



MASTERARBEIT | MASTER'S THESIS

Titel | Title

Analysis of (cloud-based) storage systems or services and
design of a storage framework for a FactChecking prototype

verfasst von | submitted by

Mohamad Day BSc

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of
Master of Science (MSc)

Wien | Vienna, 2026

Studienkennzahl lt. Studienblatt | Degree
programme code as it appears on the
student record sheet:

UA 066 921

Studienrichtung lt. Studienblatt | Degree
programme as it appears on the student
record sheet:

Masterstudium Informatik

Betreut von | Supervisor:

Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Klas

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Klas, for his invaluable advice and guidance throughout my thesis, and to my family for constant support and encouragement.

Abstract

In this master's thesis, we addressed the data storage requirements of the FactChecking prototype (FC). The main goal is to find suitable storage solutions and integrate them into FC. We started by analyzing well-established storage solutions. The focus of the analysis was to answer the questions about data type, database format (SQL, NoSQL, etc.), and database implementation (MongoDB, PostgreSQL, etc.) that should be used to store the data in the storage framework of FC. Additionally, we analyzed the storage services provided by the public cloud platforms and discussed the differences to on-premise solutions. The database solutions were evaluated regarding various aspects like performance, scalability, and costs. Based on the results of the analysis, we designed and implemented the storage framework for FC. The database services MongoDB and Cosmos DB have been integrated prototypically into FC, allowing the evaluation and flexible use of multiple database systems. The developed platform enables managing the database connections from the three supported types (MongoDB, CouchDB, and Cosmos DB), and supports migrating the data between these connections. Furthermore, it allows specifying which database connection is used by FC. Finally, we evaluated the implementation and the integrated database types by conducting performance tests. The results of the tests showed that MongoDB outperformed CouchDB and Cosmos DB across all tested operations.

Kurzfassung

Diese Masterarbeit befasst sich mit der Datenspeicherung im Factchecking-Prototyp (FC). Das Hauptziel ist, geeignete Datenspeicherlösungen zu finden und in FC zu integrieren. Zunächst analysierten wir die etablierten Datenbanken. Die Analyse konzentriert sich darauf, die Fragen zu den Datentypen, den Datenbankformaten (SQL, NoSQL usw.) und den Datenbankimplementierungen (MongoDB, PostgreSQL usw.), die für die Speicherung der Daten auf der Speicherplattform des FC eingesetzt werden können, zu beantworten. Außerdem analysierten wir die Cloud-Speicherservices und deren Unterschiede zu On-Premises-Lösungen. Die Datenbanklösungen wurden hinsichtlich verschiedener Aspekte wie Performance, Skalierbarkeit und Kosten evaluiert. Auf Grundlage der Analyseergebnisse haben wir das Speicherframework für FC entworfen und implementiert. Die Datenbanklösungen MongoDB und Cosmos DB wurden prototypisch in FC integriert, um die Evaluierung und den flexiblen Einsatz verschiedener Datenbanksysteme zu ermöglichen. Die implementierte Plattform ermöglicht die Verwaltung der Datenbankverbindungen für die drei unterstützten Datenbanktypen (MongoDB, CouchDB und Cosmos DB) und unterstützt die Datenmigration zwischen ihnen. Außerdem ermöglicht sie, die Verbindung auszuwählen, die vom FC verwendet werden muss. Abschließend haben wir Performance-Tests durchgeführt, um die Implementierung sowie die integrierten Datenbanksysteme zu evaluieren. Die Testergebnisse haben gezeigt, dass MongoDB bei allen getesteten Operationen besser abschneidet als CouchDB und Cosmos DB.

Contents

Acknowledgements	i
Abstract	iii
Kurzfassung	v
List of Tables	xi
List of Figures	xiii
Listings	xv
1. Introduction	1
1.1. Research Questions	2
1.2. Organization of the Thesis	3
2. Background & State of the Art	5
2.1. Background (FactChecking Project)	5
2.1.1. Entity Resolution	6
2.1.2. Fact Set Comparison	6
2.1.3. FactChecking Architecture	8
2.2. State of the Art	8
2.2.1. MongoDB	8
2.2.2. Cosmos DB	11
2.2.3. CouchDB	13
2.2.4. Redis	14
2.2.5. Neo4j	15
2.2.6. PostgreSQL	16
3. Approach / Design	19
3.1. Requirements	19
3.1.1. Functional Requirements	19
3.1.2. Non-Functional Requirements	20
3.2. System Design & Taken Design Decisions	21
3.2.1. DBManager Backend	22
3.2.2. DBManager Frontend	45
3.2.3. DBManager Database	49
3.2.4. FactServer Changes	52

4. Implementation	57
4.1. DBManager Backend	57
4.1.1. Main	57
4.1.2. App	59
4.1.3. Blueprints (Rest Controllers)	59
4.1.4. Data Model	63
4.1.5. Database Repositories	65
4.1.6. Views	67
4.1.7. Connection Managers	69
4.1.8. Migration Managers	77
4.1.9. Jobs	87
4.1.10. Logging	89
4.1.11. Config	91
4.1.12. Unit Test	93
4.2. DBManager Frontend	101
4.2.1. Services	102
4.2.2. Models	104
4.2.3. Components	106
4.2.4. Environments	110
4.3. Factserver	111
4.3.1. Databases Handlers	111
4.3.2. Config DB	127
4.3.3. Config Yaml	128
4.4. Deployment	129
4.4.1. Containerization	129
4.4.2. Source Code & How to Run the System	131
5. Experimental Evaluation	133
5.1. Performance Analysis	133
5.1.1. Test Environment	133
5.1.2. Migration Performance	134
5.1.3. Insertion Performance	135
5.1.4. Query Performance	136
5.2. User Experience	137
5.2.1. Connections	138
5.2.2. Migrations	141
5.2.3. Factserver Connections	147
6. Conclusion	151
6.1. Summary	151
6.2. Limitations	151
6.3. Future Work	152
Bibliography	153

A. Appendix	161
A.1. GIT README	161
A.1.1. Master Thesis Title	161
A.1.2. Project Description	161
A.1.3. Project Components	161
A.1.4. Use Cases	162
A.1.5. Deployment	162

List of Tables

3.1. Environment variables for DBManager backend	41
4.1. Environment variables used for testing	93
5.1. Migration: performance comparison by dataset size	134
5.2. Insertion: performance comparison by dataset size	135

List of Figures

2.1. The official logo of MongoDB	8
2.2. The official logo of Cosmos DB	11
2.3. The official logo of CouchDB	13
2.4. The official logo of Redis	14
2.5. The official logo of Neo4j	15
2.6. The official logo of PostgreSQL	16
3.1. Component diagram of DBManager and the interface usage by FactServer	21
3.2. Class diagram of connections blueprints	23
3.3. Class diagram of factserver and migration blueprints	24
3.4. Class diagram of connections and DBInfo	26
3.5. Class diagram of FactserverConnection and Migration	27
3.6. Class diagram of connection managers classes	29
3.7. Class diagram of indices managers	30
3.8. Class diagram of MigrationManager and Migrator	32
3.9. Class diagram of Migrator subclasses	33
3.10. Activity diagram of the migration process	34
3.11. Class diagram of jobs	35
3.12. Class diagram of repository classes	37
3.13. Class diagram of DummyDataManager	38
3.14. Class diagram of utils classes	39
3.15. Class diagram of configuration classes	42
3.16. Class diagram for views, app, and main	43
3.17. Class diagram of the whole DBManager backend	44
3.18. Class diagram of the frontend services	45
3.19. Class diagram of the frontend models	46
3.20. Class diagram of the frontend visual components	48
3.21. Diagram of the database tables	50
3.22. ER-Diagram of the database	51
3.23. Class diagram for FactServer enhancements	53
3.24. Class diagram for DBHandler implementations	54
3.25. Class diagram for FactServer data classes	55
4.1. The structure of imported design documents	74
4.2. The changes done to use DBFactory in facts_controller.py	127
5.1. Migration: performance comparison by dataset size	134

List of Figures

5.2. Insertion: performance comparison by dataset size	135
5.3. The start page of the application: MongoDB connections	138
5.4. The actions behind the options button for connections	138
5.5. The dialog for details	139
5.6. The confirmation dialog when clicking on the drop button	139
5.7. The dialog for adding test/dummy data	140
5.8. The confirmation dialog before adding test/dummy data	140
5.9. The dialog To edit existing connection	140
5.10. The notifications shown when updating a connection	141
5.11. The table of migrations	141
5.12. The actions behind the options button for migrations	142
5.13. The dialog showing the details of a successful migration process	142
5.14. The dialog showing the details of an erroneous migration process	143
5.15. The progress bar: 7%	144
5.16. The progress bar: 20%	144
5.17. The progress bar: 40%	144
5.18. The progress bar: 68%	145
5.19. The progress bar: not all items are migrated	145
5.20. The dialog for creating a new migration process	146
5.21. The selection of the processing date of a migration process	146
5.22. The check for different source and target	147
5.23. The table for active factserver connections	147
5.24. The table for factserver connections, including history	148
5.25. The dialog for creating a new factserver connection	148
5.26. The confirmation dialog before saving a new factserver connection	149

Listings

1.1. JSON-LD example	2
2.1. Example of JSON-LD where first name is called givenName	6
2.2. Example of JSON-LD where first name is called firstName	6
4.1. Registering of blueprints in main.py	58
4.2. Database setup in main.py	58
4.3. Cron jobs setup in main.py	58
4.4. The start of the jobs scheduler and the Flask app in main.py	59
4.5. Setup in app.py	59
4.6. Setup of a blueprint for cosmosdb connections, and a GET method	60
4.7. The swagger file for the method get_connection in cosmosdb_bp	60
4.8. POST method to create a new cosmosdb connecetion	61
4.9. The method create_or_update_connection	62
4.10. The method @errorhandler for error handling in cosmosdb_bp	63
4.11. The class DatabaseType representing the table database_types with two fields (id and type)	63
4.12. The method insert_static_types in class DatabaseType is used to insert static rows during the start of the application	64
4.13. The class MongoClientConnection representing the table mongodb_connections	64
4.14. The method to_dict used to return a json representation of the object . .	65
4.15. The method used to return the next Migration object for the processing job	66
4.16. The method increase_migration_processed_items_number used to update the value of the field processed_items_number	66
4.17. The method store_connection_and_set_valid_until_for_existing_conn used create a new factserver connection	67
4.18. The list of views used by the system	67
4.19. The method create_views iterate over the views list and creates the views	69
4.20. The drop views script	69
4.21. Snippet from AbstractDbManager	70
4.22. Snippet from MongoDbManager	71
4.23. Snippet from mongodb-indices	72
4.24. Snippet from CouchDbManager	73
4.25. Snippet from couchdb_views.py: import_couch_design_docs	75
4.26. Snippet from couchdb_views.py: create_views_for_couch_db	76
4.27. The design document ids.json for database cache	76
4.28. The method start_migration in MigrationManager	77

4.29. The file <code>migrators.py</code>	78
4.30. The method <code>get_migrator</code>	78
4.31. The method <code>is_applicable</code> returns a boolean value based on source and target types	79
4.32. The method <code>migrate</code>	80
4.33. The annotation <code>log_migration_execution</code>	81
4.34. The method <code>process_collection</code>	82
4.35. The method <code>preprocess_collection_migration</code>	83
4.36. The method <code>postprocess_collection_migration</code>	84
4.37. The implementation of the abstract methods of <code>Migrator</code> in <code>CouchDb-</code> <code>ToMongoDbSeqMigrator</code>	85
4.38. The implementation of <code>migrate_collection</code> in <code>CouchDbToMongoDbSeqMigrator</code>	86
4.39. The job <code>migration_processing_job</code>	87
4.40. The job <code>failing_migration_detector_job</code>	88
4.41. The registration of jobs in <code>main.py</code>	88
4.42. The configuration of logging in <code>logging_utils.py</code>	89
4.43. The method <code>log_migration_event</code>	89
4.44. The method <code>log_collection_event</code>	90
4.45. The method <code>log_completed_collection_migration</code>	90
4.46. The class <code>config</code>	91
4.47. The method <code>load_config</code>	91
4.48. The method <code>get_dummy_migration</code>	94
4.49. The method <code>get_local_couch_manager</code>	94
4.50. The method <code>assert_migration</code>	95
4.51. Snippet from <code>CosmosDbManagerTest</code>	97
4.52. Snippet from <code>MongoDbManagerTest</code>	98
4.53. The test class for the method <code>failing_migrations_detector_job</code>	99
4.54. The test class <code>TestCosmosDbToMongoDBSeqMigrator</code>	101
4.55. The abstract class <code>ConnectionAPI</code>	102
4.56. The class <code>MongodbApi</code> extending <code>ConnectionApi</code>	103
4.57. Snippet from <code>MigrationApi</code>	103
4.58. The interfaces <code>Connection</code> and <code>DatabaseType</code>	104
4.59. The interface <code>MongodbConnection</code>	104
4.60. The interface <code>MigrationRequest</code>	105
4.61. The interface <code>Migration</code>	105
4.62. The interface <code>Field</code>	106
4.63. The interface <code>ConnectionsTable</code> and its required input fields	106
4.64. The programmatic definition of the actions for connections in <code>Connection-</code> <code>sTable</code>	107
4.65. The use of <code>ConnectionAPI</code>	107
4.66. The dynamic definition of the connections table in <code>html</code>	108
4.67. The component <code>CosmosdbConnections</code>	109
4.68. The use of <code>connection-table</code> in the <code>cosmosdb-connections.html</code>	110

4.69. The file environments.prod.ts	110
4.70. Snippet showing the use of environments	110
4.71. The replacement configuration for production	111
4.72. The methods close, __enter__, and __exit__	112
4.73. The constructor of class MongoDBHandler	112
4.74. The method connect in MongoDBHandler	113
4.75. The method get_record in MongoDBHandler	114
4.76. The method execute_query in MongoDBHandler	114
4.77. The method get_facts_by_name_and_type in MongoDBHandler	116
4.78. The method run_mongo_query in MongoDBHandler	117
4.79. The method get_facts_by_name_and_type in CosmosDBHandler	117
4.80. The method run_cosmos_query in CosmosDBHandler	118
4.81. The method create_database in MongoDBFactory	119
4.82. The constructor of the class ProxyDBHandler	120
4.83. The method connect in ProxyDBHandler	121
4.84. The method run_update_in_interval in ProxyDBHandler	122
4.85. The method check_config_and_update_connection in ProxyDBHandler .	122
4.86. The delegation of methods in ProxyDBHandler	124
4.87. The factory for ProxyDBHandler	125
4.88. The class DBFactory	126
4.89. The file config_db.py	127
4.90. The class MongoDBConnection	128
4.91. The introduced configs in default.yaml	129
4.92. Snippet from docker-compose.yml	130
4.93. Snippet from .env	131
4.94. Snippet from docker-compose.yml: the service mongodb	131

1. Introduction

The very massive amount of information provided on the internet contains many contradicting data because of many reasons, such as outdated data or intentional manipulation attempts. Many people often face contradictory data without knowing about related information on other web pages that may contain the correct version of the provided information.

The FactChecking (FC) project aims to mitigate the issue of conflicting data by giving users hints about related information existing on other web pages with a consistency/similarity indicator. Conflicts are detected by comparing structured data sets related to the same entity.

To achieve the goals of the FC project, a massive amount of data collected from information found on the web pages should be stored in the FC database. This database is called the FactBase. In addition, other types of data need to be stored for different uses, like data history and feedback. For any application, dealing with different data types and a vast amount of data is a big challenge when considering key factors like performance, scalability, data privacy, and costs.

This motivates the analysis of a (cloud-based) storage system and services, including the design and the development of an appropriate storage framework for the FC prototype. The framework will provide support for managing multiple database connections of different database types and migrating the data between the databases. In addition, it will manage the connection used by the FactServer.

1. Introduction

1.1. Research Questions

These topics can be divided into the following research questions:

RQ1: Which data format is suitable for the different use cases? Should we use only one database format, like relational (SQL), non-relational (NoSQL), graph database, or a combination of data formats/database formats, considering multiple decision factors like scalability, dependency, data access, costs, complexity, and the use cases of the related data?

To address this question, we first need to understand the three different types of data structures:

- **Structured data:** this is data that follows a defined schema, like data stored in a relational database (SQL). The strictness of the schema of this data structure constitutes a limitation for many use cases.
- **Semi-Structured data:** this is data that is self-described and does not strictly follow a schema. Self-description allows for easier data processing, whereas non-strictness provides flexibility. The most well-known data formats are XML and JSON.
- **Unstructured data:** this data does not follow a schema and is not self-descriptive. Examples are raw texts, images, videos, or audio.

Regarding the storage of facts in FC, the main focus is on semi-structured data, specifically on JSON-LD [1], which is a JSON format used to represent linked data as entities. Listing 1.1 shows an example of JSON-LD:

```
1 {  
2   "@context": "https://json-ld.org/contexts/person.jsonld",  
3   "@id": "http://dbpedia.org/resource/John_Lennon",  
4   "name": "John Lennon",  
5   "born": "1940-10-09",  
6   "spouse": "http://dbpedia.org/resource/Cynthia_Lennon"  
7 }
```

Listing 1.1: JSON-LD example taken from [1]

Semi-structured data is most suitable as there is no schema uniquely defined for the facts found on web pages. For example, fact sets related to the same entity may contain different attributes based on the facts provided on each page. Using strictly structured data like relational data (SQL) would be almost impossible or very restrictive, as each entity will be represented by a specific set of attributes, namely the table's columns. In addition, defining the attributes of each entity would be challenging. Another benefit of semi-structured data or NoSQL is that the storage solutions are more flexible and scalable than structured data storage solutions [2]. Having only one kind of structured data like the feedback data does not justify the use of a separate SQL database because the NoSQL databases can also store structured data and define schema restrictions, another reason would be the huge amount of complexity for operating and dealing with a database combination within one application.

RQ2: Given a data format and a database technology like (JSON/NoSQL), which database implementation is most suitable? There are multiple database implementations ideal for the same data format. Still, each implementation has advantages and disadvantages, which will be investigated and considered when making the decision.

RQ3: Given a specific database implementation like PostgreSQL (SQL), what are the benefits from hosting this database on the cloud (if available) or on-premise (if available)? What are each implementation's advantages, disadvantages, and restrictions, and how significant are they?

To address RQ2 and RQ3, a detailed analysis of various database solutions was conducted, focusing on their key features. The results of the analysis are presented in Chapter 2. Based on the results, promising database solutions will be integrated into the existing implementation of FactServer.

RQ4: How can multiple database solutions be integrated into the FactServer to provide flexible support for all of them?

The details of the analysis, design, and implementation are presented in the chapters 3 and 4.

RQ5: How can database connections be managed, and how can data be efficiently migrated between the supported database systems? This functionality will improve the flexibility in switching between supported databases and facilitate their evaluation.

To address this, we will design a tool able to manage and migrate the data between the supported database systems. The details of the design and implementation are introduced and explained in the chapters 3 and 4.

1.2. Organization of the Thesis

Chapter 1 introduces the problem and the motivation by highlighting the need for a suitable data storage solution in FC, and presents the main research questions of this master's thesis. In Chapter 2, the FC project is briefly introduced, in Section 2.1, by describing its history, core functionalities, and architecture. In Section 2.2, the well-established database implementations are analyzed with a focus on their key characteristics, such as performance, security of the stored data, and available cloud support. Chapter 3 introduces the functional and non-functional requirements of the system, outlines the system design, and discusses the design decisions. The implementation details of each part of the system are introduced and discussed in detail in Chapter 4. The results of the evaluation of the system's performance and user experience are presented in Chapter 5. Finally, Chapter 6 lists the limitations of this work and outlines directions for future research.

2. Background & State of the Art

In this chapter, Section 2.1 briefly introduces the FactChecking project, including its core functionalities and architecture.

In Section 2.2, we discuss the different database solutions and their advantages/disadvantages, and the provided cloud support, considering the primary storage requirements of FC. The focus will be on NoSQL databases, as the objects representing the fact sets are in JSON format and require a database solution with document support.

2.1. Background (FactChecking Project)

This section briefly introduces the FactChecking project, which aims to develop an approach addressing the issue of contradictory data on the internet.

The project was established in the faculty of computer science at the University of Vienna by the research group Multimedia Information Systems (MIS). Its structure and functionalities are continuously improved and developed by students and members of MIS. The FactChecking project aims to mitigate the issue of conflicting data by giving users hints about related information on other web pages with a consistency/similarity indicator. Conflicts are detected by comparing structured data sets related to the same entity.

The following describes the recent functionalities and structure of the FC project. The description is mainly based on [3].

One JSON-LD object related to an entity is considered a set of facts (fact set), and each property of the object is called a fact [3]. Each fact is represented by two things:

- Property name: like age, name, firstName, nationality, etc...
- Property value: like 56, Max Mustermann, Max, Austria, etc...

The entity, the fact property name, and the fact property value can be seen as a relation where:

- The entity is the head or tail of the relation
- The fact property name is the relation
- The fact property value is the tail or head of the relation

In general, FC backend does not only store and manage the fact sets (FactBase) but also other types of data [3] like:

- User data (UserBase)
- Information about outdated or historical data (HistoryBase)
- Feedback data (FeedbackBase)

2. Background & State of the Art

2.1.1. Entity Resolution

In the scope of FC, an entity can be anything: a person, an organization, a city, etc.... Each entity is represented as a set of properties called facts. An entity related to a specific thing is called a named entity, for example, the city of Vienna or Al-Khwarizmi¹. To allow finding similar entities or fact sets to compare with a fact set found on a web page, FC needs to give fact sets related to the same-named entity the same ID. Just relying on a property of the fact set, like the name, as the ID of the entity can not be really useful, as we will see in Subsection 2.1.2.

FC will detect named entities on web pages and assign a specific ID to solve this issue. This process is called entity resolution. This internal ID will be called the **mid** in the scope of FC. To perform the task of giving the ID to the named entity, FC relies on Google Knowledge Graph², which is a tool provided by Google.

2.1.2. Fact Set Comparison

Comparing fact sets can be seen as one of FC's most important core functionalities. When the user visits a web page (let's call it A), the entity found on this page will be called the local entity. FC will load all fact sets related to the same named entity from the storage backend server of FC using a defined API. These loaded entities are called remote entities.

```
1 {  
2   "@context": "https://schema.org",  
3   "@type": "Person",  
4   "givenName": "Max",  
5   "lastName": "Mustermann",  
6   "job": "Student"  
7 }
```

Listing 2.1: Example of JSON-LD where first name is called givenName

```
1 {  
2   "@context": "https://schema.org",  
3   "@type": "Person",  
4   "firstName": "Max",  
5   "lastName": "Mustermann",  
6   "job": "Student"  
7 }
```

Listing 2.2: Example of JSON-LD where first name is called firstName

¹A mathematician from the 8th and 9th centuries

²See: <https://developers.google.com/knowledge-graph?hl=de>

2.1. Background (FactChecking Project)

Using JSON-LD can make the pairwise comparison of the facts of the fact sets easier, but this process is still hard due to many factors like synonyms; for example, `familyName` and `lastName` are semantically the same and usually contain the same data. Other examples of comparison issues are:

- The different date formats, like 10.10.2010 and 2010.10.10, or the textual format of the date.
- How significant a difference in values is, for example, when talking about the population of a big city like Vienna, differences in the least significant digit might be tolerated, but again, this heavily depends on the use case and the context.

As shown in the Listings 2.1 and 2.2 and described above, simply using string equality is a very restrictive comparison method and will dramatically increase the error rates.

Any comparison result can be one of the following:

1. True positive: the facts are conflicting, and the conflict was detected.
2. True negative: the facts are not conflicting, and no conflict was detected.
3. False positive: the facts are not conflicting, but a conflict was detected.
4. False negative: the facts are conflicting, but no conflict was detected.

From a statistical point of view, there are two important errors [4]:

- False positive shown in the above list item 3 is called a type I error.
- False negative shown in the above list item 4 is called a type II error.

Two important properties of any true/false test are:

- Sensitivity: the true positive rate over all items being really positive.

$$\text{Sensitivity} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}}$$

- Specificity: the false negative rate over all items being really negative.

$$\text{Specificity} = \frac{\text{True Negatives (TN)}}{\text{True Negatives (TN)} + \text{False Positives (FP)}}$$

As we can see, an increase in Type I or Type II errors causes a decrease in specificity or sensitivity, which means a decrease in the test quality, in our case, the comparison quality.

To accommodate these issues, FC performs the fact comparison task using precision metrics, a combination of similarity predicates aware of the context [3]. Using the similarity predicates should help recognize equivalent formats, so that values like 2.03.2022 and March 2, 2022, are not considered conflicting data.

The results of comparing the facts of two fact sets (A and B) can be one of the following:

- Quality: means the fact exists in A and B, and the value is similar.
- Conflict: means the fact exists in A and B, and the value is not similar.
- Local missing: means value exists in B but is missing in A.
- Remote missing: means value exists in A but is missing in B.

2. Background & State of the Art

2.1.3. FactChecking Architecture

The architecture of FC can be split into 3 building blocks:

- Frontend: FC has 3 front applications:
 - The browser extension **IdaFix**: for visualizing the result of the comparison task.
 - Content provider dashboard: for statistics.
 - The management dashboard: for user management.
- Middleware: contains the core logic of the FC. It covers functionalities like:
 - Entity resolution.
 - Precision metrics.
 - Data comparison.
- Backend: covers data storage, data processing, and data extraction.

2.2. State of the Art

In this section, a detailed analysis of various database solutions was conducted, focusing on their key features.

2.2.1. MongoDB



Figure 2.1.: The official logo of MongoDB [5]

MongoDB (Figure 2.1) is a document database that supports the storage of JSON objects. It is highly scalable and reliable thanks to the replication and sharding mechanisms it uses. In addition, MongoDB has very flexible indexing support and provides powerful data querying and aggregation capabilities.

Replication means that multiple copies of the same data set are stored on different servers to ensure high availability, fault tolerance, and disaster recovery. Having multiple copies of the same data on various servers makes read operations faster when each read request is forwarded to the closest server. A set of instances responsible for managing data replication is called a replica set. In the replica set, only one node receives write requests, which is called the primary node, and all other nodes are called secondary nodes.[6]

Sharding is the concept of distributing the data across multiple servers to allow horizontal scalability. This is needed in case the data volume and the workload are growing, and machine capabilities are limited or can not be extended, i.e., vertical scaling is unwanted or unachievable. Data is split into shards, and each shard is represented by a replica set.[7]

Internally and over the network, MongoDB stores JSON objects in a binary representation called BSON, which stands for (Binary JSON).

According to MongoDB [8], BSON has many advantages over normal string-encoded or binary-encoded blob JSON, such as:

- Supporting various types of numeric data.
- Defining data types for dates and binary data.
- Types and length information are also encoded, which allows faster traversal of the objects.
- High-performance indexing and querying.
- Being independent of the different object representations in different programming languages.

MongoDB Tools

MongoDB provides **MongoDB Compass** [9], which is a free, modern, and powerful tool with the following capabilities:

- Managing MongoDB connections.
- Managing MongoDB databases.
- Exploring the data.
- Creating visual models representing the structure of the data.
- Analyzing the data schema and providing insights like ranges and frequencies of the fields.
- Providing execution plans for queries and advanced performance metrics.
- Supporting the creation of validation rules for collections.
- Enabling natural language queries through an AI-powered feature.

TO address the users of traditional relational databases, MongoDB provides a tool called **Relational Migrator** [10]. This tool helps its users identify potential risks of a migration from SQL databases like Oracle and PostgreSQL, and can perform migrations to a MongoDB database that is hosted on the cloud or on-premises.

The tool **Mongosync** [11] is provided by MongoDB to migrate data between different MongoDB instances. This tool should help users migrate their data from on-premises instances into the cloud, especially into MongoDB Atlas.

In addition, the following extensions/plugins are provided to support integration with AI-agents:

- **MongoDB Agent Skills.**
- **MongoDB Gemini extension.**
- **MongoDB Cursor plugin.**
- **MongoDB Claude plugin.**

2. Background & State of the Art

- **MongoDB extension for GitHub CoPilot.**

MongoDB Atlas

MongoDB Atlas is a multi-cloud platform for MongoDB. The Atlas database is a MongoDB database hosted on the cloud. Using Atlas eliminates the operational overhead for developers and allows them to benefit from the advantages of the cloud. Atlas offers multi-cloud and multi-regional deployment of the database. This ensures scalability and fault tolerance. The term multi-cloud refers to the ability to scale and replicate the database across the cloud service providers GCP, AWS, and Azure at the same time. Atlas is responsible for managing the resources of the database with automated adjustment capabilities, which guarantees high performance and low latency. Atlas promises high security standards by applying access controls and data encryption. To access and manage the Atlas database, the user can either use the web interface or the CLI for Atlas.[12]

Pricing: Atlas database offers three pricing plans [13]:

- **Free:** 512 MB in a shared environment, to be used for testing and learning.
- **Flex:** resources are shared with other users and scale automatically based on usage. Storage is limited to 5 GB, and the service is intended for development and testing. Monthly costs depend on usage and typically range from \$8 to \$30, with similar pricing across the supported cloud providers.
- **Dedicated:** resources are not shared between users; the user has to choose one of the provided tiers. Storage starts from 10 GB and scales to 4 TB and beyond. Scaling over 4 TB is achieved through horizontal scaling. Pricing depends on the tier configuration and the selected provider. In general, the monthly costs range from \$57 for 10 GB to \$1195-\$33746 for 4 TB based on the computation resources (vCPUs, RAM) and the cloud provider.

Additionally, Atlas, as a platform, offers the following services [13]:

- MongoDB Search on Atlas.
- MongoDB Vector Search on Atlas.
- Atlas Stream Processing.
- Atlas Data Federation.
- Atlas Charts.
- Atlas SQL Interface.

2.2.2. Cosmos DB



Figure 2.2.: The official logo of Cosmos DB [14]

Cosmos DB (Figure 2.2) is a database solution provided by Azure and used for AI services that need dynamic scale, like ChatGPT. It can be used as a NoSQL (document, graph, key-value), relational, or vector database by selecting one of the database APIs provided by Azure. Azure ensures Cosmos DB's automatic scalability and single-digit millisecond response times. As Cosmos DB is fully managed and totally hosted on Azure Cloud infrastructure, the database instances can be hosted in the data centers that are close to the user or application accessing the data, making it easier to achieve low latency and high availability. Additionally, database administration is done by Cosmos DB by doing automatic management and updates, like capacity management.[15][16]

The following APIs are provided by Cosmos DB [17], making it a desirable alternative to existing database solutions, allowing developers to migrate existing data without the need to change the data model or the application accessing the data:

- NoSQL API (Cosmos DB native API)
- MongoDB
- PostgreSQL
- Apache Cassandra
- Apache Gremlin
- Table

When moving to Cosmos DB from another solution supported by the provided APIs, like MongoDB, only the connection string (host URL) needs to be changed, which makes the move process very smooth and allows the user to benefit from all features provided by Cosmos DB with a very minimal effort.

According to Azure [17], all APIs guarantee the main features of Cosmos DB, such as:

- Flexibility
- Scalability

2. Background & State of the Art

- Availability
- Performance
- High Throughput
- Low latency
- Global distribution

Azure recommends using the native NoSQL API when developing a new application for the following reasons [17]:

- Full control over the interface, service, and client libraries.
- New features are first rolled out for this API.
- Support for SQL syntax.
- Support for analytics.
- Performance isolation between analytical and operational workloads.

Pricing: The costs of Cosmos DB are divided into 3 parts [18]:

- Compute: the cost of computation needed to perform the database operations. It's measured by a unit called Request Units (RU). It covers the CPU, memory, and IO operations.
- Storage: The cost of storing data based on its size. It is measured in GB fractions and covers the business data and all database-related data, like indices.
- Bandwidth: The cost of transferring data. It covers the transfer between Azure regions or from/to the Azure cloud.

Azure provides the tool Capacity Calculator that helps to estimate the needed provisioned throughput.

The tool estimated the costs for 10 GB and about 50 read requests per second with about \$ 25,86 per month. The cost for 4 TB storage was estimated to be around \$ 1050. When enabling multi-region storage and scaling to 2 regions, the costs grow accordingly to \$ 2150. This is because the cost of storage is twice as much as before.

2.2.3. CouchDB



Figure 2.3.: The official logo of CouchDB [19]

Apache CouchDB (Figure 2.3) is a document database introduced in 2005 [20]. It supports storing JSON elements and retrieving them based on the ID. Communication with a CouchDB instance is done using the HTTP REST API provided by CouchDB. The data format used in the body for almost all requests and responses is JSON [21]. For querying documents, the API supports returning all documents or a specific document using its ID. CouchDB provides the ability to execute complex queries by writing views using MapReduce. CouchDB provides reliability, scalability, high availability, and fault tolerance properties by applying replication and partitioning mechanisms. Sharding is transparent to the user, which means that the user (i) does not need to define and (ii) can not change the sharding logic. In contrast to MongoDB, there is no standard cloud support for CouchDB like MongoDB Atlas. Another benefit that is provided by MongoDB Atlas and is missing in CouchDB is the cross-provider clusters.[20]

2. Background & State of the Art

2.2.4. Redis



Figure 2.4.: The official logo of Redis [22]

Redis (Remote Dictionary Server) (Figure 2.4) is a NoSQL key-value database where the focus is on speed. To achieve a very fast response time, the data is stored in memory. Redis fits very well for real-time applications and as a caching service.[23]

One restriction of Redis is that the data must fit in the available RAM. This means that a large amount of data can not fit into a Redis database.

Data Types & Redis Search

Although Redis is a key-value store, it does not only support strings as values. Rather, Redis supports complex data types, such as:

- Lists
- Sets
- Vector set
- JSON

and provides an advanced search tool called **Redis Search**. Redis Search offers enhanced search for specific data types like JSON and vectors, which allows Redis to act as a document database or a vector database. For example, Redis Search supports querying JSON objects using JSONPath syntax.[24][25]

Persistence

Despite being an in-memory database, Redis provides the ability to store data permanently. This is very important when a server needs to be restarted or any hardware fails. Depending on the use case, a Redis database can have one of the following persistence strategies [26]:

- RDB (Redis Database): snapshots of the database are stored persistently at configurable intervals. Interval tuning is a tradeoff between performance and lower

risks.

- AOF (Append Only File): storing of a change history. Any change is logged persistently, allowing for a backup by applying the stored change logs.
- RDB + AOF: (RDB) backups at configured interval + (AOF) change history from last backup.
- No persistence: for caching.

Redis Products

Redis provides three options to use its product [27]:

- Redis Software: the enterprise version of Redis; the user needs to take care of operational management.
- Redis Cloud: the database is fully managed on a public cloud service provider.
- Redis Enterprise for Kubernetes: the enterprise version of Redis, ready to be deployed on a Kubernetes cluster.
- Redis Open Source: the open source version of Redis.

