# Modernizing the Data Storage Interface of a Cloud Application

BACHELOR THESIS

**Jan Michael Rehnert**

Submitted on 23 October 2024

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Julian Lehrhuber, M.Sc.
Prof. Dr. Dirk Riehle, M.B.A.

Friedrich-Alexander-Universität
Faculty of Engineering

# Declaration of Originality

# License

# Abstract

This thesis presents the modernization of the backend architecture of QDAcity, a cloud-based web application that supports Qualitative Data Analysis (QDA). The existing backend of QDAcity is effective; however, it has limitations in terms of efficiently handling blob data, which can affect both performance and scalability. In order to address these challenges, the project introduces a new storage infrastructure, which serves to improve the management and organization of blob data associated with qualitative research. Moreover, the thesis presents a restructuring of the Datastore Application Programming Interface (API) with the objective of optimizing the system's performance and streamlining its interactions with the data. The principal enhancements include the flexibility of entity structures to support multiple blobs per entity, which is indispensable for more intricate data scenarios. In addition to these changes, the project implements strategies for better resource allocation and reduces latency in data retrieval, ensuring smoother operations. This comprehensive overhaul not only guarantees a more maintainable and scalable backend but also establishes the foundation for prospective extensions and improvements in the QDAcity platform.

# Contents

# List of Figures

# Acronyms

| | |
|---|---|
| **ACID** | atomicity, consistency, isolation, and durability |
| **API** | Application Programming Interface |
| **CRUD** | Create, Read, Update, Delete |
| **CVE** | Common Vulnerabilities and Exposures |
| **DAO** | Data Access Object |
| **DBMS** | Database Management Systems |
| **GAE** | Google App Engine |
| **GCS** | Google Cloud Storage |
| **JDO** | Java Data Object |
| **JDOQL** | Java Data Objects Query Language |
| **JPA** | Java Persistence API |
| **ORM** | Object Relational Mapping |
| **QDA** | Qualitative Data Analysis |
| **RDBMS** | Relational Database Management System |
| **SQL** | Structured Query Language |
| **URI** | Uniform Resource Locator |

# 1 Introduction

QDA is a crucial technique employed in qualitative research to interpret data by identifying and coding recurring themes or behaviors, which are typically collected through methods such as interviews. The process is iterative, involving the continuous collection and analysis of data until the emergence of new information no longer significantly alters the theory developing. A fundamental aspect of QDA is the coding process, by which specific labels are assigned to patterns in the data. This stage is of great importance and requires contextual understanding to ensure that the analysis captures more than just surface-level descriptions. The overall aim of QDA is to achieve a deep understanding of human behavior and thought processes, often culminating in the enhancement of existing theories as research progresses (Andreas Kaufmann, 2015).
QDAcity is a cloud-based application designed to facilitate QDA. The application is built with Java on the backend and React on the frontend. It operates in the Google Cloud, utilizing the *AppEngine*[1] hosting solution to ensure robust performance and scalability.
Prior to this thesis, QDAcity operated with a functional but suboptimal backend, particularly in its handling of blob data and datastore operations. The existing blob metadata management architecture had complexities that compromised its maintainability and hindered long-term adaptability. In addition, the datastore connection API, while effective, was outdated and not fully aligned with modern data management requirements. These issues made interacting with data more cumbersome and limited the system's potential for scalability and future enhancements.
In response to these challenges, this thesis aims to significantly improve the QDAcity system by modernizing its backend infrastructure. The main improvements focus on optimizing the storage and management of blob data through a redesigned approach that improves efficiency and simplifies data handling. This overhaul includes a restructured API for datastore operations to improve performance and create a more streamlined interaction with data. The backend code has also been reorganized to ensure better maintainability and extensibility, resulting in a cleaner, more efficient codebase. In addition, the thesis addresses

[1]https://cloud.google.com/appengine/docs/

the need to restructure certain entities within the datastore to support the storage of multiple blobs per entity.
Ultimately, the thesis aims to create a more robust and maintainable backend for QDAcity, laying the groundwork for future enhancements and ensuring that the system can grow and adapt to evolving requirements.

# 2 Technical Overview

As a Google Cloud-based platform, QDAcity utilizes a range of API offerings from the Google Cloud Platform. The following sections present an overview of the technologies employed, outlining their key advantages and disadvantages, while also exploring alternative options. This analysis will inform the development of the requirements for this thesis, which will subsequently be evaluated on the basis of their implementation.

## 2.1 Google Datastore

The Google Cloud offers the Google App Engine (GAE) Datastore, a highly scalable *NoSQL* database service designed to efficiently manage large amounts of data while ensuring optimal performance for web applications. Datastore provides flexibility, allowing developers to store and query data without the strict constraints of traditional relational databases, making it ideal for applications that require a schemaless dynamic data model.
The Datastore's automatic scaling ensures consistent performance as data volumes grow, which is crucial for applications handling fluctuating traffic. Its strong consistency model guarantees that all queries return the most up-to-date information, making it suitable for real-time data accuracy. Additionally, Datastore supports atomic transactions, maintaining data integrity by ensuring that all operations in a transaction either succeed together or not at all.
Entities of the same kind can have different properties and data types, which offers adaptability for rapidly evolving applications. This flexibility, combined with Google's management of the underlying infrastructure, allows developers to focus on building their applications without worrying about database maintenance ('Datastore Overview', 2024).

Unlike a traditional Relational Database Management System (RDBMS), Google Datastore is a key-value store where each entry corresponds to an entity. It functions more similarly to a *HashMap*, with the added capability to index and

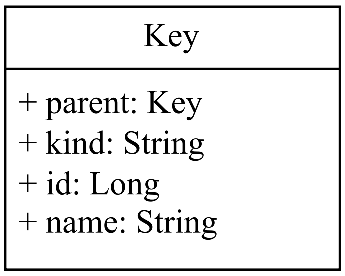

**Figure 2.1:** Datastore Key structure

query values[1]. In the Google App Engine datastore, each entity is uniquely identified by a key. Figure 2.1 displays the structure of a key, which consists of several components.

- *parent:* This is the optional ancestry reference. Each key may include a reference to its parent entity, which can also be *null*.
- *kind:* Serving as the entity type, this component is analogous to a table name in a relational database.
- *id:* Acting as the unique identifier for the entity, the ID may be auto-generated.
- *name:* This serves as an additional identifier for the key but cannot be auto-generated. It cannot be used in conjunction with the *ID*. Instead, it must be manually created as a unique string before the entity is saved, making it less convenient. In most cases, the *ID* is preferred.

Manually setting each entity property individually is tedious. In order to streamline this process, third-party libaries such as Java Data Object (JDO) and Objectify allow the definition of Java classes encompassing all the entity properties. Each of these tools provide an API that facilitates the storage and retrieval of objects from these classes, making data management much more efficient.

## 2.2 JDO

JDO is an interface-based Java model that provides persistence abstraction. Therefore, this API is a Java tool which helps the application to store and retrieve

[1]https://github.com/objectify/objectify/wiki/Concepts

data from databases (David Jordan, 2003).

> In general, each JDO implementation, as well as JPOX, comprises six distinct phases subsequent to those delineated by Leone and Chen (2007):
>
> 1. design the domain/model classes;
> 2. define their persistence using Meta-data;
> 3. compile the classes, and instrumenting them (using a JDO enhancer);
> 4. generate the database tables where the classes are to be persisted;
> 5. write the code to persist objects within the DAO layer; and
> 6. configure and run the application.

Initially, the process begins with the design of the domain/model classes, which represent the entities within the application. These classes are carefully crafted to ensure they accurately reflect the domain model and meet the specified requirements. Following this, the persistence of these classes is defined using metadata, provided through annotations or XML configuration files. This metadata outlines the mapping between the classes and the database tables, detailing aspects such as table names, column names, and relationships between entities. Once the persistence metadata is established, the domain/model classes are compiled and instrumented using a JDO enhancer. This enhancer modifies the compiled classes to include the necessary code for interacting with the JDO framework, enabling features such as transparent persistence and lazy loading. Subsequently, the database schema is generated based on the metadata definitions. This schema includes the tables, columns, and constraints required to store the data objects, ensuring that the database structure aligns with the domain model. Phase 5 involves the implementation of the Data Access Object (DAO) layer, which provides an abstraction for interacting with the database. The DAO layer contains methods for performing Create, Read, Update, Delete (CRUD) operations on the domain/model classes, ensuring that the persistence logic is separated from the business logic, thereby promoting a clean and maintainable architecture. Finally, the application is configured to utilize the JDO implementation and is executed. This configuration includes setting up the JDO properties, such as the database connection details and transaction management settings. Once configured, the application can be deployed and run, allowing it to persist and retrieve data objects as required.

Whenever new classes are introduced or changes are made to the data model, steps two through five must be repeated. This repetition is necessary, because any changes in the data model require updating the metadata, recompiling the classes with the JDO enhancer, and adjusting the database structure and storage logic to reflect the new or modified data structure (Leone & Chen, 2007).

### 2.2.1 DataNucleus

Since JDO is merely an interface, an implementation is required to manage the persistence of Java objects effectively. DataNucleus is an open source project that provides robust data management solutions for Java applications. It provides a fully compliant implementation of both the JDO's and Java Persistence API (JPA) specifications, enabling seamless and transparent persistence of Java objects (DataNucleus, 2022).

#### DataNucleus-AppEngine Plugin

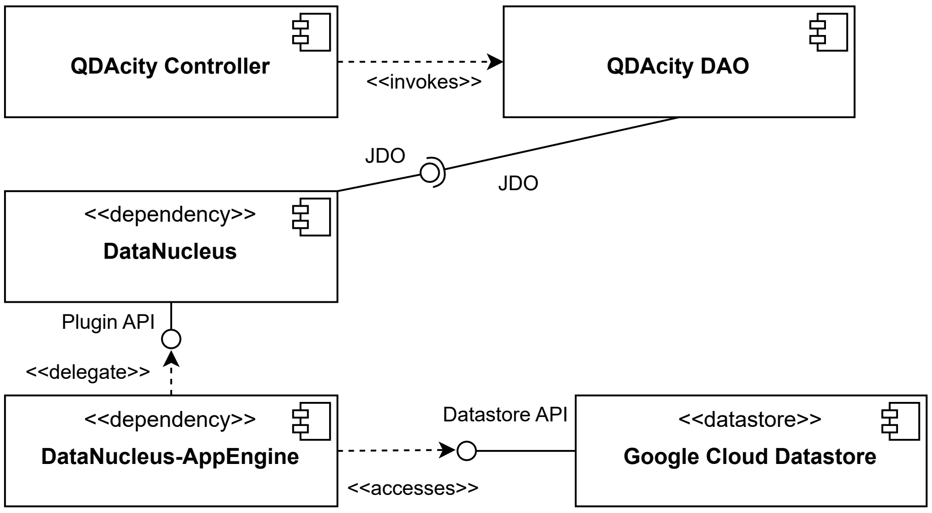

**Figure 2.2:** UML Diagram of QDAcity's Backend Architecture

The DataNucleus Plugin is a specialized extension for DataNucleus designed to work seamlessly with the GAE. It is noteworthy that the DataNucleus Plugin is provided by Google, and its most recent release occurred approximately ten years ago[2]. It is a DataNucleus plugin that provides Object Relational Mapping (ORM), that gives access to the Datastore[3]. ORM is a key concept, in object-oriented programming when interacting with Database Management Systems (DBMS). ORM allows developers to implement classes, whose objects are

[2]https://mvnrepository.com/artifact/com.google.appengine.orm/datanucleus-appengine/2.1.2

[3]https://github.com/GoogleCloudPlatform/datanucleus-appengine/tree/master/dist

translated into database entries by ORM (‘What is Object-Relational Mapping (ORM) in DBMS?’, 2024).
Figure 2.2 illustrates the relationship between the JDO API, DataNucleus, the DataNucleus Plugin, and the Google Datastore. The process begins with JDO, which initiates API calls that are directed to DataNucleus. DataNucleus acts as a mediator, handling the data operations. To interface with specific storage systems, DataNucleus relies on the DataNucleus-Plugin. This plugin extends the functionality of DataNucleus by adapting its operations to the requirements of the Google Datastore.
In this setup, when DataNucleus sends a request through the DataNucleus-Plugin, the plugin translates these requests into a format compatible with the Google Datastore. This translation process ensures that the operations specified by JDO are executed properly within the Google Datastore.
Once the Google Datastore processes the requests, it sends the results back to the DataNucleus-Plugin. The Plugin then forwards this response to DataNucleus, integrating the data or operation outcomes into the DataNucleus framework. Finally, DataNucleus returns the processed results to the JDO API, completing the data operation cycle[4, 5].

### 2.2.2 Culprits of using JDO with the Datastore

The integration of JDO with the Datastore presents several challenges, primarily due to limitations in the underlying DataNucleus plugin and the lack of continued support from Google. These factors have resulted in difficulties in maintaining long-term compatibility and adapting to future requirements. The following subsection outlines the key issues associated with this approach, which ultimately required the migration to an alternative API.

