Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Diploma Thesis No. 3679

Development of a Java Library and Extension of a Data Access Layer for Data Access to Non-Relational Databases

Christoph Schmid

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Frank Leymann
Supervisors:	Dr. Vasilios Andrikopoulos
	Steve Strauch
Commenced:	June 19, 2014
Completed:	December 19, 2014
CR-Classification:	C.2.4, C.4, D.2.11, H.2

Abstract

In the past years, cloud computing has become a vital part of modern application development. Resources can be highly distributed and provisioned on-demand. This fits well with the less strict data model of non-relational databases that allows better scaling. Many cloud providers have hosted NoSQL databases in their portfolio.

When migrating the data base layer to a NoSQL model, the business layer of the application needs to be modified. These modifications are costly, thus it is desirable to design an architecture that can adapt to changes without tight coupling to third party components.

In this thesis, we extend a multi-tenant aware Enterprise Service Bus (ESB) with Data Access Layer, modify the management application and implement a registry for the NoSQL configurations. Then, we design an architecture that manages the database connections that adds a transparency layer between the end-user application and non-relational databases.

The design is verified by implementing it for blobstores including a Java access library that manages the access from local applications to the ESB.

Additionally, we evaluate component by measuring the performance in several use scenarios and compared the results to the performance of the vendor SDKs.

Contents

List of Figures

List of Tables

List of Listings

1. Introduction

Cloud computing becomes more and more popular. Advantages are the reduced costs, better scaling, and work-load related costs. There is a variety of established suppliers on the market with similar portfolios the end-user can choose from.

There are different scenarios in which the end-user can profit from cloud computing, which include reduced costs, better scaling, and a direct connection between costs and consumption. In this thesis we will narrow on the application data.

In some scenarios, applications cannot use relational databases due to limitations of its data structure and scalability. In these cases, non-relational like databases NoSQL might be preferred. Many Cloud providers offer hosted databases, including different types of NoSQL databases.

Implementing an application using the database of a specific provider creates dependencies and introduces the danger of vendor lock-in. When the provider decides to increase the price for example, the end-user developer cannot choose another provider without the costs of major modifications to his application. On the other hand, the higher costs of developing provider-independent are often not justified.

An additional layer that manages the storage of data to in the Cloud would solve this problem.

1.1. Scope of Work

This thesis is part of a series of implementation projects that aim to extend an Enterprise Service Bus (ESB) with multi-tenant awareness, and access to cloud data bases.

The management application grants tenant users limited configuration access to the ESB's connectivity and integration services. A registry for user and services has been implemented using the application server JonAS to run the application management JBI Multi-tenancy Multi-container Support (JBIMulti2) [Muh12].

In the projects surrounding Cloud Data Access Support for ServiceMix (CDASMix), the ESB has been extended to allow routing of SQL and NoSQL requests through the router to help with the migration of applications to the cloud. It is possible to route requests to a specific data store provider through the cloud, making cloud access transparent to the tenant. The tenant however had to use a modified version of the native driver and was therefore tied to a specific implementation [Sá13].

In yet another project, a component has been implemented which is able to transform SQL statements for one SQL dialect to another [Xia13] thereby removing the dependency of the application to the specific SQL database vendor.

The target of this thesis is to reimplement the existing NoSQL data access layer and enable the access in a unified manner that allows the access to several compatible NoSQL data stores through a single interface.

In most cases, modifications to the business logic cannot be avoided when modifying applications for the use of NoSQL. Therefore, we do not pursue the possibilities of mapping NoSQL databases onto each other and only explore the implementation of a single uniform data model that can be mapped to every NoSQL database.

1.2. Outline

The remainder of this thesis is structured as follows:

- **Fundamentals, Chapter 2:** this chpter provides a general introduction to the technology concepts that are used for the design and implementation of this thesis.
- **Related Work, Chapter 3:** a discussion of related work and features of products, that provide related services to our implementation.
- **Specification, Chapter 4:** covers the requirements of the extended system and the new components.
- **Design, Chapter 5:** this chapter describes in detail the design decisions for the imlementation.
- **Implementation, Chapter 6:** covers the implementation and modification of the components as described, and a description of the validation.
- **Evaluation, Chapter 7:** a description and documentation of the evaluation of the implementation, which consists of performance measurements of the component and comparison to other SDKs
- **Conclusion and Future Work, Chapter 8:** a brief recap of the findings of this thesis and suggestions for features of future extensions.

1.3. Acronyms

API Application Programming Interface

AWS Amazon Web Services

blob Binary Large Object

CDASMix Cloud Data Access Support for ServiceMix

CSV Character Separated Values

EC2 Amazon Elastic Compute Cloud

EIP Enterprise Integation Pattern

ESB Enterprise Service Bus

IaaS Infrastructure as a Service

IP Internet Protocol

JAR Java ARchive

JBI Java Business Integration

JBIMulti2 JBI Multi-tenancy Multi-container Support

MIS Main Information Structure

NoSQL No SQL or Not Only SQL

OSGi Open Service Gateway initiative

PaaS Platform as a Service

RDBMS Relational DataBase Management System

S3 Amazon Simple Storage Service

SaaS Software as a Service

SDK Software Development Kit

SIS Secondary Information Structure

SOAP Simple Object Access Protocol

SQL Structured Query Language

SQS Amazon Simple Queue Service

UUID Universally Unique Identifier

VM Virtual Machine

WSDL Web Services Description Language

XML eXtensible Markup Language

XQJ XQuery API for Java

YCSB Yahoo Cloud Serving Benchmark

2. Fundamentals

In this chapter we give a general introduction to the technologies used in this thesis. We are be as concise as possible and only go into the details when required to understand the following chapters.

2.1. Cloud Computing

In a typical cloud infrastructure, the customer can request resources on demand through an automated self-service system, and pay dependent on the resource usage. Capabilities can be rapidly and elastically provisioned and adapted to the current demand. Access to the resources is provided through a network [MG11]. The resources are provided in a multi-tenant model, usually through virtualization of hardware components, platform and application instances.

2.1.1. Multi-Tenancy

From the provider point of view, many users from similar backgrounds must be attended to. In this scenario, the multi-tenant model is used to describe the clients. The cloud provider provides access to the resources in a multi-tenant model.

In this context, we define multi-tenancy as sharing the whole technology stack, including hardware, operating system, middleware and application instances [SALM12].

Tenant

A tenant describes a member of a group of users that share the view on an application they use. This includes to some degree the data, the configuration, and the management. Tenants are usually different legal entities [KMK12].

Tenant Space

The tenants rent a predefined share of resources to run different application instances. An important aspect of the tenant space from the provider point of view is that several tenants use the same hardware.

An example for a tenant space is a virtual processing instance in the cloud on which the user can install software [SALM12].

Isolation

Sharing resources among users of different legal entities require higher standards regarding security, privacy, and performance compared to physical instances that provide a certain degree of isolation between the instances by design.

The main aspcects of isolation between tenant of the same system are performance isolation and data isolation.

It must also be ensured that no tenant can use more than "his share" of the resources and affect the other tenants negatively.

The data of tenants must be protected from access and modification by other tenants. This aspect can be further decomposed into communication isolation, for example by using separate message paths, and application isolation, to prevent applications and services of one tenant from accessing data of another tenant's applications or services [SAS+12].

2.1.2. Service Models

Cloud services are often distinguished on the level of the provided service. The levels are infrastructure, platform, and software [MG11].

- **Infrastructure as a Service (IaaS)**: The fundamental computing resources like processing power, storage, network are virtualized, provisioned and provided to the consumer. The customer can choose the configuration from hardware configuration to operating system. Underlying cloud infrastructure is managed by the provider with limited influence by the consumer, such as selecting network components or location.

- **Platform as a Service (PaaS)**: The platform provides the customer the opportunity to deploy own software to the cloud infrastructure of the provider. The software must use the programming language and tools supported by the provider. The provider manages the hardware, network access, storage etc., and the client has full control over the deployment of the application.

- **Software as a Service (SaaS)**: Customer use the provider's applications which are running on a cloud infrastructure. The provider manages everything from hardware, storage and network to the actual software with the exception of user specific settings. The resource is accessed through a thin client like an internet browser.

2.1.3. Deployment Models

Another distinction between cloud services can be made by taking a closer look at the user group.

- **Private Cloud**: The usage of a private cloud is provisioned for a single organization with multiple consumers. Management can be provided by a division of the organization itself or by a third party. The infrastructure can be on premise or off premise.
- **Public Cloud**: A public cloud is provisioned for open use by the general public and often based on a pay-per-use consumption model [BBG11]. The hardware is located on premise of the cloud provider and managed by him.
- **Community Cloud**: The community cloud can be described as an in-between of private and public clouds. The infrastructure is shared by several organizations and support the needs of a group of users that has shared concerns regarding the requirements of the cloud. Resources are on premise of one of the organizations or off premise, managed by one of the organizations or a third party.
- **Hybrid Cloud**: A hybrid cloud is a composition of two or more different of the previously named service models. The possibility to connect the different clouds is either provided by standardized Application Programming Interfaces (APIs) among the provider or by proprietary technology [MG11].

2.1.4. Market Overview

Many cloud solutions are available that encompass the whole spectrum of cloud services. As SaaS provides usually only offer solutions for a very specialized user spectrum, we have decided to exclude them from this selection.

Amazon Web Services

Amazon Web Services (AWS)[1] is one of the first public cloud providers and has been a pioneer since 2006. Their portfolio includes offers for hosted computing, networking, content deliveries, Structured Query Language (SQL) and No SQL or Not Only SQL (NoSQL) data storage, messaging systems and many more.

The most well-known aspects of AWS are Amazon Elastic Compute Cloud (EC2) IaaS service that provides access to virtual servers and Amazon Simple Storage Service (S3), an established Binary Large Object (blob) storage service.

[1] `http://aws.amazon.com`

RackSpace

Like Amazon, RackSpace[2] provides solutions for cloud virtual servers and cloud storage, but is also focused on configuring and maintaining custom private clouds.

RackSpace provided Code for OpenStack, an open source computing software platform.

Microsoft Azure

The portfolio of Microsoft Azure[3] includes IaaS services like virtual services and cloud data services, but also a PaaS services for app- and website development. The cloud data services include relational and NoSQL databases like document stores, key/value stores, and blob stores.

Google Cloud Computing

The cloud services from Google[4] include virtual machines, SQL and NoSQL stores, and application services. The most common and dominant in the PaaS market is the Google App Engine.

Private Cloud Solutions

There are several cloud solutions available for the configuration of a private cloud. The most common are OpenStack[5] and Eucalyptus[6], both are open source.

OpenStack is backed by RackSpace, and provides as key components virtual servers and blob storage. It is part of the Ubuntu Server and widely accepted.

Eucalyptus has similar features and uses an API that is very similar to the Amazon API.

2.2. Enterprise Service Bus

The application landscape in an enterprise often consists of many applications of different vendors that need to exchange data to support the internal workflow.

Implementing point to point connections between each application that need to communicate becomes impractible slow, since the number of possible connections grows quadratic with

[2] http://www.rackspace.com
[3] http://azure.microsoft.com/en-us
[4] https://cloud.google.com
[5] http://www.openstack.org
[6] http://www.eucalyptus.com

the number of applications. Direct connections are also difficult to manage since even minor changes to the API need to be adopted by every connected component.

The purpose of an Enterprise Service Bus (ESB) is to provide a framework that allows to connect applications of different vendors inside an application landscape and can help reduce the communication overhead.

Messages are sent to the ESB which orchestrates the communication between the applications. Thus, there are no direct communications and lot of the routing and data conversion can be abstracted in the ESB. The systems are coupled in a loose way, new applications can be added without changes to the existing applications and can be modified without influencing the other components [Cha04].

A ESB must provide a message delivery system and support a service container.

2.2.1. JBI

Java Business Integration (JBI) is a specification that guides the implementation for a service-oriented architecture. It defines the service container that can be contacted via binding components.

The JBI specification has not been developed further in the last years [JBI05].

2.2.2. Normalized Message Router

One of the possibilities to provide reliable messaging is through a normalized message router. The messages must first be transformed in a normalized message format and can then be routed to the receiver. The receiver then transforms messages in a format that the endpoint can process.

2.2.3. OSGi

Another specification dealing with the management of container is Open Service Gateway initiative (OSGi). Similar to JBI, Open Service Gateway initiative (OSGi) is used to simplify the management and the complexity of large systems [OSG11].

The components are wrapped as OSGi bundles which can be started by the OSGi platform. Besides the bundle registry OSGi also supports advanced dependency and life-cycle management [dCA11]. An OSGi bundle can be used as a modular building block. Bundles are packed as Java ARchive (JAR) files with an included description file containing instructions to the OSGi environment.

Access to functionality between bundles needs to be explicitly allowed [CW13]. The trend in ESB development goes toward other components built on top of the core OSGi framework [dCA11, p. 6].

2.2.4. Enterprise Integration Pattern

Enterprise Integation Pattern (EIP) define another way to handle message routing through pattern. The pattern is specified and defines many implementable building blocks.

The next section provides a short overview of the key components to understand the basics of routing in EIP[7]. The figures for the EIP are taken from the JBoss Fuse for Developers tooling[8].

Message Channel

A message channel is a virtual connection between a message sender and receiver. One application writes message to a particular message channel, the other component then reads the messages.

Messages

Messages are packages of information that are transmitted using a message channel. These messages can then be routed through a messaging system to one or more recipient.

Pipes and Filters

In the EIP, a singe event can trigger several processing steps. These separated tasks are performed by modular components to process the message information.

A core concept is that the functionality is implemented by independent processing steps (filters) that are connected by channels (pipes) [HW03].

A filter receives a message through an exposed inbound pipe, processes it and if necessary modifies and publishes the results to the outbound pipe.

The pipes connect filter, sending output messages from one filter as input to the next filter.

Message Endpoint

A message endpoint (see Figure 2.1) connects an application to a messaging channel. It enables the application to produce and receive messages that are later processed. It is a direct connection of the application API to the API of the messaging system.

The message endpoint invokes the processing of an incoming message by the application, and transforms output of the application that needs to be handled by other data that needs to be processed to a message [HW03].

[7] A full list of the components can be found at [HW03] and `http://www.enterpriseintegrationpatterns.com/toc.html`

[8] JBoss Fuse for Developers tooling: https://www.jboss.org/products/fuse/overview/

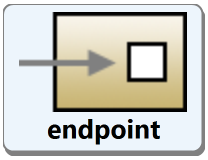
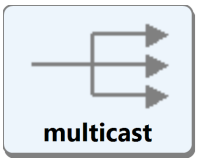


Figure 2.1.: Message Endpoint

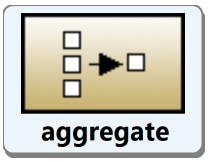
Message Routing

The concept described in the section about pipes and filters only allows the connection of two filter with each other. For advanced combinations, different routing patterns are used.

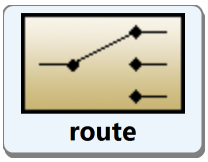
If a message must be sent to several filters, one of the pattern that can be used is the multicast pattern (see first image in Figure 2.2). It defines that a copy of a message is sent to each target.



Multicast



Aggregation



Message Router

Figure 2.2.: Message Routing Symbols

The results of a message that are sent to several filter can be aggregated to a single message again using an aggregator. An aggregator collects the messages belonging together until an exit condition is matched and then publishes a message with the aggregated information of the input messages.

Messages can be sent to one of several predefined targets, depending on the message content, using the routing pattern.

Message Filter

Message filter remove messages from the route if the passing message does not comply with a predefined condition.

Figure 2.3.: Message Filter

Message Translator

The different endpoints may need messages in a different format, or with different header providing the information. The pattern used to modify a message to match another expected format is the message translator, as shown in Figure 2.4 [Apaa], [HW03].

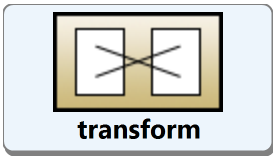


Figure 2.4.: Message Translator

2.2.5. Apache ServiceMix

Apache ServiceMix[9] is an open source ESB that offers the previously discussed features. The development started as an implementation of the JBI standard. ServiceMix Version 3 and 4 provide full support of the JBI specification, including normalized message router, management beans and support for hot deployment of JBI components [Apab].

Since then, it has been completely refactored and uses OSGi as component manager but still supports JBI up until ServiceMix 4.5.3. Since the following versions, ServiceMix 5.0 JBI support has been removed.

In version 4 and 5, ServiceMix uses Apache Karaf[10] as OSGi integration container. OSGi container provide additional features e.g. Apache ActiveMQ provides reliable messaging, EIP, messaging and routing is supported through Apache Camel, and WS-* and RESTful Web services with Apache CXF [ASM].

Apache Camel

Apache Camel implements the framework to use EIP. In addition to the routing functionality, Camel has a lot of ready-to-use implementation of many Camel components[11] that can be integrated in the Camel route as a factory for message endpoints to third party applications.

A selection of relevant component is show in Table 2.1[12]. The selection is incomplete and the functionality of the components has not been examined.

Database Name	Camel Component	Remark
Amazon DynamoDB	Camel-AWS	Key-Value Store
Amazon SimpleDB	Camel-AWS	Key-Value Store

[9] `http://servicemix.apache.org`
[10] `http://karaf.apache.org`
[11] `http://camel.apache.org/component.html`
[12] Full list can be accessed here: `http://camel.apache.org/components.html`

Redis	spring-redis	Key-Value Store
Amazon S3	Camel-AWS and Camel-Jclouds	Blobstore
Azure Blob	Camel-Jclouds	Blobstore
Riak	-	Third party implementation[13]
Cassandra	-	Third party implementation[14]
MongoDB	Camel-MongoDB	Document Store
CouchDB	Camel-CouchDB	Document Store
CouchBase Server	Camel-Couchbase	Document store in the camel-extra repository
Noe4j	Camel-Spring-Neo4j	Graph Database inthe camel-extra repository
SQL	Camel-SQL	Performing SQL queries using JDBC

Table 2.1.: Selection of Existing SQL and NoSQL Camel Components

2.3. NoSQL Data Bases

Relational DataBase Management Systems (RDBMSs) assume a well-defined structure in data, meaning that the properties of the data can be defined upfront. With this structure, the data can be accessed through SQL requests.

To store information, the data as used by the application must be mapped to the format of the data store. But the internal representation does not always comply with the strict table scheme of SQL databases [Tiw11]. In other cases, the limitations to scalability make RDBMS databases impractible to use.

In these cases, No SQL or Not Only SQL (NoSQL) databases might be the solution. NoSQL is an umbrella term for many structured data store systems that are not SQL.

2.3.1. NoSQL Data Types

There are several types of NoSQL data stores with ambiguous defined [15] main categories and their definition in the following.

Key/Value Database or Tuple Store

Key/value data bases store the information using associative arrays[16]. Keys are unique and provide access to the stored data set. This basic functionality is often extended with additional properties regarding consistency, persistency, and ordering.

[13] `https://github.com/amit1000/camel-riak`
[14] `http://github.com/ticktock/camel-cassandra`
[15] `http://nosql-database.org`
[16] associative array: other descriptions are map or dictionary

Examples for key/value databases are Berkeley DB, EHCache, Memcached, Redis, Amazon's DynamoDB, Cassandra, Riak, and Voldemort.

Blobstores

Blobstores are a specialized key/value stores. They are designed to handle values that consist of a Binary Large Object (blob). Blobstores use a "container name"/"blob name" tuple as access key to further organize the entries. Popular blobstores are Microsoft Azure Blob Service, Amazon S3, and OpenStack Object Storage.

Column-Databases

The data in column databases is stored in a column-oriented way, in contrast to the row-oriented format in RDBMS.