2.2.5. Neo4j



Figure 2.5.: The official logo of Neo4j [28]

Neo4j (Figure 2.5) is a NoSQL graph database with a native graph implementation. A native graph implementation means that the data is stored in a graph structure. The implementation can not be native when the database only provides an abstraction simulating a graph database.[29] The data model of a graph database consists of three components [30]:

- Node: represents one entity, like a person, enterprise, or student. Each Node can be assigned to a set of labels to allow classification of nodes like (Patient). A node owns a set of properties where the data about this entity is stored. A set of constraints can be applied to a node to enforce a specific data structure.
- Relationship: represents a relationship between two nodes. A relationship is directional and has exactly one type, like "has", which is the class of this relationship. Similar to a node, a relationship can have a set of properties.
- Property: a key-value pair used to store data about a node or a relationship.

The main benefits of using this structure are scalability, flexibility, and supporting the traversal of the graph in depth and breadth. Graph databases are suitable for use cases where the focus is on the relations between the entities.[30] For example, when

2. Background & State of the Art

the short path of relationships between two entities is requested, a graph database will provide full support for such operations, whereas solving this in a traditional relational database using join operations could be very complicated.

Deployment

Neo4j offers two ways to use its database:

- Self-managed deployment: data owner is responsible for operating the database.
- Fully-managed cloud (Aura): the database is deployed in one of the public cloud providers: GCP, AWS, or Azure. The storage capacity is adjustable at any time, with the option to pay upfront or use a pay-as-you-go model.[31][32]

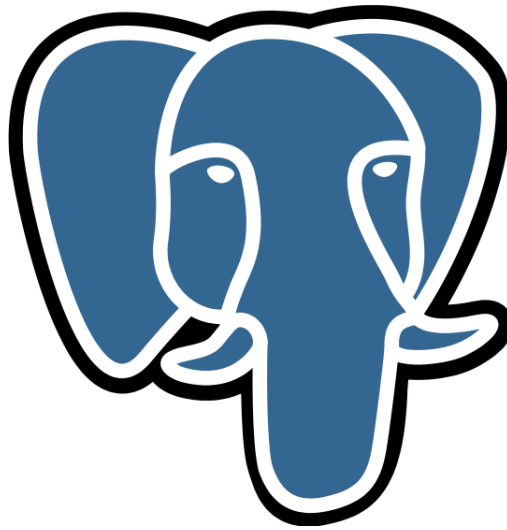


Figure 2.6.: The official logo of PostgreSQL [33]

2.2.6. PostgreSQL

PostgreSQL (Figure 2.6) is an open-source SQL database that has been in development for the last 35 years.[34]

SQL databases like PostgreSQL are intended to store data with a strict schema in relations (Tables). Tables usually have a primary key, which is a column or a set of columns with a unique value(s) representing the ID of the rows. Relationships between tables are represented using the foreign key, which is a reference from one row in a table to a row in another or the same table.

Constraints

To enforce a strict schema, each table can define a set of constraints to be checked for each row; the following are some constraints that can be defined [35]:

- Unique: a value of a specific column is not allowed to appear in more than one row.
- Not Null: ensure the field is always set for each row.
- Check: performs specific checks on the value to be stored, like an equality check or a range check.

Triggers

To enhance control over the stored data, PostgreSQL and other SQL databases have the concept of triggers. Triggers are procedures that are triggered by events like create, update, or delete operations on a specific table.[36] A database procedure is a sequence of SQL statements that are executed and can be called with input parameters and have an output [37], just like functions in programming languages.

Indexing

To offer fast access to a table, SQL databases support the indexing of specific fields. For each created index, the database will store a mapping of the values of that field into the IDs of the corresponding rows using a data structure like a B-Tree. Depending on the field type and the operations executed on it, other data structures like Hash or GIN might be more suitable than the B-Tree, which is the default one. Once an index is created, the filtering, sorting, and joining operations on the table using this field are much faster. Based on the size of a table, an index will consume storage space and can negatively affect the performance of insert, update, and delete operations. The reason for that is that the index needs to stay up to date after each operation.[38][39]

Partitioning

Another way to enhance the scalability and performance in PostgreSQL is partitioning. Partitioning means that one logical table is split into multiple smaller physical tables. Inserted data is assigned to partitions based on the value of a specific field, called the key value.[40] There are 3 forms of partitions defining how a row is assigned to a partition [40]:

- Range Partitioning: when the key value must be inside the range of values of the partition. For example, the key value is between (01.05.2026 and 31.05.2026).
- List Partitioning: when the partition has a list of acceptable key values. For example, the key value is in (1,2,8).
- Hash Partitioning: when the hash of the key value must match the hash of the partition. For example, $\text{modulo}(\text{key value}, 5) = 3$.

Accessing a partitioned table using the partition key is very fast, as only a part of the data must be scanned, which is the data stored in the corresponding partition(s). Another benefit is that the drop operation on a partition is much faster than the delete operation.[40] An example of a common use case is deleting data older than a specific date; in this case, the deletion can be achieved very easily by dropping the whole partition when the table is partitioned by the creation date.

2. Background & State of the Art

Data Types

Originally, SQL databases were not developed for storing complex or semi-structured data such as lists, XML, or JSON objects. However, PostgreSQL supports the definition of XML or JSON as the datatype of a field. It also provides powerful querying support to access the content of objects stored in:

- XML: using XPath [41].
- JSON: using JsonPath [42].

Cloud Hosting

Public cloud providers such as AWS and GCP provide support for database as a service. The service **Cloud SQL** offered by GCP allows its users to access a database hosted and managed totally by the cloud provider. The user benefits from a database service provided by a cloud provider in terms of initial hardware and software setup effort, flexibility, and maintenance. In the cloud, increasing the storage space, the number of (v)CPUs, or the memory (RAM) can be done with a few clicks only. The same applies to creating data backups. GCP restricts the storage space for PostgreSQL to about 70 TB. The tool price calculator, provided by GCP, can estimate the costx for a database based on the database type and edition, storage size, backup size, computing resources, number of instances, type of support, and the commitment of usage. The tool estimated the costs for hosting a database with 8 vCPUs, 32 GB of RAM, 4 TB of storage using SSD storage devices, with about 1.038\$ per month.

Using cloud-managed databases has some disadvantages, especially with regard to Long-term costs, compliance, and data privacy. The full control of the data is not in the hands of the data owner anymore.

Why not SQL

NoSQL databases were explicitly developed to solve the problem of storing substantial amounts of semi-structured data in a scalable and distributable way. Hence, they are preferred over SQL databases for storing documents like XML or JSON objects.

3. Approach / Design

This chapter introduces the functional and non-functional requirements of the system, outlines the system design, and discusses the design decisions taken.

3.1. Requirements

This section introduces the functional and non-functional requirements of the designed system.

3.1.1. Functional Requirements

1. The user can add a new database connection by selecting the desired database implementation supported by the system. The user should provide the required connection information, like URL, username, password, etc. Different databases can have different connection requirements.
2. The user can manage the stored connections:
 - The user can see a list of all database connections available in the system.
 - The user can modify or remove existing database connections in the system.
 - The user can see storage information about all database connections defined in the system.
 - The user can empty any collection of any database connection defined in the system.
 - The user can drop any collection of any database connection defined in the system.
 - The user can insert dummy data in a new or any collection of any database connection defined in the system.
3. The user can migrate data collections from one database connection to another by selecting the source and target connections. The source and target connections can be from different database implementations (MongoDB, CouchDB, Cosmos DB).
4. The user can see information about the executed migration processes, like start date, end date, duration, and the number of migrated items.
5. The user can define the database connection to be used by the FactServer¹.
6. The FactServer can connect to the configuration database to get the type and credentials of the connection it must use.
7. The Factserver is extended to be able to connect to MongoDB and Cosmos DB and prototypically support queries for the **facts** collection.

¹<https://git01lab.cs.univie.ac.at/factcheck/core/factserver>

3. Approach / Design

3.1.2. Non-Functional Requirements

- Performance: since we are expecting a huge amount of data, read and write operations need to be very fast, especially the read operations, as this will always be done when data needs to be fetched for comparing facts during user interactions.
- Scalability: the data amount is expected to grow continuously; for that, the storage framework should be able to store and accept huge amounts of data without requiring complex action from the application developer to allow horizontal scalability².
- Storage space: data duplication and invalid data should be avoided as much as possible.
- Security: data stored in the framework should be protected from any unauthorized access, and all available data security measures should be taken during any data processing task.
- Simplicity: the framework should allow easy and uncomplicated access to the data, regardless of the type of database used for storing the data.
- Integrability: the design and implementation should support the open-closed principle. This means the integration of a new database type should be done with minimal effort without the need to modify the existing implementation massively. This allows the project to be a good base for future work.

²To read more about horizontal scalability: <https://www.mongodb.com/resources/basics/scaling>

3.2. System Design & Taken Design Decisions

The developed DBManager tool consists of 3 components:

- Angular³ frontend application
- Python⁴ backend application
- PostgreSQL⁵ database to store migration-related data.

In addition to the DBManager tool, changes on the existing FactServer⁶ project, written in Python, have been made to allow supporting multiple database types.

Figure 3.1 illustrates the components of the system.

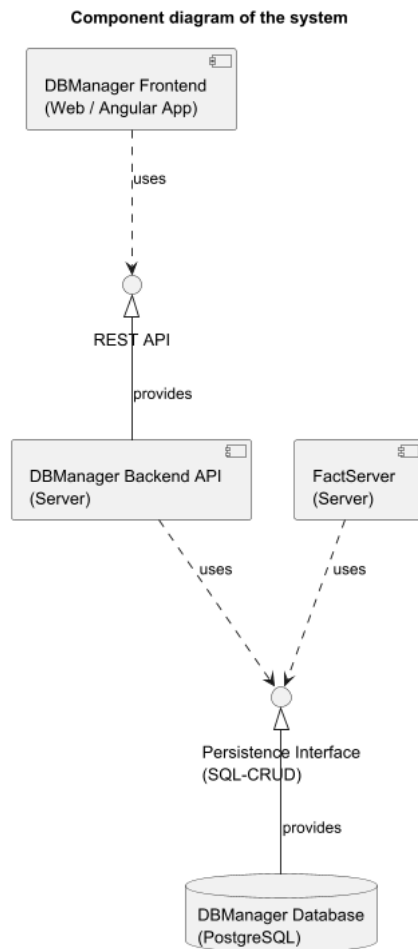


Figure 3.1.: Component diagram of DBManager and the interface usage by FactServer

³<https://angular.dev>

⁴<https://www.python.org>

⁵<https://www.postgresql.org>

⁶<https://git01lab.cs.univie.ac.at/factcheck/core/factserver>

3. Approach / Design

3.2.1. DBManager Backend

Programming language Python 3.12 was used to implement the backend server. The design of this server follows, to a certain extent, the Layered Architecture with Top Down communication, which means that the top layer(endpoints layer) is responsible for receiving requests coming from clients and forwarding them to the layer directly below (logic layer), the logic layer will, in turn, process the request data and finally forward the requests to the lowest layer, which is the layer responsible for communicating with the database to store new connections or a new migration process, or to fetch related data. One important property of this pattern is that security checks can be done in all layers, making the app more secure [43]. Design patterns such as the Abstract Class and Template Method were applied to structure the system's design and address its functional requirements, such as defining a common processing workflow for migration processes while allowing database-type-specific customization of some specific tasks.

The backend server can be seen as a collection of packages, where the classes in each package are logically similar to each other. The main packages are:

Blueprints (Rest Controllers)

This package represents the presentation layer of the backend server and is responsible for the communication with the frontend application. The application has five Blueprints, three of which are for managing connections, one for migration processes, and one for managing the connection to be used by the FactServer:

- `cosmosdb_bp`: used to manage credentials of Cosmos DB Connections.
- `couchdb_bp`: used to manage credentials of CouchDB Connections.
- `mongodb_bp`: used to manage credentials of MongoDB Connections.
- `migration_bp`: used to manage migration processes.
- `factserver_bp`: used to manage the connection to be used by the FactServer.

The blueprints of all connections (figure 3.2) and the migration blueprint (figure 3.3) provide CRUD (create, read, update, and delete) operations. The blueprint for FactServer connections (figure 3.3) provides only create and read operations.

A Swagger specification is provided, which documents all available endpoints, their request parameters, and their corresponding responses. The complete Swagger specification is available at: `<app-host>/apidocs/`. (See Appendix A.1 to know how to start the system)

3.2. System Design & Taken Design Decisions

Class diagram of connections blueprints (REST-Controller)



Figure 3.2.: Class diagram of connections blueprints

3. Approach / Design

Class diagram of factserver and migration blueprints (REST-Controller)



Figure 3.3.: Class diagram of factserver and migration blueprints

Data Model

This package contains the classes that represent the data transferred, processed, or stored within the backend system. The package consists of four main components:

- **connections:** represents the data needed to connect to a specific database, such as host and credentials (Figure 3.4).
 - **Mongodb-Connection:** represents connection data for a MongoDB.
 - **Couchdb-Connection:** represents connection data for a CouchDB.
 - **Cosmosdb-Connection:** represents connection data for a Cosmos DB.
- **db-info:** represents insights about collections of a database connection, such as name, size, and number of items (Figure 3.4).
- **migrations:** represents the data of a migration process, such as source connection, target connection, start date, finish date, and the creator of the migration (Figure 3.5).
- **factserver-connections:** represents a reference to one of the stored connections that must be used by the fact server (Figure 3.5).

The model classes for migrations, connections, and factserver-connections serve as ORM models. All connections have a name and a type.

3. Approach / Design

Class diagram of connections and dbInfo

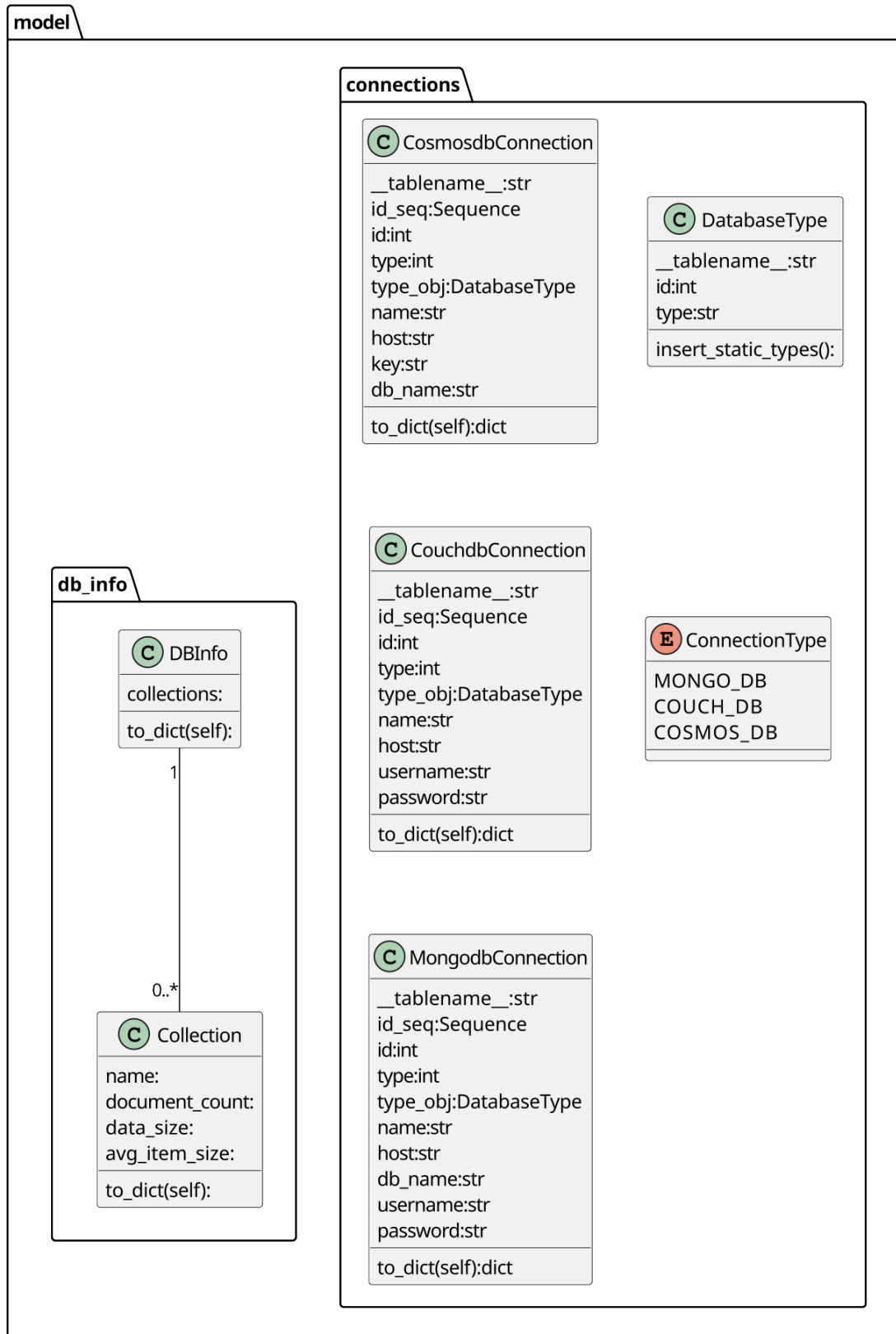


Figure 3.4.: Class diagram of connections and DBInfo

Class diagram of factserverConnection and migration dataclasses



Figure 3.5.: Class diagram of FactserverConnection and Migration

3. Approach / Design

Connection Managers

This package contains the classes that represent the managers of database connections (Figure 3.6). The class **AbstractDBManager** defines an abstract interface for interacting with a database connection. This abstraction ensures a shared interface across all database managers, which enables the use of different database implementations without requiring changes to code.

The declared methods of a database manager class include:

- **connect:** to establish a connection and store the connection client.
- **establish_connection:** to establish a connection.
- **change_connection:** to change the connection.
- **close_connection:** to close the connection.
- **has_collection:** to check if a connection has a specific data collection.
- **get_collection:** to return a specific data collection.
- **get_collection_size_or_0:** to return the number of items in a collection.
- **empty_collection_for_migration:** to empty, or drop and recreate a data collection used as a target collection.
- **get_connection_info:** to return insights about all data collections of a connection.
- **create_collection_indices:** to create indices for a specific data collection.

The classes **couchdb_views** and **mongodb_indices** provide helper functions that are used by the method **create_collection_indices** to create indices/design-documents for the corresponding database types (Figure 3.7). The configured indices can be updated in the code if needed. In addition, **couchdb_views** provides a method **import_couch_design_docs** to import and store the CouchDB design documents of a specific connection. Imported design documents will be used to create the design documents when calling the method **create_views_for_couch_db**. Currently, the configured indices are only for the following factserver data collections:

- facts
- cache
- factstats
- historical-stats

Class diagram for connection managers

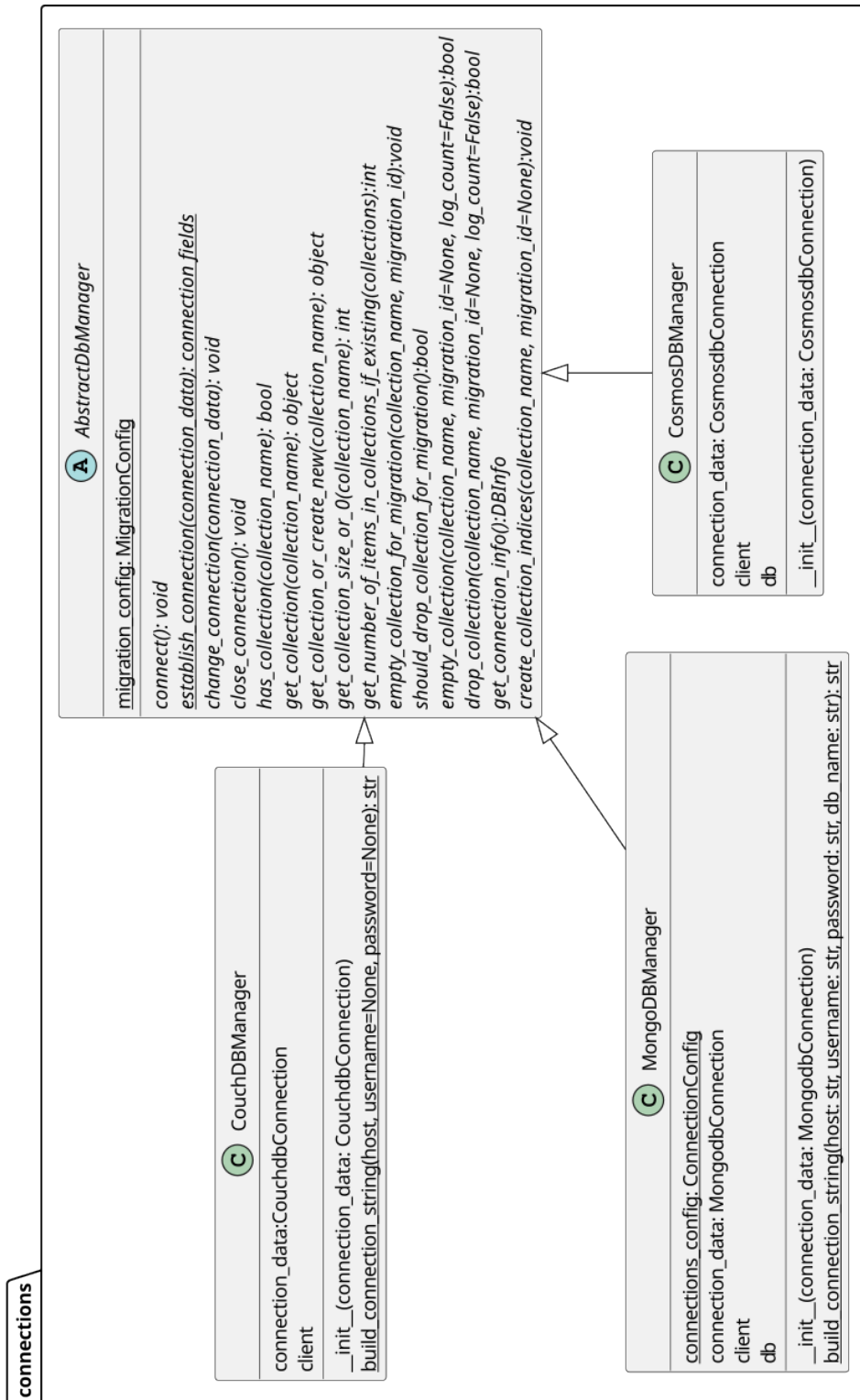


Figure 3.6.: Class diagram of connection managers classes

Class diagram for indices

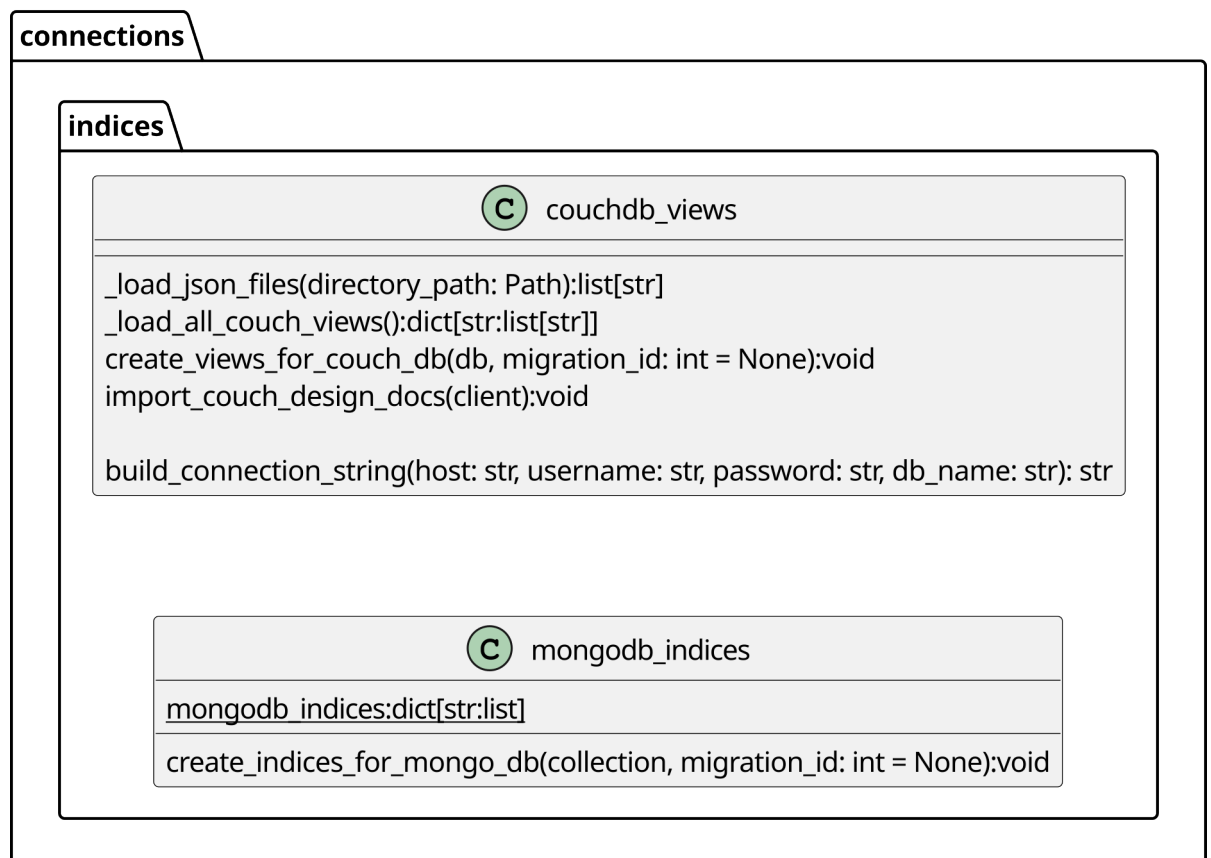


Figure 3.7.: Class diagram of indices managers

Migration

This package contains the classes MigrationManager (Figure 3.8), the abstract Migrator (Figure 3.8), and 9 subclasses of Migrator (Figure 3.9); these classes represent the managers of migration processes and contain most related logic.

The class MigrationManager has a main method called **start_migration** and supporting methods to:

- Find the appropriate connection managers based on the source and target database type of the migration.
- Find the appropriate subclass of Migrator based on the source and target database type of the migration.

The class Migrator and all its subclasses follow the **Template** design pattern. The class Migrator defines the overall migration workflow and delegates the data migration logic for individual data collections to a concrete subclass implementation. This approach improves code reusability and facilitates the integration of new database types in the future. An activity diagram of the migration process is shown in Figure 3.10.

The abstract methods of the Migrator class are:

- **get_source_type**: return the source type that can be supported by this class.
- **get_target_type**: return the target type that can be supported by this class.
- **is_parallel**: returns whether the data of a collection can be migrated in parallel.
- **migrate_collection**: perform migration of data collection from source database to target database.

The implemented methods of the Migrator class are:

- **is_applicable**: takes a Migration object as a parameter and returns true when the class is appropriate based on the source and target database types.
- **migrate**: start a migration process.
- **process_collection**: starts a migration process for a data collection.
- **preprocess_collection_migration**: performs pre-migration tasks for a collection.
- **postprocess_collection_migration**: performs post-migration tasks for a collection.
- **remove_mongodb_fields**: removes fields from a data document that was stored in MongoDB.
- **remove_couchdb_fields**: removes fields from a data document that was stored in CouchDB.
- **remove_cosmosdb_fields**: removes fields from a data document that was stored in Cosmos DB.
- **set_id_to__id**: Set id = __id, used mainly for migration into Cosmos DB.

Class diagram for Migrator and MigrationManager

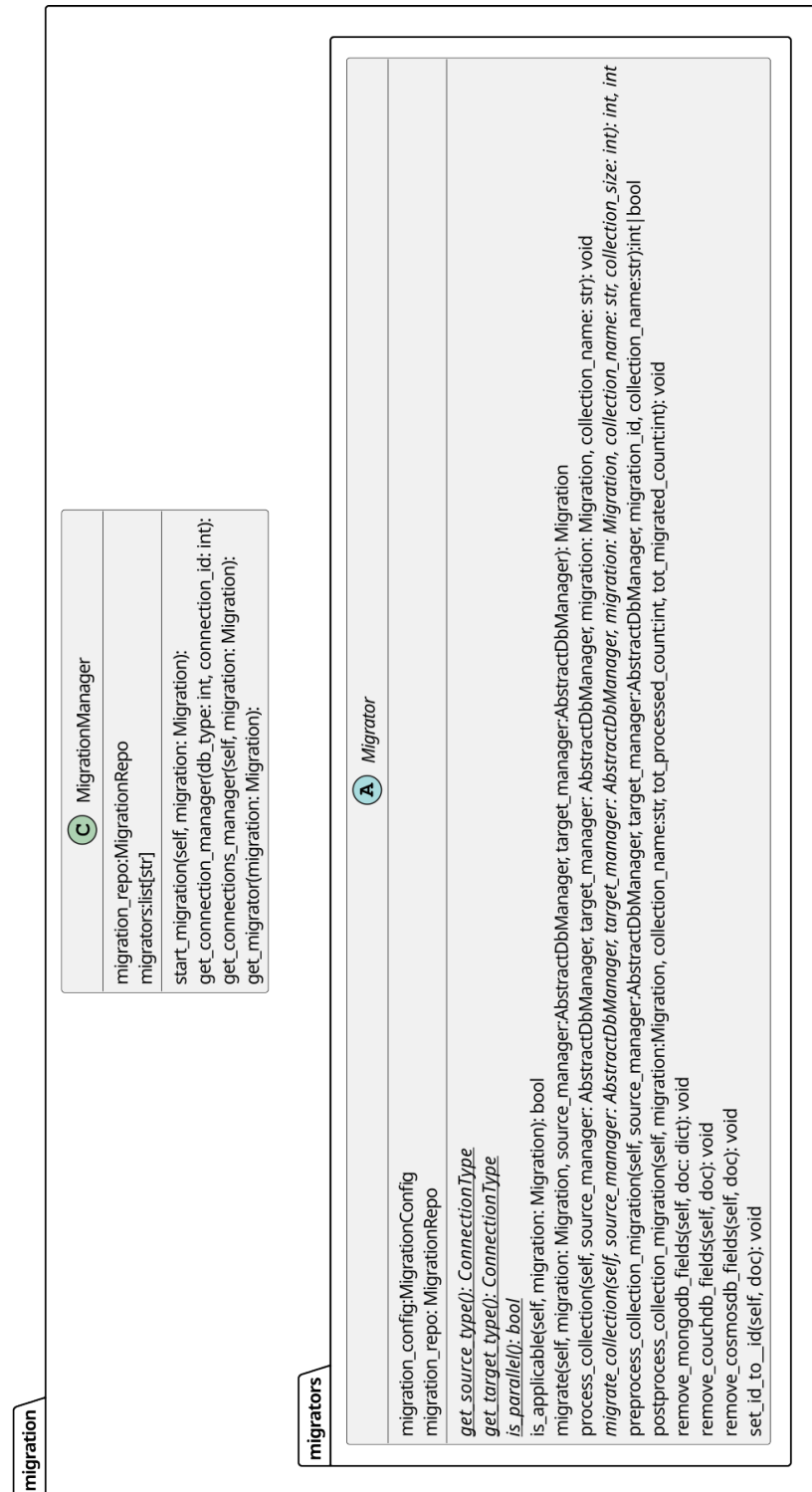


Figure 3.8.: Class diagram of MigrationManager and Migrator

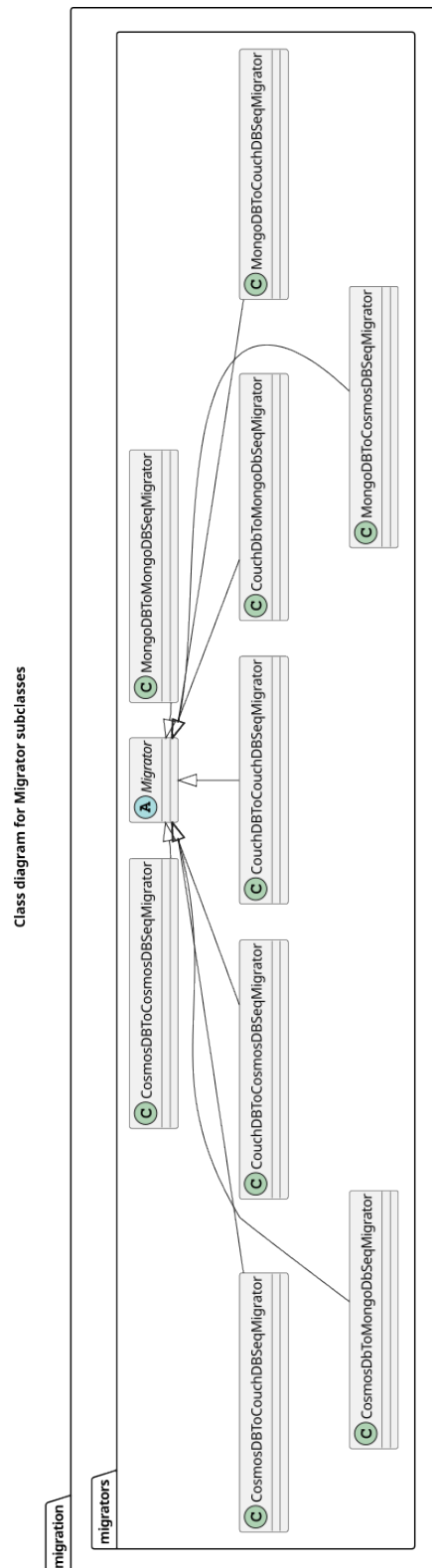


Figure 3.9.: Class diagram of Migrator subclasses

3. Approach / Design

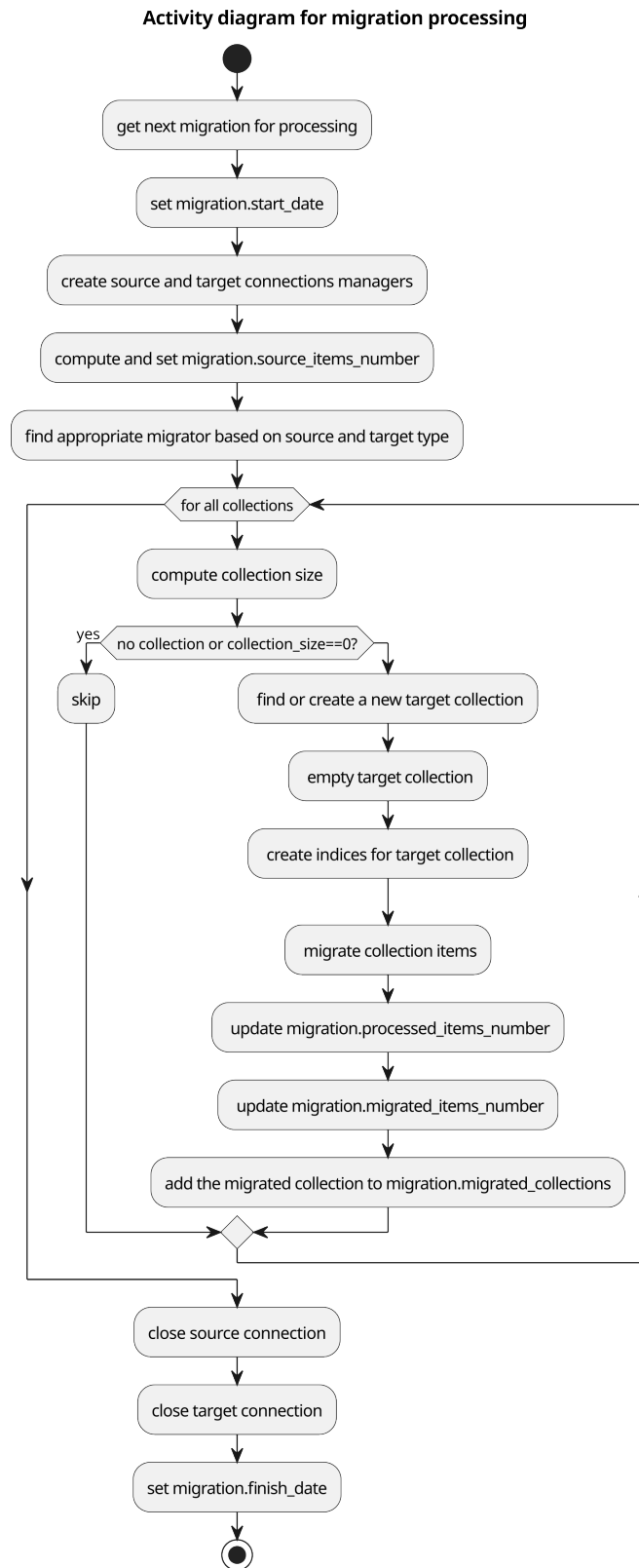


Figure 3.10.: Activity diagram of the migration process

Jobs

This package (Figure 3.11) contains the two jobs running in the backend system. A job is a method that is executed periodically at a predefined time interval. The execution interval and the number of instances can be configured separately for each job.