One significant challenge in this integration arises from the way the *Persistence-Manager*[6] is attached to objects, automatically triggering write operations when object properties are modified. Although this behavior may seem convenient, it introduces several practical issues.
Firstly, the implementation of frequent minor updates within cloud-based data stores inevitably results in elevated operational costs, which are calculated based on the number of write operations. The consolidation of updates into a single write operation serves to reduce costs and enhance performance by minimizing the number of network requests.
Moreover, implicit save operations frequently result in superfluous writes, even in the absence of significant alterations to the data. In contrast, explicit save opera-

[4]https://web.archive.org/web/20230303063554/https://cloud.google.com/appengine/docs/legacy/standard/java/datastore/jdo/overview-dn2

[5]https://www.datanucleus.org/documentation/development.html

[6]https://db.apache.org/jdo/pm.html

tions afford superior control, ensuring that data is only saved when it is required. This methodology enhances maintainability by providing clarity regarding the circumstances and locations where data is being persisted, thereby facilitating a more comprehensible and less error-prone codebase.

Another major issue arises from the outdated version of DataNucleus, particularly in its handling of inheritance. Upon cold-booting a backend instance and attempting to retrieve all objects that extend a common parent class, DataNucleus fails to return any objects, despite their previous persistence. This issue occurs because DataNucleus caches the metadata of persistent classes at runtime[7]. When querying for an entity type of a superclass after a cold boot, DataNucleus is unaware of all potential child types for that class. Consequently, it only returns objects of the exact queried type. However, if the superclass is abstract, no objects are returned. As a provisional measure, the persistable subclasses are instantiated upon the boot of the application, thus ensuring the DataNucleus type cache is correctly initialized.

A related issue arises when querying objects that inherit from a common superclass. Properties specific to the child classes are not populated and instead are set to *null* because these properties are unloaded during the initial query. This happens because these properties are unloaded during the initial query. In theory, this behavior should be manageable using JDO's *FetchGroups*[8], which are designed to control the loading of specific fields. However, when working with the native Google Datastore API, there are two main methods to retrieve entity data. One method fetches all the entity data, while the other retrieves only the entity keys. Since the DataNucleus plugin relies on the *AppEngine API 1.0 SDK*[9], it uses the native Datastore API, which restricts data retrieval to these two options. This raises the question of why the *FetchGroup* feature, designed to control the loading of specific fields, would be necessary. Configuring *FetchGroups* to load all data, given that the default configuration does not handle subclasses, introduces unnecessary overhead. This approach adds complexity without providing significant benefits, as it often results in retrieving all entity data. Therefore, relying on *FetchGroups* is not the most efficient solution and only increases system complexity.

Another issue arises with the *PersistenceManagerFactory* in JDO, which is capable of pooling multiple *PersistenceManager* objects. Although *PersistenceManager* instances can be reused, no control is available, which specific instance the factory provides. Hence, it is more convenient to generate them on demand. In case of using different instances in rapid succession, one modifying data and

[7]https://github.com/datanucleus/datanucleus-core/blob/datanucleus-core-3.2.11/src/java/org/datanucleus/metadata/MetaDataManager.java#L1736

[8]https://db.apache.org/jdo/content/api32/apidocs/javax/jdo/FetchGroup.html

[9]https://mvnrepository.com/artifact/com.google.appengine/appengine-api-1.0-sdk/1.9.80

the other one attempting to read the modified data, it is possible that the read operation may not reflect the recent changes. By consistently utilizing the same instance of the PersistenceManager, the issue was effectively circumvented [6].

Additionally, when incorporating lambda expressions within an entity class, errors arise during the enhancement process executed by DataNucleus 3.1.1, which is utilized by QDAcity. This version of DataNucleus depends on the ASM library version 4.0 for bytecode manipulation[10]. ASM 4.0 does not support Java 8 language features, meaning that features like lambda expressions cannot be used. Support for Java 8 language features was introduced in ASM 5.0 beta[11].

## 2.3 Objectify

The best-known alternative to JDO in a GAE environment is Objectify. Objectify is a Java API designed to simplify data access to the Google Cloud Datastore specifically. It strikes a balance between usability and transparency, offering a more user-friendly experience than JDO, while being far more convenient to use than Google's low-level Datastore API. Objectify aims to help beginners become productive quickly, while still providing access to the full capabilities of the Datastore (GitHub, 2024). One advantage is that it utilizes Java generics to provide type-safe Datastore keys and queries. This means that the key used to retrieve an entity must be compatible with the type of entity being accessed. An example of generic *Key* creation is shown in Listing 2.1.

**Listing 2.1:** Objectify Datastore Key Creation

```java
Key<User> key = Key.create(User.class, id);
User user = ofy().load().key(key).now();
```

## 2.4 Key Differences between JDO and Objectify

As developers consider different approaches for integrating with Google Cloud Datastore, it's important to explore how each library interacts with the platform. The integration process can significantly influence the effectiveness and efficiency of data management within applications. This section examines the specific challenges and benefits of integrating JDO and Objectify with Google Cloud Datastore, highlighting how each framework's design and dependencies affect its functionality and ease of use.

[10]https://asm.ow2.io/

[11]https://asm.ow2.io/versions.html

### 2.4.1 Complexity and Learning Curve

JDO's rich and powerful feature set contributes to a steep learning curve. Developers must familiarise themselves with numerous concepts, annotations and configurations in order to use JDO effectively. This complexity can be daunting for beginners and can take considerable time and effort to master[12]. In addition, finding a reliable implementation of the JDO interface can be challenging, as they may not function as expected, requiring developers to spend extra time troubleshooting and finding workarounds. Furthermore, JDO includes annotations that the Google Datastore does not support, such as uniqueness constraints. JDO is also a very large library, which can further complicate integration and increase the overhead for developers trying to navigate its extensive features. Nevertheless, the comprehensive functionality of JDO offers developers greater flexibility in terms of configuration and utilization when compared to Objectify.
Objectify, on the other hand, is designed for simplicity and ease of use. It provides a more streamlined approach to interacting with the Google App Engine datastore. Objectify abstracts much of the complexity inherent in JDO and provides a more intuitive API that is easier for developers to understand, especially when working with the Datastore. This results in a shorter learning curve, allowing developers to become productive more quickly. While Objectify simplifies many aspects, developers still need to configure it, open an *ObjectifyService*[13], and register each entity class, though this process is much less involved compared to the more complex setup required by JDO.

### 2.4.2 Integration with Google Cloud Datastore

Integrating with Google Cloud Datastore requires careful consideration of the tools and frameworks used, as different options offer varying levels of support and complexity. JDO supports a range of features such as queries, transactions, and object relationships, making it a robust choice for complex applications. However, integrating JDO with Google Cloud Datastore presents several challenges. Specifically, the integration process necessitates the use of additional dependencies, such as DataNucleus and the DataNucleus-Plugin. These dependencies introduce significant complexity to the setup process. Developers must manage multiple configurations and ensure compatibility between different versions of these plugins, which can be complex and time-consuming.
In contrast, Objectify simplifies the integration with the Google Datastore by requiring no additional dependencies. This streamlined approach reduces the potential for compatibility issues and simplifies the setup process. Objectify supports all native datastore features, including transactions, queries, and batch operations, while offering added conveniences such as automatic caching and an

[12]https://stackoverflow.com/questions/4232944/accessing-the-gae-datastore-use-jdo-jpa-or-the-low-level-api
[13]https://github.com/objectify/objectify/wiki/Setup#initialize-the-objectifyservice-and-register-your-entity-cla

intuitive query interface. Objectify's design philosophy emphasizes ease of use and rapid productivity, making it a preferred choice for developers working with Google Cloud Datastore. However, developers still need to create the necessary annotations and classes to map their entities to the Datastore[14].

### 2.4.3 Type Safety

While integrating with Google Cloud Datastore involves navigating varying complexities, the approach to type safety further distinguishes the two frameworks in terms of development ease and reliability. Through Java Data Objects Query Language (JDOQL), JDO provides a flexible query mechanism. JDOQL gives developers the flexibility to choose the level of type safety they require by supporting both typed and untyped queries. JDOQL's type safety is not as strong as contemporary query languages, and if queries are not written carefully, developers may run into runtime errors. However, our outdated version of DataNucleus does not yet support the usage of type-safe queries, which further limits the effectiveness of type safety in practice. In addition, the verbose and complex nature of JDO's query API requires a thorough understanding of the underlying data model and query language. As a result, writing and maintaining type-safe queries can be challenging, especially in large and complex applications (Google, 2024; Zeger Hendrikse, 2017).
On the other hand, Objectify provides a simpler and more type-safe query experience[15]. Type safety is prioritized in the user-friendly API design of Objectify, which uses Java generics. By ensuring that queries are checked at compile time, this reduces the possibility of runtime errors and raises the standard of the code as a whole. In addition, Objectify provides an easy-to-use query interface that simplifies the creation and execution of queries. As a result, developers who prioritize type safety in their applications will find Objectify a suitable choice.

### 2.4.4 Performance

Although both frameworks possess distinctive advantages in regard to type safety, their divergences also extend to performance, with each framework exhibiting a unique approach to optimizing queries and data interactions. The performance of JDO can be affected by its reliance on JDOQL for querying data, which is not as optimized for Google Cloud Datastore as native query languages. In real-world applications, developers have reported issues with poor performance, especially when dealing with large datasets or complex queries. The need to make multiple round trips to the datastore for certain operations can further degrade performance (Oscar Rosner, 2021).

[14]https://github.com/objectify/objectify/wiki/Entities
[15]https://github.com/objectify/objectify/wiki/Queries

Objectify's API is optimized for performance, offering features such as automatic caching and batch operations that can significantly reduce the number of data store interactions[16]. This results in faster query execution and improved overall performance. The framework's ability to efficiently handle large datasets and complex queries makes it a popular choice for developers seeking high performance with Google Cloud Datastore.

## 2.5 Other data storage interfaces

When evaluating alternatives to Objectify for Google Cloud Datastore, the options are rather limited. The use of JDO with the DataNucleus plugin is largely outdated and typically confined to legacy projects. In fact, Google themselves discourage the use of JDO/DataNucleus, as indicated by warnings in their documentation[17]. Furthermore, with the end of support for the Java 8 AppEngine runtime and the prohibition of deploying such runtimes after January 31, 2024[18], using JDO/DataNucleus is no longer a viable option unless legacy code is repackaged for newer runtimes. This reliance on outdated technology can hinder the ability to update project dependencies, creating compatibility issues and limiting the adoption of newer libraries or tools. Additionally, the dependency on DataNucleus complicates the migration from older Java versions, particularly from Java 8 to more recent versions, due to incompatibilities. These factors make the continued use of JDO with DataNucleus increasingly impractical in modern development environments.
In response to these challenges, Google is actively developing the *Spring Data Cloud Datastore*[19], which uses the familiar Spring Data Repository to interact with Google Cloud Datastore. The Spring Framework is a comprehensive platform designed to simplify Java enterprise application development. It provides infrastructure support, dependency injection, and a wide range of modules tailored to various application needs (João André Martins, Jisha Abubaker, 2024).
However, this option is currently not feasible for QDAcity. At the time of writing this thesis, the migration to Spring Boot was not yet complete. Furthermore, relying on another Google-provided framework carries an inherent risk, given that Google has previously discontinued other projects.
Another alternative is to use Google's Native Datastore API[20]. This API provides direct access to the features of Google Cloud Datastore, but requires significantly

[16]https://github.com/objectify/objectify/wiki

[17]https://web.archive.org/web/20230303063554/https://cloud.google.com/appengine/docs/legacy/standard/java/datastore/jdo/overview-dn2

[18]https://web.archive.org/web/20240705164743/https://cloud.google.com/appengine/docs/standard/lifecycle/support-schedule#java

[19]https://web.archive.org/web/20240910215033/https://googlecloudplatform.github.io/spring-cloud-gcp/reference/html/index.html#spring-data-cloud-datastore

[20]https://cloud.google.com/datastore/docs/concepts/overview

more code and a deeper understanding of the underlying system. Objectify essentially acts as a wrapper around the Native Datastore API, designed to streamline interactions and minimize the amount of boilerplate code.

# 3 Requirements

Building on the differences between JDO and Objectify, along with the necessity to upgrade the Java Runtime of QDAcity, this chapter outlines the requirements that must be met for this thesis. It encompasses both functional and non-functional requirements, which have been formulated using Rupp's templates. These templates facilitate the creation of requirements that are comprehensible, comprehensive and verifiable. They provide a systematic framework for formulating requirements in software engineering, emphasizing clarity and completeness. The use of predefined structures through the use of these templates enables the formulation of explicit, testable requirements, which in turn facilitate enhanced communication among stakeholders and streamline the development process(Chris Rupp, 2014).

## 3.1 Functional requirements

The term functional requirements is used to delineate the particular tasks and features that the system must provide in order to satisfy user needs and accomplish its intended purpose. In accordance with the ISO 25000 standard, these requirements are indispensable for guaranteeing that a system is valuable by effectively assisting users in their tasks. This section elucidates the functional requirements pertinent to this thesis and which have been classified into distinct categories for enhanced comprehension (Henri Basson, 2016).

