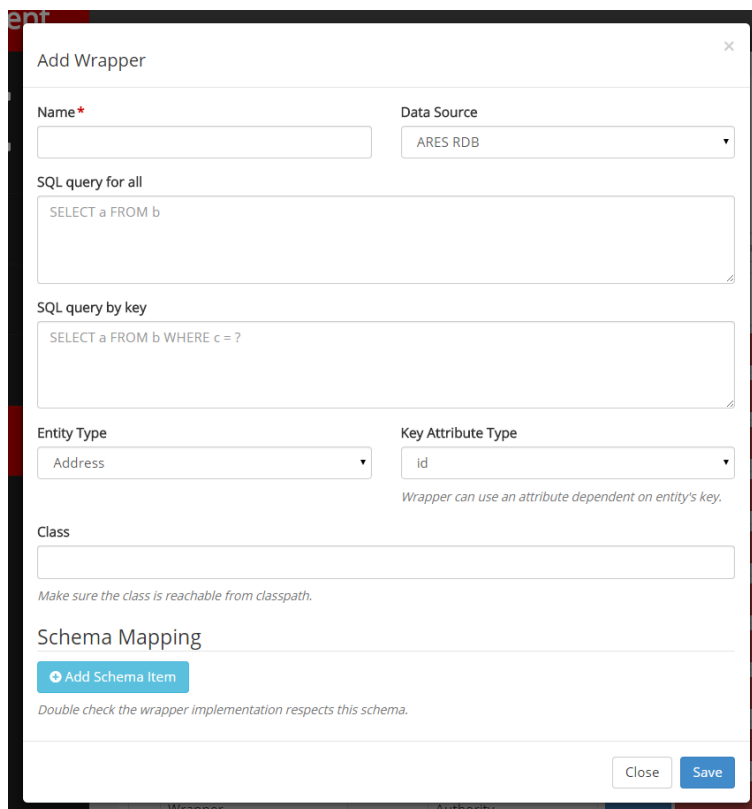
Manage wrappers - components standing above a data source.

[Add Wrapper](#) [Refresh](#)

10 records per page Search...

#	Name	Entity Type	Data Source	Options
1	Address Wrapper	Address	ARES RDB	Edit Delete Clear Cache
2	Company Wrapper	Company	ARES RDB	Edit Delete Clear Cache
3	Firms by Company Wrapper	Company	ARES RDB	Edit Delete Clear Cache
4	Seats Wrapper	Company Seat	ARES RDB	Edit Delete Clear Cache
5	Addresses by Company Wrapper	Company	ARES RDB	Edit Delete Clear Cache

Figure C.9: Wrappers management



Add Wrapper ✕

Name* Data Source

SQL query for all

SQL query by key

Entity Type Key Attribute Type
Wrapper can use an attribute dependent on entity's key.

Class

Make sure the class is reachable from classpath.

Schema Mapping
[Add Schema Item](#)
Double check the wrapper implementation respects this schema.

[Close](#) [Save](#)

Figure C.10: Add a wrapper dialog

One of the most often reasons to create an own implementation occurs when the wrapper should return a nested instance of another master entity, i.e. it cannot refer to a schema mapping between the wrapper and the primary master entity type. In such a case, it is necessary to implement the mapping.

If you are overriding a provided default implementation and want to control caching on your own, override methods `fetchDataByKey`, resp. `fetchAllData`. If you are satisfied with the default caching mechanism, override methods `fetchingByKeyProcess`, resp. `fetchingAllProcess`.

The result of the fetching methods is an instance of `DataContainer`, resp. a collection of `DataContainer` instances. The essential rule is that keys used in data containers must match the global schema, i.e. attribute types' names. If the wrapper does not or cannot use the schema mapping specified via the Admin Client it must manage the mappings on its own.

C.5.3.1 SQL

The most advanced default implementation is provided for relational databases within the `SQLWrapper` class. When the information needed to establish a connection is available and the SQL queries are properly specified via the Admin Client, it is mostly sufficient to override only the results processing in the method `processDataByKeyResults`, resp. `processAllDataResults`, although there is a default behaviour even for that.

When the support for filtering is desired, you do not necessarily have to implement the whole fetching method. There is a method `buildQueryAll(Set<Filter> filters)` that receives the active filters and returns an SQL query. By default, this method returns the SQL query for all data specified via the Admin Client. Overriding this method is the simplest way to enable filtering in SQL wrappers. Since the filtering in SQL wrappers typically lies in specification of the WHERE clause of the SQL query, there is the `WhereConditionBuilder` class that helps to easily build the condition. Note, that the recommended behaviour for filtering by strings with `>` (greater than) operator corresponds to “starts with”.

C.5.3.2 REST

`RESTWrapper` provides default functionalities for querying standard REST APIs and processing simple JSON responses. If a simple concatenation of the base URI configured in the data source and the particular resource specified within the wrapper is sufficient, no special HTTP headers or URL parameters are needed and the received JSON has a simple structure without any composite values, the default fetching method can be used.

A default method for results processing parses the JSON according to the schema mappings. If the JSON has a more complicated structure, you must implement the processing method (`processDataByKeyResults`, resp. `processAllDataResults`).

C.5.3.3 CSV

When building a wrapper for a CSV data source, you can significantly exploit default functionalities of the `CSVWrapper` class. The default fetching method parses the CSV document

according to the schema mappings where a schema item's name corresponds to a column name. The columns are then translated into particular attribute types' names.

C.5.3.4 SOAP

A recommended practice of creating a wrapper above a SOAP web service is to generate the source code from the WSDL, implement a wrapper inheriting the `SOAPWrapper` class, build a jar from all the java files and include it in the system the standard way as with other wrappers.