The Jobs are:

- **migration_processing_job**: to find scheduled migrations and process them.
- **failing_migration_detector_job**: to detect failing migrations and update their status. Failing migrations are defined as migrations that have a recorded start date but no finish date and have not been updated for a specified period of time.

Class diagram for jobs

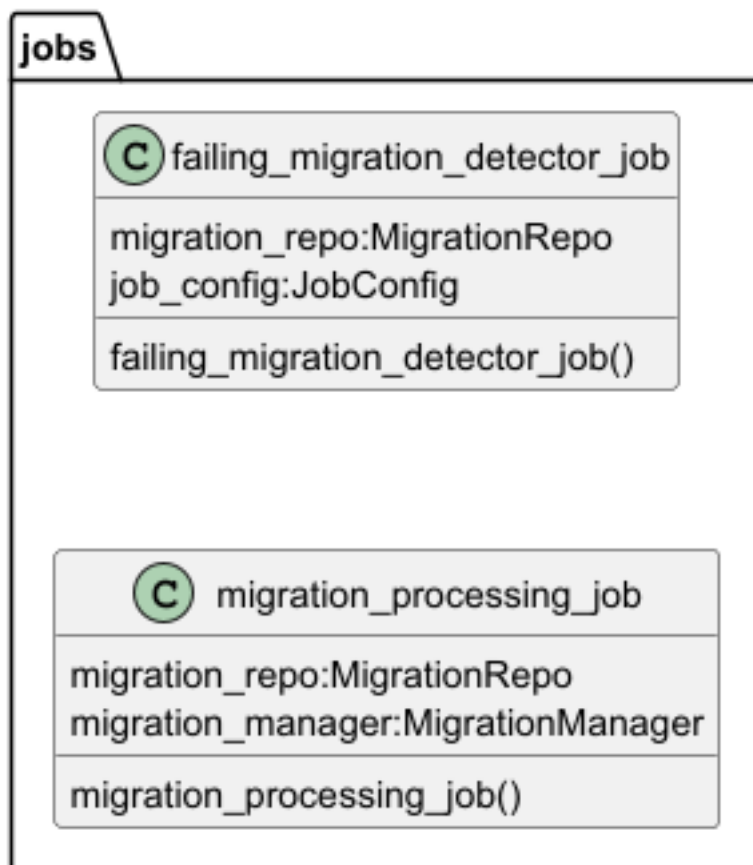


Figure 3.11.: Class diagram of jobs

Database Repositories

This package (Figure 3.12) contains the classes that act as database repositories. Each repository class encapsulates the database access logic for a specific database table. When no complex access to a table is required, i.e., only simple SELECT or UPDATE operations are performed, a repository class is unnecessary.

The repositories are:

- **MigrationRepo:** provides enhanced methods to interact with the table **migrations**, most important methods are:
 - **get_migration_for_api:** returns a list of migration objects enhanced with a synthetic status and the names of source and target database types.
 - **get_next_migration_for_processing:** selects a view to find the next scheduled migration to be processed.
 - **get_migrations_not_updated_since_m_min:** finds a running migration with no recorded progress for m minutes.
 - **increase_migration_processed_items_number:** increases the number of processed items in the migration table.
 - **increase_migration_migrated_items_number:** increases the number of migrated items in the migration table.
 - **update_migration_migrated_collections:** adds a new collection name to the list of migrated collections.
- **FactserverConnectionRepo:** provides methods to store and read entries of the table **factserver_connections**, most important methods are:
 - **get_all_connections:** returns a list of all fact server connections.
 - **get_all_valid_connections:** returns the factserver connection that is valid at the time of the call.
 - **store_connection_and_set_valid_until_for_existing_conn:** stores a new factserver connection and invalidates any valid connection by setting **valid_until** to **now()**.



Figure 3.12.: Class diagram of repository classes

3. Approach / Design

Dummy

This package (Figure 3.13) contains a class to generate and insert dummy data in the supported database types. Dummy data is intended to facilitate the demonstration of system functionality and support the evaluation process.

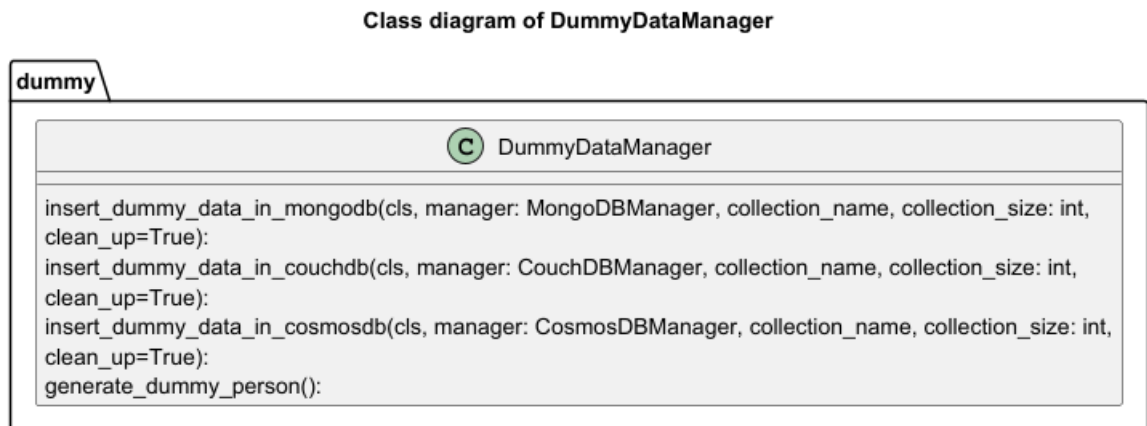


Figure 3.13.: Class diagram of DummyDataManager

Utils

This package (Figure 3.14) contains classes that provide utility methods.

The classes are:

- **annotations:** implements two annotations:
 - **log_migration_execution:** used to compute and log the total execution time of the whole migration process. This annotation annotates the method **migrator.migrate**
 - **log_collection_execution:** used to compute and log the execution time of the migration process of a single data collection. This annotation annotates the method **migrator.process_collection**
- **logging_utils:** this class provides general logging methods and methods to be used for logging events during migration processes, the methods are:
 - **log::** to log any kind of events.
 - **log_migration_event:** to log events related to a migration process.
 - **log_collection_event:** to log events related to a specific data collection of the migration process.
 - **log_completed_collection_migration:** to log the completion of the migration of a specific data collection.
- **date_utils:** provides one method to return the ISO 8601⁷ format of the date given as input parameter.

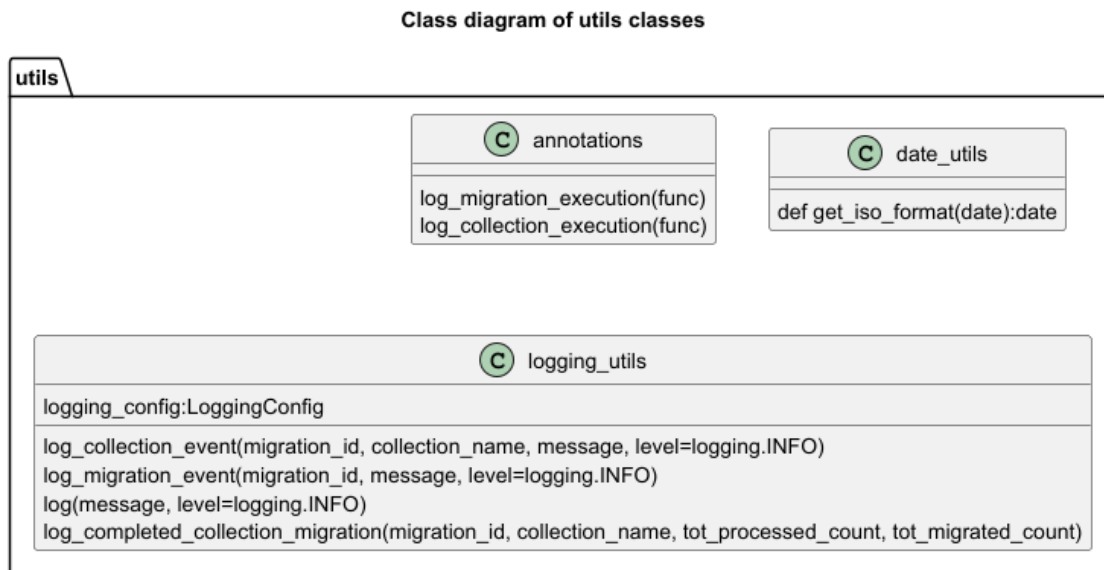


Figure 3.14.: Class diagram of utils classes

⁷ISO 8601 date format: YYYY-MM-DDThh:mm:ss

Config

This package (Figure 3.15) contains the data classes for configuration. The fields of the configuration classes are initialized based on environment variables. For all fields, a default value is provided if the corresponding environment variable is not set.

The **Config** class contains instances of all configuration classes, providing a single access point to the system's configuration.

The configuration classes are:

- **DatabaseConfig:** contains the connection information for the PostgreSQL database used by the backend system.
 - DB_HOST
 - DB_PORT
 - DB_NAME
 - DB_USER
 - DB_PASSWORD
- **ProcessingJobConfig:** contains the interval and the number of instances of the migration processing job.
 - PROCESSING_JOB_INTERVAL_IN_SEC
 - PROCESSING_JOB_INSTANCES_NUM
- **FailingMigrationsDetectorJobConfig:** contains the interval, the failing threshold, and the error message to be used.
 - FAILING_DETECTOR_JOB_INTERVAL_IN_SEC
 - FAILING_DETECTOR_ERROR_AFTER_IN_MIN
 - FAILING_DETECTOR_ERROR_MESSAGE
- **ConnectionsConfig:** contains configuration related to connection establishment.
 - MONGODB_SERVER_SELECTION_TIMEOUT_MS
- **LoggingConfig:** contains configuration related to logging.
 - LOG_LEVEL
 - COSMOS_LOG_LEVEL
- **MigrationConfig:** contains configuration related to the migration processes.
 - MONGODB_BATCH_INSERT_SIZE
 - MONGODB_CLEAN_COLLECTION_BY_DROP
 - COUCHDB_CLEAN_COLLECTION_BY_DROP
 - COSMOSDB_CLEAN_COLLECTION_BY_DROP
 - MONGODB_FIELDS_TO_REMOVE
 - COUCHDB_FIELDS_TO_REMOVE
 - COSMOSDB_FIELDS_TO_REMOVE
 - COSMOSDB_ID_FROM_ID
 - COSMOSDB_PARTITION_KEY
 - UPDATE_NUMBER_PROCESSED_ON
 - PARALLEL_PROCESSING_THREADS_NUM
 - CREATE_KNOWN_INDICES

3.2. System Design & Taken Design Decisions

Field	Description	Environment Variable	Default
host	Database host	DB_HOST	localhost
port	Database port	DB_PORT	5432
db_name	Database name	DB_NAME	DBManagerConfig
user	Database user	DB_USER	postgres
password	Database password	DB_PASSWORD	postgres
interval_in_sec	Interval in seconds	PROCESSING_JOB_INTERVAL_IN_SEC	10
max_instances	Number of instances	PROCESSING_JOB_INSTANCES_NUM	1
interval_in_sec	Interval in seconds	FAILING_DETECTOR_JOB_INTERVAL_IN_SEC	60
error_after_in_minutes	Failing threshold in min	FAILING_DETECTOR_ERROR_AFTER_IN_MIN	10
error_message	Error message	FAILING_DETECTOR_ERROR_MESSAGE	Migration set to error after being idle for more than minutes
mongodb_server_selection_timeout_ms	Mongo timeout in MS	MONGODB_SERVER_SELECTION_TIMEOUT_MS	2000
log_level	Log level	LOG_LEVEL	INFO
azure_cosmos_log_level	Cosmos log level	COSMOS_LOG_LEVEL	WARNING
mongodb_batch_insert_size	Batch insert size for mongodb	MONGODB_BATCH_INSERT_SIZE	localhost
mongodb_clean_collection_by_drop	Clean MongoDB collection by dropping	MONGODB_CLEAN_COLLECTION_BY_DROP	True
couchdb_clean_collection_by_drop	Clean CouchDB collection by dropping	COUCHDB_CLEAN_COLLECTION_BY_DROP	True
cosmosdb_clean_collection_by_drop	Clean Cosmos DB collection by dropping	COSMOSDB_CLEAN_COLLECTION_BY_DROP	True
mongodb_fields_to_remove	Fields to be removed from MongoDB source	MONGODB_FIELDS_TO_REMOVE	id, rid, self, etag, attachments, ts
couchdb_fields_to_remove	Fields to be removed from CouchDB source	COUCHDB_FIELDS_TO_REMOVE	True
cosmosdb_fields_to_remove	Fields to be removed from cosmos source	COSMOSDB_FIELDS_TO_REMOVE	True
cosmosdb_id_from_id	Set cosmos id to _id	COSMOSDB_ID_FROM_ID	True
cosmosdb_partition_key	Cosmos partition key	COSMOSDB_PARTITION_KEY	id
update_number_processed_on	Progress update interval, overridden by MONGODB_BATCH_INSERT_SIZE for MongoDB targets	UPDATE_NUMBER_PROCESSED_ON	1000
parallel_processing_threads_num	Number of threads for parallel processing	PARALLEL_PROCESSING_THREADS_NUM	os.cpu_count()
create_known_indices	Create configured indices for the target database	CREATE_KNOWN_INDICES	True

Table 3.1.: Environment variables for DBManager backend

Class diagram of config

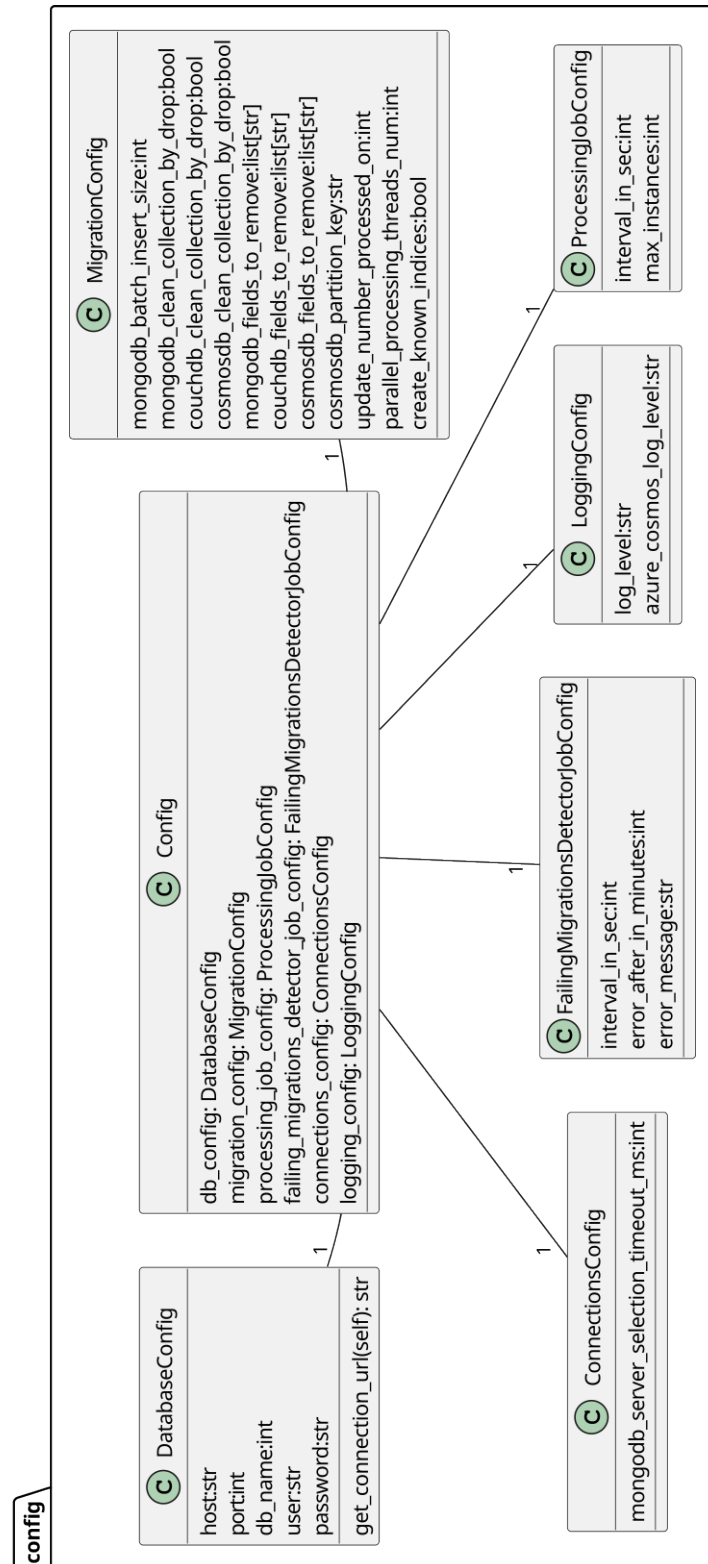


Figure 3.15.: Class diagram of configuration classes

Views, App, and Main

The class `views` has a list of the views DDL⁸, in addition, it provides methods to create and drop the list of stored views. The class `app` contains the definition of the Flask application. The class `main` is the start point of the application, it configures the jobs, creates database views, inserts static data into the database, and starts the Flask application. Figure 3.16 shows a class diagram of the classes **views**, **app**, and **main**.

Class diagram for views, app and main

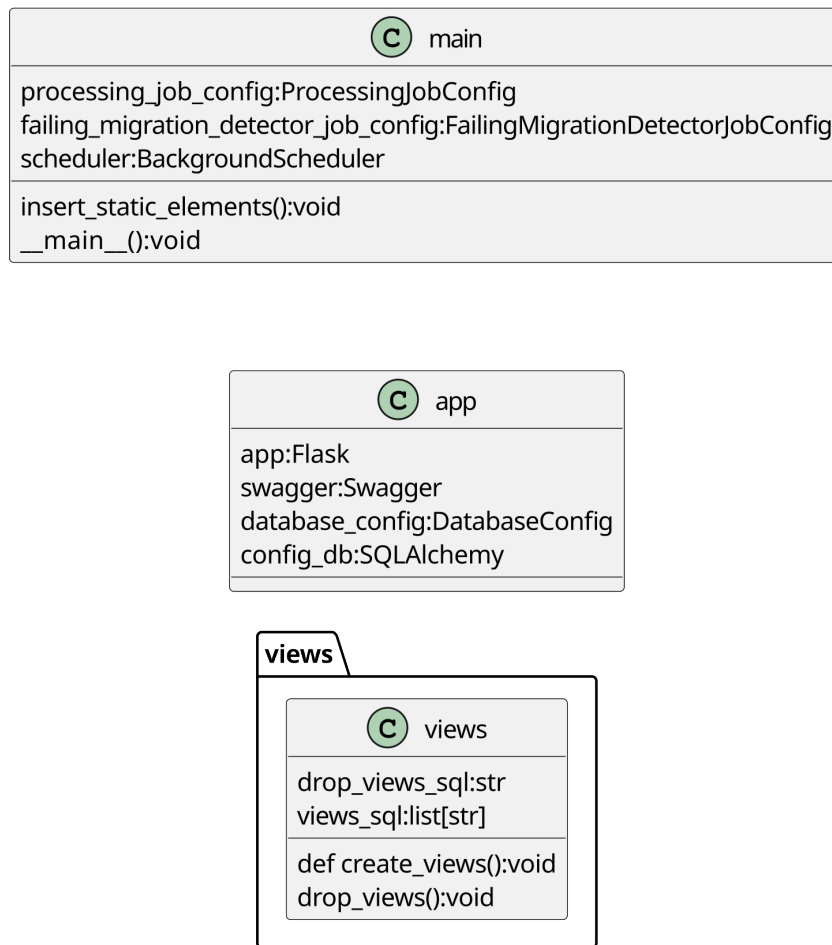


Figure 3.16.: Class diagram for views, app, and main

⁸DDL: Data Definition Language

3.2.2. DBManager Frontend

The frontend application is implemented using Angular 20.3.9, and has the following main building parts:

services

Services define the communication logic with the backend server. There are five services, three of which are for managing database connections, one is for the migrations, and the last one is for the factserver connections. These services provide methods to call the endpoints of the backend server. Figure 3.18 shows a class diagram of the frontend services.

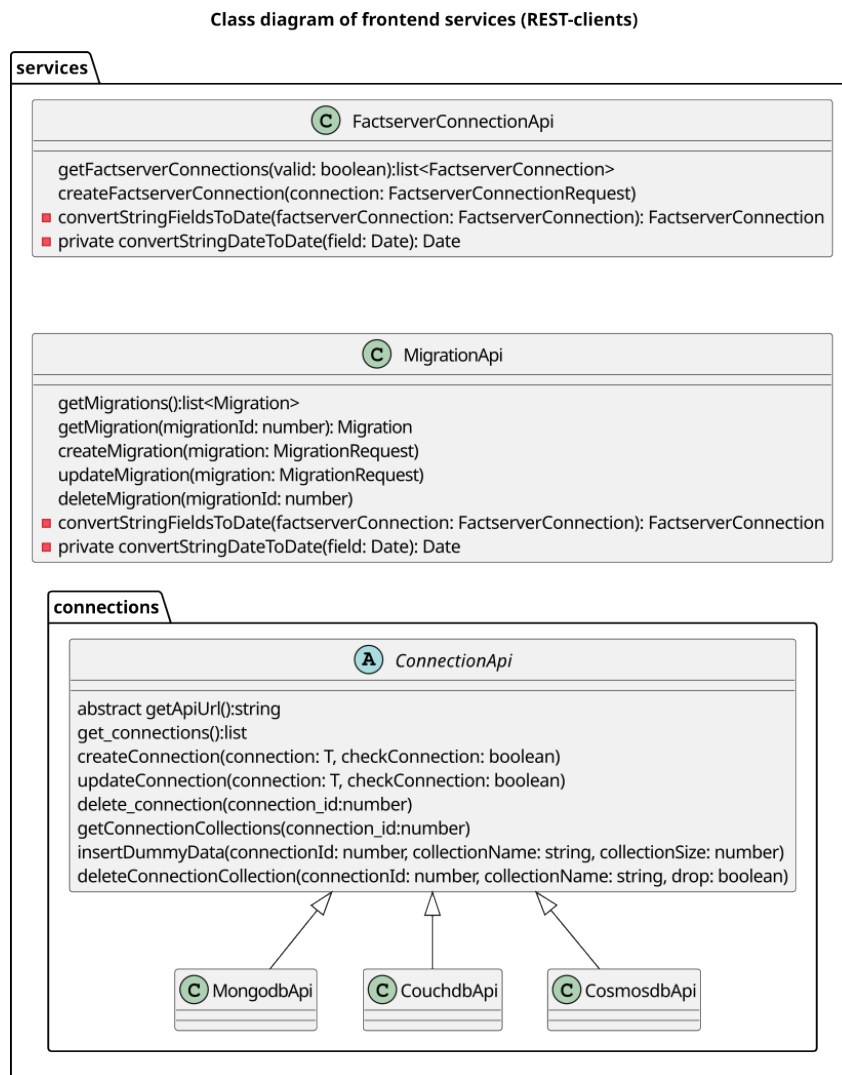


Figure 3.18.: Class diagram of the frontend services

3. Approach / Design

Models

Models (Figure 3.19) include the data classes used for communication with the backend.

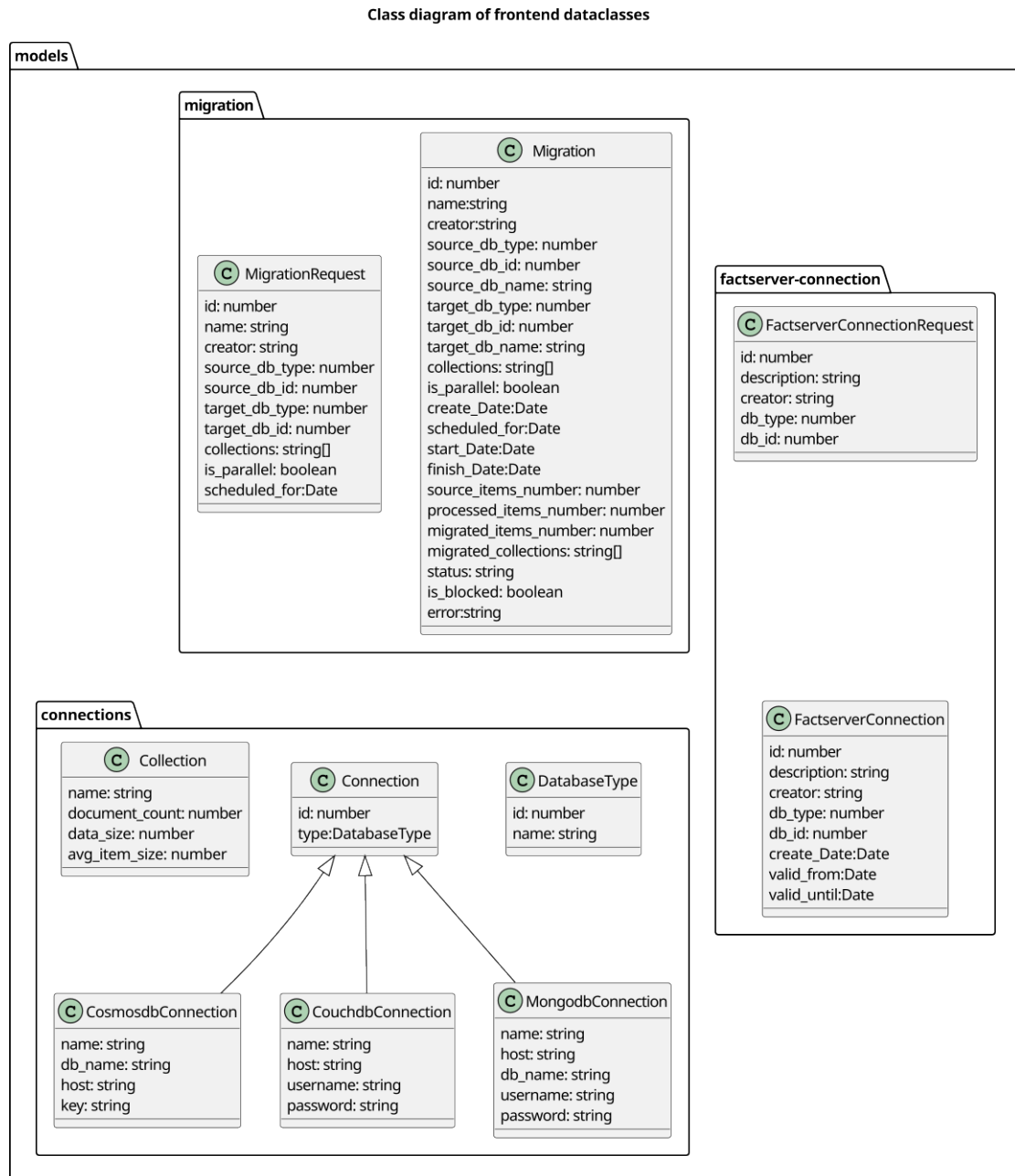


Figure 3.19.: Class diagram of the frontend models

Components

Components (Figure 3.20) define the visual components of the system. There are three main visual components:

- Connections: to manage connections.
- Migrations: to manage migrations.
- Factserver Connections: to manage factserver connections.

The Connections component has three sub-components, each for a specific database type. All three sub-components are using the Utils components, because they need to provide the same UI support.

The Utils components are:

- ConnectionUpsert: supports modifying or adding a new connection.
- ConnectionsTable: supports showing the table of all connections, and the actions that can be done for them.

This design is based on the open-closed principle, allowing new components to be integrated into the system and use the existing functionalities with minimum effort.

Class diagram of frontend visual components

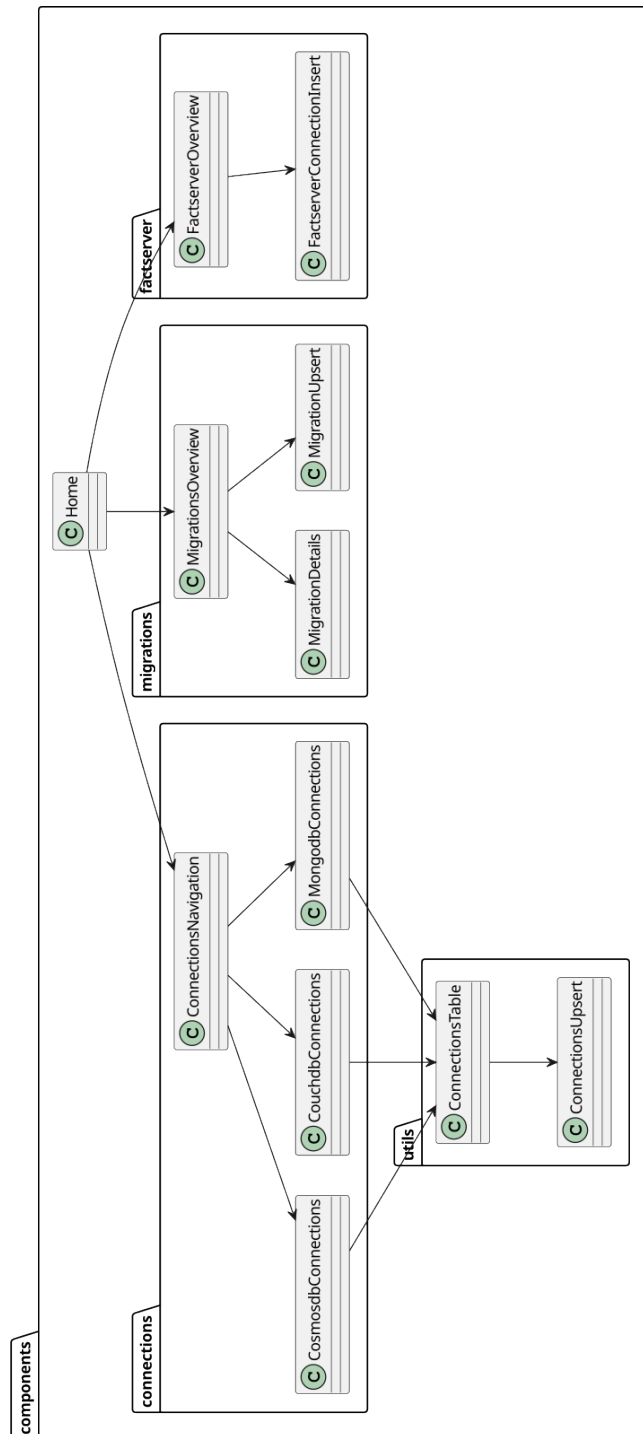


Figure 3.20.: Class diagram of the frontend visual components

3.2.3. DBManager Database

The data of the system is stored in a PostgreSQL 18.1 database.

The database has the following tables:

- `database_types`: stores the supported database implementations (static).
- `mongodb_connections`: stores the data to connect to a MongoDB database.
- `couchdb_connections`: stores the data to connect to a CouchDB database.
- `cosmosdb_connections`: stores the data to connect to a Cosmos DB database.
- `migrations`: stores the data of a migration process.
- `factserver_connections`: stores information about connections used by the factserver.

Figure 3.21 and Figure 3.22 provide a visual representation of the database structure.