For data sets with varying data entries, this reduces the required space needed since they don't need to store null values for empty columns. The items are grouped to column families, that provide access to the content [Tiw11, p. 11].

Examples for column databases are Google Bigtable, Apache HBase, and Hypertable.

Document Stores

Document stores are designed for storing, retrieving, and managing semi-structured data. The information is stored as a whole and not split into its constituent name/value pairs. The document and metadata are indexed and later used to find the requested documents faster.

Examples for document stores are CouchDB, MongoDB and Redis [Tiw11, p. 11].

Graph Databases

Graph databases are the best choice if the application data can be represented as a graph structure of nodes, edges, and properties to store data.

Nodes contain pointers to the connected edges and vice versa which makes navigation between the nodes and edges very efficient. Graph data bases often have a database specific query languages.

Popular examples are Neo4j, FlockDB, and InfiniteGraph.

XML Databases

The database model of XML databases is based on the eXtensible Markup Language (XML) format. Like graph databases, XML databases are best used when the application data already uses the same format as the storage provider. XML databases are often SQL databases with enabled XML extension, but native XML databases also exist.

Examples for XML databaes are BaseX, eXist, and Sedna.

3. Related Work

In this chapter, we will discuss related work and features of products, that provide related services to our implementation.

3.1. Cloud Computing with Private Clouds

As mentioned in Section 2.1 there are several private cloud provider. The functionality offered is usually based on the API of established cloud providers such as Amazon, Google, or Microsoft.

In this thesis, we restricted the selection in the validation to public cloud providers due to the time and effort required to manage a private cloud infrastructure.

The functionality is, however, not limited to public Clouds and all components are able to connect to private cloud services in the same way they are able to connect to the selected public cloud.

We refrain from using private Clouds in this thesis due to the required time required for setup and configuration only use public Cloud provider..

3.2. Enterprise Service Bus as a Service

Providers like AWS with the service Amazon Simple Queue Service (SQS)[1] and Microsoft Azure with the Azure Queues and Service Bus Queues[2] provide reliable messaging as cloud service to their customers.

For the messaging system in our scenario, we take the perspective of the service provider that provides his clients with a share of the service bus functionality.

Since the Cloud ESB and the Cloud storage components are part of the portfolio of a single provider, the integration of those components is supported by tutorials and white paper[3].

Our work can be considered as a competing open source implementation to the messaging products of these cloud provider.

[1] `http://aws.amazon.com/sqs`
[2] `http://azure.microsoft.com/en-us/services/service-bus`
[3] `http://aws.amazon.com/articles/1464`

3.3. NoSQL Data Bases

From the user point of view, the main goal is to access NoSQL databases without committing to a specific provider. The user would like to be able to change the supplier without any major modification to his application code. Besides the routing of the requests through a service bus which unifies the requests and routes them to the configured provider, there are other methods to reach this goal.

3.3.1. Unified Access Libraries

There are third party libraries that allow access to several NoSQL implementations. These implementation provide a unified api for a single types of NoSQL data stores.

Apache Jclouds

Apache Jclouds[4] is a universal access library, that abstract access to blob data stores of every major private and public provider.

Jclouds supports 30 cloud providers and cloud software stacks including Amazon, Azure, OpenStackand vCloud. The API supports abstractions of compute services, blobstores and load balancer.

The access is however limited to a single target provider, content dependent target selection, and replication functionality is not included.

We use the functionality of Jclouds wrapped as a Camel component to access the provider in the implementation of this thesis.

Blueprints

Tinkerpop offers the same functionality a Jclouds for graph databases with the implementation of Blueprints[5]. Blueprints is able to connect to several graph databases including Neo4j and InfiniteGraph.

3.3.2. Shared API

Another way to ease the coupling between implementation and vendor is to share a single API to access the content. This is either done by implementing a shared, standardized API by the vendors, or when "newcomers" build their own API after the quasi standard set by established products.

[4] `https://jclouds.apache.org`
[5] `http://blueprints.tinkerpop.com`

XQuery API for Java

XQuery API for Java (XQJ)[6] is an example of a shared API based on a public specification. It enables Java programmers to execute XQuery against an XML database. XQJ is based on the W3C XQuery 1.0 specification [BFM+10] and supported by most XML databases per default.

Even with a shared API modifications to the source code are still necessary to modify the driver and connection details.

Eucalyptus and AWS

An example for a shared API by established quasi standards is the connection between Eucalyptus[7] and AWS. Eucalyptus has been implemented as a private cloud mirror of AWS components. Both services include a blob data store. This simplifies the exchange between private and public Clouds and can be a selection factor for the selection of components for a hybrid cloud.

The APIs are usually similar, but not identical. Each provider tries to find customers with unique features.

[6] http://www.w3schools.com/xquery
[7] https://www.eucalyptus.com

4. Specification

This chapter covers the requirements of the extended system and the new components. Since the existing components should continue to be fully operational, the quantity structure and the requirements are partially set by the requirements for the already existing system.

The following we only lists new and changed requirements, everything else can be found in the theses of previous students [Muh12, Sá13, Xia13].

The first section covers the quantity structure in Section 4.1.

The functional requirements are defined in the form of use cases in Section 4.2 and the non-functional requirements are specified in Section 4.3.

4.1. Quantity Structure

QS 1: Management

The existing administration interface of JBI Multi-tenancy Multi-container Support (JBIMulti2) calculates with the following volume of work in normal use:

> **Quantity Structure:** One JBIMulti2 instance connected to a cluster of two ServiceMix instances should handle the following quantities without impact on other non-functional requirements [Muh12]:
>
> - The system can store 1000 registered service assemblies or services, 10 tenants with each having 1000 tenant users and corresponding records in the Configuration Registry. [...]
> - The system allows 50 tenant users or system administrators to execute [management] use cases concurrently over the Web UI and 100 management requests per second over the Web service interface.
> - A management request to the Web UI or the Web service interface is responded to in 4 seconds on normal networking conditions .

Additionally, the system must be able to perform the following NoSQL specific management tasks:

- The system can store 1000 data store configurations.

- These configurations consist of 1000 source data stores, 5000 target data stores as well as 10000 Main Information Structures (MISs) und Secondary Information Structures (SISs)[1]

QS 2: Usage

The system is able to handle 10000 separate tenant users[2]. Some of them have several active data store configurations, others have none.

The requirements for this large number of tenants can only be fulfilled by a properly equipped cluster of ServiceMix instances. A single instance should be able to handle the following workload:

- Access any of the configured data store configurations.
- Allow three active tenants with two tenant users each.
- Access data from all tenant users concurrently.
- Even though it is expected that the usage of the shared resources will influence the usage of the other tenants, no tenant user can block the shared resource.

4.2. Use Cases

In this section, only the use cases that are needed to use the Unified NoSQL Data Store component are mentioned. The use cases are separated into two groups: Management for the administrative operations (Section 4.2.2 and Figure 4.1), and usage for the use cases that apply for the end-user interacting with the Java access library (Section 4.2.3 and Figure 4.2).

Existing use cases that were used without modification are colored grey, new use cases and modified use cases are colored white in both figures.

4.2.1. Naming Convention

The “data store configuration” stores the information that is needed to route the request from the end-user through the ESB to the target NoSQL provider. It contains the “data store” and the “information structures”.

Data stores contain one “source data store” that configures the connection between the API and the connector of the ESB and at least one “target data store” that stores the access data needed to connect from the ESB to the NoSQL provider.

[1] Each of these configurations consists of one source data store, on average five target data stores configurations, and for each target data store configuration two information structures (on average).
[2] 10 tenants with 1000 users each.

The information structures “main information structure” and “secondary information structure” are needed for the mapping between source and target data store.

Additionally each target data store can be connected with a “target data store policy” which contains additional information about the usage of the data store like read-only access, write-only access, quotas etc.

4.2.2. Management

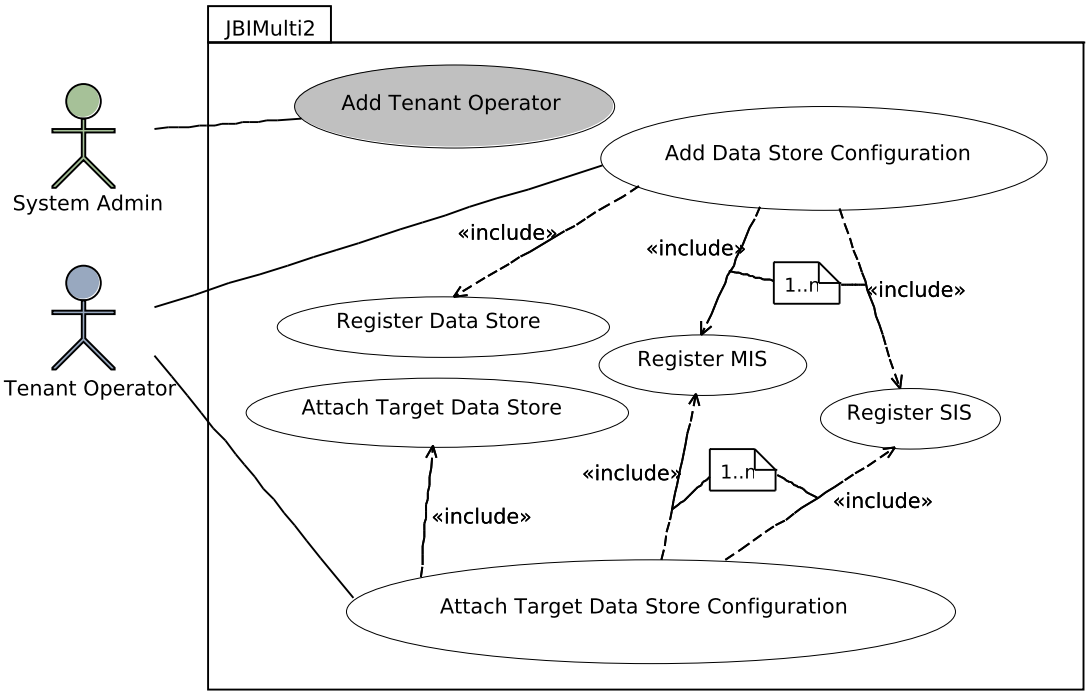


Figure 4.1.: Management Use Cases

FR 1.1: Add a new Data Store Configuration

Name	**Add Data Store Configuration**
Goal	The tenant operator wants to add a new NoSQL data store configuration.
Actor	Tenant operator
Pre-Condition	The tenant operator has been added with permission to add new data store configurations. All systems are running (PostgreSQL DB, JOnAS, ServiceMix). The Unified Blobstore component has been deployed to ServiceMix. SoapUI is running and can be used to send requests to the JOnAS JBIMulti2 management application.

* continued on the next page

Post-Condition	The data store configuration has been added and the end-user can use the data store configuration.
Post-Condition in Special Case	The configuration step has been aborted and can be executed again with correct data.
Normal Case	1. Register a new NoSQL data store. (see use case: FR 1.3: Register a new Data Store) 2. Register one or more main information structures. (see use case: FR 1.5: Register a Main Information Structure) 3. Register one or more secondary information structures, reference one of the main information structures registered before (see use case: FR 1.6: Register a Secondary Information Structure)
Special Cases	2.a Source or target data store does not exist: 1. System returns error message 2. User returns to step 3. 3.a Main information structure does not exist: 1. System returns error message 2. User returns to step 3

Table 4.1.: Use Case *Add Data Store Configuration*

FR 1.2: Attach a Target Data Store Configuration to an existing Configuration

Name	**Attach Target Data Store Configuration**
Goal	The tenant operator wants to add a new NoSQL target data store to an existing data store configuration and connect it based on the rules defined in the information structures.
Actor	Tenant operator
Pre-Condition	The tenant operator has been added with permission to add new data store configurations. All systems are running (PostgreSQL DB, JOnAS, ServiceMix). The Unified Blobstore component has been deployed to ServiceMix. SoapUI is running and can be used to send requests to the JOnAS JBIMulti2 management application. A data store configuration exists.
Post-Condition	The data store configuration has been added and the end-user can use the data store configuration.
Post-Condition in Special Case	The target data store has not been added to an existing data store configuration.

* continued on the next page

Normal Case	1. Attach one or more new NoSQL target data stores to an existing data store configuration. (see use case: FR 1.4: Attach a Target Data Store) 2. Register one or more main information structures and reference the new target data store. (see use case: FR 1.5: Register a Main Information Structure) 3. Register one or more secondary information structures and reference one of the main information structures registered before. (see use case: FR 1.6: Register a Secondary Information Structure)
Special Cases	1.a The referenced Data Store Configuration does not exist: 1. System returns error message 2. User returns to step 1 2.a Source or target data store does not exist: 1. System returns error message 2. User returns to step 2 3.a Main information structure does not exist: 1. System returns error message 2. User returns to step 3

Table 4.2.: Use Case *Attach Target Data Store Configuration*

FR 1.3: Register a new Data Store

Name	**Register Data Store**
Goal	The tenant operator wants to add a new data store, consisting of a source data store and a target data store.
Actor	Tenant operator
Pre-Condition	The tenant operator has been added with permission to add new data store configurations. All systems are running (PostgreSQL DB, JOnAS, ServiceMix). The Unified Blobstore component has been deployed to ServiceMix. SoapUI is running and can be used to send requests to the JOnAS JBIMulti2 management application.
Post-Condition	The data store configuration has been added.
Post-Condition in Special Case	No changes have been made.
Normal Case	The operator sends a request to JBIMulti2 using SoapUI with the configuration for the source and target data stores.

* continued on the next page

Special Cases	1.a A source data store or target data store with the same name already exists: 1. System returns error message 2. User returns to step 1 1.b Required data store fields are missing: 1. System returns error message 2. User returns to step 1 1.c Fields are filled with invalid characters: 1. System returns error message 2. User returns to step 1 1.d The credentials for the target data store are invalid: 1. No error is returned, invalid credentials are detected during the first request.

Table 4.3.: Use Case *Register Data Store*

FR 1.4: Attach a Target Data Store

Name	**Attach Target Data Store**
Goal	The tenant operator wants to add additional target data stores to an existing data store configuration.
Actor	Tenant operator
Pre-Condition	The tenant operator has been added with permission to add new data store configurations. All systems are running (PostgreSQL DB, JOnAS, ServiceMix). The Unified Blobstore component has been deployed to ServiceMix. SoapUI is running and can be used to send requests to the JOnAS JBIMulti2 management application. At least one data store configuration exists.
Post-Condition	The data store configuration has been altered and the end-user can use the new target data store to reference in a main information structure.
Post-Condition in Special Case	Source and target data store configuration data are not stored.
Normal Case	The operator sends a request to JBIMulti2 using SoapUI with the configuration for the attachment of the new target data stores to the existing configuration.

* continued on the next page

Special Cases	1.a One of the target data stores has a name that is already registered: 1. System returns error message 2. User returns to step 1 1.b Required data store fields are missing for one of the target data stores: 1. System returns error message 2. User returns to step 1 1.c One of the fields contains invalid characters: 1. System returns error message 2. User returns to step 1 1.d The credentials for one of the target data stores are invalid: 1. No error is returned, invalid credentials are detected during the first request.

Table 4.4.: Use Case *Attach Target Data Store*

FR 1.5: Register a Main Information Structure

Name	**Register MIS**
Goal	The tenant operator wants to add a main information structure to connect a source data store and a target data store.
Actor	Tenant operator
Pre-Condition	The tenant operator has been added with permission to add new data store configurations. All systems are running (PostgreSQL DB, JOnAS, ServiceMix). The Unified Blobstore component has been deployed to ServiceMix. SoapUI is running and can be used to send requests to the JOnAS JBIMulti2 management application.
Post-Condition	The data store configuration has been added and the end-user can use the data store configuration.
Post-Condition in Special Case	No main information structure has been added.
Normal Case	The operator sends a request to JBIMulti2 using SoapUI with the configuration for the attachment of the new target data stores to the existing configuration.
Special Cases	1.a Required fields are missing: 1. System returns error message 2. User returns to step 1 1.b Source or target data store does not exist: 1. System returns error message 2. User returns to step 1 1.c Main information structure name is invalid: 1. System returns error message 2. User returns to step 1

Table 4.5.: Use Case *Register MIS*

FR 1.6: Register a Secondary Information Structure

Name	**Register SIS**
Goal	The tenant operator wants to add a new secondary information structure to connect a source data store with a target data store.
Actor	Tenant operator
Pre-Condition	The tenant operator has been added with permission to add new data store configurations. All systems are running (PostgreSQL DB, JOnAS, ServiceMix). The Unified Blobstore component has been deployed to ServiceMix. SoapUI is running and can be used to send requests to the JOnAS JBIMulti2 management application.
Post-Condition	The data store configuration has been added and the end-user can use the data store configuration.
Post-Condition in Special Case	Source and target data store configuration data are not stored.
Normal Case	1. The operator sends a request to JBIMulti2 using SoapUI with the configuration for the attachment of the new secondary information structure.
Special Cases	1.a Required fields are missing: 1. System returns error message 2. User returns to step 1 1.b Source or target data store does not exist: 1. System returns error message 2. User returns to step 1 1.b The main information structure does not exist: 1. System returns error message 2. User returns to step 1 1.c Secondary information structure name is invalid: 1. System returns error message 2. User returns to step 1

Table 4.6.: Use Case *Register SIS*

4.2.3. Usage

FR 2.1: Store a Blob using the ESB Unified Blobstore Component

Name	**Store a Blob with the ESB**
Goal	The tenant operator wants to store a blob using the Java access library and the ESB back end.
Actor	Tenant end-user
Pre-Condition	ServiceMix with Unified Blobstore component is running. The data store has been configured.

* continued on the next page

Post-Condition	The blob has been added to the target data store(s) and can be read. (see use case FR 2.2: Read a Blob using the ESB Unified Blobstore Component)
Post-Condition in Special Case	1. The blob has been added to at least one target data store, but not all, and can be requested by using the FR 2.2: Read a Blob using the ESB Unified Blobstore Component operation. 2. The blob has not been added to any target data store.
Normal Case	1. Connect to the ESB via the Java access library using a tenant specific endpoint. (see use case FR 2.5: Connect to the ESB Unified Blobstore Component) 2. Send the add request, containing of the container name, blob name, and the blob itself. 3. Wait for all data stores to complete the upload. 4. Return success status message.
Special Cases	1.a Connection to the ESB failed, the tenant endpoint does not exist or is not configured for this tenant user: 1. Return error message with detailed error information. 2. Upload failed. 2.a No matching target data store entry exists connected to the source data store by the information structures, but a default target data store has been configured. 1. Use the default configuration(s) 2. Continue with step 3. 2.b No matching target data store entry exists connected to the source data store by the information structures, and no default target data store has been configured. 1. System returns error specific failure message. 2. Upload failed. 2.c All matching target data stores are read only. 1. System returns error specific failure message. 2. Upload failed. 2.d All writable matching target data stores are not reachable. 1. System returns error specific failure message. 2. Upload failed. 2.e Not every, but at least one writable matching target data store is reachable. 1. Proceed with step 3 as in the normal case for all reachable target data stores. 2. Return success status message with information about the failed send requests.

* continued on the next page

Special Cases (Continued)	4.a All writable and reachable target data stores return with failure status code: 1. System returns error specific failure message. 2. Upload failed 4.b All writable and reachable target data stores respond not within a reasonable time frame (timeout): 1. System returns error specific failure message. 2. Upload failed. 4.c Not every, but at least one writable and reachable target data stores return with failure status code: 1. Return success status message with information about the failed send requests. 4.d Not every, but at least one writable and reachable target data stores does not respond within a reasonable timeframe (timeout): 1. Return success status message with information about the failed send requests.