C.5.3.5 Other

For other types of data sources, you must write a wrapper from scratch. Your implementation must inherit the `Wrapper` class.

C.5.3.6 Deployment

When the implementation is ready, it can be deployed to the integration system (testing of the fetching methods before the deployment is recommended). Build a jar from the class(es), go to a directory where the system is deployed, put the jar to `WEB-INF/lib` and redeploy the application or restart the server.

C.5.4 Edit Wrapper

For editing a wrapper, click on *Edit* button in the table. Editing is done via an analogous dialog as when adding.

C.5.5 Remove Wrapper

For removing a wrapper from the system, click on *Delete* button in the table.

C.5.6 Clear Cache

To manually clear wrapper's cache, click on *Clear Cache* button in the table.

Appendix D

User Manual

It is important to note, that the system created within this thesis is not primarily intended to be used by end users. The UI Client component offering a dynamic user interface above the integration system is being developed within another diploma thesis. Nevertheless, the way of the system API usage described in following sections is the same for both the dynamic client and potential end users.

D.1 API Entry Point

From the client's point of view, the integration system stands for a RESTful API providing master entity resources and their metadata. The entry point found at

`http://{baseURI}/{context}/rest/`

returns a URI of the querying (data) service and the catalog (metadata) service. Thanks to this self-exploration approach, the above mentioned URI is the only URI the client must know.

D.2 Catalog Web Service

The catalog web service serves for metadata accessing. The clients can use this service to explore the global schema and settings of particular master entity types and attribute types. At the root URI

`http://{baseURI}/{context}/rest/catalog/`

there is a list of URIs of all master entity types registered in the system together with their resource identifiers.

Each master entity type is described with metadata located at

`http://{baseURI}/{context}/rest/catalog/{entityResource}`

The metadata includes master entity type's:

- Name
- Description
- Resource identifier
- Information if it is owned – if a master entity is owned, its identification is dependent on another master entity (in relational database terminology we would say a weak entity)
- URI of the owning master entity type for an owned type
- List of attribute type’s metadata

The attribute types have also their URI:

`http://{baseURI}/{context}/rest/catalog/{entityResource}/{attributeName}`

so it is possible to access metadata of the attribute separately. Their metadata involve:

- Name
- Cardinality (single/multiple) – a multiple attribute contains a list of values
- Fetch level - specifies when the attribute is fetched
 - *collection* – the attribute is fetched always
 - *detail* – the attribute is fetched when accessing the master entity’s detail or the attribute directly but not when accessing the collection of master entities
 - *attribute* – the attribute is fetched only when it is accessed directly
- Referenced master entity type – URI of a referenced master entity type, if null, then the attribute is a literal
- Fetched as URI (true/false) – indicates whether the referenced master entity is fetched only as its URI or as a nested master entity
- Key (true/false) – identification of the master entity used by the REST API

D.3 Querying Web Service

The querying web service at

`http://{baseURI}/{context}/rest/data/`

returns a list of URIs of all strong (not owned) master entities, i.e. such master entities whose identification is independent on other master entities and URIs of their collections are in the form:

`http://{baseURI}/{context}/rest/data/{entityResource}`

Accessing this URI for a master entity type results in a collection of all master entities of the type. To reduce number of results, the client should use filtering by specifying an HTTP header `X-Filter` in the following form:

$$\{\text{attributeName}\}\{\text{relation}\}\{\text{someValue}\}$$

where `relation` stands for =, > or <. Multiple filters can be separated by a semicolon.

Each strong retrieved master entity has its own URI in the form:

$$\text{http://}\{\text{baseURI}\}\{\text{context}\}/\text{rest}/\text{data}/\{\text{entityResource}\}/\{\text{key}\}$$

where `key` stands for the value of the master entity's key attribute.

If the master entity is owned, then its URI's form is:

$$\text{http://}\{\text{baseURI}\}\{\text{context}\}/\text{rest}/\text{data}/\{\text{owningEntityResource}\}/\{\text{owningEntityKey}\}/\{\text{entityResource}\}$$

where `owningEntityResource` and `owningEntityKey` identify the owning master entity and `key` stands for the value of the master entity's key attribute.

Appending `/\{\text{attributeName}\}` to any of the two previous URIs points directly to the attribute with given name.

In a situation when an attribute is not fetched because of the fetch level settings (e.g. the fetch level of the attribute is `attribute` but currently the entity's detail is accessed), the value of such an attribute contains a URI of the attribute. This way, the null value can be recognized from the lazily loaded attribute.

Each response of the integration system is complemented by a URI of the corresponding metadata (the master entity type when accessing a collection or an entity detail and the attribute type when accessing directly an attribute). Then, there are additional messages with errors and info notes. For example, if an attribute could not be fetched from the data source, a client finds out from an error message.

Appendix E

DVD Content

/	
doc	documentation of the source code in JavaDoc
example	files referring the demonstration of use
_ ARES REST	ARES REST data source
_ subject-1.0.war	WAR to deploy
_ project	Maven project
_ wrappers	implementation of wrappers
_ wrappers-1.0.war	WAR to deploy
_ project	Maven project
_ ares.sql	PostgreSQL dump of the ARES RDB
_ integration-ares.sql	dump of the integration system database
project	Maven project with the integration system
_ src	Java sources
_ main	main sources
_ test	test sources
_ pom.xml	POM (Project Object Model) file
text	text of the thesis
_ thesis.pdf	PDF file
_ latex	L ^A T _E X sources
readme.txt	DVD content + installation manual