3. Approach / Design

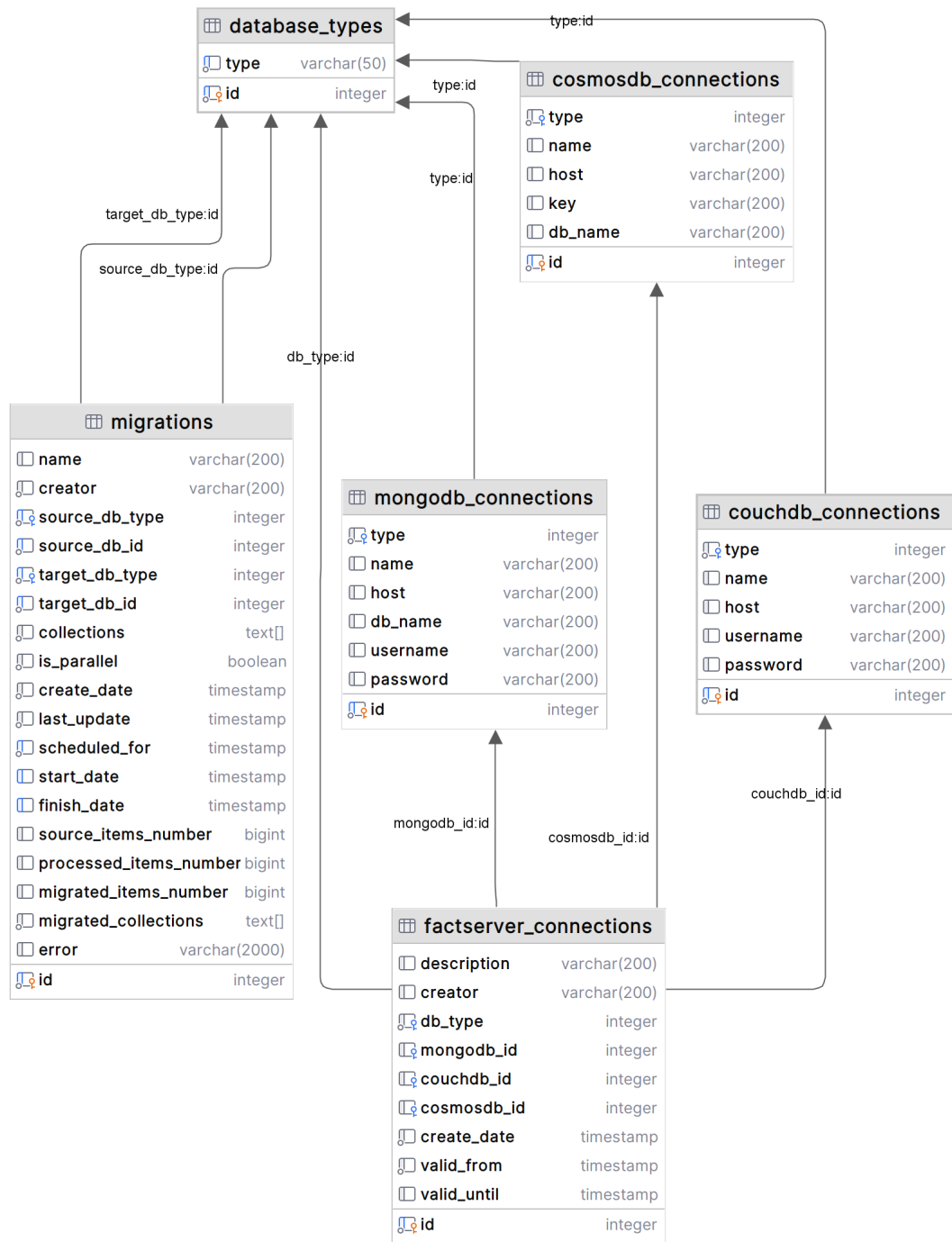


Figure 3.21.: Diagram of the database tables

3.2. System Design & Taken Design Decisions

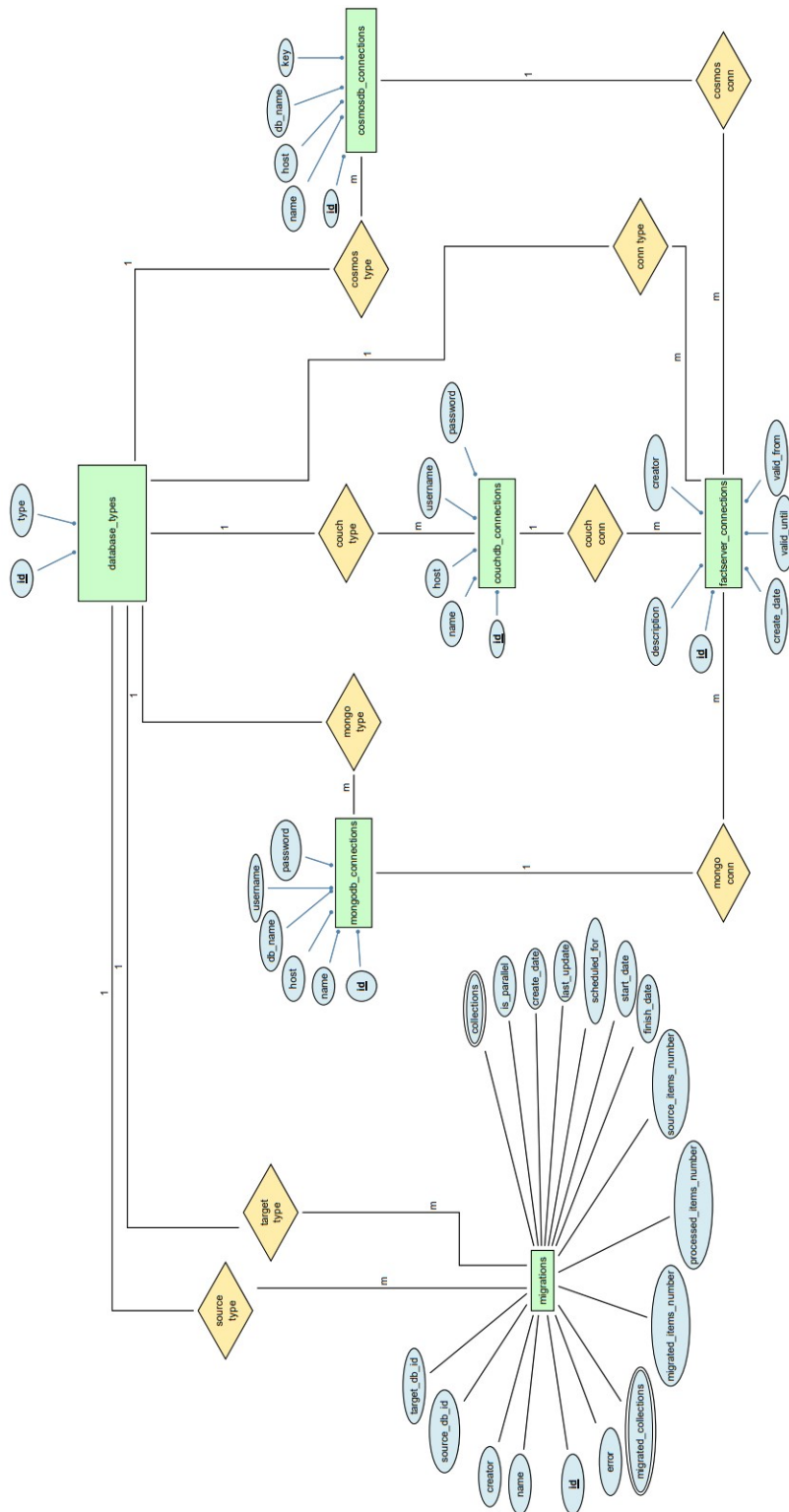


Figure 3.22.: ER-Diagram of the database

3. Approach / Design

3.2.4. FactServer Changes

The FactServer backend is implemented in Python. The original implementation uses CouchDB as a storage solution. One goal of this thesis is to enhance FactServer so that it can use multiple database implementations, specifically MongoDB, Cosmos DB, and CouchDB. The enhancement aims to provide flexible and extendable support for the databases.

The database logic was encapsulated in the abstract class **AbstractDBHandler** and its subclass **CouchDBHandler**. AbstractDBHandler provides the following methods:

- `connect`: to establish the connection to one specific data collection.
- `get_record`: to return one item by its id.
- `save_record`: to store or update an item.
- `delete_record`: to delete an item.
- `execute_query`: to execute a database select query and returns the result.

A factory class called **CouchDBFactory** was responsible for reading the connection data and creating an instance of **CouchDBHandler**.

After a detailed analysis, we introduced the following enhancements:

MongoDBHandler: A new subclass of the class AbstractDBHandler is implemented to support MongoDB. In addition, a factory class **MongoDBFactory** is implemented to support creating instances of MongoDBHandler.

CosmosDBHandler: A new subclass of the class AbstractDBHandler is implemented to support Cosmos DB. In addition, a factory class **CosmosDBFactory** is implemented to support creating instances of CosmosDBHandler.

ProxyDBHandler: A new subclass of the class AbstractDBHandler is implemented to act as a proxy handler to support a configurable database selection. A PostgreSQL database stores the information about which database implementation and the credentials for the database connection to be used. ProxyDBHandler is able to periodically check the configuration database to ensure it uses the configured connection. As FactServer, in its current implementation, is built as a set of serverless functions, the feature of periodic checks of the configuration database is made optional (can be disabled), because for each request served by the FactServer, the database is requested to select the configured connection, and a new connection is established. This feature can be useful in case the application is implemented as a long-running server, instead of a serverless implementation.

ConfigDB: The logic to connect to a configuration database (PostgreSQL), and the related model classes are implemented. This database contains the connection data that the FactServer must use.

Figure 3.23 shows the class diagram of the changes applied to FactServer. Due to space limitations, the methods of the DBHandler subclasses are shown in Figure 3.24, and the data classes are shown in Figure 3.25.

3.2. System Design & Taken Design Decisions

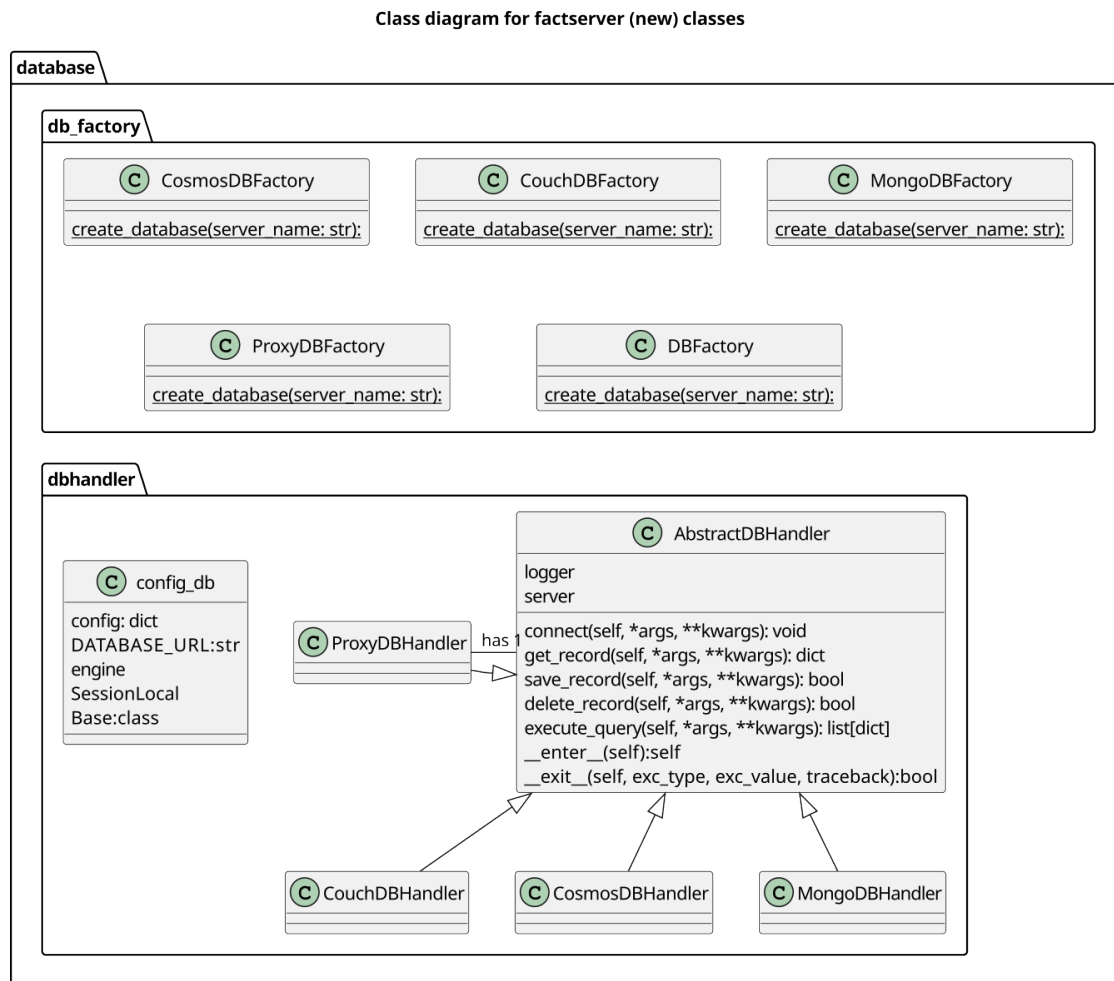


Figure 3.23.: Class diagram for FactServer enhancements

3. Approach / Design

Class diagram for DBHandler implementations

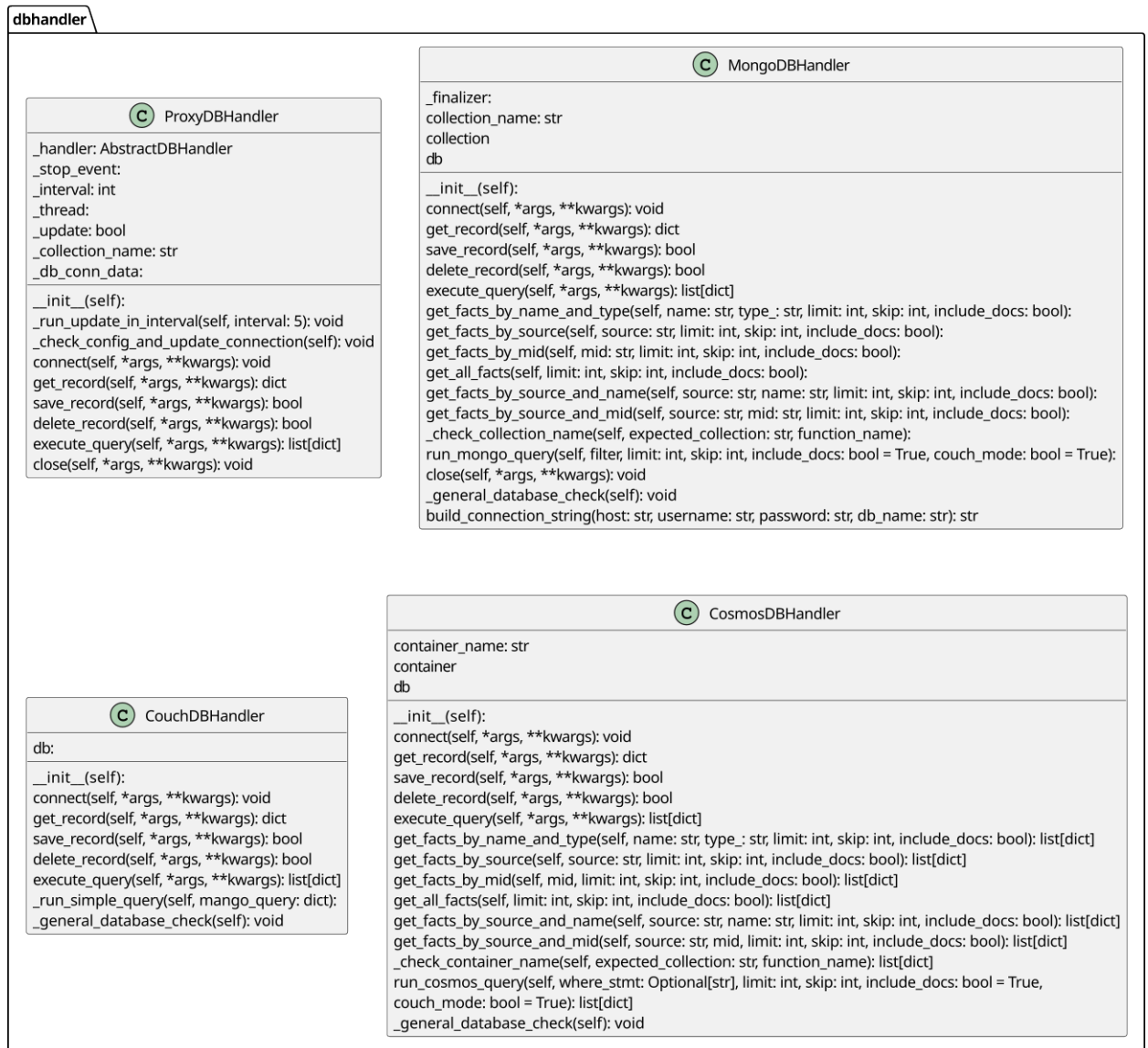


Figure 3.24.: Class diagram for DBHandler implementations

3.2. System Design & Taken Design Decisions

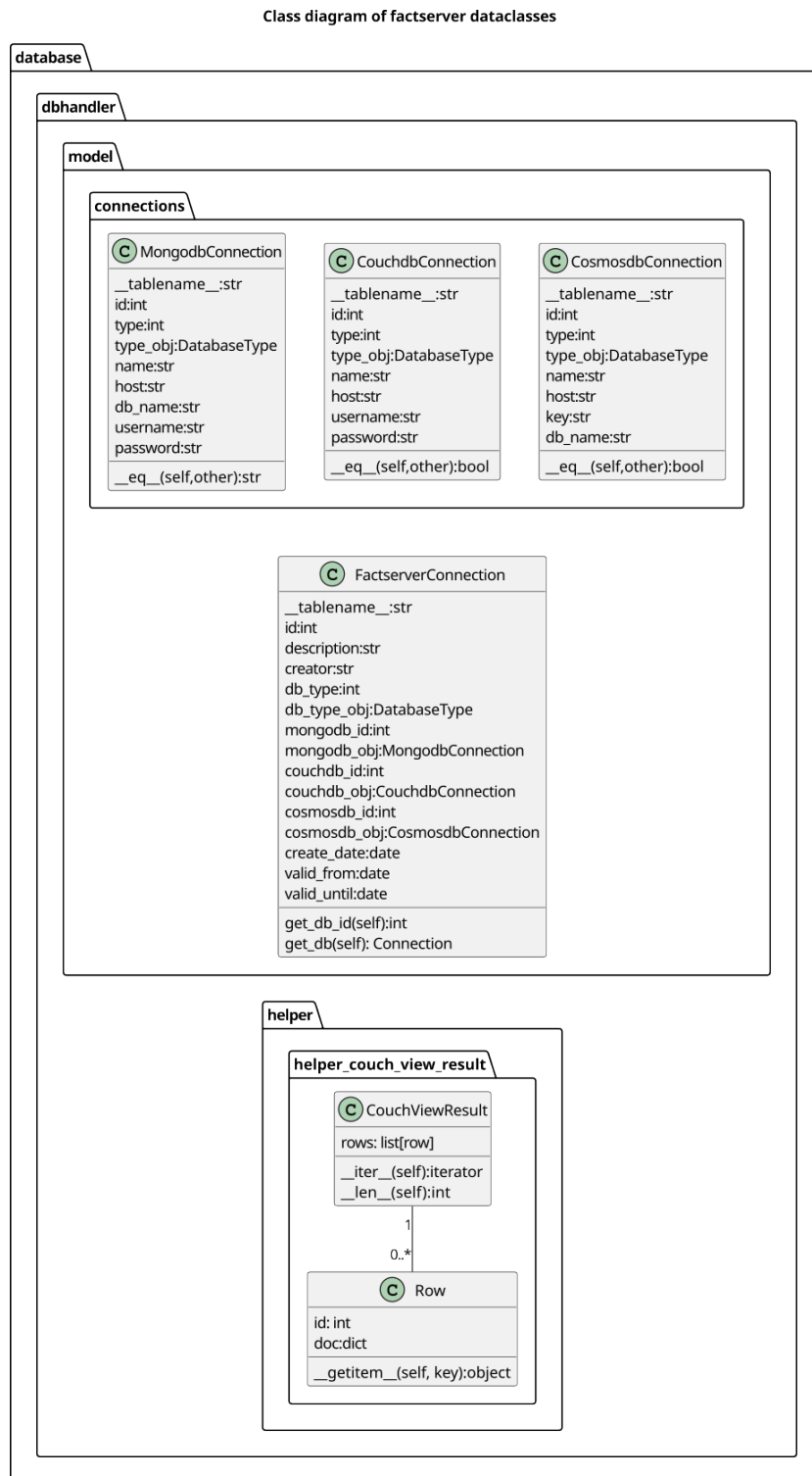


Figure 3.25.: Class diagram for FactServer data classes

4. Implementation

This chapter introduces and discusses the implementation details of each part of the system.

This includes:

- The DBManager Backend application.
- The DBManager Frontend application.
- The changes done to the Factserver.
- The Deployment using Docker.

The source code is available in the **GitLab** repository of this project. Access is restricted to the members of the project.

4.1. DBManager Backend

Programming language Python 3.12 was used to implement the Backend server. All classes and methods of the code are documented using Google Style Docstrings, which makes the code very easy to understand. The application has the following main components:

- Main
- App
- Blueprints (Rest Controllers)
- Data Model
- Database Repositories
- Views
- Connection Managers
- Migration Managers
- Jobs
- Logging
- Config
- Unit Test

4.1.1. Main

The listings 4.1,4.2, 4.2, and 4.4 present the contents of the file main.py, which is the entry point of the application. It registers the blueprints, configures the jobs, creates database views, inserts static data into the database, and starts the Flask application.

4. Implementation

```
1 # Register blueprints (Rest Controllers)
2 app.register_blueprint(cosmosdb_controller.cosmosdb_bp,
    url_prefix='/connections/cosmosdb')
3 app.register_blueprint(couchdb_controller.couchdb_bp, url_prefix
    ='/connections/couchdb')
4 app.register_blueprint(mongodb_controller.mongodb_bp, url_prefix
    ='/connections/mongodb')
```

Listing 4.1: Registering of blueprints in main.py

```
1 # Called when the application starts:
2 # Steps: Drop views, create tables if not existing, create views
    , insert static entries in the tables.
3 # Updates to model classes are not considered after the first
    creation of the tables.
4 with app.app_context():
5     drop_views()
6     # config_db.drop_all() # uncomment only when dropping the
    tables is needed!!!!
7     config_db.create_all()
8     create_views()
9     insert_static_elements()
```

Listing 4.2: Database setup in main.py

```
1 # define cron jobs, interval and instances number are defined in
    the config class: ProcessingJobConfig
2 # 1: migration_processing_job : call function
    migration_processing_job
3 # 2: failing_migrations_detector_job : call function
    failing_migration_detector_job
4 config = load_config()
5 processing_job_config = config.processing_job_config
6 failing_migration_detector_job_config = config.
    failing_migrations_detector_job_config
7 scheduler = BackgroundScheduler()
8 scheduler.add_job(id='migration_processing_job', func=
    migration_processing_job, trigger="interval",
9                     replace_existing=True,
10                    seconds=processing_job_config.interval_in_sec,
    max_instances=processing_job_config.max_instances)
11 scheduler.add_job(id='failing_migrations_detector_job', func=
    failing_migration_detector_job, trigger="interval",
12                    replace_existing=True,
13                    seconds=failing_migration_detector_job_config.
    interval_in_sec, max_instances=1)
```

Listing 4.3: Cron jobs setup in main.py

```

1 if (__name__ == "__main__"):
2     scheduler.start()
3     atexit.register(lambda: scheduler.shutdown())
4     app.run(host='0.0.0.0', port=8080)

```

Listing 4.4: The start of the jobs scheduler and the Flask app in main.py

4.1.2. App

The file app.py, shown in Listing 4.5, contains the creation and setup of the Flask application, including CORS configuration, setup of Swagger, and the setup of the connection to the database.

```

1 app = Flask(__name__)
2
3 swagger = Swagger(app)
4
5 database_config = load_config().db_config
6 app.config['SQLALCHEMY_DATABASE_URI'] = database_config.
    get_connection_url()
7 app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
8
9 config_db = SQLAlchemy(app)
10
11 CORS(app)

```

Listing 4.5: Setup in app.py

The database connection credentials are read from the config object. The attribute **SQLALCHEMY_TRACK_MODIFICATIONS** is set to False, because the application does not need these signals, and to reduce memory overhead.

4.1.3. Blueprints (Rest Controllers)

The blueprints represent the endpoints of the application. Listing 4.6 shows the initialization of the blueprint and the method to handle a GET request. The annotation `@swag_from` is used to define the Swagger file for this method. The annotation `@cosmosdb_bp.route("", methods=['GET'])` defines the REST methods the Python method is used for and the path for this resource; in our case, the path is empty. The base path (url_prefix) for each blueprint is defined in the file main.py, where the blueprints are registered. For cosmosdb_bp, the path is `/connections/cosmosdb`

4. Implementation

```
1 cosmosdb_bp = Blueprint('cosmosdb', __name__)
2
3 @swag_from('../swagger/connections/cosmosdb_controller/
4   get_connections.yml')
5 @cosmosdb_bp.route('/', methods=['GET'])
6 def get_connections():
7     """
8     Returns a list of CosmosDB connections.
9     Status Codes:
10         - 200: Successfully processed the request.
11         - 500: Internal server error during database operations.
12     Returns:
13         A list of connections
14     """
15     connections = CosmosdbConnection.query.order_by(
16         CosmosdbConnection.id.desc()).all()
17     connections = [e.to_dict() for e in connections]
18     return make_response(connections, 200)
```

Listing 4.6: Setup of a blueprint for cosmosdb connections, and a GET method

In the Swagger file shown in Listing 4.7, the description, the request parameters, the request body, and the responses are defined.

```
1 An endpoint for getting a Cosmos DB connection by ID.
2 ---
3 tags:
4   - CosmosDB Connections
5 parameters:
6   - in: path
7     name: connection_id
8     required: true
9 responses:
10    200:
11      description: Data of CosmosDB connection.
12      examples:
13        application/json: {"id": 123456,
14                           "type": {"id": 3, "name": "Cosmos DB"},
15                           "name": "My cosmosDB instance",
16                           "host": "localhost:27017",
17                           "db_name": "fc",
18                           "key": "aads51asd65dd6safd51fd"}
19    404:
20      description: Connection could not be found
21      examples:
22        application/json: { "message": "404 Not Found: No CosmosDB
23                           connection found with id: 99" }
```

Listing 4.7: The swagger file for the method get_connection in cosmosdb_bp

Listing 4.8 shows a POST method for the creation of a new connection. The request parameter `check_connection` is used to check if the connection can be established before saving the connection data.

```

1 @swag_from('../..//swagger/connections/cosmosdb_controller/
  add_connection.yml')
2 @cosmosdb_bp.route('', methods=['POST'])
3 def add_connection():
4     """
5     Save a new CosmosDB connection.
6
7     Request Body:
8     - Connection Fields, see swagger.
9
10    Request Parameters:
11    - check_connection (boolean): Whether the connection
12      establishment must be checked before saving or not.
13
14    Status Codes:
15    - 200: Successfully processed the request.
16    - 422: Connection establishment is failing.
17    - 500: Internal server error during database operations.
18
19    Returns:
20    (Response): Request response with status code
21    """
22    data = request.get_json()
23    check_connection = request.args.get('check_connection', '
false').lower() == 'true'
24    success_message = "New CosmosDB connection was added
successfully"
25    connection = CosmosdbConnection()
26    return create_or_update_connection(connection, data,
check_connection, success_message)

```

Listing 4.8: POST method to create a new cosmosdb connecetion

The method `create_or_update_connection` (Listing 4.9) maps the request data into the ORM class, stores the connection data in the database, and returns an HTTP response.

4. Implementation

```
1 def create_or_update_connection(connection: CosmosdbConnection,
2   data, check_connection, success_message: str):
3   """
4   Read the data from request body and store it in the database
5
6   Args:
7       connection (CosmosdbConnection) : Instance of the
8       CosmosDB connection
9       data (dict) : The request body
10      check_connection (boolean) : Whether connection
11      establishment must be tested before saving
12      success_message (string) : Message to return when
13      connection storage is done
14
15      Status Codes:
16          - 200: Successfully processed the request.
17          - 404: Connection does not exist.
18          - 422: Connection establishment is failing.
19          - 500: Internal server error during database operations.
20
21      Return:
22          (Response): Request response with status code
23      """
24      connection.name = data['name']
25      connection.host = data['host']
26      connection.db_name = data['db_name']
27      connection.key = data['key']
28
29      if check_connection:
30          is_successful, _, _, error = CosmosDBManager.
31          establish_connection(connection)
32          if is_successful:
33              config_db.session.merge(connection)
34              config_db.session.commit()
35              return make_response(jsonify({'message':
36          success_message}), 200)
37          return make_response(jsonify({'message': error}), 422)
38
39      config_db.session.merge(connection)
40      config_db.session.commit()
41      return make_response(jsonify({'message': success_message}),
42          200)
```

Listing 4.9: The method create_or_update_connection

In each blueprint, an exception handler method is defined (see Listing 4.10). In this application, the errorhandler methods catch the exception, log the traceback, and return an HTTP response code based on the exception type.

```

1 @cosmosdb_bp.errorhandler(Exception)
2 def handle_error(error):
3     """
4     Handle Exceptions
5     Args:
6         error (Exception): The instance of the Exception
7     Status Codes:
8         - 404: For NotFound
9         - 500: For any other Exception
10    """
11    traceback.print_exc()
12    if isinstance(error, NotFound):
13        return make_response(jsonify({'message': str(error)}),
14                               404)
15    elif isinstance(error, IntegrityError):
16        return make_response(jsonify({'message': str(error.orig.pgerror.strip())}), 500)
17    return make_response(jsonify({'message': str(error)}), 500)

```

Listing 4.10: The method @errorhandler for error handling in cosmosdb_bp

4.1.4. Data Model

The data classes are mainly the ORM classes that are used to sync the data between the application code and the database.

Each ORM class represents a table in the database. The application will, if a table does not exist, create the table in the database based on the definition of the ORM class. Modifications to the structure of a table can not be synchronized by the code once the table has been created. The class must inherit from the class **Model** that is inside the database instance. The table name is defined in the field `__table_name__`. For each field, the name, the type, and the restrictions are defined. Listings 4.11 and 4.12 present the class for the **database_types** table and the method used to insert static data into it

```

1 class DatabaseType(config_db.Model):
2     """
3     ORM object for the table database_types (MongoDB, CouchDB,
4     CosmosDB).
5     This table contains static rows, the rows will be inserted
6     in the method insert_static_types.
7     """
8     __tablename__ = 'database_types'
9     id = config_db.Column(config_db.Integer, primary_key=True)
10    type = config_db.Column(config_db.String(50), unique=True,
11                             nullable=False)

```

Listing 4.11: The class DatabaseType representing the table database_types with two fields (id and type)

4. Implementation

```
1 @staticmethod
2 def insert_static_types():
3     """
4     Insert database types (MongoDB, CouchDB, CosmosDB) into the
5     database and commit.
6     """
7     mongodb = DatabaseType(id=1, type='MongoDB')
8     config_db.session.merge(mongodb)
9     couchdb = DatabaseType(id=2, type='CouchDB')
10    config_db.session.merge(couchdb)
11    cosmosdb = DatabaseType(id=3, type='Cosmos DB')
12    config_db.session.merge(cosmosdb)
13    config_db.session.commit()
```

Listing 4.12: The method `insert_static_types` in class `DatabaseType` is used to insert static rows during the start of the application

Listing 4.13 shows the ORM class of the table for MongoDB connections. A sequence is created for the generation of IDs. The table has a foreign key to the table `database_types` with a default value of 1.

```
1 class MongoClientConnection(config_db.Model):
2     """
3     ORM object for the table mongodb_connections.
4     This table contains MongoDB connection credentials: like
5     host, password,...
6     """
7     __tablename__ = 'mongodb_connections'
8     id_seq = Sequence('mongodb_conn_seq')
9     id = config_db.Column(config_db.Integer, id_seq,
10    server_default=id_seq.next_value(), primary_key=True)
11    type = config_db.Column(config_db.Integer, config_db.
12    ForeignKey('database_types.id'), nullable=False,
13    server_default='1')
14    type_obj = config_db.relationship('DatabaseType',
15    foreign_keys=[type])
16    name = config_db.Column(config_db.String(200), nullable=True)
17    host = config_db.Column(config_db.String(200), nullable=True)
18    db_name = config_db.Column(config_db.String(200), nullable=
19    True)
20    username = config_db.Column(config_db.String(200), nullable=
21    True)
22    password = config_db.Column(config_db.String(200), nullable=
23    True)
```

Listing 4.13: The class `MongodbConnection` representing the table `mongodb_connections`

The method `to_dict`, shown in Listing 4.14, provides a representation of the object that can be used for communication with other systems. It is mainly used to convert the object for HTTP responses.

```

1 def to_dict(self):
2     """
3     Build and return dict representation of MongoDB connection.
4     Returns:
5         (dict): Dictionary representation of MongoDB connection.
6     """
7     return {
8         'id': self.id,
9         'type': {'id': self.type_obj.id, 'name': self.type_obj.
10 type},
11         'name': self.name,
12         'host': self.host,
13         'db_name': self.db_name,
14         'username': self.username,
15         'password': self.password
16     }

```

Listing 4.14: The method `to_dict` used to return a json representation of the object

4.1.5. Database Repositories

In Flask-SQLAlchemy, simple CRUD operations on a table can be done directly by the methods of the ORM class. For complex queries, it is better to have a separate class where the queries are defined, supporting readability and reusability of code.

The class **MigrationRepo** provides methods to read and update data related to the table **migrations**. Listing 4.15 shows the method `get_next_migration_for_processing` used to return the next Migration object for the processing job.

4. Implementation

```
1 def get_next_migration_for_processing(self) -> Migration:
2     """
3     Select the view ‘‘v_migrations_for_processing’’ and return
4     the first item.
5     The view v_migrations_for_processing selects migration that:
6         - has not been started yet
7         - scheduled_for < now()
8         - not blocked by any other migration (id not in
9         v_blocked_migrations)
10        - source is not target of running m having at least
11        one overlapping collection.
12        - target is not source or a target of running m
13        having at least one overlapping collection.
14    Returns:
15        (Migration): migration object when found, None
16        otherwise.
17    """
18    select_sql = text('''
19    select * from v_migrations_for_processing limit 1
20    ''').columns(*Migration.__table__.columns)
21    migration_sql = select(Migration).from_statement(select_sql)
22    return self.session.execute(migration_sql).scalars().first()
```

Listing 4.15: The method used to return the next Migration object for the processing job

Listing 4.16 shows a method implemented with native SQL syntax to allow executing the update directly in the database, preventing race conditions that can happen in parallel execution. The managed object is updated after execution to have the updated value.

```
1 def increase_migration_processed_items_number(self, migration:
2     Migration, number: int):
3     """
4     Increase the value of the field processed_items_number in
5     migration by number and commit.
6     Value of last_update is updated.
7     It is a native update to prevent race condition
8     Args:
9         migration: The migration object.
10        number: Increase of processed_items_number.
11    """
12    update = "UPDATE migrations SET processed_items_number =
13    processed_items_number + :n, last_update=now() WHERE id= :id"
14    self.session.execute(text(update), {"n": number, "id":
15    migration.id})
16    self.session.commit()
17    self.refresh(migration)
```

Listing 4.16: The method `increase_migration_processed_items_number` used to update the value of the field `processed_items_number`

The class **FactserverConnectionRepo** provides the method **store_connection_and_set_valid_until_for_existing_conn** (Listing 4.17) that creates a new factserver connection and updates the value of the field **valid_unit** for all existing valid connections to the current date (`now()`).

```

1 def store_connection_and_set_valid_until_for_existing_conn(self,
2   connection: FactserverConnection):
3     """
4     Store new factserver connection and invalidate existing
5     valid connections by setting valid until to current date.
6     Args:
7     connection(FactserverConnection): connection object
8     """
9     # update existing connections
10    (FactserverConnection.query.filter_by(valid_until=None).
11     update({FactserverConnection.valid_until: datetime.now()},
12            synchronize_session=False))
13    # save new connection and commit
14    self.save(connection)

```

Listing 4.17: The method `store_connection_and_set_valid_until_for_existing_conn` used create a new factserver connection

4.1.6. Views

Database views are used to facilitate the call of complex select statements. The file **views.py** has a list of views that will be created at the start of the application. In addition, the file provides the methods to create and drop the defined list of views. At the start of the application, the views are dropped and then created. This allows easier development and refactoring of the views, because PostgreSQL has some restrictions for updating existing views, like the order of existing fields must be kept the same as before.

