b
UNIVERSITÄT
BERN

Replication Mechanisms for Reference Data

Bachelor Thesis

Tanja Leigh Küry
from
Hofstetten-Flüh SO, Switzerland

Philosophisch-naturwissenschaftliche Fakultät
der Universität Bern

18 March 2018

Prof. Dr. Oscar Nierstrasz

Software Composition Group
Institute of Computer Science
University of Bern, Switzerland

Abstract

Growing numbers of applications interacting with each other facilitate the need to share data between various systems. The interconnected applications used to manage immigrant and asylee need the same reference data in order to consistently fulfil their purpose. This thesis examines the used replication mechanism requiring direct database access and proposes an improved design. By analysing the situation and validating alternative replication mechanisms that adhere to the defined architectural standards it was possible to propose two realistic approaches, one of which is already in development. A proof of concept using a message oriented middleware showed that extending the architecture concept might even be more future-oriented.

Acknowledgement

I would like to offer my special thanks to *Prof. Dr. Oscar Nierstrasz* for making it possible to realise this project as bachelor thesis in the Software Composition Group of the University of Bern.

Also, I want to thank the *Informatik Service Center ISC-EJPD* for providing we with the work place and requirements, as well as advice and feedback.

I am particularly grateful for the guidance and support given by *David Klaper* and *Rolf Scherer* through the whole project.

Adrian Bürki and *Didier Spicher* provided me with very valuable insight that facilitated the definition of requirements and thus the development process.

At last I want to thank *Lars Vögtlin* and everyone else that supported and encouraged me.

Contents

List of Figures

Listings

Acronyms

AMQP Advanced Message Queuing Protocol

API Application Programming Interface

EJPD Federal Department of Justice and Police

GUI Graphical User Interface

ISC IT Service Centre of the Federal Department of Justice and Police

JAX-WS Java API for XML Web Services

JPA Java Persistence API

JPQL Java Persistence Query Language

JSR 220 Enterprise JavaBeans 3.0 Specification

MIDES ”Informationssystem der Empfangs- und Verfahrenszentren und der Unterkünfte an den Flughäfen”

ORM Object-Relational Mapping

POJO Plain Old Java Object

REST Representational State Transfer

SEM State Secretary for Migration

SOAP SOAP Version 1.2

URI Unique Resource Identifier

W3C World Wide Web Consortium

WSD Web Service Description

WSDL Web Service Description Language

XML eXtensible Markup Language

XSD XML Schema Definition

ZEMIS Central Migration Information System

ZEMIS Ref ZEMIS Reference Data Management

1 Introduction

The Central Migration Information System (ZEMIS) [18] is an application to manage data about immigrants and asylees. Multiple applications with their own data that references ZEMIS data are related to it. To guarantee a consistent interpretation of the data, secondary information, such as available countries, cantons, or permits is necessary. This data is called *reference data* and managed trough a separate application, ZEMIS Reference Data Management (ZEMIS Ref), with a separate database. The two main purposes of reference data are displaying translated texts and controlling application behaviour. Due to occasional changes it needs to be replicated into the databases of these applications needing certain reference data to interpret ZEMIS data.

The current replication mechanism is based on a direct database access from the application ZEMIS Ref to the target application databases. This leads to a tight coupling and may also be a security issue. The target databases need to follow a prescribed data scheme and cannot change internally. For additional applications outside the scope of project ZEMIS that will be produced in the near future as part of an e-Government program, this replication mechanism cannot be used due to the required direct database access.

Due to political and technical changes the current ZEMIS Ref application will be replaced. Its outdated web technology presents security risks and needs to be replaced. Also, a significant part of the business logic has been integrated into the web layer, hence a rewrite of most parts of the system is required.

This project focused on exploring different options for replacing the inadequate replication mechanism. We came up with several potential approaches and explored their feasibility with stakeholders and through prototyping.

2 Background

ZEMIS Ref, the application hosting reference data required to interpret data provided by ZEMIS, currently has writing permissions to the databases of all those applications needing this data. These clients thus are tightly coupled to ZEMIS Ref as they cannot store the data in a way that suits their architecture but have to use the prescribed data scheme. Anytime a change of the scheme occurs they have to adapt but they cannot themselves change internally.

In the future applications outside the ZEMIS landscape will be developed and need access to reference data. Due to the required direct database access with hard coded properties a ZEMIS release is required to add a new client. The releases are planned for each entire year in advance, so there is simply no room for considering the schedules of the new clients when planning. Furthermore, it may be a security issue to have direct database access, as some of these application handle very sensitive data.

In this chapter, we provide the background information required to understand the purpose and proceedings of this project along with a short introduction of the technologies used.

2.1 Central Migration Information System

ZEMIS is a web application used to manage immigrant and asylee data. Several applications communicate with it over interfaces, like the application ”Informationssystem der Empfangs- und Verfahrenszentren und der Unterkünfte an den Flughäfen” (MIDES) [9] that is used to manage asylum seekers in reception and procedure centres. ZEMIS is the central system, mainly used for management of person data, while the related applications extend its functionality or need its data. The refugee aid centres regularly get refugee data in order to be able to aid them. Secondary information, necessary to guarantee a consistent interpretation of the data, is managed in another application, ZEMIS Ref.

2.2 ZEMIS Reference Data Management

Reference data is business data that changes only rarely and is used to control application behaviour, display values and interpret ZEMIS data. The web application ZEMIS Ref manages this data. Its database structure is quite complex, thus implemented with a meta data concept. The initial state concerning the replication of reference data to the target databases, as well as the meta data concept implemented to be able to do so should be understood before trying to decouple these applications with usage of a different architectural concept and technology. The next section conveys the basic design and further sections focus on the technologies used.

2.3 Initial State

ZEMIS Ref keeps track not only of values currently active, but also keeps some additional information about how data was edited. Only the currently valid values are of interest for the client application, not old values.

Whenever valid data changes ZEMIS Ref exports these changes and writes them into all target databases needing the respective data, as shown in figure 2.1, with so called replications. To be able to do so information about the database connections and replication jobs are kept, as well as their success or failure. See section 2.3.1 Meta Data Concept.

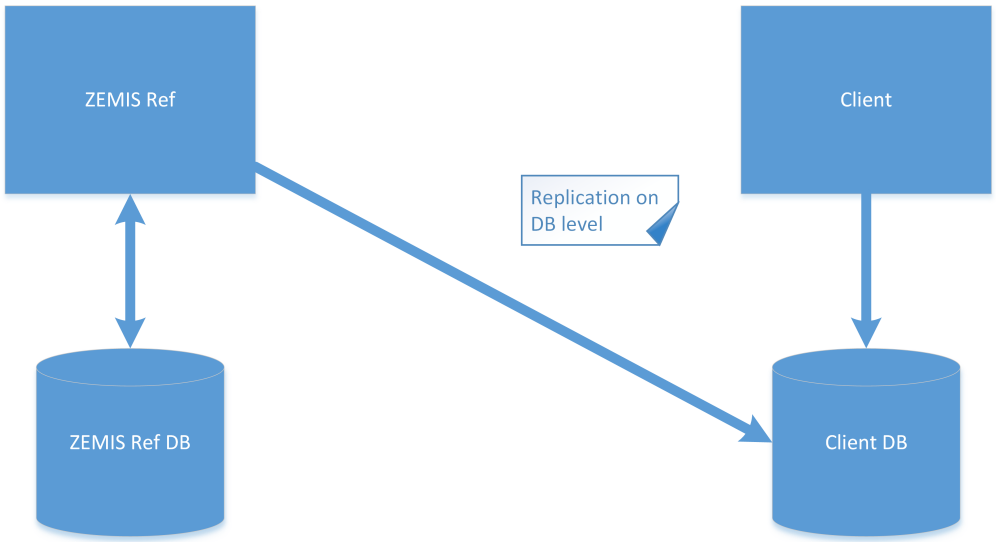

Figure 2.1: Conceptual figure of *Inital State* showing only relevant interactions.

When a data change is made, the change is reflected by making a new entry with the same key and marking the previous entry as old. This provides a crude way of versioning or historisation. There is no possibility to find out which value was active at a particular time, as old entries can be reactivated. Thus, it is not possible to trace the history of possible values. The possibility to display inactivated records is currently given and very important. For example you still need to see the country in which a person was born, even if it does not exist anymore (like Yugoslavia), but you should not be able to select it for a newborn child.

The meta model defines the data scheme which also the client applications need to follow. This tightly couples ZEMIS Ref and its clients. Moreover, to add a new client, all required information, including database authorisation information, has to be added, thus requiring a ZEMIS release.

2.3.1 Meta Data Concept

To describe the actual ZEMIS Ref tables a meta data model is used. This allows creating new tables without having to change the application code. This model describes the entities that consist of one or several tables with records belonging together as well as their relations.

The entities are normally built from one table holding the business information, called `TARP_entityname_bs`, and one holding the text values that contain text in multiple languages to support internationalisation, called `TARP_entityname_tx`. An entity can be a canton, country, authorisation code or other data not changing often. The meta model describes these entities, like name, type, characteristics and so on.

Additionally there are relational tables used to show relations between entities, with TARP_*relationname_r* as name. These are only used for defining valid combinations, thus only used to implement logic.

The meta data model is also used to manage access. Each entity has one of the target systems as data owner or is commonly owned. A user of the application can manage the data in a not normalised way, meaning he does not see each table of an entity, but all data of the entity at once, which is referenced as joined view. The tables are described by the meta model tables named as TARM_*NAME*.

The meta model consists of the following tables:

- TARM_ENTITAET: ID, data owner and name of the joined view (shown in the GUI) for each entity
- TARM_ENTITAETTEXT: Texts that shall be shown in the GUI in German, French and Italian, mapped to an entity ID.
- TARM_VIEW: Matches entities, meaning the tables it consists of, to views to show in the GUI
- TARM_FELD: Meta information for fields used for each entity, e.g. format id, name and position to show in the GUI, description, length and so on
- TARM_FORMAT: format id and name to describe the format of the fields
- TARM_KONTROLLE: Same as TARM_FELD but for the control fields
- TARM_REPL_SUBSCR: Maps clients to entities by using their IDs. This defines to which client databases ZEMIS Ref has to replicate what entities
- TARM_REPL_ENTITAET: Information about success or failure for each replicated entity for each client, containing foreign key to the replication job
- TARM_REPL_JOB: Information concerning each replication job (containing multiple replicated records)

2.3.2 Replication Mechanism

Whenever ZEMIS Ref exports reference data and imports it into a target system by writing into its database, this is called a replication. By using the web application the user can configure when replication jobs should be executed.

Mass replication, containing all changes of valid records, are currently configured to be executed during the night, while immediate replications of single or multiple records are initialised manually. The replication job is then started immediately for all selected records, independent of the date provided by the `valid from` field. This is only taken into account in a mass replication job.

If the user does not need the replication to take place immediately, he can simply mutate or create the entry, set the `valid from` date and let it be mass replicated during each night corresponding to the set date. Success or failure for each replication job are recorded.