Table 4.7.: Use Case *Store a Blob with the ESB*

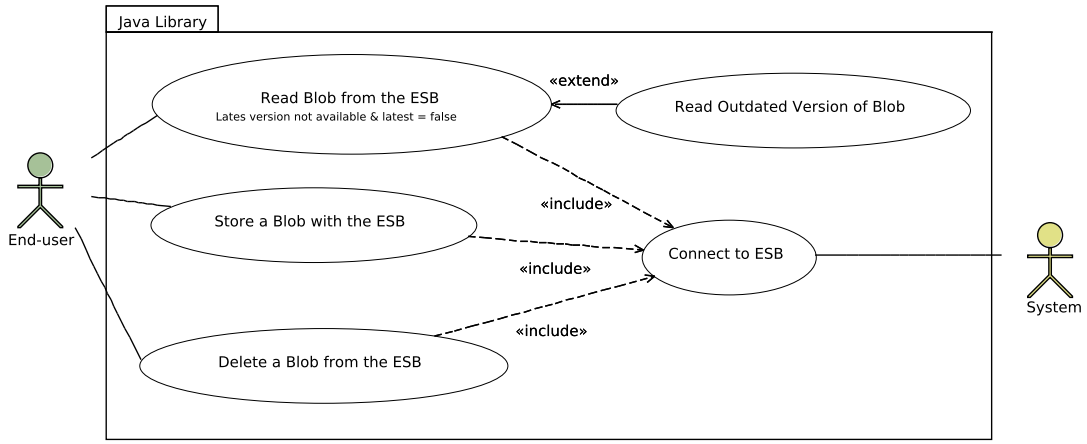


Figure 4.2.: End-user Use Cases

FR 2.2: Read a Blob using the ESB Unified Blobstore Component

Name	**Read Blob from the ESB**
Goal	The tenant operator wants to read a blob that has been stored previously using the Java access library.
Actor	Tenant end-user
Pre-Condition	ServiceMix with Unified Blobstore component is running. The data store has been configured. SoapUI is running and can be used to send requests to the JOnAS JBIMulti2 management application. A blob has been added previously.
Post-Condition	The end-user has local access to the blob
Post-Condition in Special Case	1. The end-user has no access to the blob 2. FR 2.3: Read an outdated version of a Blob
Normal Case	1. Connect to the ESB via the Java access library using a tenant specific end-point. (see use case FR 2.5: Connect to the ESB Unified Blobstore Component) 2. Send a read request to the ESB. 3. (Automatically) select one of the target data stores that store the blob in the latest version and allows reads. 4. Wait for the Java access library to complete the download. 5. Show success status message.

* continued on the next page

Special Cases	1.a Connection to the ESB failed, the tenant endpoint does not exist or is not configured for this tenant user: 1. Show error message with detailed error information 2. Download failed 2.a No matching target data store entry exists for the source data store, container and blob name combination, but a default targetds has been configured. 1. Use the default configuration(s) 2. Continue with step 3. 2.b No matching target data store entry exists for the source data store, container and blob name combination, a default has not been configured. 1. System shows error specific failure message. 2. Download failed. 2.c All matching target data stores are write only. 1. System shows error specific failure message. 2. Download failed. 2.d All readable matching target data stores are not reachable. 1. System shows error specific failure message. 2. Download failed. 2.e The selected read target is unreachable. 1. Select other target data store that stores the blob in the latest version and allows reads. 2. Continue with step 3. 2.f None of the read targets with the latest version is available. 1. Evaluate "latest-only" parameter. 1.a If true, download failed (post condition 1) 1.b If false, continue with step 4 (post-condition 2) 4.a Target data store does not respond within a reasonable time frame (timeout): 1. Continue with step 3, mark data store (temporarily) as not reachable.

Table 4.8.: Use Case *Read Blob from the ESB*

FR 2.3: Read an outdated version of a Blob

Name	**Read Outdated Version of Blob**
Goal	The tenant operator wants to read a blob that has been stored previously using the Java access library, even though none of the target data stores with the latest version is available.
Actor	Tenant end-user
Pre-Condition	ServiceMix with Unified Blobstore component is running. The data store has been configured. SoapUI is running and can be used to send requests to the JOnAS JBIMulti2 management application. A blob has been added previously. None of the target data stores with the latest version of the blob is available. A connection to the ESB has been established.

* continued on the next page

Post-Condition	The end-user has local access to the blob
Post-Condition in Special Case	1. The end-user has no access to the blob
Normal Case	1. Send a read request to the ESB. 2. (Automatically) select one of the target data stores that store the blob in the latest version and allow reads and that are available. 3. Wait for the Java access library to complete the download. 4. Show success status message with information, that not the latest version has been delivered.
Special Cases	2.a All matching target data stores are write only 1. System shows error specific failure message 2. Download failed. 2.b All readable matching target data stores are not reachable 1. System shows error specific failure message 2. Download failed. 2.e The selected read target is unreachable 1. Select other target data store that stores the blob with the latest available version. 2. Continue with step 2. 4.a Target data store does not respond within a reasonable time frame (timeout): 1. Continue with step 2, mark data store (temporarily) as not reachable.

Table 4.9.: Use Case *Read Outdated Version of Blob*

FR 2.4: Delete a Blob using the ESB Unified Blobstore Component

Name	**Delete a Blob from the ESB**
Goal	The tenant operator wants to delete a blob using the Java Library and the ESB back end.
Actor	Tenant end-user
Pre-Condition	ServiceMix with Unified Blobstore component is running. The data store has been configured.
Post-Condition	The blob has been added to the target data store(s).
Post-Condition in Special Case	1. The blob has been deleted from at least one target data store, but not all, and cannot be requested by using the FR 2.2: Read a Blob using the ESB Unified Blobstore Component operation. 2. The blob has not been deleted to any target data store.

* continued on the next page

Normal Case	1. FR 2.5: Connect to the ESB Unified Blobstore Component to the ESB via the Java access library using a tenant specific endpoint. 2. Send the delete request, containing of the container name, blob name, and the blob itself. 3. Update timestamp for each target data store, container and blob name if writing was successful. 4. Wait for all data stores to complete removal. 5. Return success status message.
Special Cases	1.a Connection to the ESB failed, the tenant endpoint does not exist or is not configured: 1. Return error message with detailed error information. 2. Deletion failed. 2.a No matching target data store entry exists for the source data store, container and blob name combination, but a default target data store has been configured. 1. Use the default configuration(s) 2. Continue with step 3. 2.b No matching target data store entry exists for the source data store, container and blob name combination, a default has not been configured. 1. System returns error specific failure message. 2. Deletion failed. 2.c All matching target data stores are read only. 1. System returns error specific failure message. 2. Deletion failed. 2.d All writable matching target data stores are not reachable. 1. System returns error specific failure message. 2. Deletion failed.

* continued on the next page

Special Cases (Continued)	2.e Not every, but at least one writable matching target data store is reachable. 1. Proceed with step 3 as in the normal case for all reachable target data store 2. Return success status message with information about the failed send requests 3.a No blob with the exact same name exists in the specified container: 1. Issue a warning but update the timestamp 2. Continue with step 4 4.a All writable and reachable target data stores return with failure status code: 1. System returns error specific failure message 2. Deletion failed 4.b All writable and reachable target data stores respond within a reasonable time frame (timeout): 1. System returns error specific failure message 2. Deletion failed 4.c Not every, but at least one writable and reachable target data store returns with failure status code: 1. Return success status message with information about the failed send requests. 4.d Not all, but at least one writable and reachable target data stores does not respond within a reasonable timeframe (timeout): 1. Return success status message with information about the failed send requests.

Table 4.10.: Use Case *Delete a Blob from the ESB*

FR 2.5: Connect to the ESB Unified Blobstore Component

Name	**Connect to ESB**
Goal	The tenant operator wants to establish a connection with the ESB NoSQL Component using the Java access library
Actor	System
Pre-Condition	ServiceMix with Unified Blobstore component is running. The data store has been configured. SoapUI is running and can be used to send requests to the JOnAS JBIMulti2 management application. A blob has been added previously.

* continued on the next page

Post-Condition	The end-user is connected to the ESB, he can now perform read, write and delete operations. (see use case: FR 2.2: Read a Blob using the ESB Unified Blobstore Component, FR 2.1: Store a Blob using the ESB Unified Blobstore Component, FR 2.4: Delete a Blob using the ESB Unified Blobstore Component
Post-Condition in Special Case	The Enduser is not connected to the ESB.
Normal Case	1. Send a "connect" request to the ESB tenant endpoint, authenticate by providing tenant id and tenant user name. 2. Wait for the connection to be established. 3. Show success status message.
Special Cases	1.a The tenant endpoint is not reachable. 1. Show error message with detailed error information 2. Connection failed 1.a The tenant endpoint does not exist 1. Show error message with detailed error information 2. Connection failed 1.b The tenant is not allowed to access the endpoint connection 1. Show error message with detailed error information 2. Connection failed 1.c The tenant is not registered 1. Show error message with detailed error information 2. Connection failed 1.d The maximum number of concurrent connections has been reached 1. Show error message with detailed error information 2. Connection failed 2.a The endpoint does not respond in a timely fashion 1. Show error message with detailed error information 2. Connection failed

Table 4.11.: Use Case *Connect to ESB*

4.3. Non-Functional Requirements

Besides the requirements that are bound with specific tasks in Section 4.2 (Use cases) there are requirements that define the mode of operation in a general way. These non-functional requirements are highly influenced by the requirements of the already existing system and its components.

4.3.1. Multi-Tenancy

As a multi-tenant aware system, the system has to comply with the requirements as listed in Section 2.1.1. In a multi-tenant environment, resources are used by several independent

parties. Ideally, the parties are able to share these resources without any knowledge and influence of each other.

The requirements to reach this goal are listed below:

NFR 1.1: Performance Isolation

Even though the resources are limited and shared among the tenants, the tenants should not influence the response time and data throughput of other tenants. At least, tenants that use the system in a non-malignant way should not be able to block the system for other tenants.

NFR 1.2: Data Isolation

The data of the tenants are stored separately. Configurations and end-user data that has been added for the users of one tenant cannot be seen or changed by users of other tenants.

The communication paths used by the message flow are created separately for each tenant.

4.3.2. Performance and Resource Usage

NFR 2.1: Minimal Performance Losses

Many of the resources needed to comply with Quantity Structure are used by external components that cannot be influenced like the network connectivity or the response time of the cloud provider. The main goal must be to reduce the performance loss of routing through the ESB and unification steps compared to a direct connection between the vendor specific API and the data store.

NFR 2.2: Scalability

The large demand created by the number of tenants and tenant users estimated in Section 4.1 (quantity structure) cannot be managed by a single static system. It should therefore be possible to add additional resources to the ESB component, if necessary.

NFR 2.3 Modest Resource Usage

Resources are limited and shared among the tenants. Beyond that, the resources of external (cloud) data stores might have to be paid separately, either static for the time available or dynamic dependent on the usage.

The component should use these resources modest and avoid unnecessary and expensive requests.

4.3.3. Consistency and Security

NFR 3.1 Consistent Management Configuration

All administrative actions have to be put into action in a timely fashion.

NFR 3.2: Consistent Data Storage

Depending on the configuration, the data can be distributed and replicated among several target data stores. This is useful if not all target data stores are available at all times. In this case, it is possible that the end-user receives different versions of the requested file, depending on which back end target data stores are available. In this scenario the end-user must be able to influence the behavior and only receive the latest version or none at all.

NFR 3.3: Security and Access control

The data of the user must be protected against access form unauthorized parties. Enduser are only allowed to access the data in a way that the tenant operator defined previously (read or write).

Additionally, the end-user should have no access to the configuration data, e.g. the name of the target data store or the access credentials.

The data transfer between end-user and the ESB endpoint, and between the ESB and the target data store must be encrypted. Connections between the internal components of the database are assumed to be in a demilitarized zone, security measures in this area are not necessary [Muh12].

NFR 3.4: Transparency

The data store configuration must be transparent to the end-user. Routing, target data store selection, and replication configuration is done by the ESB administrator.

Since this is a prototype, error messages can be communicated to the end-user, even if it contains information that should be hidden in a productive system.

4.3.4. Backward Compatibility, Extensibility, and Reusability

FR 4.1: Backward Compatibility and Ease of Integration

The NoSQL extensions integrate in a system based on ServiceMix with many extensions to internal JBI and OSGi, as well as new components. It is essential that these components continue to function.

FR 4.2: Extensibility

NoSQL is a growing field, ServiceMix components that provide access to NoSQL databases are currently developed and improved. If it is possible, independent components should be used to handle the connections between the ESB and the NoSQL data store. Therefore, it is important that other NoSQL provider can be integrated easily and in a timely fashion, and that updates to the OSGi NoSQL connections can be integrated and new functionality provided by external providers can be integrated.

This also affects the Java access library, that has to follow the changes in the back end. The implemented components must provide a simple and generic API to make it usable in other applications.

4.3.5. Maintainability and Documentation

The code will be developed further in the future, most likely by different developers without any further introduction to the code. Developer documentation is therefore an important part of this project.

NFR 5.1: Code Maintainability

The code must be maintainable by other developers without much training. To accomplish that, the code must be well documented and structured with a clear separation of the components. Forking should be avoided at all costs.

NFR 5.2: Documentation

Design decisions and the architecture has to be well-documented. The API and all functions must be described in detail.

NFR 5.3: Installation and Development Ease

It must be ensured that the administrator as well as coming developers are able to take the application to use and start customization and development. The installation procedure must be described with all required third party products and their versions.

NFR 5.4: Accessibility

The provided functionality must be easy accessible for many end-users. Developers must be able to implement new applications using the provided functionality without huge investment.

5. Design

This chapter describes in detail the design decisions for the imlementation in Chapter 6. Section 5.1 describes the logic of the matching algorithm that is used by by filter and router in Camel to connect source and target data stores based on predefined rules and the message parameter “containername” and “blobname”.

Section 5.2 shows architecture of JBIMulti2 with Cloud Data Access Support for ServiceMix (CDASMix) the modifications that that need to be implementded.

The last two sections show the design of the ServiceMix components in Section 5.5 and the Java access library in Section 5.6.

5.1. Matching

The matching algorithm is used to define the rules of the connection between source and target data stores. Possible scenarios are one source data store to one of several target data stores depending on the name of the container and blob, data replication, and read- and write-only data stores.

A minimal configuration consists of one source data store an one target data store with main and secondary information structures connecting them.

The data store configurations are configured independently for each tenant and tenant user. In the following, the tenant and tenant user information is not shown to simplify the descriptions and examples.

5.1.1. Definitions

Source Data Store

The source data store is used to configure the endpoint used by the end-user to connect to the ESB using the java access library. It must contain the Camel component that provides the endpoint as well as the connection URL and the authorized tenant user.

Target Data Store

The target data store contains the information needed to connect the ESB component to the NoSQL provider e.g. the provider credentials and the Jclouds component. The target data store also contains the information whether the data store can be used to handle read and/or write requests.

A source data store is created with a single target data store configuration, additional target data stores can be added later. If several source data stores use a connection to a single provider, each of the source data stores needs to create a separate target data store configuration.

Information Structures

The inforamtion structure is used to define the rules of the connection between source data store and target data store. It consists of two parts, the main information structure, and the secondary information structure.

The information structures are used as defined in CDASMix [Sá13, p. 50] but with extended functionality. The main information structure contains the "first layer" of abstraction of the stored data. The blobstore is related to the the name of the container[1], in which the blob is stored. The secondary information structure is the "second layer". It is connected to the name of the blob[2].

Main and secondary information structure are evaluated separately and only match if both evaluations return true.

5.1.2. Matching Types

There are several types that define how the request parameters are compared.

Absolute

Absolute matching types contain the expected name of the entity. It matches only if the request has the exact same value. A main information structure with the absolute value "container1" would only match requests with the container "container1", not "conTainer1", "container" or "container12".

Absolute information structures can be defined by using the expected value, or with the prefix "<absolute>". The previous example can be created using the value "container1" as well as "<absolute>container1".

[1] microsoft calls it container, amazon prefers the name bucket
[2] usually the filename

Regular Expression

Information structures of the type regular expression (regex) will match the request parameter against the defined regular expression. Regular expressions are defined using the prefix "<regex>". Overlapping expressions are allowed, for example the information structures "<regex>.*\.txt" and "<regex>Example\..*" would both match the blob name "Example.txt".

Default

Default matching types match every request parameter, as long as the request is not served by a configuration of any other type. For this type of matching, the information structure needs to be aware of the other information structures.

Priority

The information structures that are created using these types have different matching priorities. If two of the information structures match, the information structure with the lower priority is ignored.

Information structures with the same priority are equal. Table 5.1 shows the partitioning of information structures in priority classes based on type of their parts.

Main Information Structure	Secondary Information Structure	Priority
absolute or regex	absolute or regex	high
absolute	default	medium
default	default	low

Table 5.1.: Priority of Information Structures

5.1.3. Restrictions

Matching should be as distinct and intuitive as possible. We added the following restrictions to simplify the configuration process and to reduce the number of possibilities to a consistent subset.

As mentioned before, information structures including a default match only if no other information structure with a higher priority matches. Replication in these cases is only possible if the main, or the main and secondary information structure are identical, as shown in Table 5.2.

MIS	SIS
absolute	absolute
absolute	regex (regular expression)
absolute	default
regex	absolute
regex	regex
default	default

Table 5.2.: List of all Valid Configurations

The configurations listed in Table 5.3 lists the configurations that were marked as invalid configurations and will cause an exception during the start of the ServiceMix component.

MIS[3]	SIS[4]	Problem
default	absolute	Conflicts with MIS = absolute and SIS = default.
default	regex	Overlapping with similar information structure possible.
regex	default	Overlapping with similar information structure possible.

Table 5.3.: List of all Invalid Configurations

The first configuration can conflicts with another configuration that has an absolute main information structure and default type as secondary information structure. Since only one of these cases can be allowed, the more common case, the container set as absolute with the blob name as default, has been selected as valid. The next two constellations are prohibited to hinder the administrator to add configurations that lead to unwanted data replication. These restrictions do not limit the functional scope of the applications since the described behavior can still be implemented using regular expressions without default configurations.

5.1.4. Routing Types

With the valid configurations, it is possible to run the NoSQL storage with different configurations.

Content Dependent Routing

The Unified NoSQL component matches the information structure dependent on the container and blob name to one of several target data stores. This can be accomplished by using information structures that do not overlap, by using information structures with different priorities, or by a combination of both.

[3]MIS: Main Information Structure
[4]SIS: Secondary Information Structure

Data Replication

The information structures have the same priority and information structures that overlap at least partially. A partial overlap could be configured as an absolute information structure and a regular expression that also matches the value of the absolute information structure.

For the absolute overlap, the information structure values are identical. If no matching information structure exists for the request, an error message will be returned.

Mixed Configurations

Data replication can be combined with content dependent routing, allowing a flexible configuration of the system landscape.

5.1.5. Examples

Goal of this section is to demonstrate the system is expected to work with a few examples. The first part will describe configurations, the second part will describe how requests are affected.