Listing 4.18 shows the list of strings, each containing the SQL statement to create a view. The method **create_views**, used to insert the views, is shown in Listing 4.19.

```

1 views_sqls = [
2     """
3     create or replace view v_running_migrations as
4     select * from migrations m where m.start_date is not null
5     and finish_date is null and error is null
6     """
7     ,
8     """
9     create or replace view v_blocked_migrations as
10    select * from migrations m
11    where m.start_date is null --not processed
12    and m.scheduled_for < now() -- scheduled
13    and
14    (

```

4. Implementation

```
14     exists -- source is not target of running m having at least
15     one overlapping collection
16     (select 1 from v_running_migrations rm where
17       m.source_db_type=rm.target_db_type and m.source_db_id=rm.
18       target_db_id and m.collections && rm.collections)
19     or exists -- target is not source or a target of running m
20     having at least one overlapping collection
21     (select 1 from v_running_migrations rm where
22       (m.target_db_type=rm.source_db_type and m.target_db_id=rm.
23       source_db_id and m.collections && rm.collections) or
24       (m.target_db_type=rm.target_db_type and m.target_db_id=rm.
25       target_db_id and m.collections && rm.collections))
26   )
27   """
28   ,
29   """
30   create or replace view v_migrations_for_api as
31   select m.*,
32   coalesce(s_mongo.name,s_couch.name,s_cosmos.name) as
33   source_db_name,
34   coalesce(t_mongo.name,t_couch.name,t_cosmos.name) as
35   target_db_name,
36   (case when exists (select * from v_blocked_migrations where
37     id=m.id) then true else false end) as is_blocked
38   from migrations m
39   left join mongodb_connections s_mongo
40   on m.source_db_type = s_mongo."type" and m.source_db_id =
41   s_mongo.id
42   left join couchdb_connections s_couch
43   on m.source_db_type = s_couch."type" and m.source_db_id =
44   s_couch.id
45   left join cosmosdb_connections s_cosmos
46   on m.source_db_type = s_cosmos."type" and m.source_db_id =
47   s_cosmos.id
48   left join mongodb_connections t_mongo
49   on m.target_db_type = t_mongo."type" and m.target_db_id =
50   t_mongo.id
51   left join couchdb_connections t_couch
52   on m.target_db_type = t_couch."type" and m.target_db_id =
53   t_couch.id
54   left join cosmosdb_connections t_cosmos
55   on m.target_db_type = t_cosmos."type" and m.target_db_id =
56   t_cosmos.id
57   """]
```

Listing 4.18: The list of views used by the system

```

1 def create_views():
2     """
3     Creates the views stored in the list views_sqls in the
4     config db.
5     """
6     for view in views_sqls:
7         config_db.session.execute(text(view))
8         config_db.session.commit()

```

Listing 4.19: The method `create_views` iterate over the views list and creates the views

Listing 4.20 shows the string variable `drop_views_sql`, which contains a script to drop all views, considering the dependencies between the views. The methods for creating and dropping the database views are called in the file `main.py`.

```

1 drop_views_sql = """
2     drop view if exists v_migrations_for_api;
3     drop view if exists v_migrations_for_processing;
4     drop view if exists v_blocked_migrations;
5     drop view if exists v_running_migrations;
6     """

```

Listing 4.20: The drop views script

4.1.7. Connection Managers

The abstract class **AbstractDbManager** (Listing 4.21) defines the generic API that each connection manager needs to provide in order to be used for migration processes. It has methods to establish a connection, close it, and perform basic operations on the database structure, like creating a new collection, removing an existing collection, and computing the number of items in a collection.

4. Implementation

```
1 class AbstractDbManager(ABC):
2     """
3     Abstract class for handling database connections.
4     """
5     migration_config = load_config().migration_config
6     @abstractmethod
7     def connect(self):
8         """
9         Establish a database connection and store it
10        """
11        pass
12
13    @abstractmethod
14    def get_collection_or_create_new(self, collection_name):
15        """
16        Get a collection object by name when exists, or create
17        and return a new one
18        Args:
19            collection_name (str): the collection name
20        Returns:
21            An instance of the collection
22        """
23        pass
24
25    @abstractmethod
26    def get_connection_info(self) -> DBInfo:
27        """
28        Get a list of info about each collection in the
29        connection
30        Returns:
31            ([DBInfo]): The list of info about each collection
32            in the connection
33        """
34        pass
35
36    @abstractmethod
37    def create_collection_indices(self, collection_name,
38                                migration_id=None):
39        """
40        Create the configured indices for the collection.
41        Args:
42            collection_name: The collection name
43            migration_id: Migration id, used for logging (
44            default: None)
45        """
46        pass
```

Listing 4.21: Snippet from AbstractDbManager

Each implementation of the abstract class represents a manager for a specific database technology. Listing 4.22 shows the implementations of the abstract methods for MongoDB.

```

1 class MongoDBManager(AbstractDbManager):
2     """
3     A class to manage MongoDB connections.
4     """
5     connections_config = load_config().connections_config
6
7     def __init__(self, connection_data: MongoddbConnection):
8         """
9         Constructor for MongoDBManager, store connection data
10        and establish connection.
11        Args:
12            connection_data (MongoDBManager): Connection data (
13            host, username, password, db_name)
14        """
15        self.connection_data: MongoddbConnection =
16        connection_data
17        self.client = None
18        self.db = None
19        self.connect()
20
21    @override
22    def connect(self):
23        if self.client:
24            self.client.close()
25        is_successful, client, db, error = self.
26        establish_connection(self.connection_data)
27        if not is_successful:
28            raise Exception(error)
29        self.client = client
30        self.db = db
31
32    @override
33    def close_connection(self):
34        if self.client:
35            self.client.close()
36            log("MongoDB connection closed.")
37
38    @override
39    def get_collection_size_or_0(self, collection_name):
40        return self.get_collection(collection_name).
41        count_documents({}) if self.has_collection(collection_name)
42        else 0

```

Listing 4.22: Snippet from MongoDbManager

4. Implementation

Listing 4.23 shows a method provided to create indices for MongoDB collections. The method is used in the manager method `create_collection_indices` to create indices for a target collection before migration. The dictionary `mongodb_indices` contains the indices that should be created for each collection. The key is the name of the collection, and the value is a list of indices that need to be created. The current implementation has support for the following collections: `facts`, `cache`, and `factstats`. Changing the indices configuration can be done very easily by updating the `mongodb_indices` directly in the file `mongodb_indices.py`

```
1 # key is the name of the collection and the value is a list of
  # indices,
2 # simple index is defined as a string item and compound index as
  # a list of strings
3 mongodb_indices: dict = {
4     'facts': ['userid', 'mid', 'source', ['data.name', 'data.
  @type']], ['source', 'data.name']],
5     'cache': ['mid', 'date'],
6     'factstats': ['source', 'mid', 'type', 'stats']
7 }
8
9
10 def create_indices_for_mongo_db(collection, migration_id: int =
  None):
11     """
12     Create indices for mongodb Collection based on configured
  indices in mongodb_indices.
13     Index name: <field1_name>_<field2_name>_idx.
14     Indices order is ASCENDING.
15     Args:
16         collection: MongoClient collection instance
17         migration_id (int): id of migration, used for logging
18     """
19     if collection.name in mongodb_indices:
20         for index_field in mongodb_indices[collection.name]:
21             if isinstance(index_field, list): # compound
22                 index = [(field, ASCENDING) for field in
  index_field]
23                 index_name = f'{'_'.join(index_field)}_idx'
24             else:
25                 index = [(index_field, ASCENDING)]
26                 index_name = f'{index_field}_idx'
27                 collection.create_index(index, name=index_name)
28                 log_collection_event(migration_id, collection.name,
  f"Creating MongoDB index: {index_field}")
```

Listing 4.23: Snippet from `mongodb-indices`

Listing 4.24 shows a snippet of the implementations of abstract methods for CouchDB.

```

1 class CouchDBManager(AbstractDbManager):
2     """
3     A class to manage CouchDB connections.
4     """
5     @override
6     def empty_collection(self, collection_name, migration_id=
None, log_count=False):
7         if self.has_collection(collection_name):
8             log_collection_event(migration_id, collection_name,
9                                 "Start emptying CouchDb
collection: this process may take some time!!!")
10            message = 'Empty CouchDB collection'
11            if log_count:
12                count = self.get_collection_size_or_0(
collection_name)
13                message += ': deleted items: {}'.format(count)
14
15            db = self.get_collection(collection_name)
16            for doc_id in list(db):
17                if not doc_id.startswith("_design/"):
18                    db.delete(db[doc_id])
19            log_collection_event(migration_id, collection_name,
message)
20
21    @override
22    def drop_collection(self, collection_name, migration_id=None
, log_count=False):
23        if self.has_collection(collection_name):
24            message = 'Drop CouchDB collection'
25            if log_count:
26                count = self.get_collection_size_or_0(
collection_name)
27                message += ': deleted items: {}'.format(count)
28
29            del self.client[collection_name]
30
31            log_collection_event(migration_id, collection_name,
message)
32
33    @override
34    def create_collection_indices(self, collection_name,
migration_id=None):
35        if self.has_collection(collection_name):
36            collection = self.get_collection(collection_name)
37            create_views_for_couch_db(collection, migration_id)

```

Listing 4.24: Snippet from CouchDbManager

4. Implementation

Listing 4.25 and Listing 4.26 provide the implementation of the methods to import CouchDB views and to create views for CouchDB databases from the imported views. This helps migrate the database setup in CouchDB. The method **import_couch_design_docs(client)** is used to import the design documents of a CouchDB server and store them as JSON files inside a folder, each for a specific database. The method **create_views_for_couch_db** checks if a folder with the same name as the database exists, and if yes, it creates the design documents from it.

The importing can be triggered by starting the test **test_import_couch_design_docs** that is configured to read from the CouchDB server (**25s-fcdb.cs.univie.ac.at:5984**). The design documents have already been imported and are used for the migration processes. Figure 4.1 illustrates the folder structure of the imported design documents. Listing 4.27 shows the imported design document **ids.json**.

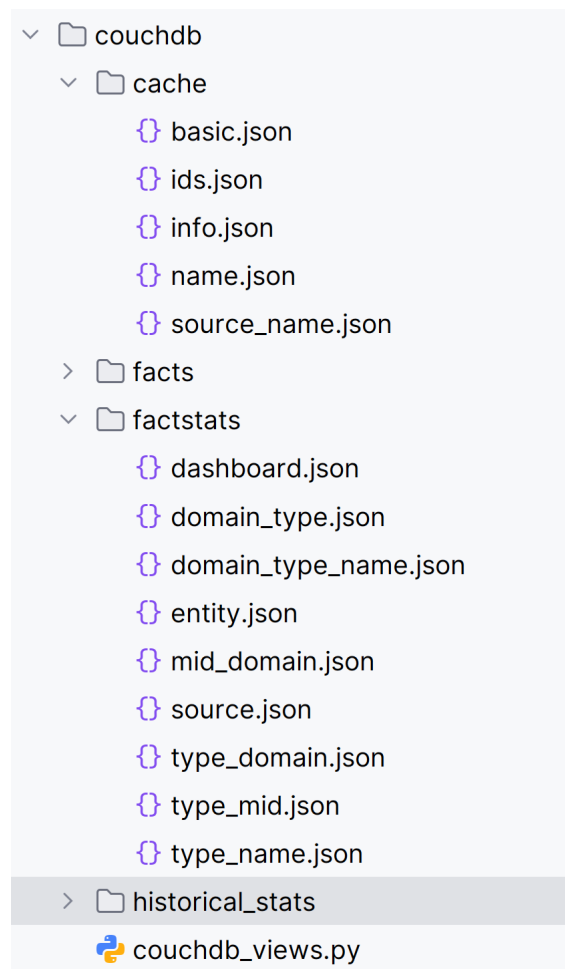


Figure 4.1.: The structure of imported design documents

```

1 def import_couch_design_docs(client):
2     """
3     Import all design documents from CouchDB client by calling
4     the view '_all_docs'
5     Databases _users and _replicator are excluded
6     For each database, the design documents are imported and
7     stored in a folder called as the name of the database ./<
8     db_name> like /cache,
9     Each design document is stored in a json file after removing
10    the field: _rev.
11    Use the test 'test_import_couch_design_docs' to import the
12    design documents of a Couchdb connection.
13    Args:
14        client: CouchDB database instance
15    """
16    for db_name in client:
17
18        # skip
19        if db_name in ('_users', '_replicator'):
20            continue
21
22        db = client[db_name]
23        db_dir = Path(__file__).parent / db_name
24        db_dir.mkdir(exist_ok=True)
25
26        print(f"Processing database: {db_name}")
27
28        for row in db.view("_all_docs", startkey="_design/",
29                           endkey="_design/\uffff0"):
30            doc_id = row.id
31            design_doc = db[doc_id]
32
33            # remove rev
34            design_doc.pop("_rev", None)
35
36            # File name: _design_name.json
37            file_name = f"{doc_id.replace('_design/', '')}.json"
38            file_path = db_dir / file_name
39
40            with open(file_path, "w", encoding="utf-8") as f:
41                json.dump(design_doc, f, indent=2, ensure_ascii=
42                False)
43
44            print(f"        Saved {file_name}")
45            # remove dir if empty:
46            if not any(db_dir.iterdir()):
47                db_dir.rmdir()

```

Listing 4.25: Snippet from couchdb_views.py: import_couch_design_docs

4. Implementation

```
1 def create_views_for_couch_db(db, migration_id: int = None):
2     """
3     Creates design document for the provided CouchDB database
4     The related design documents are taken from the folder
5     called ./<db_name> like /cache,
6     if the folder does not exist, no views are created.
7     Existing design documents in the database instance are
8     updated when they are in the folder, otherwise they are not
9     affected.
10    Args:
11        db: CouchDB database instance
12        migration_id: The id of the migration, used for logging
13    """
14    couch_views = _load_all_couch_views()
15    if db.name in couch_views:
16        design_docs = couch_views[db.name]
17        for design_doc in design_docs:
18            design_doc.pop('_rev', None)
19            doc_id = design_doc["_id"]
20            action = 'Creating'
21            if doc_id in db:
22                action = 'Updating'
23                existing_design_doc = db[doc_id]
24                design_doc["_rev"] = existing_design_doc["_rev"]
25            db.save(design_doc)
26            log_collection_event(migration_id, db.name, f"{
27                action} couch_db design_document: {doc_id}")
```

Listing 4.26: Snippet from couchdb_views.py: create_views_for_couch_db

```
1 {
2     "_id": "_design/ids",
3     "language": "javascript",
4     "views": {
5         "source-name": {
6             "map": "function(doc) { emit([doc.source, doc.name], doc.
7             mid); }"
8         },
9         "mid": {
10            "map": "function(doc) { emit(doc.mid, doc.mid); }"
11        },
12        "name": {
13            "map": "function(doc) { emit(doc.name, doc.mid); }"
14        },
15        "id": {
16            "map": "function(doc) { emit(doc._id, doc.mid); }"
17        }
18    }
19 }
```

Listing 4.27: The design document ids.json for database cache

4.1.8. Migration Managers

This subsection focuses on the implementation of the class `MigrationManager`, the abstract class `Migrator`, and all its subclasses.

The class `MigrationManager` provides a method to prepare and start migration processes. The main method of the class is `start_migration(self, migration: Migration)` (Listing 4.28), which takes a migration object, finds the appropriate connection managers and the suitable migrator based on the source and target types, and runs the method `migrate` that is provided by the migrator class.

```

1     def start_migration(self, migration: Migration):
2         """
3         Preparation:
4         - Set start_date
5         - Create Source and Target Manager instances
6         - Set source_items_number
7         - Find the suitable migrator
8         Start a migration
9         Args:
10            migration: the migration instance
11        """
12        log_migration_event(migration.id, "Start processing
migration")
13        try:
14            # preparation
15            migration.start_date = datetime.now()
16            self.migration_repo.save(migration)
17            source_manager, target_manager = self.
get_connections_manager(migration)
18            migration.source_items_number = source_manager.
get_number_of_items_in_collections_if_existing(
19                migration.collections)
20            self.migration_repo.save(migration)
21            migrator = self.get_migrator(migration)
22            # processing
23            migration = migrator.migrate(migration,
source_manager, target_manager)
24        except Exception as e:
25            log_migration_event(migration.id, f"Error while
processing migration: {str(e)}", logging.ERROR)
26            migration.error = str(e)
27            traceback.print_exc()
28        finally:
29            migration.finish_date = datetime.now()
30            self.migration_repo.save(migration)
31            log_migration_event(migration.id, "Finish processing
migration")

```

Listing 4.28: The method `start_migration` in `MigrationManager`

4. Implementation

The file **migrators.py**, shown in Listing 4.29, provides a list containing instances of all implementations of the abstract class *Migrator*. It scans all classes in a specific folder and initiates an object of the found classes.

```
1 def import_migrator_classes(packages):
2     """
3     Dynamically import all classes of a given package.
4     Method used to import all migrator classes
5     Args:
6         packages: List of packages to import
7     """
8     for package in packages:
9         for _, module_name, _ in pkgutil.walk_packages(package.
10            __path__, package.__name__ + "."):
11             importlib.import_module(module_name)
12
13 import_migrator_classes([sequential])
14
15 # a list of objects from all subclasses of Migrator
16 migrators: [Migrator] = [cls() for cls in Migrator.
17    __subclasses__()]
```

Listing 4.29: The file `migrators.py`

Listing 4.30 shows how the provided list is used by the *MigrationManager* to find the appropriate migrator for each migration process.

```
1 @staticmethod
2 def get_migrator(migration: Migration):
3     """
4     Find the suitable migrator for the migration.
5     Returns the first migrator where is_applicable(migration) is
6     True
7     Args:
8         migration(Migration): The migration instance
9     Returns:
10        (Migrator) The first migrator where is_applicable(
11        migration) is True
12    Raises:
13        Exception: When no suitable migrator is found
14    """
15    for migrator in migrators:
16        if migrator.is_applicable(migration):
17            return migrator
18    raise Exception(f"Cannot find applicable Migrator for
19    migration: {migration.id}")
```

Listing 4.30: The method `get_migrator`

The abstract class **Migrator** contains almost all the migration logic. It and its subclasses follow the **Template** design pattern, where Migrator defines the sequence of operations and delegates the implementation of some methods to its subclasses. Listing 4.31 shows the method **is_applicable**, which checks if the migrator supports the migration object given as a parameter.

```

1 def is_applicable(self, migration: Migration) -> bool:
2     """
3     Check source and target types of the migration match the
4     supported source and target types in the class.
5     Args:
6         migration:
7     Returns:
8         (bool): True when the migration can be processed by the
9         class, otherwise False.
10        It checks the source type, target type of the migration
11        must match the supported source and target type of the class.
12        Currently only sequential processing of data inside each
13        collection in migrations is supported.
14    """
15    return (self.get_source_type().value == migration.
16            source_db_type and
17            self.get_target_type().value == migration.
18            target_db_type and
19            self.is_parallel() == False) # currently only
20    sequential processing inside the collection

```

Listing 4.31: The method **is_applicable** returns a boolean value based on source and target types

The main method in Migrator is **migrate** (Listing 4.32). It iterates over the collections to be migrated and, based on the field **is_parallel**, starts the migration for them either in sequence or in parallel. The processing of collections is done in the method **process_collection** (Listing 4.34).

4. Implementation

```
1 @log_migration_execution
2 def migrate(self, migration: Migration, source_manager:
    AbstractDbManager, target_manager: AbstractDbManager):
3     """
4     Starts the migration process for a given migration.
5     When migration.is_parallel is True, collections are
6     processed in parallel,
7     the number of worker for parallel processing is taken from
8     config: parallel_processing_threads_num
9
10    ‘‘Note:‘‘ This method invalidates the migration object when
11    it is processed in parallel, hence reload and return!!!!!!
12    Args:
13    migration (Migration): The migration instance
14    source_manager: The source database manager instance
15    target_manager: The target database manager instance
16    Returns:
17    (Migration): The newly reloaded migration instance
18    """
19    if migration.is_parallel:
20        with ThreadPoolExecutor(max_workers=self.
21            migration_config.parallel_processing_threads_num) as executor
22        :
23            futures = []
24            for collection in migration.collections:
25                futures.append(
26                    executor.submit(self.process_collection,
27                        source_manager, target_manager, migration,
28                        collection))
29            for future in futures: # to detect any errors!!
30                future.result()
31    else:
32        for collection in migration.collections:
33            self.process_collection(source_manager,
34                target_manager, migration, collection)
35
36    target_manager.close_connection()
37    source_manager.close_connection()
```

Listing 4.32: The method migrate

The method migrate is annotated with **@log_migration_execution** (Listing 4.33), which logs the start and the end of the execution with the execution time of the process. The annotation **@log_collection_execution** has similar functionality and is used on the collection level.

```

1 def log_migration_execution(func):
2     """
3     Annotation to log the start, end and duration of a migration
4     process.
5     The target function must have:
6     - The 'migration' object as the 'first' argument.
7     """
8     @functools.wraps(func)
9     def wrapper(*args, **kwargs):
10         migration = args[1]
11         start = time.time()
12         log_migration_event(migration.id, f"Start migrating of
13         collections: {migration.collections}")
14         result = func(*args, **kwargs)
15         end = time.time()
16         execution_time = str(timedelta(seconds=(round(end -
17         start))))
18         log_migration_event(migration.id, f"Finish migrating all
19         collections in: {execution_time}")
20         return result
21     return wrapper

```

Listing 4.33: The annotation log_migration_execution

The method **process_collection** (Listing 4.34) applies pre- and postprocessing actions (Listing 4.35 and Listing 4.36) and migrates the data by calling the method **migrate_collection**, which is delegated to the subclasses of Migrator.

4. Implementation

```
1 @log_collection_execution
2 def process_collection(self, source_manager: AbstractDbManager,
3     target_manager: AbstractDbManager,
4     migration: Migration, collection_name:
5     str):
6     """
7     Start the migration process for a specific collection.
8
9     Preprocessing:
10     - Check collection exists
11     - Empty target collection
12
13     Processing:
14     - Migrate data items
15
16     Postprocessing:
17     - Update list of migrated collection in migration
18
19     Args:
20     source_manager: The source database manager instance
21     target_manager: The target database manager instance
22     migration: The migration instance
23     collection_name: The name of the collection to be
24     processed
25     """
26     with app.app_context(): # to support threads!!
27         migration = self.migration_repo.get_by_id(migration.id)
28         collection_size = self.preprocess_collection_migration(
29             source_manager, target_manager, migration.id,
30             collection_name)
31         if collection_size == False:
32             return
33
34         tot_processed_count, tot_migrated_count = self.
35         migrate_collection(source_manager, target_manager, migration,
36                             collection_name, collection_size)
37
38         self.postprocess_collection_migration(migration,
39             collection_name, tot_processed_count, tot_migrated_count)
```

Listing 4.34: The method process_collection

```

1  def preprocess_collection_migration(self, source_manager:
    AbstractDbManager, target_manager: AbstractDbManager,
2                                     migration_id,
    collection_name: str):
3      """
4          - Check the collection exists
5          - Log collection size
6          - Empty target collection
7      Args:
8          source_manager: The source database manager instance
9          target_manager: The target database manager instance
10         migration_id: The id of the migration
11         collection_name: The name of the collection to be
    processed
12     Returns:
13         When collection exists: collection size, False
    otherwise
14     """
15     if source_manager.has_collection(collection_name) ==
    False:
16         log_collection_event(migration_id, collection_name,
    'Skip, collection not found in source!')
17         return False
18
19     collection_size = source_manager.
    get_collection_size_or_0(collection_name)
20
21     log_collection_event(migration_id, collection_name,
22                         'Start migration, count of
    collection items: {}'.format(collection_size))
23
24     log_collection_event(migration_id, collection_name, "
    Emptying target collection")
25     target_manager.empty_collection_for_migration(
    collection_name, migration_id)
26     return collection_size

```

Listing 4.35: The method preprocess_collection_migration

4. Implementation

```
1 def postprocess_collection_migration(self, migration: Migration,
2     collection_name: str, tot_processed_count: int,
3     tot_migrated_count: int):
4     """
5     - Update list of migrated collection in migration
6     - Log Completion
7     Args:
8         migration: The migration instance
9         collection_name: The name of the collection to be
10        processed
11        tot_processed_count: Total number of processed items
12        tot_migrated_count: Total number of successfully
13        migrated items
14    """
15    self.migration_repo.update_migration_migrated_collections(
16        migration, collection_name)
17
18    log_completed_collection_migration(migration.id,
19        collection_name, tot_processed_count, tot_migrated_count)
```

Listing 4.36: The method `postprocess_collection_migration`

For each permutation of source and target database types, a specific subclass of migrator is implemented. To support all permutations for n database types, n^2 subclasses need to be implemented. This means for the three supported database types, $3^2 = 9$ subclasses are required for full support.

All subclasses are named in the following format:

<sourceType>To<targetType>SeqMigrator,

e.g.: **CouchDbToMongoDbSeqMigrator**, **MongoDbToCosmosDbSeqMigrator**

All subclasses implement the abstract methods of Migrator, which are the `get_source_type`, `get_target_type`, and `is_parallel`. These methods are used in the method `is_applicable` to find the correct implementation for a specific migration process. Listing 4.37 shows the implementation of the above-mentioned abstract methods of Migrator in **CouchDbToMongoDbSeqMigrator**.

```

1 class CouchDbToMongoDbSeqMigrator(Migrator):
2     """
3     Subclass of migrator to migrate from CouchDB to MongoDB.
4     Items of the same collection are migrated sequentially.
5     """
6
7     @staticmethod
8     @override
9     def get_source_type() -> ConnectionType:
10        """
11        The type of database, this class can migrate from.
12        Returns:
13            ConnectionType.COUCH_DB
14        """
15        return ConnectionType.COUCH_DB
16
17     @staticmethod
18     @override
19     def get_target_type() -> ConnectionType:
20        """
21        The type of database, this class can migrate to.
22        Returns:
23            ConnectionType.MONGO_DB
24        """
25        return ConnectionType.MONGO_DB
26
27     @staticmethod
28     @override
29     def is_parallel() -> bool:
30        """
31        When True, the processing of the data inside each
32        Container is done in parallel
33        Returns:
34            False
35        """
36        return False

```

Listing 4.37: The implementation of the abstract methods of Migrator in CouchDbToMongoDbSeqMigrator

The method **migrate_collection** is responsible for reading the data from the source collection and writing it into the target collection.

Listing 4.38 shows the implementation of the method in the class **CouchDbToMongoDbSeqMigrator**, which is used to migrate from CouchDB into MongoDB. First of all, the collection object is initialized, then the items of the CouchDB database are iterated in sequence. For each item, the method **remove_couchdb_fields** is called to clean up the data by removing the fields that are only CouchDB relevant, e.g.: **_rev**. The fields that need to be removed are configurable via an environment variable. For further details

4. Implementation

on environment variables, see Subsection 4.1.11.

As MongoDB has an api that accepts batch inserts, the method stores a list of items and inserts them at once, which drastically increases the performance. Once the batch insert size has been reached or the last item is being processed, the list will be inserted in MongoDB, and the number of processed and migrated items will be increased. The term "migrated items" refers to the items successfully migrated, whereas "processed items" refers to both successfully migrated and failed items. Finally, the method returns the total number of processed and migrated items, which will be used for logging.

```
1 def migrate_collection(self, couch_manager, mongo_manager,
2   migration: Migration, collection_name: str, collection_size:
3   int):
4
5     couchdb_database = couch_manager.
6     get_collection_or_create_new(collection_name)
7     mongodb_collection = mongo_manager.
8     get_collection_or_create_new(collection_name)
9
10    tot_processed_count = 0
11    tot_migrated_count = 0
12    temp_processed_count = 0
13    temp_migrated_count = 0
14    docs = []
15
16    for doc_id in couchdb_database:
17
18        temp_processed_count += 1
19        tot_processed_count += 1
20
21        doc = couchdb_database[doc_id]
22        self.remove_couchdb_fields(doc)
23        docs.append(doc)
24
25        if (tot_processed_count != 0 and (tot_processed_count %
26    self.migration_config.mongodb_batch_insert_size == 0
27    or tot_processed_count
28    == collection_size)):
29
30            try:
31                mongodb_collection.insert_many(docs, ordered=
32    False) # complete insertion even some items failed
33                temp_migrated_count += len(docs)
34            except Exception as e:
35                log_collection_event(migration.id,
36    collection_name, f"Error inserting document {doc_id}: {e}",
37    logging.ERROR)
38                temp_migrated_count += len(docs) - len(e.details
39    ["writeErrors"])
40            finally:
```

```

32         docs.clear()
33
34         log_collection_event(migration.id, collection_name,
35                             "processed {} documents out of
36                             {}".format(tot_processed_count,
37                                       collection_size))
38         self.migration_repo.increase_migration_processed_items_number(migration,
39                               temp_processed_count)
40         self.migration_repo.increase_migration_migrated_items_number(migration,
41                               temp_migrated_count)
42         tot_migrated_count += temp_migrated_count
43         temp_processed_count, temp_migrated_count = 0, 0
44
45     return tot_processed_count, tot_migrated_count

```

Listing 4.38: The implementation of `migrate_collection` in `CouchDbToMongoDbSeqMigrator`

4.1.9. Jobs

The main job in the application is the **`migration_processing_job`** (Listing 4.39); it is responsible for running the scheduled migration processes. Every time the job starts, it searches for scheduled migrations and executes exactly one migration process. The number of instances and the execution interval are configurable via environment variables.

```

1 migration_repo = MigrationRepo(config_db.session)
2 migration_manager = MigrationManager()
3 def migration_processing_job():
4     """
5     Method used for a Cron Job to start migration processes.
6     """
7     with app.app_context():
8         migration = migration_repo.get_next_migration_for_processing()
9         if migration is None:
10             return
11
12         try:
13             migration_manager.start_migration(migration)
14         except Exception as e:
15             log_migration_event(migration.id, f"Error while
16                               processing migration:{str(e)}", logging.ERROR)

```

Listing 4.39: The job `migration_processing_job`

4. Implementation

Listing 4.40) shows the job **failing_migration_detector_job** which is responsible for detecting failing migrations and updating their status. Only one instance can be running at the same time, whereas the interval is configurable.

```
1 def failing_migration_detector_job():
2     """
3     Method used for a Cron Job to detect failing migration
4     processes.
5     It detects running migrations that have not been updated for
6     a configured time.
7     Set error and finish_date in the detected migration.
8     """
9     with app.app_context():
10        migrations = migration_repo.
11        get_migrations_not_updated_since_m_min(
12            job_config.error_after_in_minutes)
13        for migration in migrations:
14            migration.error = job_config.error_message.format(
15                job_config.error_after_in_minutes)
16            migration.finish_date = datetime.now()
17            migration_repo.save(migration, False)
18            log_migration_event(migration.id, 'Idle migration
19            detected and set to error!', logging.WARNING)
```

Listing 4.40: The job failing_migration_detector_job

Listing 4.41 shows the registration and configuration of the jobs in **main.py**.

```
1 config = load_config()
2 processing_job_config = config.processing_job_config
3 failing_migration_detector_job_config = config.
4     failing_migrations_detector_job_config
5 scheduler = BackgroundScheduler()
6 scheduler.add_job(id='migration_processing_job', func=
7     migration_processing_job, trigger="interval",
8     replace_existing=True,
9     seconds=processing_job_config.interval_in_sec,
10    max_instances=processing_job_config.max_instances)
11 scheduler.add_job(id='failing_migrations_detector_job', func=
12     failing_migration_detector_job, trigger="interval",
13     replace_existing=True,
14     seconds=failing_migration_detector_job_config.
15     interval_in_sec, max_instances=1)
```

Listing 4.41: The registration of jobs in main.py

4.1.10. Logging

The file `logging_utils.py` provides methods to support a unique format for all migration and collection-related logs.

Listing 4.42 illustrates the logging format for all log messages. It contains a timestamp, the log level, and the logged message. The log level of the application is configurable via an environment variable.

```

1 logging.basicConfig(level=logging_config.log_level, format='%(
    asctime)s - %(levelname)s - %(message)s',
2                             datefmt='%d.%m.%Y %I:%M:%S.%MS')
3
4 logging.getLogger('azure.cosmos').setLevel(logging_config.
    azure_cosmos_log_level)
5 logging.getLogger('azure.core.pipeline.policies.
    http_logging_policy').setLevel(logging_config.
    azure_cosmos_log_level)

```

Listing 4.42: The configuration of logging in `logging_utils.py`

The method `log_migration_event`, shown in Listing 4.43, is used as a unified logging method for all migration-related events. It logs the migration ID and the event, which helps to identify the process being logged and the filtering of events.

```

1 def log_migration_event(migration_id, message, level=logging.
    INFO):
2     """
3     Log a message related to the migration process
4     Format: Migration id: {migration_id}, {message}
5     Args:
6         migration_id: The id of the migration process
7         message: The message to log
8         level: The log level (default: logging.INFO)
9     """
10    log(f'Migration id: {migration_id}, {message}', level=level)

```

Listing 4.43: The method `log_migration_event`

The method `log_collection_event` (Listing 4.44) is used as a unified logging method for all collection-related events. It logs the migration ID, the collection name, and the event, which helps to identify the collection being processed and the filtering of events.