Normally it is known in advance when a change is due and the records are prepared well in advance and replicated during the night. Exceptions happen occasionally, for example to make it possible to redo something that is not any more possible due to inactivated values, but that was modified wrongly and must be corrected, like for example redo a permit for a person from Yugoslawia. So sometimes values must be reactivated only for a short time and then be deactivated again, which is the reason for the possibility to manually initialise replications.

2.4 SOAP Web Service

A different approach to transmit data, i.e., replicate data between applications than using direct database access is building a web service. Web services are software systems supporting interoperable machine-to-machine communication by offering an interface over which other systems may interact in a prescribed manner. A Web Service Description (WSD) describes the interface in a format machines can process; for a SOAP Web Service this description is written in Web Service Description Language (WSDL).

SOAP Version 1.2 (SOAP), recommended by the World Wide Web Consortium (W3C), is:

> *"[...] a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment" [8].*

It basically defines an extensible messaging framework with XML technologies, independent of implementation or underlying protocols.

While there are several ways to implement web services, for communication with external applications the IT Service Centre of the Federal Department of Justice and Police (ISC) does so by following SOAP protocol.

The providing entity of the web service, a person or organisation, is the owner on who's behalf the web service provides functionality, while the requesting entity is the person or organisation using this service. Communication takes place by means of XML-Messages, using a clearly defined XSD. Communication is normally started by the requesting entity, but exceptions are possible.

2.4.1 eXtensible Markup Language Schema Definition

In a SOAP web service XML-messages are used to exchange data. XML is a markup language defining a set of rules for encoding documents that are human-readable as well as machine-readable. The structure of an XML document has to follow a defined schema to be suitable as message format for machine-to-machine communication. An approach to do so is by usage of XSD that is also encoded in XML. It basically defines a vocabulary to be used by the providing and requesting entities of the web service and ensures the XML conforms to what is expected.

The XSD is referenced in the XML document through a namespace, the name of it being the Unique Resource Identifier (URI) for the schema. The XML document can then be validated against the XSD to see whether it follows its rules or not. In the same manner a schema definition can include other schema definitions; in that way vocabularies can be reused.

2.4.2 SOAP Web Service Description

A WSD describes a web service by indicating how potential clients should interact with it. It basically works as a contract between the providing entity and the requesting entity. In that way the client knows what the web service offers and how to make use of it, while the provider only has to answer valid requests and make sure to answer with valid responses. The WSD specifies the web service location and the methods by using the basic elements in this structure, encoded in XML. Listing 1 shows the structure of a SOAP web service description.

```
<definitions>
  <types>
    XML Schema type definitions
  </types>
  <message>\label{code:message}
    Definition of an abstract message that can serve
    as input or output of an operation. It can consist
    of one or multiple part elements
  </message>
  <message>
    ...
  </message>
  ...
  <portType>
    Definiton of operations e.g. the interface
  </portType>
  <binding>
    Definition of concrete details for each operation
  </binding>
</definitions>
```

Listing 1: Basic example for a SOAP web service description

2.5 SoapUI

SoapUI is an open source web service testing tool, offering SOAP Web Service and Representational State Transfer (REST) Application Programming Interface (API) functional testing.[15]

SOAP Web Services can be functionally tested by providing the server address and web service description defined in a WSDL and letting SoapUI generate request messages following the description, optionally with adaptable sample values fulfilling the schema defined in the XSD. A request message can then be sent to the web service, which sends back a response message that can be validated against expectations. The XML files are validated against the according schema and errors are indicated, so that the user can see them easily.

2.6 JMeter

When building a SOAP Web Service, not only functionality is a factor to test, but also performance. Even though SoapUI offers functions to do so, JMeter is better suited for load and performance testing, as it is an open source software especially designed for that purpose.[5]

JMeter basically works like a form letter. To load test a SOAP Web Service, you create a Test Plan in which you define how many users shall be mocked, represented as threads, and provide the request template and where to get the variable data from. Also, server location is needed, so it knows where to send the requests to. It can detect errors, but as it is given no schema definition or WSD, there is no way for it to detect whether the responses are conform to the schema or not. It offers several summary reports where statistics about the response times, size of messages and so on are shown.

2.7 Java Persistence API

The Java Persistence API [6], already used in ZEMIS Ref, provides a solution to persistently save Plain Old Java Object (POJO) to a relational database by standardising Object-Relational Mapping (ORM). It was developed as part of Enterprise JavaBeans 3.0 Specification (JSR 220), but can also be used outside the Java EE platform.

A POJO simply needs to be annotated to describe the characteristics of the entity it represents and the resources it needs. The mappings, entity relationships and deploy-time instructions are also set by annotations. This approach simplifies developing java applications using relational databases. Moreover, by usage of the enhanced query language Java Persistence Query Language (JPQL) it is possible to be independent of the database technology, as it works on the JPA entity level and thus is automatically mapped to the required query language.

3 Methodology

While the initial idea was to focus on the meta model, it became evident that the more pressing need for change was in the replication mechanism. Thus, we focused on evaluating alternatives to this complex core piece of the application.

In order for the various stakeholders to be able to work with the solution, we first had to evaluate their needs, then find out which are most important for them.

Four different variants to replace the replication mechanism that in our opinion were able to fulfil basically all needs were established and further investigated. Interviews with the stakeholders made sure to get feedback on these variants. As the solution shall be used in the ISC, we also had to make sure they follow their rules, defined in the relevant software architecture standards.

One additional variant using a messaging service, which is currently not a standard component in the ISC reference architecture, was further investigated, to show potential improvements by using such a framework in the next iteration of the architectural standards. It is described in Appendix A ”Anleitung zu wissenschaftlichen Arbeiten”.

3.1 Adapting the Meta Data Model

The meta data model depends on the data replication mechanism used, as this mechanism influences different options for versioning and replication tracking. As versioning and representation in the application Graphical User Interface (GUI) are the core issues solved by the meta data model it must be adapted according to the new mechanism used.

3.1.1 Improving the Versioning System

When a change is made, the change is reflected by making a new record with the same key and marking the previous record as old using the `state_cd field`. This provides a crude way of versioning or historisation. Essentially, for one entity multiple records are kept, but only one is marked as the currently valid entity for the given key.

By introducing a new field containing the last mutation date of the `state_cd` field a complete historisation could be achieved. It would then be possible to see which record was in use at which date and time. Reactivation of a record not valid any more, as is currently possible, should then be prevented. Instead a new record should be produced to ensure data consistency. This isolated change was not implemented in a prototype.

3.2 Replication Variants

The approaches described in this section were defined at project start. They were later evaluated and adapted. Special focus lied on ensuring data consistency and synchronicity. These are very important aspects, as the reference data is not only used to represent data, but in some cases also to control application behaviour.

As the applications need to ensure that only combinations of certain values according to current law are possible, it is very important that changes of these combinations are consistent in all applications using reference data. For instance, valid permit types and their corresponding valid permit code combinations are such legally binding information.

Furthermore sometimes business tasks are started in one application and terminated in an other. If these applications had concurrently differing reference data, this could lead to synchronisation problems and data conflicts and thus to errors in the applications. So synchronicity must be ensured over all related applications.

For each variant a figure showing relevant data flow provides a conceptual overview, while the according architecture is described.

3.2.1 Variant 1: Push Architecture

A possibility to keep the push model is using a web service client model, as shown in figure 3.1. ZEMIS Ref sends a request containing reference data, the clients acknowledge the reception. Thus, the direct access to the target databases is no longer required, but the client systems need to provide an implementation of a web service interface we define. This means more implementation work, but less coupling between the clients and ZEMIS Ref.

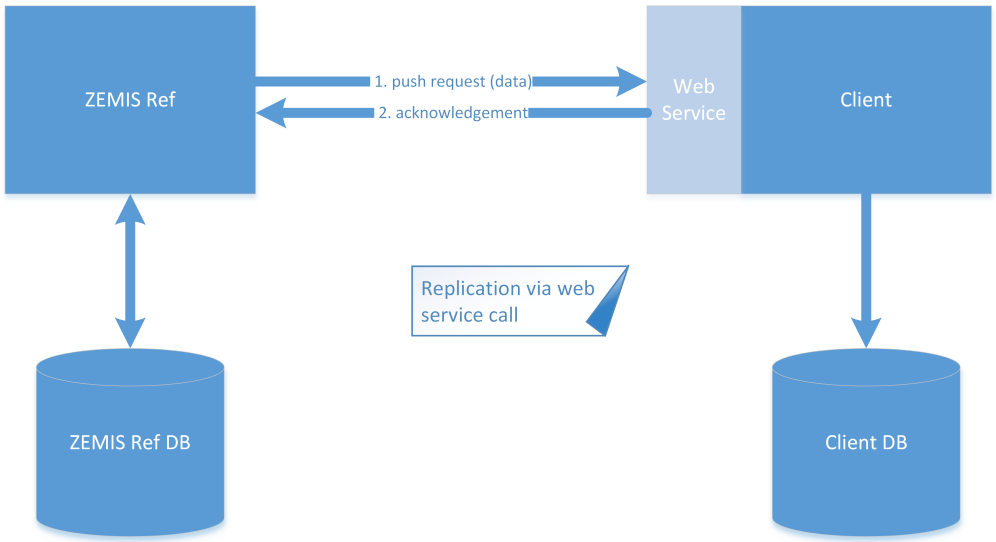

Figure 3.1: Conceptual figure of *Push Architecture*

Clients get the data and can handle it as they need, so they are free to change. It also ensures that there are no issues concerning synchronisation, as the target databases are always up to date if no failures occur. Also, future applications can be added easily to a dynamic subscriber list that replaces the current static subscriber list (see 2.3.1).

To ensure data synchronicity a two-phase commit not shown in the figure can be used as follows: ZEMIS Ref sends a notification that includes information about the data change, clients pull the data and vote for or against committing, depending on whether they were able to import it into their database. ZEMIS Ref then notifies the clients about the result. They then either commit (if all voted for committing) or abort (if not). If one of the clients is unavailable or encounters problems, this approach ensures that the data is the same in all client systems. An unanimous commitment may need several attempts.

3.2.2 Variant 2: Direct Pull Architecture

In this approach the target systems do not have their own databases for the reference data. Each time they need such data they must request the single value via the web service provided by ZEMIS Ref. The response then contains the requested data. You can see this in figure 3.2.