### 3.1.1 API Reliability and Compliance

a) The new API shall be capable of utilizing the necessary features of the Google Datastore.

b) The new API shall ensure the integrity and consistency of the data.

c) The new API shall be actively maintained, with at least one official release in the year 2024 as a criterion for active maintenance.

d) The new API shall be permissively licensed, allowing for modification, distribution, and commercial use with minimal restrictions.

### 3.1.2 API migration

a) QDAcity shall support migration from JDO to a new framework without data loss, ensuring compatibility with existing data structures while introducing new enhancements.

b) QDAcity shall enable incremental migration from JDO to the new API, allowing gradual updates without necessitating a complete transition at once.

### 3.1.3 GCS blobs

a) The QDAcity backend component for managing GCS blobs shall be able to create, read, update, and delete data.

b) The QDAcity backend component for managing GCS blobs shall be able to manage multiple blobs connected to a single Datastore entity.

## 3.2 Non-Functional Requirements

Non-functional requirements are concerned with the criteria for evaluating a system's operational performance, as opposed to its specific behaviors or functions. In accordance with ISO 25000, these requirements encompass a multitude of quality characteristics that influence how users interact with the system, including efficiency, ease of use, dependability, and safety. They delineate the manner in which a system should execute its functions and establish the benchmarks to be attained throughout its entire lifecycle. This section elucidates the non-functional requirements pertinent to this thesis, thereby ensuring that the proposed system meets the indispensable quality standards while augmenting its overall efficacy and user satisfaction (Henri Basson, 2016).

### 3.2.1 Datastore Access Encapsulation

a) Code that accesses the datastore shall be encapsulated within a dedicated layer of the software architecture, to ensure maintainability and testability.

b) Each datastore table shall be managed through a dedicated class, allowing for shared access among related entities while maintaining a clear separation of concerns.

### 3.2.2 System Maintainability and Future-Proofing

a) The refactoring shall enhance the codebase's maintainability by establishing a clear structure and reducing duplication, thereby facilitating efficient feature additions.

b) QDAcity shall be able to update dependencies and integrate new dependencies.

c) QDAcity shall be compatible with all next-generation GAE APIs.

d) QDAcity shall address and mitigate security vulnerabilities present in the previous implementation.

e) QDAcity shall decrease the number of Datastore write operations in its backend.

### 3.2.3 Documentation

a) The code shall be documented using *JavaDocs* to facilitate future development and maintenance.

b) The QDAcity wiki shall be extended to include the new API.

c) This thesis shall serve as comprehensive documentation for future developers working with QDAcity.

# 4 Architecture

This chapter provides an overview of the software architecture, beginning with an examination of the modifications made to the entire backend design. It then outlines the key architectural changes to the *Project* and *Document* classes. Finally, it details the adjustments necessary to enable the storage of multiple blobs per entity instance.

## 4.1 Structured Backend Architecture

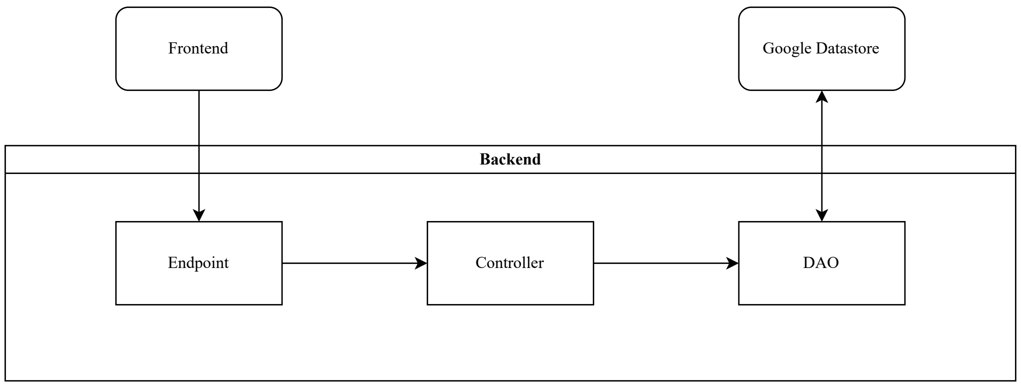

**Figure 4.1:** Endpoint Controller DAO structure

The existing backend architecture of QDAcity is beset with significant issues that require a comprehensive refactoring process. In the previous structure, Google Datastore calls were dispersed throughout the entire backend, including within the endpoint classes. This lack of organization results in a disorganized codebase, characterized by a minimal number of DAO classes and a limited number of controller classes. Consequently, the controller classes are bloated, housing excessive code that complicated maintenance and readability. Additionally, most controller methods are static, contributing to a less aesthetically pleasing and more challenging backend architecture.
Furthermore, this scattered approach complicates the desired migration from

JDO to Objectify, as the pervasive JDO calls inhibit the transition. The new structured architecture, defined as the Endpoint Controller DAO structure, provides an effective solution to these issues. As illustrated in Figure 4.1, the new design facilitates interaction between the frontend and endpoint, which is followed by an authorization. In the event of authorization being granted, the endpoint then proceeds to invoke the pertinent controller classes, which are responsible for the handling of the core logic, and calls upon the DAO classes.
The QDAcity codebase has become increasingly disorganized over time due to the contributions of multiple developers. The previous architecture, which is characterized by a widespread distribution of datastore calls, deviated from established best practices and highlighted the necessity for a more structured approach. The Endpoint Controller DAO architecture will significantly improve the organization and maintainability of the codebase. The structured design not only facilitates the modification of system logic but also simplifies the incremental migration process from JDO to Objectify. By centralizing data access within the DAO layer, the new architecture allows for a more straightforward transition, thereby facilitating adaptation to Objectify's requirements while preserving the integrity of existing functionalities.

## 4.2 Integration of the Visitor Design Patterns to Enhance the Codebase

The codebase contains several child implementations of the *BaseProject* and *BaseDocument* classes, each incorporating distinct logic for operations such as deletion, loading, and saving. To unify this logic, the visitor design pattern will be implemented and extended. The visitor pattern, a behavioral design pattern, facilitates the addition of new operations to objects without altering their underlying classes. By decoupling the algorithm from the object structure, this pattern enables the introduction of new operations to existing object structures without modifying the structures themselves. This approach is particularly advantageous for maintaining adherence to the open/closed principle in object-oriented programming (Erich Gamma, 1994).

The implementation of this design pattern offers several benefits:

- Extending the project/document classes becomes more straightforward. When a new class extending *BaseProject* or *BaseDocument* is introduced, an additional method must be incorporated into the visitor interface and its implementations. The necessary logic must then be implemented to ensure that the method is executed correctly.
- By using the visitor pattern, the logic for the datastore is separated from the data structure itself. This separation makes the codebase more modular

and easier to maintain, as the behavior can be modified independently of the object structure.

- The visitor pattern aligns with the open/closed principle by allowing the extension of functionality without altering existing code. New operations can be added without the need to modify existing classes, reducing the risk of introducing bugs and ensuring that existing functionality remains intact.
- Since the logic for different operations is centralized in the visitor implementations, changes to these operations need to be made in only one place. This centralization reduces code duplication and makes the system easier to maintain and update.
- The visitor pattern provides flexibility in terms of the operations that can be performed on objects. It allows for different operations to be applied to the same object structure, facilitating the addition of new functionality as the system evolves.

### 4.2.1 BaseProject classes

Figure 4.2 illustrates the architecture of the project classes alongside the visitor interface. The *BaseProject* class currently has five inheriting subclasses, each with a corresponding method in the visitor interface. When a new visitor class is created, it must implement all the methods defined in the visitor interface, along with the required logic. This ensures that the specific logic required for each subclass is effectively separated and executed.

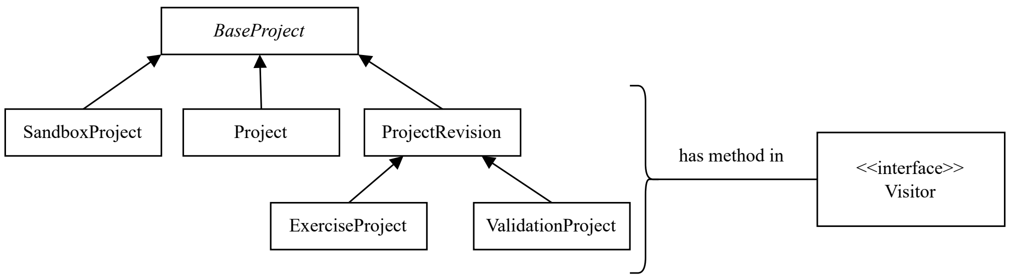

**Figure 4.2:** Architecture of *BaseProject* classes

### 4.2.2 BaseDocument classes

Figure 4.3 outlines the architecture of the document classes alongside the visitor interface. The visitor design pattern is already implemented in these classes and will be extended with a new visitor class dedicated to handling the deletion of documents. In addition to removing the document itself, this class is responsible for deleting the associated content stored in GCS. Since this content removal process must be applied to every subclass of the document, the visitor pattern is particularly effective. Each type of document can also have related data that must be deleted, which can vary depending on the specific subclass. By introducing a new visitor class specifically for these deletions, it was possible to encapsulate the deletion logic for each type of document within its respective method in the visitor interface.

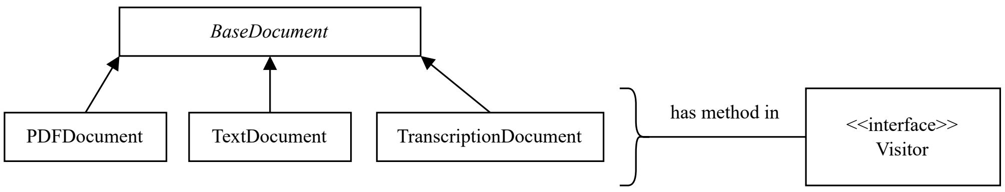

**Figure 4.3:** Architecture of *BaseDocument* classes

## 4.3 GCS Blob Architecture

QDAcity is currently required to store data in the form of blobs, which frequently exceed 1 megabyte in size, in conjunction with document and project entities. To efficiently manage this, the blobs are stored in the GCS, while only the paths to these blobs are stored in the Datastore. This approach optimizes storage and retrieval processes by leveraging the GCS for large data storage and the Datastore for metadata management. At the time of writing, the GCS paths are stored in individual entity properties. Although this implementation permits the storage of multiple blob paths with a single entity, it lacks an accessible method for associating additional blob metadata with the entities. Furthermore, the addition of further blob paths would necessitate the definition of additional entity properties. It would be optimal for this blob metadata to be stored in a list or a map, thus obviating the necessity for individual entity properties. In order to accommodate additional blob metadata stored alongside the blob path and to enhance the extensibility of an entity with further blob metadata, it was necessary to refine the architecture of the blob management system.
The already existing *GcsClient* class is responsible for the management of operations pertaining to the saving, loading, and deletion of data within GCS. The methods in this class interact with an interface that only has a single method to retrieve a string representing the GCS path where the blob information is stored. These methods need to be updated to accept a *GcsBlob* entity as a parameter. The *GcsBlob* entity is capable of storing a string representing the GCS path, a byte array containing the content of the blob and a boolean value to indicate to the client that updated content need to be sent to the server instead of directly to GCS. As a consequence of this modification, any class that requires the storage of data will then be required to save a *GcsBlob* in the Datastore, while also

assuming responsibility for managing the connection with the *GcsClient* class. It is of paramount importance to annotate the byte array in the *GcsBlob* class with *@Ignore* in order to prevent this property from being stored in the Datastore, given that it may be of considerable size.
Furthermore, a strategy design pattern will be implemented through the creation of an interface designated as *HasGcsBlobs.* The interface defines a single method for the retrieval of all necessary *GcsBlobs*, which are required for interaction with other classes. The implementation of this design pattern will lead to the realization of several pivotal enhancements. Firstly, the *HasGcsBlobs* interface allows for different implementations, thereby enabling various classes to define their own strategies for managing *GcsBlobs.* This separation has the effect of enhancing the maintainability and readability of the code. Additionally, the employment of the BlobStrategy interface enables the creation of unit tests for classes that interact with GCS blobs. By means of mocking the implementations of the *HasGcsBlobs*, it is possible to isolate and test the behavior of the aforementioned classes without having to rely on actual GCS interactions. Furthermore, the strategy design pattern allows the system to scale more effectively. In the event of new types of *GCS* blobs or storage requirements emerging, it is possible to create new implementations of the *HasGcsBlobs* interface without having to modify existing code. This extensibility ensures that the system can grow and evolve over time.

# 5 Design and Implementation

This chapter provides a detailed look at how the elements described in chapter 4 were implemented. It highlights the key features of each component and how they work together as a system. The chapter also covers specific topics related to the implementation of these components. In addition, other related implementations are discussed to give a complete understanding of how the architecture was put into practice. The following sections will go into more detail about the technical aspects and the reasons behind certain design choices.