4. Implementation

```
1 def log_collection_event(migration_id, collection_name, message,
2   level=logging.INFO):
3     """
4     Log a message related to the migration process of a
5     collection
6     Formats:
7         - Migration id is missing=> Collection: {
8           collection_name}, {message}
9         - Else: => Migration id: {migration_id}, Collection: {
10          collection_name}, {message}
11     Args:
12         migration_id: The ID of the migration process
13         collection_name: The name of the collection
14         message: The message to log
15         level: The log level (default: logging.INFO)
16     """
17     if not migration_id:
18         message = f'Collection: {collection_name}, {message}'
19     else:
20         message = f'Migration id: {migration_id}, Collection: {
21         collection_name}, {message}'
22     log(message, level=level)
```

Listing 4.44: The method `log_collection_event`

The method `log_completed_collection_migration` (Listing 4.45) logs the completion of the migration of a collection. It is used by all subclasses of `Migrator`. It logs the migration ID, the collection name, and the number of processed and migrated items.

```
1 def log_completed_collection_migration(migration_id,
2   collection_name, tot_processed_count, tot_migrated_count):
3     """
4     Log a message when the migration process of a collection has
5     been completed
6     Format: Migration completed: processed: {tot_processed_count}
7     documents, migrated: {tot_migrated_count} documents.
8     Args:
9         migration_id: The ID of the migration process
10        collection_name: The name of the collection
11        tot_processed_count: Total number of processed documents
12        tot_migrated_count: Total number of successfully
13        migrated documents
14    """
15    log_collection_event(migration_id, collection_name,
16                        f'Migration completed: "
17                        f"processed: {tot_processed_count}
18                        documents, migrated: {tot_migrated_count} documents.")
```

Listing 4.45: The method `log_completed_collection_migration`

4.1.11. Config

The class **Config** (Listing 4.46) provides access to all configuration parameters in the application. All configuration parameters are given as environment variables.

```

1 @dataclass
2 class Config:
3     """
4     Dataclass to hold instances of all settings classes.
5     """
6     db_config: DatabaseConfig
7     migration_config: MigrationConfig
8     processing_job_config: ProcessingJobConfig
9     failing_migrations_detector_job_config:
10     FailingMigrationsDetectorJobConfig
11     connections_config: ConnectionsConfig
12     logging_config: LoggingConfig

```

Listing 4.46: The class config

The file **config.py** provides the method **load_config** (Listing 4.47) that initiates and returns an object of the class **Config**, where the values of the variables are taken from either the environment variables or the defined default values.

```

1 def load_config() -> Config:
2     """
3     Read config data from environment variables and return a
4     Config object.
5     Returns:
6     (Config): Config object.
7     """
8     return Config(
9         db_config=DatabaseConfig(
10             host=os.getenv("DB_HOST", "localhost"),
11             port=int(os.getenv("DB_PORT", 5432)),
12             db_name=(os.getenv("DB_NAME", "DBManagerConfig")),
13             user=os.getenv("DB_USER", "postgres"),
14             password=os.getenv("DB_PASSWORD", "postgres")
15         ),
16         migration_config=MigrationConfig(
17             mongodb_batch_insert_size=os.getenv('
18             MONGODB_BATCH_INSERT_SIZE', 100),
19             mongodb_clean_collection_by_drop=os.getenv('
20             MONGODB_CLEAN_COLLECTION_BY_DROP', True),
21             couchdb_clean_collection_by_drop=os.getenv('
22             COUCHDB_CLEAN_COLLECTION_BY_DROP', True),
23             cosmosdb_clean_collection_by_drop=os.getenv('
24             COSMOSDB_CLEAN_COLLECTION_BY_DROP', True),
25             mongodb_fields_to_remove=os.getenv('
26             MONGODB_FIELDS_TO_REMOVE', ''),

```

4. Implementation

```
21         couchdb_fields_to_remove=os.getenv('
    COUCHDB_FIELDS_TO_REMOVE', ''),
22         cosmosdb_fields_to_remove=os.getenv('
    COSMOSDB_FIELDS_TO_REMOVE', 'id,_rid,_self,_etag,_attachments
    ,_ts'),
23         cosmosdb_id_from__id=os.getenv('COSMOSDB_ID_FROM__ID
    ', True),
24         cosmosdb_partition_key=os.getenv('
    COSMOSDB_PARTITION_KEY', 'id'),
25         update_number_processed_on=os.getenv('
    UPDATE_NUMBER_PROCESSED_ON', 100),
26         parallel_processing_threads_num=os.getenv('
    PARALLEL_PROCESSING_THREADS_NUM', os.cpu_count()),
27         create_known_indices=os.getenv('CREATE_KNOWN_INDICES
    ', True)
28     ),
29     processing_job_config=ProcessingJobConfig(
30         interval_in_sec=int(os.getenv("
    PROCESSING_JOB_INTERVAL_IN_SEC", 10)),
31         max_instances=int(os.getenv("
    PROCESSING_JOB_INSTANCES_NUM", 1))
32     ),
33     failing_migrations_detector_job_config=
    FailingMigrationsDetectorJobConfig(
34         interval_in_sec=int(os.getenv("
    FAILING_DETECTOR_JOB_INTERVAL_IN_SEC", 60)),
35         error_after_in_minutes=int(os.getenv("
    FAILING_DETECTOR_ERROR_AFTER_IN_MIN", 10)),
36         error_message=os.getenv("
    FAILING_DETECTOR_ERROR_MESSAGE",
37         "Migration set to error
    after being idle for more than {} minutes"),
38     ),
39     connections_config=ConnectionsConfig(
40         mongodb_server_selection_timeout_ms=int(os.getenv("
    MONGODB_SERVER_SELECTION_TIMEOUT_MS", 2000)),
41     ),
42     logging_config=LoggingConfig(
43         log_level=getattr(logging, os.getenv("LOG_LEVEL", "
    INFO")), logging.INFO),
44         azure_cosmos_log_level=getattr(logging, os.getenv("
    COSMOS_LOG_LEVEL", "WARNING")), logging.WARNING),
45     )
46 )
```

Listing 4.47: The method load_config

4.1.12. Unit Test

In order to ensure the quality of the implementation, unit tests have been written to detect bugs and prove the functionality of the system.

Almost all of the tests need a PostgreSQL server to connect to, and in addition, depending on the migration source and target types, MongoDB, CouchDB, and/or Cosmos DB servers.

Table 4.1 shows the environment variables that control which database servers to use in the tests.

Environment Variable	Default
DB_HOST DB_PORT DB_NAME DB_USER DB_PASSWORD	localhost 5432 DBManagerConfig postgres postgres
TEST_MONGO_HOST TEST_MONGO_DB TEST_MONGO_USERNAME TEST_MONGO_PASSWORD	localhost:27017 factbase_testing
TEST_COUCH_HOST TEST_COUCH_USERNAME TEST_COUCH_PASSWORD UNIVIE_COUCH_PASSWORD	localhost:5984 admin admin No Default
TEST_COSMOS_HOST TEST_COSMOS_DB TEST_COSMOS_KEY	https://localhost:8081 factbase_testing C2y6yDjf5R+ob0N8A7Cgv30V RDJIWEHLM+4QDU5DE2nQ 9nDuVTqobD4b8mGGyPMbI ZnqyMsEcaGQy67XIw/Jw==

Table 4.1.: Environment variables used for testing

Test Utils

The file `test_utils.py` provides supporting methods that are used among multiple tests. It provides one assert method for migration processes, one method to create dummy migration objects (Listing 4.48), and methods that create connections to the configured databases and return connection managers (Listing 4.49).

4. Implementation

```
1 def get_dummy_migration(id, creator, collections, source_type=1,
2   target_type=1, source_id=-999, target_id=-999,
3   is_parallel=False) -> Migration:
4   """
5   Returns dummy migration instance.
6   Args:
7       id: Migration id
8       creator: Creator name
9       collections: List of collection names
10      source_type: Default: 1
11      target_type: Default: 1
12      source_id: Default: 1
13      target_id: Default: 1
14      is_parallel: Default: False
15  Returns:
16      (Migration): Migration instance.
17  """
18  return Migration(id=id, creator=creator, collections=
19  collections,
20  source_db_type=source_type, source_db_id=
21  source_id,
22  target_db_type=target_type, target_db_id=
23  target_id,
24  is_parallel=is_parallel, scheduled_for=
25  datetime.now(),
26  migrated_collections=[],
27  migrated_items_number=0, processed_items_number=0,
28  source_items_number=0)
```

Listing 4.48: The method `get_dummy_migration`

```
1 def get_local_couch_manager() -> CouchDBManager:
2   """
3   Returns CouchDBManager connected to a local CouchDB server
4   or as defined in environment variables.
5   Returns:
6       (CouchDBManager): CouchDBManager instance.
7   """
8   couch_connection_data = CouchdbConnection(name="local couch
9   server",
10   host=os.getenv("
11   TEST_COUCH_HOST", "localhost:5984"),
12   username=os.getenv
13   ("TEST_COUCH_USERNAME", "admin"),
14   password=os.getenv
15   ("TEST_COUCH_PASSWORD", "admin"))
16   return CouchDBManager(connection_data=couch_connection_data)
```

Listing 4.49: The method `get_local_couch_manager`

Listing 4.50 shows the method `assert_migration` that performs multiple checks to prove that the migration process from the source to the target database was successful. It compares the number of items in the source and the target connections, considering only the affected collections, and it checks that the migration object contains the correct values about the performed migration process.

```

1 def assert_migration(assertion_object: unittest.TestCase,
2   source_manager: AbstractDbManager,
3   target_manager: AbstractDbManager,
4   migration: Migration):
5     """
6     Assert the migration process by checking that the migration
7     object has been updated correctly with regard to the source
8     and target managers.
9     Args:
10        assertion_object (unittest.TestCase): object providing
11        assertion functions.
12        source_manager: Manager for source connection.
13        target_manager: Manager for target connection.
14        migration: The migration instance.
15    """
16    # connect
17    source_manager.connect()
18    target_manager.connect()
19
20    # assert
21    assertion_object.assertEqual(sorted(migration.collections),
22    sorted(migration.migrated_collections),
23    'Migrated collections do not
24    match')
25
26    source_items_count = source_manager.
27    get_number_of_items_in_collections_if_existing(migration.
28    collections)
29    target_items_count = target_manager.
30    get_number_of_items_in_collections_if_existing(migration.
31    collections)
32
33    assertion_object.assertEqual(source_items_count, migration.
34    migrated_items_number,
35    'Migrated items number do not
36    match number of items in source')
37    assertion_object.assertEqual(target_items_count, migration.
38    migrated_items_number,
39    'Migrated items number do not
40    match number of items in Target')
41    assertion_object.assertEqual(source_items_count, migration.
42    processed_items_number,
43    'Processed items number do not

```

4. Implementation

```
match number of items in source')
28     assertion_object.assertEqual(target_items_count, migration.
    processed_items_number,
29                                     'Processed items number do not
match number of items in Target')
30
31     for collection in migration.collections:
32         source_items_count = source_manager.
    get_collection_size_or_0(collection)
33         target_items_count = target_manager.
    get_collection_size_or_0(collection)
34         assertion_object.assertEqual(source_items_count,
    target_items_count,
35                                     f'collection {collection},
    items number in source and target does not match')
36
37     # close connection
38     source_manager.close_connection()
39     target_manager.close_connection()
```

Listing 4.50: The method `assert_migration`

Connection Managers

For each connection manager, a set of unit tests has been written to verify that the concrete implementations of the abstract methods from the class **AbstractDBManager** are behaving as expected (Listing 4.51 and Listing 4.52).

```

1 class CosmosDbManagerTest(unittest.TestCase):
2     """
3     Testing class for CosmosDBManager
4     """
5     urllib3.disable_warnings()
6
7     def test_connection_using_cosmos_manager(self):
8         """
9         Test establishing a connection to CosmosDB using
10        CosmosDBManager, and calling close_connection()
11        """
12        manager = get_local_cosmos_manager()
13        manager.close_connection()
14
15    def test_drop_collection(self):
16        """
17        Test dropping container
18        """
19        manager = get_local_cosmos_manager()
20        collection_name = 'drop_collection'
21        collection = manager.get_collection_or_create_new(
22        collection_name)
23        collection.upsert_item({'id': '1'})
24        self.assertEqual(1, manager.get_collection_size_or_0(
25        collection_name), 'data was not added')
26        manager.drop_collection(collection_name, None, True)
27        self.assertFalse(manager.has_collection(collection_name)
28        )
29
30    def test_get_connection_info(self):
31        """
32        Test getting connection info does not raise exception
33        """
34        manager = get_local_cosmos_manager()
35        info = manager.get_connection_info()
36        print(info)
37        manager.close_connection()

```

Listing 4.51: Snippet from CosmosDbManagerTest

4. Implementation

```
1 class MongoDBManagerTest(unittest.TestCase):
2     """
3     Testing class for MongoDBManager
4     """
5
6     def test_crud(self):
7         """
8         Test:
9         - establishing a connection to MongoDB using
10        MongoDBManager
11        - getting or creating a collection
12        - count items
13        - insert item
14        - read item
15        - update item
16        - drop collection
17        - closing connection
18        """
19        manager = get_local_mongo_manager()
20        db = manager.db
21
22        collection_name = 'test_python_client'
23        collection = db[collection_name]
24
25        count_before = collection.count_documents({})
26
27        print(collection)
28
29        doc1 = {'test': 'client1'}
30
31        result = collection.insert_one(doc1)
32        doc1['test2'] = 500
33
34        collection.replace_one({'_id': result.inserted_id}, doc1
35        )
36
37        self.assertEqual(doc1, collection.find_one({"_id":
38        result.inserted_id}))
39
40        self.assertEqual(1, collection.count_documents({}) -
41        count_before)
42
43        collection.drop()
44        manager.close_connection()
45
46    def test_create_collection_indices(self):
47        """
48        Test creating collection indices does not raise
```

```

exception.
    """
45     manager = get_local_mongo_manager()
46
47     collection_name = 'cache'
48
49     manager.get_collection_or_create_new(collection_name)
50     manager.create_collection_indices(collection_name, 10)
51     manager.close_connection()
52

```

Listing 4.52: Snippet from MongoDBManagerTest

Jobs

The unit test shown in Listing 4.53 is written for the **failing_migration_detector_job** to test whether the method can detect failing migrations and update them; at the same time, healthy migrations should not be affected.

```

1 class FailingMigrationDetectorTest(unittest.TestCase):
2     """
3     Testing class for failing_migration_detector_job
4     """
5
6     def test_failing_migrations_detector_job(self):
7         """
8         Test for job failing_migration_detector_job
9         Preconditions: connection to config db can be
10        established.
11        """
12        with app.app_context():
13            job_config = load_config().
14            failing_migrations_detector_job_config
15
16            running_migration = get_dummy_migration(-886, '
17            RunningMigrationDetector', ['test'])
18            running_migration.start_date = datetime.now()
19            running_migration.last_update = datetime.now()
20            running_migration.finish_date = None
21            running_migration.error = None
22
23            migration_repo.save(running_migration, False)
24
25            failing_migration = get_dummy_migration(-888, '
26            FailingMigrationDetector', ['test'])
27            failing_migration.start_date = datetime.now()
28            failing_migration.last_update = datetime.now() -
29            timedelta(minutes=job_config.error_after_in_minutes)
30            failing_migration.finish_date = None
31            failing_migration.error = None

```

4. Implementation

```
27         migration_repo.save(failing_migration, False)
28
29         # run the job
30         failing_migration_detector_job()
31
32         # running not affected
33         running_migration = migration_repo.get_by_id(
34             running_migration.id)
35         running_migration.finish_date = None
36         self.assertIsNone(running_migration.error, 'error
37         message is Not None!')
38         self.assertIsNone(running_migration.finish_date, '
39         finish_date is Not None!')
40
41         # failing detected
42         failing_migration = migration_repo.get_by_id(
43             failing_migration.id)
44         self.assertIsNotNone(failing_migration.error, 'error
45         message is None!')
46         self.assertIsNotNone(failing_migration.finish_date,
47                             'finish_date is None!')
```

Listing 4.53: The test class for the method `failing_migrations_detector_job`

Migrator

The tests of Migrator subclasses cover the total migration process. For each subclass of Migrator, there is a test class.

Listing 4.54 shows the test for **CosmosDbToMongoDbSeqMigrator**. The test starts with setting up a migration object, which includes the list of the collection to be migrated and whether the processing should be done in parallel or not. Then the creation of the source and target connection managers is done. After that, the migrator subclass is created, and the migration is executed. After execution, the result is asserted, and finally, a cleanup of the created migration object is done. The annotation `@log_execution` is used to log the test name and the execution time.

```

1 class TestCosmosDbToMongoDBSeqMigrator(unittest.TestCase):
2     """
3     Test CosmosDbToMongoDbSeqMigrator
4     """
5     migration_repo = MigrationRepo(config_db.session)
6
7     @log_execution
8     def test_parallel_migration(self):
9         """
10        Parallel collections processing test, cosmos to mongo.
11        Migrated collections: facts, cache
12        """
13        with (app.app_context()):
14            migration = get_dummy_migration(id=-999,
15                                           creator='PARALLEL
16        COLLECTIONS PROCESSING TEST, COSMOS TO MONGO',
17                                           collections=['facts',
18        , 'cache'], is_parallel=True)
19            self.migration_repo.save(migration)
20
21            source_cosmos_manager = get_local_cosmos_manager('
22        test_cosmos_source')
23            target_mongo_manager = get_local_mongo_manager('
24        test_mongo_target')
25
26            migrator = CosmosDbToMongoDbSeqMigrator()
27
28            migration = migrator.migrate(migration,
29            source_cosmos_manager, target_mongo_manager)
30            assert_migration(self, source_cosmos_manager,
31            target_mongo_manager, migration)
32            self.migration_repo.delete(migration)

```

Listing 4.54: The test class TestCosmosDbToMongoDBSeqMigrator

4.2. DBManager Frontend

The frontend application is implemented using Angular 20.3.9. The application has the following main components:

- Services
- Models
- Components
- Environments

4. Implementation

4.2.1. Services

The services implement the communication logic with the backend server. All three connection services extend the abstract class **ConnectionAPI**. This abstract class implements most of the logic, and it only delegates the definition of the API URL to its subclasses. Listing 4.55 shows the implementation of the methods for calling the endpoints of the backend application.

```
1 export abstract class ConnectionApi {
2
3   protected constructor(protected http: HttpClient) { }
4
5   abstract getApiUrl(): string;
6
7   getConnections<T extends Connection>(): Observable<T[]> {
8     const url = this.getApiUrl();
9     return this.http.get<any[]>(url);
10  }
11
12  createConnection<T extends Connection>(connection: T,
13    checkConnection: boolean): Observable<any> {
14    const url = `${this.getApiUrl()}?check_connection=${
15      checkConnection}`;
16    return this.http.post(url, connection);
17  }
18
19  updateConnection<T extends Connection>(connection: T,
20    checkConnection: boolean): Observable<any> {
21    const url = `${this.getApiUrl()}/${connection.id}?
22      check_connection=${checkConnection}`;
23    return this.http.put(url, connection);
24  }
25
26  deleteConnection(connectionId: number): Observable<any> {
27    const url = `${this.getApiUrl()}/${connectionId}`;
28    return this.http.delete(url);
29  }
30
31  getConnectionCollections(connectionId: number): Observable<
32    Collection[]> {
33    const url = `${this.getApiUrl()}/${connectionId}/collections
34      `;
35    return this.http.get<Collection[]>(url);
36  }
37
38  insertDummyData(connectionId: number, collectionName: string,
39    collectionSize: number): Observable<any> {
40    const url = `${this.getApiUrl()}/${connectionId}/collections
41      /${collectionName}/dummy-data?collection_size=${
```

```

        collectionSize}';
34     return this.http.post(url, null);
35 }
36
37 deleteConnectionCollection(connectionId: number,
    collectionName: string, drop: boolean): Observable<any> {
38     const url = `${this.getApiUrl()}/${connectionId}/collections
    /${collectionName}?drop=${drop}`;
39     return this.http.delete(url);
40 }
41 }

```

Listing 4.55: The abstract class ConnectionAPI

Listing 4.56 presents the implementation of the class **MongodbApi**, which only overrides the abstract method **getApiUrl**.

```

1 export class MongodbApi extends ConnectionApi {
2     constructor(http: HttpClient) { super(http);}
3     override getApiUrl(): string {
4         return environment.apiUrl + '/connections/mongodb';
5     }
6 }

```

Listing 4.56: The class MongodbApi extending ConnectionApi

Listing 4.57 shows the service to call the migration API. Date fields are received as a string, hence they are cast into Date objects in the method **convertStringFieldsToDate**.

```

1 export class MigrationApi {
2
3     constructor(private http: HttpClient) {
4     }
5
6     apiUrl = environment.apiUrl + '/migrations';
7
8     getMigration(migrationId: number): Observable<Migration> {
9         const url = `${this.apiUrl}/${migrationId}`;
10        return this.http.get<Migration>(url).pipe(
11            map(m => {
12                return this.convertStringFieldsToDate(m);
13            })
14        );
15    }
16
17    private convertStringFieldsToDate(migration: Migration):
        Migration {
18        migration.create_date = this.convertStringDateToDate(
            migration.create_date);
19        migration.scheduled_for = this.convertStringDateToDate(
            migration.scheduled_for);

```

4. Implementation

```
20     migration.start_date = this.convertStringDateToDate(  
    migration.start_date);  
21     migration.finish_date = this.convertStringDateToDate(  
    migration.finish_date);  
22     return migration;  
23 }  
24  
25 private convertStringDateToDate(field: Date): Date {  
26     return field ? new Date(field) : undefined  
27 }
```

Listing 4.57: Snippet from MigrationApi

4.2.2. Models

Interfaces are created to present the data transferred between the frontend and backend applications.

Listing 4.58 shows the interface **Connection**, which is a superclass of all connection types. It has an ID and a database type.

```
1 export interface Connection {  
2     id?: number;  
3     type?: DatabaseType;  
4  
5     [key: string]: any;  
6 }  
7  
8 export interface DatabaseType {  
9     id: number;  
10    name: string;  
11 }
```

Listing 4.58: The interfaces Connection and DatabaseType

The interface **MongodbConnection**, shown in Listing 4.59, extends Connection and defines its own fields.

```
1 export interface MongodbConnection extends Connection {  
2     name?: string;  
3     host?: string;  
4     db_name?: string;  
5     username?: string;  
6     password?: string;  
7 }
```

Listing 4.59: The interface MongodbConnection

The interface **MigrationRequest** (Listing 4.60) is the model used to send POST requests to create new migrations.

```

1 export interface MigrationRequest {
2   id?: number;
3   name?: string;
4   creator?: string;
5   source_db_type?: number;
6   source_db_id?: number;
7   target_db_type?: number;
8   target_db_id?: number;
9   collections?: string[];
10  is_parallel?: boolean;
11  scheduled_for?: Date;
12 }

```

Listing 4.60: The interface MigrationRequest

The interface **Migration** (Listing 4.61) is the model used to represent the body of responses of GET requests from the migration API.

```

1 export interface Migration {
2   id?: number;
3   name?: string;
4   creator?: string;
5   source_db_type?: DatabaseType;
6   source_db_id?: number;
7   source_db_name?: string;
8   target_db_type?: DatabaseType;
9   target_db_id?: number;
10  target_db_name?: string;
11  collections?: string[];
12  is_parallel?: boolean;
13  create_date?: Date;
14  scheduled_for?: Date;
15  start_date?: Date;
16  finish_date?: Date;
17  source_items_number?: number;
18  processed_items_number?: number;
19  migrated_items_number?: number;
20  migrated_collections?: string[];
21  status?: string;
22  is_blocked?: boolean;
23  error?: string;
24 }

```

Listing 4.61: The interface Migration

4. Implementation

The utils interface **Field** (Listing 4.62) is used to define table columns and form fields dynamically in the code, which helps implement reusable components. The aim of this generalization is to reuse the same UI components for all three connection types and also to facilitate future extensions.

```
1 export interface Field {  
2   header: string;  
3   objectField: string;  
4   required: boolean;  
5   type: FieldType;  
6   placeholder: string;  
7   readOnly: boolean;  
8 }  
9  
10 export enum FieldType {  
11   number = 'number',  
12   text = 'text',  
13   password = 'password',  
14   date = 'date',  
15   time = 'time',  
16   datetimeLocal = 'datetimeLocal',  
17   checkbox = 'checkbox',  
18   switch = 'switch',  
19 }
```

Listing 4.62: The interface Field

4.2.3. Components

Components represent the visual elements of the frontend application.

The utility component **ConnectionTable** (Listing 4.63) is defined to provide a tabular representation of stored connections, and the actions can be applied to them. The logic implemented in this component covers most of the operations that can be done to the connections in the system.

```
1 export class ConnectionTable implements OnInit {  
2  
3   title = input.required<string>();  
4   tableColumns = input.required<Field[]>();  
5   connectionApi = input.required<ConnectionApi>();
```

Listing 4.63: The interface ConnectionTable and its required input fields

Listing 4.64 shows the programmatic definition of the actions for connections in **ConnectionTable**.

```

1 createTableMenuItems(connection: Connection) {
2   this.tableMenuItems = [
3     {
4       label: 'Options',
5       items: [
6         {
7           label: 'details', icon: 'pi pi-external-link', command:
8             () => {
9               this.showConnectionCollectionsDialog(connection)
10            },
11          {
12            label: 'delete', icon: 'pi pi-trash', command: () => {
13              this.deleteConnectionAfterConfirm(connection.id)
14            },
15          {
16            label: 'edit', icon: 'pi pi-pencil', command: () => {
17              this.showConnectionUpsertDialog(connection)
18            }
19          }
20        ]
21      }
22    ]];
23 }

```

Listing 4.64: The programmatic definition of the actions for connections in ConnectionsTable

The use of **ConnectionAPI** is shown in Listing 4.65

```

1 loadConnectionsList(showSuccess: boolean = false) {
2   if(showSuccess) {
3     this.showMessage("start loading connections",'info')
4   }
5   this.connectionApi().getConnections().subscribe({
6     next: (result) => {
7       this.data = result
8       if (showSuccess) {
9         this.showMessage("refreshed successfully!")
10      }
11    },
12    error: (error) => {
13      this.showErrorErrorMessage(error.error.message);
14    }
15  })
16 }

```

Listing 4.65: The use of ConnectionAPI

4. Implementation

Listing 4.66 shows a dynamic table definition in **connection-table.html**, columns are read from **tableColumns** and rows from the list **data**.

```
1  <p-table [value]="data" dataKey="id" size="small" stripedRows
   [scrollable]="true" scrollHeight="flex"
2      [paginator]="true" [rows]="5" [rowsPerPageOptions]="
   [5, 10, 20, 50]" [loading]="lock">
3      <ng-template #header>
4          <tr>
5              @for (tc of tableColumns(); track tc) {
6                  <th>{{ tc.header }}</th>
7              }
8              <th>
9                  <p-button rounded raised icon="pi pi-refresh" severity
   ="secondary"
10                     (click)="loadConnectionsList(true)" title="
   refresh"/>
11              </th> <!--header:refresh, data:actions -->
12          </tr>
13      </ng-template>
14
15      <ng-template #body let-element>
16          <tr>
17              @for (tc of tableColumns(); track tc) {
18
19                  <td>
20                      @if (tc.type === FieldType.password) {
21                          <p-password disabled size="small" [feedback]="
   false" toggleMask [ngModel]="element[tc.objectField]"
22                             autocomplete="false"/>
23                      } @else {
24                          {{ element[tc.objectField] }}
25                      }
26                  </td>
27              }
28              <td>
29                  <p-menu #menu [model]="tableMenuItems" (onShow)="
   createTableMenuItems(element)" [popup]="true"
30                     appendTo="body"/>
31                  <p-button (click)="menu.show($event); $event.
   stopPropagation()" title="actions"
32                     icon="pi pi-ellipsis-v" severity="secondary"
   text rounded raised/>
33              </td>
34          </tr>
35      </ng-template>
36  </p-table>
```

Listing 4.66: The dynamic definition of the connections table in html

Listing 4.67 presents the component **CosmosdbConnections** using the component **ConnectionsTable**.

```

1 export class CosmosdbConnections {
2   tableTitle = 'Cosmos DB'
3
4   ///////////////////////////////////////////////////
5   tableColumns: Field[] = [
6     {header: 'ID', objectField: 'id', required: false, type:
7       FieldType.text, readOnly: true, placeholder: ""},
8     {
9       header: 'Name',
10      objectField: 'name',
11      required: true,
12      type: FieldType.text,
13      readOnly: false,
14      placeholder: 'Local CouchDB Connection'
15    },
16    {
17      header: 'Database',
18      objectField: 'db_name',
19      required: true,
20      type: FieldType.text,
21      readOnly: false,
22      placeholder: 'facts'
23    },
24    {
25      header: 'Host',
26      objectField: 'host',
27      required: true,
28      type: FieldType.text,
29      readOnly: false,
30      placeholder: 'https://localhost:8081'
31    },
32    {
33      header: 'Key',
34      objectField: 'key',
35      required: false,
36      type: FieldType.password,
37      readOnly: false,
38      placeholder: "key1das56sa1d6"
39    }
40  ];
41
42  constructor(public cosmosdbApi: CosmosdbApi) {
43  }
44 }

```

Listing 4.67: The component CosmosdbConnections

4. Implementation

The use of **connection-table** inside **cosmosdb-connections.html** is shown in Listing 4.68.

```
1 <app-connections-table
2   [title]="tableTitle"
3   [tableColumns]="tableColumns"
4   [connectionApi]="cosmosdbApi"
5 />
```

Listing 4.68: The use of connection-table in the cosmosdb-connections.html

4.2.4. Environments

Environments are the files used to configure the parameters for specific environments. Currently, there are only two environment files:

- **environments.ts**: used for development.
- **environments.prod.ts**: used to run in a production environment.

The only parameter that can be configured is the **apiBaseUrl**. As the application is supposed to run in a Docker Compose network, both files have the value **http://localhost:8080**. The value can be changed anytime when the backend server has to be running on a different host. Listing 4.69 shows the content of the file **environments.prod.ts**.

```
1 export const environment = {
2   production: true,
3   apiBaseUrl: 'http://localhost:8080',
4 };
```

Listing 4.69: The file environments.prod.ts

The file **environment.ts** is used in the code where a value from it is needed, for example, in **CosmosdbApi** as shown in Listing 4.70).

```
1 import {environment} from '../../../environments/environments';
2 ....
3 override getApiUrl(): string {
4   return environment.apiBaseUrl + '/connections/cosmosdb';
5 }
```

Listing 4.70: Snippet showing the use of environments

During the build of the application, based on the environment, the file **environments.ts** is replaced with the file of the desired environment. The replacement is configurable in the file **angular.json** as shown in Listing 4.71.

```

1 "configurations": {
2     "production": {
3         "fileReplacements": [
4             {
5                 "replace": "src/app/environments/environments.
6                 ts",
7                 "with": "src/app/environments/environments.
8                 prod.ts"
9             }
10        ],

```

Listing 4.71: The replacement configuration for production

4.3. Factserver

The FactServer backend is implemented in Python. The original implementation uses CouchDB as a storage solution. This section shows the changes made to enhance the FactServer so that it can connect to multiple database implementations, specifically MongoDB, Cosmos DB, and CouchDB. The support of usage for MongoDB and CouchDB is done only prototypically for the collection **database/facts** in **facts_controller.py**.

The changes can be categorized into:

- Databases Handlers
- Config DB
- Config Yaml

4.3.1. Databases Handlers

AbstractDBHandler

The method `close` was added to `AbstractDBHandler` to support the release of resources when needed. Currently, this is only needed for MongoDB connections, hence the method is optional and has an empty implementation.

In Addition, the methods `__enter__` and `__exit__` (Listing 4.72) are implemented to enable the creation of context managers ((with .. as) style), allowing closing the connection automatically at the end of the context, so there is no need for an extra call of the method `close`.

4. Implementation

```
1 def close(self, *args, **kwargs):
2     """
3     close the database connection and release resources, if
4     necessary.
5     """
6     pass
7
8 # support with ... as style
9 def __enter__(self):
10     return self
11
12 def __exit__(self, exc_type, exc_value, traceback):
13     self.close()
14     return False
```

Listing 4.72: The methods `close`, `__enter__`, and `__exit__`

MongoDBHandler & CouchDBHandler

Two new subclasses of the class `AbstractDBHandler` are implemented to support MongoDB and CosmosDB. In addition, the factory classes **MongoDBFactory** and **CosmosDBFactory** are implemented to support creating the new **DBHandlers**.

To save some space, the method documentation (Docstrings) is removed from some of the following Listings.