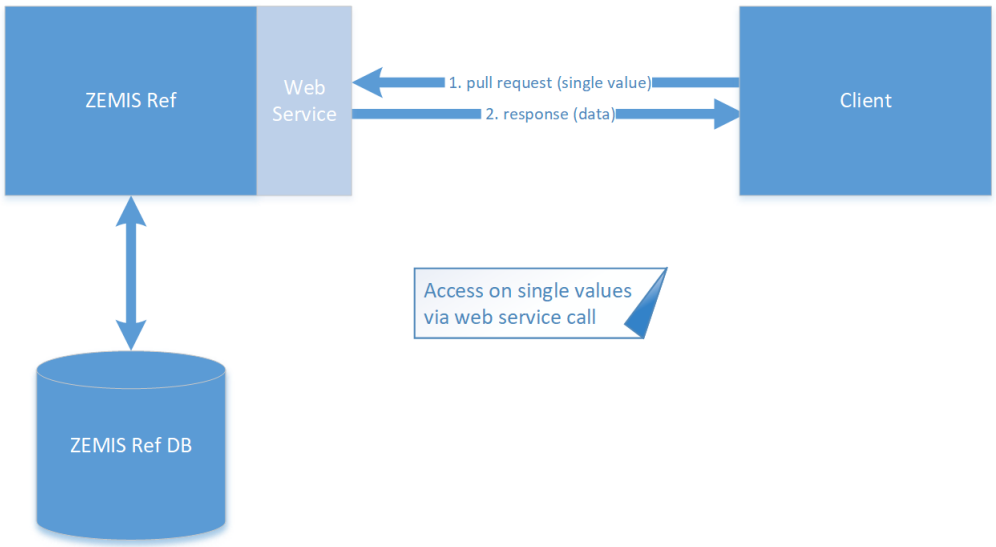

Figure 3.2: Conceptual figure of *Direct Pull Architecture*

There is no direct database access and the data is up to date all the time so there are no synchronisation problems. It gets problematic when the web service is offline or has other disturbances that might affect the performance. The client systems depend heavily on the web service and cannot be used when ZEMIS Ref is not working properly, so most likely a backup solution must be developed. Also, these requests must be fulfilled fast in order not to slow down the client systems. The web service needs to be able to fulfil many requests at once. There would be a high development effort needed regarding performance attributes and potentially increases infrastructure cost because of the high frequency, high performance requirements.

3.2.3 Variant 3: Pull Architecture with Caching

This approach uses a web service provided by ZEMIS Ref and includes caching, as in figure 3.3.

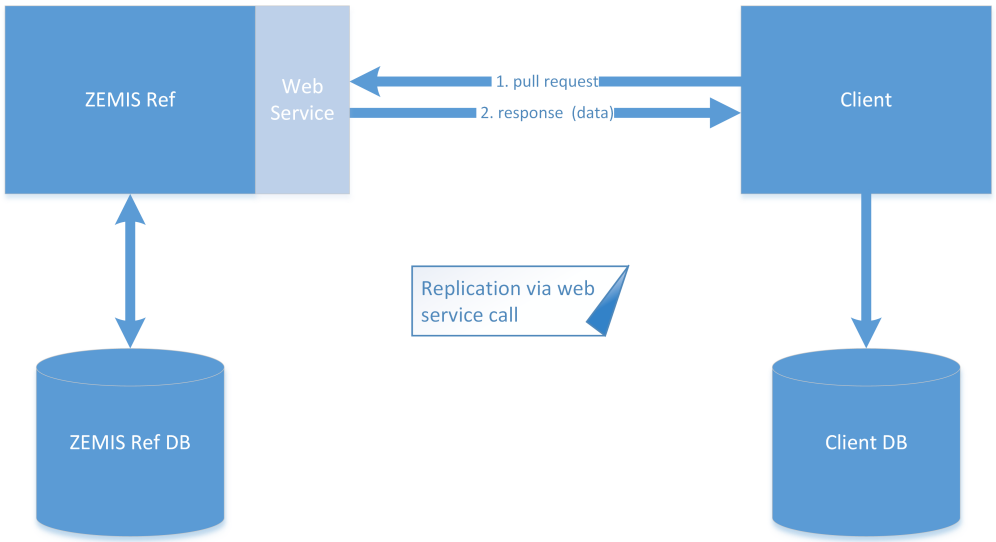

Figure 3.3: Conceptual figure of *Pull Architecture with Caching* showing only relevant interactions and reference data flow.

Via the web service clients can periodically request the full data for a table/entity or the difference to their current state (delta) based on a time stamp (pull request). ZEMIS Ref then sends a response message containing the required reference data the clients then write into their own database used to store reference data.

3.2.4 Variant 4: Push Notify to Pull Architecture

This variant requires ZEMIS Ref and its clients to offer a web service. As shown in figure 3.4 ZEMIS Ref uses the client-provided web service to notify the clients when reference data has changed or becomes active and gets an acknowledgement when the message is received. The client systems then use the web service provided by ZEMIS Ref to pull to update their data and get a response containing the requested data. They can then write it into their own database.

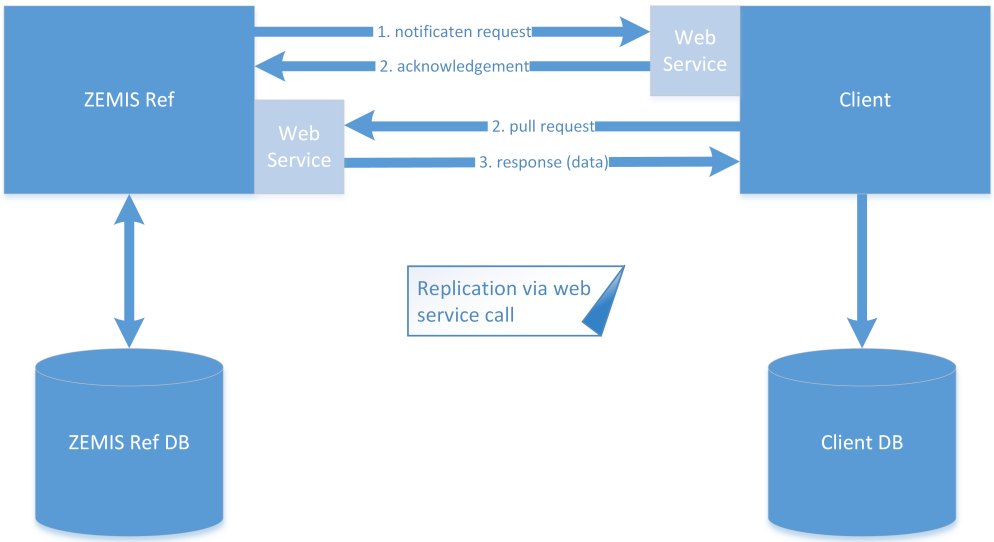

Figure 3.4: Conceptual figure of *Push Notify to Pull Architecture* showing web service calls

There are two possibilities on when the clients should request reference data that slightly differ in the number of resulting data requests.

a) The clients always pull after receiving a notification to get the status change and thus get all information about all reference data changes. For each reference data change all fours steps shown in the figure 3.4 occur.

b) The notification includes information about the status change sufficient for the client to determine whether a pull is needed or not. This must at least be some identification for the changed entities. Steps 2 and 3 shown in the figure 3.4 are only required if the client system needs the data mentioned in the notification request (step 1).

To ensure the client data concerning reference data is synchronous even if one of the clients is temporarily unavailable, there are several approaches. It is important in any case that clients are advised to pull after an unavailability.

The most promising approach is repeating the notification several times with increasing delay, so an unavailability of a client does not cause lasting asynchronicity, but the particular client simply gets the update later. It is important to include an identification for the data-change to avoid multiple pulls. This can be achieved by usage of SOAP header elements.

To reduce data load, the pulls should only request changed data, but full pulls (whole tables) must be possible also to support initial loads of the data for a new client. If for one table multiple records are modified and the clients get a notification for each of these, this would lead to a lot of notifications and pulls. To avoid this, ZEMIS Ref could wait a defined amount of time after a modification and only send a notification if no records of the same table are modified during this time. So with this delay only one notification would be sent for a sequence of updates on this table and only one pull made by each client.

3.3 Stakeholder Interviews

Stakeholders are the product manager for ZEMIS Ref in State Secretary for Migration (SEM) and the clients, for which we chose the lead developer of Adeya, a client application in development in ISC, as representing person. We proposed them the solution variants and tried to find out their requirements and what is important for them.

Didier Spicher, product manager of ZEMIS Ref, was only asked some general questions, as the most adaptions would not affect the usage of ZEMIS Ref, but only the behaviour. According to him it is important that there are no differences in the client data for a longer time, but about half an hour would be no problem.

Also, the possibility to change a record and induce the replication immediately must be kept. Sometimes the support needs to correct mistakes made by the cantonal users and thus has to do mutations that would normally not be possible due to the reference data. So then they can change the according reference data, do the required mutations in the target applications and then redo the reference data change. This is happening quite often, mainly during office hours, as a record in a state that is not allowed according to law cannot be kept in that state for a whole day.

Adrian Bürki, lead developer of Adeya and Software Architect at ISC was provided with information on the variants as in section 3.2 Replication Variants and asked to answer some questions. He supported the evaluation of the technical consequences for the different replication variants.

3.4 Evaluation of Variants

With the Stakeholder feedbacks it was possible to already see positive and negative aspects of the proposed variants and to further analyse and evaluate them. All variants remove the direct database access and let the clients define their own scheme to be used, thus loosening the coupling.

3.4.1 Evaluation Variant 1: Push Architecture

This variant is closest to the initial state, as the exchange of the data is initialised by ZEMIS Ref, just like replications.

A negative point is that the clients need to provide an implementation of a web service interface we define. Thus, a new client has a higher initial development effort to connect to ZEMIS Ref. Also, this approach contradicts the ”pull over push” policy given by the ISC.

Positive aspects are the robustness, that there is no heavy workload to be expected and that it ensures consistency. As according to Didier Spicher data differences for about half an hour are no problem the two phase commit is not needed. Also, Adrian Bürki did not see any reason to use a two phase commit.

As the meta model consists of tables to keep track of the replication jobs and entities, these could simply be adapted to keep track of the web service requests and their failure or success. Failure detection would be given, not successful requests could be repeated easily in half an hour and thus ensure consistency well enough.

Impact on the meta model

- Tarm_repl_subscr shall be dynamic subscriber list with the web service addresses
- Tarm_repl_clients must be kept
- Tarm_repl_job should be kept for traceability
- Tarm_repl_entity should be kept for traceability

3.4.2 Evaluation Variant 2: Direct Pull Architecture

A direct pull architecture where records are pulled directly when required heavily depends on the providing entity of the web service, which in this approach is ZEMIS Ref. To be able to fulfil all requests during peak times when a heavy workload is to be expected fast enough to not slow down the clients several servers would be needed. Currently 2500 calls/minute are to be expected in ZEMIS with a higher amount when considering the client applications and also the new applications to be produced in the near future.

Although this variant seems to have some very good aspects, especially concerning consistency and data synchronicity, we decided to not investigate it further due to the anticipated performance bottleneck.

Impact on the meta model

- Tarm_repl_subscr not needed
- Tarm_repl_job not needed
- Tarm_repl_entity not needed

3.4.3 Evaluation Variant 3: Pull Architecture with Caching

This variant, where ZEMIS Ref offers a web service and the clients pull periodically, adheres to the ”pull over push” policy the ISC follows. Also it is easy to add new clients, because only ZEMIS Ref needs to implement a web service interface, not the clients.

The only weakness, meaning differing data due to not clearly defined request times, is abated by regularly and often pulling clients and the statement that differences of about half an hour are not expected to cause problems or errors. This required high rate of pull requests might lead to many requests producing empty responses and thus unnecessary communication.

Also, it has the advantage that clients can pull deltas, meaning only the data differing from their current state, thus lessening performance requirements. This requires that some information to trace which client pulled when must be stored to be able to determine what data must be included in the response for a delta pull for each client application.