Before the requests can be sent to the ESB by the end-user, the following NoSQL data stores have to be configured:

C. No.[5]	SDS[6]	TDS[7]	MIS	SIS	r[8]	w[9]
1	sds1	tds1	container1	example.txt	✓	✓
2	sds1	tds2	container1	example.txt	✗	✓
3	sds1	tds3	container1	<default>	✓	✓
4	sds1	tds4	<default>	<default>	✓	✓
5	sds1	tds1	cont2	example.txt	✓	✓
6	sds1	tds-bu	cont2	<regex>(.*)\.txt	✗	✓
7	sds2	tds-readonly	c1	example.txt	✓	✗
8	sds2	tds-writeonly	c1	example.txt	✗	✓
9	sds2	tds5	cont2	example.txt	✓	✓
10	sds2	tds6	cont2	<regex>(.*)\.txt	✓	✓
11	sds3	tds7	cont	hello.txt	✗	✓
12	sds3	tds8	<default>	<default>	✓	✓

Table 5.4.: Data Store Configuration for the Example

[5] Configuration number, referenced in the following description.
[6] Source Data Store
[7] Target Data Store
[8] read
[9] write

Requests

With these configurations request will be mapped as follows:

M. No.[10]	req.	SDS	MIS	SIS	C. No.[11]	TDS
1	PUT	sds1	container1	example.txt	1, 2	tds1 and tds2
2	GET	sds1	container1	example.txt	1	tds1
3	PUT	sds1	container1	asdf.txt	3	tds3
4	GET	sds1	newcontainer1	example.txt	4	tds4
5	PUT	sds1	cont2	example.txt	5, 6	tds1 and tds-bu
6	GET	sds1	cont2	example.txt	5	tds1
7	PUT	sds1	cont2	example2.txt	5	tds-bu
8	GET	sds1	cont2	example2.txt		
9	PUT	sds2	c1	example.txt	8	tds-writeonly
10	GET	sds2	c1	example.txt	7	tds-readonly
11	PUT	sds2	cont2	example.txt	9, 10	tds5 and tds6
12	GET	sds2	cont2	example.txt	9, 10	tds5 or tds6
13	GET	sds3	cont	hello.txt		
14	GET	sds3	cont	hello2.txt	12	tds8

Table 5.5.: Requests and the Matching Configurations

Further Explanation

M. No.[10]	Explanation
1	The PUT request 1 matches with two absolute configurations, so both target data stores are used (tds1 and tds2).
2	The GET request with the same parameters only matches the first target data store because the second one does not allow write operations.
3 and 4	The third request is matched to the first configuration with default values (configuration 3). Configuration 4 with <default> as main and secondary information structure is not used, since the configuration with an absolute value as main information structure and <default> only as secondary information structure has a higher priority. Configuration 4 is used for the fourth request, since no other configuration exists.
5	Request No. 5 is an example of a request with a secondary target data store that is used as a backup. The backup data store matches every .txt file so it is easy to configure a general backup service for all files of a given type.
6	The Get-Request will be served by “tds1” and not the backup “tds-bu”, since “tds-bu” only supports write operations.

[10]Matching number
[11]Configuration number

7 and 8	The configuration is still tricky, wrong configuration can easiliy result in unwanted behaviour. In the above backup example 7, if the main data store is not configured, the blobs will be sent only to the backup target data store. Reading will fail.
9 and 10	If the update of the data stores should be managed outside of the ESB, even different data stores for reading and writing can be configured.
11 and 12	Regular expressions and absolute entries are evaluated equally. This means that the put request will be sent to both tds5 and tds6, and get requests will be served by either tds5 or tds6 (the data store with the later timestamp will be preferred, if it is not reachable the other will be used.).
13 and 14	Request 13 cannot be fulfilled because none of the data store configured is able to perform a read operation. The default data store is not used because a “non-default" configuration exists, even though it is not able to perform the required request. If the end-user tries to perform this get operation, he will see an error message. Request 14 on the other hand can be fulfilled by the default configuration with target data store tds8.

Table 5.6.: Detailed Explanation of the Matchings

5.2. Architectural Overview

Figure 5.1 shows the components of CDASMix as implemented in [Sá13]. JBIMulti2 handles the configuration through the Web Service API and the business logic, stores the configurations in the service registry database, and deploys the service assemblies through the JMS Mangement Service.

The implementation of the Unified NoSQL component uses the same configuration components but a different design for the ServiceMix components. Instead of a mix of JBI service units and OSGi components, only two OSGi components are used, one of which provides access to the service registry of the other component.

The JBIMulti2 component has alredy been extended for the implementation of the CDASMix components. These extensions need to be modified to store all the information needed configure the connections. For that, the Web Service API as well as the Business Logic need to be extended to store the additional options needed for the NoSQL component. Storage from JBIMulti2 to the service registry database is done using Hibernate. The tenant registry an the configuration registry can be used as they are.

The ServiceMix components are implemented independently from the CDASMix components and work without influencing each other.

The NoSQL Registry connects to the service registry and loads the NoSQL routing information. On startup, the Unified Blobstore component uses the NoSQL Registry to access the configurations and configure the Camel routes accordingly.

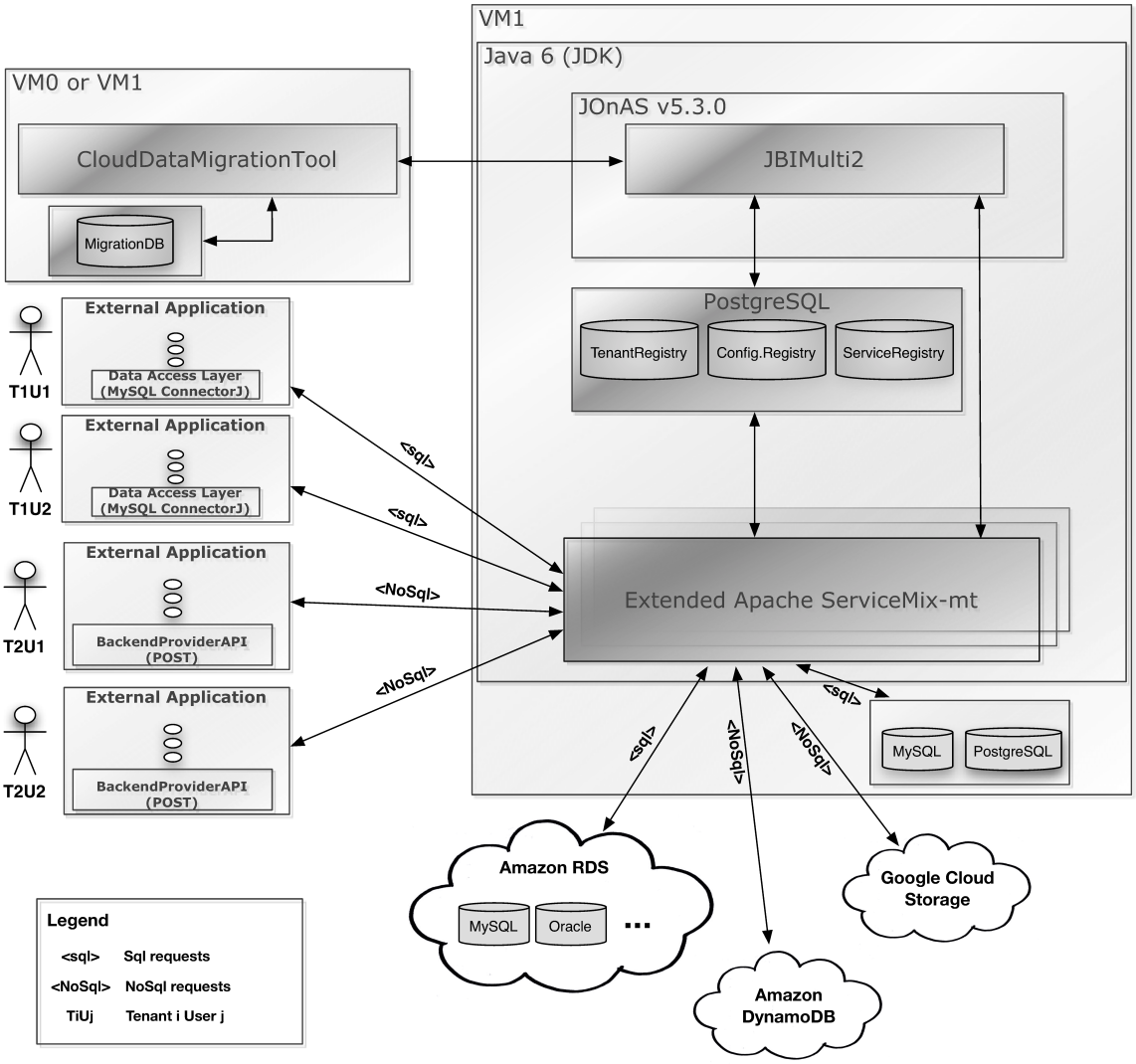


Figure 5.1.: CDASMix Before the Modifications [Sá13]

5.3. JBIMulti2 and CDASMix Modifications

5.3.1. JBIMulti2 Web Service API

The implementation of the use cases scenarios as described in Section 4.2 involves the configuration of the application required a modification of JBIMulti2 components.

The Web Service API and the underlying business logic needed to be modified. The API is accessed through Web Services Description Language (WSDL) messages sent using the application SoapUI.

Data Store

Table 5.7 shows the fields that are used to configure the application as described in the use cases FR 1.1: Add a new Data Store Configuration and FR 1.4: Attach a Target Data Store.

The column "SQL" shows the values that are needed to create an sql connection, the column "NoSQL" the values that are needed for a unified NoSQL connection.

Field	Description	SQL	NoSQL
serviceAssemblyName	JBI Service assembly used by the data store.	✓	✗
sourceDataSourceName	Name of the data source used by the end-user to identify the connection configuration.	✓	✓
sourceDataSourceType	Storage type, for example SQL or any type of NoSQL.	✓	✓
sourceDataSourceProtocol	Protocol used by the end-user to connect to the ESB, http for NoSQL.	✓	✓
sourceDataSourceURL	URL used to connect to the ESB.	✓	o[12]
targetDataSourceName	Internal name of the target data store configuration.	✓	✓
targetDataSourceType	Type of the target data store. If targetDataSource differs from the targetDataSource, the ESB data store provider must handle the transformation.	✓	✓
targetDataSourceProtocol	Protocol used to connect to the external data store provider.	✓	✓
targetDataSourceURL	URL used to connect to the target data store. Can be used to configure special proxy configurations for some Camel providers.	✓	✓
targetDataSource-NativeDriverName		✓	✗
targetDataSourceUser	Username or access name.	✓	✓
targetDataSourcePassword	Password or access credentials.	✓	✓
targetDataSource-CamelProvider	Camel component used to connect to the external component.	✗	✓
targetDataSourceReadable	Target data store can be used for get operations. Needed for advanced data replication configuration (boolean) .	✗	✓
targetDataSourceWritable	Target data store can be used for put operations. Needed for advanced data replication configuration (boolean) .	✗	✓

Table 5.7.: WSDL Fields Used to Configure the Data Store Configurations

[12] optional

Information Structures

Usage of the main and secondary information structure is the same as for CDASMix. The only difference is that no service assembly is neeeded therefore the validity checks had to be modified.

Field	Description
serviceAssemblyName	SQL only, reference to the corresponding service assembly.
sourceDataSourceName	Reference to the corresponding source data store.
targetDataSourceName	Reference to the corresponding target data store.
sourceDataSourceType	Type of the corresponding source data store.
targetDataSourceType	Type of the corresponding target data store.
mainInformationStructureName	Content of the main information structure.

Table 5.8.: WSDL Fields Used to Configure the Main Information Structure

Field	Description
serviceAssemblyName	SQL only, reference to the corresponding service assembly.
sourceDataSourceName	Name of the corresponding source data store.
targetDataSourceName	Name of the corresponding target data store.
sourceDataSourceType	Type of the corresponding source data store.
targetDataSourceType	Type of the corresponding target data store.
mainInformationStructureName	Content of the main information structure.
secondaryInformationStructureName	Content of the main information structure.

Table 5.9.: WSDL Fields Used to Configure the Secondary Information Structure

5.3.2. JBIMulti2 Domain

The domain module of JBIMulti2 creates the class representation of the configurations and stores the entries in the database using Hibernate.

Since CDASMix uses custom SQL requests to access the tables in the database the generated tables must remain identical. The only option to store the parameter needed for the Unified Blobstore component is to generate new tables that reference the existing tables.

CDASMix uses the parameter “datasourcetype” to identify CDASMix SQL entries, therefore no complications with entries of a different types of data sources are expected.

5.4. Unified NoSQL

The only NoSQL-type dependent components are the message translator and the message endpoints.

In the following sections we describe the functionality on the basis of a Unified Blobstore example configuration.

The component is an OSGi component. On startup, it accesses the ServiceMix registry to access the NoSQL configurations.

To ensure message isolation, each tenant is assigned a separate Camel context. The different tenant contexts can be shut down and restarted separately.

5.4.1. Blobstore Example

In this section, we describe the context configuration for a single user. The following text describes the EIP components in the same order as shown in Figure 5.2

1) Jetty Endpoint for Each Tenant User

The Camel-Jetty component is used to start an HTTP server that listens to inbound HTTP requests. The HTTP requests are transformed in the Camel message format, HTTP parameter of the request are translated in Camel message parameter.

The used URL for our Jetty-endpoint is in the following format:
`http://<SERVER-IP>:<PORT>/<TENANT-UUID>/<USER-UUID>`

- <SERVER-IP>: The IP of the server. This value is usually set to 0.0.0.0, thus configuring the endpoint to listen to any connections sent to the IP of the running machine.
- <PORT>: The server port, we selected 8082 for the Unified Blobstore.
- <TENANT-UUID>/<USER-UUID>: These UUID are generated by JBIMulti2 and identify the user.

The required HTTP header are UUID for tenant and user, source data store name and source data store type.

We also include a validator behind the Jetty endpoint that rejects incomplete message and unauthorized requests (not shown).

Since the requests can take some time to process (in case of a blobstore, sending several MB or GB to the cloud provider), we suggest increasing the parameter MaximumIdleTime, the time Jetty keeps a connection open.

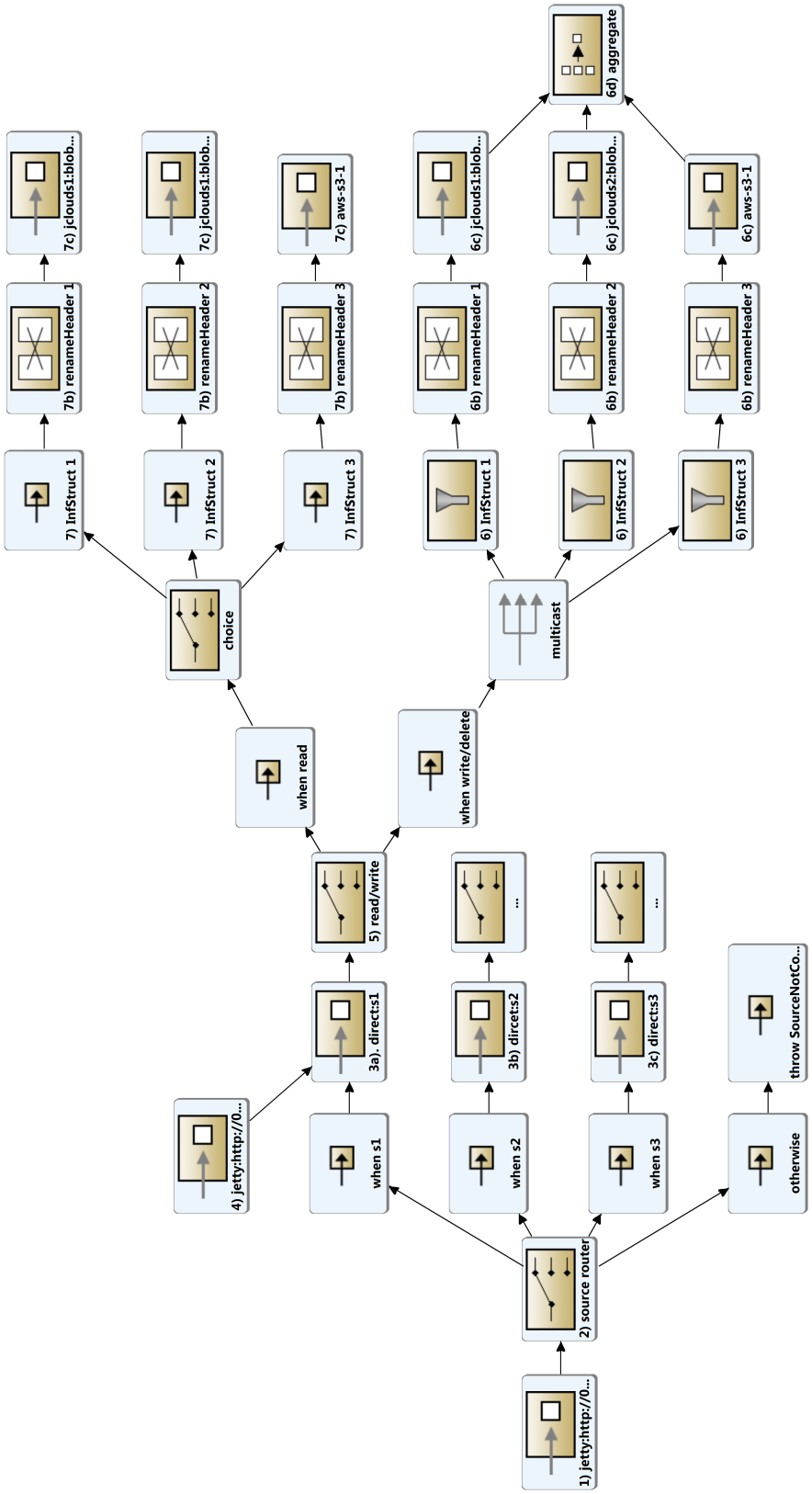


Figure 5.2.: Generated Camel Context

2) Router to Separate Source Data Stores

The next EIP component we use is a Content Dependent Router to separate messages depending on the source data store message header.

If the source data store has not been configured, an exception is thrown and communicated through Jetty as a server error.

In this scenario, we examine the path in case of source data store "s1" further.

3) Direct Endpoint

Direct is a Camel component that generates endpoints that provides synchronous invocation of any consumers when a producer sends a message exchange.

Each source data store has its own direct endpoint.

4) Additional Jetty Endpoint

It is possible to add additional message endpoints in the source data store configuration.

This can be used to configure custom URLs that are used for a single source data store endpoint. In this example, we configure another Jetty endpoint that listens to the URL `http://<SERVER-IP>:8081/s1`.

5) Read/Write Router

Requesting data from the NoSQL data store must be handled differently than changing it. In case of data replication, read requests can be responded by a single target data store while requests modifying the data like read or write must be performed by every target data store.

6a) Multicast and Information Structure Filter

The message to edit a database entry is broadcast to the route of every target data store that is writable. Based on the information structure, requests that do not concern target data store are then filtered out.

6b) Message Translation

We then need to transform the message in a format that the NoSQL Camel endpoint can process. The transformation in case of blobstores is simple, the header names and values defining the the operation, container and blob name are modified.

Other NoSQL types may require a more complex translation.

6c) NoSQL Endpoint

The message is then sent to the provider specific endpoint that sends the request to the actual database. The status message of the operation (success or failure) is set as the message body by the NoSQL endpoint.

6d) Aggregate

Last step is to aggregate the status messages of the different endpoints to a single message that can then be sent back through the initial Jetty endpoint and in case of an error displayed by the Java access library.

7a) Route to Matching Endpoint

In case of a read request, the message is sent to the first target data store that fits the information structure.

7b) and 7c) Message Translation and NoSQL Endpoint

The message is then similar to the read request translated from the unified format in a format that the NoSQL endpoint can process. Since the request is fulfilled by a single endpoint, aggregation is not necessary, and the result of the read request is then sent back using the initial Jetty Endpoint.

5.4.2. Modifications Needed for Other NoSQL Data Stores

The described design is modular and usable for any type of NoSQL data store. The only NoSQL type dependent components are the message translator and the NoSQL message endpoint.