## 5.1 Structured Backend Design

The majority of the programming work for this thesis involved the refactoring of the entire backend codebase of QDAcity, with the objective of creating a structured Endpoint Controller DAO architecture. In this structure the frontend interacts with existing endpoint classes that first create a context, which is a specialized class that initiates an *ObjectifyService* and handles additional logic such as user management. Before proceeding with any logic, the endpoints perform an authorization check. If the user is not authorized, an *UnauthorizedException* is thrown. Otherwise, the endpoint continues by calling the necessary controller classes, which handle the primary logic and call the DAO classes responsible for caching and interacting with the datastore using Objectify.
Over time, the QDAcity codebase has gradually become disorganized due to the large number of people working on the project. For example, datastore calls are scattered throughout the codebase, including within endpoint classes. This was not best practice and led to the implementation of the new structure, which required the refactoring of approximately 28000 lines of code.
The benefits of this change include the elimination of duplicate code, making the code base more readable and easier to test. In a multi-contributor project, duplicate instances of code are inevitable, nevertheless this new structure helps to mitigate the problem by making it easier to identify whether a method has already been implemented.
In addition, this structure makes it easier to modify the logic. For example, if a

*User* entity needs to be deleted along with its associated *UserLoginProviderInformation* (which is linked by a key but not embedded), only the delete method in the *UserDAO* class needs to be updated to handle both deletions.
Another advantage of this structure is its compatibility with the migration to Objectify. Unlike JDO, Objectify does not have a delete hook, but this is no longer an issue. The necessary calls can be added to the delete method of the DAO class, effectively replicating the previous behavior.

### 5.1.1 Data Access Object

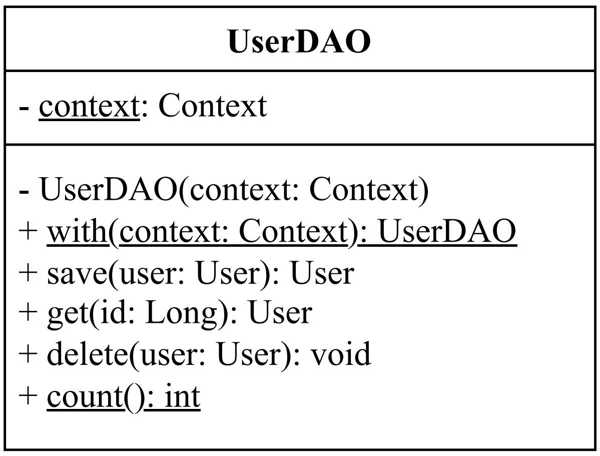
UserDAO
- context: Context
- UserDAO(context: Context)
+ with(context: Context): UserDAO
+ save(user: User): User
+ get(id: Long): User
+ delete(user: User): void
+ count(): int

**Figure 5.1:** Typical DAO Structure

One of the most crucial tasks was creating DAO classes to decouple the rest of the codebase from Google Datastore. Each entity class, with the exception of those who share a table with others, has its own dedicated DAO class. This does not include embedded entities, which lack their own DAO as they are saved/deleted/-loaded with the entity they are embedded in. Entities that share a table due to a common abstract parent class, also share one DAO class. In these cases, a single DAO class is used for all entities, with the class type passed as a parameter in the load methods. A primitive example of a DAO class with it's properties and methods is shown in 5.1.

- *context:* Initially, the primary role of the context class was to manage the *PersistenceManager* instance before migrating to Objectify. Its main function now is similar, as it currently manages the *ObjectifyService* instance.
- *with(context):* This method acts as a Factory Method, creating new DAO instances with controlled instantiation (Erich Gamma, 1994).

- *save(user):* When saving entities, this method not only uses *Objectify* but also leverages *memcache* to cache them. The saved entity is returned, allowing the method's return value to be directly assigned to any variable the caller is declaring.
- *get(id):* Retrieving an entity from the datastore is straightforward with this method, which relies on the entity's unique ID.
- *delete(user):* Removal of the entity is handled here, ensuring it is deleted from both the datastore and the cache.
- *count():* This method queries the datastore using the *DatastoreFacade* class, providing the current count of stored entities.

These are the core methods that every DAO class includes. However, many DAO classes also implement additional methods tailored to more specific needs.

- *getAll()*: This method allows for querying all entities of a certain type, often using specific properties or filters to narrow down the results.
- *saveAll(users)*: When dealing with multiple entities, this method efficiently saves all provided entities in a single operation, ensuring batch processing and better performance.
- *deleteAll(users)*: For bulk deletions, this method removes all specified entities from both the datastore and the cache, streamlining the cleanup process.
- *countByFilter(filter)*: This method provides a way to count entities based on a specific filter, enabling precise queries on the datastore without needing to load all entities.

The primary advantage of these methods is their efficiency, as the DAO only needs to connect to the Google Datastore once, to handle all entities in a single operation. This reduces the overhead of multiple datastore interactions, significantly improving performance. Additionally, by batching operations like saving, deleting, or querying entities, these methods streamline processes, minimize latency.

### 5.1.2 Entities in Depth

The transition to Objectify is facilitated by the structure of the backend, where each Datastore table is associated with its own DAO class. This design allows for an incremental migration, enabling the migration of each entity group to Objectify individually while still allowing for the coexistence of entities utilizing JDO. This approach facilitates a smooth transition without requiring a complete overhaul of the codebase at once.

Table 5.1 illustrated a structured comparison of the annotations used in JDO and Objectify. It is divided into two columns, each labeled with the respective

| **JDO Annotations** | **Objectify Annotations** |
|---|---|
| @PersistenceCapable(identityType) | @Entity |
| @PrimaryKey | @Id |
| @Persistent(valueStrategy) | are persistent automatically |
| @NonPersistent | @Ignore |
| are indexed automatically | @Index |
| @PersistenceCapable(identityType) @Inheritance(strategy = InheritanceStrategy.SUPERCLASS_TABLE) | @Subclass |

**Table 5.1:** Comparison of JDO and Objectify Annotations

framework's annotations. A notable distinction between JDO and Objectify is the approach to annotations for persistence and indexing. While JDO requires explicit annotations, Objectify simplifies this process by making all properties persistent by default. In Objectify, developers are not required to annotate each field to indicate its persistence, which results in a more streamlined code structure. Furthermore, JDO automatically indexes all properties without the necessity for specific annotations, whereas Objectify requires the use of the *@Index* annotation for the explicit indexing of fields. This distinction can enhance the efficiency of the Datastore queries. Moreover, while both frameworks support inheritance, JDO necessitates the utilization of particular strategies to facilitate subclassing, whereas Objectify simplifies this process through the *@Subclass* annotation, thereby rendering it more intuitive for developers. In general, Objectify's annotations are more concise and easier to comprehend for those new to the field. However, JDO offers a more extensive range of settings to customize the storage type, which can be advantageous for more complex data modelling requirements.

### Standard Entity Example

There are multiple ways to store entities in the datastore, requiring a decision on whether the entity should be stored in its own table or in a shared table alongside other entities. To store an entity in its own table using JDO, the class is annotated with *@PersistenceCapable*, where the *identityType* is specified to indicate the entity's identity management strategy, such as *IdentityType.APPLICATION*. The ID field is annotated with *@PrimaryKey* and requires the *@Persistent* annotation, with the *valueStrategy* determining how the ID is generated. Additionally, persistable fields like *email* require the *@Persistent* annotation, while fields like *sessionToken* that should not be saved are marked with *@NotPersistent*, as shown in Listing 5.1.

**Listing 5.1:** Common JDO Entity

```java
@PersistenceCapable(identityType = IdentityType.APPLICATION)
public class User {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    String id;

    @Persistent
    String email;

    @NotPersistent
    String sessionToken;
}
```

In contrast, when using Objectify, the class is annotated with *@Entity*, and the ID field is marked with *@Id*. Fields that should not be saved in the Datastore are annotated with *@Ignore*, while those that need to be searchable are marked with *@Index*. An example *User* class using Objectify is illustrated in Listing 5.2. A comparison of the Objectify class with the previous example reveals that the annotations are shorter, although JDO offers greater control over annotations.

**Listing 5.2:** Common Objectify Entity

```java
@Entity
public class User {
    @Id
    String id;

    @Index
    String email;

    @Ignore
    String sessionToken;
}
```

### Entities using Polymorphism

The code examples above show a common scenario where an entity is stored in the Datastore independently, without any dependencies on other entities. On the other hand, there are two methods to store multiple entities in one table. The first method is to use polymorphism and the second is to embed an entity in another entity. Listing 5.3 illustrates the utilization of polymorphism for the storage of entities using JDO.

**Listing 5.3:** JDO Entities as Subclasses

```
@PersistenceCapable(identityType = IdentityType.APPLICATION,
    table = "User")
@Inheritance(strategy = InheritanceStrategy.NEW_TABLE)
@Discriminator(strategy = DiscriminatorStrategy.CLASS_NAME)
public abstract class BaseUser {
    @PrimaryKey
    String id;
}
@PersistenceCapable(identityType = IdentityType.APPLICATION)
@Inheritance(strategy = InheritanceStrategy.SUPERCLASS_TABLE)
public class AdminUser extends BaseUser {
    @Persistent
    String adminLevel;
}
@PersistenceCapable(identityType = IdentityType.APPLICATION)
@Inheritance(strategy = InheritanceStrategy.SUPERCLASS_TABLE)
public class NormalUser extends BaseUser {
    @Persistent
    Long numberOfPosts;
}
```

In this code example, the entities are stored together in a Datastore table named *User*. Beyond simple storage, JDO enhances the process by adding extra columns to manage polymorphism. Specifically, a column called *DISCRIMINATOR* is added to the table to identify the current subclass type, such as *AdminUser* or *NormalUser*. A similar approach is demonstrated in Listing 5.4 using Objectify.

**Listing 5.4:** Objectify Entities as Subclasses

```

@Entity(name="User")
public abstract class BaseUser {
    @Id String id;
}
@Subclass(index=true)
public class AdminUser extends BaseUser {
    @Index String adminLevel;
}
@Subclass(index=true)
public class NormalUser extends BaseUser {
    Long numberOfPosts;
}
```

In addition to storing, Objectify enhances this process by automatically adding additional columns to handle the metadata for these entities. One of these columns is `^d`, which works the same way as the *DISCRIMINATOR* column in the JDO code example. Another column, `^i`, contains a list of parent classes along with the class type of the current entity. In this case, it would represent a

list with a single element corresponding to the current class type.
Both Objectify and JDO support polymorphism, but their approaches differ slightly. Objectify stores all entities in a single table and automatically adds metadata columns to track the entity's type and hierarchy. In contrast, JDO provides greater control over table structure, allowing for either shared or separate tables for subclasses through its *@Inheritance* annotation. Additionally, JDO allows you to rename the discriminator column used to distinguish between subclass types, which can not be renamed in Objectify. While Objectify streamlines the setup by handling everything in one table, JDO offers more flexibility for custom storage configurations.

### Embedded Entities

Additionally, both APIs allow for the embedding of one entity within another. Listing 5.5 demonstrates how this can be accomplished using JDO.

**Listing 5.5:** Embedded Entities in JDO

```
@PersistenceCapable(identityType = IdentityType.APPLICATION)
@EmbeddedOnly
public class Project {
    @Persistent
    String name;
}
@PersistenceCapable(identityType = IdentityType.APPLICATION)
public class User {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    String id;

    @Persistent
    @Embedded
    List<Project> projects;
}
```

The above example shows how to embed entities in JDO, where the *Project* class is marked as *@EmbeddedOnly*, allowing it to exist solely within the *User* entity. In contrast, Listing 5.6 illustrates a similar approach in Objectify, where the *Project* entity is defined without additional annotations, demonstrating its direct embedding within the *User* entity.

**Listing 5.6:** Embedded Entities in Objectify

```java
@Entity
public class Project {
    String name;
}
@Entity
public class User {
    @Id String id;
    List<Project> projects;
}
```

The Datastore representation now has exactly one table called *User* with a column named *projects* containing the list of projects for each *User*. When using embedded entities like this, it is essential to be aware of the size of a *User* entity, since a Datastore entity is limited to a size of 1 megabyte.

### Entities using Foreign Keys

Lastly, with both APIs, it is possible to connect entities indirectly by storing a key with them. Listing 5.7 shows how this can be achieved in JDO.

**Listing 5.7:** Foreign Key Entities in JDO

```java
@PersistenceCapable(identityType = IdentityType.APPLICATION)
public class User {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    String id;

    @Persistent
    Key info;
}
@PersistenceCapable(identityType = IdentityType.APPLICATION)
public class LoginInformation {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)

    @Persistent
    Key user;
}
```

In Listing 5.8, a similar approach is taken in Objectify, where both the *User* and *LoginInformation* classes utilize keys to create indirect connections between the entities.

**Listing 5.8:** Foreign Key Entities in Objectify

```java
@PersistenceCapable(identityType = IdentityType.APPLICATION)
public class User {
    @Id String id;
    Key<LoginInformation> info;
}
@Entity
public class LoginInformation {
    @Id Long id;
    Key<User> user;
}
```

This link can be used to establish secure connections between entities and manage one-to-one or one-to-many relationships. This method ensures that references are clear and unambiguous, as each *Key* uniquely identifies an entity across all tables. The primary distinction between the two is that JDO does not provide its own *Key* implementation. Instead, it uses the native Google Datastore *Key* structure. In contrast, Objectify offers its own *Key<T>* class, which serves as a wrapper for the native API *Key*. The key difference is that Objectify's implementation is generic, which can be advantageous in some scenarios.
Instead of storing a key to another value, Objectify also provides users with a *Ref* class. *Refs* are stored as native *Key* in the datastore, but when loaded it is possible to use them as actual entity objects. However, these *Refs* can lead to a *DeadRef* if the entity could not be loaded properly. For example, when loading from the cache (Objectify, 2024). This is a reference to an object, that no longer works as it should because it is disconnected from the live system ('DeadRef (Objectify App Engine 5.0 API)', 2024).