Impact on the meta model

- Tarm_repl_subscr not needed
- Tarm_repl_job not needed
- Tarm_repl_entity not needed
- New element: Probably an additional version control table or at least some fields for version control are needed to be able to trace which client pulled when.

3.4.4 Evaluation Variant 4: Push Notify to Pull Architecture

This approach, with ZEMIS Ref and the client applications providing a web service, does not produce a lot of unnecessary communication, but ensures that pull requests are only sent when in need of data. This is especially the case when including information about the data change in the notification and letting the clients determine whether they need to pull or not, as proposed for variant b. Variant a still requires the client to pull whenever it receives a notification, thus results in more communication than variant b, but this is still way less than in the 3.2.3 *pull architecture with caching*.

Data synchronicity over all the clients and data consistency are maintained when using this approach. Also the coupling is loose, as ZEMIS Ref only needs to know the web service locations of its clients, but does not have to keep track of pull request times and status. The clients themselves need to ensure they get all data they need.

What contradicts this approach is that it requires ZEMIS Ref as well as the clients to implement a web service client and interface and thus means a lot of implementation work for the clients to initially connect to ZEMIS Ref.

As the pattern described in this variant resembles a ”publish-subscribe” pattern, we came to the conclusion that a messaging service technology would suit better than the usage of web services, especially if half of the communicated information only consists of notifications. As also Adrian Bürki did propose the usage of a messaging service technology, we followed that advice by prototyping an additional variant, see A Anleitung zu Wissenschaftlichen Arbeiten.

3.4.5 Conclusion

The evaluation of the variants showed us the differences between the possible data exchange patterns. The stakeholder interviews further refined our understanding of the effects each variant has. We expected a pull with caching architecture to be the most likely option due to the efficient decoupling and adherence to ISC standards, so we started prototyping this variant to validate it. We also anticipated that the XSD required to implement the SOAP Web Service for the pull architecture with caching could be used as well for the push architecture with caching, as they must exchange the same data structure, so this was the second variant to validate.

In addition, to show the potential of a proper publish subscribe architecture, we implemented a very basic version using the message broker RabbitMQ. A description of this can be found in A ”Anleitung zu wissenschaftlichen Arbeiten”.

For the other variants we decided to not build a prototype as their disadvantages showed us they were very unlikely to be used in the ISC.

4 Validation

After evaluating the different approaches to supersede the replication mechanism requiring direct database access we started implementing a prototype of our preferred variant, the *pull architecture with caching* (see 3.2.3). As similarities in the data exchange structure, meaning the XML files, of this variant and the *push architecture* (see 3.2.1) were obvious, the data structures could partially be used for both prototypes.

To build the prototypes we followed the contract first approach, which places the main focus on the XML structure of the messages to be sent rather than on the JAVA code [7].

4.1 Designing the Prototype Structures

First of all we had to determine the offered services, as well as how and where clients should be able to claim them. These must be described in the WSD (see section 2.4 SOAP Web Service Description). With these in mind, we started designing the prototype structures by determining the XML data structure of the request and response files. Also, we produced the XSD so that the files pass the validation and the according WSD including these definitions. The basic structure and name conventions were given, as we were provided with an example in use by the ISC.

4.1.1 Pull Architecture with Caching

For this variant it is important that the client can pull all its entities (see 2.3.1 Meta Data Model). In case the client only wants to update part of its database or only step by step there is also the possibility to choose the entities to get in the response. So two request messages, one for all the entities and one for multiple entities, are needed, as well as a response message. These messages are described in the WSD along with the operations they trigger.

4.1.1.1 Web Service Description

This section outlines the basic elements used to describe the WSD for the web service for the *pull architecture with caching*.

Messages The required messages for the web service are described in the WSD as `messages` (see listing 2). These can then serve as input or output for an operation.

```
<wsdl:message name="AllMyEntitiesRefDataRequest">
  <wsdl:part name="request" element="types:allMyEntitiesRefDataRequest"/>
</wsdl:message>
<wsdl:message name="MultipleEntitiesRefDataRequest">
  <wsdl:part name="request" element="types:multipleEntitiesRefDataRequest"/>
</wsdl:message>
<wsdl:message name="RefDataResponse">
  <wsdl:part name="response" element="types:refDataResponse"/>
</wsdl:message>
```

Listing 2: Methods used for the *Pull Architecture with Caching*

The name is used to identify the message, while the `wsdl:part` element is used to define whether the according message is a request or a response and to assign the type defined in the XSD. The URI of the schema definition to use is referenced over the namespace, in this case `types:` .

PortType Additionally, also a group of operations that serves as interface for the web service was needed, see listing 3. The term `PortType` is misleading, as actually it is the interface.

```
<wsdl:portType name="ZemisRefDataPort">
  <wsdl:operation name="allMyEntities">
    <wsdl:input message="tns:AllMyEntitiesRefDataRequest"/>
    <wsdl:output message="tns:RefDataResponse"/>
    <wsdl:fault name="fault" message="tns:FaultMessage"/>
  </wsdl:operation>
  <wsdl:operation name="multipleEntities">
    <wsdl:input message="tns:MultipleEntitiesRefDataRequest"/>
    <wsdl:output message="tns:RefDataResponse"/>
    <wsdl:fault name="fault" message="tns:FaultMessage"/>
  </wsdl:operation>
</wsdl:portType>
```

Listing 3: Web service interface defined as group of operations

These operations define one of the previously defined messages as `input` and another one as `output` , as well as a message as `fault` in case a fault occurs. They basically map the requests to the according responses the web service provides and thus describe its interface and offered methods. When testing with SoapUI (see section 2.5 SoapUI) the output message can be generated automatically with mock data when provided with the message that is defined as input message.

When using contract first approach [7] the portType and its operations provide the skeleton to be implemented, meaning Java interfaces including the methods.

Service To make the web service available for clients it must be published. This is done by usage of the `service` element (listing 4).

```
<wsdl:service name="ZemisRefDataService_v1">
  <wsdl:documentation>Web service definition for Zemis Ref Data Service, used
  to sync Ref Data between Zemis Ref and its Clients.</wsdl:documentation>
  <wsdl:port name="ZemisRefDataPort" binding="tns:ZemisRefDataBindingHTTP">
    <soap:address location="http://ejpd.admin.ch/sem/zemis/refdataservice"/>
  </wsdl:port>
</wsdl:service>
```

Listing 4: Publishing the web service

Here not only the `address location` and `port` (meaning the interface) are described, but also a short `documentation` of what the service shall be used for.

4.1.1.2 XML Data Structures

Each message described in the WSD corresponds to one XML file. We first designed their structure that is shown here, then the schema definition that is explained in appendix B XML Schema Definition.

Request Messages The client may send a `AllMyEntitiesRefDataRequest` with the structure as in listing 5 to request all its entities. Just the `clientName` is required. The `startDate` can be used to overwrite the date from which on to include data changes and `modifiedOrNew` can be set to *true* to only get the entries modified or newly created since the provided start date or the stored date of the last pull.

```
<allMyEntitiesRefDataRequest xmlns="http://ejpd.admin.ch/sem/zemis/refdataservice/
    types/v1">
  <clientName>MIDES</clientName>
  <startDate>2017-06-13T09:00:00</startDate>
  <modifiedOrNew>true</modifiedOrNew>
</allMyEntitiesRefDataRequest>
```

Listing 5: XML file the client MIDES might use to pull all its entities

To request only some entities, a `MultipleRefDataRequest` in the structure as in listing 6 must be used. For each entity it is possible to get only the entries modified or newly created.

```
<multipleEntitiesRefDataRequest xmlns="http://ejpd.admin.ch/sem/zemis/refdataservice/
    types/v1">
 <clientName>MIDES</clientName>
  <startDate>2017-06-13T09:00:00</startDate>
 <requestedEntity>
    <entityName>Gemeinde</entityName>
    <modifiedOrNew>true</modifiedOrNew>
 </requestedEntity>
  <requestedEntity>
    <entityName>Adressdaten</entityName>
    <modifiedOrNew>false</modifiedOrNew>
  </requestedEntity>
</multipleEntitiesRefDataRequest>
```

Listing 6: XML file the client MIDES might use to pull two entities

Response Message The `RefDataResponse` message is a bit more complex than the `Request` messages. We had to redesign this part after testing, as the XML files were too big, see section B XML Schema Definition.

All requested entities are included in this file. The tables they consist of are sent separately. For each `table` first the `columns` with their `data type`, which is important for the client applications, are sent with the *name* as, then the `rows` with the data. To distinguish between the *old*, *new* and *modified* rows, the rows are grouped and marked with the `status` attribute.

```
<ns4:refDataResponse xmlns:ns4="http://..." xmlns="http://...">
  <ns4:refEntity joinedViewName="VARP_EMENDATION">
    <ns4:table name="VARP_EMENDATION_BS" entityPart="business">
      <ns4:columns>
        <ns4:col dataType="ALPHA_NUM">KEYCD</ns4:col>
        <ns4:col dataType="NUM">TAGESGERICHT</ns4:col>
        ...
      </ns4:columns>
      <ns4:rows status="old">
        <ns4:row index="0">
          <ns4:entry>asw</ns4:entry>
          <ns4:entry>357766</ns4:entry>
          ...
        </ns4:row>
        ...
      </ns4:rows>
      <ns4:rows status="modified">
        <ns4:row index="0">
          ...
        </ns4:row>
        ...
      </ns4:rows>
      <ns4:rows status="new">
        <ns4:row index="0">
          ...
        </ns4:row>
      </ns4:rows>
      ...
    </ns4:table>
    ...
  </ns4:refEntity>
  ...
</ns4:refDataResponse>
```

Listing 7: Example of a response XML file

4.1.2 Push Architecture

For the push variant the workflow goes in the opposite direction, as shown in figure 3.1. The client does not have the freedom to choose, but changes are pushed at certain times via the offered web service and the client must be able to deal with them.

For the data structures, this means the previously defined response structure (see B.0.1) can be reused, but must be defined as request, while the request structures cannot be used, but a new response structure must be defined.

4.1.2.1 Web Service Description

The web service must be offered by the client applications. This section describes the parts that are not depending on a specific client application, but should be offered by all of them.

Messages The required messages for the web service are described in the WSD as `messages` (see listing 8). These can then serve as an input or output for an operation.

```
<wsdl:message name="RefDataRequest">
  <wsdl:part name="request" element="types:refDataRequest"/>
</wsdl:message>
<wsdl:message name="RefDataResponse">
  <wsdl:part name="response" element="types:refDataResponse"/>
</wsdl:message>
<wsdl:message name="FaultMessage">
  <wsdl:part name="parameter" element="respfault:FaultInfo"/>
</wsdl:message>
```

Listing 8: Methods used for the *Push Architecture*

The `RefDataRequest` message is used by ZEMIS Ref to send a message including changed reference data. The client should then answer with a message of type `RefDataResponse` . This is described in the `portType` .

portType The operation serving as interface should be defined as follows in listing 9.

```
<wsdl:portType name="RefDataPort">
  <wsdl:operation name="refDataRequest">
    <wsdl:input message="tns:RefDataRequest"/>
    <wsdl:output message="tns:RefDataResponse"/>
    <wsdl:fault name="fault" message="tns:FaultMessage"/>
  </wsdl:operation>
</wsdl:portType>
```

Listing 9: Web service interface defined as group of operations

There is only one `operation` , as only this one is needed.