As listed in Section 2.2.5, Apache Camel provides many Camel components that can be used to generate NoSQL message endpoints. If they provide the required functionality, if they need to be extended or if they work at all should be checked before starting the implementation.

5.5. ServiceMix Components

5.5.1. NoSQL Registry

The NoSQL Registry component loads the configurations added through JBIMulti2 and provides access to the Unified Blobstore in a easily processable format.

The NoSQL Registry is split in two parts: an interface that is used to abstract the functionality needed by the NoSQL component, and the actual implementation.

The implementation uses the interface to register itself to the OSGi BundleContextRegistry. The Unified Blobstore component then loads the registry on startup, accessing only the functions provided by the interface. This increases the modularity and affects the NFR 5.1: Code Maintainability, since the registry can be replaced later on without affecting the Unified Blobstore component.

The registry uses Hibernate and the same class model as JBIMulti2 in the application domain. Therefore, further changes and additions to the class model are added automatically to the registry, which increases maintainability

5.5.2. Unified Blobstore

The Unified Blobstore component loads the configuration from the NoSQL Registry on startup and configures the Camel contexts, one for each tenant. The ESB component uses Camel components, and the configuration dependent routing is implemented using Camel routes.

The context can be stopped and reconfigured independently from all the other tenant contexts, and the routes of each tenant have no intersections to comply with NFR 1.2: Data Isolation.

The ServiceMix Blobstore uses Camel-Jetty to configure an access point to receive messages. For each tenant user a separate Jetty endpoint is configured. Jetty provides an http endpoint that can be accessed using a predefined URL. The URL for the user is in the following format:

`http://<SERVER-IP>:<PORT>/<TENANT-UUID>/<USER-UUID>`

The server IP is usually set to 0.0.0.0 to configure Jetty to creates a public access point that can be used by replacing it with the actual server Internet Protocol (IP) address. The Universally Unique Identifiers (UUIDs) are generated by JBIMulti2 and must be known by the user.

A Camel route is then defined that directs messages according to the configured rules defined as in Matching and sent to one or more Camel-Jclouds endpoints that manage the connection from to the NoSQL provider. The routing is implemented using EIP.

Alternative Design

Another possibility to implement the ServiceMix NoSQL component would be to create an OSGi component without Camel, that handles the request directly, transforms it to a format that is supported by the native driver, and sends the request using the native driver. The component could easily use the native driver and therefore use the full support of the available operations with the performance of the native driver. This design has been rejected due to the fact that the connection driver for every NoSQL store had to be implemented manually. Using Camel, a lot of components are already available, or will be in future releases.

5.6. Java Access Library

The java access library can be used by local implementations to connect to the ServiceMix endpoint. Each request is self-contained, meaning that no connection has to be established beforehand.

Connection details are configured in the constructor, the needed parameters are:

- URL to the Jetty endpoint
- tenant and user UUID
- name of the source data store

Supported methods are as defined in Section 4.2 put, get and delete. Error messages during the routing are disclosed by the Unified Blobstore component with an http error code, the message is wrapped in a application specific exception and presented to the user.

6. Implementation

This chapter covers the implementation and modification of the components as described in Chapter 5 Design.

The first section describes the third party components that are used in the implementation and the steps required to set up the development environment and run CDASMix.

The second section shows the required modifications to the JBIMulti2 management application.

The last two sections provide the details of the implementation of the ServiceMix components and the Java access library that can be used to connect from local applications to the ESB.

6.1. Third Party Components

The installation guide of JBIMulti2 and CDASMix uses ServiceMix in version 4.3.0 that includes Camel in version 2.6 (released in January 2011).

Since then, Camel implemented several features that can be used to connect from Camel routes to different NoSQL databases. The development is still active, and new functions and bug-fixes are implemented in each release. Therefore it would be the best solution to keep ServiceMix and Camel up to date to profit from the continuing development.

To comply with the backwards compability non-functional requirement in Section 4.3.4, the existing JBIMulti2 and CDASMix modules must continue to work. Apache removed JBI support from the ServiceMix implementation from version 5 upwards which is needed to run the JBIMulti2 and CDASMix ServiceMix components.

The compromise is to update ServiceMix to the latest stable version that still supports JBI which is ServiceMix 4.5.3. Every other program has been used in the version as suggested in the CDASMix installation manual.

This decision determined the selection of other dependent components. Table 6.1 lists the applications that are used during the implementation. This information is useful to find the right source code, documentations and support.

Program	Version	Remark
JOnAS	5.2.2	Runs the configuration server of JBIMulti2
ServiceMix	4.5.3	Latest version that supports JBI and therefore JBI-Multi2

Camel	2.10.7	Part with ServiceMix 4.5.3
Camel-Jclouds	2.10.7	Camel component.
Camel-Jetty	2.10.7	Camel component.
Jclouds-Blobstore	1.4.0	Version used by Camel-Jclouds 2.10.7.
Hibernate	3.6.4.Final	Used by JBIMulti2 to store management configurations to the PostgreSQL database
PostgreSQL	9.1-901.jdbc3	Used by JBIMulti2.
Httpclient	4.3.5	Library used by the Unified Blobstore Client Library to send requests from a local application to the Jetty component.

Table 6.1.: Selected Programs and Version Used for the Implementation

Another possibility would have been to upgrade the Camel version inside of ServiceMix. Posts on the mailing list suggest that it is possible[1], but it there are no official guides and the risk of unexpected side effects seemed too high.

6.1.1. Update of ServiceMix

Before the implementation, we had to upgrade ServiceMix to version 4.5.3, and install the ServiceMix part of JBIMulti2 and CDASMix.

One possibility is to download the default assembly[2] from the website and add JBI support using the Karaf management console. After that, an error message is written in the startup log file[3]. This can be solved by uninstalling one of the two Apache Aries Transaction Manager.

There are no working test scenarios for JBIMulti2 and CDASMix, verifying the functionality thoroughly after a major update cannot be conducted thoroughly. We did, however, install the JBIMulti2 and CDASMix components in ServiceMix, verified that no startup error was shown in the error log, and then started the existing SoapUI use scenario.

6.2. JBIMulti2 and CDASMix Modifications

This section describes the modifications to the management interface. The order is following the path of a configuration requests, from the modified Simple Object Access Protocol (SOAP) requests sent through SoapUI to the modified JBIMulti2 request handler and then to the configuration database.

We tried to ensure the that the other JBIMulti2 and CDASMix components continue to function by only changing these components when absolutely necessary.

[1] http://servicemix.396122.n5.nabble.com/-td5638386.html
[2] http://servicemix.apache.org/downloads/servicemix-4.5.3.html
[3] http://servicemix.396122.n5.nabble.com/-tp5719860.html

6.2.1. WSDL Request

Listing 6.1 shows an example SOAP request that adds a minimal NoSQL source and target data store configuration. Additional target data stores can be added in additional steps.

The requests are similar structured to the CDASMix requests that are used to configure SQL connections. The keyword of the blobstore has been defined as "unified-nosql-blobstore", to separate these configurations from the configurations of CDASMix.

Source and target data store are then connected by a main and secondary information structure as shown in Listing 6.2 and Listing 6.3. The request is then handled by the JBIMulti2 API.

```
<wsdl:attachNoSQLDataSource>
  <wsdl:sourceDataSourceName>nosqlsource</wsdl:sourceDataSourceName>
  <wsdl:sourceDataSourceType>unified-nosql-blobstore-1.0.0</wsdl:sourceDataSourceType>
  <wsdl:sourceDataSourceComponent>camel-jetty</wsdl:sourceDataSourceComponent>
  <wsdl:sourceDataSourceConnectionURI>http://0.0.0.0:8182/nosqlsource</
      wsdl:sourceDataSourceConnectionURI>
  <wsdl:targetDataSourceName>targetds</wsdl:targetDataSourceName>
  <wsdl:targetDataSourceType>unified-nosql-blobstore-1.0.0</wsdl:targetDataSourceType>
  <wsdl:targetDataSourceComponent>jclouds</wsdl:targetDataSourceComponent>
  <wsdl:targetDataSourceProvider>azureblob</wsdl:targetDataSourceProvider>
  <wsdl:targetDataSourceUser>${CDAS Accounts#azure_1_user}</wsdl:targetDataSourceUser>
  <wsdl:targetDataSourcePassword>${CDAS Accounts#azure_1_password}</
      wsdl:targetDataSourcePassword>
  <wsdl:targetDataSourceReadable>true</wsdl:targetDataSourceReadable>
  <wsdl:targetDataSourceWritable>true</wsdl:targetDataSourceWritable>
</wsdl:attachNoSQLDataSource>
```

Listing 6.1: SoapUI Request to add a NoSQL Data Store Configuration.

```
<wsdl:attachDataSourceSecInformationStructure>
  <wsdl:sourceDataSourceName>nosqlsource</wsdl:sourceDataSourceName>
  <wsdl:targetDataSourceName>targetds</wsdl:targetDataSourceName>
  <wsdl:sourceDataSourceType>unified-nosql-blobstore-1.0.0</wsdl:sourceDataSourceType>
  <wsdl:targetDataSourceType>unified-nosql-blobstore-1.0.0</wsdl:targetDataSourceType>
  <wsdl:mainInformationStructureName>&lt;default&gt;</wsdl:mainInformationStructureName>
  <wsdl:secondaryInformationStructureName>&lt;default&gt;</
      wsdl:secondaryInformationStructureName>
</wsdl:attachDataSourceSecInformationStructure>
```

Listing 6.2: SoapUI Request to add a Default Main Information Structure.

```
<wsdl:attachDataSourceMainInformationStructure>
  <wsdl:sourceDataSourceName>nosqlsource</wsdl:sourceDataSourceName>
  <wsdl:targetDataSourceName>targetds</wsdl:targetDataSourceName>
  <wsdl:sourceDataSourceType>unified-nosql-blobstore-1.0.0</wsdl:sourceDataSourceType>
  <wsdl:targetDataSourceType>unified-nosql-blobstore-1.0.0</wsdl:targetDataSourceType>
  <wsdl:mainInformationStructureName>&lt;default&gt;</wsdl:mainInformationStructureName>
  </wsdl:attachDataSourceMainInformationStructure>
```

Listing 6.3: SoapUI Request to add a Default Main Information Structure.

6.2.2. JBIMulti2

The required modification to JBIMulti2 are the extension of the Web service API, and adding methods to handle the requests including validation and storage.

JBIMuitl2 Web Service API

The first step to extend the JBIMulti2 configuration back-end was to add new requests to the JBIMulti2 WSDL file. A listing of the required additions of the WSDL file an be found in Appendix A.1.

Maven is then used to generate the Java classes that are used to access the data of the SOAP messages. Validation methods for the new requests are added. The handling of the requests to add main- and secondary information structure requests had to be modified since the unified NoSQL store does not need a service assembly.

After the validation the information is stored in the database “ServiceRegistry”. Hibernate is used for this step, the new NoSQL classes had to be added to the persistence.xml configuration file.

Service Registry Database and Persistence Manager

To store the entries in the database the class structure representing the database has to be written and marked with Hibernate annotations. The classes are then added to the Hibernate configuration file.

In result the configuration “validate” should be used. The generated database can be seen in Figure 6.1. Yellow tables are new, grey tables already existed, and the light grey table is not used for NoSQL blobstores.

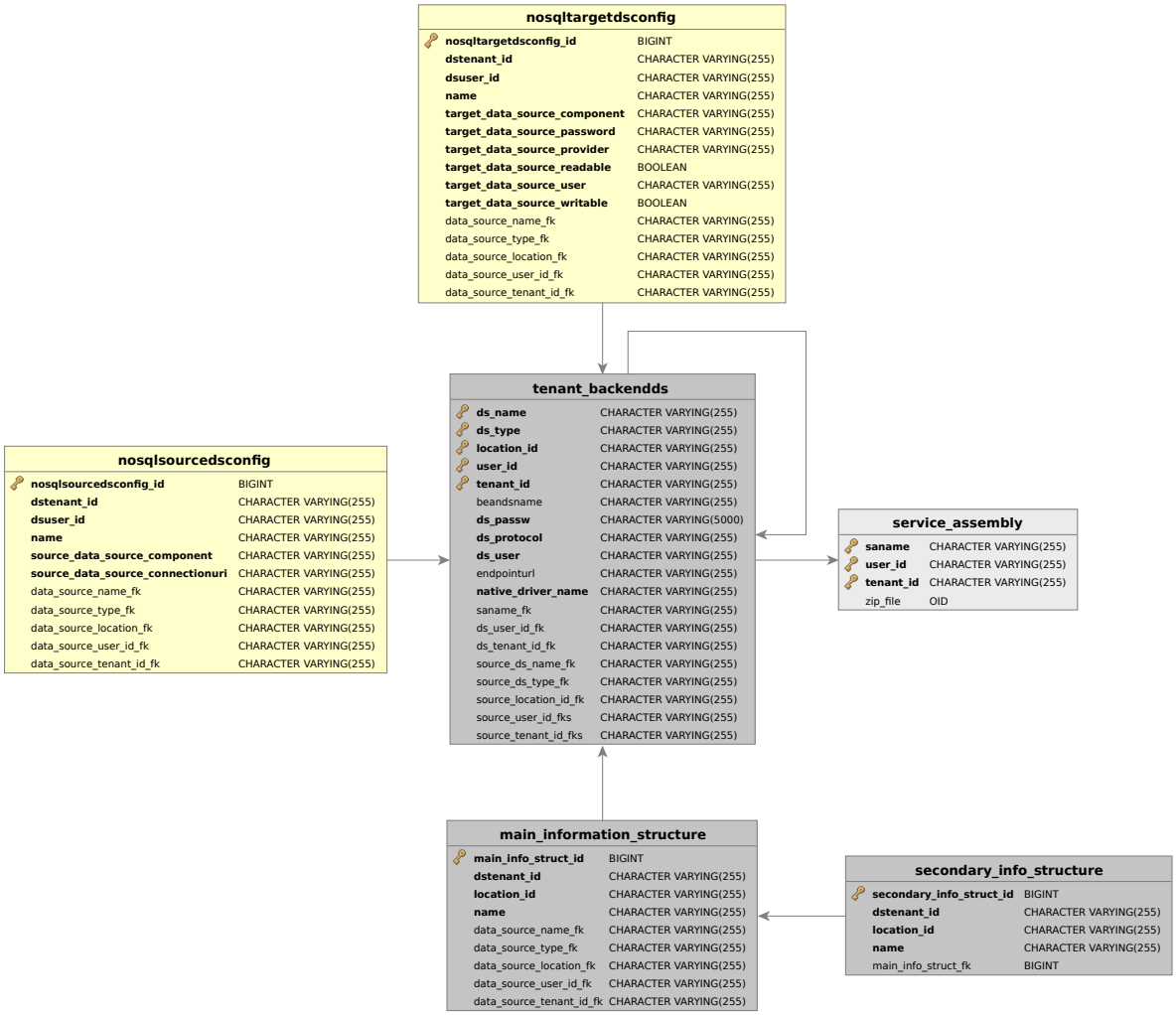


Figure 6.1.: Modified Service Registry Database

6.3. ServiceMix Components

The ServiceMix components are new implementations, with no code reuse from JBIMulti2 or CDASMix. The Unified Blobstore component is pure OSGi without any JBI components, therefore the in CDASMix required connection to service units is not necessary.

Due to the version restrictions listed in Table 6.1, it was not possible to use the latest version of ServiceMix, Camel and JClouds. Thus getting help from the user mailing list is complicated if the concerning version is deprecated.

The components are not equally well-implemented, some offer only limited functionality or have errors in documentation[4] and implementation[5].

The best practice for the implementation using Camel components is to read the description of the components on the website[6], download the source code, check the examples and the JUnit test cases, and then examine the behaviour using a debugger once the component conducts unexpectedly.

ServiceMix can be started "out of the box" with a debug parameter and waits for connection on port 5005 for debugging with eclipse. Source code of the libraries and their dependencies can be obtained using maven in the pom.xml configuration, as shown in Listing 6.4.

```
1 <plugin>
2   <groupId>org.apache.maven.plugins</groupId>
3   <artifactId>maven-eclipse-plugin</artifactId>
4   <configuration>
5     <downloadSources>true</downloadSources>
6     <downloadJavadocs>true</downloadJavadocs>
7   </configuration>
8 </plugin>
```

Listing 6.4: Maven Snippet to Configure Autmatic Source Code and Javadoc

6.3.1. NoSQL Registry

The NoSQL registry abstracts the access to the NoSQL configurations.

It is composed of two parts: an NoSQL Registry interface that defines the access methods and implements the back end independent functions, and the NoSQL Registry OSGi component. The NoSQL Registry registers a reference of itself in the context registry of the OSGi container manager with the implemented interface, as shown in Listing 6.5.

This reference can be accessed by the Unified Blobstore end every other Unified NoSQL components.

[4] http://camel.465427.n5.nabble.com/-tp5750789.html
[5] http://camel.465427.n5.nabble.com/-td5757810.html
[6] http://camel.apache.org/components.html

The OSGi implementation uses the same classes and Hibernate configuration as JBIMulti2 to access the database.

Connection to Hibernate

Using applications that were not implemented for OSGi in an OSGi environment can be tricky. Each bundle has an associated classloader that is responsible for loading the classes inside the bundle [CW13, p. 16].

Hibernate uses the classloader to access the configuration file containing the configuration for the connection as well as the path to the classes representing the database content.

OSGi modules started by ServiceMix have the classloader of ServiceMix configured, which means that Hibernate is unable to load the configurations from the expected location inside the OSGi component, but will instead look in a directory of ServiceMix.

Listing 6.6 shows the workaround[7]. The classloader has to be temporarily set to the classloader of the current class, then Hibernate is able to find the configurations.

After this modification, Hibernate works as usual.

Matching

Listing 6.7 shows the matching check. First step is to validate that the input matches the information structure by matching the container name against the main information structure and the blob name against the secondary information structure. If the information structure is completely or partially based on a default type, the other "competing" information structures are checked. If none of them matches and has a higher priority (no replication involving default), the information structure "matches".

This method is later used in the camel routes to configure router and filter.

6.3.2. Camel-Jclouds

Camel-Jclouds is the component of the unified datastore that establishes the connection between the ESB and the blobstore cloud provider.

[7] `http://apache-felix.18485.x6.nabble.com/-td4835872.html`

```
1 public void start(BundleContext context) throws Exception {
2   registry = Registry.getInstance();
3   reg = context.registerService(IRegistry.class.getName(), registry, null);
4 }
```

Listing 6.5: Registration in the Activator Using the Interface

```
1 ClassLoader cl = Thread.currentThread().getContextClassLoader();
2 Thread.currentThread().setContextClassLoader(this.getClass().getClassLoader());
3
4 emf = Persistence.createEntityManagerFactory("serviceRegistry");
5 entityManager = emf.createEntityManager();
6
7 Thread.currentThread().setContextClassLoader(cl);
```

Listing 6.6: Workaround for Hibernate in an OSGi Environment

```
1  public boolean matches(String containername, String blobname) {
2    boolean matches = matchesIsolated(containername, blobname);
3
4    if(isDefault() && matches) {
5      // Only match if no other information structure with a higher priority matches.
6      // ABSOLUTE and REGEX > mis ABSULUTE or REGEX and sis DEFAULT > mis and sis DEFAULT
7      for (InformationStructure is : competingInformationStructures) {
8        if(is.priority > priority && is.matchesIsolated(containername, blobname) ) {
9          matches = false;
10         break;
11       }
12     }
13   }
14   return matches;
15 }
```

Listing 6.7: Matching of Information Structures

Delete Implementation

In the current version, Camel Jclouds only supports read and write operations, no deletions. To offer the full functionality as described in Section 4.2, we extended the component.