### Migration Work

Although the storage methods between both APIs are largely similar, there are instances where additional effort is required to ensure that no data is lost during migration.
Whenever the data schema needs to be altered to improve the overall structure, a additional method is needed to load the old properties stored in the Datastore and load them into their new structure. An example method for this is illustrated in Listing 5.9.

**Listing 5.9:** Example @AlsoLoad Method

```
@Entity
public class User {
    private List<String> paths;

    private void loadOldProperty(@AlsoLoad("path") String oldpath
       ) {
        if (path == null) return;
        this.paths.add(path);
    }
}
```

In this example, we had just one path stored before the migration, and now we want to store multiple paths. To avoid losing any data, this method is automatically executed by Objectify after the *User* entity is loaded from the Datastore. We then store the old path value and can now also store other paths within the list.
In the process of migrating old properties to new structures, a challenge arises whereby only the properties of entities that have been loaded will be migrated. For example, if a *User* has not logged in for an extended period and, therefore, is not loaded, its data will not be transferred to the new format. To accelerate the migration process, migration tasks have been implemented to load all stored entities that require migration and then save them again. This ensures that the method for loading old properties is invoked for each relevant entity. Once this migration task is completed, the migration method can be safely removed from the codebase.
A key distinction between JDO and Objectify lies in their ability to query polymorphic entities. Although Objectify supports querying across polymorphic entities natively, JDO does not offer this feature directly. In JDO, filtering queries based on the subclass type requires using the native Google Datastore API to query the discriminator column. Objectify, on the other hand, allows for querying polymorphic entities by simply including the entity's class type when loading. However, Objectify searches for the subclass type in its own `^d` column. Before the migration, this information was stored in a column named *DISCRIMINATOR*, which led to a mismatch. To resolve this, a new *String* property was added to these classes, annotated with JDO's *@Persistent(column = "^d")*, aligning the column names. Combined with a method that loads the old discriminator value and stores it in the new property, along with a migration task, this solution allowed for a seamless transition without data loss or service interruptions.

### 5.1.3 Migrate child entities to embedded entities

To optimize the datastore, certain entities have been migrated from being linked by foreign key relationships to being embedded within another entity. Con-

sequently, an embedded entity is no longer part of its datastore table, but is included in the entity, where it is embedded in. However, due to Google's 1 megabyte limit on entity size, not all related entities can be embedded.
Previously, JDO automatically stored only the key values of objects, while the backend worked on the entire instance, not just the keys. With the migration to Objectify, this behavior had to be reconstructed by creating a new field to store the instances. A method with a parameter annotated with *@AlsoLoad* and a *Ref* type on the embedded entity was implemented to load old values from existing entities and set the new embedded values using the old values. If the old entities had a parent key property, this property would also need to be given to the embedded entity, which JDO automatically created before.
There were additional challenges during the migration, particularly with queries involving newly embedded entities. Therefore, significant changes to the queries were required. Until all entities are fully migrated, queries need to be written to take into account both old non-embedded entities and new embedded entities. These queries need to filter out any duplicates before returning a consolidated list.
Migration scripts are required to transfer all the old data into the new embedded entity groups. With the data migration now complete, the *@AlsoLoad* method is ready for removal, allowing for the simplification of complex queries once again.

## 5.2 Integration of Design Patterns to Enhance the Codebase

In this section, the visitor design pattern will be explored in greater depth. This includes a detailed explanation of the various visitor implementations, highlighting how each one functions and interacts within the pattern.

### 5.2.1 *BaseProject* classes

For the project classes a visitor design pattern was implemented to manage the load and delete operations within the *BaseProjectDAO* class. The *BaseProjectDAO* class is a DAO class responsible for handling data store interactions for all project classes. Figure 5.2 illustrates the hierarchical structure of the *BaseProject* class, which serves as the parent class to several subclasses, including *Project*, *SandboxProject*, and *ProjectRevision*. The *ProjectRevision* class is further extended by two additional subclasses, *ExerciseProject* and *ValidationProject*.
By applying the visitor design pattern, the load and delete logic were encapsulated in separate visitor classes. This approach kept the *BaseProjectDAO* class clean and focused on its primary responsibilities, while the visitor classes managed the specific operations. This separation of concerns not only improved the

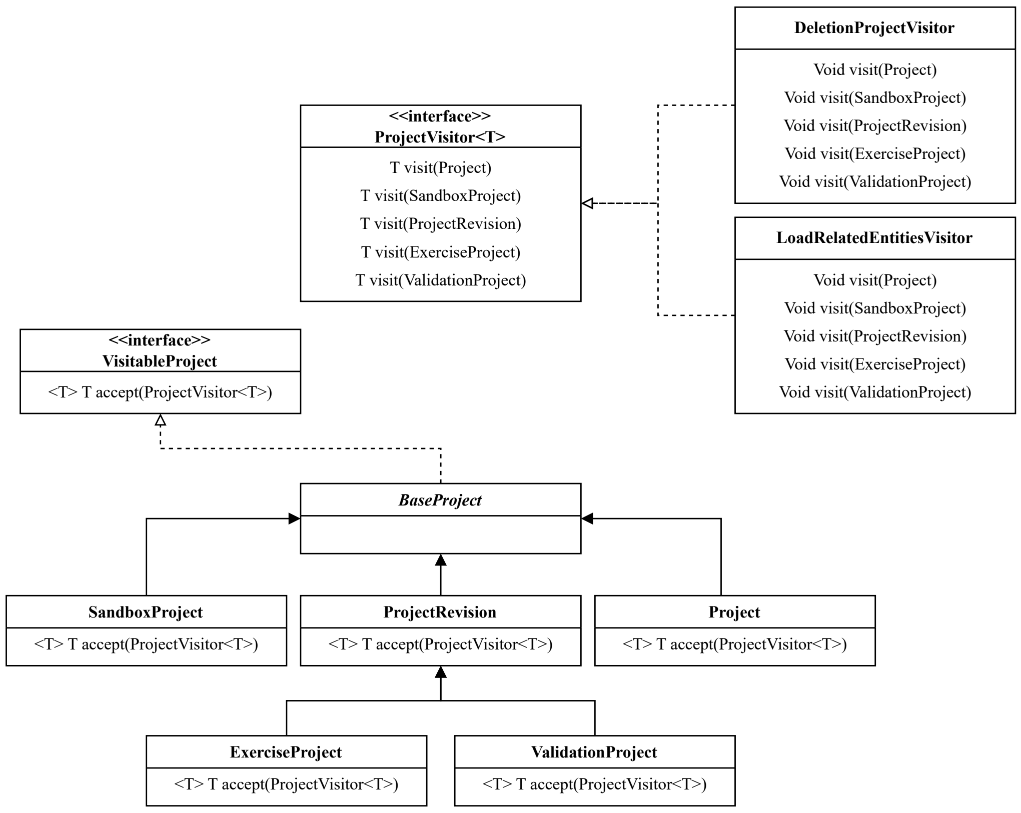
DeletionProjectVisitor
Void visit(Project)
Void visit(SandboxProject)
Void visit(ProjectRevision)
Void visit(ExerciseProject)
Void visit(ValidationProject)
<<interface>>
ProjectVisitor<T>
T visit(Project)
T visit(SandboxProject)
T visit(ProjectRevision)
T visit(ExerciseProject)
T visit(ValidationProject)
LoadRelatedEntitiesVisitor
Void visit(Project)
Void visit(SandboxProject)
Void visit(ProjectRevision)
Void visit(ExerciseProject)
Void visit(ValidationProject)
<<interface>>
VisitableProject
<T> T accept(ProjectVisitor<T>)
BaseProject
SandboxProject
<T> T accept(ProjectVisitor<T>)
ProjectRevision
<T> T accept(ProjectVisitor<T>)
Project
<T> T accept(ProjectVisitor<T>)
ExerciseProject
<T> T accept(ProjectVisitor<T>)
ValidationProject
<T> T accept(ProjectVisitor<T>)

**Figure 5.2:** UML Diagram of the Project Visitor Design Pattern

maintainability of the code, but also made it easier to extend and modify the operations without changing the existing class hierarchy.

***ProjectRevision:*** Figure 5.3 outlines the architecture of the *ProjectRevision* class, along with its parent class and child classes. This structure introduces additional properties compared to *BaseProject*, including a list of *Snapshot* keys and a corresponding list of *Snapshot* entities. Since *Snapshot* entities can exceed the 1 megabyte entity size limit imposed by Google for stored entities, only their keys and GCS blob information are stored in the Datastore. Since the loading of a *ProjectRevision* entity will not automatically load the *Snapshot* entities, but only their keys, the *Snapshot* entities have to be manually loaded and set in the *ProjectRevision#snapshotEntities* property.

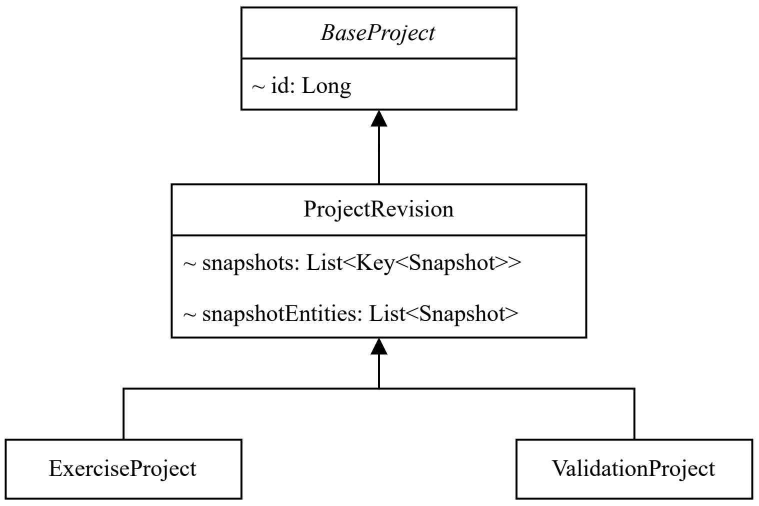

**Figure 5.3:** UML Diagram of the *ProjectRevision* architecture

***LoadRelatedEntitiesVisitor:*** When loading a project of type *ProjectRevision*, it is essential to load all related *Snapshot* entities to ensure they are available for backend operations. The *LoadRelatedEntitiesVisitor* class is used to retrieve the *Snapshot* entities by their keys and populating the *snapshotEntities* property with the loaded data.

***DeleteProjectVisitor:*** Conversely, as the Datastore does not facilitate cascading deletes, the *DeleteProjectVisitor* class was implemented to eradicate all entities that are associated with a Project entity. This visitor is then invoked by the *BaseProjectDAO* class when a project deletion operation is initiated, thus preventing the creation of orphaned related entities within the Datastore.

### 5.2.2 *BaseDocument* classes

QDAcity's document classes already implement a visitor design pattern illustrated in figure 5.4, which has been extended by a new visitor class called *DocumentDeletionVisitor*. This class is designed to manage the different actions required to delete different types of documents. For example, when deleting an *TranscriptionDocument* entity, the corresponding *TranscriptionJob* entities must also be deleted.
Furthermore, the *DocumentDeletionVisitor* class handles the deletion of content stored in the GCS, which varies between document types, because the different document types have differing amounts of blobs attached to them. By encapsulating the deletion logic within this visitor class, the codebase remains clean and modular. This approach ensures that the specific deletion requirements of each document type are adequately addressed, without adding complexity to the main document classes. Additionally, the *DocumentDeletionVisitor* class can be extended or modified to accommodate new deletion requirements, ensuring that the system remains flexible and adaptable.