Service The `service` must be offered by the client applications. It could look like listing 10.

```
<wsdl:service name="ZemisRefDataService_v1">
  <wsdl:documentation>Web service definition for Zemis Ref Data Service to push
  reference data.</wsdl:documentation>
  <wsdl:port name="RefDataPort" binding="tns:RefDataBindingHTTP">
    <soap:address location="some URL"/>
  </wsdl:port>
</wsdl:service>
```

Listing 10: Publishing the web service

4.1.2.2 XML Data Structures

The response structure defined for the *pull architecture with caching* (see B.0.1 refDataResponse) must only be renamed, as it should be a `refDataRequestType` instead of a `refDataResponseType` .

```
<xs:complexType name="refDataRequestType">
  <xs:sequence>
    <xs:element name="refEntity" type="tns:entityType"
    maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Listing 11: Definition of the refDataRequestType

The structure must not be adapted, and all elements defined for in the XSD can be reused. An additional `messageId` element might be useful to be able to map request

messages to response messages, preferably in the soap header element [2]. This must then also be included in the according response message.

4.1.3 Direct Pull Architecture

For the *direct pull architecture* the client application only needs to be able to pull the record for one key it is provided with. So here simply a request method for such a pull is needed, along with the according response. The structures already designed could only partially be reused, but the basic concept is the same.

4.2 Building the Prototypes

This section describes how the prototype for the *pull architecture with caching* was built.

We did not build a prototype for the *push architecture*, as the ISC started developing a new application including the proposed *push architecture*, see section 5.3 Developments of Needs and Circumstances.

During validation of the variants we decided to not build a prototype for the *direct pull architecture*, due to its disadvantages, see 3.4.2 Evaluation Variant 2: Direct Pull Architecture.

4.2.1 Building the Prototype for Pull Architecture with Caching

After having defined the structures we used Java API for XML Web Services (JAX-WS) [4] to generate the required classes. This was done in the command line interface and only required the WSD and XSD location.

One class for every `type` defined in the XSD was generated. These POJOs were not connected with the database. Their functionality is to provide a basic container in which any given XML corresponding with the XSD can be represented. So for a `allMyEntitiesRefDataRequest` sent to the web service automatically an object of type `AllMyEntitiesRefDataRequestType` is created. It is set as input parameter for the method `allMyEntities` .

To match these given objects to the database, we used JPA (see section 2.7 Java Persistence API) as ORM. Thus, we created a JPA entity for each required `TARM_`-table, but not for the `TARP_`-tables that represent the business entities.

4.2.1.1 Creating the JPA Entities

JPA Entites To build a JPA entity, you simply have to annotate it with `@Entity` . Additionally, as it shall map with a database table the annotation `@Table` is used with the *name* of the table as parameter, as in listing 12.

```
@Entity
@Table(name = "TARM_ENTITAETTEXT")
public class EntityText {
   ...some code ...
}
```

Listing 12: Annotating an entity

For the columns of the table, fields in the class are created and annotated with `@Column` , as in listing 13. To create a field whose definition differs from the column name it can be used with an additional parameter called *name*. `@Id` is used to mark it as primary key. If more than one column is marked as key, JPA assumes that they together compose a key.

```
@Id
@Column(name = "LANG_CD")
private String langCd;
```

Listing 13: Annotating a column

JPA Queries In order to be able to access the data belonging to a JPA entity, JPQL queries can be used. These must be defined for the entity by annotating. For our prototype we used *namedQueries* as in listing 14.

```
//gets all the clients with the given name
@NamedQuery(name = "Client.findByName", query = "select c from Client c where
    name = :name"),
```

Listing 14: Defining named queries

By calling this *namedQuery* with the name of the client as parameter, we will get the client entity object.

Relations To realise a relation to an other JPA entity one or both of them must declare it, depending on the type of relation. We used one-to-one, many-to-many and one-to-many relations, but will only explain a one-to-many relation in detail.

For the one-to-many relations we used a bidirectional association [6], as in listings 15 and 16. This lets both entities know about the relation.

The entity *entity* has a one-to-many relation to the entity *entityText*.

```
@Entity
@Table(name="TARM_ENTITAET")
public class RefEntity {
    ...
    @OneToMany(mappedBy = "refEntity")
    private Set<EntityText> entityText;
    ...
}
```

Listing 15: Declaring the one-to-many relation

The `RefEntity` object thus has a `Set` of `EntityText` objects. The parameter `mappedBy` sets the field with name `"refEntity"` of the class `EntityText` as foreign key.

For the many-to-one relation, this is done by annotating `@ManyToOne` and telling the system which column in the other entity must be used for the mapping. The `@JoinColumn` is used for that.

```
@Entity
@Table(name = "TARM_ENTITAETTEXT")
public class EntityText {
    ...
    @Id
    @ManyToOne
    @JoinColumn(name = "ENTITAET_ID", referencedColumnName = "ID") private
      RefEntity refEntity;
    ...
}
```

Listing 16: Declaring the many-to-one relation

The `referencedColumnName = "ID"` tells the system that the column in the table *TARM_ENTITAET* must be used as foreign key.

4.2.1.2 Implementing the Methods

With the tools provided by the JAX-WS implementation a Java interface representing the web service operations was generated. Implementing this interface can be seen as creating the functionality of the web service.

Signature and response type of these methods were defined in the WSD, see listing 3. So when a `AllMyEntitiesRefDataRequest` message reaches the web service, the method corresponding to the operation is invoked, in this case the `allMyEntities` method.

allMyEntities The method `allMyEntities` is invoked when a message of type `AllMyEntities` reaches the web service. The `request` object is of type `AllMyEntitiesRefDataRequestType` and contains the data from the XML request file.

To be able to get all entities belonging to the requesting client application, we have to create a `client` object. Hence, we have to get the data representing the requesting client from the database. The `createNamedQuery` in line 1 of listing 17 uses the JPQL query defined for the JPA entity class *Client*.

```
1 Query clientByNameQuery = em.createNamedQuery("Client.findByName");
2 clientByNameQuery.setParameter("name", clientName);
```

Listing 17: Using the *namedQuery* to get the client from the database

Inside the method we can now use the `client` object. As all its data is represented in the JPA entities, we can now access its business entities by simply using the `client` 's getters.

Now, we can create the response XML file. This is exactly the opposite of what we did with the request. Instead of taking out the values from the POJO structure and filling it into the JPA entities, we have the JPA entities and need to create and fill the container representing the response.

So, we create a *response* object. For all the business entities belonging to the client, meaning also the ones that are commonly owned, we have to create and add a new *EntityType* object.

```
1 RefDataResponseType response = new RefDataResponseType();
2 for (RefEntity re : client.getRefEntities()) {
3    Helper.addNewEntityType(em, response, re);
4 }
```

Listing 18: Create *response* and add its *entities*

The helper method creates the *entityType*, appends it to the *response* and sets its attribute *joinedViewName*. It also adds the tables belonging to the *entity*.

We proceed to build the *response* object by adding the required web service type objects (defined in section B.0.1) and filling them with corresponding values from the database. Which values to use for the entries depends on the *modifiedOrNew* and *startDate* values received with the request.

In the end we have the *response* object containing all data the client requested. By returning it, the XML file is automatically generated and sent back to the client.

multipleEntities Implementing this method does not differ much from the *allMyEntities* method. The only difference is that not all business entities belonging to the requesting client must be added to the *response*, but only the ones whose *entityName* was given in the request.

As soon as we have the *client* object, we also have all its *entities* and can pick the ones mentioned in the request. Then, we fill the *response* object as described above and return it.

4.3 Findings from the Prototypes

Designing and building the prototypes helped us find and understand not only their advantages and disadvantages, but also shortcomings of our approach.

The data structures were designed mainly with the idea in mind that the XML files should be human readable. As only the client systems must in the end be able to work with these files, this was not necessary but only leading to files bigger than actually needed, see 4.3.2 Problems.

4.3.1 Testing

During development soapUI was used to test functionality of the web service. Especially for the pull variant it was very helpful to be able to see the output file generated with soapUI (see 2.5). When the basic functionality was achieved we used JMeter (see 2.6) to load test and see how fast it worked. As we found out, it was quite fast, but we just had a small database with not many tables and only some records per table.

4.3.2 Problems

Regarding generated file size an adaption of the XML data structure was necessary, as the files were quite big despite the small number of records used. When the files are too big this leads to a timeout problem with the firewall used at the ISC. There was room to reduce file size, as the XML files were generated under the aspect of human readability, not on effectiveness concerning space usage.

An XML file fulfilling the first version of the defined data structure is shown in listing 19.

```
<table name = ”Persons”>
  <row index=”0” status=”new”>
    <active>true</active>
    <entry column = ”place” type=”ALPHA_NUM”>Subingen</entry>
    <entry column = ”name” type=”ALPHA_NUM”>djBobo</entry>
    <entry column = ”age” type=”ALPHA_NUM”>45</entry>
  </row>
  <row index=”1” status=”modified” >
    <active>false</active>
    <entry column = ”place” type=”ALPHA_NUM”>Bern</entry>
    <entry column = ”name” type=”ALPHA_NUM”>John Doe</entry>
    <entry column = ”age” type=”ALPHA_NUM”>22</entry>
  </row>
</table>
```

Listing 19: First version of the file containing reference data

As you can see in listing 19 the column names `column = "place"` and data types `type = "ALPHA-NUM"` are repeated in every row of the table. So we moved this information to a separate part above the rows to reduce the size of the files, see listing 20.