Listing 4.73 shows the constructor of class **MongoDBHandler**.

```
1 def __init__(self):
2     self._finalizer = weakref.finalize(self, self.close) #
3     ensure resources are released when object is destroyed
4     self.logger = logging.getLogger(__name__)
5     self.collection_name = None
6     self.server = None
7     self.db = None
8     self.collection = None
```

Listing 4.73: The constructor of class `MongoDBHandler`

The method `connect` in `MongoDBHandler` is shown in Listing 4.74.

```

1 def connect(self, *args, **kwargs):
2     if (not args and not kwargs) or (len(args) < 2 and '
collection_name' not in kwargs):
3         self.logger.error("Connection Parameter missing: '
collection_name' was not provided.")
4         raise DatabaseException("Connection Parameter missing: '
collection_name' was not provided.")
5
6     host = kwargs.get('host', args[0] if len(args) > 0 else '
localhost:27017')
7     user = kwargs.get('user', args[1] if len(args) > 1 else None
)
8     password = kwargs.get('password', args[2] if len(args) > 2
else None)
9     db_name = kwargs.get('db_name', args[3] if len(args) > 3
else None)
10    collection_name = kwargs.get('collection_name', args[4] if
len(args) > 4 else None)
11
12    connection_string = self.build_connection_string(host, user,
password, db_name)
13
14    self.logger.debug("Connecting to MongoDB server at %s, using
database: %s", host, db_name)
15    self.logger.debug("Trying to access collection: %s" %
collection_name)
16
17    try:
18        self.collection_name = collection_name
19        self.server = MongoClient(connection_string,
serverSelectionTimeoutMS=5000)
20        self.server.admin.command("ping")
21        self.db = self.server.get_database()
22        self.collection = self.db[collection_name]
23    except ServerSelectionTimeoutError as e:
24        self.logger.error("Could not connect to MongoDB server:
%s" % e)
25        raise DatabaseException("Could not connect to MongoDB
server.") from e
26    except Exception as e:
27        self.logger.error("An unexpected error occurred while
connecting to MongoDB server: %s" % e)
28        raise DatabaseException("An unexpected error occurred
while connecting to MongoDB server.") from e

```

Listing 4.74: The method `connect` in `MongoDBHandler`

4. Implementation

Listing 4.75 shows the implementation of the method `get_record` in `MongoDBHandler`.

```
1 def get_record(self, *args, **kwargs):
2     self._general_database_check()
3
4     _id = kwargs.get('_id', args[0] if args else None)
5
6     if not _id:
7         raise NoResultsFoundError("_id not provided")
8
9     doc = self.collection.find_one({"_id": _id})
10    if not doc:
11        raise NoResultsFoundError("No object found with id: %s"
12                                   % _id)
13    return doc
```

Listing 4.75: The method `get_record` in `MongoDBHandler`

Listing 4.76 shows the method `execute_query` for `MongoDBHandler`. Similar logic is implemented for `CosmosDBHandler`. In contrast to CouchDB, MongoDB queries are not stored as views in the database and are just called from the code. For MongoDB, each query can be written directly in the code. Hence, for each query implemented in CouchDB, a method is needed in MongoDB. To avoid changing a lot of code sections, the decision has been taken to keep using the method `execute_query` and implementing it as a gateway. Based on the name of the view, the method will call the respective method containing a MongoDB query and return its result. For the future, we can avoid this general method by calling the respective methods directly, as this style is only suitable for how CouchDB works.

```
1 def execute_query(self, *args, **kwargs):
2
3     self._general_database_check()
4
5     view_name = kwargs.get('view', args[0] if args else None)
6
7     if not view_name:
8         if "mango_query" in kwargs:
9             raise ValueError("No support for Mango queries for
10                               MongoDB. handler")
11             raise ValueError("View must be provided as an argument
12                               or keyword argument 'view'.")
13
14     # Arguments to be checked
15     key = kwargs.get('key', None)
16     start_key = kwargs.get('start_key', None)
17     end_key = kwargs.get('end_key', None)
```

```

17     # Optional arguments with default values
18     limit = kwargs.get('limit', 999999999)
19     skip = kwargs.get('skip', 0)
20     include_docs = kwargs.get('include_docs', True)
21     reduce = kwargs.get('reduce', False)
22     group = kwargs.get('group', False)
23
24     try:
25         # facts queries
26         if self.collection.name == 'facts':
27             match view_name:
28                 case 'name_type/nomid':
29                     name = key[0]
30                     type_ = key[1]
31                     return self.get_facts_by_name_and_type(name,
32 type_, limit, skip, include_docs)
33                 case 'basic/source':
34                     source = key
35                     return self.get_facts_by_source(source,
36 limit, skip, include_docs)
37                 case 'basic/mid':
38                     mid = key
39                     return self.get_facts_by_mid(mid, limit,
40 skip, include_docs)
41                 case 'basic/all':
42                     return self.get_all_facts(limit, skip,
43 include_docs)
44                 case 'source_name/mid':
45                     source = key[0]
46                     name = key[1]
47                     return self.get_facts_by_source_and_name(
48 source, name, limit, skip, include_docs)
49                 case 'source_mid/_id':
50                     source = key[0]
51                     mid = key[1]
52                     return self.get_facts_by_source_and_mid(
53 source, mid, limit, skip, include_docs)
54
55         # not known view
56         self.logger.error("View not found: %s" % view_name)
57         raise DatabaseException("View not found: {}", view_name)
58
59     except Exception as e:
60         self.logger.error("An unexpected error occurred: %s" % e
61 )
62
63         raise DatabaseException("An unexpected error occurred
64 while executing the query.") from e

```

Listing 4.76: The method `execute_query` in `MongoDBHandler`

4. Implementation

Listing 4.77 shows the method implemented to execute a query equivalent to the CouchDB view **name_type/nomid**. First of all, a check is executed to make sure the connection is open for the collection named **facts**. Then, a general method called **run_mongo_query** is called to execute a MongoDB query with a filter specified in the parameter.

```
1 # facts queries
2 def get_facts_by_name_and_type(self, name: str, type_: str,
   limit: int, skip: int, include_docs: bool):
3     """
4     for 'name_type/nomid'
5     Args:
6         name (str): the filter value for the field 'data.name'
7         type_ (str): the filter value for the field 'data.@type'
8         limit (int): the maximum number of results to return
9         skip (int): the number of results to skip
10        include_docs (bool): Whether to include the documents or
   only id in the results.
11    Returns:
12        A list containing the results of the query
13    """
14    self._check_collection_name('facts', self.
   get_facts_by_name_and_type.__name__)
15    return self.run_mongo_query({"data.name": name, "data.@type"
   : type_}, limit, skip, include_docs)
```

Listing 4.77: The method `get_facts_by_name_and_type` in `MongoDBHandler`

The method **run_mongo_query** (Listing 4.78) executes the query and returns a result that optionally can have the same structure as CouchDB query results. This way is not ideal, but it was done like this to avoid changing the existing code and to adhere to the project scope. Ideally, the existing code should be changed to be completely database-agnostic.

```

1 def run_mongo_query(self, filter, limit: int, skip: int,
2   include_docs: bool = True, couch_mode: bool = True):
3     """
4     Return the result of the query.
5     Args:
6         filter (dict): d key value dict to restrict result
7         limit (int): the maximum number of results to return
8         skip (int): the number of results to skip
9         include_docs (bool): Whether to include the documents or
10        only id in the results (default is True).
11        couch_mode (bool): Whether to return a list of (id, doc)
12        or directly the items themselves (default is True).
13        Allows using existing code (that expects couch-
14        structured result) without modification.
15    Returns:
16        A list containing the results of the query
17    """
18    if include_docs:
19        items = list(self.collection.find(filter).limit(limit).
20        skip(skip))
21    else:
22        items = list(self.collection.find(filter, ['_id']).limit
23        (limit).skip(skip))
24    if couch_mode:
25        if include_docs:
26            return CouchViewResult(rows=[Row(id=item['_id'], doc
27            =item) for item in items])
28        else:
29            return [{'id': item['_id']} for item in items]
30    return items

```

Listing 4.78: The method run_mongo_query in MongoDBHandler

Listing 4.79 shows the method `get_facts_by_name_and_type` in `CosmosDBHandler`.

```

1 def get_facts_by_name_and_type(self, name: str, type_: str,
2   limit: int, skip: int, include_docs: bool):
3     self._check_container_name('facts', self.
4     get_facts_by_name_and_type.__name__)
5     return self.run_cosmos_query(f"""c.data.name": {name}, c.
6     data.@type: {type_}""", limit, skip, include_docs)

```

Listing 4.79: The method get_facts_by_name_and_type in CosmosDBHandler

4. Implementation

The method `run_cosmos_query` is shown in Listing 4.80.

```
1 def run_cosmos_query(self, where_stmt: Optional[str], limit: int
  , skip: int, include_docs: bool = True,
2     couch_mode: bool = True):
3     query = f"""SELECT {'*' if include_docs else 'c._id'} FROM c
  {('where ' + where_stmt) if where_stmt else ''} OFFSET {skip}
  LIMIT {limit}"""
4     items = self.container.query_items(query=query,
  enable_cross_partition_query=True)
5
6     if couch_mode:
7         if include_docs:
8             return CouchViewResult(rows=[Row(id=item['_id'], doc
  =item) for item in items])
9         else:
10            return [{'id': item['_id']} for item in items]
11    return items
```

Listing 4.80: The method `run_cosmos_query` in `CosmosDBHandler`

Listing 4.81 shows the implementation of a factory class to create an instance of **MongoDBHandler**. For the mapping from `server_name` to the collection name, this factory is using the existing configurations for CouchDB collections.

The current implementation supports only one server to be used for all collections. When needed, this can very easily be changed to follow the existing implementation for CouchDB, which supports different servers for different collections. The class **CosmosDBFactory** is very similar to the factory class of MongoDB.

```

1 class MongoDBFactory:
2     """
3     Factory class to create MongoDBHandler instances.
4     """
5     @staticmethod
6     def create_database(server_name: str):
7         """
8         Create a MongoDBHandler instance based on the given
9         server name and the configured mongodb server
10        In this implementation, the server_name is used to look
11        up the container name
12        in the full_config dictionary.
13
14        Args:
15            server_name (str): The name of the server to connect
16            to.
17
18        Raises:
19            ValueError: If the server name is not found in the
20            configuration.
21
22        Returns:
23            MongoDBHandler: An instance of MongoDBHandler
24            connected to the specified database.
25        """
26        mongodb = 'mongodb'
27
28        if not server_name in full_config:
29            raise ValueError(f"Server name {server_name} not
30            found in configuration.")
31        if not mongodb in full_config:
32            raise ValueError(f"MongoDB config '{mongodb}' not
33            found in configuration.")
34
35        mongodb_config = full_config[mongodb]
36        server_name_config = full_config[server_name] # to get
37        the collection name from server name
38
39        database = MongoDBHandler()
40        database.connect(
41            host=mongodb_config['host'],
42            user=mongodb_config['user'],
43            password=mongodb_config['password'],
44            db_name=mongodb_config['db_name'],
45            collection_name=server_name_config['db'] # the name
46            of the collection
47        )
48        return database

```

Listing 4.81: The method create_database in MongoDBFactory

4. Implementation

ProxyDBHandler

The class **ProxyDBHandler** is a subclass of the class **AbstractDBHandler** and acts as a proxy handler to support a flexible database selection.

It delegates the calls of its methods to the object representing an implementation of the class **AbstractDBHandler**.

Listing 4.82 shows the constructor of the class **ProxyDBHandler**. The use of the fields is as follows:

- **handler**: representing the "real" DBHandler.
- **db_conn_data**: the object representing the connection data (Host, Password, ...) for the used database (MongoDB, CouchDB, Cosmos DB).
- **thread**: used to run a method to check the config database for any updates.
- **stop_event**: to control the thread.
- **collection_name**: the name of the collection the handler will connect to.
- **update**: whether the proxy should check periodically for changes in the config database.
- **interval**: the interval for update checks, only relevant when update is True.

```
1 def __init__(self):
2     """
3     Initialize a newly created instance of the ProxyDBHandler,
4     """
5     self.logger = logging.getLogger(__name__)
6     self._handler = None
7     self._db_conn_data = None
8     self._stop_event = threading.Event()
9     self._thread = None
10
11     self._collection_name = None
12     self._update = None
13     self._interval = None
```

Listing 4.82: The constructor of the class ProxyDBHandler

The method **connect** (Listing 4.83) is used to connect to a database collection. It calls the method **check_config_and_update_connection**, where the config database is queried to get the configured database connection, and connects to it. When update is True, the thread is created to run the method **run_update_in_interval** (Listing 4.84). This method calls the method **__check_config_and_update_connection** in a regular interval. Using update, a long-running connection can be updated during run time, without the need to shut down the application and start it again. As the current implementation of FactServer is following the serverless approach by using Azure Functions¹, the connection is established and closed for each request, therefore there is no need to set **update** to **True**. It also might cause an issue if the connection changes during a long-running request/query. Hence, the default value of **update** is **False**.

¹<https://azure.microsoft.com/en-us/products/functions>

```

1 def connect(self, *args, **kwargs):
2     """
3     Connect to a new (collection(MongoDB), database(CouchDB),
4     container(Cosmos DB)).
5     Establish the connection with the configured db
6     When update=True: start a thread that will periodically
7     check for updates on database connections based on the value
8     of update_interval.
9     If an update is detected, connect to the new database.
10    Args:
11        *args: Arguments for the connection, e.g. server address
12        , database name
13        collection_name (str): The name of the collection to
14        connect to
15        update (bool): whether change of configured db must
16        be done.(default: False)
17        update_interval (int): time between config db checks
18        for updates.(default 5)
19
20    **kwargs: Keyword arguments for the connection, e.g.
21    host, port, username, password, etc.
22        collection_name (str): The name of the collection to
23        connect to
24        update (bool): whether change of configured db must
25        be done.(default: True)
26        update_interval (int): time between config db checks
27        for updates.(default 5)
28    """
29
30    self._collection_name = kwargs.get('collection_name', args
[0] if len(args) >= 1 else None)
31    update = kwargs.get('update', args[1] if len(args) >= 2 else
False)
32    update_interval = kwargs.get('update_interval', args[2] if
len(args) >= 3 else 5)
33
34    self.close()
35
36    self._check_config_and_update_connection()
37
38    if update:
39        self._stop_event.clear()
40        self._thread = threading.Thread(target=self.
_run_update_in_interval, args=[update_interval], daemon=True)
41        self._thread.start()

```

Listing 4.83: The method connect in ProxyDBHandler

4. Implementation

```
1 def _run_update_in_interval(self, interval: 5):
2     """
3     method running in a thread to periodically check config db
4     for updates
5     Args:
6         interval (int): interval between checks in sec.(default
7         5)
8     """
9     while not self._stop_event.wait(1): # while thread was not
    stopped
        self._check_config_and_update_connection()
        time.sleep(interval)
```

Listing 4.84: The method `run_update_in_interval` in `ProxyDBHandler`

Listing 4.85 shows the method `_check_config_and_update_connection`, which first queries the config database to get the currently valid connection, and based on the database type, initiates the object handler with the corresponding `DBHandler`. The update on the used connection is logged to document the change of the connection.

```
1 def _check_config_and_update_connection(self):
2     """
3     Check config db for any update and connect to the configured
4     database
5     Raises:
6         DatabaseException: If a valid database connection was
7         not found in the config db.
8         DatabaseException: If the found database connection has
9         an unsupported type.
10        DatabaseException: for any unexpected exception.
11    """
12    with SessionLocal() as session:
13        fs_conn: FactserverConnection = session.scalar(select(
14            FactserverConnection
15        ).where((
16            FactserverConnection.valid_from <= datetime.now()),
17            (
18            FactserverConnection.valid_until == None) | (
19            FactserverConnection.valid_until > datetime.now()))))
20    if fs_conn is None:
21        raise DatabaseException("No valid database found")
22
23    connection_data = fs_conn.get_db()
24
25    if self._db_conn_data == connection_data:
26        return
27    action = 'updating' if self._db_conn_data else 'creating
28    ,
```

```

22     self.logger.info('start %s database connection to: %s (%
s)'),
23                     action, connection_data.type_obj.type,
connection_data.name)
24     try:
25         match connection_data.type:
26             # MongoDB
27             case ConnectionType.MONGO_DB.value:
28                 self._handler = MongoDBHandler()
29                 self._handler.connect(host=connection_data.
host, user=connection_data.username,
30                                     password=
connection_data.password, db_name=connection_data.db_name,
31                                     collection_name=self.
_collection_name)
32             # CouchDB
33             case ConnectionType.COUCH_DB.value:
34                 self._handler = CouchDBHandler()
35                 self._handler.connect(server=connection_data
.host, user=connection_data.username,
36                                     password=
connection_data.password, db_name=self._collection_name)
37             # CosmosDB
38             case ConnectionType.COSMOS_DB.value:
39                 self._handler = CosmosDBHandler()
40                 self._handler.connect(host=connection_data.
host, key=connection_data.key,
41                                     db_name=
connection_data.db_name, container_name=self._collection_name
)
42             case _:
43                 raise DatabaseException(
44                     f"Unimplemented database connection Type
: {connection_data.type_obj.type}")
45             except Exception as ex:
46                 self.logger.error(
47                     f"An error occurred while {action} the
connection to new database. Exception message: %s" % ex)
48                 raise DatabaseException("Error while connecting to
database") from ex
49
50     action = 'updated' if self._db_conn_data else 'created'
51     self._db_conn_data = connection_data
52     self.logger.info('Database connection %s successfully to
: %s (%s)',
53                     action, connection_data.type_obj.type,
connection_data.name)

```

Listing 4.85: The method check_config_and_update_connection in ProxyDBHandler

4. Implementation

Listing 4.86 shows the delegation of calls to the actual DBHandler in ProxyDBHandler.

```
1 def get_record(self, *args, **kwargs):
2     return self._handler.get_record(*args, **kwargs)
3
4 def save_record(self, *args, **kwargs):
5     return self._handler.save_record(*args, **kwargs)
6
7 def delete_record(self, *args, **kwargs):
8     return self._handler.delete_record(*args, **kwargs)
9
10 def execute_query(self, *args, **kwargs):
11     return self._handler.execute_query(*args, **kwargs)
12
13 def close(self, *args, **kwargs):
14     if self._thread and self._thread.is_alive():
15         self._stop_event.set()
16         self._thread.join()
17
18     if self._handler:
19         self._handler.close(*args, **kwargs)
```

Listing 4.86: The delegation of methods in ProxyDBHandler

Listing 4.87 shows the class **ProxyDBFactory**.

```

1 class ProxyDBFactory:
2     """
3     Factory class to create ProxyDBHandler instances.
4     """
5
6     @staticmethod
7     def create_database(server_name: str):
8         """
9         Create a ProxyDBHandler instance based on the given
10        server name
11        In this implementation, the server_name is used to look
12        up the (db name(CouchDB), collection name(MongoDB) or
13        container name(Cosmos DB))
14        in the full_config dictionary.
15        Args:
16            server_name (str): The name of the server to connect
17            to.
18        Raises:
19            ValueError: If the server name is not found in the
20            configuration.
21        Returns:
22            ProxyDBHandler: An instance of ProxyDBHandler
23            connected to the specified database.
24        """
25        proxy_db = 'proxyDB'
26        if not server_name in full_config:
27            raise ValueError(f"Server name {server_name} not
28            found in configuration.")
29        if not proxy_db in full_config:
30            raise ValueError(f"{'{proxy_db}'} not found in
31            configuration.")
32
33        server_name_config = full_config[server_name] # to get
34        the container name from server name
35        proxy_db_config = full_config[proxy_db]
36
37        collection_name = server_name_config['db']
38        update = proxy_db_config.get('update', False)
39        update_interval = proxy_db_config.get('updateInterval',
40        5)
41
42        database = ProxyDBHandler()
43        database.connect(collection_name=collection_name, update
44        =update, update_interval=update_interval)
45        return database

```

Listing 4.87: The factory for ProxyDBHandler

4. Implementation

DBFactory

The class `DBFactory`, shown in Listing 4.88, is a factory that creates a `DBHandler` based on the fields (`useProxyDB`, `defaultDB`) in the YAML config file, like `myconfig.yaml`. For more details, see Subsection 4.3.3.

```
1 class DBFactory:
2
3     @staticmethod
4     def create_database(server_name: str):
5         """
6         Create an instance of: ProxyDBHandler when "useProxyDB"
7         is True, else the configured db in "defaultDB".
8         When both values are missing it returns CouchDBHandler.
9         When defaultDB is unknown, it returns CouchDBHandler.
10        The instance is created based on the given server name
11        In this implementation, the server_name is used to look
12        up the (db name(CouchDB), collection name(MongoDB) or
13        container name(Cosmos DB))
14        in the full_config dictionary.
15        Args:
16            server_name (str): The name of the server.
17        Raises:
18            ValueError: If the server name is not in the config.
19        Returns:
20            handler: An instance of ProxyDBHandler/
21            CouchDBHandler/MongoDBHandler/CosmosDBHandler connected to
22            the specified database.
23        """
24        use_db_manager = full_config.get('useProxyDB', False)
25        default_db = full_config.get('defaultDB', 'couchdb')
26
27        if use_db_manager:
28            factory = ProxyDBFactory()
29        else:
30            match default_db:
31                case 'couchdb':
32                    factory = CouchDBFactory()
33                case 'mongodb':
34                    factory = MongoDBFactory()
35                case 'cosmosdb':
36                    factory = CosmosDBFactory()
37                case _:
38                    logger.warning(f"Unknown defaultDB '{
39default_db}', defaulting to 'couchdb'")
40                    factory = CosmosDBFactory()
41        return factory.create_database(server_name)
```

Listing 4.88: The class `DBFactory`

Figure 4.2 shows a screenshot of the changes made to use the **DBFactory** instead of **CouchDBFactory** for **facts_controller.py**.

try:	79	79	try:
factory = CouchDBFactory()	80	80	factory = DBFactory()
fact_db = factory.create_database("database")	81	81	with factory.create_database("database") as fact_db:
fmgr = FactManager(fact_db)	82	82	fmgr = FactManager(fact_db)
factset = fmgr.get_by_id(identifier)	83	83	factset = fmgr.get_by_id(identifier)

Figure 4.2.: The changes done to use DBFactory in facts_controller.py
(left: old version, right: new version)

4.3.2. Config DB

The file config_db.py implements the logic to connect to the configuration database (PostgreSQL).

Listing 4.89 shows the reading of credentials from the config and the initialization of a session object.

```

1 config = full_config['proxyDB']['configDB']
2
3 DATABASE_URL= f"postgresql://{config['username']}:{config['
    password']}'@{config['host']}:{config['port']}/{config['db']}"
4
5 engine = create_engine(
6     DATABASE_URL,
7     pool_pre_ping=True,
8     future=True,
9 )
10
11 class Base(DeclarativeBase):
12     pass
13
14 SessionLocal = sessionmaker(
15     bind=engine,
16     autoflush=False,
17     autocommit=False,
18 )

```

Listing 4.89: The file config_db.py

4. Implementation

Listing 4.90 shows the ORM class for MongoDB connections. The method `__eq__` is implemented to check whether a connection has been changed or not.

```
1 class MongoClient(Base):
2     __tablename__ = 'mongodb_connections'
3     id = mapped_column(Integer, primary_key=True)
4     type = mapped_column(Integer, ForeignKey('database_types.id'
5 ))
6     type_obj = relationship('DatabaseType', foreign_keys=[type])
7     name = mapped_column(String(200))
8     host = mapped_column(String(200))
9     db_name = mapped_column(String(200))
10    username = mapped_column(String(200))
11    password = mapped_column(String(200))
12
13    def __eq__(self, other):
14        if not isinstance(other, MongoClient):
15            return False
16        return (self.id == other.id and self.type == other.type
17                and self.host == other.host and self.db_name ==
18                other.db_name
19                and self.username == other.username and self.
20                password == other.password)
```

Listing 4.90: The class MongoClient

4.3.3. Config Yaml

The following configuration parameters are introduced into **default.yaml** (Listing 4.91) to support the features done in the scope of this project.

- **useProxyDB**: (True|False): when True, DBFactory will return ProxyDBHandler, else a handler based on the value of defaultDB, the default value is couchdb.
- **defaultDB**: mongodb (couchdb|mongodb|cosmosdb) use this database implementation in case useProxyDB is False.
- **proxyDB**: only needed when useProxyDB is True.
 - **configDB**: the credentials to connect to the configDB (Postgres).
 - **update**: whether the config database should be checked for updates.
 - **updateInterval**: time between update checks in sec.
- **mongodb**: the credentials for the MongoDB server (only needed when useProxyDB is False and defaultDB = mongodb).
- **cosmosdb**: the credentials for the Cosmos DB server (only needed when useProxyDB is False and defaultDB = cosmosdb).

When setting useProxyDB=False and defaultDB=couchdb, almost all the changes done in the scope of this project will be totally ignored.

```

1 #DBManger
2
3 useProxyDB: False # (True/False): when True, DBFactory will
   return ProxyDBHandler, else a handler based on the value of
   defaultDB, the default value is couchdb.
4
5 defaultDB: mongodb #(couchdb/mongodb/cosmosdb) use this database
   implementation in case useProxyDB is False
6
7
8 proxyDB: # only needed when useProxyDB is True.
9   configDB: # the credentials to connect to the configDB (
   Postgres).
10     host: 'localhost'
11     port: '5432'
12     db: '<DB_NAME>'
13     username: '<USERNAME>'
14     password: '<POSTGRES>'
15   update: False # whether the config database should be checked
   for updates
16   updateInterval: 5 # time between update checks in sec
17
18
19 mongodb: # the credentials for the MongoDB server. (only needed
   when useProxyDB is False and defaultDB = mongodb).
20   host: 'localhost:27017' #host and port
21   user: '<USER>'
22   password: '<PASSWORD>'
23   db_name: '<DB_NAME>'
24
25 cosmosdb: # the credentials for the Cosmos DB server. (only
   needed when useProxyDB is False and defaultDB = cosmosdb).
26   host: 'https://localhost:8081' ##protocol, host and port
27   key: '<key>'
28   db_name: '<DB_NAME>'

```

Listing 4.91: The introduced configs in default.yaml

4.4. Deployment

4.4.1. Containerization

All services (frontend, backend, and backend database) can be deployed using Docker by building and running the **docker-compose.yml** file.

Environment variables are set inside the file **.env**, which is used by the **docker-compose.yml**. For more details on the environment variables, see Paragraph **Config** in Subsection 3.2.1.

4. Implementation

Listing 4.92 shows the definition of:

- DBManager Frontend, port: 4200.
- DBManager Backend, port: 8080.
- DBManager Database (PostgreSQL), port: 5432.

env_file: defines the environment variables file to be used for the service. All services use the file **.env** (Listing 4.93).

```
1 version: '3.8'
2
3 services:
4
5   db-manager-frontend:
6     build: ./db-manager-frontend
7     image: db-manager-frontend
8     container_name: db-manager-frontend
9     depends_on:
10      - db-manager-backend
11     ports:
12      - "4200:4000"
13
14   db-manager-backend:
15     build: ./db-manager-backend
16     image: db-manager-backend:latest
17     container_name: db-manager-backend
18     depends_on:
19      - postgres
20     ports:
21      - "8080:8080"
22     extra_hosts:
23      - "localhost:host-gateway" # localhost requests are
24      forwarded to hosting machine
25     env_file:
26      - .env
27
28   postgres:
29     image: postgres:18.1-alpine
30     container_name: postgres
31     ports:
32      - "5432:5432"
33     volumes:
34      - postgres_data:/var/lib/postgresql/data
35     env_file:
36      - .env
```

Listing 4.92: Snippet from docker-compose.yml

For testing purposes, the Docker-Compose file has services for MongoDB, CouchDB, and Cosmos DB; these services are ignored by default. For more details, see Appendix A.1.

```

1 # mongodb, couchdb, no cosmosdb
2 #COMPOSE_PROFILES=mongodb, couchdb
3
4 # or mongodb, couchdb, cosmosdb:
5 #COMPOSE_PROFILES=mongodb, couchdb, cosmosdb
6
7 ##### db-manager-backend
8 #####
9 #db config
10 DB_HOST: localhost
11 DB_PORT: 5432
12 DB_NAME: DBManagerConfig
13 DB_USER: postgres
14 DB_PASSWORD: postgres
15 #migration config
16 MONGODB_BATCH_INSERT_SIZE: 100
17 MONGODB_CLEAN_COLLECTION_BY_DROP: True

```

Listing 4.93: Snippet from .env

Listing 4.94 illustrates the service MongoDB, which is started only when the profile **mongodb** is active.

```

1  mongodb:
2    image: mongo:8.2
3    container_name: mongodb
4    ports:
5      - "27017:27017"
6    volumes:
7      - mongodb_data:/data/db
8    env_file:
9      - .env
10   profiles: [ 'mongodb' ]

```

Listing 4.94: Snippet from docker-compose.yml: the service mongodb

4.4.2. Source Code & How to Run the System

The **README** of the **GitLab** repository of this project contains a complete guide with all the commands needed to run the system. The full README is provided in Appendix A.1 for reference.

5. Experimental Evaluation

This chapter introduces the system’s performance and user experience in detail.

5.1. Performance Analysis

Several tests were conducted to evaluate the system and database performance across a range of operations. This includes:

- Migration Performance.
- Insertion Performance.
- Query Performance.

5.1.1. Test Environment

The performance tests were conducted on a machine equipped with a 12th Gen Intel Core i7 processor (2.10 GHz), 32 GB RAM, and a 1 TB SSD, running 64-bit Windows 11. All databases involved in the evaluation were hosted locally as background services on the machine to ensure that all databases involved in the tests have similar computational resources. Furthermore, the delay introduced by network transfer is excluded, allowing for focusing on the performance of the implementation.

5. Experimental Evaluation

5.1.2. Migration Performance

The performance of the different migrators was evaluated by running migration processes on datasets of varying sizes and recording the runtime.

The migration processes ran under the following conditions:

- The runtime does not consider the time for dropping existing data or the creation of source data.
- The collections have no indices or views.
- For CouchDB to CouchDB migrations, the source database was running as a Docker service, and the target as a background service.

Table 5.1 shows the runtime in seconds for the migration of dummy data sets between the databases.

Migration\Size	1000	5000	10000	25000	50000
Cosmos DB → Cosmos DB	21	84	170	414	811
Cosmos DB → CouchDB	10	56	110	274	550
Cosmos DB → MongoDB	1	2	5	12	25
CouchDB → Cosmos DB	17	85	170	426	900
CouchDB → CouchDB	11	59	147	380	852
CouchDB → MongoDB	6	23	45	115	207
MongoDB → Cosmos DB	17	85	171	430	861
MongoDB → CouchDB	11	61	121	315	621
MongoDB → MongoDB	0	0	1	1	3
Average	10.44	50.56	104.44	263	536.67

Table 5.1.: Migration: performance comparison by dataset size

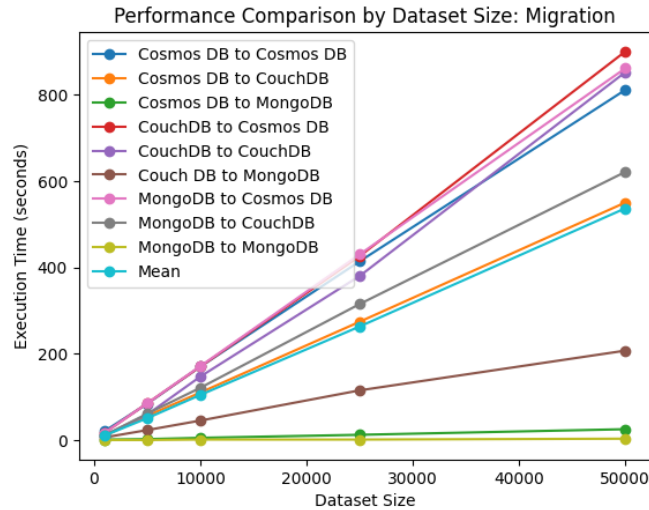


Figure 5.1.: Migration: performance comparison by dataset size

As shown in Table 5.1 and Figure 5.1, migrations into the **MongoDB** database were very fast. The speed of reading a stream from MongoDB and Cosmos DB is almost the same. This can be seen by checking the runtime of **CosmosDB** $\rightarrow$ **CouchDB** and **CosmosDB** $\rightarrow$ **CouchDB**. The runtime for reading from CouchDB is a bit higher. In most cases, the runtime increased linearly with the dataset size.

5.1.3. Insertion Performance

Table 5.2 and Figure 5.2 show the runtime in seconds for inserting dummy data sets into the databases.

Database\Size	1000	5000	10000	25000	50000	100000
MongoDB	0	0	0	0	1	1
CouchDB	10	51	99	260	518	1014
Cosmos DB	16	80	160	387	807	1620
Average	8.67	43.67	86.33	215.67	442	878.33

Table 5.2.: Insertion: performance comparison by dataset size

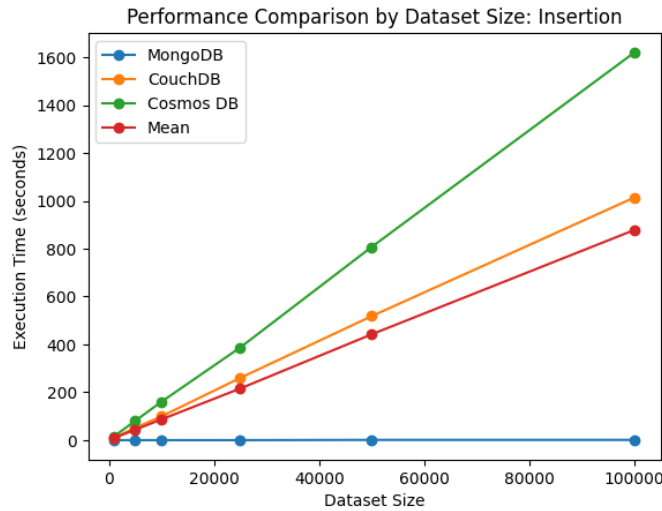


Figure 5.2.: Insertion: performance comparison by dataset size

In accordance with the runtime analysis of the migration processes, the insertion in **MongoDB** was the fastest compared to CouchDB and CosmosDB. The runtime increased approximately linearly with the dataset size.

5. *Experimental Evaluation*

5.1.4. Query Performance

The following is the query time on a **facts** dataset containing 25000 items. The filtering was done by the field **source**:

- **MongoDB (indexed)**: 0.003 sec → 333 items per second.
- **CouchDB (with view)**: 0.004 sec → 250 items per second.
- **Cosmos DB**: 0.06 sec → 17 items per second.

Again, **MongoDB** shows the best performance over CouchDB and Cosmos DB. Compared to CosmosDB, MongoDB was 20 times faster in this test.

5.2. User Experience

This chapter shows screenshots of the User Interface (UI) and discusses how users interact with the system and how intuitive and responsive it is.

The following UI design decision has been taken to enhance the user experience:

- The visual style and color scheme are uniform across all elements of the UI.
- Selected tabs are highlighted to help users easily identify their current location.
- Tables provide a refresh function to update data without requiring a full page reload.
- Tables are paginated to enhance the clarity of data presentation and minimize the need for scrolling.
- Users can select the table page size to customize the number of displayed rows.
- Passwords are masked by default and can be revealed if needed.
- Buttons use colors that reflect their intended function.
- Symbol buttons are clearly labeled to enhance user interaction.
- Critical operations prompt a confirmation dialog before they are executed to prevent errors.
- Operations that take significant time, such as API requests, are executed asynchronously to ensure the UI remains responsive.
- Feedback on operations, including both successful actions and errors, is provided through toast notifications to inform the user in real time.

5. Experimental Evaluation

5.2.1. Connections

The start page of the application is the table for MongoDB connections. The tab connections and sub-tab MongoDB are preselected (Figure 5.3).

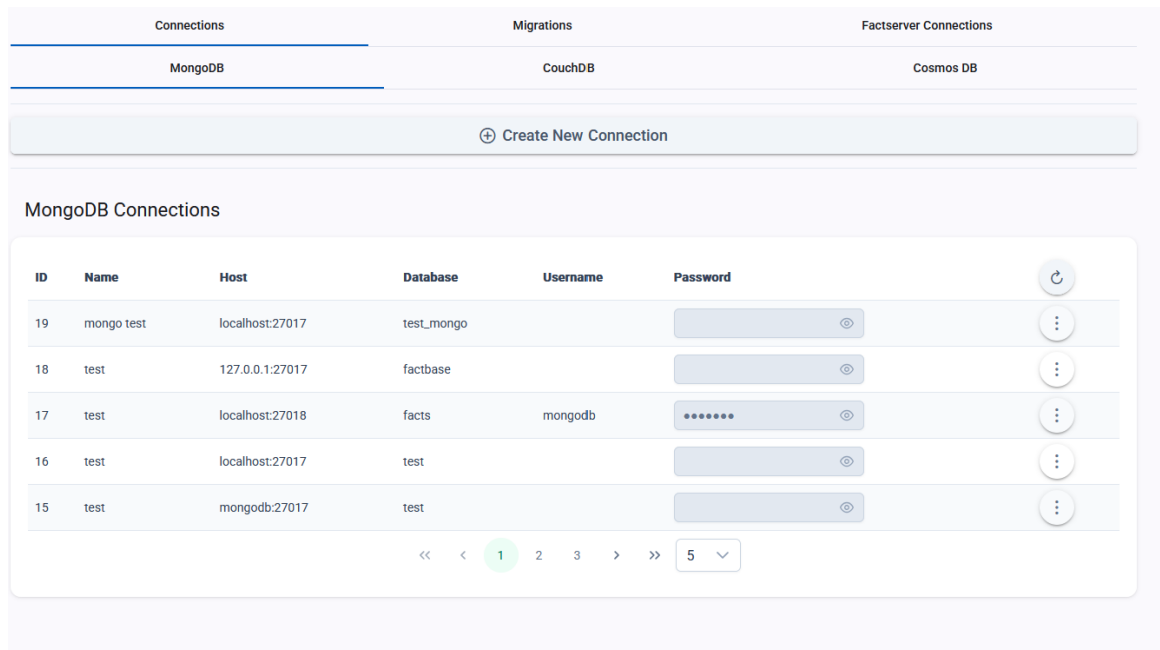


Figure 5.3.: The start page of the application: MongoDB connections

The options button (Figure 5.4) reveals three buttons hidden behind it:

- **details:** to show an overview of the collections and the stored data within this connection (Figure 5.5).
- **delete:** to delete the connection from the system (Figure 5.6).
- **edit:** to edit the connection data (host, password, ...) (Figure 5.9).