Like every Camel component, the main function of the Camel-Jclouds component is the mapping of messages to the API of the implemented application. The code required to to add mapping for delete operations can be seen in Listing 6.8.

Other functionality, like listing existing container and blobs, can be added like that.

```
1 } else if (JcloudsConstants.DELETE.equals(operation)) {
2   blobStore.removeBlob(container, blobName);
3 }
```

Listing 6.8: Required Modifications to Camel-Jclouds to add Delete

StackOverflowError Workaround

Camel Jclouds has a bug in the used version that prevents the processing of messages of a specific type and larger than a certain threshold[8]. The bug will be fixed in the next releases[9], but since we are bound in the development to a version that is no longer actively developed, we had to implement a workaround.

As workaround, we deactivated caching and added a processor that extracts the message and stores it in a temporary file. The behavior is almost the same as the default caching, but since the format of the message is a different the StackOverflowError is not triggered.

If the implementation is run using a Camel version 2.13.3, 2.14.1 or 2.15.0 or later, the workaround can be removed.

6.3.3. Unified Blobstore

The Unified Blobstore component provides the entry point for the connections from the java access library and manages the routing to the target blobstores. Like the NoSQL registry, it is a OSGi component.

On startup, the activator accesses the NoSQL Registry to load the source and target data store configurations and the information structures.

The component was implemented after the design in Section 5.4. Blobs can be very huge, performing the read and write operations can thus take much longer than for any other SQL and NoSQL request.

[8] http://camel.465427.n5.nabble.com/-td5757810.html
[9] http://camel.465427.n5.nabble.com/-td5757810.html

The maximum idle time of the Jetty component must be set to a value that cannot be exceeded in normal usage, as shown in Listing 6.9.

```
List<String> tenants = registry.getNoSQLTenants();
for (String tenant : tenants) {
  CamelContext camelContext = camelContextFactory.createContext();

  // Increase maxIdleTime of Jetty, otherwise the request will be resent to the following components after 200 seconds
  // This has been fixed in later versions of camel.
    JettyHttpComponent jettyComponent = camelContext.getComponent("jetty", JettyHttpComponent.class);
    jettyComponent.addSocketConnectorProperty("maxIdleTime", (60 * 60 * 1000));

    List<INoSQLSourceDataStore> sourceDataStores = registry.getNoSQLSourceDataStores(tenant);

    RouteBuilder sourceRouter = new TenantEntryRouteBuilder(sourceDataStores);
  camelContext.addRoutes(sourceRouter);
  for (INoSQLSourceDataStore sourceDataStore : sourceDataStores) {
    RouteBuilder route = new NoSQLDataRouteBuilder(sourceDataStore);
    camelContext.addRoutes(route);
  }
  camelContext.start();
  camel.put(tenant, camelContext);
}
```

Listing 6.9: Configuring the Camel Contexts for Each Tenant

The message translator that transforms the message from the uniform format to the format of the target message endpoint only needs to rename the header to the format required by the Camel-Jclouds component.

6.4. Java Access Library

The Java library can be included by the end-user to connect to the Unified Blobstore component. It provides a simple interface to write, read, and delete blobs. The interface can be seen in Listing 6.11.

```
public interface BlobStore {
  public void put(File f, String container, String blob) throws UnifiedBlobstoreException;
  public void get(File f, String container, String blob) throws UnifiedBlobstoreException;
  public void delete(String container, String blob) throws UnifiedBlobstoreException;

  public String getName();
}
```

Listing 6.11: Methods Provided by the Unified NoSQL Library to Access Blobstores

```
# Operation
cdasmix-operation=CamelJcloudsOperation
cdasmix-operation.PUT=CamelJcloudsPut
cdasmix-operation.GET=CamelJcloudsGet
cdasmix-operation.DELETE=CamelJcloudsDelete

# Name of the Container / Bucket. Used in the Main Information Structure.
cdasmix-container=CamelJcloudsContainerName

# Name of the Blob / File. Used in the Secondary Information Structure.
cdasmix-blob=CamelJcloudsBlobName
```

Listing 6.10: Mapping Between the Unified Format and the Camel-Jclouds Format

The Java Blobstore Library a simple HTTP client that uses the Apache Httpclient library to establish the connection. Required message parameters are set as message header. Listing 6.12 shows as an example the HTTP request to write a blob.

```
  HttpPost httppost = new HttpPost(connectionURL);
  httppost.setConfig(config);

  httppost.setHeader(Constants.CDASMIX_SOURCE_DATA_STORE, sourceDataStore);
  httppost.setHeader(Constants.CDASMIX_OPERATION, Constants.CDASMIX_PUT);
  httppost.setHeader(Constants.CDASMIX_CONTAINTER, container);
  httppost.setHeader(Constants.CDASMIX_BLOB, blob);

  httppost.setHeader(Constants.TENANT_UUID, tenantUUID);
  httppost.setHeader(Constants.USER_UUID, userUUID);

  FileEntity reqEntity = new FileEntity(file);
  reqEntity.setChunked(true);
  httppost.setEntity(reqEntity);
  CloseableHttpResponse response = httpclient.execute(httppost);
```

Listing 6.12: Excerpt of the Write Operation in the NoSQL Library

6.5. Validation

We evaluate the modifications to JBIMulti2 and the implementation of the NoSQL Registry and the Unified Blobstore by configuring and using an extensive use case scenario. The evaluation takes place in a single virtual machine.

The scenario create has three tenants with one user each T1U1, T2U1 and T3U1.

Tenant T1U1 is used to validate default configurations, T2U1 is used for the validation of

content dependent routing, and replication is validated by the configurations for tenant T3U1.

The added configurations can be seen in Table 6.2.

Tenant	SDS[10]	TDS[11]	IS[12] in the format (MIS/SIS)[13], regular expressions are not used.
T1U1	sds1	tds1s3	(«default»/«default»)
T1U1	sds1	tds2sAzure	('container1'/«default»)
T1U1	sds1	tds1s3	('container1'/'s3.txt')
T2U1	sds2	tds3s3	('container2'/'hello1.txt'), ('container2'/'hello2.txt')
T2U1	sds2	tds4Azure	('container2'/'hello3.txt'), ('container2'/'hello4.txt')
T3U1	sds3	tds5s3	('cont3'/'a.txt'), ('cont4'/«default»)
T3U2	sds3	tds6Azure	('cont3'/'a.txt'), ('cont4'/«default»), («default»/«default»)

Table 6.2.: NoSQL Data Store Configurations of the Evaluation

In the first step, we validate that the configuration were correctly entered in the PostgreSQL database. We start the NoSQL Registry and the Unified Blobstore component. The component has a debugging function that writes the content of the NoSQL Registry to the console on startup, thereby making sure that the NoSQL Registry is working correctly.

In the second step, we send write requests to through the Unified NoSQL component and validate that the correct target data store is selected.

Tenant	SDS	Container	Blob name	S3	Azure
T1U1	sds1	container0	example.txt	✗	✓
T1U1	sds1	container1	example.txt	✓	✗
T1U1	sds1	container1	s3.txt	✗	✓
T2U1	sds2	container2	hello1.txt	✓	✗
T2U1	sds2	container2	hello2.txt	✓	✗
T2U1	sds2	container2	hello3.txt	✗	✓
T2U1	sds2	container2	hello4.txt	✗	✓
T3U1	sds3	cont3	a.txt	✓	✓
T3U2	sds3	cont4	something.txt	✓	✓
T3U2	sds3	something	something.txt	✗	✓

Table 6.3.: NoSQL Data Store Configurations of the Evaluation

As seen in the first three requests sketched in Table 6.3, the priority works as defined. The

[10]SDS: Source Data Store
[11]TDS: Target Data Store
[12]IS: Information Structures
[13](Main Information Structure/Secondary Information Structure). Type default use the keyword «default», type absolute are in quotation marks

absolute configuration is preferred to configuration that is partially of the type default. The completely default configuration has the lowest priority and is only selected if no other information structure can be used.

The second block proves that it is possible to define explicit routes, separating the message based on predefined rules and sending them to the correct target data store.

Replication is validated by the remaining requests, both for default configurations as well as information structures of the type absolute.

7. Evaluation

This chapter documents the performance of the created blobstore prototype. The evaluation is conducted from the perspective of the end-user with focus on the performance difference introduced by routing through a middleware and by using a different driver to establish the connection.

The performance test measures the time (in ms) needed for the execution of a specific request with focus on the difference between a direct connection using the driver provided by the vendor, the direct connection using the Jclouds driver and the connection through the ESB middleware using the Java access library.

7.1. Execution Environment

The test environment is deployed in a single Virtual Machine (VM) running on Ubuntu 10.04.The resources assigned to the VM are four processors (3.4 GHz) and four GB of dedicated memory. During the execution the processor and memory usage is monitored and is ensured that processor usage on any processor never exceeds 60 percent and that the memory usage of never exceeds 3 GB.

The VM is connected to the internet with a connection allowing up to 50 MBit/s download and 2,5 MBit/s upload speed.

7.1.1. Preparation

Before the first test run we created both, the accounts with the target data store provider and the used container. We used the default configuration for each step.

ServiceMix, PostgreSQL and JOnAS with the modified JBIMulti2 management interface are installed and configured according to the manual. The performance test scenario consists of one tenant with one user. One souce data store is configured to route all requests with the container ‘amazononly” to the provider S3, and all requests to the container “azureonly” zu Azure Blob.

After these configurations the JOnAS server was closed to free resources and ServiceMix with the Unified Blobstore component was started.

7.2. Test Program

There are no sufficient benchmark programs available for blobstores. Yahoo offers a general framework for NoSQL benchmark tests named Yahoo Cloud Serving Benchmark (YCSB), but drivers for blobstores are currently not part of the implementation. JClouds has in its repository a very basic performance test that they use to compare the native driver with the Jclouds driver by repeatedly uploading a single file to the cloud in its repository[1].

The JClouds performance test has been used as a reference for the implementation of a custom benchmarking tool that can be used to compare the upload times using several drivers and several providers. The implementation creates a simple wrapper class for each driver that directs the read, write and delete requests to the driver implementation. Requests to the blobstore driver are by design self-sufficient, therefore opening and closing connections was done by the driver.

```
1  @Override
2  public void put(File file, String container, String blob) throws Exception {
3    CloudBlobContainer c = blobClient.getContainerReference(container);
4    CloudBlockBlob b = c.getBlockBlobReference(blob);
5    b.upload(new FileInputStream(file), file.length());
6  }
7  @Override
8  public void get(File file, String container, String blob) throws Exception {
9    CloudBlobContainer c = blobClient.getContainerReference(container);
10   CloudBlockBlob b = c.getBlockBlobReference(blob);
11   b.download(new FileOutputStream(file));
12 }
13 [...]
```

Listing 7.1: Wrapper for the Azure Driver

```
1  Blobstore;File;Size;Operation;Start;End;Result;Comment
2  Amazon AWS S3;a-1kB.bin;1024;put;1415341327332;1415341328211;879;ok
3  Amazon AWS S3;a-1kB.bin;1024;put;1415341328211;1415341329092;881;ok
4  [...]
5  UnifiedBlobstore;c-100kB.bin;102400;get;1415343588922;1415343591421;2499;ok
6  UnifiedBlobstore;c-100kB.bin;102400;get;1415343591422;1415343593920;2498;ok
7  UnifiedBlobstore;c-100kB.bin;102400;get;1415343593921;1415343597251;3330;ok
8  [...]
```

Listing 7.2: Output of the Test Evaluation Program

[1] https://github.com/jclouds

The evaluation compares the native drivers (aws-java-sdk and auzre-storage) with a direct connection using the Jclouds-blobstore driver and the connection through the ServiceMix middleware using the unified-nsql-library. The native drivers were used in the latest version as provided by the vendor Software Development Kit (SDK) (aws-java-sdk 1.9.3 and azure-storage 1.3.1), while the Jclouds driver was used in the same version that is used by the Camel-Jclouds component (Jclouds-blobstore 1.4.0).

The test program performed each operation repeatedly and measured the system time before and after every request. The results were stored in a Character Separated Values (CSV) file and later analyzed using LibreOffice.

7.2.1. External Influences

There are several performance factors of a connection to a cloud NoSQL database that cannot be influenced or predicted by the end-user like the workload of other users on the same network and server or internal details of the management by the cloud provider. Even if the client and the provider are in the same network performance fluctuation can occur.

The article "Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance" for example measures a covariance of 54 per cent during measurements for upload from an EC2 machine directly to S3 [SDQR10].

To get to a meaningful comparisons between the drivers it was attempted to keep the external influences as constant as possible. The test scenarios were executed in sequential order. Long running test scenarios were split into smaller parts to effect every driver measurement in the same way.

7.2.2. Workload

The workload consists of files between 1 KB and 100 MB. To prevent the possibility of compression, each file was filled with random data. File names have been changed every time to prevent caching.

The files with the sizes 1 KB, 10 KB, 100 KB, 1 MB have been sent to, read, and then deleted 100 times using every combination of driver and provider. The test has been repeated 50 times for 10 MB Files and 8 times for 100 MB file. The last workload was only sent to the provider Amazon.

Files larger than 100 MB had been sent during preparations, but were excluded from the evaluation once it became apparent that the only measurable difference between the test runs was the bandwidth performance and not the driver performance.

7.3. Execution

7.3.1. Warm-Up

The ServiceMix component and the provider are prepared in separate steps.

The ESB warm-up was oriented on the warm-up in the performance tests of the previous ServiceMix components. To reach every component, we sent 400 requests to the Jetty component that were routed to a dummy endpoint.

Measurements during the preparation showed that only the first operation, and sometimes the second took significantly longer than all the following operations. As provider warm-up, a 1 KB file was sent, read and deleted using the driver that was about to get tested 4 times.

7.3.2. Test Run

The tests for each provider were run separately, execution of the workload as described in Section 7.2.2 for a single driver took between four minutes (1 KB) and two hours (10 MB). To keep external conditions as as described in Section 7.2.1 as constant as possible, the 100MB workload file was sent once to each driver, this step has been repeated eight times. The aim was to distribute time dependent fluctuations equally to the measurements of all drivers.

As measurement, the system time was taken before and after every request and stored in a CSV file that was later imported in LibreOffice.

7.3.3. Result Validation

Results were consistent, the results during the actual measurements and runs during the configuration process where the same. Larger files can be sent but measurements have been excluded due to the fact that it can be expected that the only influential factor for differences between the measurements was the workload of network and provider hardware of other users.

Most of the measurements only differ in a minor percentage. There are, however, some requests that take up to tree times the median time to complete. Since these spikes occurred equally with every driver, it is assumed that they are caused by internal factors of the provider or the connection.

7.4. Results

During the test phase, all requests were fulfilled and no connection errors occurred. The measurements are commented in the following paragraphs itemized by request type.

7.4.1. Delete

As seen in Figure 7.1 and Figure 7.2, the delete operation is not influenced by the size of the corresponding file.

The performance of the amazon SDK is almost identical to the performance using the Java access library. On average, the request time for a delete operation as shown in Figure 7.1 and the corresponding Table 7.1 is between 456 ms and 486 ms with all drivers. The measurements indicates that the JClouds driver and the native driver have almost the same performance, and the overhead of the Unified Blobstore is barely measurable.

The native driver provided by the provider azure as shown in Figure 7.2 on the other hand is about 2.5 times faster than the Jclouds driver and the unified driver. Since the unified NoSQL blobstore driver uses the Jclouds driver, it is safe to assume that the overhead is caused by the Jclouds driver implementation, and not by other components of the middleware.

The slightly better results of the unified driver compared to the Jclouds driver can be attributed to a few outliers, as the median as shown in Table 7.2 is almost identical.

	Native Driver				**Jclouds Driver**				**Unified NoSQL**			
	avg	**med**	**min**	**max**	**avg**	**med**	**min**	**max**	**avg**	**med**	**min**	**max**
1 KB	460	450	444	1327	456	448	410	1267	463	457	419	1104
10 KB	461	452	446	1271	458	451	439	1070	471	460	422	1305
100 KB	460	451	439	1106	456	448	408	1080	475	462	433	1352
1 MB	461	450	413	1608	460	451	412	1130	477	462	423	1327
10 MB	471	452	446	1326	458	446	431	1078	465	455	445	889
100 MB	465	466	451	481	449	456	415	460	480	486	426	498

Table 7.1.: Delete Request Sent to Provider AWS S3 (in ms)

	Native Driver				**Jclouds Driver**				**Unified NoSQL**			
	avg	**med**	**min**	**max**	**avg**	**med**	**min**	**max**	**avg**	**med**	**min**	**max**
1 KB	238	236	231	343	622	600	578	1448	615	601	591	948
10 KB	246	235	231	630	617	600	569	1325	615	600	588	1058
100 KB	231	225	222	727	610	600	556	953	618	601	558	1209
1 MB	242	235	231	540	636	600	585	1498	622	601	563	1259
10 MB	260	232	227	1250	606	601	582	690	627	602	582	1196

Table 7.2.: Delete Request Sent to Provider Azure (in ms)

7.4.2. Read

Figures 7.3 and 7.4 show the measurements for read operations on a logarithmic scale. Read performance of the S3 driver and the Jclouds driver is almost identical, while the Unified Blobstore drivers is significantly slower.

The time needed to read a small file using Unified Blobstore is on average twice as long as it is to read using the S3 or JClouds driver. This trend is also present with operations that use

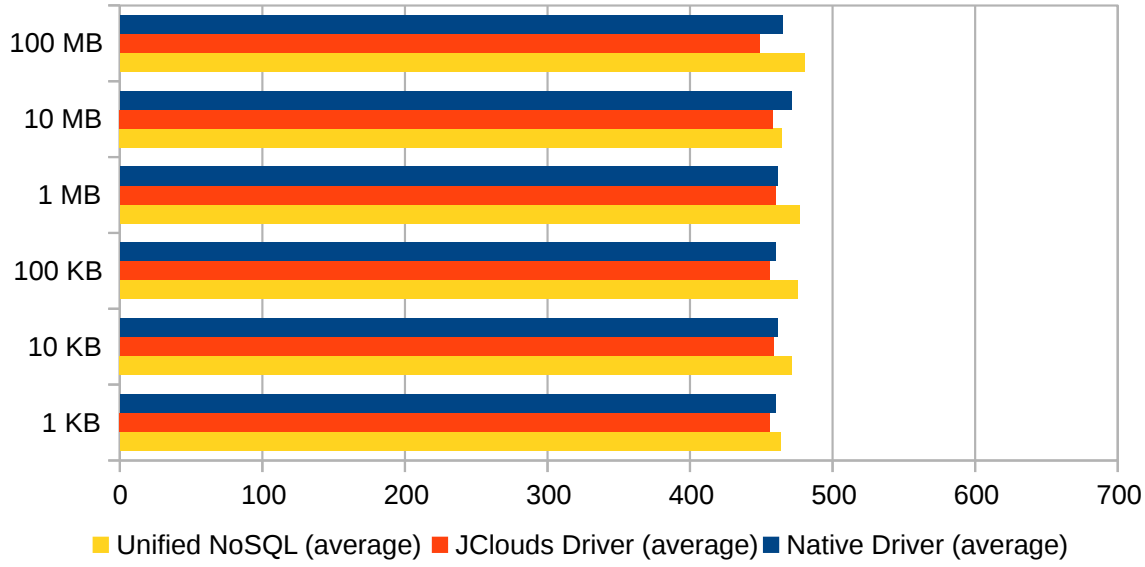


Figure 7.1.: Delete Measurements for Provider AWS S3 (Table 7.1)

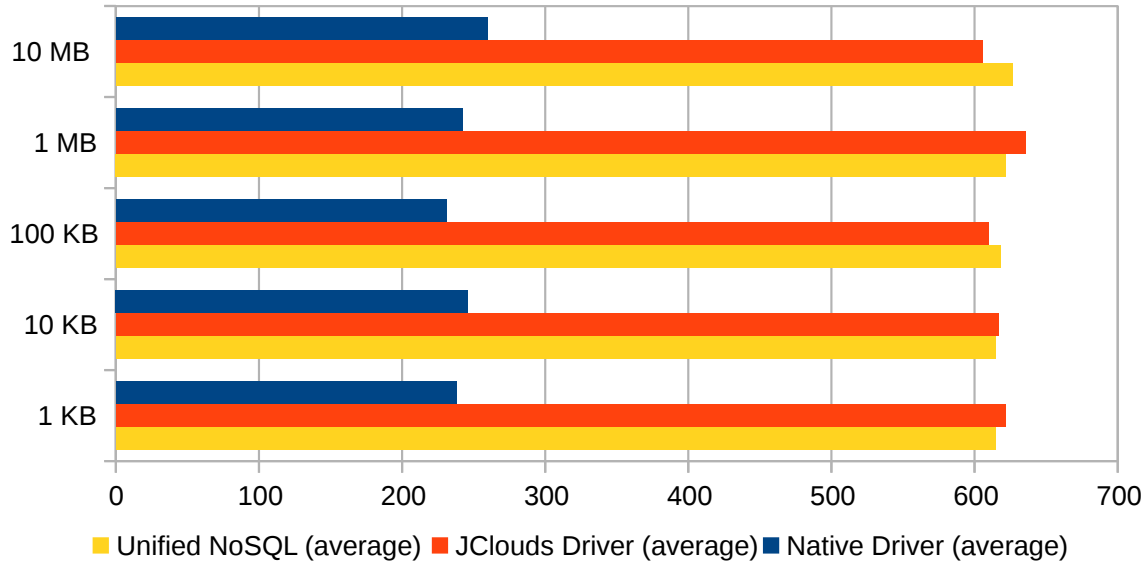


Figure 7.2.: Delete Measurements for Provider Azure (Table 7.2)

large files, even though the difference reduces to around 20 percent[2].