```
<table name=”Persons”>
  <ns4:columns>
    <col dataType=”ALPHA_NUM”>place</col>
    <col dataType=”ALPHA_NUM”>name</col>
    <col dataType=”NUM”>age</col>
    <col dataType=”BOOLEAN”>active</col>
  </ns4:columns>
  <row index=”0” status=”new”>
    <entry>Subingen</entry>
    <entry>djBobo</entry>
    <entry>45</entry>
    <entry>1</entry>
  </row>
  <row index=”1” status=”modified”>
    <entry>Bern</entry>
    <entry>John Doe</entry>
    <entry>22</entry>
    <entry>1</entry>
  </row>
</table>
```

Listing 20: Second version of the file containing reference data

Furthermore the information whether the row, meaning one record, is active or not was at first in a separate tag. This makes sense when the file must be human readable, but does not improve machine readability. Also, it needs more CPU time to treat this column different from the others.

Even though on the first glance the second version in listing 20 seems to be using more space, the opposite is true, more so with increasing number of rows. With these adaptions file size was reduced significantly.

To check the adaptions and to be sure that such issues caused by a database much smaller than the original do not occur any more we were provided with a mock database with a similar size following the prescribed meta data model. Our tests with the mock database confirmed our suspicion that this new format reduced the average response size significantly.

5 Conclusion and Future Work

After establishing requirements with stakeholders, and then building and improving prototypes, our understanding of the different options for replication mechanisms has improved. This section summarises our thoughts about past work and future work on reference data replication.

5.1 Conclusions

By analysing ZEMIS Ref and the applications interacting with it weak points could be identified and improvements could be proposed. Even though the prototypes were still in construction, the analysis of ZEMIS Ref and its architecture showed us that there is a lot potential for improvements. The proposal of *push architecture with caching* (see 3.2.1) already gave an application currently under development the opportunity to be independent of the prescribed data scheme, see 5.3.

The meta data model was not significantly adapted as intended, because the future application design is still undetermined.

5.2 Future Work

The application ZEMIS Ref is End-Of-Life as its web technology reached End-Of-Life in 2013. This presents security risks. Additionally, a rewrite of most parts of the system is required because a significant part of the business logic has been integrated into the web layer.

5.2.1 Management of Reference Data trough Clients

As reference data can actually be owned by a ZEMIS Ref client application, the idea came up to let the clients create new entities in their owned tables. This is not possible with any of the proposed variants, nor with the initial state, but could be an improvement. That way entities only used by one application could be managed by the application itself. A client application currently can only be owner of an entity if no other application needs it, so there would be no problem concerning data synchronicity, provided this does not change.

5.2.2 Extended Functionality

The functionality provided by ZEMIS Ref is very basic and was designed only for the purpose to manage reference data by occasional changes. Usage is time consuming and very ineffective. For the expected growing numbers of applications needing this data, including applications outside the scope of project ZEMIS, there is a need for improvement.

5.3 Developments of Needs and Circumstances

The ISC currently develops a new application that is part of the ZEMIS landscape. As ZEMIS Ref is going to be overhauled in the next few years, they decided to in a first step already develop the new application in a way that decouples it from the direct database access, but that offers a web service interface over which ZEMIS Ref can push changes, thus following the *push architecture with caching* approach described in 3.2.1. It means more work for them in the moment, but less so later, as the application does not have to be adapted as soon as ZEMIS Ref is overhauled.

The decision to use this approach rather than the pull approach is based on the fact that it is closest to the initial state and thus affects ZEMIS Ref the least. So it is the cheapest and easiest solution for the moment, as only minor changes to ZEMIS Ref are needed. It is planned to use the *pull architecture with caching* approach described in 3.2.3 in the next step, when ZEMIS Ref is rewritten.

A

Anleitung zu wissenschaftlichen Arbeiten

This chapter shall work as a guide on how to implement a prototype for data exchange between several applications by using RabbitMQ [11]. It is assumed that one of these applications is working as central application that must supply the other applications with the data they need. You should have some basic knowledge of Java programming.

A.1 Background

RabbitMQ is an open source message broker implementing Advanced Message Queuing Protocol (AMQP) [10]. This protocol is an open standard that allows passing of messages between various kinds of applications. It enhances interoperational communication and sharing of resources and thus facilitates building service oriented architectures.

AMQP has the following viewpoint:

> *"[...] messages are published to exchanges, which are often compared to post offices or mailboxes. Exchanges then distribute message copies to queues using rules called bindings. Then AMQP brokers either deliver messages to consumers subscribed to queues, or consumers fetch/pull messages from queues on demand." [13].*

It was originally designed for the banking industry, thus is secure and reliable and focuses on interoperability [10], hence it should fit the requirements for our purpose.

Figure A.1 shows the basic concept of the AMQP protocol.

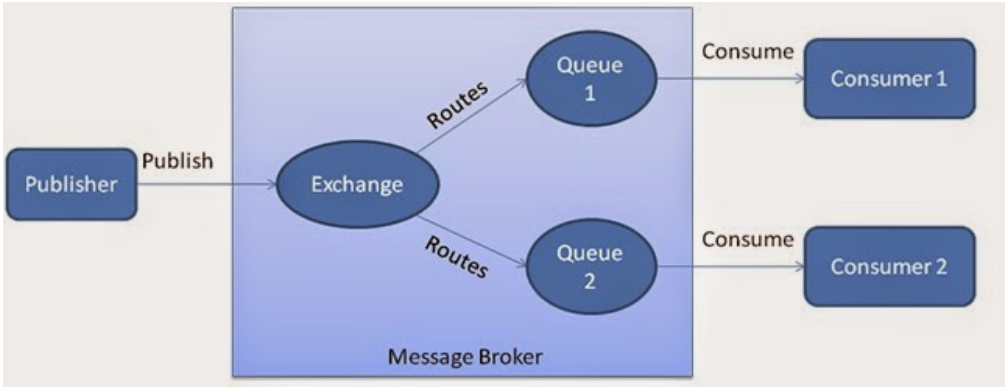

Figure A.1: Conceptual figure of Advanced Message Queuing Protocol [16]

As you can see, *publishers* (also called *producers*) send messages to the message broker that then routes them to *consumers* that process the messages. RabbitMQ is such a message broker.

A.1.1 Basic Concepts

The following terms are used to describe the basic concepts used for building such a data exchange infrastructure.

Producer This is the application that produces and sends messages to an exchange. It also declares the routing keys of a message.

Message A message is what is exchanged between applications via the protocol. The producer must provide them with a *routing key* that is then used for routing. In RabbitMQ, these are binary blobs of data.

Exchange Receives messages from producers and sends them further to queues. They must exactly know to which queue(s) a certain message should be forwarded. Four types of exchanges are offered:

- **Direct** delivers messages to queues based on the message routing key. A message goes to the queues whose binding key exactly matches the routing key of the message
- **Topic** routes messages to one or many queues based on matching between a message routing key and the pattern that was used to bind a queue to an exchange
- **Headers** designed for routing on multiple attributes that are more easily expressed as message headers than a routing key

- **Fanout** routes messages to all of the queues that are bound to it and the routing key is ignored

Queue Works as a buffer to store messages that can then be consumed by a consumer.

Consumer This is the application that receives the messages by consuming them from one or more queues.

Binding Represents a relationship between an exchange and a queue. Works as a rule an exchange uses (among other things) to route messages to queues.

A.2 Developing the Prototype

Before starting you should make sure to download and install RabbitMQ [12] on your computer, as explained in the installation guide. Make sure you can start the server like in figure A.2.

```
➜  ~ rabbitmq-server

              RabbitMQ 3.6.12. Copyright (C) 2007-2017 Pivotal Software, Inc.
  ##  ##      Licensed under the MPL.  See http://www.rabbitmq.com/
  ##  ##
  ##########  Logs: /usr/local/var/log/rabbitmq/rabbit@localhost.log
  ######  ##        /usr/local/var/log/rabbitmq/rabbit@localhost-sasl.log
  ##########
              Starting broker...
 completed with 10 plugins.
```

Figure A.2: Starting the RabbitMQ server

A.2.1 Preparation

To start, you should look at the provided tutorials [12] to find out what setup fits best for the specific use case. This can be done by looking at involved entities.

Producer and Consumer In our specific case we want to send reference data from the application ZEMIS Ref, the *producer*, to several other applications, the *consumers*. There is no need for an explicit answer message if the message sending is reliable. It is possible to include acknowledgements, so that the *producer* is sure the messages were received and can resend them if needed.

Messages - Routing Key You should also think of a practical way on how to route the messages so they reach all consumers needing the data. This enables you to define what type of *exchange* to use.

As we know (see section 2.3.1) reference data can either be owned by one single consumer or be commonly owned, which is defined on entity level. This means we already have an attribute that can serve as routing key, provided that the producer creates a separate message for each entity.

Exchange Because we have only one routing key per message and a limited amount of possible keys that can not spontaneously change (actually number of consumers plus one for *common*) we determined to use a *direct exchange*.

Setup Our setup matches the setup of tutorial 4 ”Routing” [14] as shown in figure A.3.

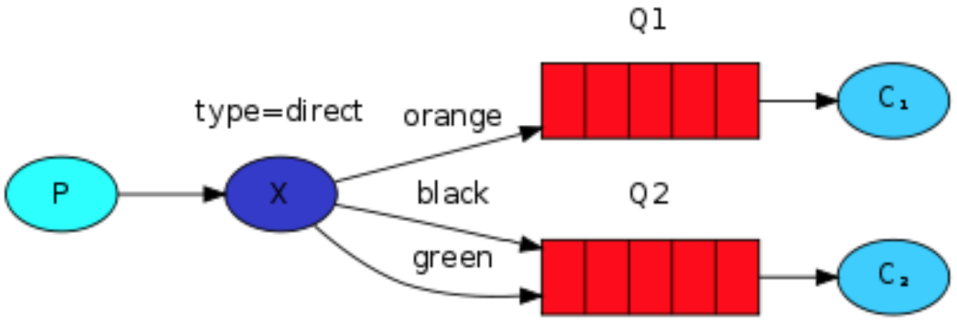

Figure A.3: Setup of the routing-approach [14]

The *producer* (P) will be ZEMIS Ref, the *exchange* (X) direct and the *routing key* (orange, black, green) for each message its entity’s owner. The work must not be distributed between several workers, as described in tutorial 2 [12], thus each *consumer* (C1, C2) will have it’s own *queue* (Q1, Q2). Each *queue* will have two binding keys: one for the name of the *consumer* it belongs to and one for `common`. These are compared with the routing key of each message, so that each queue and consumer only gets the data it needs. The consumer can access its queue any time later, as the messages are kept.

Example C1 is the application ZEMIS, its queue Q1 thus has the binding keys `zemis` and `common`. C2 is the application MIDES, Q2 thus has the binding keys `mides` and `common`. Now a record in the commonly owned entity `Zentrumsadressen` is changed. ZEMIS Ref thus creates a message with routing key `common`. The exchange forwards the message to Q1 and Q2, as they both subscibed to get messages with routing key `common`. When a record belonging to an entity owned by ZEMIS is mutated, only C1 will get this message.

A.2.2 Implementing the Prototype

For the producer as well as each consumer a java application is needed. They must contain certain parts: starting the transaction, create a connection and channel to use. This is done as in listing 21.

```
1 //Connection
2 ConnectionFactory factory = new ConnectionFactory();
3 factory.setHost("localhost");
4 Connection connection = factory.newConnection();
5 //Channel
6 Channel channel = connection.createChannel();
7 //Add a direct exchange
8 channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.DIRECT);
```

Listing 21: Basics for setting up producer or consumer

The next steps are not the same for consumers and producers, so we look at it separately.

A.2.2.1 Producer

For producing messages we use the structures defined to create our XML file.

For this prototype we want to call the main method with the `ID`'s of the entities we want to send in a message as arguments. We need to create a *response* with one entity for each given id.

```
1 for (String id : args) {
2    RefDataResponseType response = createResponse(id, em);
3    ...
4 }
```

Then, we have to serialise this response so that it can be sent as stream of bytes. This is done by usage of *Apache serialization utils* [1]. Of course, the response class must `implement Serializable` [3].

```
byte[] data = SerializationUtils.serialize(response);
```

Next, we have to set the *routing key*. As defined (paragraph A.2.1 Messages - Routing Key) we take the `joinedViewName` of the first (and only) `entity` in the `response`.

```
String key = response.getRefEntity().get(0).getJoinedViewName();
```

We now have to publish the previously defined byte array *data* with the defined `routing key` and publish it via the created `exchange`. This is the actual sending of the `message`.

