Migration of a web service back-end from a relational to a document-oriented database

Sebastian Drenckberg¹, Marius Politze²

¹ IT Center RWTH Aachen University, Seffenter Weg 23, 52074 Aachen, drenckberg@itc.rwth-aachen.de
² IT Center RWTH Aachen University, Seffenter Weg 23, 52074 Aachen, politze@itc.rwth-aachen.de

Keywords
relational database, migration, document-oriented-database, MongoDB, LINQ, SQL

1. ABSTRACT
Evolutionary changes are often used to gradually increase the quality of a software. However, if back-end systems have to be replaced with a different technologies, the resulting changes are more radical. One example for such a change is the migration of database servers from a relational to a document-oriented storage engine. Based on an example application, we present an approach to perform such a migration and conclude some general guidance for migrating future applications.

2. INTRODUCTION
Due to increased mobility and the rising number of students, universities have to standardize existing processes, improve cooperation between institutions, reduce overall costs and increase efficiency. Additionally, students have changed the demands on their universities and its employees regarding universities as education service providers. To better support learning, teaching and research, process supporting IT systems were introduced as mentioned by Bischof (Bischof, Gebhard, & Steves, 2005) and Barkhuus (Barkhuus & Dourish, 2004). Generally, IT infrastructure and applications are becoming more important to the universities’ processes and employees and to students and their daily life. This leads to increased competition among the universities to present the best and most appealing services to their students. There are many examples of small-scale developments that support individual learning use cases. Some of many examples are the works of Ebert (Ebert & Haupt, 2015), Breitkopf (Breitkopf, Grau Turuelo, & Banos Garcia, 2017) or Küppers (Küppers, Politze, & Schroeder, 2017). Often these services are instantiated by a student or research project but have to be maintained by universities technical personnel on the long term. Without considering enhancements, maintenance costs alone can be high to keep the services operational and migrate them towards new technologies. One of these services of Aachen University is an Audience Response System (ARS) used to support large-scale lectures with more than one thousand participants. The ARS is currently supporting more than 40 lectures and other events. While most of them are traditional lectures in a single lecture hall, some are broadcasted live to a remote room or even to multiple locations like the students’ homes. These and other aspects make each lecture unique in some way. This is why the feedback feature was developed in a way that can be easily adopted and extended to fit different scenarios (Politze, Decker, Schaffert, & Küppers, 2015). Nevertheless, as part of the application lifecycle, the technological basis needs to be migrated so the service can be continuously operated.

In the current infrastructure, depicted on the left in Figure 1, the database server poses a single point of failure. An off site backup protects from data loss but allows neither automatic fail-over nor scaling. Like many modern real time web applications, this scenario requires scalable application and database software architectures. Our goal therefore is to migrate the application to a more scalable topology that uses replication in order to distribute data store and access to all available nodes, as displayed on the right in Figure 1.

Currently the application uses Microsoft SQL Server to store data. To communicate with the database, application servers use LINQ to SQL as an Object Relational Mapper (ORM). Past analysis however have shown that the current technology stack does not meet the posed requirements in terms of scalability and reliability (Drenckberg, 2016). The database should thus be migrated from Microsoft SQL Server...
to MongoDB. Changing the database back-end, however, also affects parts of the application logic. Based on the example of the ARS, a general approach shows standard cases when migrating from a relational to a document-oriented database model. As there are many services using the infrastructure, the goal is to generalize these cases to develop guidelines for migration of these other services (Politze, Schaffert, & Decker, 2016).

3. EXISTING STRUCTURE

As a relational database, the current SQL Server supports the use of constraints. This means that there can be references between datasets in the database. Using an ORM further allows accessing the referenced datasets directly via the object model.

The database consists of five relations depicted in Figure 2. The relation Channel contains information about the events or lectures that use the ARS. A Channel has multiple Messages send by the users. A Message in turn can have multiple tags used by teachers for filtering. The relation MessageTag is used to realize this $n:m$ relationship in the current database model. Last but not least a Message can contain an image attachment stored in the Photo relation.