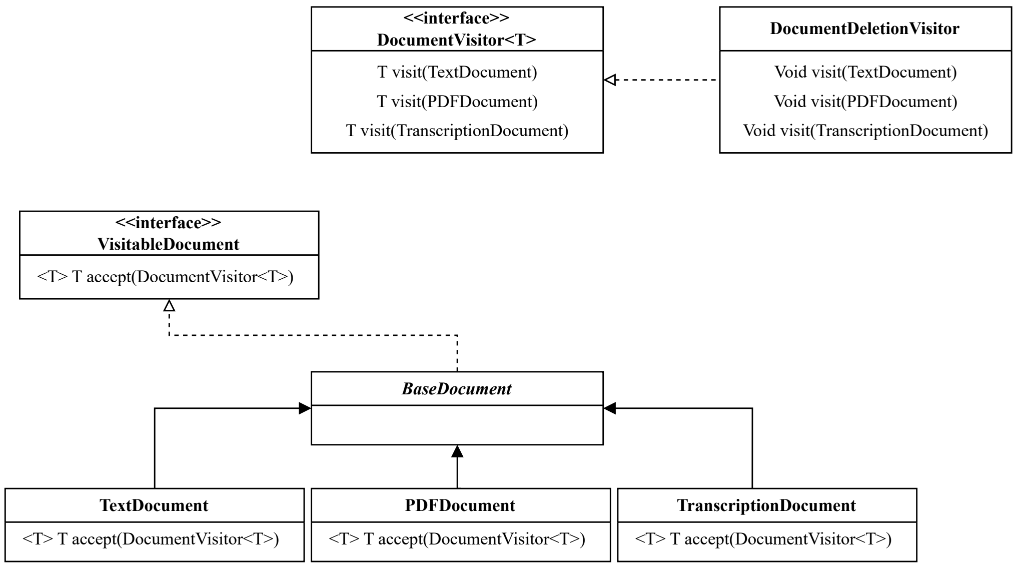

**Figure 5.4:** UML Diagram of the Document Visitor Design Pattern

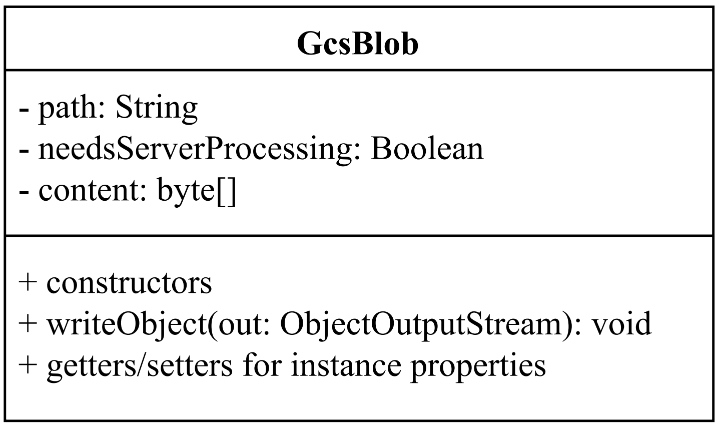
GcsBlob
- path: String
- needsServerProcessing: Boolean
- content: byte[]
+ constructors
+ writeObject(out: ObjectOutputStream): void
+ getters/setters for instance properties

**Figure 5.5:** UML Diagram of the *GcsBlob* class

## 5.3 GCS Blob & GcsClient design

### 5.3.1 *GcsBlob* class

Figure 5.5 illustrates the structure of the *GcsBlob* class. The *GcsBlob* class has three private instance properties.

- *path:* This property is a string representing the path to the blob in GCS. It is used to locate and access the blob within the storage system.
- *needsServerProcessing:* This boolean property indicates whether the content needs to be sent to the server for processing before being stored in GCS. If set to *true*, the content will be processed by the server. Otherwise, it is stored directly in GCS.
- *content:* This property is a byte array containing the actual content of the blob. It contains the data that needs to be stored in GCS. To prevent this potentially large property from being stored in the datastore, it is annotated with *@Ignore*.

In order to enable the caching of objects that have a *GcsBlob* as a property in the *memcache*, it is necessary to consider the *memcache*'s 1 megabyte record size limit. This issue has been addressed by implementing a custom *writeObject* method, which ensures that the *GcsBlob*'s content data, which can be larger than 1 megabyte, is not serialized.

The *writeObject(ObjectOutputStream)* method temporarily saves the *content* property in a local variable, sets *content* to *null* to exclude it from the serialization,

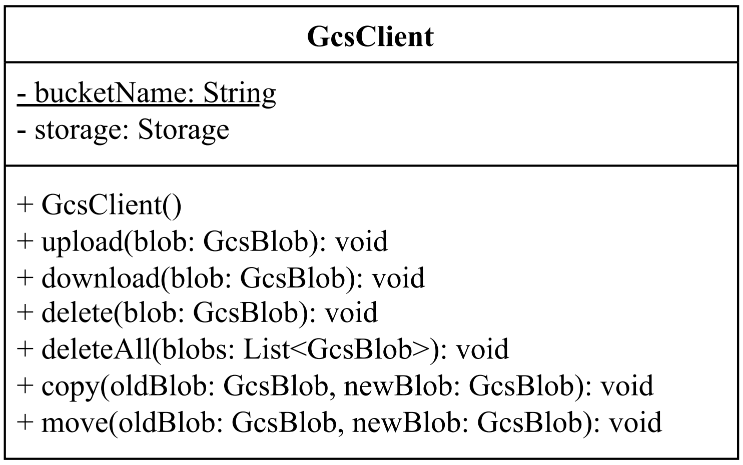
GcsClient
- bucketName: String
- storage: Storage
+ GcsClient()
+ upload(blob: GcsBlob): void
+ download(blob: GcsBlob): void
+ delete(blob: GcsBlob): void
+ deleteAll(blobs: List<GcsBlob>): void
+ copy(oldBlob: GcsBlob, newBlob: GcsBlob): void
+ move(oldBlob: GcsBlob, newBlob: GcsBlob): void

**Figure 5.6:** *GcsClient* UML Diagram

calls *out.defaultWriteObject()* to serialize the object without the *content* property, and then restores the *content* property from the local variable after serialization.

### 5.3.2 *GcsClient* class

In order to interact with the GCS, the *GcsClient* class is implemented. Figure 5.6 illustrates the structure of this class, which has two properties:

- *bucketName:* Each Google Cloud Project can have multiple GCS buckets, each with its own files and configurations. In this context, the static final *String* stores the name of the GCS bucket, which is derived from the application ID. It is employed to specify the target bucket for all GCS operations.
- *storage:* This particular instance of the *Storage* class, which is provided by the GCS library, is employed for the purpose of interacting with the GCS. It is initialized with the requisite service account credentials and the relevant project ID within the constructor.

The introduction of the new *GcsBlob* class has necessitated modifications to certain methods inside the *GcsClient* class. These methods now employ the GCS path stored within the *GcsBlob*.

- *upload(GcsBlob):* This method uploads the content of a *GcsBlob* to the GCS.
- *download(GcsBlob):* This method downloads the content of a *GcsBlob* from the GCS and saves it in the *GcsBlob*.

- *delete(GcsBlob):* This method removes content from GCS at the *GcsBlob's* path without altering the *GcsBlob* instance, which may still contain the blob data even after the deletion. This ensures that the instance remains unchanged, and its data is not invalidated by the deletion process.
- *deleteAll(List<GcsBlob>):* This method performs the same function as the *delete(GcsBlob)* method, but for a list of *GcsBlobs*.
- *copy(GcsBlob, GcsBlob):* This method enables the transfer of a blob to another location within the same GCS bucket.
- *move(GcsBlob, GcsBlob):* The objective of this method is to invoke the *copy(GcsBlob, GcsBlob)* method and subsequently remove the blob situated at the former path.

Prior to the modification, the aforementioned methods received an interface parameter, which held the path where the data is stored inside the GCS. This interface was implemented by all classes that stored a GCS Uniform Resource Locator (URI).

### 5.3.3 Strategy Design Pattern

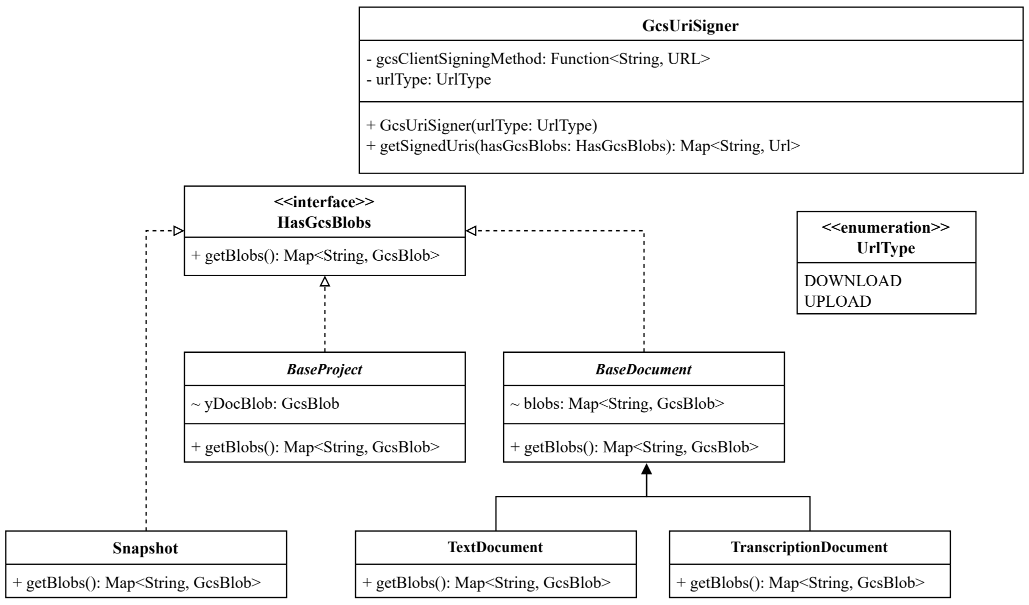
GcsUriSigner
- gcsClientSigningMethod: Function<String, URL>
- urlType: UrlType
+ GcsUriSigner(urlType: UrlType)
+ getSignedUris(hasGcsBlobs: HasGcsBlobs): Map<String, Url>
<<interface>>
HasGcsBlobs
+ getBlobs(): Map<String, GcsBlob>
<<enumeration>>
UrlType
DOWNLOAD
UPLOAD
BaseProject
~ yDocBlob: GcsBlob
+ getBlobs(): Map<String, GcsBlob>
BaseDocument
~ blobs: Map<String, GcsBlob>
+ getBlobs(): Map<String, GcsBlob>
Snapshot
+ getBlobs(): Map<String, GcsBlob>
TextDocument
+ getBlobs(): Map<String, GcsBlob>
TranscriptionDocument
+ getBlobs(): Map<String, GcsBlob>

**Figure 5.7:** Strategy Design Pattern

Figure 5.7 shows a detailed view of the structure and relationships between several key components, including the *HasGcsBlobs* interface and the *UrlType* enumeration, within a system that manages blobs in the GCS. Any class that stores at least one *GcsBlob* instance implements the *HasGcsBlobs* interface, allowing for consistent and efficient handling of blobs across different classes. The *GcsUriSigner* class is responsible for securing access in order for the resources to be accessible by e.g. the QDAcity frontend. This class contains two main properties:

- *gcsClientSigningMethod:* Is populated with a function that accepts a string and returns a URL. The specific method used is determined by the *urlType* property, as different procedures are required to generate signed URLs depending on whether the operation is a download or an upload to the GCS. This function handles this distinction accordingly.
- *urlType:* An enumeration that specifies whether the signed URL is for a download or upload operation, guiding the behavior of the *gcsClientSigningMethod*.

This class currently provides a single method for handling *HasGcsBlobs* instances. The method iterates over the provided map, sign the URI of each *GcsBlob*, and subsequently add them to the returned map. In instances where the *needsServerProcessing* property of a *GcsBlob* is set to *true*, the method simply skips this iteration. Prior to this modification, the *needsServerProcessing* property had to be stored directly in the entities themselves, which made the architectural design less adaptable for future requirements, as some blobs within the same entity class may not require additional processing.
The *HasGcsBlobs* interface serves a crucial role in the architectural framework. It defines a single method, *getBlobs()*, which returns a Map<String, GcsBlob>. This approach improves code reusability and ensures consistency across the system.
The *BaseDocument* class now includes a *Map<String, GcsBlob>* property called *blobs*, where each document type has a content blob, and additional *GcsBlob* instances specific to the document can be added to the map. This structure allows different document types to store both the *contentBlob* and any other relevant *GcsBlobs* in a unified way, making the retrieval of all *GcsBlob* instances straightforward and easily extendable for future requirements.
All project classes, on the other hand, currently include a *yDocBlob* property. When this has to be extended in the future, this property could also be migrated to a collection or map.
Previously, the backend handled URL signing for each entity class individually, using a separate class that implemented the visitor design pattern to manage document blobs. With the introduction of the *GcsUriSigner* class, these signing tasks have been streamlined. The *GcsUriSigner* consolidates all signing operations into a single method, simplifying the process and ensuring consistency across different use cases. This change enhances maintainability by centralizing URL signing in one location.

# 6 Evaluation

In this chapter, both the functional and non-functional requirements outlined in chapter 3 are thoroughly evaluated to determine how well they have been met.

## 6.1 Functional requirements

| | **Reg 1a** | **Reg 1b** | **Reg 1c** | **Reg 1d** | **Reg 2a** | **Reg 2b** | **Reg 3a** | **Reg 3b** |
|---|---|---|---|---|---|---|---|---|
| fulfilled | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| partly fulfilled | | | | | | | | |
| not fulfilled | | | | | | | | |

**Table 6.1:** Functional Requirement Evaluation Results

As demonstrated in Table 6.1, all functional requirements are fulfilled. The table presents a comprehensive overview of the fulfillment status, indicating that each requirement was successfully met. The subsequent section provides a detailed examination of each functional requirement and elucidates the methods through which they have been addressed and implemented.