```
channel.basicPublish(EXCHANGE_NAME, key, null, data);
```

A.2.2.2 Consumer

For the consumer you have to define which exchange to use for that channel and to create the queue and its bindings. We will use the same exchange as for the `producer`.

```
channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.DIRECT);
String queueName = channel.queueDeclare().getQueue();

channel.queueBind(queueName, EXCHANGE_NAME, "MIDES");
channel.queueBind(queueName, EXCHANGE_NAME, "COMMON");
```

This consumer represents the application MIDES and thus wants to get all messages with `routing key` "MIDES" or "COMMON".

Next you need to create the consumer itself that overrides the `handleDelivery` method and let it consume messages from the queue:

```
Consumer consumer = new DefaultConsumer(channel) {
      @Override
      public void handleDelivery(String consumerTag, Envelope envelope,
            AMQP.BasicProperties properties, byte[] body) throws IOException {
         RefDataResponseType response =
            (RefDataResponseType)SerializationUtils.deserialize(body);

   };
   channel.basicConsume(queueName, true, consumer);
```

We deserialise the content back to a `RefDataResponseType` that can be used by MIDES to update its database.

A.2.3 Conclusion

This prototype shows a basic example of how to use RabbitMQ as a replication mechanism. It works independent of the application architectures used and is secure and reliable. Various usage scenarios are possible, so that it can be adapted according to the needs. The effort of using a message oriented middleware such as RabbitMQ seems to be very small in comparison to a web service oriented architecture.

B
XML Schema Definition

This section shows the design of the XSDs for the prototypes.

B.0.1 Pull Architecture with Caching

The types defined in the XSD are described here, as well as their purpose.

generalRefDataRequest As the two defined requests share certain parts, we implemented this shared structure as a `generalRefDataRequest` (see listing 22) that is extended.

```
<xs:element name="generalRefDataRequest"
type="tns:generalRefDataRequestType"/>
<xs:complexType name="generalRefDataRequestType">
  <xs:sequence>
      <xs:element name="clientName" type="xs:string"/>
      <xs:element name="startDate" type="xs:dateTime" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

Listing 22: generalRefDataRequest as defined in the XSD

The `generalRefDataRequest` expects name of the client application as `clientName` . In the initial meta data model, this would be the `description` (see section 2.3.1 Meta Data Model). Also, because there might be influences that mess up the client's database, we figured that the client should be able to override the date from which on data changes shall be considered. So also the optional tag `startDate` that can be used to provide a date to use instead of the stored date of the last pull.

The two request messages extending the `generalRefDataRequest` can actually be used by the client applications. They are shown in listings 23 and 25 and shortly explained here.

allMyEntitiesRefDataRequest The `allMyEntitiesRefDataRequest` adds a `modifiedOrNew` (See paragraph B.0.1 modifiedOrNew) element that can be set to `true` to solely get records modified or newly created since the `startDate` or the stored date of the last pull, respectively. It is set to `false` per default and must occur at least `0` times, which means it is optional.

```
<xs:complexType name="allMyEntitiesRefDataRequestType">
  <xs:complexContent>
    <xs:extension base="tns:generalRefDataRequestType">
      <xs:sequence>
        <xs:element name="modifiedOrNew" type="tns:modifiedOrNewType"
        minOccurs="0" default="false"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Listing 23: allMyEntitiesRefDataRequest request as defined in the XSD

An example for a `allMyEntitiesRefDataRequest` from client system `MIDES` that pulls all values modified or newly created after the tenth of September, 2017 is shown in listing 24.

```
<allMyEntitiesRefDataRequest>
  <clientName>MIDES</clientName>
  <startDate>2017-09-10</startDate>
  <modifiedOrNew>true</modifiedOrNew>
</allMyEntitiesRefDataRequest>
```

Listing 24: Example of a pull request

multipleEntitiesRefDataRequest This element adds a sequence of elements to facilitate requests for any subset of entities. These `requestedEntity` elements must be used at least once but may occur `unbounded` times. The client may use the element `modifiedOrNew` (see B.0.1 modifiedOrNew) separately for each requested entity to only get the records modified or new for this entity.

```
<xs:element name="multipleEntitiesRefDataRequest"
type="tns:multipleEntitiesRefDataRequestType"/>
<xs:complexType name="multipleEntitiesRefDataRequestType">
  <xs:complexContent>
    <xs:extension base="tns:generalRefDataRequestType">
      <xs:sequence maxOccurs="unbounded">
        <xs:element name="requestedEntity"
        type="tns:requestedEntityType"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Listing 25: multipleEntitiesRefDataRequest request as defined in the XSD

The client can pull any subset of its entities by providing their names, as in listing 26. The client system `MIDES` requests its entities `Gemeinde` with all valid records and `Zentrumsadresse` with only the records modified or newly created since the tenth of September, 2017.

```
<?xml version="1.0" encoding="UTF-8" standalone ="yes"?>
<allMyEntitiesRefDataRequest xmlns="http://ejpd.admin.ch/sem/zemis
/refdataservice/types/v1">
<multipleEntitiesRefDataRequest>
  <clientName>MIDES</clientName>
  <startDate>2017-09-10</startDate>
  <requestedEntity>
    <entityName>Gemeinde</entityName>
    <modifiedOrNew>true</modifiedOrNew>
  </requestedEntity>
  <requestedEntity>
    <entityName>Zentrumsadresse</entityName>
    <modifiedOrNew>false</modifiedOrNew>
  </requestedEntity>
</allMyEntitiesRefDataRequest>
```

Listing 26: Example of a pull request for several entities

modifiedOrNew The element `modifiedOrNew` is defined in the XSD as follows (listing 27). The `type` is defined as `boolean` and thus an XML file using this element must send a value defined as boolean by the schema definition referenced with the namespace `xs` , these are `true` or `1` and `false` or `0`.

```
<xs:simpleType name="modifiedOrNewType">
  <xs:restriction base="xs:boolean"/>
</xs:simpleType>
```

Listing 27: modifiedOrNew Type defined in the XSD

It is optional for both requests and assumed `false` if not given. If set to `true` the client only gets the delta, meaning the records modified or newly created since his last pull. Having this element provides a possibility to not only pull deltas, but also all valid records, for example as initial pull or occasionally to make sure there are no inconsistencies. As different clients have different needs, we let the client choose what to get in the response.

refDataResponse The `refDataResponse` contains all data requested by the client. It was designed in the beginning, but then later redesigned, along with the types used for the response. The first version is shown here in listing 28, while the final version can bee seen in section Testing 4.3.1.

```
<xs:complexType name="refDataResponseType">
  <xs:sequence>
    <xs:element name="refEntity" type="tns:entityType"
    maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Listing 28: Definition of the refDataResponseType

The `refDataResponse` expects a sequence of `entityTypes` with name `<refEntity/>` .

entityType This element is defined as follows in listing 29. It consists of one (`minOccurs=1` is implicitly given) or two `table` elements (`maxOcurs=2`), as the entities are normally built from two tables, one containing the business values and the other containing the text in all languages. See the chapter meta data model 2.3.1 for further information.

```
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://ejpd.admin.ch/sem/zemis/refdataservice/types/v1"
    xmlns:tns="http://ejpd.admin.ch/sem/zemis/refdataservice/types/v1"
    xmlns:base="http://ejpd.admin.ch/bfm/zemis/base/types/v3"
    elementFormDefault="qualified"
    version="1.0">
    ...
<xs:complexType name="entityType">
  <xs:sequence>
    <xs:element name="table" type="tns:tableType" maxOccurs="2"/>
  </xs:sequence>
  <xs:attribute name="joinedViewName" type="xs:string" use="required"/>
</xs:complexType>
```

Listing 29: Definition of the entityType

In order not to have to send data multiple times, we designed the `entityType` in a way that supports sending the tables separately, but connected over their *joinedViewName*. For the relational tables the same structure is used. That is why the attribute `joinedViewName` is used for the *joinedViewName* of the entity provided in the meta data table `TARM_ENTITAET`. `xs:string` means it is a type string as defined in the schema provided by w3 [17].

tableType The tables consist of columns and rows, thus the `tableType` element has a sequence of `columns` and `rows` . It can have up to three `rows` elements, see paragraph B.0.1 rowsType for explanation. It also has two attributes, one to provide the `name` and the other, the `entityPart` , to declare which part of the entity the table forms, meaning whether it is a *business*, *text* or *relational* table. Only these values can be used, as the `entityPartType` element only accepts these, due to the `xs:restriction` element.

```
<xs:complexType name="tableType">
  <xs:sequence>
    <xs:element name="columns" type="tns:columnsType"/>
    <xs:element name="rows" type="tns:rowsType" maxOccurs="3"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="entityPart" type="tns:entityPartType" use="required"/>
</xs:complexType>

<xs:simpleType name="entityPartType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="business"/>
    <xs:enumeration value="text"/>
    <xs:enumeration value="relation"/>
  </xs:restriction>
</xs:simpleType>
```

Listing 30: tableType and entityPartType element definition

columnsType This type is used to declare the data types of the `rows` belonging to the table. It has a sequence of `col` elements. These have an attribute `dataType` used to tell the client what type the data in the `rows` has. See the definition in listing 31.

```
1  <xs:complexType name="columnsType">
2    <xs:sequence>
3      <xs:element name="col" type="tns:colType" maxOccurs="unbounded"/>
4    </xs:sequence>
5  </xs:complexType>
6
7  <xs:complexType name="colType">
8    <xs:simpleContent>
9      <xs:extension base="xs:string">
10       <xs:attribute name="dataType" type="tns:dataTypeType"/>
11     </xs:extension>
12   </xs:simpleContent>
13 </xs:complexType>
14
15 <xs:simpleType name="dataTypeType">
16   <xs:restriction base="xs:string">
17     <xs:enumeration value="ALPHA_NUM"/>
18     <xs:enumeration value="NUM"/>
19     <xs:enumeration value="BETRAG"/>
20     <xs:enumeration value="DATUM"/>
21     <xs:enumeration value="BOOLEAN"/>
22     <xs:enumeration value="RATIONAL"/>
23     <xs:enumeration value="CLOB"/>
24   </xs:restriction>
25 </xs:simpleType>
```

Listing 31: columnType and dataTypeType element definition

As you can see from, line 15 on (in listing 31) there are seven datatypes reference data can be associated with.

rowsType As this type is used to group *old*, *modified* and *new* `row` elements, it has the attribute `status` that only accepts these values and a sequence of `row` elements. For each status one group of rows is sent. See listing 32.

```
<xs:complexType name="rowsType">
  <xs:sequence>
    <xs:element name="row" type="tns:rowType" minOccurs="0" maxOccurs="
    unbounded"/>
  </xs:sequence>
  <xs:attribute name="status" type="tns:statusType" use="optional"/>
</xs:complexType>

<xs:complexType name="rowType">
  <xs:sequence>
    <xs:element name="entry" type="tns:entryType" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="index" type="xs:int" use="required"/>
</xs:complexType>

<xs:simpleType name="statusType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="old"/>
    <xs:enumeration value="new"/>
    <xs:enumeration value="modified"/>
  </xs:restriction>
</xs:simpleType>
```

Listing 32: rowType element definition

The `rowType` has `entry` elements that contain the data. It has a required `index` attribute that is used to count the rows.

Example Listing 33 shows an example for a response XML file that follows the schema definition described, thus the final version of the XSD.