---

**Figure 1:** Database topologies with a single dedicated server and multiple shared servers

**Figure 2:** Structure in database before migration
The ORM LINQ to SQL offers a programming interface that integrates seamlessly into the LINQ language extensions offered by the C# programming language. The code sample below shows a database query using LINQ to get all channels associated with a user:

```csharp
public static List<DASChannel> GetAllChannelsForUser(string[] personGguids) {
    using (var context = new DASDataContext()) {
        var channels = from c in context.DirectFeedbackChannels
                        orderby c.Name
                        where personGguids.Contains(c.OwnerGuid)
                        select c;

        return channels.Select(c => new DASChannel(c, true)).ToList();
    }
}
```

This roughly translates into an SQL statement like

```
SELECT * FROM DirectFeedbackChannel WHERE OwnerGuid IN ('...', '...', '...')
```

which is then being processed by the database server. The sample above also shows the connection handling offered by the ORM. A DataContext allows addressing the relations and attributes directly from code. As such, the compiler checks statements written in LINQ for syntax and type.

4. DATABASE MIGRATION

Obviously, migration of the database engine also requires migration of the object mapper and database connection classes. It is however required that the general functionality of the migrated class structure retains the described properties like accessing with LINQ and compile time syntax checking. The database migration is therefore performed in three steps:

1. Analysis and simplification of the current database model,
2. Conversion of database connection and object models, and
3. Validation and migration of existing data.

4.1. Analysis and Simplification

To simplify the current structure, the three relations MessageTags, Tags and Photos were removed and integrated into the message structure. The document for channels retains its initial structure. Thus, after the migration of the data structures, only two kinds of documents are stored in the database: DASMessageObj and DASChannelObj.

In addition to the previous attributes, such as the channel ID and the actual message text, tags are stored directly in the message document. The photo has also been integrated into the message document. Figure 3 shows the document structure after simplification.

![Figure 3: Simplified Message and Channel documents](image-url)
4.2. Conversion

Obviously, the LINQ to SQL ORM can no longer be used to access MongoDB. The MongoDB.Driver that comes with the database, however, includes connectors for the database. It furthermore offers the ability to access documents using LINQ statements by using generics and thus provides basic object mapping functionalities. The generic mapping functions are being strictly typed in a custom connector class, DASDataConnector, in turn becoming a new object mapper to access the data from code. A separate class, MongoDBConnector, handles the connection to the database and thus is reusable for other strictly typed implementations.

The migrated Connector now offers a similar interface to query data from the database. As shown in the code fragment below, after instantiating the class, documents can be queried directly again using LINQ Language features such as compile time syntax checks.

```csharp
public static List<DASChannel> GetAllChannelsForUser(string[] personGguids) {
    var connector = new DASConnector();
    var channels = from c in connector.ChannelsQueryable
                    where personGguids.Contains(c.OwnerGuild)
                    select c;
    return channels.Select(x => new DASChannel(x, true)).ToList();
}
```

This time the MongoDB.Driver translates the query into a MongoDB specific query language basing on the JavaScript Object Notation (JSON).

As in the case of SQL, the database driver sends these queries to the database server and the result set is transferred to the client.

Comparing both methods, before and after the migration, it is easy to see that only the instantiation of connector classes has changed, but not neither the return type nor the actual LINQ statements. Thus, changes made to the database layer do not propagate further in the application. Similarly, the context in all other classes belonging to the Audience Response System has been replaced.

Because of the simplified structure, queries that used constraints must additionally be adapted. In the Audience Response System, these were only used to access the relevant channel from a message. These functions have been modified in a way that the corresponding channel must be retrieved first. Afterwards, the associated messages can be found and returned.

4.3. Validation

After migrating the database and application logic, it is necessary to validate that the provided functionalities still work as expected. Generally, a set of tests should validate that the interfaces provided still function according to the specification.

For the Audience Response System, unit and integration tests are already available and can serve this purpose. For the validation a total of 35 tests, displayed in Figure 4, in three categories: Messaging, Surveys and Channel administration were used.