### 6.1.1 API Reliability and Compliance

**Req 1.a: The new API shall be capable of utilizing the necessary features of the Google Datastore.**

Objectify takes full advantage of the rich feature set of the Google Datastore. The API supports all native Datastore functionality. This ensures that the system can utilize the full capabilities of the Google Datastore to provide robust and efficient data management[1].

**Req 1.b: The new API shall ensure the integrity and consistency of the data.**

[1]https://github.com/objectify/objectify/wiki

Objectify ensures the integrity and consistency of the data by supporting transactions that enforce atomicity, consistency, isolation, and durability (ACID) properties. Each Objectify transaction opens its own *ObjectifyService*, which separates the caches from each other. Furthermore, transactions must include an ancestor in order to be queryable[2].

**Req 1.c: The new API shall be actively maintained, with at least one official release in the year 2024 as a criterion for active maintenance.**

The latest version of Objectify, released in July 2024, demonstrates that the library is actively maintained, with ongoing support and updates from its developers. Therefore, it can be concluded that Objectify satisfies the criterion for active maintenance[3].

**Req 1.d: The new API shall be permissively licensed, allowing for modification, distribution, and commercial use with minimal restrictions.**

Objectify meets this requirement, as it is licensed under the MIT License. The MIT License is a permissive open-source license that allows users to freely use, modify, distribute, and even sell copies of the software with very few restrictions. The only obligation is to include the original copyright notice and license in any copies or significant portions of the software. Therefore, Objectify satisfies the requirement of being permissively licensed[4].

### Req 2: QDAcity shall support the transition to the new API, ensuring the preservation of data and compatibility with current data structures while integrating new features during the migration from JDO.

### 6.1.2 API migration

**Req 2.a: QDAcity shall support migration from JDO to a new framework without data loss, ensuring compatibility with existing data structures while introducing new enhancements.**

The migration from JDO to Objectify was carefully managed to guarantee the integrity of the data. Initially, the compatibility with existing data structures was preserved by using methods to load old properties and save them in the new properties. Furthermore, migration scripts were employed to update entities, ensuring a seamless transition while introducing new enhancements.

[2]https://github.com/objectify/objectify/wiki/Concepts#transaction-limitations
[3]https://github.com/objectify/objectify/releases/tag/6.1.2
[4]https://github.com/objectify/objectify?tab=MIT-1-ov-file#licence.txt

**Req 2.b: QDAcity shall enable incremental migration from JDO to the new API, allowing gradual updates without necessitating a complete transition at once.**

Following the refactoring of all JDO code into DAO classes, the migration of entity groups was conducted in a sequential manner. This approach permitted the coexistence of JDO and Objectify within the codebase, facilitating an effective transition. The incremental migration strategy minimized disruption and enabled continuous functionality during the upgrade process, thereby demonstrating that the requirement was effectively met.

### 6.1.3 GCS blobs

**Req 3.a: The QDAcity backend component for managing GCS blobs shall be able to create, read, update, and delete data.**

Methods within the *GcsClient* class have been designed with the specific intention of facilitating interaction with the GCS. It provides a comprehensive set of methods that are capable of ensuring effective management of data in the form of blobs.
The creation of data in the GCS is facilitated by the *upload(GcsBlob blob)* method. This method uploads the content of a *GcsBlob* to GCS. The retrieval of data from the GCS is facilitated by the *download(GcsBlob blob)* method, which loads the content from the GCS and stores the data in a *GcsBlob* instance. The update functionality is intrinsic to the *upload(GcsBlob blob)* method. When a *GcsBlob* object with an existing URI is passed to this method, the content of the blob in the GCS is overwritten with the new content provided in the *GcsBlob* object. The deletion of data in the GCS is managed by the *delete(GcsBlob blob)* method.

**Req 3.b: The QDAcity backend component for managing GCS blobs shall be able to manage multiple blobs connected to a single Datastore entity.**

The entity classes have been enhanced to support the creation and administration of collections or maps of *GcsBlobs*. This allows multiple blobs to be associated with a single Datastore entity. By storing these collections or maps in the Google Datastore, the system can efficiently manage multiple blobs for each entity.

## 6.2 Non-functional requirements

| | **Reg 1a** | **Reg 1b** | **Reg 2a** | **Reg 2b** | **Reg 2c** | **Reg 2d** | **Reg 2e** | **Reg 3a** | **Reg 3b** | **Reg 3c** |
|---|---|---|---|---|---|---|---|---|---|---|
| fulfilled | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| partly fulfilled | | | | | | | | | | |
| not fulfilled | | | | | | | | | | |

**Table 6.2:** Non-Functional Requirement Evaluation Results

As evidenced in Table 6.2, all non-functional requirements are fulfilled. The subsequent section provides a detailed examination of each non-functional requirement, outlining the methods and strategies employed to address and implement them. This includes aspects such as performance, reliability, and usability, ensuring that the system not only meets its functional goals but also excels in delivering a high-quality user experience and robust performance under various conditions.

### 6.2.1 Datastore Access Encapsulation

**Req 1.a: Code that accesses the datastore shall be encapsulated within a dedicated layer of the software architecture, to ensure maintainability and testability.**

Initially, code accessing the Google Datastore was dispersed throughout the backend codebase, which resulted in maintenance challenges and inefficiencies. By centralizing all datastore access within DAO classes, the system ensures that no datastore calls will be made outside of these classes. The refactoring effort involved reorganizing approximately 28,000 lines of code into a more structured endpoint-controller-DAO architecture. As a result, the codebase is now more organized, with clear separation of concerns, and the amount of duplicate code has been significantly reduced. This approach has improved maintainability by making it easier to locate and update datastore access logic, enhanced performance by optimizing data access patterns, and increased readability by providing a consistent structure. The new architecture not only meets the requirements but also provides a robust foundation for future development and maintenance.

**Req 1.b: Each datastore table shall be managed through a dedicated class, allowing for shared access among related entities while maintaining a clear separation of concerns.**

DAO classes have been implemented to encapsulate all code responsible for establishing connections to the datastore. A total of 43 DAO classes were created to ensure comprehensive coverage of all data access needs. These classes effectively manage interactions with the datastore tables, allowing related entities that share a table to access data through the appropriate class. This design promotes a clear separation of concerns and enhances maintainability and testability within the codebase.

### 6.2.2 System Maintainability and Future-Proofing

**Req 2.a: The refactoring shall enhance the codebase's maintainability by establishing a clear structure and reducing duplication, thereby facilitating efficient feature additions.**

The refactoring process has greatly improved the maintainability of the codebase by introducing a clear and organized structure. By centralizing data store access within DAO classes and establishing a well-defined architecture that separates endpoints, controllers, and DAO classes, the codebase has become more modular and easier to manage. A significant portion of duplicate code has been refactored into the DAO classes, further simplifying the codebase and reducing redundancy. Additionally, the code was cleaned up, such as removing the iteration over child entities, a step that was necessary with JDO, but is no longer required in the new structure.
This new structure facilitates extensions by providing a consistent framework for adding new functionality. Developers can now follow established patterns when introducing new endpoints, controllers, and DAO classes, ensuring that existing code is not disrupted. This modular approach enables integration of new functionality, reducing the risk of bugs or inconsistencies.
In addition, the improved maintainability of the codebase allows future changes to be implemented more efficiently. The clear separation of concerns between different architectural layers allows developers to focus on specific areas of the code without having to understand the entire system.

**Req 2.b: QDAcity shall be able to update dependencies and integrate new dependencies.**

As a consequence of the migration from JDO to Objectify, five dependencies were removed: *datanucleus-appengine*[5], *datanucleus-core*[6], *datanucleus-api-jdo*[7], *jdo-api*[8], and *javax-persistence*[9]. This migration not only streamlined the codebase but also established a more flexible architecture that allows for the straightforward upgrading of existing dependencies. Furthermore, the new structure

[5]https://mvnrepository.com/artifact/com.google.appengine.orm/datanucleus-appengine/2.1.2

[6]https://mvnrepository.com/artifact/org.datanucleus/datanucleus-core/3.1.3

[7]https://mvnrepository.com/artifact/org.datanucleus/datanucleus-api-jdo/3.1.3

[8]https://mvnrepository.com/artifact/javax.jdo/jdo-api/3.0.1

[9]https://mvnrepository.com/artifact/javax.persistence/javax.persistence-api/2.2

supports the integration of new dependencies, which was not feasible with the previous setup. As a result, developers can now efficiently manage and incorporate third-party libraries and tools, enhancing the overall functionality and adaptability of the system.

**Req 2.c: QDAcity shall be compatible with all next-generation GAE APIs.**

Previously, the system was dependent on an outdated version of the DataNucleus-AppEngine dependency, which was no longer supported by Google. The lack of support prevented the application from upgrading the DataNucleus v3 dependency to DataNucleus v6. and constrained its compatibility with any prospective Java runtimes. However, through the migration to Objectify, the obsolete dependencies were removed, thus enabling the application to be compatible with the Second-generation Java runtimes[10]. This transition ensures that the system can leverage the latest features and improvements

| | **Low** | **Mid** | **High** | **Critical** |
|---|---|---|---|---|
| CVE-2019-17571 | | | | x |
| CVE-2020-1945 | | x | | |
| CVE-2021-36373 | | x | | |
| CVE-2021-4104 | | | x | |
| CVE-2022-23302 | | | x | |
| CVE-2022-23305 | | | | x |
| CVE-2022-23307 | | | x | |

**Table 6.3:** Categorization of Common Vulnerabilities and Exposures (CVE) by Severity

**Req 2.d: QDAcity shall address and mitigate security vulnerabilities present in the previous implementation.**

As a consequence of the migration, the system has become more modular and less reliant on outdated dependencies. The modularity of the system allows for the updating of individual components without compromising the overall stability of the system. Moreover, the active maintenance and community support provided by Objectify facilitate the integration of new updates and security patches. This proactive approach to dependency management mitigates the risk of vulnerabilities and ensures that the system remains current with the latest advancements. This reduction in complexity not only enhances the system's performance, but also facilitates future updates and enhancements. Furthermore, this migration resulted in the removal of seven security vulnerabilities. Table 6.3 categorizes

[10]https://web.archive.org/web/20240901144234/https://cloud.google.com/appengine/migration-center/standard/migrate-to-second-gen/java-differences

the removed CVEs by their severity. Among these, three were rated as high and two as critical. These vulnerabilities were addressed by removing outdated and vulnerable dependencies, leading to a significant improvement in the security posture of the system. The following CVE were mitigated by removing these deprecated libraries:

- **Memory Issues:**
  - **CVE-2021-36373[11]:** A memory-related error that posed a risk in the previous implementation was resolved. In specific instances, the application may encounter a memory error when processing a specially designed *TAR* file.
  - **CVE-2020-1945[12]:** The vulnerability in *Apache Ant* permitted the leakage of sensitive information and the potential manipulation of code through the unsafe handling of temporary files. The resolution of this issue was achieved by the removal of the obsolete JDO dependency.
- **Deserialization Vulnerabilities:**
  - **CVE-2022-23307[13], CVE-2022-23302[14], CVE-2021-4104[15], CVE-2019-17571[16]:** These vulnerabilities, which originated from the insecure deserialization of data, were addressed by eliminating the usage of the outdated *DataNucleus-AppEngine* API that permitted these attack vectors.
- **Structured Query Language (SQL) Injection:**
  - **CVE-2022-23305[17]:** The vulnerability in *Log4j 1.2.x* allowed attackers to manipulate SQL queries through crafted inputs, due to insecure handling of SQL statements in the *JDBCAppender*. By removing dependencies tied to this version of *Log4j*, the risk of SQL injection was eliminated.

The migration to Objectify not only brought improvements in performance and maintainability but also fortified the system against these critical security vulnerabilities. By removing obsolete dependencies that were known to carry security flaws, the system is now more secure and less prone to potential threats, providing a higher degree of safety for both data and users.

[11]https://nvd.nist.gov/vuln/detail/CVE-2020-1945
[12]https://nvd.nist.gov/vuln/detail/CVE-2021-36373
[13]https://nvd.nist.gov/vuln/detail/CVE-2022-23307
[14]https://nvd.nist.gov/vuln/detail/CVE-2022-23302
[15]https://nvd.nist.gov/vuln/detail/CVE-2021-4104
[16]https://nvd.nist.gov/vuln/detail/CVE-2019-17571
[17]https://nvd.nist.gov/vuln/detail/CVE-2022-23305

| **Operation** | **Old Version *put* requests** | **Current Version *put* requests** |
|---|---|---|
| *projects* | 7 | 4 |
| *createRevision* | 4 | 3 |
| *settleAction* | 10 | 6 |

**Table 6.4:** Example comparison of the number of Datastore *put* requests in the old and current versions

**Req 2.e: QDAcity shall decrease the number of Datastore write operations in its backend.**

One of the main metrics examined throughout the course of this thesis was the number of Google Datastore *put* requests generated by the application. At the beginning of the project, the number of *put* requests was considerably higher, as evidenced by Table 6.4. By the end of the thesis, there was a notable reduction in the number of requests, with decreases of 42.86%, 25%, and 40% across different API endpoints respectively. Notably, this decrease was observed primarily in more complex requests, while simpler requests remained unchanged. This reduction can be directly attributed to the migration from JDO to Objectify. JDO inherently associates objects with *PersistenceManager*[18], which can result in implicit write operations at unexpected times, without explicit commands. This behavior frequently results in inefficiencies, as Datastore operations occur in an unpredictable manner. In contrast, Objectify offers a more controlled and explicit approach to data persistence. With Objectify, write operations are only made when explicitly called, leading to more efficient data management and significantly fewer unnecessary datastore operations.