In contrast, receiving files using JClouds takes considerably longer compared to Azure Blob, as shown in Figure 7.4.

Reading small files as shown in Table 7.3 takes about 400 ms longer using the JClouds driver, and an additional second using the Unified Blobstore driver. The effect on larger files still measurable, with JClouds being about 10 percent slower than the native driver and Unified Blobstore being 15 percent slower than the native driver.

	Native Driver				**Jclouds Driver**				**Unified NoSQL**			
	avg	**med**	**min**	**max**	**avg**	**med**	**min**	**max**	**avg**	**med**	**min**	**max**
1 KB	248	237	232	758	645	601	566	2140	1579	1313	1158	2897
10 KB	244	237	232	690	662	601	572	2186	1753	1519	1180	3165
100 KB	521	458	230	1379	1018	923	568	2165	2382	2380	1226	3817
1 MB	3695	3480	1675	9225	3560	3413	1758	6135	5285	4990	2711	11631
10 MB	26822	26185	16409	37376	29746	28766	19895	44941	30407	30125	23446	47249

Table 7.3.: Read Request Sent to Provider Azure (in ms)

	Native Driver				**Jclouds Driver**				**Unified NoSQL**			
	avg	**med**	**min**	**max**	**avg**	**med**	**min**	**max**	**avg**	**med**	**min**	**max**
1 KB	462	453	445	1310	478	441	403	1805	1128	905	849	2113
10 KB	597	627	433	1496	470	449	411	1716	1191	1090	863	2833
100 KB	1384	1422	805	2611	1048	1044	611	1821	2622	2507	1737	4706
1 MB	3913	3838	1841	9388	4471	4345	1851	7980	5821	5469	3908	10746
10 MB	30905	27903	18909	71433	27823	24457	16866	62155	31745	30252	22108	67534
100 MB	283656	223874	170777	671048	247178	219681	158692	372518	277097	276835	234597	337946

Table 7.4.: Read Request Sent to Provider AWS S3 (in ms)

[2]Using formula ($|V_1\text{-}V_2| / ((V_1 + V_2)/2)$) * 100, rounded to nearest multiple of five

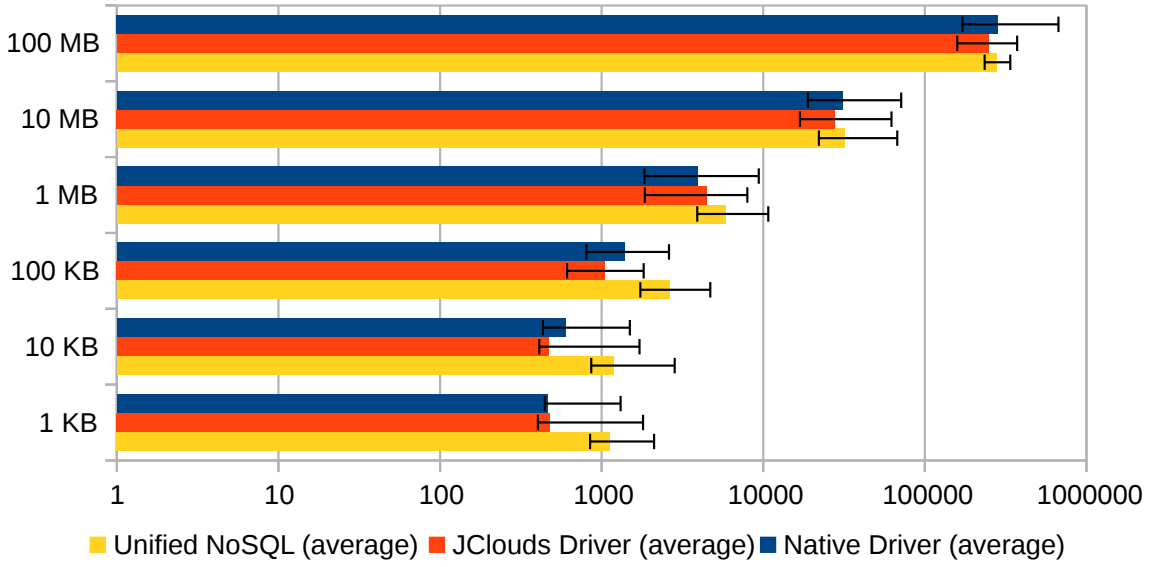


Figure 7.3.: Read Measurements for Provider AWS S3 (Table 7.4)

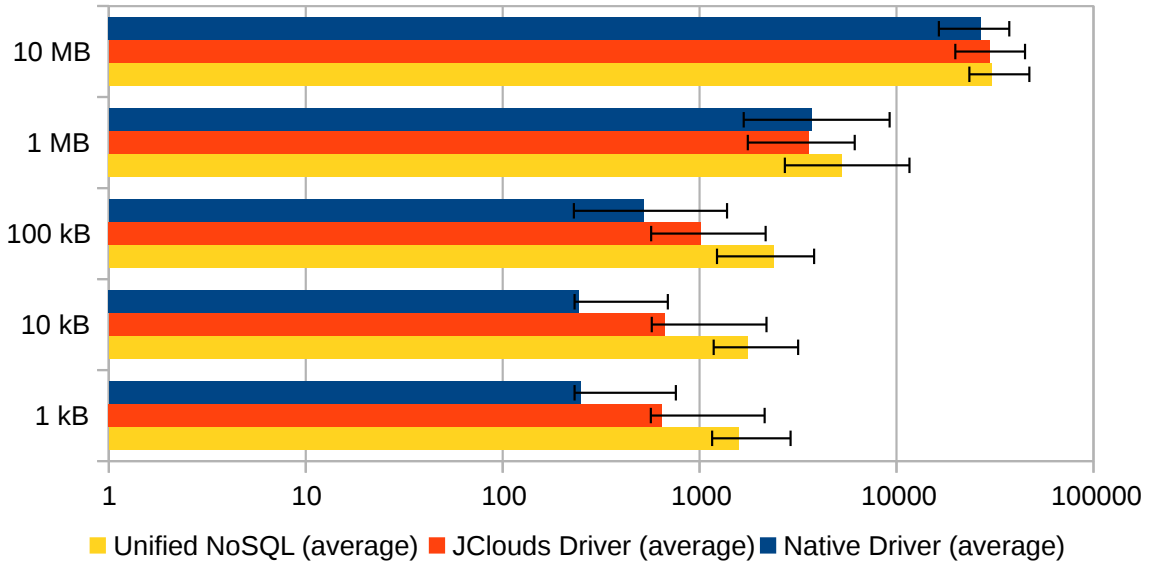


Figure 7.4.: Read Measurements for Provider Azure (Table 7.3)

7.4.3. Write

The outcome of the write test operation are the most interesting. As seen in Figures 7.5 and 7.6, sending smaller files to the provider using the Unified Blobstore driver can be even faster than sending it via the native driver, for both S3 and Azure Blob.

For bigger files, the measurements adjust to similar values, and the native drivers seem to become faster than the Jclouds and the unified driver, but this might also be due to the limited measurements that were performed for huge files.

Reason might be that the native driver performs additional steps for advanced error handling, e.g. S3 calculates MD5 hash value before the transmission.

	Native Driver				Jclouds Driver				Unified NoSQL			
	avg	**med**	**min**	**max**	**avg**	**med**	**min**	**max**	**avg**	**med**	**min**	**max**
1 KB	885	871	862	1768	238	218	212	1450	252	233	226	1269
10 KB	865	855	827	1830	434	416	407	1503	463	439	412	1522
100 KB	1415	1397	1182	2624	992	940	740	2370	992	953	757	2211
1 MB	8334	8234	7855	9780	7925	7777	7259	9810	8036	7903	7108	10237
10 MB	76623	76916	67103	82827	77153	76987	70390	92211	76251	76257	69820	82151
100 MB	742663	733505	681109	906160	739534	744264	707629	769204	765534	761962	743388	793014

Table 7.5.: Write Request Sent to Provider AWS S3 (in ms)

	Native Driver				Jclouds Driver				Unified NoSQL			
	avg	**med**	**min**	**max**	**avg**	**med**	**min**	**max**	**avg**	**med**	**min**	**max**
1 KB	1251	1202	1180	1838	256	244	236	1113	266	257	249	867
10 KB	1121	1100	1033	2093	488	472	464	1205	487	482	470	688
100 KB	1642	1604	1374	2099	1043	1023	794	1557	1075	1074	801	1445
1 MB	9416	9393	7592	11304	9064	8992	7698	11170	8986	8957	6776	10704
10 MB	86142	87164	73902	91526	87045	88154	60041	91696	92415	89957	70350	121837

Table 7.6.: Write Request Sent to Provider Azure (in ms)

7.4.4. Conclusion

The overview in Figure 7.7 and 7.8 shows the average time for the three supported normed to the performance of the native driver, shown as green line with 100 percent.

It shows that for both provider S3 and Azure Blob, the performance advantages and disadvantages are most important for small files. The difference becomes negligible when the transferred file is large.

Most of the difference is caused by the difference between Jclouds and the native driver, not by the overhead introduced by the middleware.

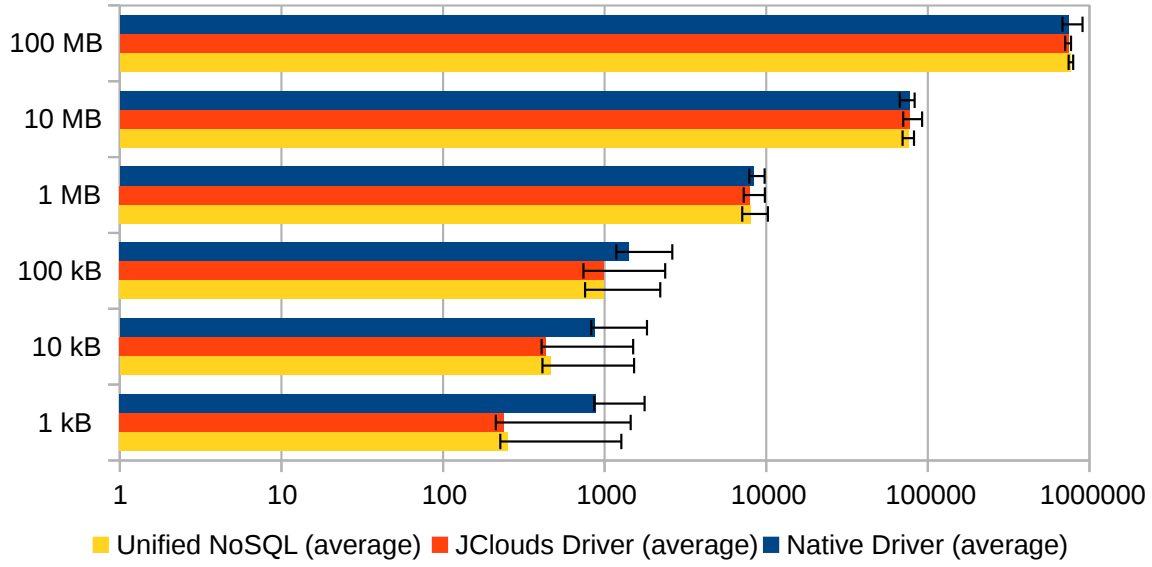


Figure 7.5.: Write Measurements for Provider Azure (Table 7.5)

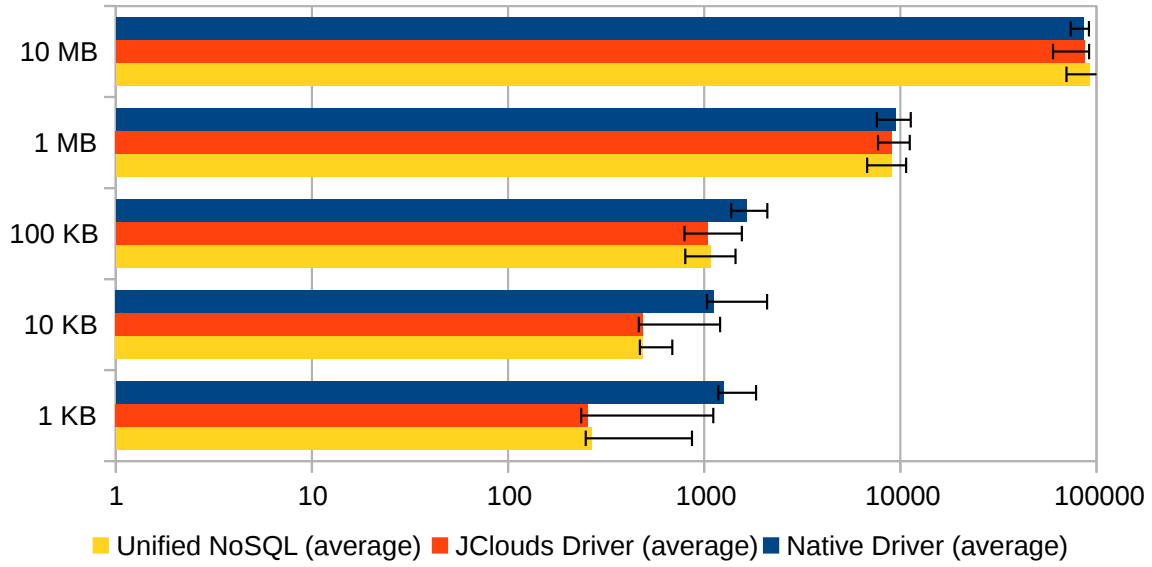


Figure 7.6.: Write Measurements for Provider Azure (Table 7.6)

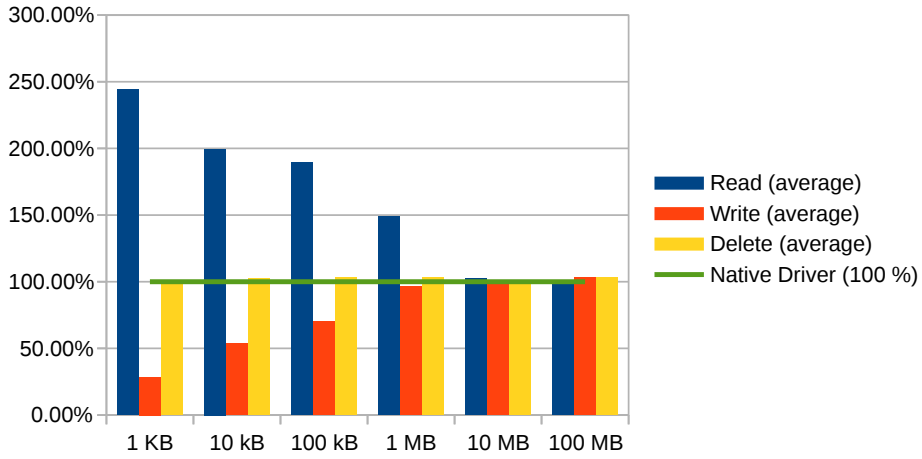


Figure 7.7.: Compact Overview of the Amazon Results

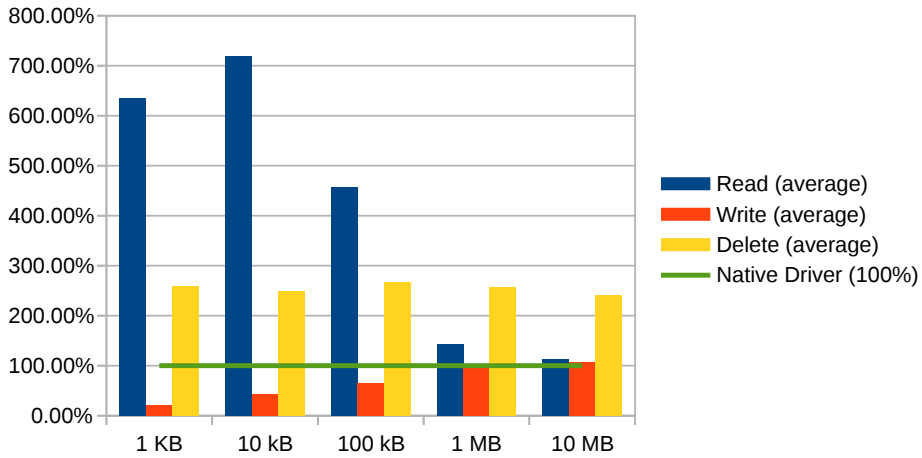


Figure 7.8.: Compact Overview of the Azure Results.

8. Conclusion and Future Work

In the last chapter, we recap the findings of the thesis and go into the possibilities to extend the functionality.

8.1. Conclusion

In this thesis, we were able to modify the existing JBIMulti2 management application to support the registration of data required for the Unified NoSQL design.

We compared different possibilities of implementing an OSGi container that is able to to manage the data access layer as part of a multi-tenant aware ESB with CDASMix extensions. We decided to use the EIP pattern support of Camel for the implementation. Thus, we were able to create an easily portable design that can be adapted for other NoSQL data store types.

Then we implemented the design by the Unified Blobstore component which provides access to the blob storage provider Amazon S3 and Azure Blob Storage.

We evaluated the performance and compared the results to the performance of direct connections using the providers native SDKs and JClouds. We proved that the performance loss introduced by routing through the ESB is manageable and in most cases smaller than the difference introduced by using a different connection library.

8.2. Future Work

In the limited time frame of this thesis, we were unable to implement and verify every possible use case that occurred during the specification. In this section, we will describe briefly the possibilities to extend this implementation for additional use case scenarios including the required modifications.