```
<ns4:refDataResponse xmlns:ns4="http://..." xmlns="http://...">
  <ns4:refEntity joinedViewName="VARP_EMENDATION">
    <ns4:table name="VARP_EMENDATION_BS" entityPart="business">
      <ns4:columns>
        <ns4:col dataType="ALPHA_NUM">KEYCD</ns4:col>
        <ns4:col dataType="NUM">TAGESGERICHT</ns4:col>
        <ns4:col dataType="ALPHA_NUM">WAHLLOKAL</ns4:col>
        <ns4:col dataType="BOOLEAN">CTL_ACT_CD</ns4:col>
      </ns4:columns>
      <ns4:rows status="old">
        <ns4:row index="0">
          <ns4:entry>asw</ns4:entry>
          <ns4:entry>357766</ns4:entry>
          <ns4:entry>q</ns4:entry>
          <ns4:entry>1</ns4:entry>
        </ns4:row>
        ...
      </ns4:rows>
      <ns4:rows status="modified">
        <ns4:row index="0">
          ...
        </ns4:row>
        ...
      </ns4:rows>
      <ns4:rows status="new">
        <ns4:row index="0">
          ...
        </ns4:row>
      </ns4:rows>
      ...
    </ns4:table>
    ...
  </ns4:refEntity>
  ...
</ns4:refDataResponse>
```

Listing 33: Example of a response XML file

B.0.2 Push Architecture

The response structure defined for the *pull architecture with caching* (see B.0.1 refDataResponse) must only be renamed, as it should be a `refDataRequestType` instead of a `refDataResponseType` .

```
<xs:complexType name="refDataRequestType">
  <xs:sequence>
    <xs:element name="refEntity" type="tns:entityType"
    maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Listing 34: Definition of the refDataRequestType

The structure must not be adapted, and all elements defined for the *pull architecture with caching* can be reused. An additional `messageId` element might be useful to be able to map request messages to response messages, preferably in the soap header element [2]. This must then also be included in the according response message.

C
Declaration

”Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.”

D Interviews

D.1 Questions for Adrian Bürki

General Questions

Q: What are advantages of having your own tables?

A: Having my own tables is more flexible and probably a bit more lightweight. Do we really need bi-temporal data storage? If not, I can just do without. Do we really need categories and additional parameters? If not, no need to have it in the model. However using an existing component with its data model has advantages of its own, no need to reinvent the wheel.

Q: Would you like to use full pulls periodically or prefer deltas?

A: I'm perfectly fine with deltas. In case of an incident (inconsistency) an option to force a full sync would be nice though.

Q: How many users for Adeya/eRetour?

A: The application consist of four parts which will be build one after the other. For version one there will be two teams at SEM and a handful of specialist at the Cantons accessing the application. So there will be couple of hundred users. Later on the numbers will go up, but I assume it will never be much more than a thousand.

Q: Do you want to have new entries when they get active or before, including information about when they will be active?

A: I don't need to get entries before they are active. What I need though, are old entries which are inactive, since there may be old data referencing these non-active entries.

Variant 1: Push Architecture

A: I don't like the Hollywood approach (don't call us, we call you). I want to be in charge, I want to tell when to sync and what to sync. I don't want to wait for the push to happen. As far as I know we have kind of a pull is preferred over push policy from the architecture point of view. So I would expect opposition, if you have to present this solution in front of the architecture board.

A: What about on the fly adding entries from the client? Can I have my own UI in the client and push new entries to the central Ref-data store?

Q: Two-phase commit: Is voting an option, or too much effort? What do you think about it?

A: I don't quite get question about voting. The central Ref-data store is the master, why would you allow a client to vote against using the data?

Q: Commit and abort methods must be defined - is it a lot of effort?

A: Clients need to provide a web service to which Zemis-ref can push changes. That's all the web service must be able to do, plus giving an answer. You then must be able to add them to your database. What do you think about this?

Variant 2: Direct Pull Architecture

A: The web 3.0 approach, data is in the cloud and readily available. That works for all the weather apps out there, so why should it not work for us. We may have to do some caching in the browser, but all single page JavaScript apps can do that just fine.

A: The central Ref-data store has to guarantee, that no entries are deleted. Entries can be deactivated but never be deleted, so that the key can still be referenced. I don't mind loosing referential integrity on the database layer.

A: Same question as above, can I add new entries from the client? My users want to dynamically add new values if 'purple with a shade of grey' is missing from the list of hair color.

Q: You do not have your own tables, but always pull when you need a value. Is that an advantage?

- No need to store the data
- No need to update the data - always up to date
- No need to keep version tracking

Q: What about performance? Would your application be in trouble if it takes longer for a request to be fulfilled? If yes, do you have an idea how to avoid this?

Q: Do you think a backup solution is mandatory? What might be possible as backup solution?

Q: What about referential integrity?

Variant 3: Pull Architecture with Caching

A: That is the approach I prefer. The setup is quite similar to what we have in our internal code table solution, except that it is using web services instead of batches. On top of that I can cache the data in my own structure, no need to store everything if parts are not needed in my application.

Q: You do not have to implement a web service, but only a web service client to pull from the web service provided by Zemis-ref.

Q: You need to cache the data. Costs? Better to cache the data than always having to pull (including performance issues)?

Q: How often do you think pulls are needed?

A: We would be pulling at least once a day. Actually once every three hours would be realistic.

Variant 4: Push notify to pull

A: I kind of like that approach as well. Although we have to talk about technology, if you want to do notification broadcasts. Web services does not feel right, that begs for a messaging infrastructure - JMS topics comes to mind.

A: Reading through the question I think is too much of a hassle and a lot of work for a very limited use case. If an application requires real-time data, let them pull every five minutes, problem solved.

Q: How hard is it to implement a web service to get notifications?

- Web service client: only pull delta
- What format would be convenient for you? What could you imagine?

Q: What information is needed to determine whether a pull is required?

Q: How much effort is it to write code to determine whether to pull or not, depending on this information?

Q: How bad are the disadvantages?

- Clients need to offer a web service and a web service client interface (Zemis-ref also)
- Clients need to cache the ref data
- Clients need to determine whether to pull or not, depending on the information included in the notification
- Performance must be good even if all clients pull together which is almost guaranteed
- Requests can be answered with an offset in between
- Zemis ref needs knowledge for deltas (like time, version)

Q: Notification includes the following, do you agree?

- Information about the change, at least which table, so clients can determine whether to pull or not, at least for b)
- ID for the status change/notification to avoid unnecessary double-pulls

D.2 Questions and Answers: Didier Spicher

Q: Dürfen alle auf alle Tabellen zugreifen? Oder braucht es eine Einschränkung?

A: Mit ist besser

Q: Wie oft kommt es vor, dass auf Produktion Einträge für denselben Tag vorbereitet werden? Laut Marlène öfters, sie macht das so für sedex und Meldeverfahren, der Support anscheinend auch. Wäre es schlimm, wenn man Änderungen nicht am selben Tag publizieren könnte?

A: Kommt vor, sollte schnell sein. Auf Test sehr schnell (konfigurierbar?)

Q: Push: Schlimm, wenn es bei einem Client fehlt und dieser die Daten erst später hat? Two-phase commit nötig oder übertrieben?

A: Kurze Zeit nicht schlimm, so zB eine halbe Stunde ist kein Problem

Q: Ist die Historisierung so wie sie jetzt ist ok?

A: Habe ich noch nicht angeschaut

Q: Sollen Clients neue Einträge hinzufügen dürfen? Neue Tabellen?

A: Einträge in der eigenen Tabelle ok, neue Tabellen auch. Eher als Addon anschauen

Q: Dürfen Clients ihre eigenen Tabellen/Daten erfassen? Berechtigung, data owner? Wie sehr muss das eingeschränkt werden?

A: Nur eigene, nicht diejenigen von anderen Applikationen oder die COMMON-Tabellen

A: Zugriffberechtigung soll so bleiben wie bisher (GUI)

Bibliography

[1] SerializationUtils (Apache Commons Lang 3.7 API). https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/SerializationUtils.html, 2001-2017. Accessed: 2018-03-03.

[2] John Ibbotson. SOAP Version 1.2 Usage Scenarios. W3C note, W3C, July 2003. http://www.w3.org/TR/2003/NOTE-xmlp-scenarios-20030730/#S5.

[3] Serializable (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html, 2017. Accessed: 2018-03-03.

[4] Java API for XML Web Services. https://docs.oracle.com/javase/7/docs/technotes/guides/xml/jax-ws/, 2007. Accessed: 2018-03-03.

[5] Apache JMeter - Apache JMeter™. http://jmeter.apache.org. Accessed: 2018-01-24.

[6] The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 317. https://jcp.org/en/jsr/detail?id=317, 2018. Accessed: 2018-03-03.

[7] Martin Kalin. *Java Web Services: Up and Running: A Quick, Practical, and Thorough Introduction*. " O'Reilly Media, Inc.", 2013.

[8] Noah Mendelsohn, Yves Lafon, Henrik Frystyk Nielsen, Anish Karmarkar, Jean-Jacques Moreau, Martin Gudgin, and Marc Hadley. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). W3C recommendation, W3C, April 2007. http://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

[9] SR 142.314 Asylverordnung 3 vom 11. August 1999 über die Bearbeitung von Personendaten (Asylverordnung 3, AsylV 3). https://www.admin.ch/opc/de/classified-compilation/19994786/index.html#a1a, 2018. Accessed: 2018-03-03.

[10] OASIS. Advanced Message Queuing Protocol (AMQP) Version 1.0. http://docs.oasis-open.org/amqp/core/v1.0/amqp-core-complete-v1.0.pdf, 2012. Accessed: 2018-03-02.

[11] RabbitMQ - Messaging that just works. https://www.rabbitmq.com, 2007. Accessed: 2018-03-02.

[12] RabbitMQ - Getting started with RabbitMQ. https://www.rabbitmq.com/getstarted.html, 2007. Accessed: 2018-02-28.

[13] RabbitMQ - AMQP 0-9-1 Model Explained. https://www.rabbitmq.com/tutorials/amqp-concepts.html, 2007. Accessed: 2018-03-02.

[14] RabbitMQ - RabbitMQ tutorial - Routing. https://www.rabbitmq.com/tutorials/tutorial-four-java.html, 2007. Accessed: 2018-03-01.

[15] REST & SOAP Automated API Testing Tool — SoapUI. https://www.soapui.org. Accessed: 2018-01-24.

[16] Spring tutorials: Rabbit MQ basic concepts. http://springguide.blogspot.ch/2015/03/rabbit-mq-and-spring-amqp-basics.html, 2015. Accessed: 2018-03-01.

[17] XML Schema. http://www.w3.org/2001/XMLSchema, 2014. Accessed: 2018-02-26.

[18] Verordnung über das Zentrale Migrationsinformationssystem (ZEMIS). https://www.sem.admin.ch/sem/de/home/aktuell/gesetzgebung/archiv/vo_zemis.html, 2007. Accessed: 2018-03-03.