### 6.2.3 Documentation

**Req 3.a: The code shall be documented using *JavaDocs* to facilitate future development and maintenance.**

Extensive documentation has been incorporated throughout the codebase to support future development and maintenance. Each DAO class is now documented, offering clear explanations of their purpose, methods, and usage. This thorough documentation ensures that developers can easily grasp the functionality and responsibilities of each class, facilitating smoother onboarding and collaboration. The comprehensive documentation includes:

- **Class Descriptions:** All added classes have detailed explanations of their purpose and functionality.
- **Method JavaDocs:** All methods inside DAO classes have clear descrip-

[18]https://db.apache.org/jdo/pm.html

tions, including parameters, return values, and any exceptions that may be thrown.

This documentation not only fulfills the requirement but also significantly enhances the overall quality of the codebase. It ensures that future development and maintenance efforts are well-supported, thereby promoting a more efficient and collaborative development process.

**Req 3.b: The QDAcity wiki shall be extended to include the new API.**
The QDAcity Wiki has been updated to include comprehensive documentation on the new Objectify API. The documentation provides an overview of Objectify, highlighting its advantages in simplifying and streamlining common datastore operations such as querying, saving, and deleting entities. It also includes a quick start guide for defining entities and demonstrates how to create an entity class and annotate it appropriately. By including this detailed information, the QDAcity wiki not only meets the requirement but also serves as a valuable resource for developers, ensuring they have the necessary guidance to effectively use the new API.

**Req 3.c: This thesis shall serve as comprehensive documentation for future developers working with QDAcity.**
This thesis includes detailed explanations of key components, such as the migration from JDO to Objectify, the refactoring efforts undertaken to improve maintainability, and the overall functionality of the system. It offers a comprehensive examination of the architectural design, design decisions, and implementation strategies of the QDAcity project. The thesis presents this information in a structured and accessible manner, thereby serving as an essential resource for future developers. It not only aids in understanding the existing codebase but also supports onboarding and promotes best practices within the development team. Furthermore, the documentation includes examples, diagrams, and references to relevant resources, ensuring that developers have the necessary tools and knowledge to effectively contribute to the QDAcity project.

# 7 Outlook

While QDAcity's backend codebase has undergone significant modernization, there still remain areas that require further improvement. This section outlines key milestones for future development efforts on QDAcity, highlighting areas that have not yet been fully handled.

## 7.1 Upgrading Objectify to v6

An important step into the future is the upgrade from the current Objectify v5 to the new Objectify v6. Migration to a newer Objectify version was not yet possible, because it requires the use of the new datastore emulator, which is not usable for Java8 applications. This new version introduces several major changes, the most notable being the transition from the proprietary *ApiProxy* interface to the more versatile *Google Cloud SDK*. Google is phasing out this proprietary interface in favor of standards-based interfaces for cloud services. This makes Objectify v6 more future proof. The *Google Cloud SDK* represents the future of Google, offering developers greater flexibility, compatibility, and long-term support[1].

## 7.2 Upgrading to Java 11/17

With the removal of DataNucleus and its plugin, the application can now upgrade its Java version. Previously, the DataNucleus-Plugin prevented this upgrade due to compatibility issues with newer Java versions. These dependencies were tightly coupled to older Java versions, making it impossible to upgrade to a newer version.
With Java 8's general availability date back in March 2014 and premier support having ended in March 2022, the extended support, which lasts until December 2030, is now the key phase for maintaining security and stability. As a result, upgrading to more recent long-term support versions such as Java 11 or 17 is

[1]https://github.com/objectify/objectify/wiki/FrequentlyAskedQuestions

becoming increasingly important for businesses looking to stay up to date and secure in the long term[2]. There are numerous advantages to making this transition. One of the primary benefits is enhanced performance. Java 11 and 17 introduce optimized features such as ahead-of-time compilation, which improves the performance of Java programs while reducing the application's startup time. Java 17 also introduces new, improved garbage collection mechanisms and better just-in-time compilation. Security is another critical improvement with Java 17. The latest version includes advanced security features that better protect user data and ensure the integrity of Java applications. This is crucial for maintaining a secure and reliable application environment. Additionally, there are several new language-level improvements. For example, pattern matching for switch statements makes the code cleaner and more readable. These enhancements simplify the development process and improve code maintainability. Furthermore, upgrading to Java 11 or 17 also facilitates easier integration with modern frameworks. The latest API's and libraries are designed to work seamlessly with contemporary development tools, providing better support and integration ('Top 7 Reasons To Migrate From Java 8 to Java 17', 2023).

## 7.3 Optimize Cache usage

Many of the DAO classes in the codebase already store and delete entities using the *memcache*. However, the cache load calls are scattered throughout the codebase. In the future, these cache load calls should be moved to the DAO classes. The load methods should first attempt to retrieve data from the cache, and if nothing is found, then proceed to load from the Datastore. Additionally, several methods in the *Cache* class require refactoring because they are marked as deprecated, yet continue to be used.

## 7.4 Use more Key properties

Currently, foreign key class members that reference other entities in the datastore are often stored as *Long* IDs or *String* names. For example, *BaseDocument#projectId* references the project to which a document belongs. However, projects are distributed across four different database tables, each representing a different type of project. This distribution means that IDs are only unique within their respective tables, leading to potential ambiguity when referencing these projects.
To address this vagueness, the Datastore *Key* of the project should be stored

[2]https://web.archive.org/web/20241004224253/https://www.oracle.com/java/technologies/java-se-support-roadmap.html

directly. This approach ensures that references are unambiguous, as each *Key* uniquely identifies an entity across all tables.

# 8 Conclusion

In conclusion, this thesis has examined the transition from using JDO with DataNucleus to Objectify for interacting with the Google Datastore. While JDO was once a suitable interface, the now obsolete DataNucleus plugin no longer meets current standards. The lack of updates and support for the DataNucleus plugin has made it increasingly difficult to maintain and extend the system, and in some cases it has become impossible to update or upgrade certain dependencies. In contrast, Objectify is a well-maintained API designed specifically for the Google Datastore. It provides a modern and efficient approach to data management, enabling significant improvements to the project. Migrating to Objectify has brought several benefits, including improved performance, smoother integration with the Google Cloud Datastore and improved type safety, all while being future-proof. The refactoring process of approximately 28,000 lines, which involved centralizing datastore access within DAO classes and implementing a structured architecture comprising endpoints, controllers, and DAO classes, has greatly improved the maintainability and extensibility of the codebase. As part of this effort, around 30 controller classes were created, further enhancing the organization and making the backend cleaner and more structured. In addition, the comprehensive documentation introduced throughout the codebase further supports ongoing development and maintenance. Overall, the move to Objectify and the associated architectural improvements have resulted in a more organized, efficient and maintainable system. These changes provide a solid foundation for future development, ensuring that the project can continue to evolve and effectively meet the needs of its users.

In terms of GCS blobs management, the introduction of the *GcsBlob* has constituted a pivotal enhancement in the system, ensuring efficient management of substantial data storage requirements within the GCS environment. This implementation has introduced practical tools and patterns for efficiently managing data, including the ability to create, read, update, and delete blobs, as well as to handle multiple blobs tied to a single datastore entity. At the heart of this functionality is the *GcsClient* class, which acts as the primary interface for interacting with GCS, offering simple methods for managing blobs. Complementing this, the *GcsBlob* class efficiently organizes the data needed for blob manage-

ment, while the *BlobStrategy* interface provides flexibility, enabling the system to easily accommodate different approaches for handling blobs. This design ensures greater adaptability and future scalability.
Tasks completed as part of this thesis included submitting around 190 merge requests and resolving around 200 issues in GitLab, which were able to completely fulfill all requirements. These activities were instrumental in moving the project forward. In addition to these primary tasks, several proactive measures were taken to improve areas outside of the original scope, further contributing to the overall improvement of the project. Time was spent updating legacy code, correcting inefficient implementations and resolving numerous IntelliJ warnings. A major initiative was the refactoring of several older controller classes that had previously relied on static methods, which cluttered the back-end code and reduced flexibility. Converting these to instance methods improved the structure, clarity and maintainability of the backend.
While these efforts have a positive impact on the project, several areas still require attention and development. As outlined in chapter 7, important milestones include upgrading Objectify to version 6, moving to Java 11 or 17, optimizing cache usage, and using more Datastore key properties to ensure clearer references throughout the datastore. Addressing these issues will be essential to maintain and improve the quality and longevity of QDAcity, ensuring it remains efficient and relevant for future developers.
In conclusion, the modernization of QDAcity's backend has not only led to a notable enhancement in its capacity to manage large-scale data storage requirements, but has also resulted in optimized overall system performance. The incorporation of sophisticated technologies has led to the creation of a more efficient, scalable, and maintainable infrastructure, which has reduced operational overhead. These enhancements provide a robust and flexible foundation for future innovations, enabling the seamless integration of new features and technologies. Consequently, QDAcity is now better positioned to adapt to evolving user needs, ensuring long-term reliability, security, and user satisfaction.

# References

Andreas Kaufmann, D. R. (2015). Improving traceability of requirements through qualitative data analysisanalysis (Open Source Research Group, Computer Science Department, Ed.). Retrieved September 3, 2024, from https://open.fau.de/server/api/core/bitstreams/57a12f24-d054-4269-bc26-f4a57f486603/content

Chris Rupp. (2014). Requirements templates — the blueprint of your requirement. Retrieved August 25, 2024, from https://www.sophist.de/fileadmin/user_upload/Bilder_zu_Seiten/Publikationen/RE6/Webinhalte_Buchteil_3/Requirements_Templates_-_The_Blue_Print_of_your_Requirements_Rupp.pdf

DataNucleus (Ed.). (2022). Jdo getting started guide (v5.2). Retrieved August 14, 2024, from https://www.datanucleus.org/products/accessplatform/jdo/getting_started.html

Datastore overview. (2024). Retrieved August 17, 2024, from https://cloud.google.com/datastore/docs/concepts/overview

David Jordan, C. R. (2003). Java data objects (O'Reilly & Associates, Ed.). Retrieved October 12, 2024, from https://books.google.de/books?hl=de&lr=&id=dTr7AAAAQBAJ&oi=fnd&pg=PR7&dq=jdo&ots=vTBdXaaJD9&sig=skRZsxuwoSdMl2KomX5DRwfPMxo&redir_esc=y#v=onepage&q=jdo&f=false

Deadref (objectify app engine 5.0 api). (2024). Retrieved August 18, 2024, from https://www.javadoc.io/doc/com.googlecode.objectify/objectify/5.0/com/googlecode/objectify/impl/ref/DeadRef.html

Erich Gamma, R. H. (1994). Design patterns - design patterns, elements of reusable object-oriented software.

GitHub. (2024). Objectify/objectify: The simplest convenient interface to the google cloud datastore. Retrieved August 15, 2024, from https://github.com/objectify/objectify/wiki

Google. (2024). Datastore-abfragen in jdo. Retrieved August 18, 2024, from https://cloud.google.com/appengine/docs/legacy/standard/java/datastore/jdo/queries?hl=de

Henri Basson, M. B. (2016). Qualitative evaluation of manufacturing software units interoperability using iso 25000 quality model (Springer International Publishing Switzerland, Ed.). Retrieved September 29, 2024, from http://www.wellesu.com/https://link.springer.com/chapter/10.1007/978-3-319-30957-6_16

João André Martins, Jisha Abubaker. (2024). Spring framework on google cloud. Retrieved August 20, 2024, from https://googlecloudplatform.github.io/spring-cloud-gcp/reference/html/index.html#spring-data-cloud-datastore

Leone, A., & Chen, D. (2007). Implementation of an object oriented data model in an information system for water catchment management: Java jdo and db4o object database. *Environmental Modelling & Software*, *22*(12), 1805–1810. https://doi.org/10.1016/j.envsoft.2007.05.016

Objectify. (2024). Entities · objectify/objectify wiki. Retrieved August 17, 2024, from https://github.com/objectify/objectify/wiki/Entities

Oscar Rosner. (2021). Profiling and optimizing performance in the cloud. Retrieved October 5, 2024, from https://oss.cs.fau.de/wp-content/uploads/2021/10/rosner_2021.pdf

Top 7 reasons to migrate from java 8 to java 17. (2023). *GeeksforGeeks*. Retrieved September 1, 2024, from https://www.geeksforgeeks.org/top-reasons-to-migrate-from-java-8-to-java-17/

What is object-relational mapping (orm) in dbms? (2024). *GeeksforGeeks*. Retrieved August 15, 2024, from https://www.geeksforgeeks.org/what-is-object-relational-mapping-orm-in-dbms/

Zeger Hendrikse. (2017). Intro to jdo queries. *Baeldung*. Retrieved August 20, 2024, from https://www.baeldung.com/jdo-queries