8.2.1. Support for other NoSQL Data Stores

In the current implementation, JBIMulti2 can be used to add the configuration universally for every NoSQL data store type. To add a new NoSQL data store implementation, the first step is to select or implement the Camel component that produces the message endpoint for the NoSQL database connection. A full list of the existing endpoints can be found on the Camel

website[1]. If the required component does not exist or does not provide the required functions and needs to be implemented, it might be possible to add the component to the official Camel repository[2]. The step is to design unified message model that can be transformed in the format that the NoSQL data stores require. Last, the message translator needs to be implemented.

8.2.2. Performance Isolation

Performance isolation is currently not implemented. Camel supports the Throttler Pattern[3], that can be used to limit the throughput on message paths. The parameter can be added to the source data store configuration, thus, a throttler can be used to ensure an upper limit of throughput and utilization for each tenant, because it is configurable per Camel route.

8.2.3. Camel Route Update at Runtime

Each tenant has its own Camel context, that can be stopped, edited, and started independently from the other tenants. To implement modification updates without restarting the Unified Blobstore component, we would have to implement a mechanism that detects route changes, then stops the concerning context, and restarts it with the updated configurations.

[1] http://camel.apache.org/components.html
[2] http://camel.apache.org/contributing.html
[3] http://camel.apache.org/throttler.html

Appendix A.

Sourcecode

A.1. jbimulti2.wsdl

Extracts of modifications of the the jbimulti2.wsdl file.

A.1.1. attachNoSQLDataSource

```
1
2  <xs:element name="attachNoSQLDataSource">
3   <xs:complexType>
4    <xs:sequence>
5     <xs:element name="sourceDataSourceName" type="xs:string"/>
6     <xs:element name="sourceDataSourceType" type="xs:string">/>
7     <xs:element name="sourceDataSourceComponent" type="xs:string" default="camel-jetty"/>
8     <xs:element name="sourceDataSourceConnectionURI" type="xs:string"/>
9     <xs:element name="targetDataSourceName" type="xs:string"/>
10    <xs:element name="targetDataSourceType" type="xs:string"/>
11    <xs:element name="targetDataSourceComponent" type="xs:string"/>
12    <xs:element name="targetDataSourceProvider" type="xs:string"/>
13    <xs:element name="targetDataSourceUser" type="xs:string"/>
14    <xs:element name="targetDataSourcePassword" type="xs:string"/>
15    <xs:element name="targetDataSourceReadable" type="xs:boolean" default="true"/>
16    <xs:element name="targetDataSourceWritable" type="xs:boolean" default="true"/>
17   </xs:sequence>
18  </xs:complexType>
19 </xs:element>
```

Listing A.1: Request to add a new NoSQL Data Store

A.1.2. attachNoSQLTargetDataSource

```
1
2  <xs:complexType name="NoSQLTargetDataSource">
3   <xs:sequence>
4    <xs:element name="targetDataSourceName" type="xs:string" minOccurs="1" maxOccurs="1"/>
5    <xs:element name="targetDataSourceType" type="xs:string" minOccurs="1" maxOccurs="1"/>
```

```
6     <xs:element name="targetDataSourceComponent" type="xs:string" minOccurs="1"
7       maxOccurs="1"/>
8     <xs:element name="targetDataSourceProvider" type="xs:string" minOccurs="1" maxOccurs="1"/>
9     <xs:element name="targetDataSourceUser" type="xs:string" minOccurs="1" maxOccurs="1"/>
10    <xs:element name="targetDataSourcePassword" type="xs:string" minOccurs="1" maxOccurs="1"/>
11    <xs:element name="targetDataSourceReadable" type="xs:boolean" default="true"/>
12    <xs:element name="targetDataSourceWritable" type="xs:boolean" default="true"/>
13   </xs:sequence>
14  </xs:complexType>
15  <xs:element name="attachNoSQLTargetDataSource">
16   <xs:complexType>
17    <xs:sequence>
18     <xs:element name="sourceDataSourceName" type="xs:string" minOccurs="1" maxOccurs="1"/>
19     <xs:element name="sourceDataSourceType" type="xs:string" minOccurs="1" maxOccurs="1"/>
20     <xs:element name="targetDataSources" type="tns:NoSQLTargetDataSource"
21       maxOccurs="unbounded"/>
22    </xs:sequence>
23   </xs:complexType>
24  </xs:element>
```

Listing A.2: Request to Attach a NoSQL Target Data Store

A.2. NoSQL Registry

NoSQL Registry interface, including the database independant functionalities. We removed some comments and formatting spaces to compress the appendix size.

A.2.1. IRegistry.java

```
1  package de.unistuttgart.iaas.cdasmix.nosql.registry;
2
3  import java.util.List;
4
5  public interface IRegistry {
6    public abstract INoSQLTargetDataStore getNoSQLTargetDataStore(String targetDataStoreName,
         String dsTenantId, String dsUserId);
7    public abstract INoSQLSourceDataStore getNoSQLSourceDataStore(String sourceDataStoreName,
         String dsTenantId, String dsUserId);
8    public abstract List<INoSQLSourceDataStore> getNoSQLSourceDataStores();
9    public abstract List<INoSQLSourceDataStore> getNoSQLSourceDataStores(String dsTenantId);
10   public abstract List<String> getNoSQLTenants();
11   /**
12    * @return a string containing the content of the registry in a user-friendly string.
13    */
14   public abstract String dumpRegistry();
15 }
```

Listing A.3: IRegistry, the NoSQL Registry Interface

A.2.2. InformationStructure.java

```
1  package de.unistuttgart.iaas.cdasmix.nosql.registry;
2
3  import java.util.Set;
4
5  public class InformationStructure {
6  [...]
7    public InformationStructure(String mis, String sis) {
8
9      if(mis.startsWith(Keywords.DEFAULT))  {
10       misType = Type.DEFAULT;
11       this.mis = null;
12     } else if(mis.startsWith(Keywords.REGEX)) {
13       misType = Type.REGEX;
14       this.mis = mis.substring(Keywords.REGEX.length());
15     } else {
16       misType = Type.ABSOLUTE;
17       if(mis.startsWith(Keywords.ABSOLUTE))
18         this.mis = mis.substring(Keywords.ABSOLUTE.length());
19       else
20         this.mis = mis;
21     }
22
23     if(sis.startsWith(Keywords.DEFAULT)) {
24       sisType = Type.DEFAULT;
25       this.sis = null;
26     } else if(sis.startsWith(Keywords.REGEX)) {
27       sisType = Type.REGEX;
28       this.sis = sis.substring(Keywords.REGEX.length() );
29     } else {
30       sisType = Type.ABSOLUTE;
31       if(mis.startsWith(Keywords.ABSOLUTE)) {
32         this.sis = sis.substring(Keywords.ABSOLUTE.length() );
33       else
34         this.sis = sis;
35     }
36
37     if (misType == Type.DEFAULT && sisType == Type.DEFAULT)
38       priority =  0;
39     else if (misType != Type.DEFAULT && sisType == Type.DEFAULT)
40       priority =  1;
41     else if (misType != Type.DEFAULT && sisType != Type.DEFAULT)
42       priority = 2;
43     else
44       throw new IllegalArgumentException("Invalid␣Input:␣" + mis +"/"+ sis + "␣default␣sis␣
           is␣only␣valid␣when␣mis␣is␣also␣default");
45
46   }
47 [...]
48   public boolean isDefault() {
49     return misType == Type.DEFAULT || sisType == Type.DEFAULT;
50   }
```

```
51
52    public boolean matches(String containername, String blobname) {
53      boolean matches = matchesIsolated(containername, blobname);
54
55      if(isDefault() && matches) {
56        // Only match if no other information with a higher priority matches.
57        // ABSOLUTE and REGEX > mis ABSULUTE or REGEX and sis DEFAULT > mis and sis DEFAULT
58        for (InformationStructure is : competingInformationStructures) {
59          if(is.matchesIsolated(containername, blobname) && is.priority > priority)
60            matches = false;
61        }
62      }
63      return matches;
64    }
65
66    /**
67     * Checks if the container/blobname matches, ignoring other information structure with a
          higher priority.
68     */
69    public boolean matchesIsolated(String containername, String blobname) {
70      boolean mismatches = false;
71      boolean sismatches = false;
72
73      if(Type.ABSOLUTE == misType) {
74        mismatches = mis.equals(containername);
75      } else if(Type.REGEX == misType) {
76        mismatches = containername.matches(mis);
77      } else { // if (Type.DEFAULT = misType).
78        mismatches = true;
79      }
80
81      if(Type.ABSOLUTE == sisType) {
82        sismatches = sis.equals(blobname);
83      } else if(Type.REGEX == sisType) {
84        sismatches = blobname.matches(sis);
85      } else { // if (Type.DEFAULT = misType).
86        sismatches = true;
87      }
88      return mismatches && sismatches;
89    }
90  [...]
91  }
```

Listing A.4: InformationStructure

A.2.3. INoSQLSourceDataStore

```
1  package de.unistuttgart.iaas.cdasmix.nosql.registry;
2
3  import java.util.Map;
4
5  public interface INoSQLSourceDataStore {
6
7    String getName();
8
9    public abstract Map<InformationStructure, INoSQLTargetDataStore> getTargetDataStores(
         String container, String blobname, Operation operation);
10   public abstract Map<InformationStructure, INoSQLTargetDataStore> getTargetDataStores();
11   public abstract String getUserId();
12   public abstract String getTenantId();
13   public abstract String getComponent();
14   public abstract String getURI();
15   public abstract String getPassword();
16   @Override
17   public boolean equals(Object obj);
18 }
```

Listing A.5: INoSQLSourceDataStore

A.2.4. INoSQLTargetDataStore

```
1  package de.unistuttgart.iaas.cdasmix.nosql.registry;
2
3  public interface INoSQLTargetDataStore {
4
5    String getName();
6
7    boolean supportsRead();
8    boolean supportsWrite();
9
10   String getType();
11   String getUser();
12   String getPassword();
13   String getComponent();
14   String getProvider();
15 }
```

Listing A.6: INoSQLTargetDataStore

A.3. Performance Test

A.3.1. BlobStorePerformanceTest.java

```
1  package de.unistuttgart.iaas.cdasmix.nosql.evaluation;
2
3  import java.io.File;
4  import java.io.FileNotFoundException;
5  import java.io.IOException;
6  import java.io.PrintWriter;
7  import java.net.URISyntaxException;
8  import java.security.InvalidKeyException;
9
10 import de.unistuttgart.iaas.cdasmix.access.BlobStore;
11 import de.unistuttgart.iaas.cdasmix.access.UnifiedBlobstore;
12 import de.unistuttgart.iaas.cdasmix.nosql.evaluation.helper.Array;
13 import de.unistuttgart.iaas.cdasmix.nosql.evaluation.wrapper.AmazonWrapper;
14 import de.unistuttgart.iaas.cdasmix.nosql.evaluation.wrapper.AzureWrapper;
15 import de.unistuttgart.iaas.cdasmix.nosql.evaluation.wrapper.JCloudsWrapper;
16
17 public class BlobStorePerformanceTest {
18
19   private static String amazonBucket     = "amazononly";
20   private static String azureBucket     = "azureonly";
21   private static String replication     = "unistuttgart";
22 [...]
23   private File[] testFiles;
24   private File file_a = new File("testfiles/a-1kB.bin");
25   private File file_b = new File("testfiles/b-10kB.bin");
26   private File file_c = new File("testfiles/c-100kB.bin");
27
28   private int runCounter = 0;
29   public BlobStorePerformanceTest() throws Exception {
30
31     unifiedBlobstore = new UnifiedBlobstore(this.getClass().getResourceAsStream("/unified-
         blobstore.properties"));
32     amazon = new AmazonWrapper(this.getClass().getResourceAsStream("/amazon.properties"));
33     jcloudsAmazon = new JCloudsWrapper(this.getClass().getResourceAsStream("/amazon.
         properties"));
34     amazonBlobStores = new BlobStore[]{amazon, jcloudsAmazon, unifiedBlobstore};
35
36     azure = new AzureWrapper(this.getClass().getResourceAsStream("/azure.properties"));
37     jcloudsAzure = new JCloudsWrapper(this.getClass().getResourceAsStream("/azure.properties
         "));
38
39     azureBlobStores = new BlobStore[]{azure, jcloudsAzure, unifiedBlobstore};
40
41     //testFiles = new File[]{file_a, file_b, file_c, file_d, file_e};
42     //testFiles = new File[]{/*file_a, file_b, file_c, file_d, */file_e/*, file_f, file_g
         */};
43     testFiles = new File[]{file_f};
44
45     protocolFolder = new File("protocol/" + System.currentTimeMillis()+"/");
```

```
46      protocolFolder.mkdirs();
47    }
48
49    public void run() throws Exception {
50      run(azureBlobStores, "azure", azureBucket);
51      run(amazonBlobStores, "amazon", amazonBucket);
52
53      jcloudsAzure.stop();
54      jcloudsAmazon.stop();
55    }
56
57
58    private void run(BlobStore[] targets, String provider, String bucket) throws Exception {
59      runCounter++;
60
61      PrintWriter readProtocol = new PrintWriter(protocolFolder + "/" +  provider + "-read-"
            + runCounter + ".csv");
62      PrintWriter writeProtocol = new PrintWriter(protocolFolder + "/" +provider + "-write-"
            + runCounter + ".csv");
63      PrintWriter deleteProtocol = new PrintWriter(protocolFolder + "/" + provider + "-delete-
            "  + runCounter + ".csv");
64
65      readProtocol.println("Blobstore;File;Size;Operation;Start;End;Result;Comment");
66      writeProtocol.println("Blobstore;File;Size;Operation;Start;End;Result;Comment");
67      deleteProtocol.println("Blobstore;File;Size;Operation;Start;End;Result;Comment");
68
69      for (File file : testFiles) {
70  [...]
71        for (BlobStore blobstore : targets) {
72          warmup(blobstore, file_a, bucket, "readyUp", maxDeltaPercent, writeProtocol);
73          for(int i=0; i<numberOfOperations ; i++) {
74            System.out.print(i + "␣");
75            String blobname = blobstore.getName() + file.getName() + i;
76            testPut(blobstore, file, bucket, blobname, writeProtocol);
77            writeProtocol.flush();
78          }
79
80          warmup(blobstore, file_a, bucket, "readyUp", maxDeltaPercent, readProtocol);
81          for(int i=0; i<numberOfOperations ; i++) {
82            System.out.print(i + "␣");
83
84            String blobname = blobstore.getName() + file.getName() + i;
85            testGet(blobstore, file, bucket, blobname, readProtocol);
86            readProtocol.flush();
87          }
88
89          warmup(blobstore, file_a, bucket, "readyUp", maxDeltaPercent, deleteProtocol);
90          for(int i=0; i<numberOfOperations ; i++) {
91            System.out.print(i + "␣");
92            String blobname = blobstore.getName() + file.getName() + i;
93            testDelete(blobstore,file, bucket, blobname, deleteProtocol);
94            deleteProtocol.flush();
95          }
96        }
97      }
```

```
98      readProtocol.close();
99      writeProtocol.close();
100     deleteProtocol.close();
101   }
102 [...]
103   private static void testPut(BlobStore b, File f, String container, String blobname,
          PrintWriter protocol) {
104     long start = 0;
105     long end = 0;
106
107     String operation = "put";
108     String status = "";
109
110     try {
111       start = System.currentTimeMillis();
112       b.put(f, container, blobname);
113       end = System.currentTimeMillis();
114       status = "ok";
115     }
116     catch( Throwable th) {
117       end = System.currentTimeMillis();
118       status = th.getMessage();
119     }finally {
120       String measurement =b.getName() + ";" +f.getName() + ";"+ f.length() + ";"+ operation
              + ";"+ start + ";"+ end+ ";"+ (end - start) +";"+status;
121       protocol.println(measurement);
122       System.out.println(measurement);
123     }
124   }
125 [...]
126 }
```

Listing A.7: Evaluation program

Bibliography

[Apaa] Apache Software Foundation. Apache Camel: Enterprise Integration Patterns. http://camel.apache.org/enterprise-integration-patterns.html.

[Apab] Apache Software Foundation. Documentation. http://servicemix.apache.org/docs/5.0.x/.

[ASM] The Apache Software Foundation. Apache ServiceMix. http://servicemix.apache.org/.

[BBG11] R. Buyya, J. Broberg, and A. M. Goscinski. *Cloud Computing Principles and Paradigms*. Wiley Publishing, 2011.

[BFM+10] A. Berglund, M. F. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 Data Model (XDM). Technical report, 12 2010.

[Cha04] D. A. Chappel. *Enterprise Service Bus: Theory in Practice*. O'Reilly Media, 2004.

[CW13] H. Cummins and T. Ward. *Enterprise OSGi in Action: With Examples Using Apache Aries*. Manning Publications Co., Greenwich, CT, USA, 2013.

[dCA11] A. de Castro Alves. *OSGi in Depth*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2011.

[HW03] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[JBI05] Java Business Integration (JBI) 1.0, Final Release, 2005. JSR-208, http://jcp.org/aboutJava/communityprocess/final/jsr208/.

[KMK12] R. Krebs, C. Momm, and S. Kounev. Architectural Concerns in Multi-tenant SaaS Applications. In F. Leymann, I. Ivanov, M. van Sinderen, and T. Shan, editors, *CLOSER*, pages 426–431. SciTePress, 2012.

[MG11] P. Mell and T. Grance. The NIST Definition of Cloud Computing. Technical Report 800-145, National Institute of Standards and Technology (NIST), Gaithersburg, MD, September 2011.

[Muh12] D. Muhler. Extending an Open Source Enterprise Service Bus for Multi-Tenancy Support Focusing on Administration and Management. Diploma Thesis 3226, Institute of Architecture of Application Systems, University of Stuttgart, 2012.

[OSG11] OSGi Alliance. OSGi Service Platform: Core Specification Version 4.3, 2011. http://www.osgi.org/Download/Release4V43/.

[Sá13] S. G. Sáez. Extending an Open Source Enterprise Service Bus for Cloud Data Access Support. Diploma Thesis No. 3419, Institute of Architecture of Application Systems, University of Stuttgart, 2013.

[SALM12] S. Strauch, V. Andrikopoulos, F. Leymann, and D. Muhler. ESB^{MT}: Enabling Multi-Tenancy in Enterprise Service Buses. In *Proceedings of the 4th IEEE International Conference on Cloud Computing Technology and Science (CloudCom'12)*, pages 456–463. IEEE Computer Society Press, December 2012.

[SAS+12] S. Strauch, V. Andrikopoulos, S. G. Sáez, F. Leymann, and D. Muhler. Enabling Tenant-Aware Administration and Management for JBI Environments. In *Proceedings of the 5th International Conference on Service-Oriented Computing and Applications (SOCA'12)*, pages 206–213. IEEE Computer Society Conference Publishing Services, December 2012.

[SDQR10] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *Proc. VLDB Endow.*, 3(1-2):460–471, September 2010.

[Tiw11] S. Tiwari. *Professional NoSQL*. Wrox programmer to programmer. John Wiley, Hoboken, N.J. Wiley Chichester, 2011.

[Xia13] S. Xia. Extending an Open Source Enterprise Service Bus for SQL Statement Transformation to Enable Cloud Data Access. Master Thesis No. 3506, Institute of Architecture of Application Systems, University of Stuttgart, 2013.

All links were last followed on December 18, 2014.

Declaration

I hereby declare that the work presented in this thesis is entirely my own. I did not use any sources and references other than those listed.

I have marked all direct or indirect statements from other sources contained therein as quotations.

Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before.

The electronic copy is consistent with all submitted copies.

Stuttgart, December 19, 2014 _____

(Christoph Schmid)