![Figure 4: Results of unit tests for messages, surveys and channels](image-url)
5. GENERALIZATION

In general, normal forms serve to separate data in logical units, so that duplicate entries and inconsistencies are avoided. However, this often entails a higher computational effort because data must be collected from different relations. Additionally, relational database systems require some workarounds, such as intermediate relations, to implement many-to-many relations according to normal forms. The migration to document-oriented databases allows simplifying these workarounds by allowing lists, optional content and hierarchical structure in the documents.

Based on the three possible archetypes of relations (Schicker, 2017) the following guidelines can be summarized:

In the case of a 1 : 1 relation, there are two possibilities. On the one hand, a reference can remain as in the relational database model, using a foreign key. The referenced data then is stored in an additional document. On the other hand, they can be embedded directly into another like shown in Figure 5.

**Figure 5: 1 : 1 relation in an ER diagram (left) or modelled by embedding documents (right)**

The 1 : n relationship requires a new attribute to be added to the document referencing the n-other records. This attribute then holds a list of the complete documents. In some cases, this may be impractical: especially if records can be referenced from, multiple types of documents it is likely beneficial to fall back to referencing IDs like displayed in Figure 6.

**Figure 6: 1 : n relation in an ER diagram or modelled by linking documents**

The most complex case of a relation is the n : m relation. A relational database requires the usage of an intermediate table to references the different data records while adhering normal forms. This is necessary since lists in a single attribute contradict the requirement of attributes being atomic. As in the previous case, lists can, however, be used in document-oriented databases to model the relationship. As shown in Figure 7 the same pattern as in the previous case can be used.

**Figure 7: n : m relation in an ER diagram or modelled by linking documents with lists of IDs**
6. CONCLUSION

The migration of a web service back-end to a different database system should be well considered. Without a basic concept, it is not possible to successfully perform such a migration without service interruptions. The new database system should be evaluated through test scenarios, which also consider future use cases and requirements. Looking at availability and scalability, in the case of the Audience Response System, **MongoDB** presented as a viable choice.

In general, already existing structure of the database relations must be analyzed and adapted to the new system. Subsequently, optimization points of this structure can be characterized. Because **MongoDB** is not a relational but a document-oriented database system, it was possible to simplify workarounds used to adhere normal forms: For example, typical intermediate relations can be removed as it is possible that multiple references can be directly stored in documents. Furthermore, it is also possible to combine or remove tables by hierarchically integrating information from multiple relations into a single document.

Migration to a document-oriented database system additionally allows an optimization of the applications, since each document can be individually modified or extended in its structure. This allows applications to change the stored data more evolutionary and thus increases the overall maintainability of the software.

The migration process furthermore showed the importance of automated unit and integration tests to validate successful migration. As minimal acceptance criteria, these tests allowed comparing the software before and after the migration. The additional effort of having not only unit tests but also integration tests that validate the system, as a whole pays off when urged to perform changes to back-end systems.

Looking at this first use case, the migration was very successful. Compared to the old technology stack, it did not only increase long-term maintainability of the software but also reduced overall resource consumption. The set of generalized guidelines further ease future migrations planned in the near future.

7. REFERENCES


8. AUTHORS’ BIOGRAPHIES

Sebastian Drenckberg, B.Sc. is software developer at the IT Center of RWTH Aachen University since 2017. In 2017, he finished his B.Sc. studies in Scientific Programming at FH Aachen University of Applied Sciences and his apprenticeship as a mathematical-technical software developer.

Marius Politze, M.Sc. is research associate at the IT Center RWTH Aachen University since 2012. His research is focused on service-oriented architectures supporting university processes. He received his M.Sc. cum laude in Artificial Intelligence from Maastricht University in 2012. In 2011, he finished his B.Sc. studies in Scientific Programming at FH Aachen University of Applied Sciences. From 2008 until 2011, he worked at IT Center as a software developer and later as a teacher for scripting and programming languages.