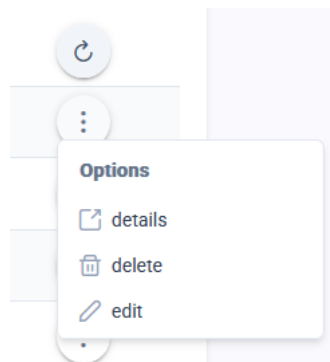


Figure 5.4.: The actions behind the options button for connections

Collections

Collection	Items Count	Items Size (Bytes)	Average Item Size (Bytes)	Empty	Drop
test15251	0	N/A	N/A		
cache	152	26321	173		
facts	25082	79488756	3169		

<<

<

1

>

>>

5

Add Test Data

Close

Figure 5.5.: The dialog for details

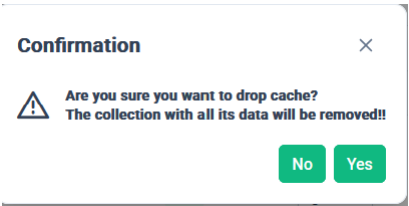


Figure 5.6.: The confirmation dialog when clicking on the drop button

5. Experimental Evaluation

When clicking on the button **Add Test Data**, a dialog will be opened (Figure 5.7). The collection for dummy data can either be selected from the existing collections, or a new one will be created. Before proceeding, the user is asked to confirm the action (Figure 5.8).

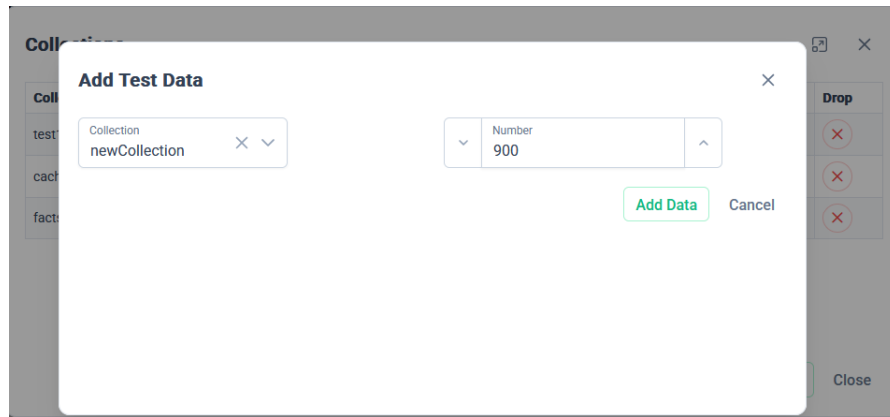


Figure 5.7.: The dialog for adding test/dummy data

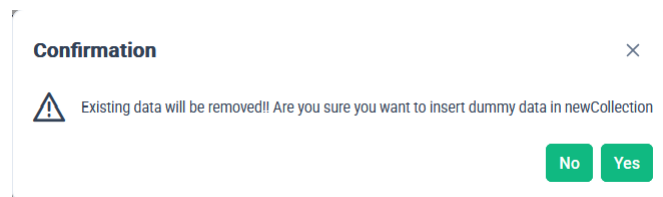


Figure 5.8.: The confirmation dialog before adding test/dummy data

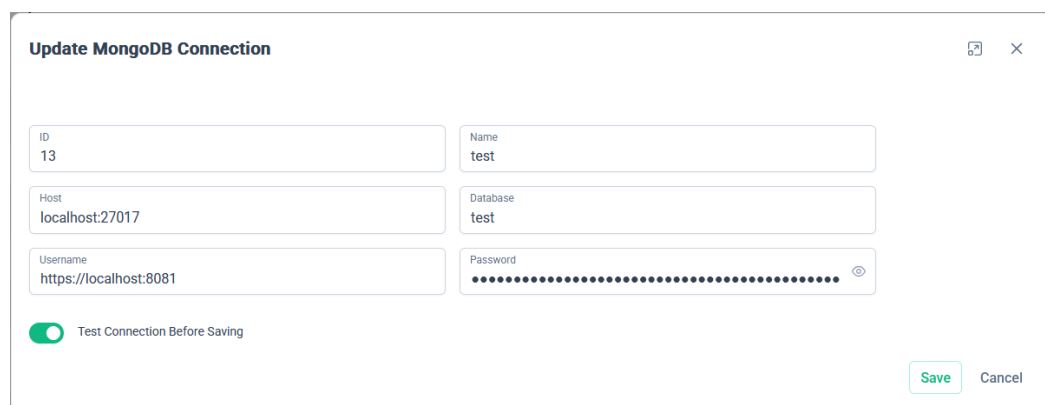


Figure 5.9.: The dialog To edit existing connection

The dialog for creating and updating a connection is the same (Figure 5.9). The field

ID is read-only. The switch **Test Connection Before Saving** is turned on by default. The user is notified about the result of the insertion or update as shown in Figure 5.10.

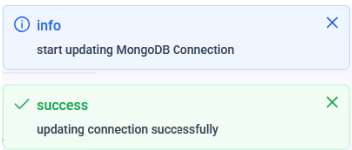


Figure 5.10.: The notifications shown when updating a connection

5.2.2. Migrations

Figure 5.11 shows the table of the migration processes.

Connections

Migrations

Factserver Connections

⊕ Create New Migration

Migrations

ID	Name	Creator	Source DB	Source Connection	Target DB	Target Connection	Collections	Create Date	Status	
999		MigrationManager SEQUENTIAL TEST Mongo TO Mongo	MongoDB	source server	MongoDB	target server	cache	14.1.2026, 10:16:05	failed	
888		FailingMigrationDetector	MongoDB		MongoDB		test	24.3.2026, 18:05:19	failed	
5	test migration couch to mongo	User123	CouchDB	test	MongoDB	factbase		15.1.2026, 10:01:29	failed	
4	test	Momo	MongoDB		Cosmos DB	Local COSMOS Factsbases	cache	6.12.2025, 23:38:45	finished	
3	TEST COSMOS TO COSMOS	Momo	Cosmos DB	Local COSMOS TEST SOURCE	Cosmos DB	Local COSMOS Factsbases	cache	6.12.2025, 23:27:40	finished	

<< < 1 2 > >>

5 ▾

Figure 5.11.: The table of migrations

5. Experimental Evaluation

The options button (Figure 5.12) reveals three buttons hidden behind it:

- details: to show the details of the migration process (Figure 5.13 and Figure 5.14).
- delete: to delete the migration from the system.
- edit: to edit the migration process, it will be deactivated once the migration has been started.

Migrations										
ID	Name	Creator	Source DB	Source Connection	Target DB	Target Connection	Collections	Create Date	Status	
8	GUI TEST	Mohamad	MongoDB	test	MongoDB	Target server	test123	27.3.2026, 16:44:31	scheduled	
7	test GUI	Mohamad	MongoDB	source server	MongoDB	Target server	facts,cache	27.3.2026, 16:13:09	finished	
6	GUI TEST	Mohamad	MongoDB	source server	MongoDB	Target server	facts	27.3.2026, 15:22:29	failed	
5	test migration couch to mongo	User123	CouchDB	test	MongoDB	factbase		15.1.2026, 10:01:29	failed	
4	test	Momo	MongoDB		Cosmos DB	Local COSMOS Factsbases	cache	6.12.2025, 23:38:45	finished	

Figure 5.12.: The actions behind the options button for migrations

Migration Details

ID3

Statusfinished

NameTEST COSMOS TO COSMOS

CreatorMomo

Create Date6.12.2025, 23:27:40

Scheduled For6.12.2025, 22:27:00

Start Date6.12.2025, 23:27:42

Finish Date6.12.2025, 23:27:49

Duration00:00:07

Source DB TypeCosmos DB

Source DB NameLocal COSMOS TEST SOUR

Target DB TypeCosmos DB

Target DB NameLocal COSMOS Factsbases

Source Items Number152

Processed Items Number152

Migrated Items Number152

Parallel Processingtrue

Collections

cache

Migrated Collections

cache

Progress

Processed100%

Migrated100%

refresh

Close

Figure 5.13.: The dialog showing the details of a successful migration process

Migration Details

ID5

Statusfailed

Name
test migration couch to mongo

Creator
User123

Error
'<=' not supported between instances of 'str' and 'int'

Create Date
15.1.2026, 10:01:29

Scheduled For
15.1.2026, 10:01:29

Start Date
26.3.2026, 20:19:17

Finish Date
26.3.2026, 20:19:17

Duration
00:00:00

Source DB Type
CouchDB

Source DB Name
test

Target DB Type
MongoDB

Target DB Name
factbase

Source Items Number
0

Processed Items Number
0

Migrated Items Number
0

Parallel Processing
true

Collections

Migrated Collections

Progress

Processed

Migrated

refresh

Close

Figure 5.14.: The dialog showing the details of an erroneous migration process

5. Experimental Evaluation

The details dialog of a failed migration includes the text field **Error** highlighted in red, as shown in Figure 5.14.

The color of the progress bar changes from red to green based on the percentage of progress, as illustrated in the figures 5.15, 5.16, 5.17, 5.18, and 5.19.

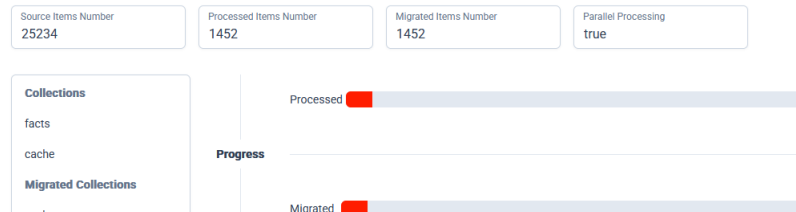


Figure 5.15.: The progress bar: 7%

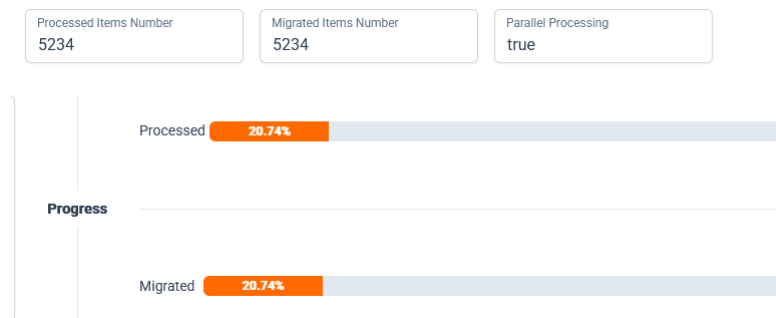


Figure 5.16.: The progress bar: 20%

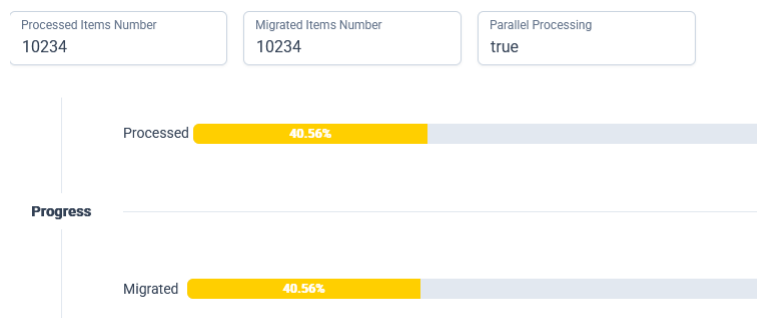


Figure 5.17.: The progress bar: 40%

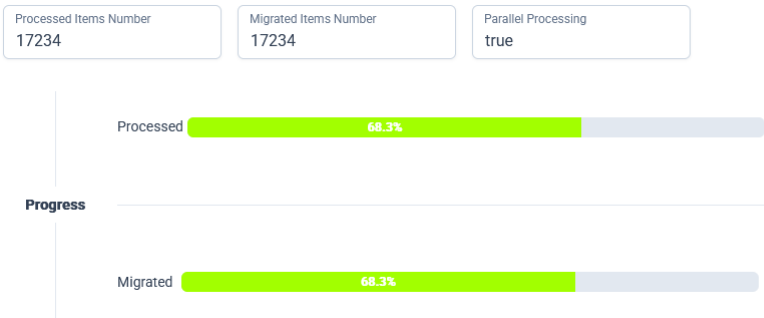


Figure 5.18.: The progress bar: 68%

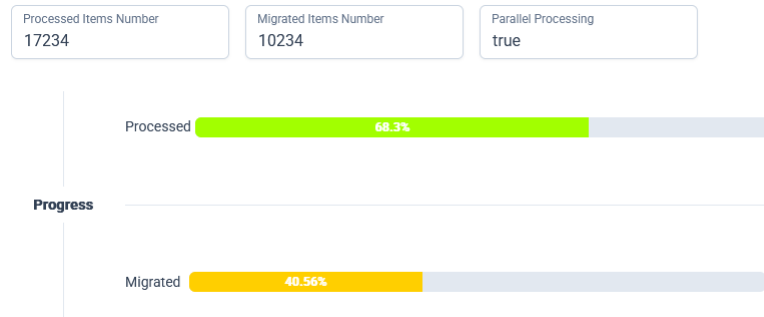


Figure 5.19.: The progress bar: not all items are migrated

Figure 5.20 shows the dialog for creating a new migration process. Once a source or target type is selected, the list of connections of that type is loaded. Once the source connection is selected, the list of collections is loaded. The user can select which collections have to be migrated. In addition, the start date of the migration process can be selected (Figure 5.21). The switch **Parallel Processing of Collections** is used to enable the parallel processing of collections. Before saving, checks will be done to ensure that the source and target connections are not the same and that at least one collection is selected, as shown in Figure 5.22.

5. Experimental Evaluation

Create New Migration

ID:

Name:

Creator:

Source Database Type:

Source Database Connection:

Target Database Type:

Target Database Connection:

Search by name:

Search by name:

Scheduled For:

☒ Parallel Processing of Collections

Figure 5.20.: The dialog for creating a new migration process

Create New Migration

ID:

Name:

Creator:

Source Database Type:

Source Database Connection:

Target Database Type:

Target Database Connection:

Search by name:

Search by name:

Scheduled For:

☒ Parallel Processing of Collections

Figure 5.21.: The selection of the processing date of a migration process

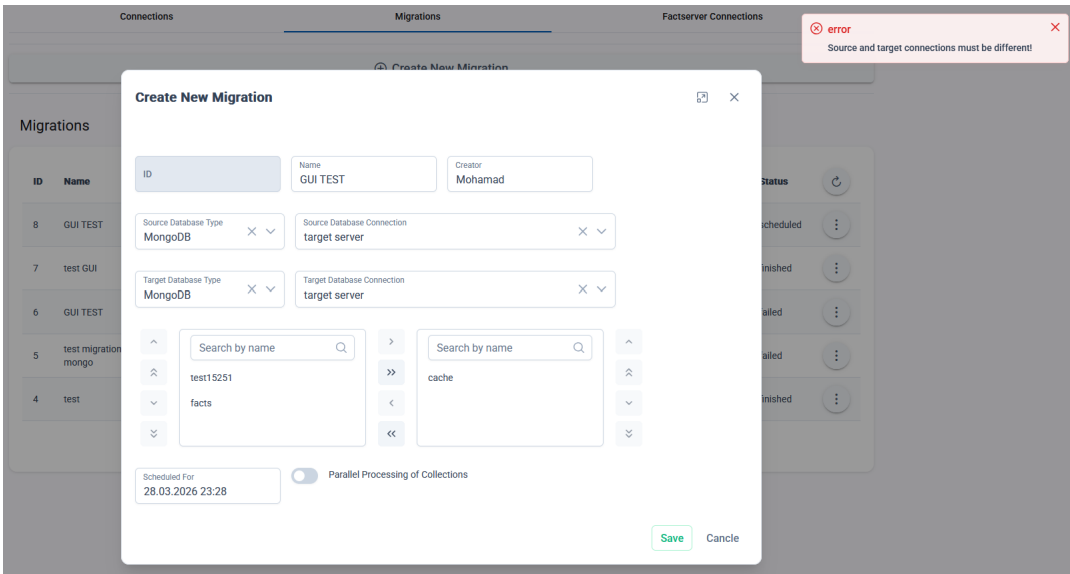


Figure 5.22.: The check for different source and target

5.2.3. Factserver Connections

Figure 5.23 shows the table of the active factserver connections. The switch **only active** filters the table to show only the currently active connection. When the switch is turned off, the table shows the history of active connections, including the current one (Figure 5.24).

Connections

Migrations

Factserver Connections

⊕ Create New Factserver Connection

Factserver Connections

ID	Creator	Description	Database Type	Connection Name	Create Date	Valid From	Valid Until		
8	User123	test couchdb 1 factserver connection	CouchDB	test	15.1.2026, 10:01:26	15.1.2026, 10:01:26			

<< < 1 > >>

5 ▾

Figure 5.23.: The table for active factserver connections

5. Experimental Evaluation

Connections

Migrations

Factserver Connections

Create New Factserver Connection

Factserver Connections

ID	Creator	Description	Database Type	Connection Name	Create Date	Valid From	Valid Until		
8	User123	test couchdb 1 factserver connection	CouchDB	test	15.1.2026, 10:01:26	15.1.2026, 10:01:26			
4	test	momo	MongoDB	Target server	28.12.2025, 14:05:38	28.12.2025, 14:05:38	27.3.2026, 16:52:31		
2	test	123	Cosmos DB	Local COSMOS TEST TARGET	6.12.2025, 22:42:13	6.12.2025, 22:42:13	28.12.2025, 14:05:38		
1	Momo	test	Cosmos DB	Local COSMOS TEST SOURCE	6.12.2025, 22:41:55	6.12.2025, 22:41:55	6.12.2025, 22:42:13		

<<

<

1

>

>>

5

>

Figure 5.24.: The table for factserver connections, including history

The button **Create New Factserver Connection** opens a dialog (Figure 5.25) where the user can select the type of connection and the connection. Before saving, the user will see a confirmation dialog (Figure 5.26).

Create New Factserver Connection

Creator
Mohamad

Description
Move to MongoDB

Database Type
MongoDB

Database Connection
source server

Save

Cancel

Figure 5.25.: The dialog for creating a new factserver connection

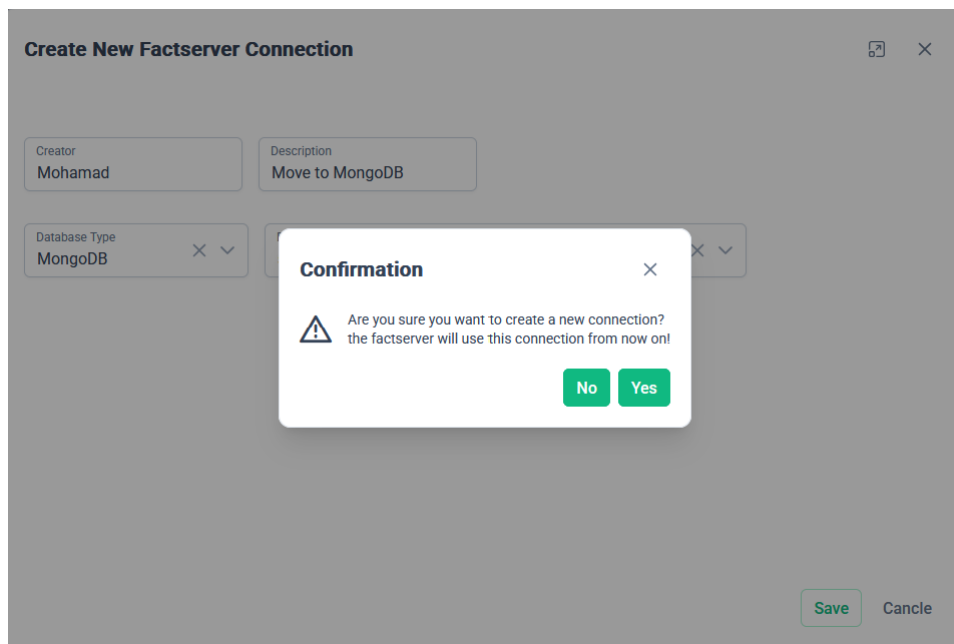


Figure 5.26.: The confirmation dialog before saving a new factserver connection

6. Conclusion

6.1. Summary

In this master's thesis, we analyzed storage solutions for the FactChecking prototype. The major focus of the analysis was to answer the research questions about the fitting data type and suitable data storage solutions that can be used for FC. Based on the analysis result, we designed and implemented the database framework called DBManager, allowing the management of the connection credentials of multiple database types and migrating the data between the managed connections. The database types MongoDB and Cosmos DB were integrated prototypically into the FactServer, supporting flexible access to multiple database implementations. Chapter 1 introduced the problem and the motivation by highlighting the need for a suitable data storage solution in FC, and presented the main research questions of this master's thesis. In Chapter 2, the FC project was briefly introduced, in Section 2.1, by describing its history, core functionalities, and architecture. In Section 2.2, the well-established database implementations were analyzed with a focus on their key characteristics, such as performance, security of the stored data, and available cloud support. Chapter 3 introduced the functional and non-functional requirements of the system, outlined the system design, and discussed the design decisions. The implementation details of each part of the system have been introduced and discussed in detail in Chapter 4. We evaluated the system's performance and user experience and presented the results in Chapter 5. Finally, Chapter 6 listed the limitations of this work and outlined directions for future research.

6.2. Limitations

This project is subject to the following limitations:

- Support is limited to three database types (MongoDB, CouchDB, and Cosmos DB).
- The current implementation supports only basic connection configurations using username and password credentials. Advanced security features, including SSL/TLS encryption, certificate-based authentication, and other enhanced authentication methods, are not supported. This applies to the configuration database (PostgreSQL) as well as to the supported database types for migrations.
- The system does not support user authentication, hence no role-based access rights in the user interface.
- No possibility to see a sample of the stored data in a database connection.
- No possibility to restart a failed migration process.
- Performance tests were conducted on a local machine.

6. Conclusion

- Databases used to run the performance tests are running on a local machine with limited resources. No real production setup.
- Factserver support for MongoDB and Cosmos DB is only prototypical. Not all queries are implemented.
- Only one supported language in the user interface: English.
- No support for a dark mode in the user interface.

6.3. Future Work

In future work, the identified limitations can be addressed to increase the level of security and the user experience (UX). The focus will be on providing full support for MongoDB in factserver. Furthermore, we intend to integrate a wider range of database types and implement advanced authentication mechanisms, such as SSL.

The performance of the migration logic can be further investigated and enhanced through the implementation of parallel access on the same datasets, where feasible. Error handling in the system can be improved by incorporating mechanisms that suggest potential root causes along with possible solutions.

Artificial intelligence (AI) can be integrated to implement a chatbot that can start migration processes and perform supported operations. Moreover, the chatbot could offer decision support by explaining the key advantages and disadvantages of each supported database, for example, in terms of expected performance and use-case suitability.

Bibliography

- [1] JsonLd. JSON for Linking Data. <https://json-ld.org/>. Last accessed 17 November 2024.
- [2] IBM. SQL vs. NoSQL Databases: What's the difference. <https://www.ibm.com/think/topics/sql-vs-nosql>. Last accessed 24 November 2024.
- [3] Mara Sophie Aichinger. Towards a cloud-native framework for resolving issues in (semi-)structured data with human feedback in the context of FactCheck++, 2023. Last accessed 31 January 2026.
- [4] Scribbr. Type I & Type II Errors | Differences, Examples, Visualizations. <https://www.scribbr.com/statistics/type-i-and-type-ii-errors/>. Last accessed 25 November 2024.
- [5] MongoDB. Press Kit, MongoDB. <https://brandfolder.com/mongodb/press-kit>. Last accessed 14 April 2026.
- [6] MongoDB. Replication. <https://www.mongodb.com/docs/manual/replication/>. Last accessed 04 January 2025.
- [7] MongoDB. Sharding in MongoDB. <https://www.mongodb.com/resources/products/capabilities/sharding>. Last accessed 04 January 2025.
- [8] MongoDB. JSON And BSON. <https://www.mongodb.com/resources/basics/json-and-bson>. Last accessed 01 January 2025.
- [9] MongoDB. MongoDB Compass: The free GUI tool for data exploration. <https://www.mongodb.com/products/tools/compass>. Last accessed 18 April 2026.
- [10] MongoDB. MongoDB Relational Migrator. <https://www.mongodb.com/products/tools/relational-migrator>. Last accessed 18 April 2026.
- [11] MongoDB. Mongosync. <https://www.mongodb.com/try/download/mongosync>. Last accessed 18 April 2026.
- [12] MongoDB. Atlas Database. <https://www.mongodb.com/products/platform/atlas-database>. Last accessed 18 April 2026.
- [13] MongoDB. MongoDB Pricing. <https://www.mongodb.com/pricing>. Last accessed 19 April 2026.

Bibliography

- [14] Azure. Azure Icons. <https://az-icons.com/icon/azure-cosmos-db>. Last accessed 14 April 2026.
- [15] Microsoft Azure. Integrated Vector Database - Azure Cosmos DB. <https://learn.microsoft.com/en-us/azure/cosmos-db/vector-database>. Last accessed 03 May 2026.
- [16] Microsoft Azure. Azure Cosmos DB - Database for the AI Era. <https://learn.microsoft.com/en-us/azure/cosmos-db/introduction>, 2024. Last accessed 05 January 2025.
- [17] Microsoft Azure. Choose an API in Azure Cosmos DB. <https://learn.microsoft.com/en-us/azure/cosmos-db/choose-api>, 2024. Last accessed 05 January 2025.
- [18] Microsoft Azure. Azure Cosmos DB pricing. <https://azure.microsoft.com/en-us/pricing/details/cosmos-db/standard-provisioned/>. Last accessed 22 April 2026.
- [19] CouchDB. CouchDB-visual-identity. <https://github.com/apache/couchdb-www/blob/asf-site/CouchDB-visual-identity/logo/CouchDB-vertical-logo.png>. Last accessed 18 April 2026.
- [20] MongoDB. CouchDB and MongoDB Compared. <https://www.mongodb.com/resources/compare/documentdb-vs-mongodb/couchdb-vs-mongodb>, 2024. Last accessed 09 January 2025.
- [21] Couch DB. API Basics. <https://docs.couchdb.org/en/stable/api/basics.html>, 2024. Last accessed 09 January 2025.
- [22] Redis. Logos, Redis Brand Guidelines. <https://brand.redis.io/document/18>. Last accessed 23 April 2026.
- [23] IBM. What is Redis. <https://www.ibm.com/think/topics/redis>. Last accessed 25 April 2026.
- [24] Redis. Redis Data Types. <https://redis.io/docs/latest/develop/data-types/>. Last accessed 25 April 2026.
- [25] Redis. Redis Search. <https://redis.io/docs/latest/develop/ai/search-and-query/>. Last accessed 25 April 2026.
- [26] Redis. Redis persistence. https://redis.io/docs/latest/operate/oss_and_stack/management/persistence/. Last accessed 23 April 2026.
- [27] Redis. Redis products. <https://redis.io/docs/latest/operate>. Last accessed 25 April 2026.
- [28] Neo4j. Neo4j Brand Portal. <https://neo4j.bynder.com/share/9CCA5D86-122D-4D8F-8FF824C306847BC3/>. Last accessed 26 April 2026.

- [29] Neo4j. What is Neo4j. <https://neo4j.com/docs/getting-started/whats-neo4j/>. Last accessed 26 April 2026.
- [30] Neo4j. What is a graph database. <https://neo4j.com/docs/getting-started/graph-database/>. Last accessed 26 April 2026.
- [31] Neo4j. Aura with cloud provider marketplaces. <https://neo4j.com/docs/aura/cloud-providers/>. Last accessed 26 April 2026.
- [32] Neo4j. Instance actions. https://neo4j.com/docs/aura/managing-instances/instance-actions/#_adjust_storage. Last accessed 26 April 2026.
- [33] PostgreSQL. PostgreSQL logo. https://wiki.postgresql.org/images/9/9a/PostgreSQL_logo.3colors.540x557.png. Last accessed 18 April 2026.
- [34] PostgreSQL. PostgreSQL: The World’s Most Advanced Open Source Relational Database. <https://www.postgresql.org/>, 2024. Last accessed 15 February 2025.
- [35] PostgreSQL. PostgreSQL: 5.5. Constraints. <https://www.postgresql.org/docs/current/ddl-constraints.html>. Last accessed 26 April 2026.
- [36] PostgreSQL. PostgreSQL: Overview of Trigger Behavior. <https://www.postgresql.org/docs/8.2/trigger-definition.html>. Last accessed 27 April 2026.
- [37] PostgreSQL. PostgreSQL: User-Defined Procedures. <https://www.postgresql.org/docs/current/xproc.html>. Last accessed 27 April 2026.
- [38] PostgreSQL. PostgreSQL: Indexes. <https://www.postgresql.org/docs/current/indexes.html>. Last accessed 27 April 2026.
- [39] PostgreSQL. PostgreSQL: Index Types. <https://www.postgresql.org/docs/current/indexes-types.html>. Last accessed 29 April 2026.
- [40] PostgreSQL. PostgreSQL: Table Partitioning. <https://www.postgresql.org/docs/18/ddl-partitioning.html>. Last accessed 27 April 2026.
- [41] PostgreSQL. PostgreSQL: XML Functions. <https://www.postgresql.org/docs/9.1/functions-xml.html>. Last accessed 27 April 2026.
- [42] PostgreSQL. PostgreSQL: jsonpath Type. <https://www.postgresql.org/docs/18/datatype-json.html#DATATYPE-JSONPATH>. Last accessed 27 April 2026.
- [43] Zhenan Tu. Research on the Application of Layered Architecture in Computer Software Development. *Journal of Computing and Electronic Information Management*, 11:34–38, 11 2023.

Acronyms

AWS Amazon Web Services. 10, 16, 18

FC FactChecking. 1–3, 5–8, 151

GCP Google Cloud Provider. 10, 16, 18

Glossary

- CORS** Cross-Origin Resource Sharing, a security mechanism used by the browser. 59
- GIN** Generalized Inverted Index, a type of index for composite values. 17
- JSON** JavaScript Object Notation, a Lightweight data format. 2, 3, 5, 74
- NoSQL** Non-relational database systems. 2, 3, 5
- ORM** Object-Relational Mapping, a technique that converts data between object-oriented languages and relational databases, for more details, see: <https://aws.amazon.com/what-is/object-relational-mapping/>. 25, 61, 63–65, 128
- SQL** Structured Query Language, a Language for relational databases. 2, 3
- XML** Extensible Markup Language, a human-readable data format. 2

A. Appendix

A.1. GIT README

A.1.1. Master Thesis Title

Analysis of Cloud-based storage systems/services and design of a storage framework for a FactChecking prototype.

A.1.2. Project Description

The prototypical implementation of the framework (called DBManager) is a dashboard allowing for managing different connections of different database implementations and supporting the migration of data between them.

Supported database implementations are:

- MongoDB
- CouchDB
- Azure Cosmos DB

Additionally, the framework allows selecting the database connection to be used by the factserver.

A.1.3. Project Components

- **code**: This folder contains the implementation part of this project.
- **code/db-manager-frontend**: This folder contains the frontend application implemented with the framework Angular 20.3.9 .
- **code/db-manager-backend**: This folder contains the backend application implemented in Python 3.12.
- **code/docker-compose.yml**: Docker-compose file to start the applications and databases. See Subsection A.1.5 below.
- **code/.env**: File to define environment variables to be used in the Docker containers.
- **code/database**: This folder contains helper scripts for the database.
- **seminar**: This folder contains documents related to the master seminar.
- **thesis** This folder contains a written master thesis in PDF format and as LaTeX, it also contains all figures (screenshots, diagrams, etc.) used in the thesis.
- **presentation** This folder contains the slides for the thesis presentation (.pdf and .odp)

A. Appendix

A.1.4. Use Cases

- The user can add a new database connection by selecting the desired database implementation supported by the system, and providing the required connection information like URL, username, password, certificates, etc. Different databases can have different connection requirements.
- The user can manage the stored connections:
 - The user can see a list of all databases available in the system.
 - The user can modify or remove existing database connections in the system.
 - The user can see storage information about all database connections defined in the system.
 - The user can empty any collection of any database connection defined in the system.
 - The user can drop any collection of any database connection defined in the system.
 - The user can insert dummy data in a new or any collection of any database connection defined in the system.
- The user can migrate data collections from one database connection to another by selecting the source and target connections. The source and target connections can be from different database implementations (MongoDB, CouchDB, Cosmos DB).
- The user can see information about the executed migration processes, like start date, end date, duration, and the number of migrated items.
- The user can define the database connection to be used by the **factserver**.
- The FactServer can connect to the configuration database to get the type and credentials of the connection it must use.
- The Factserver is extended to be able to connect to MongoDB and Cosmos DB and prototypically support queries for the **facts** collection.

A.1.5. Deployment

- All services (Frontend/Backend/Backend Database) can be deployed using Docker by running the docker-compose.yml file.
- For testing purposes, the Docker Compose file has services for MongoDB, CouchDB and Cosmos DB, these services are ignored by default. To enable considering them, go to **code/.env** , uncomment the **COMPOSE_PROFILES** line and add the databases that need to be included. For example:

```
1 COMPOSE_PROFILES=mongodb,couchdb,cosmosdb
2
```

- Environment variables are set inside the file **code/.env**, which is used by the docker-compose.yml.
- The ports used in docker-compose.yml:
 - db-manager-frontend: 4200

- db-manager-backend: 8080
- postgres: 5432
- mongodb: 27017
- couchdb: 5984
- comsodb: 8081, 10250-10255
- The system was tested on Windows 11 with Firefox 150.0.2 (64-Bit).

How to Start the System Using Docker

- Docker should have been installed and running on the machine.
- Clone or download this repo and enter the **code** folder of the project.
 - The file **docker-compose.yml** is located there.
- Open the command line interface (CLI).
- If you want to run the system for the first time, run the following command to build the Docker images:

```
1    docker -compose build
2
```

- After Docker images have been built, the system can be started by running the following command to start the file docker-compose.yml:

```
1    docker -compose up
2
```

- To access the User Interface (UI): go to your browser and enter: **http://localhost:4200/**.
- To access the Swagger API of the backend system, go to your browser and enter: **http://localhost:8080/apidocs/**.
- After finishing using the system, please stop the Docker containers by entering **Ctrl+C** in the CLI where you have started docker-compose.
- To stop and remove the containers and Docker resources, run:

```
1    docker -compose down
2
```

General Notes

- The expected host format for connections:
 - MongoDB: **<host>:<port>**.
 - CouchDB: **<protocol>://<host>:<port>** . When the protocol is missing, the default protocol is **http**.
 - Cosmos DB: **<protocol>://<host>:<port>**.
- For MongoDB, the credentials database (**authSource**) is hardcoded as **admin**.
- For all databases, make sure the credentials used have sufficient rights!
- The backend **url** used by the frontend is defined in:

A. Appendix

- **code/db-manager-frontend/src/app/environments/environments.ts**: for local development. Default: **http://localhost:8080**.
 - **code/db-manager-frontend/src/app/environments/environments.prod.ts**: for prod environment (used for Docker). Default: **http://localhost:8080**.
- The port of the backend can be modified in the file: **code/db-manager-backend/main.py**. Default: **8080**.
- The packages used in the backend are defined in the file: **code/db-manager-backend/requirements.txt**.
 - packages can be installed by running: **pip install -r requirements.txt** inside the folder **code/db-manager-backend**.
- When running the application in Docker:
 - **http://localhost** requests are forwarded to the hosting machine.
 - **https://localhost** requests can not be forwarded to the hosting machine, please use **https://host.docker.internal** instead.
 - The application can not connect to a Cosmos DB Emulator running on the hosting machine because of an SSL certificate issue.
- Make sure the ports used in Docker Compose are not already used by the host machine.